UNIT-1

I. Program Structure in Java:

1. Introduction
2. Writing Simple Java Programs
3. Elements or Tokens in Java Programs
4. Java Statements
5. Command Line Arguments
6. User Input to Programs
7. Escape Sequences
8. Comments
9. Programming Style.

II. Data Types Variables and Operators :

1. Introduction Data Types in Java
2. Declaration of Variables Data Types
3. Type Casting
4. Scope of Variable Identifier
5. Literal Constants, Symbolic Constants
6. Formatted Output with printf() Method
7. Static Variables and Methods
8. Attribute Final
9. Introduction to Operators
10. Precedence and Associativity of Operators
11. Assignment Operator (=)
12. Basic Arithmetic Operators
13. Increment (++) and Decrement (- -) Operators
14. Ternary Operator Relational Operators
15. Boolean Logical Operators
16. Bitwise Logical Operators.

III. Control Statements:

1. Introduction
2. if Expression
3. Nested if Expressions
4. if–else Expressions
5. Ternary Operator?:
6. Switch Statement
7. Iteration Statements
8. while Expression
9. do–while Loop
10. forLoop
11. Nested for Loop
12. For–Each for Loop
13. Break Statement
14. Continue Statement.

I. Program Structure in Java:

1. Introduction

Java is an Object Orient and high-level programming language, originally developed by Sun Microsystems and released in 1995. Java runs on a variety of platforms, such as Windows, Mac OS, Linux etc.

In Java, there are three types of programs as follows:
Stand- aline application programs : These programs are made and run on users computers
Applet programs : These programs are meant to run in a web page on the Internet.
Java Servlets : These programs run in computers that provide web services. They are also ofter called server isde programs or servlets.

2. Writing Simple Java Programs

Java is an Object oriented language. Every java program imports packages which are provides necessary classes and interfaces.
For example:

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

After import statement, every java program starts with the declaration of the class. A program may have one or more classes.

A class declaration starts with the keyword class, followed by the identifier or name of the class. Giving the name of a package at the top is optional.

Class declarion contains name of the class and body of the class. The body of the class may consist of several statements and is enclsed between the braces {}.

A sample of class declarions is as follows.

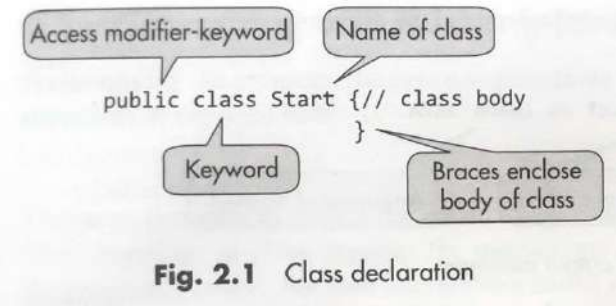


Fig. 2.1 Class declaration

Here:

public is access specifier. This class is accessible to any ouside code. Otherwise the class is accessibe to only same package classes.

class is a keyword of java language which is used to declare the class.

The class body starts with the left brace { and ends with the right colsing brace }.

// are comments which are neglected by the compiler.

A class body may comprise statements for declaration of variables. constants, expressions, and methods.

Simple program in Java. Save this code in Text Document as "**Start.java"**

```
public class Start()
{
   public static void main()
   {
         System.out.println("Hello World");
   }
}
```

Here:

class is a keyword

Start is a name the class. Here class is declared as public, so it is available to all the classes.

main() is the method which initiate and terminate the program (in java functions are called as Methods)

println() is method of object "**out"**. "**out**" is the object of class "**System".** println() prints the string "Hello World" on the screen/monitor.

Compiling and Running Java program

Java compiler first converts the source code into an intermediate code, known as bytecode or virtual machine code. To run the bytecode, we need the Java Virtual Machie (JVM). JVM exists only inside the computer memory and runs on top of the Operating System. The bytecode is not machine specific. The Java interpreter converts the bytecode into Machine code. The following diagram illustrates the process of compiling and running Java programs.

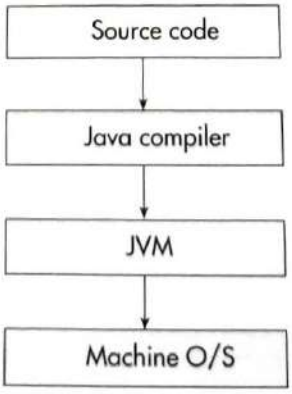


Fig. 2.3 Compiling and running a Java program

For compiling the program, the Java compiler javac is run, specifying the name of the source file on the command line as depicted here:

javac Start.java

The Java compiler creates a file called Start.class containing the bytecode version of the program. The java interpreter in JVM executes the instructions contained in

this intermediate Java bytecode version of the program. The Java interpreter is called with "java" at the command prompt.

C:\> java Start

Output:

Hello World

Here java command calls the Java interpreter which executes the Start.class (bytecode of Start.java).

3. Elements or Tokens in Java Programs

Java program contains different types of elements like white spaces, comments and tokens. A token is the smallest program element which is recognized by the compiler and which treats them as defined for the compiler. A program is a set of tokens which comprise the following elements:

Identifiers or names: Identifier is the name of variables, methods, classes etc.

Rules for framing Names or Identifiers.

- It should be a single word which contains alphabets a to z or A to Z, digits 0 to 9, underscore (_).
- It should not contain white spaces and special symbols.
- It should not be a keyword of Java.
- It should not be Boolean literal, that is, true or false.
- It should not be null literal.
- It should not start with a digit but it can start with an underscore.
- It can comprise one or more unicode characters which are characters as well as digits.

Conventions for Writing Names

- Names of packages are completely in lower-case letters such as mypackage, java.lang.
- Names of classes and interfaces start with an upper-case letter.
- Names of methods start with a lower-case character.
- Names of variables should start with a lower-case character.

Keywords: These are special words defined in Java and represent a set of instructions.

- The keywords represent groups of instructions for the compiler.
- These are special tokens that have a predefined meaning and their use is restricted.
- keywords cannot be used as names of variables, methods, classes, or packages.
- These are written in the lower case.
- Keywords of Java Language are as follows:

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

Literals: These are values represented by a set of character, digits, etc.

- A literal represents a value which may be of primitive type, String type, or null type.
- The value may be a number (either whole or decimal point number) or a sequence of characters which is called String literal, Boolean type, etc.
- A literal is a constant value.

Types of Literals:

i. Integer literals

- Sequences of digits.
- The whole numbers are described by different number systems such as decimal numbers, hexadecimal numbers, octal numbers, and binary numbers.
- Each number has a different set of digits.

Decimal Integer Literals

- These are sequences of decimal digits which are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9.
- Examples of such literals are 6, 453, 34789, etc.

Hex Integral Literals

- These are sequences of hexadecimal digits which are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.
- The values 10 to 15 are represented by A, B, C, D, E, and F or a, b, c, d, e, and f.
- The numbers are preceded by 0x or 0X. Examples are 0x56ab o0X6AF2, etc.

Octal Integer Literals

- These are sequences of octal digits which are 0, 1, 2, 3, 4, 5, 6, and 7.
- These numbers are preceded by 0. Examples of literals are 07122, 04, 043526.

Binary Literal

- These are sequences of binary digits.
- Binary numbers have only two digits—0 and 1 and a base 2.
- Examples of such literals are 0b0111001, 0b101, 0b1000, etc.

ii. Floating point literal

- These are floating decimal point numbers or fractional decimal numbers with base 10. Examples are 3.14159, 567.78, etc.

iii. Boolean literal

- These are Boolean values. There are only two values—true or false.

iv. Character literal

- These are the values in characters.
- Characters are represented in single quotes such as ‘A’, ‘H’, ‘k’, and so on.

v. String literal

- These are strings of characters in double quotes. Examples are “Delhi”, “John”, “AA”, etc.

vi. Null literal

- There is only one value of Null Literal, that is, null.

Separators: These include comma, semicolon, period(.), Parenthesis (), Square brackets [], etc

Symbol	*Name*	*Description*
()	Parentheses	Parentheses are used for the following purposes: 1. To change precedence level in expressions 2. To contain a list of parameters in a definition of methods 3. To contain expression in control statements 4. To enclose cast types in explicit type casting
{}	Braces	Braces are used for the following purposes: 1. To enclose definitions of classes and methods 2. To enclose blocks of statements 3. To initialize arrays and string objects 4. To enclose local scopes
[]	Square brackets	Square brackets are used for enclosing index values in array elements and for defining arrays
,	Comma	Commas are used for separating the following: 1. Variables in parameter or declaration lists 2. Identifiers in variable declarations 3. Expressions in for loop statement
;	Semicolon	A semicolon is used for terminating statements in a program.
.	Period	The period is used for the following: 1. To link an object of class with its method 2. To link classes with sub-packages and sub-packages to packages

Operators: Operators are mostly represented by symbols such as +, -, *, etc

Types of Operators:

Arithmetic Operators:

Operator	Description
+	Addition or Unary plus
-	Subtraction or Unary minus
*****	Multiplication
/	Division
%	Modulus

Relational Operators:

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

Logical Operators:

Operator	Description
&&	Greater than
||	Greater than or equal to
!	Less than

Assignment Operators:

Operator	Description
+=	Add and assign to
-=	Subtract and assign to
*=	Multiply and assign to
/=	Divide and assign to
%=	Modulus and assign to

Increment / Decrement Operators:

Operator	Description
++	Increment by one
--	Decrement by one

Bitwise Operators:

Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise compliment
>>	Shift Right
<<	Shift Left
>>>	Shift right with Zero fill

Conditional Operators:

Operator	Description
?:	Used to construct Conditional expression

4. Java Statements

A Statement is a instruction to the computer. A program is a set of statements or instructions. The statements specify the sequence of actions to be performed when some method or constructor is invoked. The statements are executed in the sequence in the specified order. The important Java statements are as follows.

Statement	Description
Empty statement	These are used during program development.
Variable declaration statement	It defines a variable that can be used to store the values.
Labeled statement	A block of statements is given a label. The labels should not be keywords, previously used labels, or already declared local variables.
Expression statement	Most of the statements come under this category. There are seven types of expression statements that include assignment, method call and allocation, pre-increment, post increment, pre-decrement, and post decrement statements.
Control statement	This comprises selection, iteration, and jump statements.
Selection statement	In these statements, one of the various control flows is selected when a certain condition expression is true. There are three types of selection statements including if, if-else, and switch.
Iteration statement	These involve the use of loops until some condition for the termination of loop is satisfied. There are three types of iteration statements that make use of while, do, and for
Jump statement	In these statements, the control is transferred to the beginning or end of the current block or to a labeled statement. There are four types of Jump statements including break, continue, return, and throw.
Synchronization statement	These are used with multi-threading
Guarding statement	These are used to carry out the code safely that may cause exceptions (such as division by zero, and so on). These statements make use of try and catch block, and finally

5. Command Line Arguments

- A Java application can accept any number of arguments from the command line.
- These arguments can be passed at the time of running the program. This enables a programmer to check the behavior of a program for different values of inputs.
- When a Java application is invoked, the runtime system passes the command line arguments to the application's main method through an array of strings.
- It must be noted that the number of arguments
-
- in an array. To ensure this, we can make use of the length property of the array. This is illustrated in

Program 2.2: Example showing command line argument

```
class vehicle
 public static void main(String args[])
{
      int x = args.length;
      for(int i=0; i<x; i++)
    {
      System.out.println(args[i]);
    }
}
Output
```

(After compiling, type the following lines on the command prompt. It produces the output as)

```
C:\> Java vehicle “Car Cycle Motorbike"
    Car
    Cycle
    Motorbike
```

Program 2.3: Illustration of command line

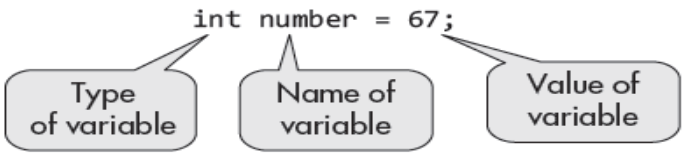
```
class Sum
{
public static void main{String args)

int  s=1;

for(int i e; icargs length; 1++){

s=s+Integer.parseInt(args(1]);

 System.out.prinln(" The addition of passed arguments 15" );

}
```

Output

```
After compiling, the following lines are typed on the command
prompt: It produces the output as

C:\> Java Sun 20 15 17
```

Declaration of Variables

- A program may involve variables: variables are objects whose values may change in the program.
- A variable is declared by first writing its type, followed by its name or identifier as illustrated here.



- However, a variable should also be initialized, that is, a value should be assigned to it before it is used in an expression. The line ends with a semicolon (:) as shown in the above figure.

Examples:

```
double price = 28.5;
char ch = "C":
String name - "John";
```

Program 2.4 illustrates the declaration and output of some data types

```
class  PrintOut
{
  public static void main (String args[])
  {
   String name = "Sunita"; //"name" is variable, value is "Sunita"
   String str = "Hello"; //String set has value- "Hello!"

   int length = 50; // Variable name "length", value 50
   int width = 8; // Variable name "width" value 8

   System.out.print("Name= " + name);
   System.out.print("Str = "+ str);  //print statement

   System. out.println();

   System.out.println("Length "+ length);
   System. out.println("Width " + width);

   System.out.println(str + name); // println statement
  }
}

c:\>javac PrintOut.java

c:\>java PrintOut
Name= SunitaStr = Hello
Length 50
Width 8
HelloSunita
```

6. User Input to Programs

- The class Scanner of package java.util to carry out input to the program.
- Java Scanner class is a text scanner that breaks the input into tokens using a delimiter. The delimiter is whitespace by default.
- Importing the class is the first step.

```
import java.util.Scanner;
```

- The object of the class Scanner is declared as follows.

```
Scanner scaninput = new Scanner (System. in);
```

- The object "`scaninput`" invokes the method `nextInt()` which reads the value typed by the user. This value is assigned to n. The value of m is similarly obtained.
- The other useful methods of Scanner class are nextDouble() and nextLine().

Program 2.6: Illustration of a user's input from keyboard into program

```
import java.util.Scanner;
public class Arithmetic
{
      public static void main(String[] args)
      {
      Scanner scaninput = new Scanner (System. in);
      int n;
      int m;
      System.out. print( "Enter the value of n : ");
      n=scaninput.nextInt();
      System.out. print( "Enter the value of m : ");
      m=scaninput.nextInt();

      System.out.println("Sum of two numbers is ="+(n+m));

      System.out.println("Product of two numbers is ="+ n*m);

      System.out.println("Modulus of (n % m) is =" + n%m);

      System.out.println("Division of two numbers is =" + n /m );
      }
}
```

Output

```
C:\>javac Arithmetic.java

C:\>java Arithmetic
Enter the value of n : 10
Enter the value of m : 3
Sum of two numbers is =13
Product of two numbers is =30
Modulus of (n % m) is =1
Division of two numbers is =3

C:\>
```

7. Escape Sequences

Escape Sequences character is preceded by a backslash (\)has a special meaning to the compiler. Escape sequences are as follows.

Table 2.7 Escape sequences

Escape sequences	Description	Example of code
\'	Single quote	"\'"
\"	Double quote	"\""
\\	Backslash	"\\"
\b	Back space	"\b"
\ddd	Octal character	"\132"
\uxxxx	Hexadecimal unicode character	"\u0042"
\f	Form feed	"\f"
\n	New line	"\n"
\r	Carriage return	"\r"
\t	Tab	"\t"

Program 2.9: Program on Escape Sequences

```
class Escape
{
   public static void main(String[] args)
   {
   int n=256, a=0, b=70;
   System.out.println("Value of a =" + a + "\n b= "+b +"\n");
   System.out.println("\u0041 \t" + " \u0042 \t"+"\132");
   System.out.println("\"Value of b\" = "+b);
   System.out.println("\'Value of n\' = " +n);
   }
}
```

Output:

```
C:\>javac Escape.java

C:\>java Escape
Value of a =0
 b= 70
A        B      Z
"Value of b" = 70
'Value of n' = 256
```

8. Comments

- Comments are Line of Text which is not a part of the compiled program.
- Comments are used for documentation to explain source code.
- They are added to the source code of the program.
- Java supports three types of comments as:

1. Single-line comment: These comments are started with two front slash characters (//)
   Example:
   ```
   // This is Single line comment
   ```

2. Multi-line comment : These comments are enclosed with /* and */
   Example:
   ```
   /* It is Multi line
      Comments */
   ```

3. Documentation comment: These comments are enclosed with /** and */. It is different from multi line comments in that it can extracted by javadoc utility to generate an HTML document for the program.
   Example:
   ```
   /** It is documentation
      Comments */
   ```

9. Programming Style.

- An analysis of the programming exercises will throw some light on the look and feel of a program. The team members should easily understand each other's code.
- For a beginner, it is better to develop a habit of writing a program in a proper style so that there is no conflict between the current habits and the company's imposition of style rule at a later stage.
- There are no set patterns of good style and bad style: however, if the programmer takes care of a few requirements on the programs as discussed, the resulting style will be better

1. **The program should present a clean and orderly look. In order to develop a clean program, the programmer can adapt the following measures:**

(a) The white spaces are neglected by the compiler.

(b) Indenting is another method often used to improve the looks and readability

(c) Use of Lambda expression and method reference of Java 8 enhances the look

(d) Include a space before and after operators

2. **The program should be easy to understand. The programmer should take care of the following aspects:**

(a) If it is team work, certain conventions about naming should be preset so that a team member can easily identify an item.

(b) The makers of Java have a set of rules which are followed in the Java library. The same rules or an even better convention may be set.

(c) Judicial use of comments can increase understandability. Use of too many comments makes confusion in the program.

(d) It is better to use already defined and tested library methods rather than user-defined methods

(e) Expressions like z = x++ + - - c*++k; should be avoided.

3. **Debugging should be easy. The programmer may adopt the following measures to ensure easy debugging.**

(a) The vertical alignment of a similar item enhances the ability to find errors.

(b) The code line should not be too long.

(c) The variables should be declared close to the places of their use

(d) If it is a big program, it should be divided into small segments. In Java, it is easy because the program may comprise separate classes.

(e) The methods should not be too big.

(f) Each source code file should have one class.
(g) The braces should be vertically aligned, if possible.

4. The program should be easy to use. The following points

(a) The input into program should be easy,

(b) The output should be self-explanatory should be taken care of

(c) The names used should imply the output type such as price, weight, length, and so on.

(d) Confusing and long names must be avoided.

5. **The program should be easy to maintain**

(a) The program should be easily modifiable to ensure simplicity in fixing errors.

(b) The comments can help in modification of the program, fixing errors.

6. The program should be fail-safe. The failure of a program should not be catastrophic.

II. Data Types Variables and Operators :

i. Introduction Data Types in Java

The Java language programs deal with the following entities:

I. Primitive data

2. Classes and objects

3. Interfaces and their references

4. Arrays

5. Methods

The primitive data and their types are defined independent of the classes and interfaces, and the arrays and methods derive their types from the first three entities.

An array is n collection of items that may be of primitive type, class objects, or references. The type of an array can be determined from the type of elements present in it.

Data Types in Java

- Data Type is the type of the data which computer accepts. Every variable and expression has a data type that is known at the compile time.
- The declaration of data type with a variable or expression limits the types of values that a variable can have or the expression it can evaluate.
- Java is an object-oriented programming language based on classes and interfaces.
- Java defines some primitive (basic) data types that are not reference types of any class or interface. Eight primitive (basic) types of data are defined in Java. The type names are also the keywords shown here in bold letters
  1. **Integral types**—byte, short, int, long
  2. **Floating point types** - float, double
  3. **Character type** -char
  4. **Boolean type** – Boolean values – True, False

There is a non-data type called void and no data can be of type void. This type is used for methods that do not return any value.

Java is a case-sensitive language. This means that it takes Area, area, and AREA are three different objects.

2. Declaration of Variables - Data Types

A variable is declared by first declaring its type followed by its identifier or name, which is given as follows:

```
type  Identifier;
```

Here type is the primitive data type and Identifier is the name of the variable.

Declaration and Initialization of the variable is as follows.

```
type identifier = value;
```

Ex:

byte n; // declares a variable of type byte.

short m = 67; // declares and initiates a short number

int length; //declares length-a variable of type int

length - 50; // value is re-assigned after declaration

char ch = 'A'; // declares a character variable ch.

Non-primitive Types

These are the class and interface types. The name of a class or interface is the name of type. A class object is declared as

```
Class_identifier   object_identifier;
```

Similarly, an interface reference is declared as

```
Interface_identifier   reference_identifier;
```

Example:

```
String str "Delhi":
```

Data Types

i. Integers

Integers are whole numbers, that is, they represent numbers that do not have a fractional part. The integers can be declared in four types according to the size of memory allocated for them

byte

short

int

long

Table 3.1 Integer types and ranges

Type name	Memory allocated in bytes	Range of values of variables
byte	1 byte (8 bits)	–128 to 127
short	2 bytes (16 bits)	–32,768 to 32,767
int	4 bytes (32 bits)	–2,147,483,648 to 2,147,483,647
long	8 bytes (64 bits)	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Program 3.1 illustrates the integer data types.

```
class DataType
{
      public static void main (String args[])
      {
      byte a= 4, b=8 ;// variables of type byte
      short c = 67,  d = 98; // variables of type short
      int e = 7000, f = 20000; // variables of type int
      long secondsInYear = 365 * 24 * 60 * 60; // long type
      //e=d+f;
      System.out.println("(b + a) = " + (b+a));
      System.out.println("b + a = " + b + a);
      System.out.println("c = "+ c + " \td = " + d);
      System.out.println("e = " + e + "\t f= " + f);
      System.out.println("Seconds in a year = "+  secondsInYear);
      }
}
```

Output:

```
C:\ >javac DataType.java
C:\ >java DataType
(b + a) = 12
b + a = 84
c = 67  d = 98
e = 7000          f= 20000
Seconds in a year = 31536000
```

ii. Characters

- A variable may have value in terms of a character in which the type of variable is char.
- These characters represent integer values.
- Java supports Unicode for the representation of characters.
- Unicode supports all the character sets of all the major languages of the world.
- The initial version of Unicode allocated 2 bytes for storing characters.
- The range of values for characters in the initial version of Unicode comprised from '\u0000' to '\uffff ' that is from 0 to 65535 both end values inclusive.

Program 3 2 illustrates anthmetic operations on character constants and variables

```
class Datachar {
      public static void main (String args[])
      {
      char ch1='E', ch2, ch3 ;

      System.out.println("ch1 =" +ch1); // printing ch1

      ch2=ch1++;
      System.out.println("ch2 =" + ch2); // printing ch2

      ch3=++ch1;
      System.out.println("ch3 =" + ch3); // printing ch3
      }
}
Output:
C:\ >javac Datachar.java
C:\ >java Datachar
ch1 =E
ch2 =E
ch3 =G
```

iii. Floating Point Numbers

- The numbers that are not whole numbers, or those that have fractional part, Examples are 3.141, 476.6, and so on.
- Java supports two types of such numbers.

Float: This type is used for single-precision decimal floating point numbers, that is, 7 digits after the decimal point. These numbers are stored on 4 bytes.

Program 3.4: Illustration for working with float and double data

```
class FloatType
{
      public static void main (String args[])
      {
      float width =20.0f,  length = 40.5f;
      float rectArea = length *  width;

      System.out.println("Width = "+ width);
      System.out.println("Length = "+ length);
      System.out.println("Rectangle Area = "+ rectArea);

      double dia=10.0, pi=3.14159;
      double areaCircle = pi* dia * dia / 4;

      System.out.println("Diameter " + dia);
      System.out.println("Area of Circle = " + areaCircle);
      }
}
```

Output

```
C:\>javac FloatType.java
C:\>java FloatType
Width = 20.0
Length = 40.5
Rectangle Area = 810.0
Diameter 10.0
Area of Circle = 78.53975
```

iv. Boolean Type - Data

- For dealing with logical statements, variable of type boolean is supported.
- The value of boolean type variable is either *true* or *false*.
- The boolean type variables are *unsigned* as similar to char type variables.
- The variable may be declared as follows :

```
boolean a;
a = x > y;
```

If the aforementioned logical statement is correct, The value of a is `true`, otherwise the value of a is `false`. In Java `true` and `false` are not converted into numerical values, which is the case in other Languages. A boolean type variable is allocated to one byte, that is, 8 bits for storing its values Program

3.5 illustrates the application of boolean type variables,

```
class Boolean
{
      public static void main (String args[])
      {
      double x= 5.5, y=10.5, p=4.0;
      int n=40, m=50;
      boolean a,b,c,d;

      a= x>y;
      b= y>p;
      c= y==x;
      d= x<=y;

      System.out.println("a = " + a +" and b =" +b);
      System.out.println("Now c= "+c+ " and d = "+d);
      }
}
```

```
C:\>javac Boolean.java

C:\>java Boolean
a = false and b =true
Now c= false and d = true
```

3. Type Casting

Converting one data type to another data type is called as Type Casting. There are two types of type casting. They are,

i. Implicit Type casting
ii. Explicit Type casting

i. Implicit Type casting

Implicit type casting is done automatically by a compiler when we assign a value to the variable.

Example:

```
int a=10;
double b;
b=a;
```

Here a is integer and d is double variables. Integer value is automatically converted into double by the compiler.

ii. Explicit Type casting

The explicit type casting is carried out by the following code:

type variable = (new_type) variable;

- It is illustrated by the following code lines:

```
double D = 6.865;

int A = (int) D;
```

- In such a conversion, there is loss of data

Program 3.7: Illustration of type casting.

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

      int a=4, b = 8, c = 9, d,e;
      double x= 3.0, y=6.5, z,k;

      d=c/a;
      k=a+y;
      e = a + (int)y;
      z=(double)c/a;

      System.out.println("k = " + k +" and e =" +e);
      System.out.println("d= "+ d + " and z  = "+ z);
      }
}
```

Output

```
C:\ >javac TypeCast.java

C:\ >java TypeCast
k = 10.5 and e =10
d= 2 and z  = 2.25
```

4. Scope of Variable Identifier

The scope and lifetime of a variable is the part of the program in which it is visible and holds the last entered value In Java, there are distinctly two types of scopes.
(a) class scope and
(b) method scope

- A variable declared in a class has class scope and scope of a variable declared in a method has method scope.
- The variables declared in a block have block scope.
- Thus, the variables defined in main() at the beginning have scope in entire main() method, however, those defined in a block have block scope.
- A block starts with the left brace ({) and ends with the right brace (}).

In the case of nested blocks of statements, the scope of variables is governed by the following rules.

1. The scope starts from the point the variable is defined in the block (declared and value assigned to it).

2. Although the variable may be defined anywhere in a block, it can be used only in the statements appear ing after its definition Therefore, there is no use in defining a variable at the end of block.

If there are nested blocks, a variable defined in the outer block is visible in the inner blocks also and it cannot be redefined with the same name in the inner blocks

Program 3.10: Illustration of block scope

```
public class ScopeA
{
static int x = 5; // Variable scope within  the Class
public static void main (String args[])
 {
 System.out.println("Class Scope variable - outside main()  x = " + x);
 int y = 10;
 System.out.println("Variable y Scope  within main()  y = " + y);
  { // Anonymous Block
    int z = 20;
    System.out.println("Variable z Scope within Anonymous Block  z = " + z);
  }
 }
}
```

```
E:\>javac ScopeA.java

E:\>java ScopeA
Class Scope variable - outside main()  x = 5
Variable y Scope  within main()  y = 10
Variable z Scope within Anonymous Block  z = 20

E:\>java ScopeA
```

5. Literal Constants, Symbolic Constants

i. Literal Constants

Literal Constants are as follows.

- Each character is a constant value.
- An array of characters or a string also represents a constant value.
- The digits in decimal system, octal system, or hexadecimal system represent constant values.
- The char literals are enclosed in single quotes (‘ ’). The examples ‘A’ and ‘B’. These may also be written as `\u0041` and `\u0042`.
- The string literals are enclosed in double quotes.
  Examples : "Delhi”, "I am going out."
- The decimal integral literals examples are 5417, 684
- The octal number literals are prefixed with 0 (Zero). Example. `072 , 042`
- The floating point literal examples are 684.62f, 5.245E+ 2f
- The literal of Boolean data type are true and false. In Java, neither true is converted to 1 nor

ii. Symbolic Constants

A Symbolic constant is a variable whose value does not change throughout the program. Some of the examples include PL NORTH, EAST etc.

Name of constant	*Value of constant*
Pi	3.145926535....
c (speed of light)	299,792,458 m/s
Gravitational Constant	$6.67300X\ 10^{-11}$ ($m^3\ Kg^{-1}\ s^{-2}$)
Natural Log (e)	2.718281828

It is usually preferred to declare the symbolic constants using all the capital letters in a program as follows:

```
public static final int c =299792458;
public static final double PI = 3.21415;
```

Program 3.13 illustrates the use of symbolic constant

```
public class SymbolicConst
{
  public static final double PI = 3.1415926535;
  public static void main (String args[])
  {
    double r = 25.0, perimeter;
    perimeter = 2*PI*r;
    System.out.println("radius=" + r);
    System.out.println("Perimeter of circle = " + perimeter);
   }
}

C:\>javac SymbolicConst.java

C:\>java SymbolicConst
radius=25.0
Perimeter of circle = 157.079632675
```

6. Formatted Output with printf() Method

In Java, the formatting of output may be carried out in two ways:

1. By using class Formatter 2. By method printf()

- The Formatter class is used to format the output.
- The printf() method is easy and simple to use, and hence, it is mote popular than other methods.
- The following formatting objectives may be realized by using print method
  i. Right and left justification
  ii. Precision of floating point numbers by regulating the number of digits after the decimal point
  iii. Aligning a number of numbers in a column
  iv. Controlling the placement of characters and strings at desired locations
  v. Representing integers in octal and hexadecimal systems
  vi. Representing floating point numbers in exponential form or regular form
  vii. Representing the time and date in different formats.

The syntax of the method printf () method is as follows

```
System.out.printf("Formatting string" variables separated by comma);
```

- The formatting string specifics the output format for each variable that consists of percent (%) sign followed by a conversion letter.
- Thus, the format string for output of an integer and character is "X" and or respectively.
- The order of variables in variable list should match with the list of formats in formatting string.
- The following Table lists the conversion letters for different types of variables.

Table 3.4 Conversion characters for different types of variables

Type variable	*Conversion letter*	*Formatting string*
Integers (base 10)	d	"%d"
Integer (base 8) octal	o	"%o"
Integer (base 16) hexadecimal	X or x	"%X" or "%x"
Character	C or c	"%C" or "%c"
String	S or s	"%S" or "%s"
Floating point number base 10	f	"%f"
Floating point number base 10 exponential form	E or e	"%E" or "%e"
Floating point number base 16	A or a	"%A" or "%a"
Floating point number in exponential (e) or regular format (f) depending upon the magnitude of variable. If the magnitude lies between 10^{-3} and 10^{7}, regular format is used. Outside these limits, it is formatted in exponential format.	G or g	"%G" or "%g"
For providing space between output values, the empty strings or tab "\t" may be used.	" " or \t	Example of code "%d %f %s" or "%d\t%f\t%s"

Program 3.14: illustration of formatting strings for output of different types of variables

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

      int n = 713;
      float x = 45.86f;
      double d= 56.754;

      String str = "Delhi";
      char ch = 'A';
      System.out.printf( "%d   %f  %f  %c \t %s \n", n, x, d,ch,
str);

      //for conversion into hexadecimal number
      System.out.printf("Hexadecimal value of 163 = %X \n", 163);

      //for conversion into octal
      System.out.printf("Octal value of 163 = %o\n", 163);
      }
}
```

```
Output

C:\>javac FormatPrintf.java

C:\>java FormatPrintf
713   45.860001  56.754000  A    Delhi
Hexadecimal value of 163 = A3
Octal value of 163 = 243
```

7. Static Variables and Methods

Static Variables:

- **The static variables are class variables**. **Only one copy of such variables is kept in the memory and all the objects share that copy**.
- **The static variables are accessed through class reference**, whereas the instance variables are accessed through class object reference
- The variables in a class may be modified by modifier static.
- The non-static variables declared in a class are instance variables Each object of the class keeps a copy of the values of these variables.

Static Methods:

- The static methods are similar to class methods and can be invoked without any reference of object of class, however, class reference (name of class) is needed, as in the following example The method like sart() is declared as static method in Math class and is called

```
Math.sqrt(5); // Finds square root of 5
```

The static method is called using the method name that is preceded by the class name; in this case. Math and period ().

Program 3.18: illustration of using static methods of class Math

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

 System.out.println("The Square root root of 16 = "+ Math.sqrt(16));
 System.out.println("The cubroot root of 27 = "+ Math.cbrt(27));
 //printing five random variables
 for(int i =1; i<=5;i++)
      System.out.println("Random Number "  + i + " = " +
                          (int)(100 *Math.random()));
      }
}
```

```
E:\>javac StaticMethods.java

E:\>java StaticMethods
The Square  root root of 16 = 4.0
The cubroot root of 27 = 3.0
Random Number 1 = 77
Random Number 2 = 69
Random Number 3 = 83
Random Number 4 = 2
Random Number 5 = 66

E:\>java StaticMethods
The Square root root of 16 = 4.0
The cubroot root of 27 = 3.0
Random Number 1 = 67
Random Number 2 = 31
Random Number 3 = 10
Random Number 4 = 13
Random Number 5 = 40
```

8. Attribute Final

Final Variable:

The value of a variable declared final cannot be changed in the program. It makes the variable a constant. A few examples of declarations are as follows:

```
final double PI =  3.14159; // The value of PI cannot be changed in its scope

final int M = 900;  // The value of M cannot be changed in its scope

final double X = 7.5643; // The value of x cannot be changed in its scope.
```

- As mentioned in the comments, the values of PI, M, and x cannot be changed in their respective scopes.

Final Method:

- The attribute final may be used for **methods** as well as for classes. These are basically connected with inheritance of classes.
- When final keyword is used with Java method, it becomes the final method.
- A final method cannot be overridden in a sub-class.

Final Class:

- A Java class with final modifier is called final class A final class cannot be sub-classed or inherited. Several classes in Java are final including String, Integer, and other wrapper classes.

- There are certain important points to be noted when using final keyword in Java
  i. New value cannot be reassigned to a variable defined as final in Java.
  ii. Final keyword can be applied to a member variable, local variable, method, or class.
  iii. Final member variable must be initialized at the time of declaration.
  iv. Final method cannot be overridden in Java
  v. Final class cannot be inheritable in Java
  vi. Final is different from finally keyword, which is used on Exception handling in Java

Example- 1:

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

      int n =10; // Normal variable
      final int f = 20; // final variable

      System.out.println("n = "+ n);
      System.out.println("f = "+ f);

      n = 50; // Now the value 50 is assigned to variable n
      f = 60; // Error : f is final variable can not be changed
      }
}
```

Output:

```
C:\>javac Final.java
Final.java:13: error: cannot assign a value to final variable f
        f = 60; // Error : f is final variable can not be changed
        ^
1 error
C:\>
```

Example-2:

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

      int n =10; // Normal variable
      final int f = 20; // final variable

      System.out.println("n = "+ n);
      System.out.println("f = "+ f);

      n = 50; // Now the value 50 is assigned to variable n
      System.out.println("n = "+ n);
      }
}
```

```
C:\>javac Final.java

C:\>java Final
n = 10
f = 20
n = 50
```

9. Introduction to Operators

An operator is a symbol that tells the computer to perform certain mathematical and logical calculations.

-The different types of Java operators are,

a) Arithmetic operators
b) Relational operators
c) Logical operators
d) Increment or decrement operators
e) Assignment operators
f) Conditional operators
g) Bitwise operators
h) Special operators

10.Precedence and Associativity of Operators

If the expression contains several operators with the same precedence level, the expression is evaluated according to its *associativity*.

For example, in

Z = 6*4%5;

Both the operators * and % have same precedence level.

Highest						
++ (postfix)	– – (postfix)					
++ (prefix)	– – (prefix)	~	!	+ (unary)	– (unary)	(*type-cast*)
*	/	%				
+	–					
>>	>>>	<<				
>	>=	<	<=	instanceof		
==	!=					
&						
^						
|						
&&						
||						
?:						
->						
=	op=					
Lowest						

11. Assignment Operator (=)

Assignment operators:

-Assignment operators are used to assign the result of an expression to a variable.

Operator	Meaning
=	Assignment

-In addition, java has a set of short-hand assignment operators of the form

V OP=EXP;

- It is equivalent to

V=V OP EXP;

-The short-hand assignment operators are

+= , -= , *= , /= , %=

Example:

a+=b ---------- a=a+b
a-=b ---------- a=a-b
a*= b ---------- a=a*b
a/=b ---------- a=a/b
a%=b ---------- a=a%b

12. Basic Arithmetic Operators

Operator	Meaning
+	Addition or Unary plus
-	Subtraction or Unary minus
*	Multiplication
/	Division
%	Modulo division

-Integer division truncates any fractional part.

-The modulo division produces the remainder of an integer division.

Examples: a+b, a-b, a*b, a/b, a%b

Arithmetic Expression Types:

Integer arithmetic expression:

-An arithmetic operation involving only integer operands is called integer arithmetic.

int + int = int
int - int = int
int * int = int
int / int = int (truncates decimal part)
int % int = int (results remainder)

Real arithmetic expression:

- An arithmetic operation involving only real operands is called real arithmetic.

real + real = real
real - real = real
real * real = real
real / real = real
real % real = not allowed

Mixed-mode arithmetic expression:

-An arithmetic operation involving one operand is real and the other operand is integer, then it is called mixed-mode arithmetic.

real+int=real
real-int=real

real*int=real
real/int=real
real%int=not allowed

13. Increment (++) and Decrement (- -) Operators

-The increment operator is ++ which means +1
-The decrement operator is - - which means -1
-The operand must be either incremented or decremented by 1.
Example:
m=5;y=++m; results m=6 and y=6.
m=5;y=m++; results m=6 and y=5.
m=5;y= --m; results m=4 and y=4.
m=5;y=m--; results m=4 and y=5.

14. Ternary Operator or Conditional operators:

-It is also known as Ternary operator.
-The symbols used to construct a conditional expression are ? and **:**
-A conditional expression is of the form,

exp1?exp2:exp3;

Example:
(a>b)?System.out.println("a is greater")**:** System.out.println ("b is greater");
- It is equivalent to if-else statement in java

Example:

```
if(a>b)
   System.out.println("a is greater");
else
   System.out.println ("b is greater");
```

15. Relational Operators

-The comparison between two operands or expressions is done with the help of relational operators.

Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
>=	Greater than or equal to
==	Equality
!=	Inequality

Example:

a>b
a>=b
a<b
a<=b
a==b
a!=b

16.Boolean Logical Operators

-The java language has three logical operators.

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

-The logical operators &&, || are used when we we want to test more than one condition.
&& - used when all the conditions must be true.
|| - used when any of the conditions must be true.

-A logical expression yield a value 0 or 1, according to the truth table.

Operand1	Operand2	Operand1&&operand2	Operand1||operand2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Example:
(a>b)&&(a>c)
(a>b)||(a>c)
-The logical not is used as a negation or complement of the expression.
Example:
!(y<10) means (y>=10)

17.Bitwise Logical Operators.

Bitwise operators:

-'C' provides bitwise operators that operate on data at the bit-level.
-Bitwise operators interpret operands as string of bits.
-These bit strings are then interpreted according to data type.

-Bitwise operators are of 2 types.
i) Logical bitwise operators.
ii) Shift bitwise operators.

i) Bitwise Logical operators

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	One's complement

Bitwise AND operator:

-The Bitwise AND (&) is a binary operator that requires two integral operands(character or integer).
-It does a bit-by-bit comparison between two operands.
-The result of the comparison is 1 only when both bits are 1, otherwise it is 0.

First operand bit	Second operand bit	Result(&)
0	0	0
0	1	0
1	0	0
1	1	1

Bitwise OR operator:

-The Bitwise inclusive OR (|) is a binary operator that requires two integral operands(character or integer).
-It does a bit-by-bit comparison between two operands.
-The result of the comparison is 0 only when both bits are 0, otherwise it is 1.

First operand bit	Second operand bit	Result(|)
0	0	0
0	1	1
1	0	1
1	1	1

Bitwise exclusive OR:

-The Bitwise exclusive OR (^)is a binary operator that requires two integral operands(character or integer).
-It does a bit-by-bit comparison between two operands.
-The result of the comparison is 1 only if one of the operands is 1, otherwise it is 0.

First operand bit	Second operand bit	Result(^)
0	0	0
0	1	1
1	0	1
1	1	0

One's complement:

-The one's complement (~) is a unary operator applied to an integral value.
-The result is 1 when the original bit is 0 and it is 0 when the original bit is 1.

Original bit	Result(~)
0	1
1	0

ii) Shift Bitwise operators:

-The shift Bitwise operators move bits to the right or left.

Operator	Meaning
>>	Bitwise shift right
<<	Bitwise shift left
>>>	Shift right with zero fill

Bitwise shift-right operator:

-It moves some number of bits from right to left.
-It requires two integral operands.

Syntax:

Operand1 >> Operand2;

-Here, Operand1 is value to be shifted.
Operand2 is number of bits to be shifted.

Bitwise shift-left operator:

-It moves some number of bits from left to right.
-It requires two integral operands.

Syntax:

Operand1 << Operand2;

-Here, Operand1 is value to be shifted.
Operand2 is number of bits to be shifted.

18.Special Operators.

- Java supports some special operators such as,
    a) instanceof operator
    b) member selection operator

a) instanceof operator:

- The instanceof is an object reference operator and returns true if the object on the left-hand side is an instance of the class given on the right hand side.
- This operator allows us to determine whether the object belongs to a particular class or not.

Example:

```
person  instanceof  student
```

- It is true if the object person belongs to the class student, otherwise it is false.

b) member selection operator:

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

Example:

```
person.age            // Reference to the variable age
person.salary( )      // Reference to the method salary( )
```

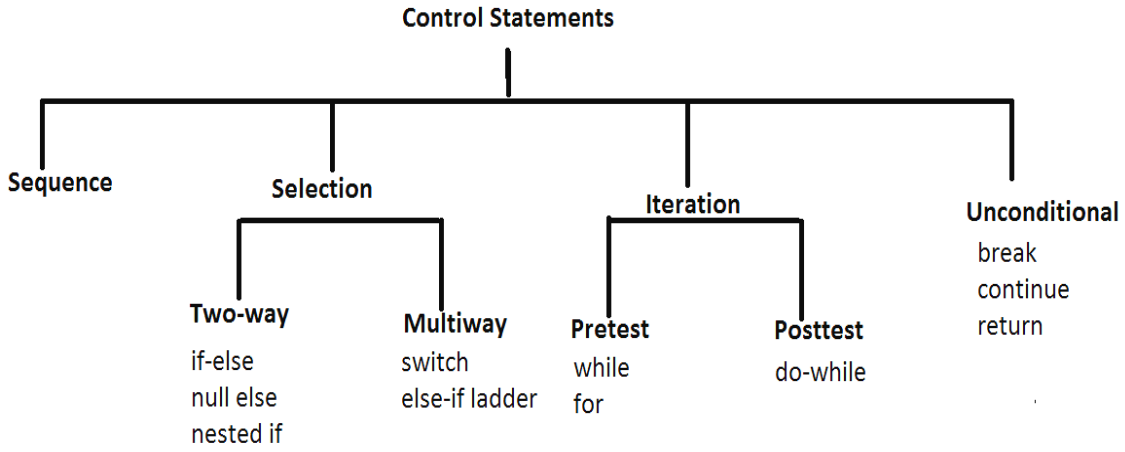
- It is also used to access classes and sub packages from a package.

III. Control Statements:

1. Introduction

CONTROL STATEMENTS: (FLOW OF CONTROL)

- A Control statement is a statement used to control the flow of execution in a Java Program.



SELECTION STATEMENTS:

-Also called as conditional or decision-making control statements.

-There are two types in Selection control statements.

i) Two-way selection control statements

ii) Multi-way selection control statements

i) Two-way selection control statements:

-The different two-way selection statements are,

a) if-else statement

b) null else statement

c) Nested-if statement

2. if Expression - also called as " null else statement "

Syntax:

```
if(condition)
   {
      statements;
   }
next statement;
```

Example:

```
if(a==2)
{
      p++;
 }
System.out.println("program over");
```

3. Nested if Expressions

-if within if is called as Nested-if.

Syntax:

```
if(condition-1)
{
      if(condition-2)
      {
            Statement-1;
      }
      else
      {
            Statement-2;
      }
}
else
{
      if(condition-3)
      {
            Statement-3;
       }
      else
      {
            Statement-4;
      }
}
next statement;
```

Example:

```
if(a>b)
   {
      if(a>c)
      {
            System.out.println("a is greater");
      }
      else
      {
        System.out.println ("c is greater");
      }
   }
else
   {
      if(b>c)
      {
            System.out.println("b is greater");
      }
      else
      {
            System.out.println("c is greater");
      }
   }
```

4. if–else Expressions

Syntax:

```
if(condition)
   {
        true-block statements;
   }
else
   {
        false-block statements;
   }
next statement;
```

Example:

```
if(a>b)
   {
        System.out.println("a is greater");
   }
else
   {
        System.out.println("b is greater");
   }
```

b) else-if ladder statement:

Syntax:

```
if(condition-1)
   {
     Statement-1;
   }
else if(condition-2)
   {
     Statement-2;
   }
……..……
……..……..
else if(condition n-1)
   {
     Statement-(n-1);
   }
else
   {
     Statement-n;
   }
next statement;
```

Example:

```
if(a>b&&a>c&&a>d)
   {
     System.out.println(“a is greater”);
   }
else if(b>a&&b>c&&b>d)
   {
     System.out.println(“b is greater”);
   }
else if(c>a&&c>b&&c>d)
   {
     System.out.println(“c is greater”);
   }
else
   {
     System.out.println(“d is greater”);
   }
```

5. Ternary Operator?:

In Java, the **ternary operator** is a type of Java conditional operator. The meaning of **ternary** is composed of three parts.

The **ternary operator (? :)** consists of three operands. It is used to evaluate Boolean expressions. The operator decides which value will be assigned to the variable. It is the only conditional operator that accepts three operands.

It can be used instead of the if-else statement. It makes the code much more easy, readable, and shorter.

Syntax:

Expression1 ? Expression2 : Expression3

- The first expression is the test condition.
- If it evaluates true, the Expression2 is executed; otherwise Expession3 is executed.

Example-1:

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

            int a=10;
            int b = (a<20)? 100 :200; // a < 20  if statement
            System.out.println("b= "+b);
      }
}
```

```
Output:
C:\>javac Ternary.java

C:\>java Ternary
b= 100
```
Example-2:

```
public class Ternary
{
      public static void main (String args[])
      {
            int a=10;
            int b = (a>20)? 100 :200; // a>20 – else statement
            System.out.println("b= "+b);
      }
}

Output:
C:\>javac Ternary.java

C:\>java Ternary
b= 200
```

6. Switch Statement

Syntax:

```
switch(expression)
   {
case value-1:statement-1;break;
case value-2:statement-2;break;
……..
……..
case value-n:statement-n;break;
default: default statement;
   }
next statement;
```

Example:

```
switch(digit)
   {
case 0: System.out.println(“ZERO”);break;
case 1: System.out.println(“ONE”);break;
case 2: System.out.println(“TWO”);break;
case 3: System.out.println(“THREE”);break;
case 4: System.out.println(“FOUR”);break;
case 5: System.out.println(“FIVE”);break;
case 6: System.out.println(“SIX”);break;
case 7: System.out.println(“SEVEN”);break;
case 8: System.out.println(“EIGHT”);break;
case 9: System.out.println(“NINE”);break;
default: System.out.println(“Enter between 0-9”);
}
```

7. Iteration Statements

LOOP STATEMENTS:

-The iteration control statements are also called as Repetition or Iteration control statements.
-A looping process includes the following four steps.

-Setting and initialization of a counter.
-Execution of the statements in the loop body.
-Test for a specified condition (loop control expression) for execution of a loop
-Incrementing or Decrementing counter.

i) Pretest and Posttest loops:

-In a **Pretest loop**, the condition is checked before we execute a loop body.
-It is also called as **entry-controlled loop**.
-In the **Posttest loop**, we always execute the loop body atleast once.
-It is also called as **exit-controlled loop**.

8. while Expression

a) while statement:

Syntax:

```
while(condition)
  {
loop body;
  }
next statement;
```

Example:

```
n=10,i=1,sum=0;
while(i<=n)
   {
sum=sum+i;
i++;
        }
System.out.println("sum="+sum);
```

9. do–while Loop - forLoop

Syntax:

```
do
   {
      loop body;
   }while(condition);
next statement;
```

Example:

```
n=10,i=1,sum=0;
do
   {
sum=sum+i;
i++;
   } while(i<=n);
System.out.println("sum="+sum);
```

10. forLoop

for statement:

Syntax:

```
for(initialization;condition;inc or dec)
  {
      loop body;
  }
next statement;
```

Example:

```
n=10,i,sum=0;
for(i=1;i<=n;i++)
{
      sum=sum+i;
}
System.out.println("sum="+sum);
```

11.Nested for Loop

Syntax:

```
for(initialization;condition;inc or dec)
   {

   for(initialization;condition;inc or dec)
   {
        Inner loop body;
   }

   Outer loop body;
   }
next statement;
```

Example:

```
   n=10,i, j, sum=0;
   for(i=1;i<=n;i++)
   {

        sum=sum+i;
   }
   System.out.println("sum="+sum);
```

12.For–Each for Loop

The Java for-each loop or enhanced for loop. It provides an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements. The advantage of the for-each loop is that it eliminates the possibility of bugs and makes the code more readable. It is known as the for-each loop because it traverses each element one by one.

***Advantages*:**

1. Less clutter in code, especially when iterators are used.
2. Less chances of errors.
3. Improves overall readability of program.

***Limitations*:**

1. It is designed to iterate in forward direction only.
2. In iteration, it takes a single step at a time.
3. It cannot simultaneously traverse multiple arrays or collections.

Syntax:

```
for (Object obj : Collection_name)
{
   Body of loop
}
```

Example:

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

      int myArray[] = {10,20,30,40,50};

      for (int x:myArray)
       {
            System.out.println(" "+x);
       }
      }
}
```

Output:

```
   C:\>javac ForEach.java

   C:\>java ForEach
    10
    20
    30
    40
    50
```

13.Break Statement

Unconditional control statements:

-The unconditional control statements are,

a) break statement
b) continue statement

break statement:

-The break statement skips from the loop or block in which it is defined.
-The control then automatically goes to the first statement after the loop or block.
-The general format is

break;

Example:

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

   //printing the values from 1 to 10
   for(int i =0; i<10;i++)
      {
       if (i==5)
         break;  //Break Statement
       System.out.println( i );
      }
   }
}
```

Output:

```
C:\>javac Break.java

C:\>java Break
0
1
2
3
4
```

Here loop is stopped due to break statement.

14.Continue Statement.

-The continue statement is used for continuing next iteration of loop statements.
-When it occurs in the loop, it does not terminate but it skips the statements after it.
-It is useful when we want to continue the program without executing any part of the program.
-The general format is

continue;

Example:

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

      //printing the values from 1 to 10
      for(int i =0; i<10;i++)
         {
          if (i==5)
            continue; // Continue statement
          System.out.println( i );
         }
      }
}
Output:
-------
C:\>javac Break.java

C:\>java Continue
0
1
2
3
4
6
7
8
9
```

Here 5 is not printed because of continue statement. The Iteration at the condition `i = =5` is skipped or jumped to next statement.

UNIT-2

I. Classes & Objects:

1. Classes and Objects: Introduction
2. Class Declaration and Modifiers
3. Class Members
4. Declaration of Class Objects
5. Assigning One Object to Another
6. Access Control for Class Members
7. Accessing Private Members of Class
8. Constructor Methods for Class
9. Overloaded Constructor Methods
10. Nested Classes
11. Final Class and Methods
12. Passing Arguments by Value and by Reference
13. Keyword this.

II. Methods:

1. Introduction
2. Defining Methods
3. Overloaded Methods
4. Overloaded ConstructorMethods
5. Class Objects as Parameters in Methods
6. Access Control
7. Recursive Methods
8. Nesting of Methods
9. Overriding Methods
10. Attributes Final and Static.

I. Classes & Objects :

1. Classes and Objects: Introduction

Classes:

- A class is defined as collection of similar objects.
- Classes are user-defined data types and behave like the built-in types of a programming language.
- In the real-world, classes are invisible only objects are visible.

Example:

- man is an object representing a class called Animal.
- We can see the object called man but we cannot see the class called Animal.

Syntax:

Animal man;

- It will create an object man belonging to the class Animal.

Objects:

- Objects are basic run-time entities in an object-oriented system.

(or)

Any real world entity is called an object.

(or)

Objects are the combination of data and methods.

Example: Person, Place, bank account, …., so on.

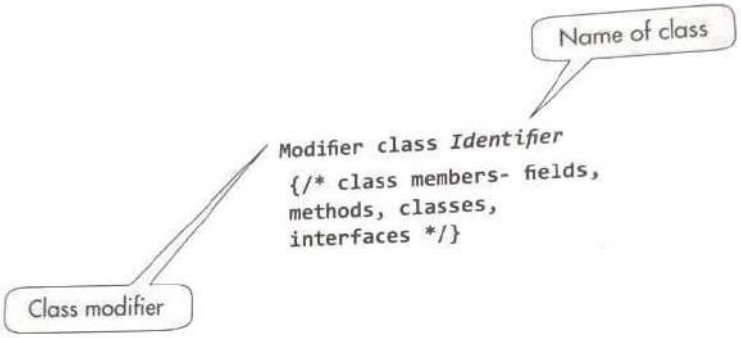
- In the real-world only objects are visible but classes are invisible.
- The most important benefits of an objects are
  - Modularity
  - Reusability
- The properties of objects are two types
  - visible
  - invisible
- Let man is an object, then visible properties are eyes, ears, hands, legs,…so on and invisible properties are name, blood group,…. so on.
- Every object contains three basic elements
  - Identity (name)
  - State (variables)
  - Behavior (methods)

Object = Data + Methods

2. Class Declaration and Modifiers

A class declaration starts with the Access modifier. It is followed by keyword class, which is followed by the name or identifier. The body of class is enclosed between a pair of braces { }.

Syntax:



Example:

```
public class MyFarm
{ double length;      // instance variable
double width;         // instance variable
}
```

- The class name starts with an upper-case letter, whereas variable names may start win lower-case letters.
- In the case of names consisting of two or more words as in MyFarm, the other words for with a capital letter for both classes and variables. In multiword identifiers, there is no blank space between the words
- The class names should be simple and descriptive.
- Class names should start with an upper-case letter and should be nouns. For example, it could include names such as vehicles, books, and symbols.
- It should have both upper and lower-case letters with the first letter capitalized
- Acronyms and abbreviations should be avoided

Class modifiers:

- Class modifiers are used to control the access to class and its inheritance characteristics.
- Java consists of packages and the packages consist of sub-packages and classes Packages can also be used to control the accessibility of a class
- These modifiers can be grouped as (a) access modifiers and (b) non-access modifiers. Table 5.l gives a description of the various class modifiers.

Table 5.1 Summary of class modifiers

Modifiers	*Description*
Access modifiers	
No modifier	The class declared without an access modifier is accessible to the classes in its own package only. It is not visible to classes in other packages.
public	A class declared public is accessible to all classes in all packages.
private	A class specified as private has the highest degree of protection and is accessible to other members of outer (nesting) class.
protected	The class is accessible to other members of outer (nesting) class as well as by classes derived from the outer class.
Non-access modifiers	
final	A class declared final cannot have sub (derived) classes.
abstract	A class declared abstract must have one or more abstract methods as its members or its super class contains an abstract method that is not fully defined in the present class.
strictfp	The modifier makes all the float and double expressions declared in the class and in its nested classes to be FP-strict. In an FP-strict expression, all intermediate values are either float value set or the double value set. This implies that the same result is obtained if the operations involving floating point variable is performed in any platform. The results of all FP-strict expressions must be supported by IEEE 754 arithmetic for floats and double operands.

Examples:

1. A class **without modifier.**

```
class Student
{
      /* class body*/
}
```

2. A class with modifier

```
public class Student
{
      /* class body*/
}
```

(or)

```
private class Student
{
      /* class body*/
}
```

(or)

```
protected class Student
{
      /* class body*/
}
```

(or)

```
final class Student
{
      /* class body*/
}
```

(or)

```
abstract class Student
{
      /* class body*/
}
```

3. Class Members

The class members are declared in the body of a class. These may comprise fields (variables in a class). methods, nested classes, and interfaces. The members of a class comprise the members declared in the class as well as the members inherited from a super class. The scope of all the members extends to the entire class body.

The fields comprise two types of variables

1. **Non Static variables :** These include *instance* and *local variables* and vanes in scope and value.

   (a) Instance variables: These variables are individual to an object and an object keeps a copy of these variables in its memory.

   (b) Local variables: These are local in scope and not accessible outside their scope.

2. **Class variables (Static Variables) :** These variables are also qualified as static variables. The values of these variables are common to all the objects of the class. **The class keeps only one copy of these variables and all the objects share the same copy.** As class variables belong to the whole class, these are also called class variables.

Example:

```
class CustomerId
{
      static int  count=0; // static variable
      int id; // instance variable
      CustomerId() // Constructor
      {
          count++;
          id = count ;
      }
      int getId() // Method
      {
          return id;
      }
      int localVar()
      {
        int a=10; //Local variable

         return a;
      }
}

class Application
{
   public static void main(String[] args)
   {
      CustomerId obj = new CustomerId();
      System.out.println("Customer Id = " + obj.getId());
      System.out.println("Local Variable = " + obj.localVar());

   }
}
```

Output:

```
C:\>javac Application.java

C:\>java Application
Customer Id = 1
Local Variable = 10
```

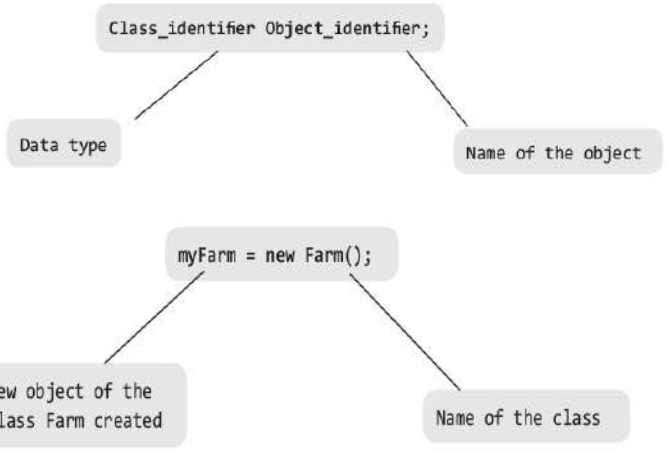
4. Declaration of Class Objects

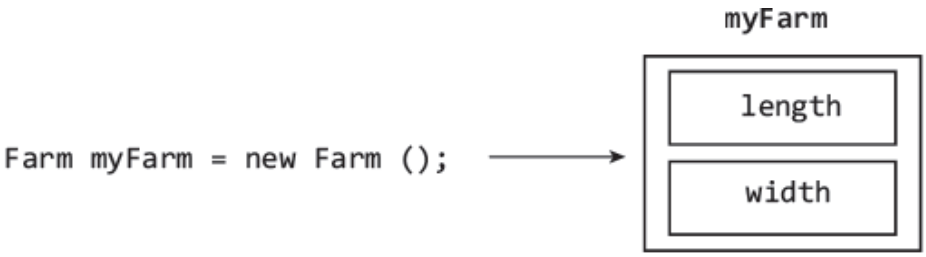
Creating an object is also referred to as instantiating an object.

- Objects in java are created dynamically using the new operator.
- The new operator creates an object of the specified class and returns a reference to that object.

Syntax: (creating an object)

className objectReference=new className();





Example:

```
Farm myFarm = new Farm();
```

5. Assigning One Object to Another

Java provides the facility to assign one object to another object

Syntax:

new_Object = old_object;

all the properties of old_object will be copied to new object.

Example:

```
class Farm
{
   double length;
   double width;
   double area()
   {
            return length*width;
    }
}
public class FarmExel
{
   public static void main (String args[])
   {
     Farm farm1 = new Farm(); //defining an object of Farm
     Farm farm2 = new Farm(); //defining new object of Farm

     farm1.length = 20.0;
     farm1.width = 40.0;

     System.out.println("Area of form1= " + farm1.area());

     farm2 = farm1;  // Object Assignment

     System.out.println("Area of form2 = " + farm2.area());

   }
}
```

Output:

```
C:\>javac FarmExel.java

C:\>java FarmExel
Area of form1= 800.0
Area of form2 = 800.0
```

6. Access Control for Class Members

In Java, There are three access specifiers are permitted:

- public
- protected
- private

The coding with access specifiers for variables is illustrated as

```
Access_specifier type identifier;
```

Detials of Access specifiers are as follows.

Table 5.2 Access specifiers and their effect

Access specifier	*Access*
No access specifier	Access is permitted to any other class belonging to the same package.
public	Access is permitted to any other class in any Java package.
protected	Access is permitted from subclass of this class in any package and to any class in the same package. The access is not permitted to any other class in another package.
private	Access is permitted only to members of the same class.

7. Accessing Private Members of Class

- Private members of a class, whether they are instance variables or methods, can only be accessed by other members of the same class
- Any outside code cannot directly access them because they are private. However, interface public method members may be defined to access the private members
- The code other than class members can access the interface public members that pass on the values.

Example:

```
public class Farm
{
      private double length;  // private member data
      private double width;  // private member data

      //definition of public methods
      public double area() {return length*width;}
      public void setSides(double l, double w)
                  { length=l; width = w;     }
      public double getLength(){return length;}
      public double getWidth(){return width;}
}
```

```
class FarmExe3
{
      public static void main (String args[])
      {

      Farm farm1 =new Farm();
      double farmArea;

      farm1.setSides(50.0,20.0);
      farmArea = farm1.area();

      System.out.println("Area of farm1 = "+ farmArea);
      System.out.println("Length of farm1 = "+ farm1.getLength());
      System.out.println("Length of farm1 = "+ farm1.getWidth());

      }
}
```

Output

```
C:\>javac PrivateMembers.java

C:\>java FarmExe3
Area of farm1 = 1000.0
Length of farm1 = 50.0
Length of farm1 = 20.0
```

In the above program, the two object variables `length` and `width` are declared private. The first thing is to assign values to these variables for an object. This is done by defining a public method `setSides()`, which is invoked by the class object for entering values that are passed to length and width variables The method setsides may be defined as

```
public void setsides (int 1, int w){length = 1; width = w;}
```

The class also defines another method `area()` to which the values are passed for calculation of area when the method area() is invoked by the object. For obtaining values of **`length`** and **`width`** by outside code, two public methods are defined as

```
public double getLength(){return length;} //Function for getting length
public double getWidth(){return width;}   // Function for getting width
```

These methods may be invoked by objects of the class to obtain the values of variables as follows:

```
farm1.getLength()

farm1.getWidth()
```

8. Constructor Methods for Class

- A constructor is a special method of the class and it is used to initialize an object whenever the object is created.
- A Constructor is a special method because,
  - Class name and Constructor name both must be same
  - Doesn't contain any return type
  - Automatically executed when object is created.
- Constructors are two types
  i. Default Constructor (without arguments)
  ii. Parameterized Constructor (with arguments)

Example:

```
class Perimeter
 {
  Perimeter()  // default Constructor
     {
       System.out.println("No parameters");
      }
   Perimeter(double r)  // Parameterized Constructor
    {
      System.out.println("Perimeter of the Circle="+(2*3.14*r));
    }
  Perimeter(int l, int b) // Parameterized constructor
    {
      System.out.println("Perimeter of the Rectangle="+(2*(l+b)));
    }
 }
class ConstructorDemo
 {
   public static void main(String args[])
    {
      Perimeter p1=new Perimeter();
      Perimeter p2=new Perimeter(10);
      Perimeter p3=new Perimeter(10,20);
    }
 }
Output
   E:\>javac ConstructorDemo.java

   E:\>java ConstructorDemo
   No parameters
   Perimeter of the Circle=62.800000000000004
   Perimeter of the Rectangle=60
```

9. Overloaded Constructor Methods

Like other methods, the constructors may also be overloaded. **The name of all the overloaded constructor methods same as the name of the class, but parameters have to be different either in number of type a order of parameters in each definition**.

Example:

```
class Perimeter
 {
  Perimeter()
     {
       System.out.println("No parameters");
      }
   Perimeter(double r)  //Constructor Overloading
    {
      System.out.println("Perimeter of the Circle="+(2*3.14*r));
    }
  Perimeter(int l, int b) // Constructor Overloading
    {
      System.out.println("Perimeter of the Rectangle="+(2*(l+b)));
    }
 }
class ConstructorDemo
 {
   public static void main(String args[])
    {
      Perimeter p1=new Perimeter();
      Perimeter p2=new Perimeter(10);
      Perimeter p3=new Perimeter(10,20);
    }
 }
Output
   C:\>javac ConstructorDemo.java

   C:\>java ConstructorDemo
   No parameters
   Perimeter of the Circle=62.800000000000004
   Perimeter of the Rectangle=60
```

10.Nested Classes

A nested class is one that is declared entirely in the body of another class or interface. The class, which is nested, exists only long as the enveloping class exists. Therefore, the scope of inner class is limited to the scope of enveloping class. There are four types of nested class.

Nested static class is like any other static member of the enveloping class.

i. Member Inner Class.
ii. Anonymous Class
iii. Local Class
iv. Static Nested Class

i. Member Inner Class.

A class which is declared within class is called Member inner class.

The inner class has access to all the members of the enveloping class including the members declared public, protected or private.

Example:

```
class Outer
{
      double outer_x;
      double outer_y;
      Outer (double a, double b)
      {
         outer_x = a;
         outer_y = b;
      }
      double outer_add()
      {
         return outer_x+outer_y;
      }
        void outer_display()
      {
      Inner in = new Inner();
      in.inner_display();
      }

      class Inner  // Inner Class
      {
            void inner_display()
            {
               System.out.println("x+y = "+ outer_add());
            }
      }
}
```

```
class NestedClassDemo
{
      public static void main (String args[])
      {
            Outer obj =new Outer(10,20);
            obj.outer_display();
      }
}
```

Output:

```
C:\>javac NestedClassDemo.java

C:\>java NestedClassDemo
x+y = 30.0
```

ii. Anonymous Class

- **Anonymous classes are inner classes without a name.**
- It is defined inside another class. Because class has no name it cannot have a constructor method and its objects cannot be declared outside the class.
- Therefore, an anonymous class must be defined and initialized in a single expression.
- An anonymous class may be used where the class has to be used only once.
- An anonymous class extends a super class or implements an interface, but keywords extend or implements do not appear in its definition. On the other hand, the names of super class and interface do appear.
- An anonymous class is defined by operator new followed by class name it extends, argument list for the constructor of super class, and then the anonymous class body.

```
Example:
abstract class Person
{
   abstract void display(); //abstract method
}

class AnonymousClass
{
   public static void main (String args[])
   {
    Person obj = new Person() { // Creating an object of Anonymous class
    void display()
    {
     System.out.println("In display() method ");
    }
   }; // anonymous class closes

  obj.display(); // Calling anonymous class method
 }
}
```

```
Output:
-------
C:\>javac AnonymousClass.java

C:\>java AnonymousClass
In display() method
```

iii. Local Class

- **A local class is declared in a block or a method**, and hence, their scope is limited to the block of method. The general properties of such classes are as follows
    - These classes can refer to local variables or parameters, which are declared final
    - These are not visible outside the block in which they are declared and hence, the access modifiers such as public, private, or protected do not apply to local classes.

Example:

Example:

```
class LocalClassDemo
{
  public static void main (String args[])
  {
      class Local     // Local class defined
      {
            int x;
            Local(int a) { x =a; }
            public  void display()
            {
              System.out.println("x = "+ x);
            }
      }

      Local localObj = new Local(10);
      localObj.display();
  }
}
```

Output

```
C:\>javac LocalClassDemo.java

C:\>java LocalClassDemo
x = 10
```

iv. Static Nested Class

- The main benefit of Static Nested classes is that their reference is not attached to outer class reference.
- Object may be accessed directly.
- These classes cannot access non-static variables and methods. They can access only static variables and methods
- Static nested class can be referred by its class name.

Example:

```
class Outer
{
   static double outer_x;
   static double outer_y;
   Outer (double a, double b)
   {
      outer_x = a;
      outer_y = b;
   }
   static double outer_add()
   {
      return outer_x+outer_y;
   }
        static void outer_display()
   {
   Inner in = new Inner();
   in.inner_display();
   }

   static class Inner  // Static Inner Class
   {
         void inner_display()
         {
            System.out.println("x+y = "+ outer_add());
         }
   }
}

class StaticNestedClass
{
   public static void main (String args[])
   {
         Outer obj =new Outer(10,20);
         obj.outer_display();
   }
}
```

Output:

```
C:\>javac StaticNestedClass.java

C:\>java StaticNestedClass
x+y = 30.0
```

11.Final Class and Methods

- A final class is a class that is declared as a **final** which cannot have a subclass

Example:

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

class B extends A
{
   int b;
        B(int x,int y)
   {
       super(x);
       this.b=y;
   }
   void display()
   {
     System.out.println("b = "+ b);
   }
}

class FinalClass
{
   public static void main (String args[])
   {
         A objA= new A(10);
         B objB= new B(100,200);

         objA.display();
         objB.display();
   }
}
```

Output:

```
C:\>javac FinalClass.java
FinalClass.java:11: error: cannot inherit from final A
class B extends A
                ^
1 error
```

12.Passing Arguments by Value and by Reference

Arguments are the variables which are declared in the method prototype to receive the values as a input to the Method(Function).
Example:

```
int  add(int  a, int  b) // method prototype
{
    //Body of the method add
   return   a+b;
}
```

Here **a** and **b** are called as arguments. (also called as **formal arguments**)

Arguments are passed to the method from the method calling
Ex:
int x=10,y=20;
add(x , y); // method calling

Here x and y are **actual arguments**.

Arguments can be passed in two ways

i. Call by value
ii. Call by reference

i. <u>Call by value</u>

In call by value **actual arguments are copied in to formal arguments**.
Example:

```
class Swap
{
   int a,b;
   void setValues(int p, int q)
   {
      a=p;
      b=q;
   }
   void swapping()
   {
         int temp;
         temp =a;
         a=b;
         b=temp;
   }
   void display()
   {
     System.out.println("In Swap Class: a= "+a+" b= "+b);
   }
```

```
}
class CallByValue
{
  public static void main (String args[])
  {
   int x=10,y=20;
   System.out.println("Before Swap : x= "+x+ " y="+y);
   Swap obj =new Swap();
   obj.setValues(x,y);
   obj.swapping();
   obj.display();
   System.out.println("After Swap : x= "+x+ " y="+y);
  }
}
```

Output:

```
C:\>javac CallByValue.java

C:\>java CallByValue
Before Swap : x= 10 y=20
In Swap Class: a= 20 b= 10
After Swap : x= 10 y=20
```

ii. Call by reference

In call by reference the **object will be passed to the method as an argument**. At that time the actual and formal arguments are same.
That means any occurs in actual arguments will be reflected in the formal arguments.

Example:

```
class Swap
{
      int a,b;
      void setValues(Swap objSwap)
      {
         a = objSwap.a;
         b = objSwap.b;
      }
      void swapping()
      {
            int temp;
            temp =a;
            a=b;
            b=temp;
      }
      void display()
      {
        System.out.println("In Swap Class: a= "+a+" b= "+b);
      }
}
```

```
class CallByReference
{
  public static void main (String args[])
  {
Swap obj =new Swap();
obj.a=10;
obj.b=20;
System.out.println("Before Swap : obj.a = "+ obj.a+" obj.b="+ obj.b);

obj.setValues(obj); // call by reference
obj.swapping();
obj.display();
System.out.println("After Swap: obj.a = "+ obj.a+ " obj.b="+ obj.b);
  }
}
```

Output:

```
C:\1. JAVA\PPT Programs>javac CallByReference.java

C:\1. JAVA\PPT Programs>java CallByReference
Before Swap : obj.a = 10 obj.b=20
In Swap Class: a= 20 b= 10
After Swap : obj.a = 20 obj.b=10
```

13.Keyword this.

The keyword **this** provides reference to the current object.

Example:

```
class Add
{
     int a,b;
     void setValues(int a, int b)
     {
        this.a = a;
        this.b = b;
     }

     void add()
     {
       System.out.println("Sum = "+ (a+b) );
     }
}
class ThisKeyword
{
  public static void main (String args[])
  {
```

```
      Add obj= new Add();
      obj.setValues(10,20);
      obj.add();
 }
}
```

Output:

C:\ >javac ThisKeyword.java

C:\ >java ThisKeyword

Sum = 30

II. Methods

1. Introduction

A method in Java represents an action on data or behaviour of an object. In other programming languages, **the methods are called functions** or procedures.

A method is an encapsulation of declarations and executable statements meant to execute desired operations.

A few types of actions and behaviour of Methods are as follows.

1. It could involve carrying out computation on data presented to method.
2. The action may simply be rearranging the elements of an object. for example, sorting arrays.
3. The action may comprise finding or searching elements in the list.
4. The action may simply be the initialization of an object.
5. Methods may create images, voice, and multimedia as well as display.
6. Methods may define how an object will communicate with other objects.
7. It may simply answer an enquiry.
8. A method may tell whether an action is permissible or not.

- In Java, a method must be defined inside a class and an interface.
- An interface represents an encapsulation of constants, classes, interfaces, and one or more abstract methods that are implemented by a class.
- A method cannot be defined inside another method, but it can be defined inside a local class

2. Defining Methods

A method definition comprises two components:

1. ***Header*** that includes modifier, type, identifier, or name of method and a list of parameters.
   - The parameter list is placed in a pair of parentheses.
2. ***Body*** that is placed in braces ({ }) and consists of declarations and executable statement and other expressions.

Method definition:

```
Modifier return_type  method_name (datatype Parameter_Name,…)
{
  /*Statements --
    Body of the method*/
}
```

Modifier description is as follows.

Table 6.1 Brief description of non-access modifiers

Non-access modifiers	*Description*
static	With this modifier, a method may be called without an object of class. However, class reference is needed.
final	Method declared final cannot be modified (overridden) in a subclass.
native	It is used only for methods. It indicates that the method is implemented in a platform-dependent language like C.
transient	A variable is declared transient if it is desired that it should not be part of the persistent state of the object, that is, the transient variable value is not stored, whereas non-transient variables are stored when an object is stored in persistent memory.
synchronized	It is applied to make method thread safe. When it is used, it ensures that the method can be accessed by only one thread at a time.
volatile	It indicates to the compiler that a variable can be modified by the other thread.

Example:

```
class Add
{
      int a,b;
      void setValues(int x, int y) // method with two arguments
      {
         a = x;
         b = y;
      }

      void add() // method without arguments
      {
        System.out.println("Sum = "+ (a+b) );
      }
}
class MethodDemo
{
  public static void main (String args[])
  {
      Add obj= new Add();
      obj.setValues(10,20); // method calling
      obj.add();
 }
}
```

Output:

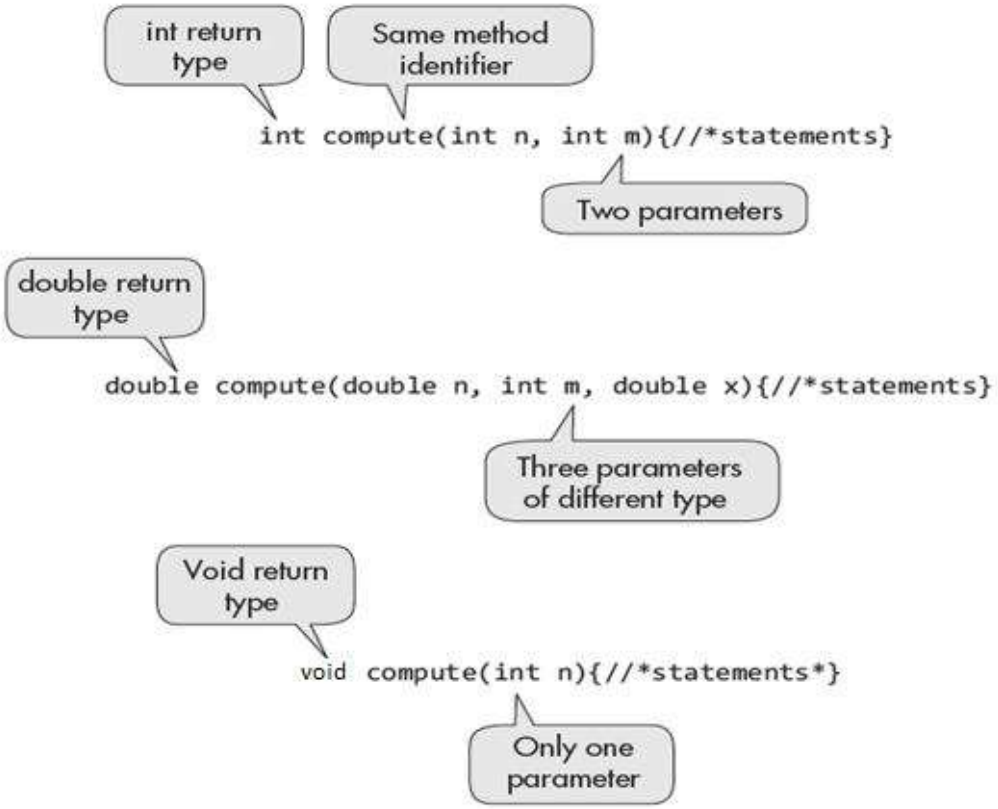
```
C:\ >javac MethodDemo.java

C:\ >java MethodDemo
Sum = 30
```

3. Overloaded Methods

Methods with the same name and scope are permitted provided they have different signatures that include the following:

i. Number of parameters
ii. Data types of parameters
iii. Their order in the parameter list

The compiler executes the version of the method whose parameters match with the arguments. For example, the following types of declarations in the scope are permissible:



Example:

```
class Add
{
int a,b;
void setValues(int a, int b) // method with two arguments
{
   this.a = a;
   this.b = b;
}
void add() // method without arguments
{
  System.out.println("In add() method Sum = "+ (a+b) );
}
//Method overloding - integer datatype arguments
void add(int a, int  b)
{
 System.out.println("In add(int, int) Method- sum= "+ (a+b) );
}

//Method overloding - double datatype arguments
void add(double a, double b)
{
 System.out.println("In add(double, double) MethodSum = "+ (a+b) );
}
}

class MethodOverload
{
  public static void main (String args[])
  {
  Add obj= new Add();
  obj.setValues(10,20); // method calling
  obj.add(); // calling method without arguments
  obj.add(15,30);// calling method with integer datatype arguments
  obj.add(10.3, 30.4); // calling method with double datatype arguments
  }
}
```

```
C:\>javac MethodOverload.java

C:\>java MethodOverload
In add() method Sum = 30
In add(int, int) Method- sum= 45
```

```
In add(double, double) MethodSum = 40.7
```

4. Overloaded ConstructorMethods

A constructor method is automatically called whenever a new object of the class is constructed. It **creates and initializes the Object**.

A constructor method has the **same name as the name of class** to which it belongs. **It has no type and it does not return any value**. **It only initializes the object**.

The **constructor method may also be overloaded** by changing the number of default values. Therefore, constructors with different parameters may be declared. For the remaining parameters, it will pick up default values when these are not specified in the object definition.

Example:

```
class AddDemo
{
 int a,b;
 AddDemo() // Constructor without arguments
 {
   a=10;
   b=20;
 }

 // Constructor Overloading with arguments
 AddDemo(int x, int y)
 {
   a = x;
   b = y;
 }

void add() // method without arguments
{
 System.out.println("a = " + a + ", b = "+ b+ ":  Sum = "+ (a+b) );
}
}

class ConstructorOverload
{
  public static void main (String args[])
  {
  AddDemo obj1= new AddDemo(); //calling constructor without arguments
  obj1.add();

  AddDemo obj2= new AddDemo(150,60); //calling constructor with arguments
  obj2.add();
 }
}
```

Output:

```
C:\>javac ConstructorOverload.java

C:\>java ConstructorOverload
a = 10, b = 20:  Sum = 30
a = 150, b = 60:  Sum = 210
```

5. Class Objects as Parameters in Methods

Objects can be passed as parameters to the Methods just like primitive data types. It is called as Call by Reference.

Example:

```
class AddDemo
{
 int a,b;

 void add(AddDemo obj2) // method with Object as an
argument
 {
   System.out.println("Sum = "+ (obj2.a + obj2.b)
);
 }
}
class ObjectAsParameters
{
  public static void main (String args[])
  {
 AddDemo obj1= new AddDemo();
 obj1.a=180;
 obj1.b=50;
 obj1.add(obj1);
 }
}
```

Output:

```
C:\>javac ObjectAsParameter.java

C:\>java ObjectAsParameter
Sum = 230
```

6. Access Control

Java supports access control at the class level and at the level of class members. At the class level, the following two categories are generally used:

i. **default case no modifier applied :** In the default case, when no access specifier is applied, the class can be accessed by other classes only in the same package
ii. **public :** A class declared public may be accessed by any other class in any package.

In a class, Java supports the information hiding mechanism so that the user of a class does not get to know how the process is taking place. A class contains data members and method members or a nested class.

To access any of the members data method, or nested class-can be controlled by the following modifiers.

i. private
ii. protected

iii. public
iv. default case-no modifier specified

i. **private :** The private members can o**nly be accessed by the** other members (methods) of the **same class**. No other code outside the class can access them.
Ex:

```
private int x;
private int getx()
{
   return x;
}
```

ii. **protected :** The protected members can accessed by own class and **derived class only**.

```
protected int x;
protected int getx()
{
   return x;
}
```

iii. public : The public members can accessed **by all the classes**.
Ex:

```
public int x;
public int getx()
{
   return x;
}
```

iv. **default case (** no modifier specified): The default members can accessed by **all the classes within the package only.**

```
int x;
int getx()
{
   return x;
}
```

7. Recursive Methods

A Method which is calling itself is called as Recursive Method.

Example: Recursive method to find factorial of a given number.

```
class Fact
{
     int factorial (int n)
     {
        if(n<2)
          return n;
        else
          return n*(factorial(n-1));
     }

}

class FactDemo
{
   public static void main(String[] args)
   {
     Fact obj =new Fact();
     int n=5;
     int res = obj.factorial(n);
     System.out.println("Factorial of " + n + " = " +res);
   }
}
```

Output:

```
C:\>javac FactDemo.java

C:\>java FactDemo
Factorial of 5 = 120
```

8. Nesting of Methods

A method calling in another method with in the class is called as Nesting of Methods.

Example:

```
class Rectangle
{
 void perimeter(int l, int w)
 {
   System.out.println("Length ="+l+", Width= "+w);
   System.out.println("Perimeter = " + (l+w));
 }

 void area(int l, int w)
 {
    perimeter(l,w); // Nesting of Method
    System.out.println("Area = " + (l*w));
 }
}

class RectangleDemo
{
   public static void main(String[] args)
   {
        Rectangle obj = new Rectangle();
        obj.area(5,4);
   }
}
```

Output:

```
C:\ >javac RectangleDemo.java

C:\ >java RectangleDemo
Length =5, Width= 4
Perimeter = 9
Area = 20
```

9. Overriding Methods

See this topic in Inheritance.

10. Attributes Final and Static

See Attribute Final from UNIT-1: Section II – Topic - 8

See Static Variable and Method from UNIT-1 : Section II – Topic -7

UNIT-3

I. Arrays:

1. Introduction
2. Declaration and Initialization of Arrays
3. Storage of Array in Computer Memory
4. Accessing Elements of Arrays
5. Operations on Array Elements
6. Assigning Array to Another Array
7. Dynamic Change of Array Size
8. Sorting of Arrays
9. Search for Values in Arrays
10. Class Arrays
11. Two-dimensional Arrays
12. Arrays of Varying Lengths
13. Three-dimensional Arrays
14. Arrays as Vectors.

II. Inheritance:

1. Introduction
2. Process of Inheritance
3. Types of Inheritances
4. Universal Super Class- Object Class
5. Inhibiting Inheritance of Class Using Final
6. Access Control and Inheritance
7. Multilevel Inheritance
8. Application of Keyword Super
9. Constructor Method and Inheritance
10. Method Overriding
11. Dynamic Method Dispatch
12. Abstract Classes
13. Interfaces and Inheritance.

III. Interfaces:

1. Introduction
2. Declaration of Interface
3. Implementation of Interface
4. Multiple Interfaces
5. Nested Interfaces
6. Inheritance of Interfaces
7. Default Methods in Interfaces
8. Static Methods in Interface
9. Functional Interfaces
10. Annotations.

I. Arrays:

1. Introduction

An array is a structure consisting of a group of elements of the same type. When a large number of data values of the same *type* are to be processed, it can be done efficiently by declaring an array of the data *type*.

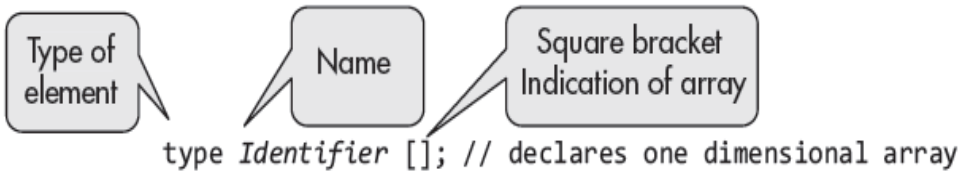
The complete data gets represented by a **single object with a single name** in the computer memory. **An array is a sequence of objects of the same data type**. The type of data that the array holds becomes the type of the array, which is also called *base type* of the array.

If the array elements have values in whole numbers, that is, of type int, the type of array is also int. If it is a sequence of characters, the type of array is char;

If it is an array of floating point numbers of type float, the type of array is also float.
An array can hold objects of a class but cannot be a mixture of different data types.

Syntax:

```
        datatype arrayName[];
or
        type identifier[];
```



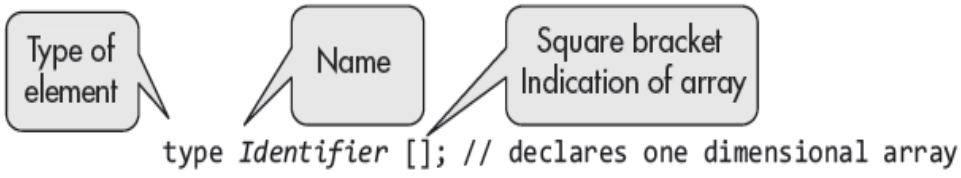
Examples:

```
int numbers []; // an array of whole numbers
char name []; // A name is an array of characters
float priceList[]; // An array of floating point numbers.
```

2. Declaration and Initialization of Arrays

Declaration of Array:

```
        datatype arrayName[];
or
        type identifier[];
```



Examples:

```
int numbers []; // an array of whole numbers
char name []; // A name is an array of characters
fl oat priceList []; // An array of floating point numbers.
```

Initialization of Arrays:

An array may be initialized by mentioning the values in braces and separated by commas. For example, the array pencils may be initialized as below:

```
int pencils [] = {4, 6, 8, 3};
```

3. Storage of Array in Computer Memory

- The operator *new* allocates memory for storing the array elements.
- For example, with the following declaration

```
int[] numbers = new int[4];
```

  here 4 elements will be created and assigned to the array "**numbers"**
- The declaration and initialization may as well be combined as:

```
int numbers [] = {20,10,30,50};
```

- A **two-dimensional** array may be declared and initialized as,

```
int [][] array2d = new int [][] {{1, 2, 3}, {4, 5, 6}};
```

  or as

```
int [][] array2D = {{1, 2, 3}, {4, 5, 6}};
```

4. Accessing Elements of Arrays

- The individual member of an array may be accessed by its index value.
- The index value represents the place of element in the array.
- The first element space is represented by numbers [0], and index value is 0.

  Note that the value of an array element is different from its index value.

  Example:

```
int numbers [] = {20,10,30,50};
```

  Here
  number[0] is the first element
  number[1] is the second element
  number[2] is the third element and so on…

  the value at number[0] is 20,
  value at number[1] is 10,
  value at number[2] is 30,
  value at number[3] is 20,
  value at number[4] is 50.

Determination of Array Size

- The size or length of an array may be determined using the following code:

```
int arraySize = array_identifier.length;
```

- The size of array numbers is determined as:

```
int size = numbers.length;
```

- The elements of a large array may be accessed using a *for* loop.

  For example, the elements of array numbers may be accessed as

```
for (int i = 0; i<size; i++)
     System.out.println(x);
```

Example-1:

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

   int numbers[] = new int[4];

   numbers[0] = 10;
   numbers[1] = 20;
   numbers[2] = 30;
   numbers[3] = 40;

   for(int i =0 ;i<numbers.length; i++)
           System.out.println(numbers[i]);

   }

}
```

Output:

```
C:\>javac NumArray.java

C:\>java NumArray
10
20
30
40
```

Example-2: One Dimensional Array - NumArray2.java

```
class NumArray2
{
   public static void main(String[] args)
   {
    int numbers [] = {20,10,30,50};

    for(int i=0 ; i<numbers.length; i++ )
           System.out.println(numbers[i]);
   }
}
```

Output:

```
C:\>javac NumArray2.java

C:\>java NumArray2
10
20
30
40
```

Example-3: One Dimensional Array - NumArray3.java

```
class NumArray3
{
      public static void main(String[] args)
      {

      int numbers[] = new int[]{100,200,300,400} ;

      for(int i =0 ;i<numbers.length; i++)
            System.out.println(numbers[i]);

      }
}
```

Output:

```
C:\>javac NumArray3.java

C:\>java NumArray3
100
200
300
400
```

Example-4: One Dimensional Array - StringArray.java

```
class StringArray
{
      public static void main(String[] args)
      {
      String names[] = {"Red", "Blue", "Green", "Black", "White"} ;

      for(String i :names)
            System.out.println(i);

      }
}
```

Output:

```
C:\>javac StringArray.java

C:\>java StringArray
Red
Blue
Green
Black
White
```

Use of *for–each* Loop

- the *for–each* loop may be used to access each element of the array.

```
for (int x: numbers)
   System.out.println(x);
```

- For a two-dimensional array the nested *for–each* loops are used.

Example : Two Dimensional Array – TwoDimArray.java

```
class TwoDimArray
{
    public static void main(String[] args)
    {
   int pArray[][]= {{1,2,3},{4,5,7}};
   for(int[] y : pArray)
        {
          for(int x : y)
             System.out.print( x + " ");
          System.out.println();
        }
    }
}
```

Output:

```
C:\>javac TwoDimArray.java

C:\>java TwoDimArray
1 2 3
4 5 7
```

5. Operations on Array Elements

i. Arithmetic Operations on Arrays:

Arithmetic operations can be applied on Array elements.

Example:

```
int[] array1 = new int []{1,2,3,4,5};
array1[0] = array1[0] + 10;
```

Here the value of the first element is added 10. Now its value is changed from 1 to 11.

ii. Arrays as Parameters of Methods

Arrays can be passed to methods just like variables.

Example:

```
display(array1);  // Method calling
  . . . . ..
void display(int[] array)  //Method definition
{
     for (int x : array)
         System.out.println(x + " ");
}
```

Example: ArrayOperations.java

```
class ArrayOperations
{
  public static void main(String args[])
  {
   int[] array1 = new int []{1,2,3,4,5};

   System.out.println("Before Adding - Array elements are :");
   display(array1); //Passing an array to display() method
   System.out.println();

   // Operations on Arrays
   for(int i =0; i< array1.length; i++)
     array1[i] = array1[i] + 10;  // Adding 10 to each element

    System.out.println("After Adding - Array elements are :");
    display(array1);
    }

    static void display(int[] array)  //Method definition
    {
      for (int x : array)
           System.out.print(x + "   ");
    }
   }
```

Output:

```
C:\>javac ArrayOperations.java

C:\>java ArrayOperations
Before Adding - Array elements are :
1   2   3   4   5

After Adding - Array elements are :
11   12   13   14   15
```

6. Assigning Array to Another Array

- In Java, an array may be assigned to another array of same data type.
- In this process, the second array identifier is the reference to the first array.
- The second array is not a new array, instead only a second reference is created.
- This is illustrated in this program, array1 is assigned to array2. Then, array1 is modified array2 also gets modified, which shows is not an independent array.

```
class ArrayAssignment
{
     public static void main(String args[])
     {
      int[] array1 = new int []{1,2,3,4,5};
      int[] array2 = new int[array1.length];


      System.out.println("Array-1 elements are :");
        display(array1);
        System.out.println();

      array2 = array1; // Array Assignment

        System.out.println("Array-2 elements are :");
        display(array2);  // Method calling
        System.out.println();

      //Modification of array2 elements
        for(int i=0;i<array2.length;i++)
          array2[i]+=100;  //adding 100 to each element of array2

      System.out.println("After Modification of Array2 \n Array-1
elements are :");
        display(array1);
        System.out.println();
       }

     static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

```
C:\ >javac  ArrayAssignment.java

C:\ >java ArrayAssignment
Array-1 elements are :
1  2  3  4  5
Array-2 elements are :
1  2  3  4  5
After Modification of Array2
 Array-1 elements are :
101  102  103  104  105
```

7. Dynamic Change of Array Size

Java allows us to change the array size dynamically during the execution of the program. In this process the array destroyed along with the values of elements. In the following program, the array contains 5 elements. It is again defined with 10 elements with the same array name.

Example:

```
class DyanamicArraySize
{
     public static void main(String args[])
     {
      int[] array1 = new int []{1,2,3,4,5};

      System.out.println("Before Changing Array Size: array1 = ");
      display(array1);

      //Changing array size
      array1 = new int[10];

      System.out.println("\nAfter Changing Array Size: array1 = ");
      display(array1);

      //adding values to the array elements
      for(int i=0;i<10;i++)
        array1[i] = 5*(i+1);

      System.out.println("\nAfter Modification : array1 = ");
      display(array1);
     }
     static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

Output:

```
C:\>javac DynamicArraySize.java

C:\>java DyanamicArraySize
Before Changing Array Size: array1 =
1   2   3   4   5
After Changing Array Size: array1 =
0   0   0   0   0   0   0   0   0   0
After Modification : array1 =
5   10   15   20   25   30   35   40   45   50
```

8. Sorting of Arrays

Sorting of arrays is often needed in many applications of arrays. For example, in the preparation of "examination results" , "order of grades acquired by students" or "Student names in alphabetical order of dictionary style". The arrays may be sorted in ascending or descending order. Several methods are used for sorting the arrays that include the following:

1. Bubble sort

2. Selection sort

3. Sorting by insertion method

1. Bubble Sort

This method of sorting is the simplest to understand but the most inefficient one; however, it can be use fully employed for short arrays.

In the process, if the sorting is done in ascending order, the first element is compared with the second element. If the value of the second is smaller than the first, the elements are inter changed, that is, the second is made first and first is made second.

However, if the second is higher than the first, then no action is taken. The second element is then compared with the third element and the aforementioned procedure is repeated. The third is then compared with the fourth.

The process is repeated till the last . This process places the largest value as the last element. The process is again element repeated for the next largest value from the remaining elements until the last element of the array is reached. Thus, the process is repeated (n-1) times to completely sort the array.

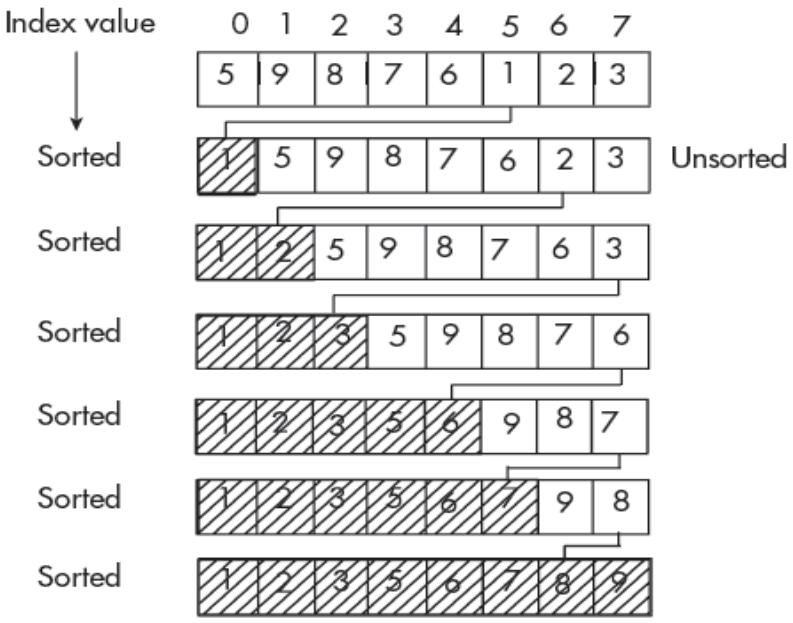


Fig. 7.6 Bubble sort method

Example:

```
class BubbleSort
{
     public static void main(String args[])
     {
      int[] array1 = new int []{5,8,9,2,4,1,7,6};

      System.out.println("Before Sorting : array1 = ");
      display(array1);

      // Buble Sorting
      int t;
      for(int i=0; i<array1.length; i++)
      {
         for(int j=array1.length-1; j>0 ; j--)
          {
            if(array1[j-1]>array1[j])
            {
            t=array1[j];
            array1[j] = array1[j-1];
            array1[j-1] =t;
            }
          }
      }
      System.out.println("\nAfter Sorting : array1 = ");
      display(array1);
```

```
    }
    public static void display(int[] array)  //Method definition
    {
      for (int x : array)
          System.out.print(x + "   ");
    }
}
```

Output:

```
C:\ >javac BubbleSort.java

C:\ >java BubbleSort
Before Sorting : array1 =
5   8   9   2   4   1   7   6
After Sorting : array1 =
  1    2   4   5   6   7   8   9
```

2. Selection Sort

- Let us take an array with elements 9, 7, 4, 3, 6, and 8.
- If the array is being sorted in ascending order, pick the element with the lowest value, that is 3, and place at the first place as shown in the second line of Figure.
- From the remaining elements that are 9, 7, 4, 6, 8, again pick the lowest value, that is, 4 and place it next to 3, as shown in the third line of the figure.
- Then, from the remaining elements with values 9, 7, 6, 8, again pick the lowest value, that is, 6 and place it next to 4.
- From the remaining three values, 9, 7, 8, again pick the lowest value, that is, 7 and place it next to 6 on its right side.
- Then, from the remaining two values 9 and 8, pick the lowest, that is, 8 and place it next to 7.
- The highest value goes to the last place. The array is sorted with the lowest value at the first place and the largest value at the end.
- The process is better than the bubble sort but still not the most efficient.

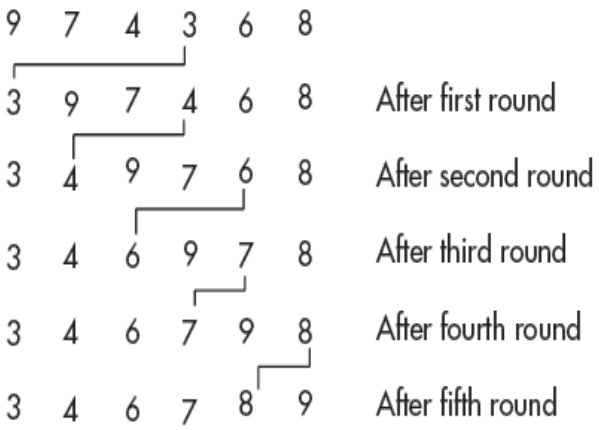


Fig. 7.7 Selection sort

Example:

```
class SelectionSort
{
     static int min=0;
     public static void main(String args[])
     {
      int[] array1 = new int []{9,7,4,3,6,7};

      System.out.println("Before Sorting : array1 = ");
      display(array1);

      // Selection Sort

      int minIndex=0;

      for(int i=0; i<array1.length; i++)
      {
         min=array1[i];
         for(int j=i+1; j<array1.length ; j++)
          {
              if(min>array1[j])
           {
              min=array1[j];
                 minIndex=j;
           }
          }
         for(int j=minIndex;j>i;j--)
            array1[j]=array1[j-1];
           array1[i]=min;
```

```
      }
     System.out.println("\nAfter Sorting : array1 = ");
     display(array1);
     }

     public static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

Output:

```
C:\ >javac SelectionSort.java

C:\ >java SelectionSort
Before Sorting : array1 =
9   7   4   3   6   7
After Sorting : array1 =
3   4   6   7   7   9
```

3. Insertion Sort

- Sorting algorithm builds a final sorted array one item at a time.
- In this method, the value at any index is compared to all the prior elements.
- The input data is inserted into the correct position in the sorted list and the process is repeated until no input element remains.

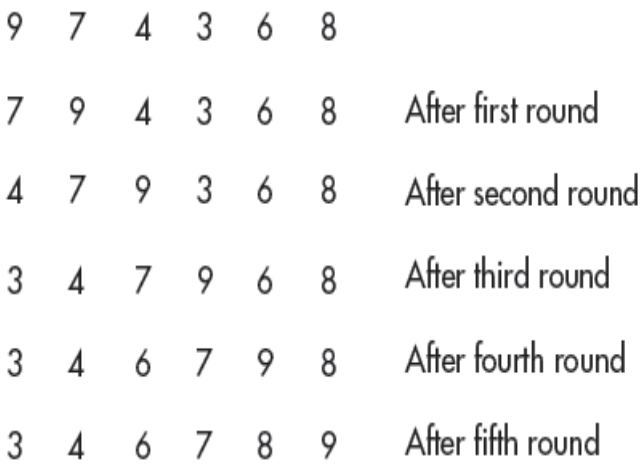


Fig. 7.8 Insertion sort

Example: InsertionSort.java

```
class InsertionSort
{
     static int min=0;
     public static void main(String args[])
     {
   int[] array1 = new int []{9,7,4,3,6,7};

   System.out.println("Before Sorting : array1 = ");
   display(array1);

   // Insertion Sort
   int len = array1.length;
   int key =0;

   int i=0;
   for(int j=1; j<len;j++)
   {
       key= array1[j];
       i=j-1;
       while(i>=0 && array1[i]>key)
       {
         array1[i+1] = array1[i];
         i=i-1;
         array1[i+1]=key;
       }

           /* System.out.println("\nAfter "+(i+1)+" Iteration
array1=");
       display(array1);*/
    }

    System.out.println("\nAfter Sorting : array1 = ");
    display(array1);
     }

     public static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
}
```

Ouput:

```
C:\ >javac InsertionSort.java

C:\ >java InsertionSort
Before Sorting : array1 =
9   7   4   3   6   7
After Sorting : array1 =
3   4   6   7   7   9
```

9. Search for Values in Arrays

Searching an array for a value is often needed. Let us consider the example of searching for your name among the reserved seats in a rail reservation chart, etc. Two methods are used in searching, They are :

1. Linear search
2. Binary search for sorted arrays.

1. Linear search

The method may be applied to any array.
The key value is compared to the value of the elements of the array successively.
If a match is found, it is noted, and the program ends there.
Otherwise, the complete array is searched, and if no match is found, it is reported that the value is not there in the array.

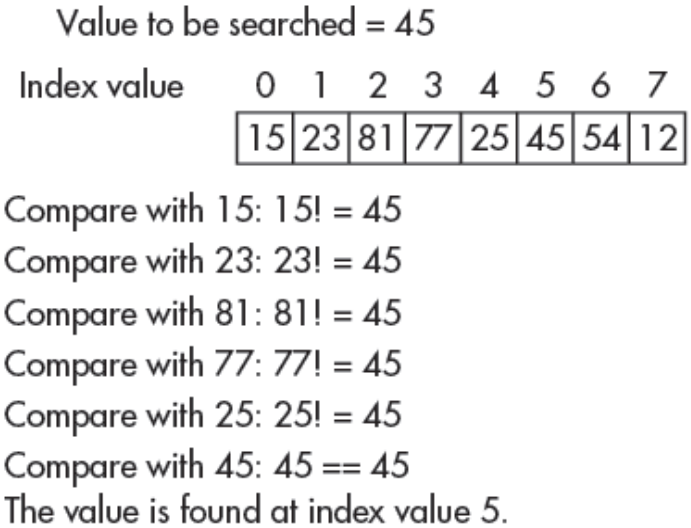


Fig. 7.9 Illustration of linear search

Example:

```
import java.util.Scanner;
class LinearSearch
{
  public static void main(String args[])
  {
   boolean b =false;
   int[] array = {67,78,85,44,25,65,36};
   Scanner in = new Scanner(System.in);
   System.out.println("Enter the number which you want to
search");
   int key = in.nextInt();

   for(int i=0; i<array.length; i++)
   {
            if(array[i] == key)
      {
         System.out.println("Your number is at index = " + i);
         b=true;
            }
   }
         if(b!=true)
    System.out.println("Your number is not in the array");
   }
}
```

Output:

```
C:\>javac LinearSearch.java

C:\>java LinearSearch
Enter the number which you want to search
44
Your number is at index = 3

C:\>java LinearSearch
Enter the number which you want to search
65
Your number is at index = 5

C:\>java LinearSearch
Enter the number which you want to search
888
Your number is not in the array
```

2. **Binary search for sorted arrays.**

- It is a very efficient method of search but it is applicable only to the sorted arrays.
- The beginning, end, and the midpoint of the array are defined first.
- The key value is compared with the value at midpoint.
- If it does not match, then it is checked in which half of the array the key value lies by checking whether the key value is more or less than the value at midpoint.
- The array is truncated to the half in which the key value lies.
- This process is repeated on that half, that is, it is again divided into two halves where the value is compared with the midpoint;
- if match is not found, it is determined in which half of the truncated array the value lies.
- The process is very useful for searching large sorted arrays

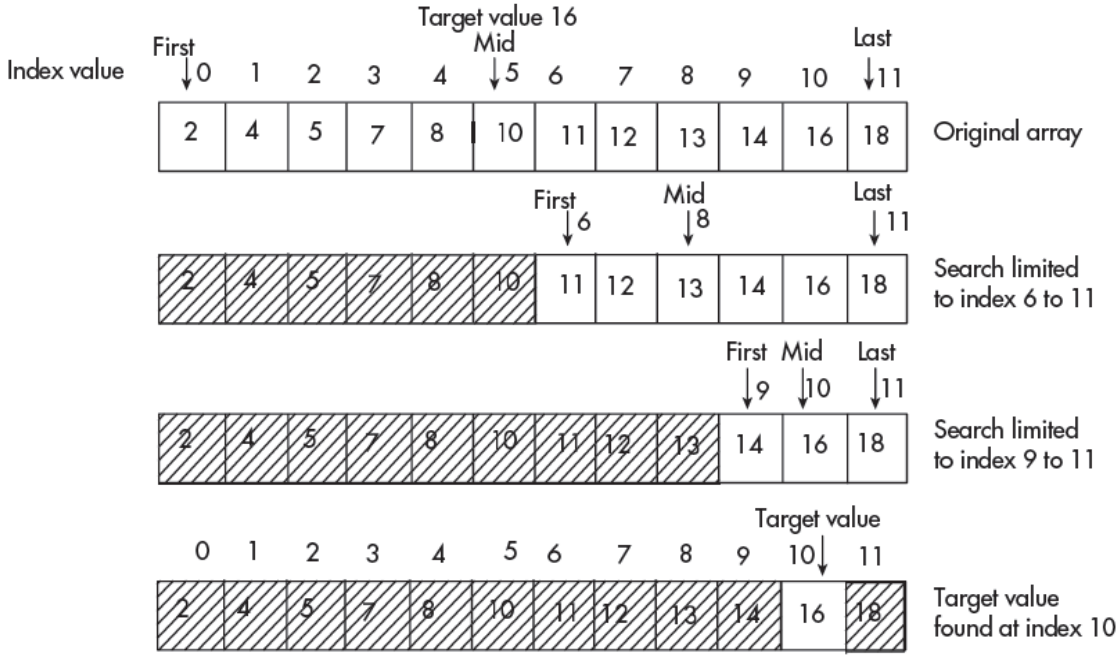


Fig. 7.10 Illustration of binary search

Example:

```
import java.util.Scanner;
class BinarySearch
{
  public static void main(String args[])
  {
      boolean b =false;
      int[] array = {15, 20, 43, 45, 76, 80, 86, 88, 90, 94, 96,
98};
      int length= array.length;
      Scanner in = new Scanner(System.in);
      System.out.println("Enter the number which you want to
search");
      int key = in.nextInt();
```

```
      int beginning,end,mid;
      beginning =0;
      end = length-1;
      while( beginning <=end)
      {
        if(beginning == end && array[end]!=key)
        {
          System.out.println("Your number is not in the array");
          break;
        }
        mid = (beginning +end )/2;
        if(array[mid]==key)
        {
          System.out.println("Your number is in the array at index =
" + mid);
          break;
        }
        else if(array[mid]<key)
          beginning = mid+1;
        else
          end = mid-1;
      }
   }
}
```

Output:

```
C:\>javac BinarySearch.java

C:\>java BinarySearch
Enter the number which you want to search
45
Your number is in the array at index = 3

C:\>java BinarySearch
Enter the number which you want to search
96
Your number is in the array at index = 10

C:\>java BinarySearch
Enter the number which you want to search
783
Your number is not in the array
```

10. Class Arrays

- The package java.util defines the class Arrays with static methods. –
- for general processes that are carried out on arrays such as
  - sorting an array for full length of the array or for part of an array,
  - binary search of an array for the full array or part of an array,
- for comparing two arrays if they are equal or not.
- for filling a part or the full array with elements having a specified value.
- for copying an array to another array.
- The sort method of Arrays class is based on quicksort technique.
- The methods are applicable to all primitive types as well as to class objects.

The methods of class Arrays are as follows:

Sort

- The class defines several overloaded methods for sorting arrays of different types.
- As per Java SE7, the sort method for int array is

1. public static void sort (int[] array)

Example: PredefinedMethodSort.java

```
import java.util.Arrays;
class PredefinedMethodSort
{
     public static void main(String args[])
     {
      int[] array1 = new int []{41,4,31,14,5};

      System.out.println("Array-1 elements are :");
         display(array1);
         System.out.println();

      //Predefined Method Sort of Arrays class
      Arrays.sort(array1);

      System.out.println("After Sort, Array-1 elements are :");
         display(array1);
         System.out.println();
       }

     static void display(int[] array)  //Method definition
     {
       for (int x : array)
           System.out.print(x + "   ");
     }
```

```
}

Output:

C:\>javac PredefinedMethodSort.java

C:\ >java PredefinedMethodSort
Array-1 elements are :
41   4   31   14   5
After Sort, Array-1 elements are :
4   5   14   31   41
```

Searching

There are two versions of overloaded binary Search method that are defined in class Arrays.

1. public static int binarySearch(int [] array, int key)

Equals

public static boolean equals (int [] a, int [] b)

The output is a Boolean value—it returns true, if the elements and their order in the two arrays are same; otherwise, it returns false.

Fill

The two versions of method fill defined in class Arrays are as follows.

1. public static void fill (byte [] array, byte value)
   The method fills the entire array with a specified value.
2. public static void fill(byte [] array, int startIndex, int endIndex, byte value)

The method fills the specified subset of an array with the specified value

CopyOf

This method was added in Java SE 6.

1. public static byte [] copyOf(byte [] original, int length)
   The method copies the array into a new array of specified length
2. copyOfRange
   public static char [] copyOfRange(char [] original, int fromIndex, int toIndex)

asList

public static <T> List<T> asList(T… array)
The method returns a fixed-sized list backed by the array.

toString

The method header is given as public static String toString (int [] array)
The method returns a string representation of the array elements.

11. Two-dimensional Arrays

- An array may hold other arrays as its elements.
- If the elements of an array are one-dimensional arrays, the array becomes a two-dimensional array.
- A two-dimensional array is treated as an array of arrays, and each of these arrays may have a different number of elements.
- E.g. Matrices are two-dimensional arrays.
- List of telephone numbers is another such example.
- A two-dimensional array may be defined as:
- int telNumber [][] = new int [5][10];
- The array will contain 5 numbers each having 10 digits.
- A two-dimensional array may as well be defined and initialized as
- int arrayB [ ][ ] = {{11, 12, 13 }, {7, 6, 4}};
- The two-dimensional array may as well be declared as
- int arrayC [][] = new int[2][3]{{1,2,3}. {4.5.6},{7,8,9}}

Example: TwoDimArray.java

```
class TwoDimArray
{
    public static void main(String[] args)
    {
      int num2D[][]= {{1,2,3},{4,5,6},{7,8,9}};

      for(int[] y : num2D)
      {
        for(int x : y)
            System.out.print( x + " ");
        System.out.println();
      }
    }
}
```

Output:

```
C:\ >javac TwoDimArray.java

C:\ >java TwoDimArray
1 2 3
4 5 6
7 8 9
```

12.Arrays of Varying Lengths

- A two-dimensional array is treated as an array whose elements are one-dimensional arrays, which may have different sizes.
- A two-dimensional array may be declared as
  `int a2D [][] = new int [3 ][];`
- The arrays may as well be declared as
  `int array [][] = {{5, 7, 8 },{10, 11 }, {4, 3, 2, 7,5 }};`

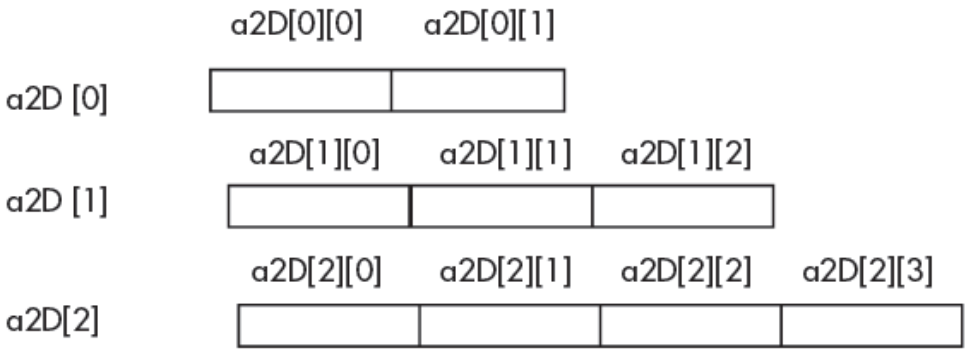


Fig. 7.11 Two-dimensional arrays of varying lengths

Example: TwoDimArray2.java

```
class TwoDimArray2
{
    public static void main(String[] args)
    {
     int num2D[][]= {{5,7,8},{10,11},{4,3,2,7,5}};

     for(int[] y : num2D)
     {
       for(int x : y)
          System.out.print( x + " ");
       System.out.println();
     }
    }
}
C:\>javac TwoDimArray2.java

C:\>java TwoDimArray2
5 7 8
10 11
4 3 2 7 5
```

13.Three-dimensional Arrays

- When an array holds two-dimensional arrays as its elements, the array is a three-dimensional array.
- A practical example includes an array of matrices.
- Each basic element of such an array needs three index values for its reference.
- A three-dimensional array may be declared as

```
int tDArray [][][]; //Declaration
double d3Array [][][]; //Declaration
```

Example: ThreeDimArray.java

```
class ThreeDimArray
{
    public static void main(String[] args)
    {
     int num3D[][][]= {  { {1,2,3},   {4,5,6},   {7,8,9}    },
                 { {11,12,13},{14,15,16},{17,18,19} }
                };

       for(int[][] z: num3D)
       {
      for(int[]y : z)
      {
       for(int x : y)
           System.out.print( x + " ");
       System.out.println();
      }
     System.out.println();
      }
    }
}
```

Output:

```
C:\1. JAVA\UNIT-3.1>javac ThreeDimArray.java
C:\1. JAVA\UNIT-3.1>java ThreeDimArray
1 2 3
4 5 6

7 8 9


11 12 13

14 15 16
```

17 18 19

14. Arrays as Vectors.

- Similar to Arrays, vectors are another kind of data structure that is used for storing information.
- Using vector, we can implement a dynamic array.
- The following are the vector constructors:
- Vector() creates a default vector having an initial size of 10.
- Vector(int size) creates a vector whose initial capacity is specified by size.
- Example
  `Vector vec = new Vector(5);` // declaring with initial size of 5
- Vector(int size, int incr) creates a vector with initial capacity specified by size and increment is specified by incr.
- The increment is the number of elements added in each reallocation cycle.

Advantages of Vectors.

Vectors have a number of advantages over arrays.

i. Vectors are dynamically allocated, and therefore, they provide efficient memory allocation.
ii. Size of the vector can be changed as and when required.
iii. They can store dynamic list of objects.
iv. The objects can be added or deleted from the list as per the requirement.

Table 7.1 Some of the important methods of Vector class

Method	*Description*
void add(int index, Object element)	Inserts the specified element at the specified position in the given vector
void addElement(Object obj)	Adds the specified component to the end of the given vector and increases its size by one
void clear()	Removes all the elements from the given vector
int capacity()	Returns the current capacity of the given vector
void copyInto(Object[] anArray)	Copies the components of the given vector into the specified array
Object firstElement()	Returns the first component (i.e., item at index 0) of the given vector
Object lastElement()	Returns the last component of the given vector
Enumeration elements()	Returns the enumeration of the components of the given vector
Object get(int index)	Returns the element at the specified position in the given vector
Object remove(int index)	Removes the element at the specified position in the given vector
int size()	Returns the number of elements currently in the vector

Example: VetorArray.java

```
import java.util.*;
class VectorArray
{
    public static void main(String[] args)
    {
     Scanner in = new Scanner(System.in);
     System.out.println("Enter the Vector capacity : ");
     int n = in.nextInt();

     //Declaring Vector with capacity n
     Vector<Integer> vec = new Vector<Integer>(n);

System.out.println("Intial size of Vector" + vec.size());
System.out.println("Intial capacity of Vector" + vec.capacity());
     //Adding the elements to the vector
     for(int i=0; i<n;i++)
        vec.add(i+10);

System.out.println("Current size of Vector" + vec.size());
System.out.println("Intial capacity of Vector" + vec.capacity());

     //Printing the vector elements
     System.out.println("Vector Elements are ");
     for (int i = 0; i < vec.size(); i++)
           System.out.print(vec.get(i) + " ");

     //removing element from Vector at index 3
     vec.remove(3);

     System.out.println("\n Vector Elements after removing element
at index 3");
     for (int i = 0; i < vec.size(); i++)
           System.out.print(vec.get(i) + " ");
    }
}
```

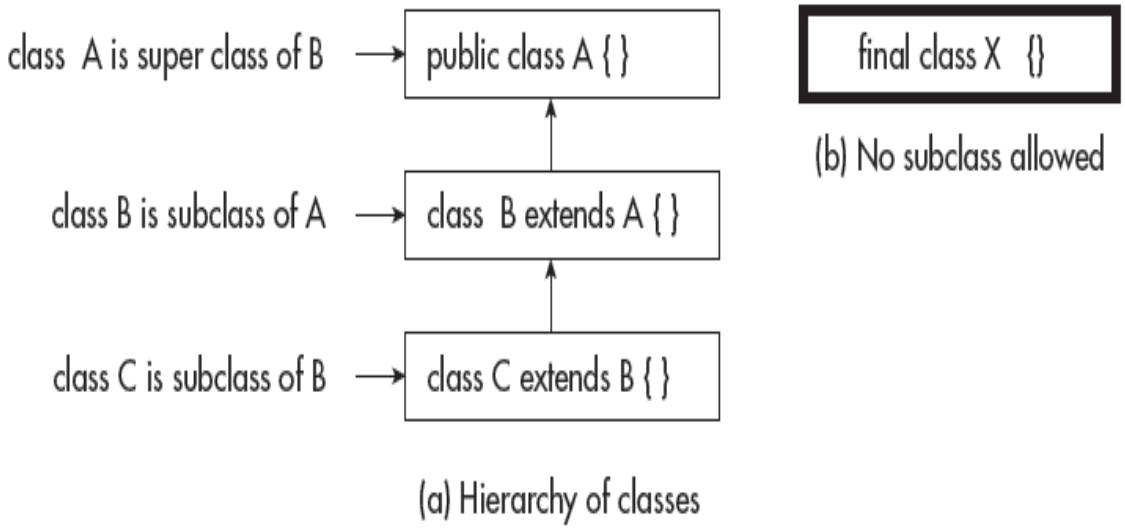
Output:

```
C:\>javac VectorArray.java

C:\>java VectorArray
Enter the Vector capacity :
7
Intial size of Vector0
Intial capacity of Vector7
Current size of Vector7
Intial capacity of Vector7
Vector Elements are
10 11 12 13 14 15 16
 Vector Elements after removing element at index 3
   10 1 12 14 15 16
```

II. Inheritance:

1. Introduction

- Inheritance is the backbone of object-oriented programming (OOP).
- It is the mechanism by which a class can acquire properties and methods of another class.
- Using inheritance, an already tested and debugged class program can be reused for some other application.
- **Super class** This is the existing class from which another class, that is, the subclass is generally derived.
- In Java, several derived classes can have the same super class.
- **Subclass** A class that is derived from another class is called subclass.
- In Java, a subclass can have only one super class.



Benefits of Inheritance

- It allows the reuse of already developed and debugged class program without any modification.
- It allows a number of subclasses to fulfil the needs of several subgroups.
- A large program may be divided into suitable classes and subclasses that may be developed by separate teams of programmers.

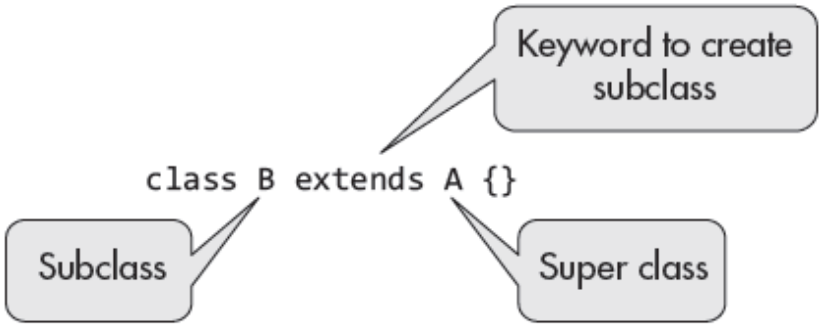
Disadvantages of Inheritance

1. The tight coupling between super and subclasses increases and it becomes **very difficult to use them independently**.
2. **Program processing time increases** as it takes more time for the control to jump through various levels of overloaded classes.
3. **When some new features are added** to super and derived classes as a part of maintenance, the changes affect both the classes.
4. **When some methods are deleted** in super class that is inherited by a subclass, the

methods of subclass will no longer override the super class method.

2. Process of Inheritance

- Inheritance means deriving some characteristics from something that is generic.
- In the context of Java, it implies deriving a new class from an existing old class, that is, the super class.
- A super class describes general characteristics of a class of objects.
- A subset of these objects may have characteristics different from others.
- There are two ways of dealing with this problem.
- Either, make a separate class for the subset to include all the characteristics or, to have another class that inherits the existing class, extend this class to include the special characteristics.



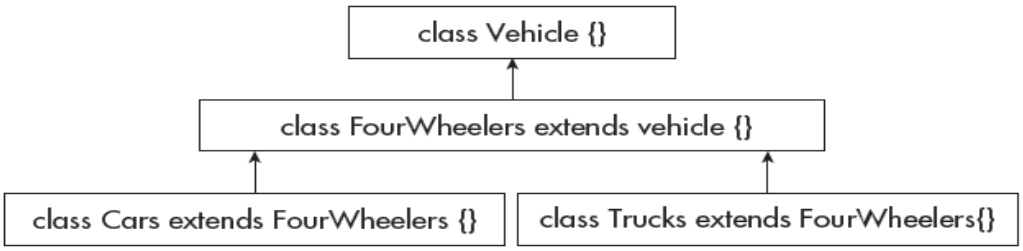


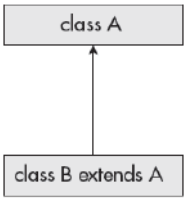
Fig. 8.4 Example for process of inheritance

3. Types of Inheritances

The following types of inheritances are supported by Java.

1. Single inheritance
2. Multilevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance using interfaces

i. **Single inheritance:** It is the simple type of inheritance. In this, a class extends another one class only.



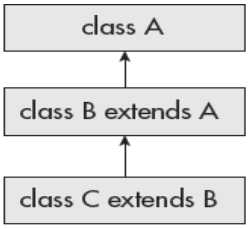
Example: SingleInheritance.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Super Class Method");
      }
}
class DemoB extends DemoA
{
      void displayB()
      {
            System.out.println("Sub Class Method");
      }
}
class SingleInheritance
{
     public static void main(String args[])
     {
      DemoA objA = new DemoA();
      objA.displayA();

      DemoB objB = new DemoB();
      objB.displayB();
       }
}
```

Output:

```
C:\>javac SingleInheritance.java
C:\>java SingleInheritance
Super Class Method
Sub Class Method
```

ii. **Multilevel inheritance:** In this type, a derived class inherits a parent or super class; The derived class also acts as the parent class to other class.

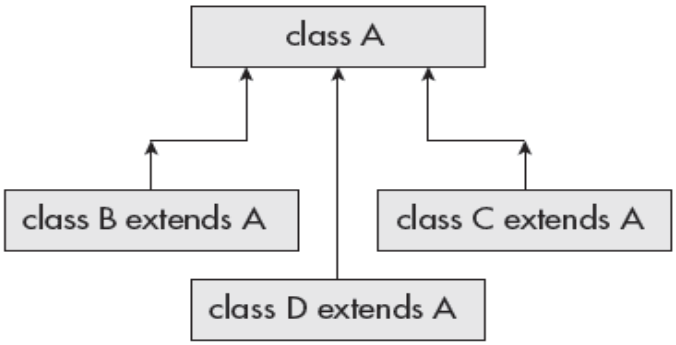


Example: Multilevel.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Class-A Method");
      }
}

class DemoB extends DemoA
{
      void displayB()
      {
            System.out.println("Class-B Method");
      }
}

class DemoC extends DemoB
{
      void displayC()
      {
            System.out.println("Class-C Method");
      }
}

class Multilevel
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA();
      objA.displayA();
```

```
      //calling class-B method
      DemoB objB = new DemoB();
      objB.displayB();

      //calling class-C method
      DemoC objC = new DemoC();
      objC.displayC();
       }
}
Output:
C:\>javac Multilevel.java

C:\>java Multilevel
Class-A Method
Class-B Method
Class-C Method
```

iii. **Hierarchical inheritance:** In this type, one class is inherited by many sub classes.



(c)

Example: Hierarchical.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Class-A Method");
      }
}

class DemoB extends DemoA
{
      void displayB()
      {
            System.out.println("Class-B Method");
```

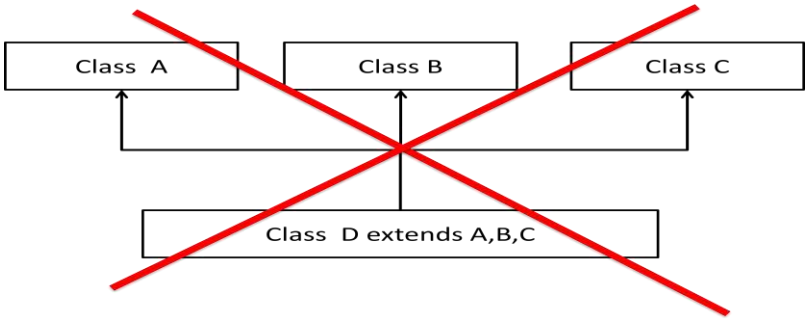
```
      }
}

class DemoC extends DemoA
{
      void displayC()
      {
            System.out.println("Class-C Method");
      }
}

class Heirarchical
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA();
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB();
      objB.displayB();

      //calling class-C method
      DemoC objC = new DemoC();
      objC.displayC();
       }
}
```

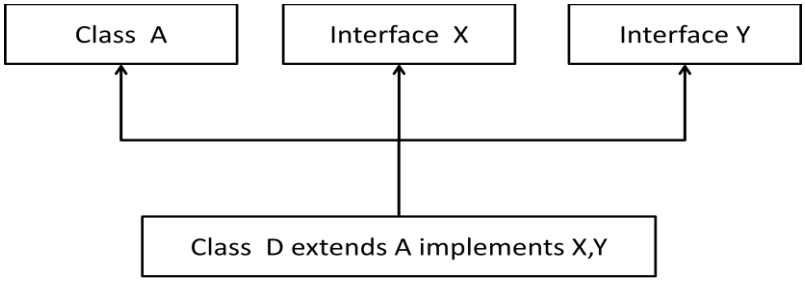
Output:

```
C:\ >javac Hierarchical.java

C:\ >java Hierarchical
Class-A Method
Class-B Method
Class-C Method
```

iv. Multiple inheritance: In this, a class is extending more than one class.

- Java does not support multiple inheritance.
- This implies that a class cannot extend more than one class.
- Suppose there is a method in class A. This method is overridden in class B and class C in their own way.
- Since class C extends both the classes A and B.
- So, if class C uses the same method, then there will be ambiguity as which method is called.





Example: Multiple.java

```
interface X
{
	int x=10;
}

interface Y
{
	int y=20;
}


class DemoA
{
	void displayA()
	{
		System.out.println("Class-A Method");
	}
}
```

```
class DemoB extends DemoA implements X,Y
{
      void displayB()
      {
            System.out.println("Class-B Method : x+y = " +
(x+y));
      }
}


class Multiple
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA();
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB();
      objB.displayB();

       }
}
```

Output:

```
C:\ >javac Multilevel.java

C:\ >java Multilevel
Class-A Method
Class-B Method
Class-C Method
```

4. Universal Super Class : " Object " Class

- **Object** class is a special class and it is at the top of the class hierarchy tree.
- It is the parent class or super class of all in Java.
- Hence, it is called Universal super class.
- Object is at the root of the tree and every other class can be directly or indirectly derived from the Object class.

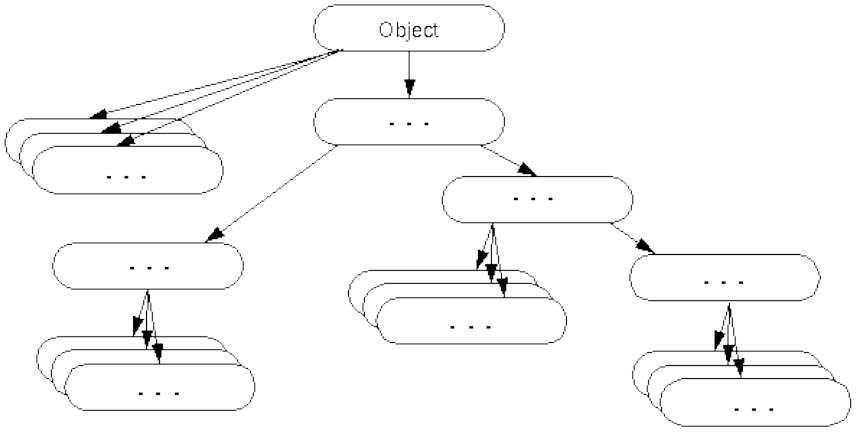


Table 8.2 Methods of object class

Method	*Description*
public String toString()	Returns the string representation of this object.
public int hashCode()	Returns the hashcode number for the given object.
public boolean equals(Object obj)	Compares the given object to this object.
Public final Class getClass()	Returns the Class class object of this object. The Class can be used to get the metadata of this class.

Example:ObjectEquals.java

```
class DemoA
{
      void displayA()
      {
            System.out.println("Class-A Method");
      }
}

class ObjectEquals
{
     public static void main(String args[])
     {
      DemoA obj1 = new DemoA();
      DemoA obj2 = new DemoA();
      boolean test;
```

```
      //Checking - if both object are equal
      test = obj1.equals(obj2);
      display(test);

      // Object assignment
      obj1=obj2;

      //Checking - if both object are equal after assigning the
objects
      test = obj1.equals(obj2);
      display(test);
      }

     public static void display(boolean test)
      {
      if (test)
        System.out.println("Both objects are same");
      else
        System.out.println("Both objects are different");
      }
}
```

Output:

```
C:\>javac ObjectEquals.java

C:\>java ObjectEquals
Both objects are different
Both objects are same
```

5. Inhibiting Inheritance of Class Using Final

- **A class declared as final cannot be inherited further**.
- Class variables or instance variables are declared as constant to make local variables.
- When a class is inherited by other classes, its methods can be overridden.
- In order to prevent the methods from being overridden, that method can be declared as final.

Example:FinalClass.java

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

class B extends A
{
      int b;
        B(int x,int y)
      {
          super(x);
          this.b=y;
      }
      void display()
      {
        System.out.println("b = "+ b);
      }
}

class FinalClass
{
      public static void main (String args[])
      {
            A objA= new A(10);
            B objB= new B(100,200);

            objA.display();
            objB.display();
      }
}
```

Output:

```
C:\>javac FinalClass.java
FinalClass.java:11: error: cannot inherit from final A
class B extends A
                ^
1 error
```

6. Access Control and Inheritance

- A derived class access to the members of a super class may be modified by access specifiers.
- There are three access specifiers, that is, public, protected, and private.
- The code for specifying access is Access-specifier type member_identifier;

Table 8.3 Access specifiers

Access specifiers	*Access*
No access specifier	Access permitted to any other class belonging to the same package
public	Access permitted to any class in any package
protected	Access permitted to any subclass in any package, also to any class in the same package
private	Access permitted only to members of the same class. No outside code has access to a private member of a class

Table 8.4 Access control

Class	*Access permitted (Yes/No)*			
	No specifier	*Public*	*Protected*	*Private*
Same class members	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Any other class in same package	Yes	Yes	Yes	No
Subclass in different package	No	Yes	Yes	No
Any other class in different package	No	Yes	No	No

Example-1: DefaultAccess.java

```
class DemoA
{
   int a;
   void displayA()
   {
         System.out.println("Class-A Method : a = " + a);
   }
}

class DemoB extends DemoA
{
   int b;
   void displayB()
   {
System.out.println("Class-B Method : a = " + a + "  b = " + b );
   }
}


class DefaultAccess
{
     public static void main(String args[])
     {
   //calling class-A method
   DemoA objA = new DemoA();
   objA.a=100;  // accessing all classes in the same package
   objA.displayA();

   //calling class-B method
   DemoB objB = new DemoB();
   objB.a=200; // objA and objB are different objects. a is
assigned with 200
   objB.b=300; // accessing all classes in the same package
   objB.displayB();

      }
}
```

Output:

```
C:\>javac DefaultAccess.java

C:\>java DefaultAccess
Class-A Method : a = 100
Class-B Method : a = 200  b = 300
```

Example -2: PrivateAccess.java

```
class DemoA
{
      private int a;
      DemoA(int x)
      {
        a = x;
      }
      void displayA()
      {
            System.out.println("Class-A Method : a = " + a);
      }
}

class DemoB extends DemoA
{
      int b;
      DemoB(int p, int q)
      {
        super(p);
        b=q;
      }
      void displayB()
      {
            displayA();
            System.out.println("Class-B Method : b = " + b );
      }
}

class PrivateAccess
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA(150);
      //objA.a=100;  // Error, Can't access private variable
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB(500,1000);
      // objB.a=200; // objA and objB are different objects. Error,
Can't access private variable
      //objB.b=300; // accessing all classes in the same package
      objB.displayB();

       }
}
```

Output:

```
C:\>javac PrivateAccess.java

C:\>java PrivateAccess
Class-A Method : a = 150
Class-A Method : a = 500
Class-B Method : b = 1000
```

Example: ProtectedAccess.java

```
class DemoA
{
      protected int a;
      DemoA(int t)
      {
        a = t;
      }
      void displayA()
      {
            System.out.println("Class-A Method : a = " + a);
      }
}

class DemoB extends DemoA
{
      int b;
      DemoB(int p, int q)
      {
        super(p);
        b=q;
      }
      void displayB()
      {
      System.out.println("Class-B Method : a= " + a + " b = " + b );
      }
}

class DemoC extends DemoB
{
      int c;
      DemoC(int x, int y, int z)
      {
        super(x,y);
        c=z;
      }
      void displayC()
      {
      System.out.println("Class-B Method : a = " + a + " b = " + b +
" c = " + c);
      }
```

```
}

class ProtectedAccess
{
     public static void main(String args[])
     {
      //calling class-A method
      DemoA objA = new DemoA(100);
      //objA.a=100;  // Error, Can't access protected variable
      objA.displayA();

      //calling class-B method
      DemoB objB = new DemoB(200,300);
      objB.displayB();

      //calling class-C method
      DemoC objC = new DemoC(250,500,750);
      objC.displayC();

       }
}
```

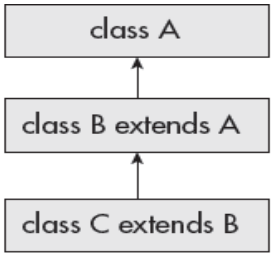
Output:

```
C:\>javac ProtectedAccess.java

C:\>java ProtectedAccess
Class-A Method : a = 100
Class-B Method : a= 200 b = 300
Class-B Method : a = 250 b = 500 c = 750
```

7. Multilevel Inheritance

In this type, a derived class inherits a parent or super class;

- The derived class also acts as the parent class to other class.



See Example : Multilevel.java

8. Application of Keyword Super

The keyword super is used for two purposes:

First, to distinguish between the variables having the same name in super class and subclass.

- When the member is called with an object of subclass, the subclass value will be presented and super class value will get hidden.
- For getting super class value, the keyword super is used.

Second, it is used in defining the constructor of subclass.

- Instead of repeating the assignment of variables of super class, we simply qualify the variable with super.

Example: SuperDemo.java

```
class A
{
      int a;
      A(int x)
      {
        a = x;
      }
      void displayA()
      {
            System.out.println("Class-A Method : a = " + a);
      }
}
class B extends A
{
      int b;
      B(int p, int q)
      {
        super(p); // Calling Super class Construtor
        b=q;
      }
      void displayB()
      {
  // Refering Super class with super keyword
  System.out.println("Class-B Method : a = " + super.a + " b = " + b );
      }
}
class SuperDemo
{
     public static void main(String args[])
     {
      //Creating class B object by calling sub class constructor
      B objB = new B(500,1000);

      //calling class-B method
      objB.displayB();

      }
}
```

Output:

```
C:\>javac SuperDemo.java
C:\>java SuperDemo
```

```
Class-B Method : a = 500 b = 1000
```

9. Constructor Method and Inheritance

- For getting super class value, the keyword super is used.
- Second, it is used in defining the constructor of subclass.
- Instead of repeating the assignment of variables of super class, we simply qualify the variable with super.
  - Example: SuperDemo.java

10.Method Overriding

- It is one of the ways in which polymorphism can be implemented.
- When both super class and its subclass contain a method that has the same name and type signature, the super class definition of the method is overridden by definitions in subclass.
- It is different from the overloaded method in which only the name is same but parameter list has to be different either in type or in number of parameters or order of parameters.
- In the case of overloaded methods, the parameter lists are matched to choose the appropriate method that may be in super class or subclass.
- When two methods with the same name and type signature are defined in super (base) class as well as in subclass (derived class), the subclass definition overrides the super class definition when the method is called by object of subclass;
- It will execute the method defined in subclass and hide the definition of super class.

Binding

- It involves associating the method call to method body. There are two types of binding as follows:

Static binding : When the binding is performed at compile time by the compiler, it is known as static or early binding.

- For instance, binding for all static, private, and final methods is done at the compile time.

Dynamic binding : It is also called late binding. Here, the compiler is not able to resolve the call (or binding) at compile time.

- Method overriding is one such example where dynamic binding is involved.
- The basic difference between static and dynamic binding is that static binding occurs at compile time, whereas dynamic binding happens at run time.

Example: MethodOverriding.java

```
class A
{
  void display()
  {
        System.out.println("Super Class Method");
  }
}

class B extends A
```

```
{
  void display()
  {
        System.out.println("Sub Class Method");
  }
}

class MethodOverriding
{
     public static void main(String args[])
     {
  //calling the class A method
  A objA = new A();
  objA.display();

  //calling the class B method
  objA = new B();
  objA.display();
      }
}
```

Output:

```
C:\>javac MethodOverriding.java

C:\>java MethodOverriding
Super Class Method
Sub Class Method
```

11.Dynamic Method Dispatch

- It is a mechanism by which runtime polymorphism is achieved for overridden method in Java.
- It is implemented through super class reference. A super class reference can refer to an object of its subclass.
- A base class pointer can refer to derived class object. There may be many subclasses inherited from a super class.
- Each subclass has its own version or definition of the overridden method.
- The dynamic method dispatch chooses the right version of the method corresponding to the object reference.
- In the method, first a reference variable of super class is created.
- The value of subclass object is assigned to the variable and the overridden method is called by the super class reference.

Example: MethodOverriding.java

12.Abstract Classes

The **abstract** keyword is a non-access modifier, used for classes and methods:

- **Abstract class:** is a restricted class that cannot be used to create objects (to access it, it must be inherited from another class).
- **Abstract method:** can only be used in an abstract class, and it does not have a body. The body is provided by the subclass (inherited from).

Example: Abstract.java

```
abstract class A
{
      int a;
      void setValue(int x)
      {
         a=x;
      }
      abstract void display();
}

class B extends A
{
      int b;
      void setValues(int x, int y)
      {
         a=x;
         b=y;
      }
      void display()
      {
      System.out.println("Class-B Method : a = " + a + " b = " + b);
      }
}

class C extends A
{
      int c;
      void setValues(int x, int y)
      {
         a=x;
         c=y;
      }
      void display()
      {
      System.out.println("Class-B Method : a = " + a + " c = " + c);
      }
}
```

```
class Abstract
{
     public static void main(String args[])
     {
      //calling class-B method
      System.out.println("Through objB");
      B objB = new B();
      objB.setValues(10,20);
      objB.display();

      //calling class-C method
      System.out.println("Through objC");
      C objC = new C();
      objC.setValues(150,250);
      objC.display();
       }
}
```

13.Interfaces and Inheritance.

- Multiple inheritance of classes is not permitted in Java.
- To some extent, this restriction can be overcome through interfaces.
- A class may implement more than one interface besides having one super class.
- An interface can extend one or more interfaces, and a class can also implement more than one interface.
- An interface is a collection of constants and abstract methods that are implemented by a class.
- An interface cannot implement itself like a class;
- An interface just contains the method head, and there is no method body. The class that implements the interface contains the full definition of the method.

Example: Multiple.java

III. Interfaces:

1. Introduction

- https://www.w3schools.com/java/java_interface.asp
  An interface is a completely "abstract class" that is used to group related methods with empty bodies
- https://www.javatpoint.com/interface-in-java
  An interface in Java is a blueprint of a class. It has static constants and abstract methods.
- https://www.geeksforgeeks.org/interfaces-in-java/
  Like a class, an interface can have methods and variables, but the methods declared in an interface are by default abstract (only method signature, no body).
- An interface also introduces a **new reference type**.
- An interface represents an **encapsulation of constants, classes, interfaces,** and one or more abstract methods that are implemented by a class.
- An interface **does not contain instance variables**.
- **An interface cannot implement itself**; it has to be implemented by a class.
- **The methods in an interface have no body**.
- Only headers are declared with the parameter list that is followed by a semicolon.
- The **class that implements the interface has to have full definitions of all the abstract methods** in the interface.
- **An interface can be implemented by any number of classes** with their own definitions of the methods of the interface.
- **Different classes can have different definitions of the same methods but the parameter list must be identical to that in the interface**.
- Thus, **interfaces provide another way of dynamic polymorphic implementation of methods**.
- **Any number of interfaces can be implemented by a class.**
- This fulfils the need for multiple inheritance.
- **The multiple inheritances of classes are not allowed in Java**, and therefore, interfaces provide a stopgap arrangement.

Similarities between Interface and Class

- Declaring an interface is similar to that of class; the keyword *class* is replaced by keyword *interface*.
- Its accessibility can be controlled just like a class.
- An interface declared public is accessible to any class in any package, whereas the ones without an access specifier is accessible to classes in the same package only.
- One can create variables as object references of interface that can use the interface.
- It can contain inner classes (nested classes) and inner interfaces.
- Since Java 8, an interface can have full definitions of methods with default or static modifiers.

Types of Interfaces

Top level interfaces

- It is an interface that is not nested in any class or interface.
- It comprises a collection of abstract methods.
- It can contain any number of methods that are needed to be defined in the class.

Nested interface

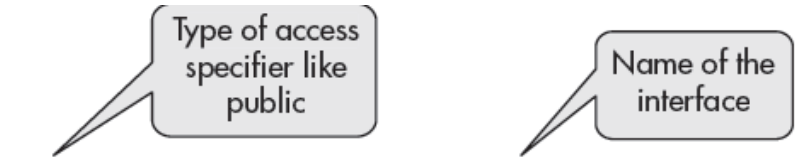
- It is an interface that is defi ned in the body of a class or interface.
- In nested interfaces, one or more interfaces are grouped, so that it becomes easy to maintain.
- It is referred to by the outer interface or class and cannot be accessed directly.

Generic interface

- Like a class, an interface is generic if it declares one or more types of variables.
- It comprises methods that accept or return an object.
- Thus, we can pass any parameter to the method that is not of the primitive type.

2. Declaration of Interface

- Declaration of an interface starts with the access modifier followed by keyword interface.
- It is in then followed by its name or identifier that is followed by a block of statements;
- These statements contain declarations of variables and abstract methods.
- The variables defined in interfaces are implicitly public, static, and final.
- They are initialized at the time of declaration. The methods declared in an interface are public by default.

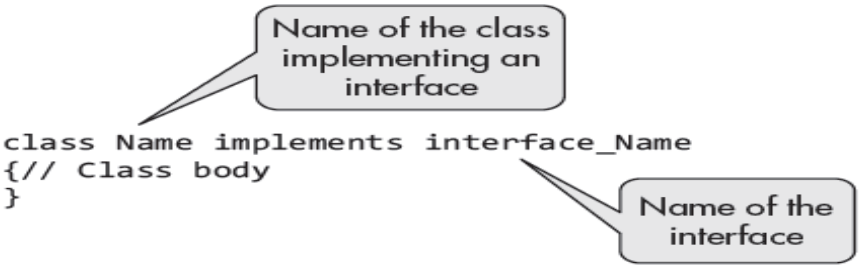


Members of Interface

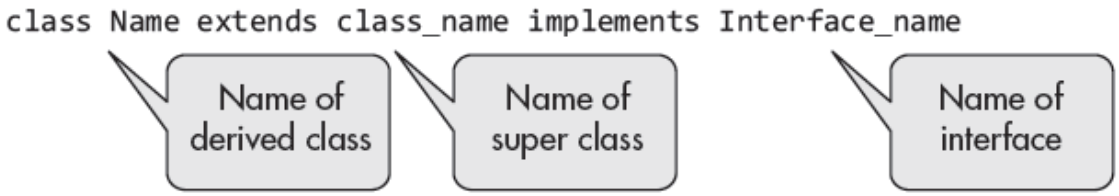
- The members declared in the body of the interface.
- The members inherited from any super interface that it extends.
- The methods declared in the interface are implicitly public abstract member methods.
- The field variables defined in interfaces are implicitly public, static, and final.
- However, the specification of these modifiers does not create a compile-type error.
- The field variables declared in an interface must be initialized; otherwise, compile-type error occurs.
- Since Java SE8, static and default methods with full definition can also be members of interface.

3. Implementation of Interface

Declaration of class that implements an interface

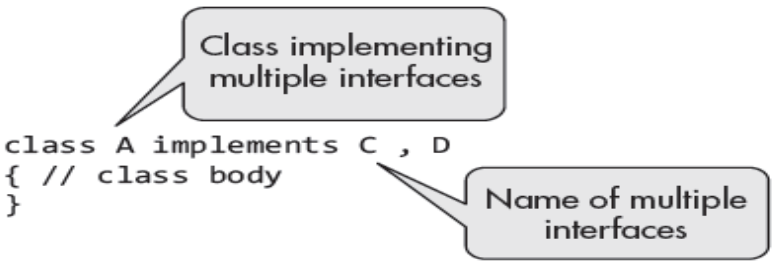


If a class extends another class as well as implements interfaces, it is declared as



4. Multiple Interfaces

- Multiple interfaces can also be implemented in Java.
- For this, the class implements all the methods declared in all the interfaces.
- When the class is declared, names of all interfaces are listed after the keyword *implements* and separated by comma.
- As for example, if class A implements interfaces C and D, it is defined as



Example: Multiple.java

Interface References:

- For interface references, variables can be declared as object references.
- In this case, the object reference would use interface as the type instead of class.
- The appropriate method is called on the basis of actual instance of the interface that is being referred to.

Example: InterfaceRef.java

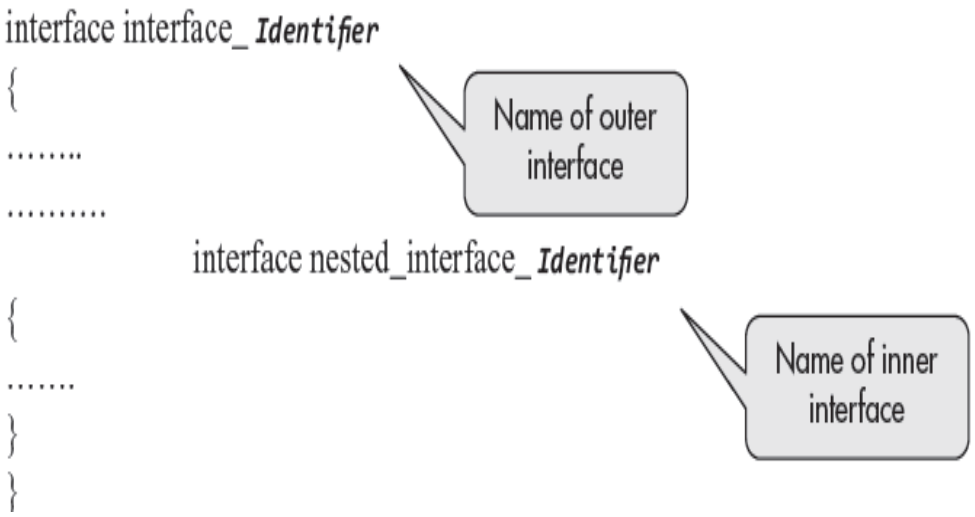
```
interface X
{
      int x = 10;
      public void display();
}

interface Y
{
      int y = 20;
      public void add();
}
class A implements X,Y
{
      public void display()
      {
      System.out.println("Class-B Method : x = "+ x + " y = " + y);
      }
      public void add()
      {
      System.out.println("Class-B Method : x+y = "+ (x+y));
      }
}

class InterfaceRef
{
     public static void main(String args[])
     {
      //Reference of X
      X objX = new A();
      objX.display();

      //Reference of Y
      Y objY = new A();
      objY.add();
       }
}
```

Output:

```
C:\ >javac InterfaceRef.java

C:\ >java InterfaceRef
Class-B Method : x = 10 y = 20
Class-B Method : x+y = 30
```

5. Nested Interfaces

- An interface may be declared as a member of a class or in another
- interface.
- In the capacity of a class member, it can have the attributes that are applicable to other class members.
- In other cases, an interface can only be declared as public or with default (no-access modifier) access.
- Syntax of nested interface in another interface is given as



Example: NestedInterface.java

```
interface OuterX
{
   int x = 10;
   public interface InnerY
   {
         int y = 20;
   }
}

class A implements OuterX, OuterX.InnerY
{
   void display()
   {
         System.out.println("Class-A Method : x = "+ x + " y
= " + y);
   }
}

class NestedInterface
{
     public static void main(String args[])
     {
   //calling class-A method
   A objA = new A();
   objA.display();
      }
}
```

Output:

```
C:\ >javac NestedInterface.java

C:\ >java NestedInterface
Class-A Method : x = 10 y = 20
```

6. Inheritance of Interfaces

Inheritance of Interfaces is similar to the Inheritance of classes. Interface can be derived from another interface.

```
Syntax:

access_specifier interface  NewInterface  extends OldInterface
{
  //Body of the interface
}

Example:

interface A{}
interface B{}
interface C extends A,B
{
//Body of the interface
}
```

Example Program: InterfaceInheritance.java

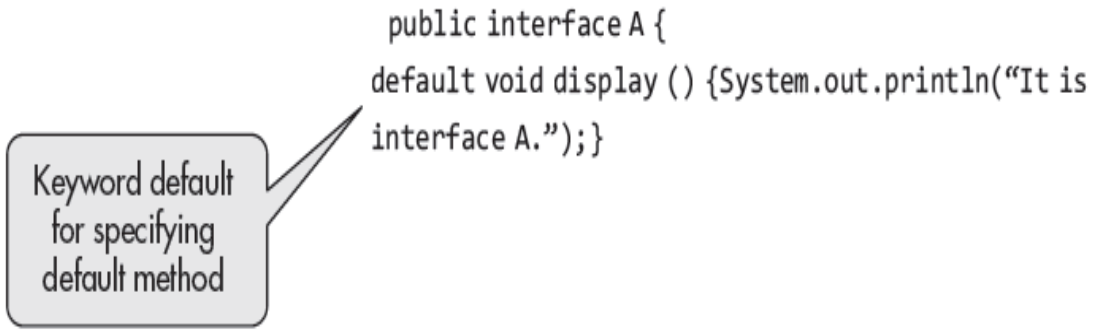
```
interface X
{
   int x = 10;
}

interface Y extends X
{
   int y = 20;
}

class A implements Y
{
   void display()
   {
System.out.println("Class-A Method : x = "+ x + " y = " + y);
   }
}
class InterfaceInheritance
{
     public static void main(String args[])
     {
       //calling class-A method
       A objA = new A();
       objA.display();
     }
```

```
}
Output:
C:\1. JAVA\UNIT-3.3>javac InterfaceInheritance.java

C:\1. JAVA\UNIT-3.3>java InterfaceInheritance
Class-A Method : x = 10 y = 20
```

7. Default Methods in Interfaces

- The enhancement in Java 8 allowing the interface to have full definition of default methods and static methods that are implicitly inherited by the class implementing the interface.
- Java allows only one super class, by using default methods in interface java allows that interface behaves just like an abstract super-class.
- New functionality can be added to the interface, which is inherited by classes implementing the interface.
- The inherited methods are also members of the class, and therefore, these maybe called other methods of class.
- A default method cannot be declared final.
- A default method cannot be synchronized; however, blocks of statements in the default method may be synchronized.
- The object class is inherited by all classes. Therefore, a default method should not override any non- final method of object class.
-

A default method is declared with keyword *default* as



Example : DefaultMethods.java

```
interface X
{
   int x = 10;
   default void display()
   {
      System.out.println("Method in Interface X the value x
= " + x);
   }
}

class A implements X
{

}


class DefaultMethods
{
     public static void main(String args[])
     {
   //calling class-A method
   A objA = new A();
   objA.display();
      }
}
```

Output:

```
C:\>javac DefaultMethods.java

C:\>java DefaultMethods
Method in Interface X the value x = 10
```

8. Static Methods in Interface

- The Java version 8 allows full definition of static methods in interfaces.
- A static method is a class method.
- For calling a static method, one does not need an object of class.
- It can simply be called with class name as

```
class_name.method_name()
```

Example: StaticMethods.java

```
interface X
{
   int x = 10;
   static void display()
   {
      System.out.println("Method in Interface X the value x
= " + x);
   }
}

class StaticMethods
{
     public static void main(String args[])
     {
   //calling static method by specifying interface X
   X.display();
      }
}
Output:
C:\>javac StaticMethods.java

C:\>java StaticMethods
Method in Interface X the value x = 10
```

Additional Information

Java Lambda Expressions

- Lambda Expressions were added in Java 8.
- A lambda expression is a **short block of code** which takes in parameters and returns a value.
- Lambda expressions are **similar to methods**, but they **do not need a name** and **they can be implemented right in the body of a method**.

Syntax
The simplest lambda expression contains a single parameter and an expression:

```
Parameter -> expression
```

To use more than one parameter, wrap them in parentheses:

```
(Parameter1, Parameter2) -> { Code of Block }
```

Example1:

```
import java.util.ArrayList;

public class LambdaExpressions
{
      public static void main(String[] args)
      {
      ArrayList<Integer> numbers = new ArrayList<Integer>();
      numbers.add(5);
      numbers.add(9);
      numbers.add(8);
      numbers.add(1);
      numbers.forEach((n) -> { System.out.println(n); });
      }
}
```

```
C:\>javac LambdaExpressions.java

C:\>java LambdaExpressions
5
9
8
1

C:\1. JAVA\UNIT-3.3>
```

9. Functional Interfaces

- In Java SE8, a new package java.util.function on functional interfaces has been introduced for writing Lambda functions.
- Functional interfaces are interfaces with one abstract method.
- They are also called SAM or single abstract method type.
- However, a functional interface can have more than one static and default methods.
- The programmer may include an annotation, to lessen the work of complier.
  @FunctionalInterface
- By adding the above annotation, it can be helpful in detecting compile time errors.
- If the functional interface contains more than one abstract method, the compiler will throw an error.

Example:Funcanal.java

```
import java.util.function.Function;
import java.util.function.BinaryOperator;

class Functional
{
     public static void main(String args[])
     {
   //declaring logrithm and minimum as functional interfaces
   Function <Double, Double> logrithm = Math::log;
   BinaryOperator<Integer> minimum = Math::min;

   //calling logrithm.apply() which is method reference to
the Function
   System.out.println("Log of 10 to the base e = "+
logrithm.apply(10.0));

   //calling minimum.apply() which is method reference to
the BinaryOperator
   System.out.println("Minimum of 20 and 46 is    "+
minimum.apply(20,46));
     }
}
```

Output:

```
C:\ >javac Functional.java

C:\ >java Functional
Log of 10 to the base e = 2.302585092994046
Minimum of 20 and 46 is  20
```

Functional Consumer<T>

- The interface declaration is

  @FunctionalInterface
  public interface Consumer {void accept(T t);}
- It declares one abstract method void accept(T t).
- The method only consumes its argument. It does not give any return value.

Example: ConsumerDemo.java

```
import java.util.function.Consumer;

class ConsumerDemo
{
      public static void main(String args[])
      {
      //declaring logrithm and minimum as functional interfaces
      System.out.print("double value =  ") ;
      Consumer<Double> FunInt1 = (Double d) -> { display(d); };
      FunInt1.accept(3.14);

      System.out.print("\nString Array =  ") ;
      Consumer<String> FunInt2 = (String s) -> { display(s); };
      String[] sray = {"CSE","ECE","CIVIL"};
      for(String str: sray)
        FunInt2.accept(str);

      System.out.print("\nInteger Array =  ") ;
      Consumer<Integer> FunInt3 = (Integer n) -> { display(n); };
      Integer[] iray = {1,2,3,4,5};
      for(Integer num: iray)
        FunInt3.accept(num);
      }
      public static<T> void display(T t)
      {
            System.out.print(t +" ");
      }
}
```

Output:

```
C:\ >javac ConsumerDemo.java

C:\ >java ConsumerDemo
double value =  3.14
String Array =  CSE ECE CIVIL
Integer Array =  1 2 3 4 5
```

10.Annotations.

- Annotation framework in Java language was first introduced in Java 5 through a provisional interface Apt; It is **a type of metadata** that can be **integrated with the source code without affecting the running of the program**.
- Annotations may be retained up to runtime and may be used to instruct the compiler and runtime system to do or not to do certain things.
- Since Java SE 8, the annotations may be applied to classes, fields, interfaces, methods, and type declarations like throw clauses.
- The annotations are no longer simply for metadata inclusion in the program but have become a method for user's communication with compiler or runtime system.
- An annotation like @Override consists of two distinct words @ and Override.
- It may as well be written as @ Override;
- The name Override is the name of the interface that defines the annotation.
- There are a number of annotations that are predefined and are part of the package java.lang.annotation.
- However, a programmer may also defi ne an annotation.

Example: Annotational.java

```
class A
{
 public void display()
 {
      System.out.println("In class A");
 }
}

class B extends A
{
 @Override public void display()
 {
      System.out.println("In class B");
 }
}

class C extends A
{
 @Override public void display()
 {
      System.out.println("In class C");
 }
}
class Annotational
{
      public static void main(String args[])
      {
      A objA = new A();
      B objB = new B();
      C objC = new C();
```

```
      objA.display();
      objB.display();
      objC.display();
 }
}
```

Output:

```
C:\1. JAVA\UNIT-3.3>javac Annotational.java

C:\1. JAVA\UNIT-3.3>java Annotational
In class A
In class B
In class C
```

If we change the method name in class C as displayC( ), Error will generate as

```
C:\1. JAVA\UNIT-3.3>javac Annotational.java
Annotational.java:19: error: method does not override or implement a
method from a supertype
        @Override public void displayC()
        ^
1 error
```

UNIT-4

I. Packages and Java Library:

1. Introduction
2. Defining Package
3. Importing Packages and Classes into Programs
4. Path and Class Path
5. Access Control
6. Packages in Java SE
7. Java.lang Package and its Classes
8. Class Object
9. Enumeration
10. class Math
11. Wrapper Classes
12. Auto-boxing and Auto- unboxing
13. **java.util** Classes and Interfaces
14. Formatter Class
15. Random Class Time Package
16. Class Instant (java.time.Instant)
17. Formatting for Date/Time in Java
18. Temporal Adjusters Class
19. Temporal Adjusters Class.

II. Exception Handling:

1. Introduction
2. Hierarchy of Standard Exception Classes
3. Keywords throws and throw
4. try catch and finally Blocks
5. Multiple Catch Clauses
6. Class Throwable
7. Unchecked Exceptions
8. Checked Exceptions
9. try-with-resources
10. Catching Subclass Exception
11. Custom Exceptions
12. Nested try and catch Blocks
13. Rethrowing Exception
14. Throws Clause

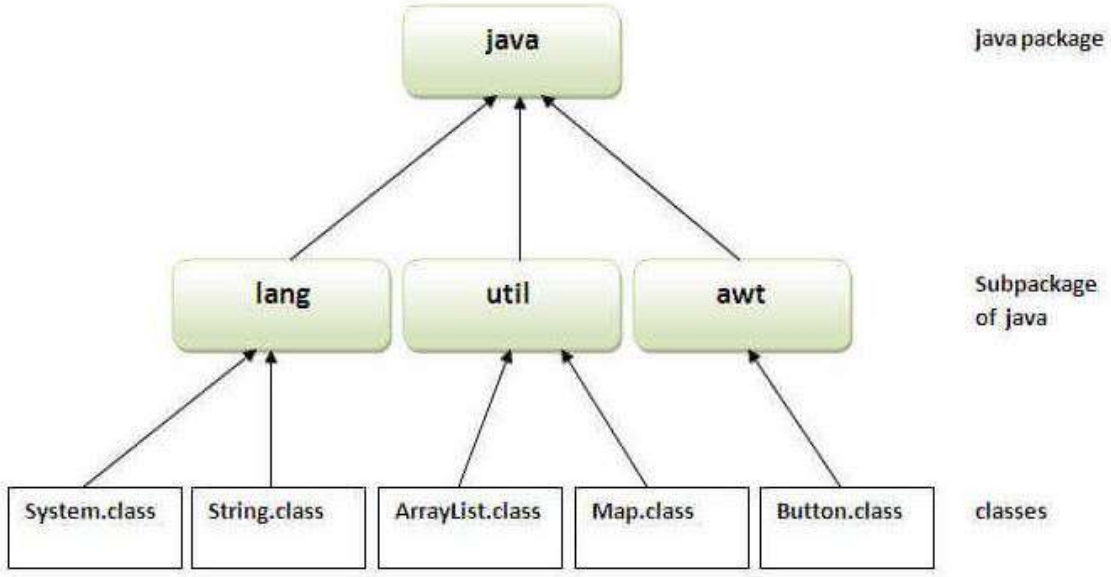
I. Packages and Java Library:

1. Introduction

- A **Java package** is a group of similar types of classes, interfaces and sub-packages.
- Package in java can be categorized in two form,
  i. built-in package and
  ii. user-defined package.
- There are many built-in packages such as java.lang, java.awt, java.javax, java.swing, java.util, etc.
- Here, we will have the detailed learning of creating and using user-defined packages.

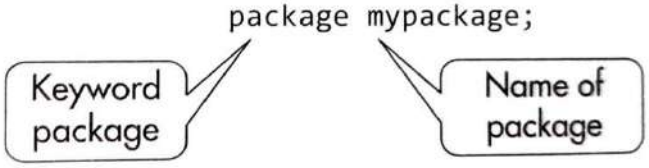
Advantage of Java Package

- Java package is used to categorize the classes and interfaces so that they can be easily maintained.
- Java package provides access protection.
- Java package removes naming collision.



2. Defining Package

- A package comprises a group of similar type of classes, interfaces, and sub-packages. It is defined with the keyword package followed by the name of the package and a semicolon, as illustrated in the following figure.



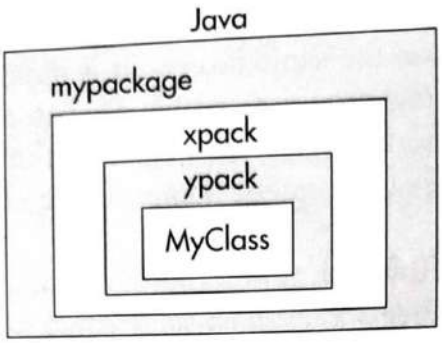


Fig. 10.1 Illustration of hierarchy of packages

```
java.mypackage.xpack.ypack.MyClass;
```

Here different levels in package are seperated by a period (.)

3. Importing Packages and Classes into Programs

- In Java, the **import** statement is used to bring certain classes or the entire packages, into visibility. As soon as imported, a class can be referred to directly by using only its name.
- The import statement is a convenience to the programmer and is not technically needed to write complete Java program. If you are going to refer to some few dozen classes into your application, the import statement will save a lot of time and typing also.
- In a Java source file, the import statements occur immediately following the package statement (if exists) and before any class definitions.
- If a program needs several classes of a package, it is better to import the whole package. This may be done as illustrated in the following example. For importing package awt contained in package java, the code is.

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

Importing Class in Programs

- All the classes defined in Java or by a programmer have to be in a package.
- A class may be include in a program by importing it through its fully qualified name.
- The selection of packages, sub-packages, and classes achieved by dot (.) selection operator, as shown in the following code

```
import java.packageP1.packageP2.package3.ClassName
```

Here

```
java      – Name of the package
packageP1 – Sub Package of Java
packageP2 – Sub Package of packageP1
packageP3 – Sub Package of packageP2
ClassName - Name of the Class in PackageP3.
```

User-defined Packages and Classes

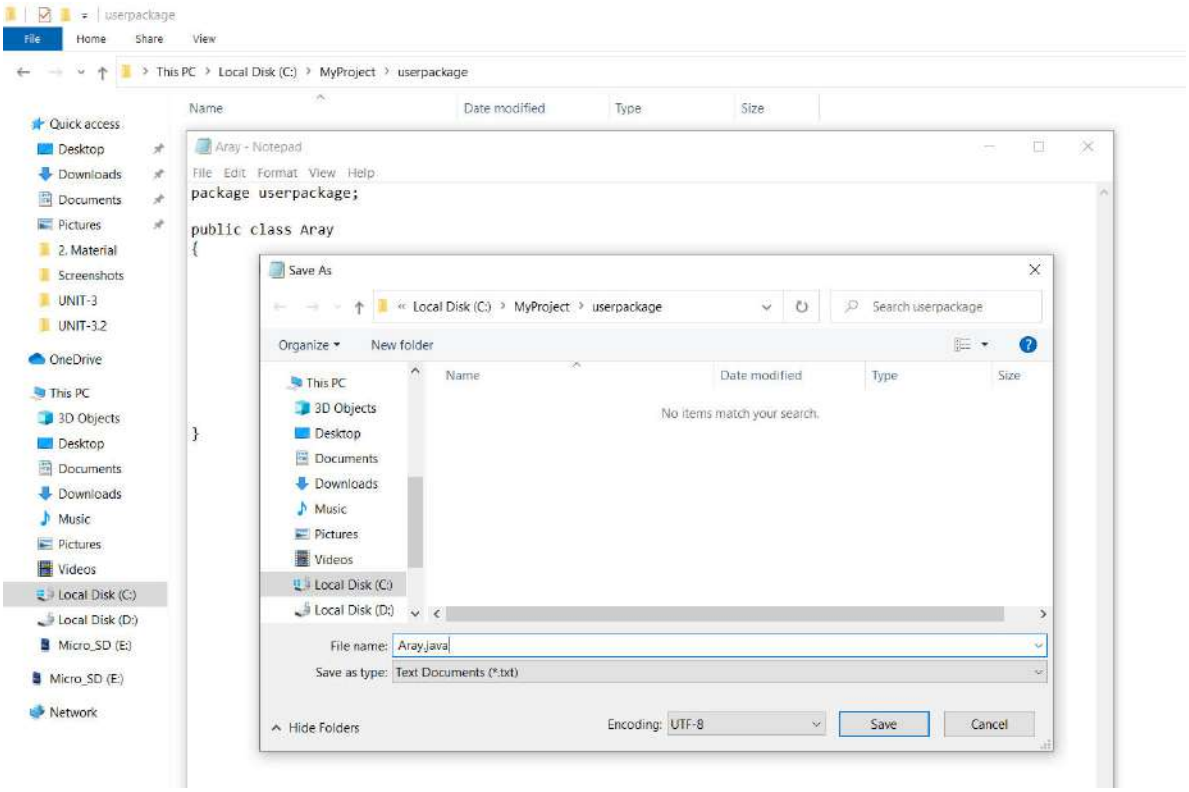
- One of the important features of the Java programming language is that it allows a programmer to build his/her own packages and classes and make use of them whenever the need arises for the use of standard Java library packages and classes.
- A programmer can save his/her own packages and classes in a directory, and from there, it can be imported to the ones required into his/her application program by using import keyword.
- For this, the following points should be taken care of while constructing a folder for a user's own package.

1. The folder name should be same as the name of the package desired to be created,
2. Create the desired class required to imported. Save it in the folder.
3. Compile the folder and class. However, do not run it because the class will not have the main. The application class to which it is imported will have the main method.
4 The application class should be saved outside this folder in any directory of your choice.

Example: The following class is constructed and saved in "userpackage" folder, which is saved in MyProject folder in Local Disk (C)

Program1: Aray.java in userpackage folder

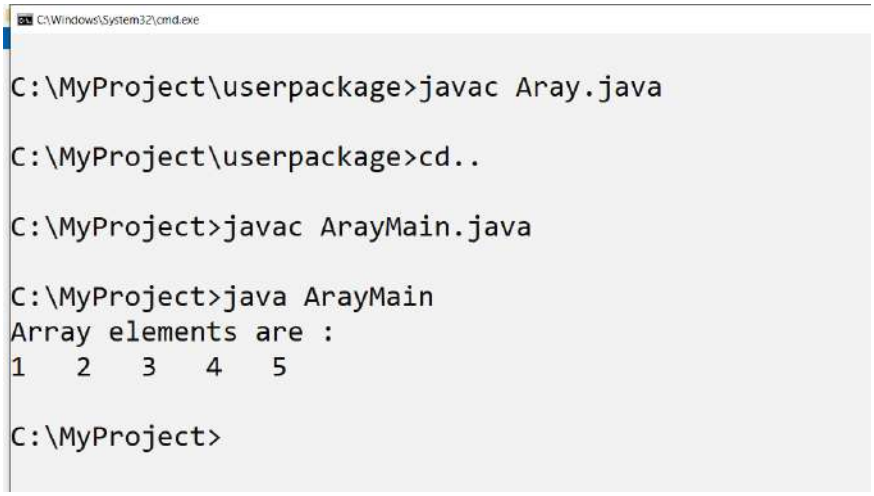
```
package userpackage;

public class Aray
{
      public Aray()
      {
            int[] num= new int []{1,2,3,4,5};
            System.out.println("Array elements are : ");
            for(int x: num)
              System.out.print(x + "   ");
            System.out.println();
      }
}
```



Program2: ArayMain.java in MyProject folder in Local Disk (C)

```
import userpackage.*;

public class ArayMain
{
        public static void main(String args[])
        {
            Aray objAray = new Aray();
        }
}
```

Execution Process:



4. Path and Class Path

- The PATH is an environmental variable used by the operating system to find the executive binary files

  Javac, which compiles the source code into bytecode, and
  java, which interprets the bytecode through for execution of the program
- such commands Path tells the system where to locate the JDK files that contain these commands.
- If the programmer is using classes or packages other than the Java standard their location is specified through classpath.
- The directory tree after the Java Development Kit (JDK) has been installed in Windows OS would look as illustrated in the following Diagram. The path is indicated by arrows

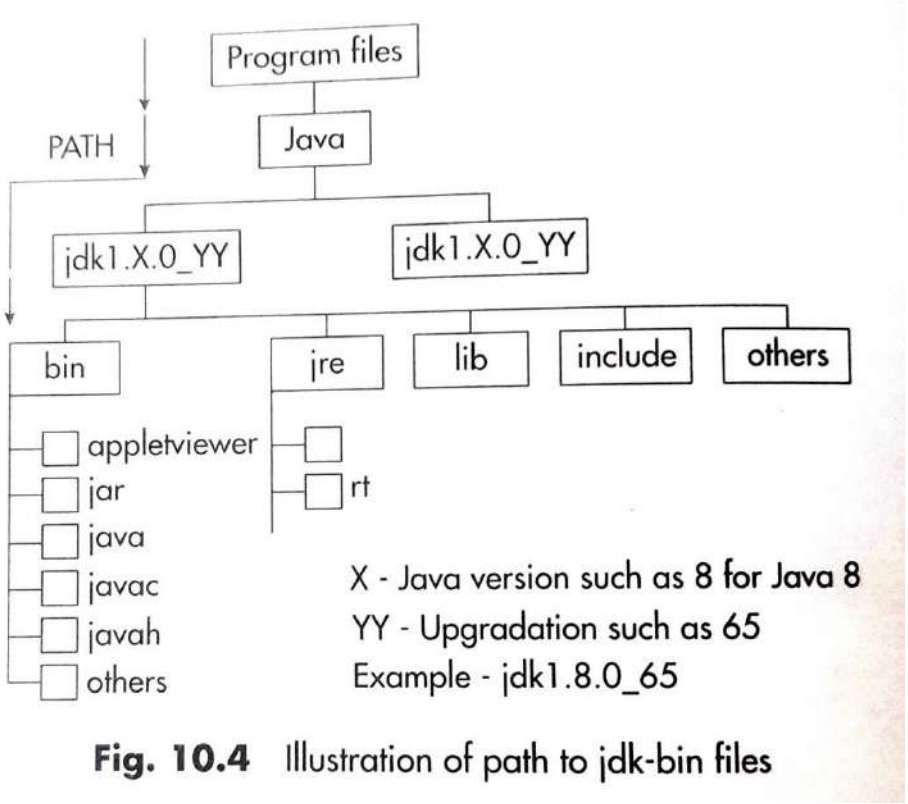


Fig. 10.4 Illustration of path to jdk-bin files

The system has to reach the location of jdk1.x.0_ yy\bin, which contains javac and java commands The path may be set in two ways

1. Set the path on Command Prompt for individual programs.
2. Set the path as value of environmental variable. Once done, you can run any number of programs without bothering about it, as done in running **ArayMain** class.

The path can be set for individual programs. Therefore, the path to JDK files installed in our computer is specified as

```
C:\Program Files\Javaljdk1.8.0_65\bin
```

Path is set by command set as

```
set path=%path%;C:\Program Files\Javaljdk1.8.0_65\bin
```

The important thing about setting path is that there will be no blank in assignment.

CLASSPATH is another environment variable. It is used by System or Application ClassLoader to locate and load **.class** that comprises the compiled Java bytecodes. Like path, the classpath is also not case sensitive and you may type it as CLASSPATH, ClassPath, or classpath.

Enter the path value as indicated in the following figure.

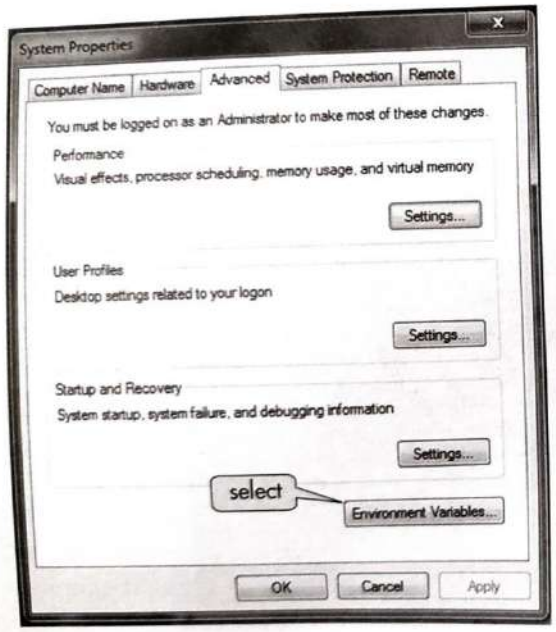


Fig. 10.7 Advanced system setting

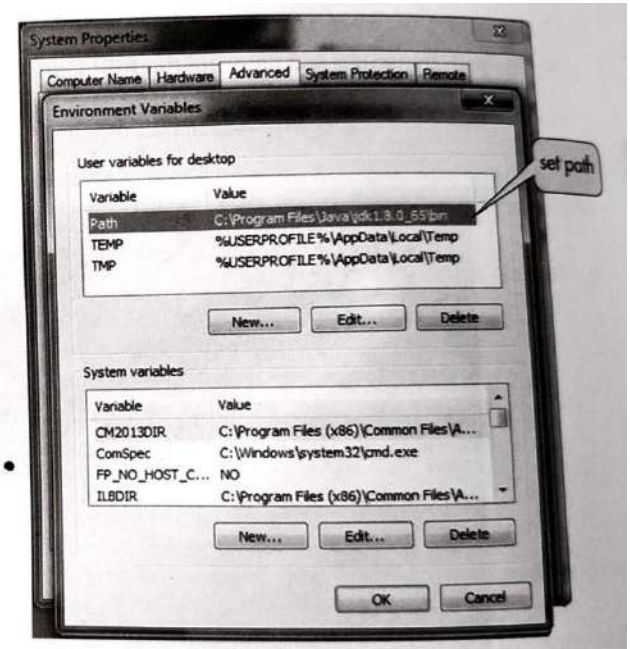


Fig. 10.8 Setting path value

5. Access Control

In Java, There are three access specifiers are permitted:

- public
- protected
- private

The coding with access specifiers for variables is illustrated as

```
Access_specifier type identifier;
```

Detials of Access specifiers are as follows.

Table 10.1 Access specifiers for classes

Access specifier	*Example of code*	*Access permitted*
No access specifier	`class A {}`	Accessible to classes in same package
public	`public class B{}`	Accessible to all classes in all packages
protected	`public class A{ protected class C {} }`	Used with nested classes; members of host (outer) class can access protected class because it is a member of the class
private	`public class A{ private class C {} }`	Used with nested classes; members of host (outer) class can access private class because it is a member of the class

Table 10.2 Access specifiers and permissible access for class members and different packages

Access specifier	*Access*
No access specifier	Access permitted to any other class belonging to the same package
public	Access permitted to any class in any package
protected	Access permitted to its subclasses in any package and also to any class in the same package
private	Access permitted to members of the same class only; no access permitted to any code outside the class

Table 10.3 Access to members permitted by specifiers

Class/Access specifier	*Access permitted Yes/No*			
	No specifier	*Public*	*Protected*	*Private*
Members of same class	Yes	Yes	Yes	Yes
Subclass in same package	Yes	Yes	Yes	No
Any other class in same package	Yes	Yes	Yes	No
Subclass in a different package	No	Yes	Yes	No
Any other class in any package	No	Yes	No	No

6. Packages in Java SE

Java has been evolving since its inception as Java 1. The latest version of Java is Java SE 8 (The **Java** Standard Edition 8). Originally all were supposed to be part of Java SE 7; however, it was taking more time, and Java SE 7 was released with some changes/ modifications and the remaining enhancements of Java were released in Java 8. It is worthwhile to at the additions and modifications in Java 7 and Java 8.

7. java.lang Package and its Classes

java.lang package is imported into every Java program by default. The programmer does not have to use the statement `import java.lang;` in the program. This package contains classes and interfaces that are essential for the design of Java programming language and are as follows:

i. Object class is the super class of all other classes in Java. It is the root of the class hierarchy
ii. Classes encapsulate the primitive data types in Java.
iii. Classes access system resources and other low-level entities.
iv. The class Math provides standard mathematical functions, for example, sine, cosine, square root, etc
v. The class Throwable is the super class of all error and exception classes in Java. It deals with the object that can be thrown by the throw statement. Subclasses of Throwable represent errors and exceptions
vi. The class Thread controls each thread in a multithreaded program.
vii. Classes String and StringBuffer provide commonly used operations on character strings.

It is all very necessary for a programmer to know the different classes available

Table 10.4 gives a brief description of the classes available in this package.

Class name	Brief description
Boolean	Wrapper class for Boolean values 'true' and 'false'
Byte	Wrapper class to provide reference to type byte
Character	Wrapper class to provide reference to type char and it supports two byte Unicode characters
Class	Subclass of Object, used for referring classes as objects
Double	Wrapper class to provide reference to type double
Enum	Defines methods that support enumeration
Float	Wrapper class to provide reference to type float
Integer	Wrapper class to provide reference to type int
Long	Wrapper class to provide reference to type long
Math	Contains several mathematical methods
Number	Abstract super class for the Wrapper classes such as Byte, Short, Integer, Float, and Double
Object	Super class to all other classes in Java
Package	For managing packages
String	Subclass of Object; it defines methods for handling strings.

8. Class Object

The class Object is one of the important classes of Java because it is the super class to all other classes. The class has one constructor and defines 11 methods that are available to all the objects. The methods of class are described in following Table.

Table 10.6 Methods defined in Object class

Method	*Description*
`Object clone()`	Creates a new copy of invoking object
`boolean equals(Object obj)`	Returns true if invoking object and argument object are equal
`void finalize()`	Executed when the object is garbage collected and the method needs to be overridden by a class
`final Class getClass()`	Identifies the class of invoking object
`int hashCode`	Returns hash code of the invoking object
`final void notify()`	Notifies the thread waiting on invoking thread
`final void notify all()`	Notifies all the threads waiting on invoking thread
`String toString()`	Returns string equivalent of invoking object
`final void wait()`	Invoking thread waits till notified and throws `InterruptedException`
`final void wait((long milliseconds))`	Makes the invoking thread wait for a specified time in milliseconds and throws `InterruptedException`

Table 10.6 *(Contd)*

Method	*Description*
`final void wait(long milliseconds, int nanoseconds)`	Makes the invoking thread wait for a specified time in milliseconds and nanoseconds, if the system permits nanoseconds and throws `InterruptedException`

9. Enumeration

- An enum is a special "class" that represents a group of **constants** (unchangeable variables, like final variables).
- To create an enum, use the enum keyword (instead of class or interface), and separate the constants with a comma. Note that they should be in uppercase letters:

The members of an enum type declaration are as follows:

i. Members declared in the body of the enum type declaration.
ii. Members inherited from the super class Enum<E>.
iii. An enum declaration is implicitly final unless at least one declared constant has a class in its body
iv. Constants declared, which are implicitly public, static, and final.
v. Constants' body may have annotations, fields, expressions, or a class.
vi. Enum keyword has the properties of a class; therefore, it can implement interfaces and its member constants can have inner class.

Example:

```
class Enum
{
   enum Level { LOW, MEDIUM, HIGH}
   public static void main(String args[])
   {
   System.out.println("First Value = " + Level.LOW);
   }
}
```

Output:

```
C:\ >javac Enum.java

C:\ >java Enum
First Value =  LOW
```

10. class Math

The Math class is an important class of `java.lang` package. It defines methods for evaluating several mathematical functions such as trigonometry functions (sine, cosines, etc.), exponential functions, rounding off functions, and other miscellaneous mathematical functions. The class is defined as a final class.

```
public final class Math extends Object
```

The class inherits methods from object class.

`double E` - Returns value of E (the base of natural logarithm) as double number
`double PI` - Returns the PI value which is 3.14159 , (ratio of circumference to diameter of circle as double number
All the methods defined in Math class are static, and thus, they may be called without object; however, the name of the class is required.
For example, the codes for calling sqrt() and pow() methods are

`Math.sqrt(10);` // returns square root of 10

`Math.pow(2,4);` // returns 2 to the power 4

Example: MathDemo.java

```
class MathDemo
{
     public static void main(String args[])
     {
      System.out.println("Value of E = "+ Math.E);
      System.out.println("Value of PI = "+ Math.PI);
      System.out.println("Squar Root of 25 = "+ Math.sqrt(25));
      System.out.println("2 to the power of 4 = "+ Math.pow(2,4));
       }
}
```

Output:

```
C:\ >javac MathDemo.java

C:\ >java MathDemo

Value of E = 2.718281828459045

Value of PI = 3.141592653589793

Squar Root of 25 = 5.0

2 to the power of 4 = 16.0
```

11. Wrapper Classes

- Wrapper class wraps (i.e., encloses) a primitive data type and provides its object representation.
- In Java, a simple data type can also be converted into an object using Wrapper classes.
- Many data structures in Java are designed to operate on objects.
- Wrapper classes that encapsulate a primitive data type within an object.
- **The eight primitive data types**, namely
    i. boolean,
    ii. byte,
    iii. short,
    iv. int,
    v. long,
    vi. float,
    vii. double, and
    viii. char,
- These 8 primitive data types are not objects of classes; hence, they cannot be passed on by references and they are passed on by value only.
- Therefore, in order to provide object representation, **eight Wrapper classes are defined** in java.lang package which is imported by default in all the Java programs.
- The **eight Wrapper classes are**

i. Boolean
ii. Byte,
iii. Short,
iv. Integer,
v. Long,
vi. Float,
vii. Double
viii. Char

Methods of Wrapper Classes :

i. equals()
ii. isInfinite()
iii. isNaN()
iv. byteValue()
v. compareTo(type Object)
vi. doubleValue()
vii. floatValue()
viii. hashCode()
ix. intValue()
x. longValue()
xi. ShortValue()
xii. toHexString(type number)
xiii. valueOf(type number)
xiv. valueOf(String str)
xv. toString(type number)

Example: WrapperClasses.java

```
class WrapperClasses
{
     public static void main(String args[])
     {
      // for int values
      int i = 10;
      Integer in = i;
      System.out.println("Value of in = "+ in);

      //for float values
      float f = 25.6f;
      Float fn = f;
      System.out.println("Value of fn = "+ fn);

       }
}
```

Output:

```
C:\>javac WrapperClasses.java

C:\ >java WrapperClasses
Value of in = 10
Value of fn = 25.6
```

12. Auto-boxing and Auto- unboxing

Auto-boxing

- Auto-boxing involves automatic conversion of the **primitive data types into its corresponding wrapper types** so that the value may be represented by reference as the objects of other classes.
- This includes the conversion of
    - int to Integer,
    - double to Double,
    - float to Float,
    - boolean to Boolean, etc.,

Auto-unboxing

- Auto - unboxing is the reverse process in which the **wrapping class objects are converted into the corresponding primitive data types**.
- This includes conversion of
    - Integer to int,
    - Long to long,
    - Double to double, etc.

Example: AutoBoxing.java

```
class AutoBoxing
{
     public static void main(String args[])
     {
      // Auto Boxing
      int i = 10;
      Integer in = new Integer(i); //Auto Boxing: Primitive to Wrapper
      System.out.println("Value of Wrapper variable in = "+ in);

      //Auto Unboxing
      Integer x = 20;
      int y = x.intValue(); //Auto Unboxing : Wrapper to Primitive
      System.out.println("Value of primitive variable y = "+ y);

       }
}
```

Output:

```
C:\>javac AutoBoxing.java

C:\>java AutoBoxing
Value of Wrapper variable in = 10
Value of primitive variable y = 20
```

13. java.util Classes and Interfaces

java.util is an important package of the Java library. It contains classes and interfaces of collections, event model, and miscellaneous utility classes. The classes and interfaces contained in this package are very important as these are required for many programs. For instance, it comprises Math class that supports Mathematical expressions.

Classes in java.util package

AbstractCollection	AbstractList	AbstractMap	AbstractQueue
AbsractSequentiallist	AbstractSet	ArrayList	Arrays
BitSet	Calendar	Collections	Currency
Date	Dictionary	EnumMap	EnumSet
EventListenerProxy	EventObject	Formatter	HashMap
HashSet	Hashtable	IdentityHashMap	LinkedHashSet
LinkedList	ListSources Bundle	Locale	Observable
PriorityQueue	Properties	PropertyPermission	Property
ResourceBundle	Random	ResourceBundle	Scanner
ServiceLoader	SimpleTimeZone	Stack	StringTokanizer
Timer	TimeTask	TimeZone	TreeMap
TreeSet	UUID	Vector	WeakHashMap

(Write any ten class names in the Examination)

Interfaces in java.util package

Collection	Comparator	Deque	Enumeration
EventListener	Formattable	Iterator	List
ListIterator	Map	Map. Entry	NavigableMap
NavigableSet	Observer	Queue	RandomAccess
Set	SortedMap	SortedSet	

(Write any ten interface names in the Examination)

Scanner Class

Java Scanner class is a text scanner that breaks the input into tokens using delimiter, which is whitespace by default. Delimiter is a character that identifies the beginning or end of character string and it is not a part of the character string.

The received string can then be converted into values of different types using the various next() methods. The application of this class is not thread safe. Java Scanner class extends Object class and implements Iterator and Closeable interfaces. For instance, the following code allows the user to input into the program an integer number through the keyboard.

Sample Code:

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter your roll no.");
int rollno = sc.nextInt();
```

Example : UserInput.java

```
import java.util.Scanner;
public class UserInput
{
      public static void main(String[] args)
      {
      Scanner scaninput = new Scanner (System. in);
      int n;
      int m;
      System.out. print( "Enter the value of n : ");
      n=scaninput.nextInt();
      System.out. print( "Enter the value of m : ");
      m=scaninput.nextInt();

      System.out.println("Sum of two numbers is ="+(n+m));

      }
}
```

Output

```
   C:\>javac UserInput.java

   C:\>java Arithmetic
   Enter the value of n : 10
   Enter the value of m : 3
   Sum of two numbers is =13
```

Radix in java.util package

Radix is another name of base of the number system being referred to.
Radix values specifies the base value of the numbers ie 2 for Binary Numbers, 8 for Octal Numbers, 10 for Decimal Numbers (The default radix number is 10) and 16 for HexaDecimal Numbers

- **`radix()`** method prints the current radix value
- **`useRadix()`** method used to change the radix value

Example : TestRadix.java

```
import java.util.Scanner;
public class TestRadix
{
   public static void main(String[] args)
   {
      int n;

      //for Decimal Numbers
Scanner input = new Scanner(System.in);
System.out.println("For Decimal numbers- radix = "+ input.radix());
System.out.print("Enter an integer number: ");
n = input.nextInt();
System.out.println("The number n = " + n);

      //for HexaDecimal Numbers
input.useRadix(16);
System.out.println("\nFor HexaDecimal numbers radix = "+ input.radix());
System.out.print("Enter a number with base 16: ");
n = input.nextInt ();
System.out.println("The number n= " + n);

      //for Octal Numbers
input.useRadix(8);
System.out.println("\nFor Octal numbers radix = "+ input.radix());
System.out.print("Enter a number with base 8 : ");
n = input.nextInt();
System.out.println("The number n = " + n);

//for Binary Numbers
input.useRadix(2);
System.out.println("\nFor Binar numbers radix = "+ input.radix());
System.out.print("Enter a number with base 2 : ");
n = input.nextInt();
System.out.println("The number n = " + n);

   }
}
```

Output:

```
E:\>javac TestRadix.java
E:\ >java TestRadix
For Decimal numbers- radix = 10
Enter an integer number: 56
The number n = 56

For HexaDecimal numbers radix = 16
Enter a number with base 16: 55
The number n= 85

For Octal numbers radix = 8
Enter a number with base 8 : 41
The number n = 33

For Binar numbers radix = 2
Enter a number with base 2 : 11110
The number n = 30
```

14. Formatter Class

- The class Formatter supports the formatting of the output of characters, strings, and numeric types.
- The declaration of this class is highly inspired by the printf method of C programming language. It is the printf method of C in Java clothing;
- In Java, printf() method returns an object of print stream.
- Class Formatter provides support for layout justification, alignment, formats for numeric, string, and date/time data.
- The class declaration is

```
public final class Formatter
                extends Object
                implements Closeable, Flushable
```

- The format string for output of numeric values comprises a % sign followed by a conversion character.
- The field width and precision are specified between the % sign and the conversion character.

Table 10.17 Conversion characters

Conversion character			Description
Lower case/Upper case		Category	
b	or B	General	Boolean output—true/false in lower case or upper case
c	or C	Character	Represent in Unicode character
h	or H	General	Represent as hexadecimal number
s	or S	General	Represent as string
d		Integral	Output formatted as decimal integer used for byte, short, int, and long
o		Integral	Represent as octal number
x	or X	Integral	Represent as hexadecimal number in lower or upper case
e	or E	Floating point	For representation in scientific notation
f		Floating point	For floating point numbers—float or double
G	or G	Floating point	Normal or exponential representation
n		Separator	Result on right side of n is shifted to next line
%		Percent	Results in % sign
t	or T	Date and time	Conversion character for time and date

Example : FormatterDemo.java

```
import java.util. Formatter;
public class FormatterDemo
{
      public static void main(String[] args)
      {
      int x = 165;
      float f = 432.14159f;
      double d = 5654.87543;
      char ch = 'a';
      String str = "Hello World";

      //Declaration of two objects formt and fmt
      Formatter formt = new Formatter(), fmt= new Formatter();

      //Formatting and printing formt object
      formt.format("x = %d, f= %f, d= %f, ch= %c and str= %s", x, f,
d, ch, str);
      System.out.println(formt);

      //Formatting and printing fmt object
      fmt.format("Upper case of ch = %C and str = %S ", ch, str);
      System.out.println(fmt);
      }
}
```

Output:

```
C:\>javac FormatterDemo.java

C:\>java FormatterDemo
x = 165, f= 432.141602, d= 5654.875430, ch= a and str= Hello World
Upper case of ch = A and str = HELLO WORLD
```

15. Random Class

The class Random of java.util package is generally used for generating random numbers of different types such as int, double, float, long, and byte. The method random() of class Math may also be used for generating double numbers.

Example: Random.java

```
import java.util.Random;
public class RandomNumbers
{
     public static void main(String[] args)
     {
     Random rand = new Random();

     // Random Integer values
     System.out.print("The random numbers are: ");
     for (int i = 0; i<10; i++)
         System.out.print((1 + rand.nextInt(6))+ " ");

     // Random Double values
     System.out.print("\nTHe double random numbers are: ");
     for (int i = 0; i<5; i++)
        System.out.printf(" %.3f", (rand.nextDouble()));


     // Random Boolean values
     System.out.print("\nTHe boolean random values are: ");
     for (int i = 0; i<5; i++)
       System.out.print((rand.nextBoolean())+ " ");
     }
}

Output:
C:\ >javac RandomNumbers.java

C:\ >java RandomNumbers
The random numbers are: 5 1 1 5 2 6 5 2 5 2
THe double random numbers are:  0.289 0.543 0.337 0.243 0.250
THe boolean random values are: false true false false false
```

16. Time Package

- This package provides support for date and time-related features for program developers.
- It comprises classes that represent date/time concepts including instants, duration, date, time, time zones, and periods.
- The date and time framework in Java prior to Java SE 8 consisted of classes such as
    - `java.util.Date,`
    - `java.util.Calendar`, and
    - `java.util.SimpleDateFormatter`

  which posed problems to programmers because of the following reasons:
  These classes are
    i. not thread safe
    ii. present a poor show.
    iii. poor design.
    iv. it was difficult to work with different calendar systems followed in different parts of the world.

- The result is the development of an entirely new package `java.time` that has classes that are thread safe; these classes have robust design and they can be adapted to work with different calendar systems.

Example: TimePrg.java

```
import java.time.LocalDate;
import java.time.LocalTime;
import java.time.LocalDateTime;
import java.time.Period;

// declaring the class.
class TimePrg
{
  public static void main(String[] args)
  {
  //creating an object
  LocalTime time = LocalTime.now();
  System.out.println("The time at present is :" + time);

  // creating another object
  LocalTime newTime;
  newTime = time.plusHours (2);
  System.out.println("The modified time:"+ newTime);

  // creating an object
  LocalDate date = LocalDate.now();
  System.out.println("The date today is : "+date);

  LocalDateTime datetime = LocalDateTime.now();
  System.out.println("The date and time at present are:
"+datetime);
```

```
  //1 year, 3 months and 10 days
  Period p = Period.of(1,3,10);

  LocalDateTime newdate = datetime.plus(p);
  System.out.println("The newdate is:" + newdate);
  }
}
```

Output:

```
C:\ >javac TimePrg.java

C:\ >java TimePrg
The time at present is :07:24:57.342943800
The modified time:09:24:57.342943800
The date today is : 2021-06-12
The date and time at present are: 2021-06-12T07:24:57.354018300
The newdate is:2022-09-22T07:24:57.354018300
```

17. Class Instant (java.time.Instant)

Java Instant class is used to represent the specific moment on the time line. It inherits the Object class and implements the Comparable interface.

Table 10.23 Methods of class Instant

Method	*Description*
isAfter	Compares two time instants `Instant.now() isAfter Instant.now().minusHours(1);`
isBefore	Compares two time instants `Instant.now() is before Instant.now().plusHours(1);`
plus	Adds time to instant. An example of this code is `Instant later = Instant.now().plusMinutes(30);`
minus	Subtracts time from Instant. An example of this code is `Instant before = Instant.now().minusHours(2);`
until	Returns how much time exists between two time Instant objects

Example: TimeInstantDemo.java

```
import java.time.Instant;
public class TimeInstantDemo
{
  public static void main(String[] args)
  {
    Instant now = Instant.now();
    System.out.println("Now Time = "+ now);

    Instant before = Instant.now().minusSeconds(600);
    System.out.println("before Time = "+ before);

    Instant later = Instant.now().plusSeconds(900);
    System.out.println("later Time = "+ later);

  }
}
```

Output:

```
C:\ >javac TimeInstantDemo.java

C:\ >java TimeInstantDemo
Now Time = 2021-06-12T02:25:19.772322600Z
before Time = 2021-06-12T02:15:19.800125300Z
later Time = 2021-06-12T02:40:19.804155800Z
```

Example2: UntilDemo.java

```
import java.time.*;
import java.time.temporal.*;

public class UntilDemo
{
      public static void main(String[] args)
      {
       long td;
            LocalTime time = LocalTime.parse("10:15:30");
            LocalTime time1 = LocalTime.now();

      //Calculating time difference (in hours) thorugh until()
method
      td = time1.until(time, ChronoUnit.HOURS);
            System.out.println("Time  difference  =  "+  td  +  "
Hours");
```

```
      //Calculating time difference (in minutes)thorugh until()
method
      td = time1.until(time, ChronoUnit.MINUTES);
            System.out.println("Time  difference  =  "+  td  +  "
Minutes");

      //Calculating time difference (in seconds)thorugh until()
method
      td = time1.until(time, ChronoUnit.SECONDS);
            System.out.println("Time  difference  =  "+  td  +  "
Seconds");
   }
}
```

```
Output:
C:\>javac UntilDemo.java

C:\>java UntilDemo
Time difference = 2 Hours
Time difference = 169 Minutes
Time difference = 10188 Seconds
```

18.Formatting for Date/Time in Java

The old API for formatting has been revised in Java 8 to a new class for formatting, printing, and parsing **Date/Time** objects through DateTimeFormatter class.

DateTimeFormatter class declaration is as follows.

```
   public final class DateTimeFormatter extends object
```

The class provides the following three ways to achieve the desired results.

1. By using the predefined constants in the language that represent a particular format of formatting date/ time objects.
2. By using patterns defined by the programmer.
3. By using localized format.

Example: DateTimeFormatter.java

```
import java.time.format.DateTimeFormatter;
import java.time.LocalDate;
import java.time.DateTimeException;
import java.time.LocalDateTime;

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

```
      //defining an instance
      LocalDateTime datetime = LocalDateTime.now();
      System.out.println("Default format of LocalDateTime: "+ datetime);
      System.out.println("\nIn 150_LOCAL_DATE_TIME Format: " +
datetime.format(DateTimeFormatter.ISO_LOCAL_DATE_TIME));

      LocalDate date= LocalDate.now();
      System.out.println("\nDefault format of LocalDate= " + date);


      System.out.println("\nIn ISO LOCAL DATE Format: " +
date.format(DateTimeFormatter.ISO_LOCAL_DATE));
      System.out.println("\nUser format:" + datetime. format
(DateTimeFormatter.ofPattern("dd-MMM-yyyy hh:mm:ss")));
      }
}
```

Output:

```
C:\>javac DateTimeFormatDemo.java

C:\>java DateTimeFormatDemo
Default format of LocalDateTime: 2021-06-12T19:24:32.807557700

In 150_LOCAL_DATE_TIME Format: 2021-06-12T19:24:32.8075577

Default format of LocalDate= 2021-06-12

In ISO LOCAL DATE Format: 2021-06-12

User format:12-Jun-2021 07:24:32
```

19. Temporal Adjusters Class

Temporal Adjuster Interface

The java time temporal defines an interface by name TemporalAdjuster that defines methods that take a temporal value and return an adjusted temporal value according to a user's specifications.

Temporal Adjusters class

The package also defines a class TemporalAdjusters, which contains predefined used by a programmer. Some examples of adjusters are as follows.

i. firstDayOfMonth()
ii. lastDayOfMonth()
iii. firstDayOfyear()
iv. lastDayOfYear
v. firstDayOfWeek ()

vi. lastbayOfWeek()
vii. firstInMonth (DayOfWeek. Monday)
viii. lastInMonth (DayOfWeek. FRIDAY)

All these methods are static methods and may be used with static import.
Static import is used to access any static member of a class directly without referring its class name. There are two general forms of declaration of static import statements.

The first form of static import statement involves importing only a single member of a class as shown:

```
import static package.class-name.static-member-name:
```

As for instance,

```
import static java.lang.Math.sqrt;
```

The second form of static import statement involves importing all the static members of a class as

```
import static package.class-type-name.*;
```

This is shown in the following example:

```
import static java.lang.Math.;
```

It is different from import statement that allows the programmer to access classes of a package without providing package name or qualification. It is to be noted that import statement makes the classes and interfaces accessible, whereas the static import provides accessibility to static members of the class only.

Example: TemporalAdjusterDemo.java

```
import java.time.*;
import java.time.temporal.TemporalAdjusters;

public class TemporalAdjusterDemo
{
      public static void main(String[] args)
      {
      // defining an object

      LocalDate date1= LocalDate.of(2015, Month.NOVEMBER, 20);
      DayOfWeek dow1 = date1.getDayOfWeek();

      System.out.println("The day of week on 20th Nov 2015 =" +
dow1);

      System.out.println("The first day of November = "+
date1.with(TemporalAdjusters.firstDayOfMonth()));
```

```
     System.out.println("The last day of November = "+
date1.with(TemporalAdjusters.lastDayOfMonth()));
     }
}
```

Output

```
C:\>javac TemporalAdjusterDemo.java


C:\>java TemporalAdjusterDemo
The day of week on 20th Nov 2015 =FRIDAY
The first day of November = 2015-11-01
The last day of November = 2015-11-30
```

II. Exception Handling:

1. Introduction

- In Java, an exception is an *object* of a relevant exception class. **When an error occurs in a method, it throws out an exception object** that contains the information about where the error occurred and the type of error. **The error (exception)** object is passed on to the runtime system, which searches for the appropriate code that can handle the exception.
- The event handling code is called ***exception handler*.**
- **Exception handling** is a mechanism that is used to handle runtime errors such as classNotFound and IO. This ensures that the normal flow of application is not disrupted and program execution proceeds smoothly.
- **The exception handling code** handles only the type of exception that is specified for it to handle. **The appropriate code** is the one whose specified type matches the type of exception thrown by the method.
- If the runtime system finds such a handler, it passes on the exception object to the handler. **If an appropriate handler is not found, the program terminates**.
- The method that creates an exception may itself provide a code to deal with it.
- **The try and catch blocks are used to deal with exceptions.**
- The code that is likely to create an exception is kept in the try block, which may include statements that may create or throw exceptions.
- The exception object has a data member that keeps information about **the *type* of exception** and it becomes an argument for another block of code that is meant to deal with the exception.
- **There may be more than one catch blocks.**
- There are basically two models of exception handling.

1. Termination Model

- According to **termination model**, when the method encounters an exception, further processing in that method is terminated and control is transferred from the point where an exception occurs to the point where its nearest matching exception handler (i.e., the catch block) is located.
- This model is analogous to the real-life situation when the gas in the cylinder gets over while cooking.
- Under the termination model, the cooking process will stop, whereas in resumption model, you would change the cylinder with a new one and continue with the cooking process.

2. Resumption model

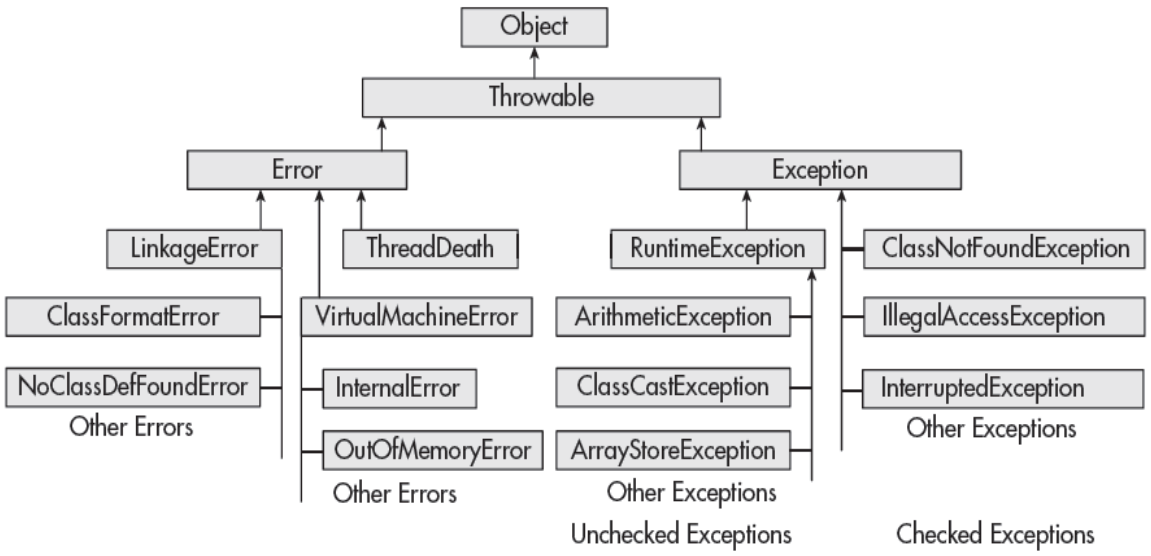
- Thus, the alternative approach is based on the **resumption model** wherein the exception handler tries to rectify the exception situation and resume the program.
- Resumption model was mostly used in earlier languages including PL/I, Mesa, and BETA.
- The implementation of resumption model in contemporary languages such as C++ and

Java is rarely found as the effort to implement this model is quite high.

- This is because the program code becomes quite cumbersome and difficult to understand, and further, it is more error prone.
- However, there are situations where the resumption model is quite useful.

2. Hierarchy of Standard Exception Classes

- In Java, exceptions are instances of classes derived from the class Throwable which in turn is derived from class Object.
- Whenever an exception is thrown, it implies that an object is thrown.
- Only objects belonging to class that is derived from class Throwable can be thrown as exceptions.
- The next level of derived classes comprises two classes: the class Error and class Exception.
- Error class involves errors that are mainly caused by the environment in which an application is running.
- All errors in Java happen during runtime.



- Examples
  OutOfMemoryError occurs when the **JVM runs out of memory** and **StackOverflowError** occurs when the **stack overflows**.
- Exception class represents exceptions that are mainly caused by the application itself.
- **It is not possible to recover from an error using try–catch blocks**.
- **The only option available is to terminate the execution** of the program and **recover from exceptions using either the try–catch block or throwing an exception.**
  - The exception class has several subclasses that deal with the exceptions that are caught and dealt with by the user's program.
  - The Error class includes such errors over which a programmer has less control.

- A programmer cannot do anything about these errors except to get the error message and check the program code.
- A programmer can have control over the exceptions (errors) defined by several subclasses of class Exception.

The subclasses of Exception class are broadly subdivided into two categories.
Unchecked exceptions These are subclasses of class **RuntimeException**, derived from Exception class. For these exceptions, **the compiler does not check** whether the method that throws these exceptions has provided any exception handler code or not.

Checked exceptions These are direct subclasses of the **Exception** class and are not subclasses of the class RuntimeException. These are called so because **the compiler ensures (checks)** that the methods that throw checked exceptions deal with them.

This can be done in two ways:
1. The method provides the exception handler in the form of appropriate try–catch blocks.
2. The method may simply declare the list of exceptions that the method may throw with throws clause in the header of the method.
   - In this case, the method need not provide any exception handler.
   - It is a reminder for those who use the method to provide the appropriate exception handler.
   - If a method does not follow either of the aforementioned ways, the compilation will result in an error.

Methods defined in class Throwable

Table 11.1 Methods defined in class `Throwable`

Method name	*Description*
`getMessage()`	It returns a string that gives information about the current exception and consists of a fully qualified name of the exception class and a relevant brief description.
`toString()`	The class `Throwable` overrides the method `toString()` of Object class for displaying messages on screen.
`printStackTrace()`	It traces and displays the hierarchy of method calls that resulted in the exception. The information will be displayed on the screen in the case of a console program.

Example: TryCatchDemo.java

```
class TryCatchDemo
{
   public static void main(String args[])
  {
  int i=6,j=0,k;
  try {
        System.out.println("Entered try block");
        k=i/j;
        System.out.println("Exiting try block");
  }
   catch (ArithmeticException e)
   {
        System.out.println("e = " + e);
   }
   System.out.println("End of the Program ");
   }
}
```

Output:

```
C:\ >javac TryCatchDemo.java

C:\ >java TryCatchDemo
Entered try block
e = java.lang.ArithmeticException: / by zero
End of the Program
```

3. Keywords throws and throw

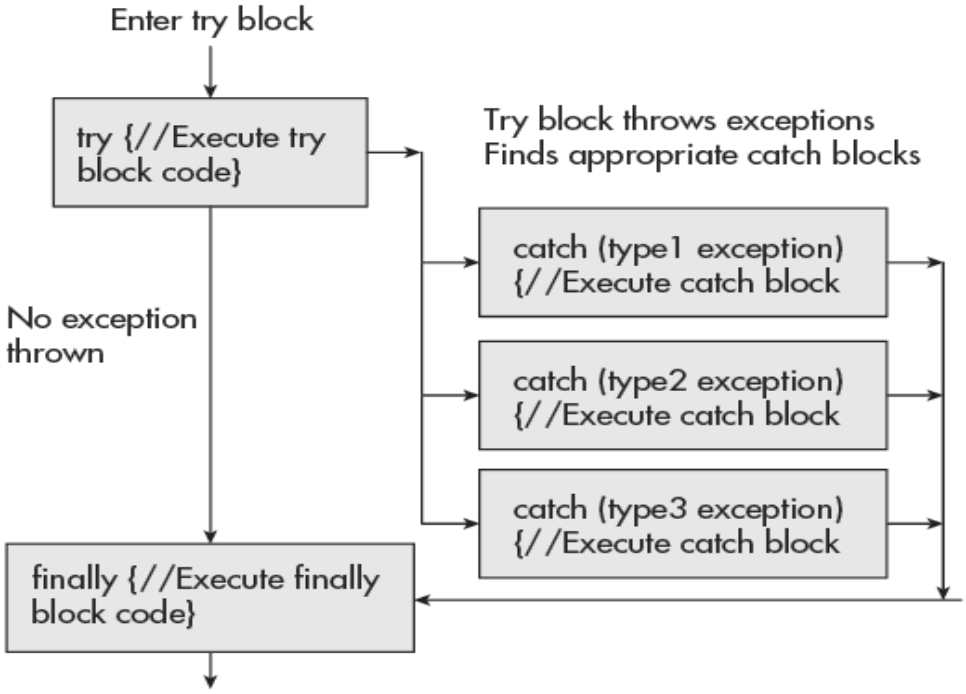
- If a method can cause one or more checked exceptions directly or indirectly by calling other methods and throw exceptions and does not deal with them, it must declare the list of exceptions it can throw by using the *throws* clause.
- By doing this, the caller of the method can ensure that appropriate arrangements are made to deal with the exceptions.
- For this, keep the method in a try block, and provide suitable catch blocks.
- The program will not compile if this is not done.
- **The throws clause comprises the keyword throws followed by the list of exceptions that the method is likely to throw separated by commas.**
- The method declaration with throws clause is

```
type Method_Name(type parameters)  throws  list-of-Exceptions
{
  /* Body of Method*/
 }
```

4. try catch and finally Blocks

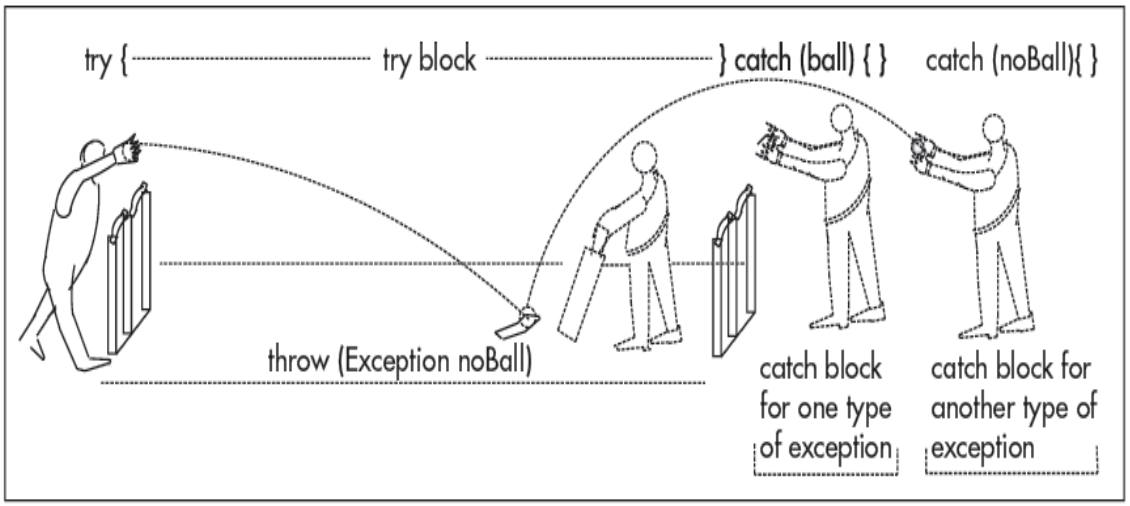
try {} Block

- The program code that is most likely to create exceptions is kept in the try block, which is followed by the catch block to handle the exception.
- In normal execution, the statements are executed and if there are no exceptions, the program flow goes to the code line after the catch blocks.
- However, if there is an exception, an exception object is thrown from the try block.
- Its data members keep the information about the *type* of exception thrown.
- The program flow comes out of the try block and searches for an appropriate catch block with the same *type* as its argument.



catch {} Block

- A catch block is meant to catch the exception if the *type* of its argument matches with the *type* of exception thrown.
- If the *type* of exception does not match the *type* of the first catch block, the program flow checks the other catch blocks one by one.
- If the *type* of a catch block matches, its statements are executed.
- If none matches, the program flow records the *type* of exception, executes the finally block, and terminates the program.



Example: TryCatchDemo2.java

```
class TryCatchDemo2
{
   public static void main(String args[])
   {
   int i=6,j=0,k;
   try {
         System.out.println("Entered try block");
         k=i/j;
         System.out.println("Exiting try block");
   }
   catch (ArithmeticException e)
   {
       System.out.println("e = " + e);
   }

   j=2; // Value of j Changed
   System.out.println("When J =2, i/j = " + i/j);
   System.out.println("End of the Program ");
    }
}
```

Output:

```
C:\ >javac TryCatchDemo2.java

C:\ >java TryCatchDemo2
Entered try block
e = java.lang.ArithmeticException: / by zero
When J =2, i/j = 3
End of the Program
```

finally {} Block

- This is the block of statements that is always executed even when there is an exceptional condition, which may or may not have been caught or dealt with.
- Thus, finally block can be used as a tool for the clean up operations and for recovering the memory resources.
- For this, the resources should be closed in the finally block.
- This will also guard against situations when the closing operations are bypassed by statements such as continue, break, or return.
-

Finalize() in Exception Handling

- finalize() method releases system resources before the garbage collector runs for a specific object.
- JVM allows finalize() to be invoked only once per object
- **The syntax for using finalize() method is given as follows:**

```
@Override //
protected void finalize() throws Throwable
{
   try{
   …….
   } catch(Throwable t)
     {
        throw t;
    }finally{
     super.finalize();
    }
}
```

- Following points need to be taken into consideration when using finalize() method:

1. **super.finalize()** method is always called in finalize() method.
2. Time critical application **logic should not be placed in finalize()** method.

5. Multiple Catch Clauses

- Java allows to write multiple catch block in the program.
- Often the programs throw more than one *type* of exception, at that time more catch block need to be used.
- The exception classes are connected by Boolean operator OR.

Example: MultiCatchExample.java

```
class MultiCatchExample
{
   public static void main(String args[])
   {
  try {
       int a=0, b=56, c;
```

```
            int aray[] = {4,5,6,7};
            System.out.println("The Array elements are : ");

            for(int i=0; i<10;i++)
             System.out.println(aray[i] + " ");
            System.out.println("a = " + a + "  b = " + b);
            c=b/a;
        } catch (ArithmeticException e)
        {
         System.out.println("\nInteger Division by 0, e = " + e);
        } catch (ArrayIndexOutOfBoundsException ae)
        {
System.out.println("\nArray Index Out of Bounds, ae = " + ae);
        }
           System.out.println("End of the Program ");
         }
    }
```

Output:

```
C:\ >javac MultiCatchExample.java

C:\>java MultiCatchExample
The Array elements are :
4
5
6
7

Array Index Out of Bounds, ae =
java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds
for length 4
End of the Program
```

6. Class Throwable

- The class is declared as

```
public class Throwable extends Object
        implements serializable
```

The class Throwable is super class to all the error and exception classes.

- The program code can throw only objects of this class or objects of its subclasses.
- Only the object of Throwable class or its subclasses can be the arguments of catch clause.
- It would be a bad programming practice to throw Throwable or make Throwable as argument of catch blocks because it will hide desirable information.

Table 11.2 Constructors of class Throwable

Constructor	Description
Throwable ()	Constructs a new throwable with no message
Throwable(String message)	Constructs a throwable with specified message
Throwable(String message, Throwable cause)	Constructs a throwable with specified method and cause
Throwable(Throwable cause)	Constructs a throwable with specified cause
protected Throwable(String message, Throwable cause, Boolean enablesuppression)	Constructs a throwable with specified message, cause, enabled/disabled suppression, and enabled/disabled stack trace

Methods in Throwable Class:

void addSuppressed (Throwable exception)

Throwable fill InStackTrace()

Throwable getCause()

String getMessage ()

String getLocalizedMessage()

StackTraceElement[] getStackTrace()

Throwable [] get Suppressed()

Throwable initCause (Throwable cause)

void printStackTrace()

void printStackTrace(printStream strm)

void printStackTrace(printwriter strm)

void setStackTrace(

StackTraceElement[] stackTrace)

String toString()

Example: TryCatchDemo3.java

```
class TryCatchDemo3 extends Throwable
{
   public static void main(String args[])
   {
      int i=6, j=0, k;
      String str;
      TryCatchDemo3 ct = new TryCatchDemo3();
      try {
           System.out.println("Entered in try block");
           k=i/j;
              System.out.println("Entered in try block");
      } catch (ArithmeticException e)
      {
      System.out.println("Entered in catch block");
      System.out.println("ct.getStackTrace() = " +
ct.getStackTrace());
      System.out.println("Exception e = " + e);

      } finally{
      System.out.println("It is finally block");
      System.out.println("The class is : " + ct.getClass());
      }

     System.out.println("End of the Program ");
   }
}
```

Output:

```
C:\1. JAVA\UNIT-4.2>javac TryCatchDemo3.java

C:\1. JAVA\UNIT-4.2>java TryCatchDemo3
Entered in try block
Entered in catch block
ct.getStackTrace() = [Ljava.lang.StackTraceElement;@452b3a41
Exception e = java.lang.ArithmeticException: / by zero
It is finally block
The class is : class TryCatchDemo3
End of the Program
```

7. Unchecked Exceptions

- The unchecked exception classes are subclasses of class
- RuntimeException, which is further a subclass of class Exception.
- The runtime exceptions are not checked by the compiler.
- These exceptions form a subgroup of classes that are subclasses of class Exception.
- It is not checked by the compiler whether the programmer has handled them or not.

Table 11.4 Unchecked subclasses of RuntimeException

Name of class	*Description of exception condition*
ArithmeticException	Exceptional arithmetic condition such as division by zero
ArrayIndexOutOfBoundsException	Exception arises when out of bound index number is used
ArrayStoreException	Exception arises when we try to use an object of wrong *type* in an array
ClassCastException	Exception condition arising out of invalid *type* object
IllegalArgumentException	Exception arises due to the use of an invalid argument in a method like placing a negative number as argument of sqrt()
IllegalStateException	Exception due to incorrect state of an application
IllegalThreadStateException	Exception arises when the operation attempted is not compatible with the thread state
IndexOutOfBoundsException	When index number used for array, vector, or string element is out of bounds
NegativeArraySizeException	Exception due to the use of negative number for size of array
NumberFormatException	Exception due to invalid conversion of string into number
NullPointException	Exception due to the use of object with null reference
SecurityException	When the code does an illegal operation that violates security
TypeNotPresentException	When the specified *type* is not found
UnsupportedOperationException	Exception due to unsupported operation

Example: UnCheckedDemo.java

```
class UnCheckedDemo
{
   public static void main(String args[])
   {
      int a =12,b = 0, c = 14;
      int sum, d;
      sum = a+b+c;
      System.out.println("Sum = " +sum);

      try {
            System.out.println("Entered try block");
            d=a/b;
            System.out.println("d = " + d);
            System.out.println("Exiting try block");
      }
      catch (ArithmeticException e)
      {
       System.out.println("Do not divide by Zero, Exception =" + e);
```

```
        }
        System.out.println("End of the Program ");
    }
}
```

Output:

```
C:\>javac UnCheckedDemo.java

C:\>java UnCheckedDemo
Sum = 26
Entered try block
Do not divide by Zero, Exception =java.lang.ArithmeticException: /
by zero

End of the Program
```

8. Checked Exceptions

Checked exceptions are direct subclasses of the **Exception** class and are not subclasses of the class RuntimeException. These are called so because **the compiler ensures (checks)** that the methods that throw checked exceptions deal with them

Table 11.5 Checked Exceptions

Exception	*Description*
`ClassNotFoundException`	Exception because class is not found
`FileNotFoundException`	Exception if the file does not exist
`CloneNotSupportedException`	If an object does not implement Cloneable interface, an attempt to clone will result in this exception
`IllegalAccessException`	Attempt to access a class not permitted access
`InstantiationException`	Attempt to create objects of an abstract class or of an interface will result in this exception
`InterruptedException`	When one thread is interrupted by another thread
`NoSuchFieldException`	Request for a nonexistent field
`NoSuchMethodException`	Request for a nonexistent method

Example: MyThread.java

```
        class MyThread
        {
           public static void main(String args[])
           {
            MyThread obj = new MyThread();
            obj.method2();
            System.out.println("End of the Program ");
           }

           public static void method2()
           {
            Thread t1 = new Thread();
            Thread t2 = new Thread();
            try{
            System.out.println("Entered try block");

            t2.sleep(200);
            System.out.println("Square   root   of   64   =   "
        +Math.sqrt(64));
            t1.sleep(1000);
            t2.notify();

            System.out.println("Exiting try block");
            }catch (InterruptedException ie)
            {
                System.out.println("Exception =" + ie);
            }
           }
}
```

Output:

```
C:\ >javac MyThread.java

C:\ >java MyThread
Entered try block
Square root of 64 = 8.0
Exception in thread "main" java.lang.IllegalMonitorStateException
        at java.lang.Object.notify(Native Method)
        at MyThread.method2(MyThread.java:20)
        at MyThread.main(MyThread.java:6)
```

9. try-with-resources

- Generally, the resources such as file, data bases, or Internet connections that are opened in try block are closed in finally block, which is always executed.
- This makes a sure-shot method for recovery of sources and memory.
- It is seen that programmers who otherwise are adept at opening sources such as data bases and files often forget to close these sources when their relevance is over.
- These heavyweight sources keep lurking in JVM, and thus, overloading JVM and creating problems in memory allocation.
- In order to redress the problem, Java 7 has added AutoCloseable.

Example: AutoCloseExample.java

```
class SourceX implements AutoCloseable
{
   @Override
   public void close() throws Exception
   {
      System.out.println("The SourceX is closed");
   }
}

public class AutoCloseExample
{
   public static void main(String args[])
   {
      try  {
           System.out.println("Entered in Try Block");
           SourceX objS = new SourceX();
           objS.close();
           System.out.println("Exit from Try Block");
      } catch (Exception ex)
      {
          System.out.println("Exception ex = " + ex);
      }

     System.out.println("End of the Program ");
   }
}

Output:
C:\>javac AutoCloseExample.java

C:\>java AutoCloseExample
Entered in Try Block
The SourceX is closed
Exit from Try Block
End of the Program
```

10. Catching Subclass Exception

If the super class method does not declare any exception, then subclass overridden method cannot declare (or throws) checked exceptions but it can declare (or throws) unchecked exceptions. This is illustrated in the following Programs.

Example1 : SubClass.java - Sub Class cannot throws **Checked** Exception

```
import java.io.*;

class A
{
   void display()
   {
   System.out.println("Super class display method");
   }
}

public class SubClass extends A
{
   //sub class cannot throws Checked Exception
   void display() throws IOException
   {
   System.out.println("Sub class display method");
   }

   public static void main(String[] args)
   {
   A obj1 = new A();
   obj1.display();

   SubClass obj2 = new SubClass();
   obj2.display();
   }
}
```

Output:

```
C:\>javac SubClass.java
SubClass.java:13: error: display() in SubClass cannot override
display() in A
        void display() throws IOException
             ^
  overridden method does not throw IOException
1 error
```

Example2 : SubClass.java - Sub Class can throws **Unchecked** Exception

```
import java.io.*;

class A
{
      void display()
      {
      System.out.println("Super class display method");
      }
}

public class SubClass2 extends A
{
      //sub class can throws UnChecked Exception
      void display() throws ArithmeticException
      {
      System.out.println("Sub class display method");
      }

      public static void main(String[] args)
      {
      A obj1 = new A();
      obj1.display();

      SubClass2 obj2 = new SubClass2();
      obj2.display();
      }
}
```

Output:

```
C:\>javac SubClass2.java

C:\>java SubClass2
Super class display method
Sub class display method
```

11. Custom Exceptions

- It is also called as user defined exception.
- A programmer may create his/her own exception class by extending the exception class and can customize the exception according to his/her needs.
- Using Java custom exception, the programmer can write their own exceptions and messages.

Example: ExceptionEx.java

```
class UserException extends Exception
{
      UserException (String Message)
      {
       super(Message);
      }
}

class ExceptionEx
{
  public static void main(String args[])
  {
   byte a = 4, b = 9;
   try
   {
   if(a/b== 0)
   throw new UserException("It is integer division.");
   }catch (UserException Ue)
   {
    System.out.println("The exception has been caught.");
    System.out.println(Ue.getMessage());
   }
 }
}
Output:

C:\>javac ExceptionEx.java

C:\>java ExceptionEx
The exception has been caught.
It is integer division.
```

12. Nested try and catch Blocks

- In nested try–catch blocks, one try–catch block can be placed within another try's body.
- Nested try block is used in cases where a part of block may cause one error and the entire block may cause another error.
- In such cases, exception handlers are nested.
- If a try block does not have a catch handler for a particular exception, the next try block's catch handlers are inspected for a match.
- If no catch block matches, then the Java runtime system handles the exception.

```
try{// main try block
    //statements
    try {   // try block 1
    //statements
    try {   // try block 2
    /* statement */}
    catch(Exception  e1)
    {// statements. Innermost catch block}
    }
    catch(Exception e2)
    {/*statements. Middle catch block.*/}
    }
catch()
    {/* Statements. outer most catch block*/}
```

Example: NestedTry.java

```
class NestedTry
{
  public static void main (String args[])
  {
    int [] array = {6, 7};
    //outer try block
    try {
       System.out.println("Entered outer try block.");
       // inner try block
       try {
         System.out.println("Entered inner try block.");
         for (int i = 0; i<= 2; i++)
          System.out.println("Array Element["+i+"]= " + array[i]);
          System.out.println ("Exiting try block.");
         }
```

```
           // inner catch block
           catch (ArrayIndexOutOfBoundsException e)
            {
        System.out.println("ArrayIndexOutOfBoundsException caught");
        System.out.println("Exiting inner catch block.");
              }

       int n = 5, j=2 ;
          for (j =2; j>=0; j--)
             System.out.println ("n/j= " + n/j);
         }
       // outer catch block
          catch (ArithmeticException E)
          {
            System.out.println ("Arithmetic Exception caught");
          }
  }
}
```

Output:

```
C:\>javac NestedTry.java

C:\>java NestedTry
Entered outer try block.
Entered inner try block.
Array Element[0]= 6
Array Element[1]= 7
ArrayIndexOutOfBounds Exception caught
Exiting inner catch block.
n/j= 2
n/j= 5
Arithmetic Exception caught
```

13. Rethrowing Exception

- An exception may be thrown and caught and also partly dealt within a catch block, and then, rethrown.
- In many cases, the exception has to be fully dealt within another catch block, and throw the same exception again or throw another exception from within the catch block.
- When an exception is rethrown and handled by the catch block,
  - the compiler verifies that the type of re-thrown exception is meeting the conditions that the try block is able to throw
  - and there are no other preceding catch blocks that can handle it.

Example: Rethrow.java

```
class Rethrow
{
  public static void main (String Str[])
  {
   int[] array = {6, 7};
   try {
   System.out.println ("Entered inner try block.");
   for (int i = 0; i<= 2; i++)
      System.out.println ("Array Element["+i+"]= " + array[i]);
   System.out.println ("Exiting try block.");
   }catch (ArrayIndexOutOfBoundsException e)
   {
 System.out.println ("ArrayIndexOutOfBounds Exception caught");
 System.out.println ("Throwing e and exiting inner catch block.");
 throw e;
   }
  }
}
```

Output:

```
C:\ >javac Rethrow.java

C:\ >java Rethrow
Entered inner try block.
Array Element[0]= 6
Array Element[1]= 7
ArrayIndexOutOfBounds Exception caught
Throwing e and exiting inner catch block.
Exception              in              thread              "main"
java.lang.ArrayIndexOutOfBoundsException: Index 2 out of bounds
for length 2
        at Rethrow.main(Rethrow.java:9)
```

14. Throws Clause

The **throws clause** specifies that the method can throw an exception. **throws** is a keyword.

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

We can throw either checked or unchecked exception

Syntax of java throws

```
return_type method_name() throws exception_class_name
{
//method code
}
```

Example: Computation.java

```
class MyException extends Exception
{
      MyException(String Message)
      {
      super (Message);
      }
}

public class Computation
{
      public void Compute() throws MyException
      {
         int n = 5, m = 2, z, i ;
         for (i = 0; i<5; i++)
         {
            if ((m-i) == 0)
                   throw new MyException ("Denominator is zero.");
               System.out.println ("z = n/(m-i) = " + n/(m-i));
            }
        }
        public static void main (String args[])
        {
         try
            {
            new Computation ().Compute();
         }catch (MyException me)
         {
            System.out.println("Division by zero exception caught.\n
me =" + me);
            }
      }
}
```

Output:

```
C:\ >javac Computation.java

C:\ >java Computation
z = n/(m-i) = 2
z = n/(m-i) = 5
Division by zero exception caught.
 me =MyException: Denominator is zero.
```

UNIT-5

I. String Handling in Java:

1. Introduction
2. Interface Char Sequence
3. Class String
4. Methods for Extracting Characters from Strings
5. Methods for Comparison of Strings
6. Methods for Modifying Strings
7. Methods for Searching Strings
8. Data Conversion and Miscellaneous Methods
9. Class String Buffer
10. Class String Builder.

II. Multithreaded Programming:

1. Introduction
2. Need for Multiple Threads
3. Thread Class
4. Main Thread- Creation of New Threads
5. Thread States
6. Thread Priority-Synchronization
7. Deadlock and Race Situations
8. Inter-thread Communication - Suspending
9. Resuming and Stopping of Threads.

III. Java Database Connectivity:

1. Introduction
2. JDBC Architecture
3. Installing MySQL and MySQL Connector
4. JDBC Environment Setup
5. Establishing JDBC Database Connections
6. ResultSet Interface
7. Creating JDBC Application
8. JDBC Batch Processing
9. JDBC Transaction Management

I. String Handling in Java:

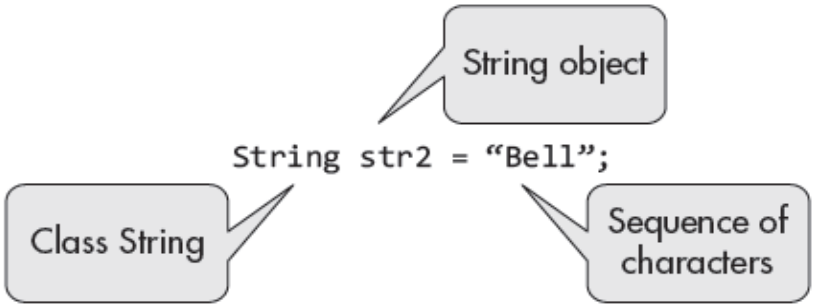
1. Introduction

- Strings are treated differently in Java compared to C language;
- In the latter case, it represents an array of characters terminated by null character.
- **In Java, a string is an object of a class**, and **there is no automatic appending of null character by the system.**
- In Java, there are three classes that can create strings and process them with nearly similar methods.
  - (i) class String
  - (ii) class StringBuffer
  - (iii) class StringBuilder
- All the three classes are part of java.lang package.
- All the three classes have several constructors that can be used for constructing strings.
- In the case of **String** class, an object may be created as

String str1 = "abcd";

Here :

- String is a Predefined class of **`java.lang`** package
- Str1 is an object not a variable.
- "abcd" is string literal



- The aforementioned two declarations are equivalent to the following:

```
char s1[] = {'a', 'b', 'c', 'd'};
char s2[] = {'B', 'e', 'l', 'l'};
```

Storage of Strings

- The memory allocated to a Java program is divided into two segments:
  - (i) Stack
  - (ii) Heap
- The objects of class String have a special storage facility, which is not available to objects of other two String classes or to objects of any other

class.

- The variables are stored on heap, whereas the program is stored on stack.
- Within the heap, there is a memory segment called 'String constant pool'.
- The String class objects can be created in two different ways:

```
String strx = "abcd";
String strz = new String("abcd");
```

Example: StringTest.java

```
public class StringTest
{
   public static void main(String args[])
   {
      String strx = "abcd"; //Object stored in pool
      String stry = "abcd";  // only one "abcd" exists in the pool
      String strz = new String("abcd"); //new object
      String str1 = new String("abcd"); // new object
      String s2 = new String(); // empty string
      String s1 =""; //empty string - no space in the string

     System.out.println("Are reference of strx and stry same?  " +
(strx == stry));
     System.out.println("Are reference of strx and strz same?  " +
(strx == strz));
     System.out.println("Are reference of str1 and strz same?  " +
(str1 == strz));
     System.out.println("Are reference of s1 and s2 same?  " + (s1
== s2));
     System.out.println("Are strings in strx and strz equal?  " +
strx.equals(strz));
   }
}
```

Output:

```
C:\>javac StringTest.java

C:\>java StringTest
Are reference of strx and stry same?  true
Are reference of strx and strz same?  false
Are reference of str1 and strz same?  false
Are reference of s1 and s2 same?  false
Are strings in strx and strz equal?  true
```

Immutability

i. String class

- **The string objects created by class String are immutable**.
- By immutable implies that once an object is created, its value or contents cannot be changed.
- Neither the characters in the String object once created nor their case (upper or lower) can be changed.
- New String objects can always be created and assigned to older String object references.
- Thus, when you change the content of a string by defining a new string, the old and new remain in the memory.
- The immutable objects are **thread safe** and so are the String objects.

ii. StringBuffer class

- **The objects created by class StringBuffer are mutable.**
- These are stored on the heap segment of memory outside the String constant pool.
- The contents of StringBuffer strings may be changed without creating new objects.
- The methods of StringBuffer are synchronized, and hence, **they are thread safe**.

iii. StringBuilder class

- **The objects of class StringBuilder are also mutable but are not thread safe.**
- The operations are fast as compared to StringBuffer and there is no memory loss as is the case with String class.
- The class StringBuilder has the same methods as the class StringBuffer.
- Therefore, if multithreads are not being used, then StringBuilder class should be used to avoid memory loss.

Properties of String, StringBuffer, and StringBuilder objects

Table 12.1 Properties of String, StringBuffer, and StringBuilder objects

Property	*String*	*StringBuffer*	*StringBuilder*
Storage area	Constant string pool and heap	Heap	Heap
Object mutability	Not mutable	Mutable	Mutable
Thread safety	Safe	Safe	Not safe
Performance	Fast	Slow	Fast

2. Interface CharSequence

- It is an interface in java.lang package.
- It is implemented by several classes including the classes String, StringBuffer, and StringBuilder.

It has the following four methods.

(i) char charAt(int index): The method returns character value at specified index value.

(ii) int length(): This method returns the length of this (invoking) character sequence.

(iii) CharSequence subsequence(int startIndex, endIndex):
The method returns a subsequence from start index to end index of this sequence Throws IndexOutOfBoundsException.

(iv) String to String(): The method returns a string containing characters of the sequence in the same.

Example: CharSq.java

```
public class CharSeq
{
   public static void main(String args[])
   {
      CharSequence csq1 = "ABCD";
      System.out.println("Second  letter  in  csq1  =  "  +
csq1.charAt(1));
      System.out.println("Length of csq1 = " + csq1.length());

      CharSequence csq2 = new StringBuffer("XYZ12345");
      CharSequence csq3 = new StringBuffer("Amrit");
      CharSequence  sq  = csq2.subSequence(2,6);

      System.out.println("The csq3 = " + csq3);
      System.out.println("The csq2 = " + csq2);
System.out.println("The sub sequence(2,6) of csq2  sq = " + sq);
   }
}
```

```
Output:

C:\>javac CharSeq.java

C:\>java CharSeq
Second letter in csq1 = B
Length of csq1 = 4
The csq3 = Amrit
The csq2 = XYZ12345
The sub sequence(2,6) of csq2  sq = Z123
```

3. Class String

- The class String is used to represent strings in Java. It is declared as

```
public final class String
extends Object
implements serializable, comparable<String>, charSequence
```

- The String class is final, it cannot be extended.
- Following are the constructors of class String:

1. ***String()***

- This constructor creates a string without any characters. See the following examples.

```
String str = new String();
String str1 = "";
```

2. ***String (byte* [] *barray)***

- It constructs a new string by decoding the specified byte[] barray by using a computer's default character set. The following code
- constructs str2 = "ABCDE".

```
byte []bray = new byte[]{65, 66, 67, 68, 69};
String str2 = new String(bray);
```

Example: StringArray.java

```
public class StringArray
{
   public static void main(String args[])
   {
      byte []bray = new byte[]{65, 66, 67, 68, 69};
      String str2 = new String(bray);

      System.out.println("str2 =" + str2);
   }
}
```

```
Output:
C:\>javac StringArray.java

C:\>java StringArray
str2 =ABCDE
```

3. ***String (byte* [] *barray, Charset specifiedset)***

- It constructs a new string by decoding the specified byte array (bray) by using specified character set.
- Some of the Charsets supported by Java are UTF8, UTF16, UTF32, and ASCII.
- These may be written in lower case such as utf8, utf16, utf32, and ascii.

Example: StringUnicode.java

```
public class StringUnicode
{
   public static void main(String args[])
   {
      byte []unicode = new byte[]{'\u0041','\u0042','\u0043'};
      String str = new String(unicode);

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

Output:

```
C:\>javac StringUnicode.java

C:\>java StringUnicode
str =ABC
```

4. ***String(byte*[] *bray, int offset, int length, String charsetName)***

- The constructor constructs String object by decoding the specified part of byte array using specified Charset.
- For example

  `String str4 = new String(bray,1, 3, “ascii”);`

Example: StringArray2.java

```
public class StringArray2
{
   public static void main(String args[]) throws Exception
   {
      byte []bray = new byte[]{65, 66, 67, 68, 69};
      String str = new String(bray,1,3,"ascii");

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

```
Output:
C:\>javac StringArray2.java

C:\>java StringArray2
str =BCD
```

5. *String (byte*[] *barray, string charsetName}*

- The constructor constructs a string by decoding the byte array using specified Charset.
  `String str3 = new String(bray, "UTF8");`
- The method throws UnsupportedEncodingException if the given charset is not supported.

Example: StringUnicode2.java

```
public class StringUnicode2
{
   public static void main(String args[]) throws Exception
   {
      byte []unicode = new byte[]{'\u0041','\u0042','\u0043'};
      String str = new String(unicode, "UTF8");

      System.out.println("str =" + str);
   }
}
C:\>javac StringUnicode2.java

C:\>java StringUnicode2
str =ABC
```

4. Methods for Extracting Characters from Strings

Table 12.2 Methods for extraction of characters and substrings

Method	*Description*
char charAt(int n)	Extracts a character at a specified index position
void getChars(args)	Extracts more than one character at a time
byte[] getBytes()	Encodes a given string into a sequence of bytes and returns an array of bytes
int length()	Finds out the length of a given string
char toCharArray()	Converts the string into an array of characters and returns the array

The signatures of the first two methods are as:

1. ***char charAt (int n),*** where *n* is a positive number, which gives the location of a character in a string.

- It returns the character at the specified index location.

- Thus, in String ‘Delhi’ the index value of ‘D’ is 0, index value of ‘e’ is 1, ‘h’ is 3, and so on.

2. void getChars (int SourceStartindex, int SourceEndindex, char targetarray[], int targetindexstart)

- This method takes characters from a string and deposits them in another string.
- int *SourceStartindex* and int Source*Endindex* relate to the source string.
- char *targetarray*[] relates to the array that receives the string characters. The int *targetindexstart* is the index value of target array.

Example: MethodsChar.java

```
public class MethodsChar
{
   public static void main(String args[])
   {
      String str1 = "Delhi";
      String str2 = "Vijayawada";
      String str = "I am going to Delhi and from there to Mumbai";

      int begin =14;
      int end =19;
      char aSTR[] = new char[end -begin];
      str.getChars(begin, end, aSTR, 0);

      System.out.println("Length of string str1 =" + str1.length());
      System.out.println("Fourth Character in Delhi = " +
str1.charAt(3));
      System.out.println("Fifth Character in Vijayawada = " +
str2.charAt(4));
      System.out.print("aSTR = ");
      System.out.println(aSTR);
   }
}
Output:
C:\>javac MethodsChar.java

C:\>java MethodsChar
Length of string str1 =5
Fourth Character in Delhi = h
Fifth Character in Vijayawada = y
aSTR = Delhi
```

5. Methods for Comparison of Strings

Table 12.3 Methods for comparison of strings

Method	*Description*
`int compareTo (String str)`	Compares the invoking string with argument string lexicographically. It returns a value less than 0, 0, or more than 0 if the invoking string is either less than, equal, or more than argument string, respectively.
`int compareToIgnoreCase (String str)`	Compares the invoking string with argument string lexicographically ignoring the case differences. The outputs are as described for `compareTo`.
`boolean equals()`	Compares the characters inside the two strings. Returns true if the match occurs, otherwise false. It is different from operator == in which case only the references of the two strings are compared.
`boolean equalsIgnoreCase()`	Compares the strings ignoring the case of letters
`boolean regionMatches(String str)`	Compares a specified region in one string with a specified region in another string. Returns true if the two match, otherwise returns false.
`endsWith(String str)`	Returns true if a string ends with the given substring
`startsWith(String str)`	Returns true if a string starts with the given substring

- The operator == simply compares the references. E.g.

  if (str1==str2)
- The contents are compared by the method equals() as

  if (str1.equals(str3))

Example: MethodCompare.java **compareTo() method**

```
public class MethodCompare
{
   public static void main(String args[])
   {
      String str1 = "AA";
      String str2 = "ZZ";
      String str3 = new String(str1);

      System.out.println("str2.compareTo(str1) =" +
str2.compareTo(str1));
      System.out.println("str1.compareTo(str2) =" +
str1.compareTo(str2));
      System.out.println("str3.compareTo(str1) =" +
str3.compareTo(str1));
   }
}
```

Output:

```
C:\>javac MethodCompare.java

C:\>java MethodCompare
str2.compareTo(str1) =25
str1.compareTo(str2) =-25
str3.compareTo(str1) =0
```

Example: StringMethods.java - equals() and length() methods

```
public class StringMethods
{
   public static void main(String args[])
   {
      String str1 = "AA";
      String str2 = "ZZ";
      String str3 = new String(str1);

      System.out.println("str3 = " + str3);

      System.out.println("str2.equals(str1) =" + str2.equals(str1));

      System.out.println("str1.equals(str2) =" + str1.equals(str2));
      System.out.println("str3.equals(str1) =" + str3.equals(str1));

      System.out.println("Length of str2 =" + str2.length());
   }
}
```

Output:

```
C:\>javac StringMethods.java

C:\>java StringMethods
str3 = AA
str2.equals(str1) =false
str1.equals(str2) =false
str3.equals(str1) =true
Length of str2 =2
```

6. Methods for Modifying Strings

- The ways to modify strings are as follows:
- 1. Define a new string.
- 2. Create a new string out of substring of an existing string.
- 3. Create a new string by adding two or more substrings.
- 4. Create a new string by replacing a substring of an existing string.
- 5. Create a new string by trimming the existing string.

Table 12.4 Methods for modifying strings

Method	*Description*
`substring(int indexbegin)`	Creates new string as a substring of existing string starting from *indexbegin* to end of string
`substring (int indexbegin, int indexend)`	Creates new string that has characters from *indexbegin* to *indexend* of the existing string
`String concat (String S)`	Creates a new string by concating the string S with the existing string. It returns a new String object that contains characters from both the String objects
`String replace(char current, char replacement)`	Returns string in which all occurrences of a character are replaced with another character
`String replace(Char Sequencecurrent, Char Sequence replacement)`	Returns string in which all occurrences of a sequence of characters is replaced with another sequence of characters
`String trim()`	Trims/removes the white spaces at the beginning and the end of current string to create a new string. Original string is obtained if there is no white space characters at the beginning or end of string

Example: ModifyString.java -replace() and substring() methods

```
public class ModifyString
{
   public static void main(String args[])
   {
      String str1 = "Belhi";
      String mstr1 = str1.replace('B', 'D');
      System.out.println("Before Modification str1 = " + str1);
      System.out.println("Modified string mstr1 = " + mstr1);

      String str2 = "       WELCOME      ";
      System.out.println("str2 =" + str2);
      String mstr2 = str2.trim();
      System.out.println("mstr2 =" + mstr2);

      String str3 = "I am going to Delhi and from there to Mumbai";

      String mstr3 = str3.substring(0,19);
      System.out.println("mstr3 =" +  mstr3);

      String mstr4 = str3.substring(19);
      System.out.println("mstr4 =" +  mstr4);
   }
}
```

Output:

```
C:\ >javac ModifyString.java

C:\ >java ModifyString
Before Modification str1 = Belhi
Modified string mstr1 = Delhi
str2 =       WELCOME
mstr2 =WELCOME
mstr3 =I am going to Delhi
mstr4 = and from there to Mumbai
```

7. Methods for Searching Strings `indexOf()`

(i) *`int indexOf(int character/substring)`* - searches for the first occurrence of a specified character in the string.

(ii) `int indexOf(int` *`character/substring`*`, int` *`index`*`)` - searches for the first occurrence of a specified character or substring and the search starts from the specified index value, that is, the second argument.

Example: StringSearch.java

```
public class SearchString
{
      public static void main (String args[])
      {
            String str1 = "Help the needed ones";
            String str2 = "One has to take care of one's health
oneself";
            System.out.println("The index of \"e\" in the String str1
is at index = " + str1.indexOf('e'));
            System.out.println ("The last index of \"e\" in str1 is
at index = " + str1.lastIndexOf('e'));
            System.out.println ("The last occurrence  \"of\" in
String str2 is at index = " + str2. lastIndexOf("of"));
            System.out.println("The occurrence of \"e\" after index 8
in str1 = " + str1.indexOf('e', 8));
            System.out.println("The index of last occurrence of \"n\"
= " + str1. lastIndexOf('n', 15));
      }
}
```

Output:

```
E:\ >javac SearchString.java

C:\>java SearchString
The index of "e" in the String str1 is at index = 1
The last index of "e" in str1 is at index = 18
The last occurrence  "of" in String str2 is at index = 21
The occurrence of "e" after index 8 in str1 = 10
The index of last occurrence of "n" = 9
```

8. Data Conversion and Miscellaneous Methods

Table 12.5 Some important remaining methods of class `String`

Method	*Description*
`int codePointAt(int index)`	Returns code point (character) at specified index
`int codePointBefore(int index)`	Returns code point of character before the specified index value
`int codePointCount(int indexStart, int indexEnd)`	Returns the code points in the range specified by indexStart and indexEnd
`boolean contains(CharSequence cseq)`	Returns true if the string contains the specified CharSequence
`boolean contentEquals(CharSequence cseq)`	Compares the invoking string with specified CharSequence and returns true if equal
`boolean contentEquals(BufferString bufstr)`	Compares the invoking string with specified `StringBuffer` string and returns true if equal
`static String copyValueOf(char [] chary)`	Returns a string that represents the character sequence in the char array
`static String copyValueOf(char [] chary, int offset, int count)`	Returns a string that represents the character sequence in the specified part of the array
`static String format(Locale loc, String format, Object ...arguments)`	Returns the formatted string according to the specified format and other objects
`byte [] getbytes()`	Converts the invoking string into an array of bytes using the default Charset of the computer and stores the result into a new byte array
`byte [] getbytes(Charset cset)`	Encodes the string into a sequence of bytes by using the given Charset and returns byte array and stores the result into a new byte array

Example: StringValue.java

```
public class StringValue
{
      public static void main(String[] args)
      {
      int n = 70;
      long l= 25674;
      float fit = 45.76f;
      String s1 = new String("Railways");
      String s2 = new String();
      String s3 = s2.valueOf (fit);
      char [] array = {'D', 'e', '1', 'h', 'i'};

      System.out.println("s2.valueOf(n) = " + s2.valueOf(n));
      System.out.println("s2.valueOf(l) = " + s2.valueOf(l));
      System.out.println("s3 = " + s3);
      System.out.println("s2.valueOf(array) = " + s2.valueOf(array));
      System.out.println("s1.toString() = " +s1.toString());
      }
}

C:\>javac StringValue.java

C:\>java StringValue
s2.valueOf(n) = 70
s2.valueOf(l) = 25674
s3 = 45.76
s2.valueOf(array) = De1hi
s1.toString() = Railways
```

9. Class String Buffer

It defines the strings that can be modified as well as the number of characters that may be changed, replaced by another, a string that may be appended, etc.

- The strings are also thread safe. For the strings created by class StringBuffer, the compiler allocates extra capacity for 16 more characters so that small modifications do not involve relocation of the string.

The class is declared as follows:

public final class StringBuffer
extends Object
implements Serializable, CharSequence

Table 12.6 Constructors of `StringBuffer` class

Constructor	*Description*
`StringBuffer()`	Creates an empty buffer string but keeps capacity for 16 characters
`StringBuffer(int capacity)` `throws NegativeArraySizeException`	Sets the size of the buffer string capacity equal to the sum of the argument and 16 characters, so that the string may be expanded without relocation
`StringBuffer (String str)` `throws NullPointException`	Creates and initializes buffer string with the string `str` but it reserves the space for 16 more characters so that it may be modified without relocation
`StringBuffer(CharSequence charSeq)` `throws NullPointException`	Creates `StringBuffer` object that is made out of character sequence

Table 12.7 Frequently used methods of `StringBuffer` class

Method	*Description*
`StringBuffer append(boolean b)`	Appends string representation of argument
`StringBuffer append(char ch)`	Appends string representation of argument
`StringBuffer append(char[] c)`	Appends string representation of argument

Method	*Description*
`StringBuffer append(CharSeq s)`	Appends string representation of argument
`StringBuffer append(double d)`	Appends string representation of argument
`StringBuffer append(float flt)`	Appends string representation of argument
`StringBuffer append(int n)`	Appends string representation of argument
`StringBuffer append(long lng)`	Appends string representation of argument
`StringBuffer append(CharSeq s)`	Appends string representation of argument
`StringBuffer append(String str)`	Appends the string *str* at the end of invoking `StringBuffer` object
`StringBuffer append (CharSequence,int start,int end)` `throws IndexOutOfBoundsException`	Appends a specified subsequence to the invoking sequence
`append (char[] str,int offset,int length)throws IndexOutOfBoundsException`	Appends the string representation of char array to this sequence
`int capacity()`	Returns allocated capacity of invoking string
`char charAt(int index)throws IndexOutOfBoundsException`	Returns value of character at the specified *index* value
`void setCharAt(int index,char c)throws IndexOutOfBoundsException`	Sets a specified character c at the specified index value *index* in invoking string

Example1: StringBufferDemo.java

```
class StringBufferDemo
{
      public static void main (String args[])
      {
      StringBuffer bufStr = new StringBuffer ("Hello World
Example" );
      System.out.println("bufStr = " + bufStr);
      System.out.println("Length of bufStr =" +
bufStr.length());

      bufStr.setLength (11);
      System.out.println("New Length of bufStr = " +
bufStr.length());

      System.out.println("Capacity of bufStr =" +
bufStr.capacity());
      System.out.println("New bufStr =" + bufStr );

      char ch=bufStr.charAt(4);
      System.out.println("Character at 4th position = " + ch);

      bufStr.setCharAt(7, 'e');
      System.out.println(" Now New bufStr =" + bufStr);
      }
}
```

Output:

```
C:\>javac StringBufferDemo.java

C:\>java StringBufferDemo
bufStr = Hello World Example
Length of bufStr =19
New Length of bufStr = 11
Capacity of bufStr =35
New bufStr =Hello World
Character at 4th position = o
 Now New bufStr =Hello Werld
```

Example2: StringBufferDemo2.java

```
class StringBufferDemo2
{
      public static void main (String args[])
      {
      StringBuffer bufStr = new StringBuffer ("Hello World");
      System.out.println("bufStr =" + bufStr);
      bufStr.reverse();

      System.out.println("After reversing bufStr =" + bufStr);
      StringBuffer str = new StringBuffer("Delhi is a city.");

      System.out.println("Before insert, str = " + str);
      str.insert(11, "very big ");
      System.out.println("After insert, str = " + str);

      str.delete (11,16);
      System.out.println("After deletion str = " + str);

      str.replace (15, 21, "market.");
      System.out.println("After replace str = " + str);

      str.deleteCharAt(21);
      System.out.println("After delete dot, str = " + str);

      str.append(" of").append(" electronic goods.");
      System.out.println("After append str = " + str);
      }
}
```

Output:

```
C:\>javac StringBufferDemo2.java

C:\>java StringBufferDemo2
bufStr =Hello World
After reversing bufStr =dlroW olleH
Before insert, str = Delhi is a city.
After insert, str = Delhi is a very big city.
After deletion str = Delhi is a big city.
After replace str = Delhi is a big market.
After delete dot, str = Delhi is a big market
After append str = Delhi is a big market of electronic goods.
```

10.Class String Builder.

The StringBuilder class is the subclass of Object in java.lang package.

This class is used for creating and modifying strings. Its declaration is as follows:

```
public final class StringBuilder extends Object
 implements Serializable, CharSequence
```

The four constructors of the class are described as follows:

1. StringBuilder()—Creates a StringBuilder object with no characters but with initial capacity of 16 characters.

2. StringBuilder(CharSequence *chSeq)*—Creates a StringBuilder object with characters as specified in CharSequence *chSeq*.

3. StringBuilder(int *capacity*)—Creates a StringBuilder object with specified capacity. It throws NegativeArraySizeException.

4. StringBuilder(String str)—Creates a StringBuilder object initialized with contents of a specified string. It throws NullPointException if str is null.

Table 12.8 Methods of `StringBuilder` class

Method	*Description*
`StringBuilder append(boolean b)`	Appends string version of `boolean` argument `b`
`StringBuilder append(char ch)`	Appends string version of `char` argument
`StringBuilder append(char[] str)`	Appends string version of `char` array argument
`StringBuilder append(char [] str, int offset, int length) throws IndexOutOfBoundsException`	Appends string version of `char` subarray to the string
`StringBuilder append(CharSequence ch) throws IndexOutOfBoundsException`	Appends specified `char` sequence to the string
`StringBuilder append(char Sequence s, int start, int end) throwsIndexOutOfBoundsException`	Appends specified part of sequence to string
`StringBuilder append(double dbl)`	Appends string version of `double` argument to string
`StringBuilder append(float flt)`	Appends string version of `float` argument to string
`StringBuilder append(int k)`	Appends string version of `int` argument `k`
`StringBuilder append(long lg)`	Appends string version of `long` argument
`StringBuilder append(String str)`	Appends `String str` to current string
`StringBuilder append(Object obj)`	Appends string version of `Object obj` to string
`StringBuilder append(StringBuffer sbuff)`	Appends specified string to invoking string
`StringBuilder appendCodePoint(int CodePoint)`	Appends string version of `CodePoint` argument to invoking string
`int capacity()`	Returns current capacity of invoking string
`char charAt(int index)throws IndexOutOfBoundsException`	Returns character at specified index value

Example: StringBuildDemo.java

```
public class StringBuildDemo
{
      public static void main(String[] args)
      {
      StringBuilder builder1= new StringBuilder("Delhi");
      System.out.println ("Before append, builder1 = " + builder1);
      builder1.append (-110024);
      System.out.println ("After append, builder1 = " + builder1);

      StringBuilder builder2 = new StringBuilder();
      System.out.println("The length of builder2 = "+
      builder2.length());

      System.out.println("The capacity of builder2 = "+
      builder2.capacity());

      System.out.println("Before append, builder2 = " + builder2);
      builder2.append("New Delhi");
      System.out.println("After append, builder2 = " + builder2);
      }0
}
```

Output:

```
C:\>javac StringBuildDemo.java

C:\>java StringBuildDemo
Before append, builder1 = Delhi
After append, builder1 = Delhi-110024
The length of builder2 = 0
The capacity of builder2 = 16
Before append, builder2 =
After append, builder2 = New Delhi
```

Example: StringBuildDemo2.java

```
import java.lang.StringBuilder;

public class StringBuildDemo2
{
      public static void main(String[] args)
      {
      StringBuilder builder1= new StringBuilder();

      builder1.insert(0,"Java is a programming language");
      System.out.println ("The string builder =" + builder1);

      builder1.insert (10, "object oriented ");
      System.out.println("The new string is =" + builder1);

      builder1.deleteCharAt(8).delete(7,8);
      System.out.println("The new string after delete = " + builder1);

      StringBuilder builder2 = new StringBuilder();
      builder2.insert(0, "Delhi is a big city");
      System.out.println("The builder2 = " + builder2);

      builder2.insert (0,"New ").insert (18, " business").replace (27, 35,
" center.");
      System.out.println("After modification builder2 = " + builder2);
      }
}
```

Output:

```
C:\>javac StringBuildDemo2.java


C:\>java StringBuildDemo2

The string builder =Java is a programming language

The new string is =Java is a object oriented programming language

The new string after delete = Java is object oriented programming language

The builder2 = Delhi is a big city

After modification builder2 = New Delhi is a big business center.
```

II. Multithreaded Programming

1. Introduction

Multithreading is a Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU. Each part of such program is called a thread. So, threads are light-weight processes within a process.

Threads can be created by using two mechanisms :

1. Extending the Thread class
2. Implementing the Runnable Interface

- Some computer **programs may be divided into segments** that can run independent of each other or with minimal interaction between them.
- **Each segment may be considered as an independent path of execution called a thread**.
- If the computer has multiple processors, the threads may run concurrently on different processor thus saving computing time.
- Threads are useful even if the computer has a single-core processor.
- The different processes in a computer take different times.

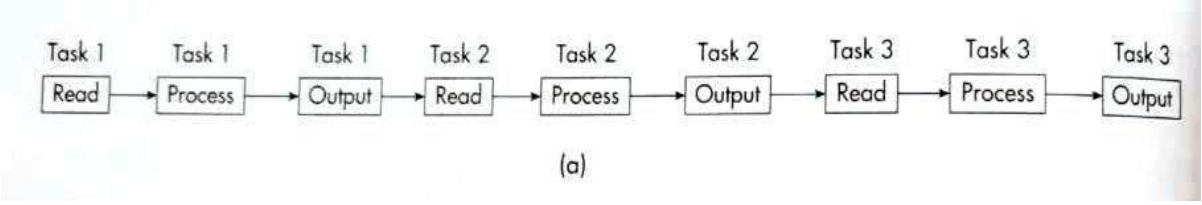
2. Need for Multiple Threads

- The present speed of about 15 GHz is about the upper possible limit because beyond this, the cooling problems are tremendous.
- Further, increase in throughput of computer is possible only by dividing the program into segments that are data dependent and can be processed simultaneously by more than one processor;
- thus, it decreases the total time of computation.
- This is the basis on which supercomputers are built.
- In a supercomputer, thousands of processors are employed to concurrently process the data.
- Hardware developers have gone a step further by placing more than one core processor in the same CPU chip.
- Thus, now, **we have multi-core CPUs.**

Multithreaded Programming for Multi-core Processor

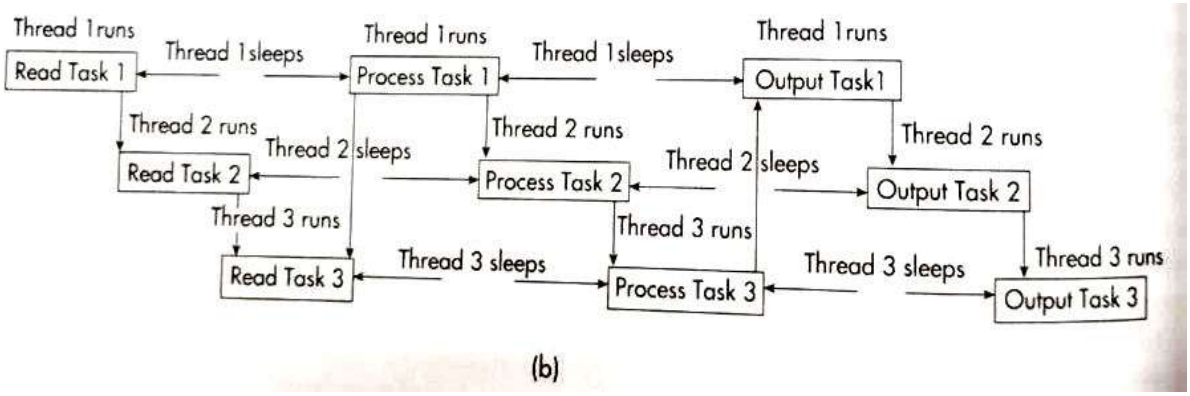
- A CPU may have two cores - dual core or four cores - quad, six cores, or more.
- CPUs having as many as 50 cores have also been developed.
- Moreover, computers with multi-core CPU are affordable and have become part of common man's desktop computer.
- Advancements in hardware are forcing the development of suitable software for optimal utilization of the processor capacity. **Multithread processing is the solution**.
- Multithread programming is inbuilt in Java and CPU capacity utilization may be improved by having multiple threads that concurrently execute different parts of a program.

A. How Single Processor – tasks carried out in Single Sequence is illustrated in the following diagram.



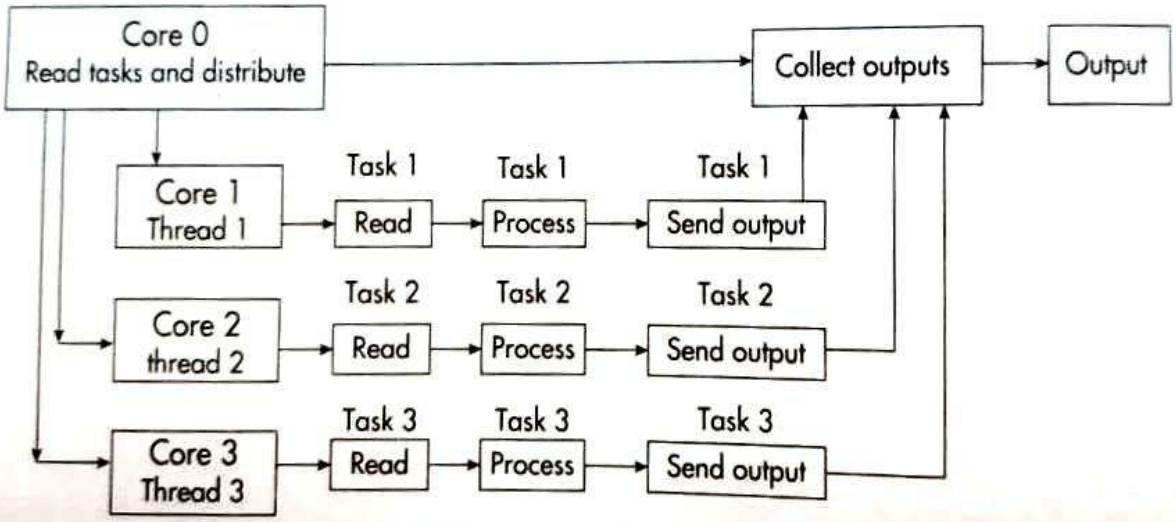
(a)

B. The following diagram shows the Single processor – threads share the time of processor



(b)

C. The following diagram shows the Multi-core processor – threads execute concurrently



(c)

3. Thread Class

- In Java, threads are based on the class Thread belonging to java.lang package, that is, java.lang.Thread.
- A thread is an object of class Thread and can invoke the instance methods defined in the class.
- Even if a thread is not explicitly defined, one thread is automatically created by the system for the execution of the main method. **It is generally called main Thread**.
- A thread does not start working on birth. It has to invoke the start() method that gives it the life, otherwise it is a lifeless thread object.
- After getting life, a thread executes a code of instructions (target) as specified by the user in the overloaded method run().

The Thread class has defined several constructors.

i. **Thread(): ** It allocates a new thread object as thread(null, null, generatedname). Every thread must have a name.

ii. **Thread(String threadname)**: It allocates a thread with name specified by the user. It is of the form thread(nall null, name). A thread may be created as

```
Thread t2 new Thread("MyThread");
```

iii. **Thread(Runnable object)** : The constructor allocates a thread with a specified target. The name by the compiler as `Thread-0`, `Thread-1`, and so on.

iv. **Thread (Runnable object, String threadName)**: Thread is allocated with a specified target and user specified name threadnume.

v. **Thread (ThreadGroupgroup, Runnable object)**: It allocates thread with specified group and target.

vi. **Thread (ThreadGroupgroup, Runnable object, String Threadname)**: The constructor allocates thread with specified thread group, target, and thread name.

Example: ThreadX.java

```
public class ThreadX extends Thread
{
      public void run()
      {
      System.out.println("It is Threadx class");
      }

      public static void main(String args[])
      {
      Thread a= new Thread (new ThreadX(), "FirstThread");
      Thread b= new Thread (new ThreadX());
      System.out.println("Name of a = " + a.getName());
      System.out.println("Name of b = "+ b.getName());
      th.start();
      t1.start();
      }
}
```

Output:

```
C:\ >javac ThreadX.java

C:\ >java ThreadX
Name of a = FirstThread
Name of b = Thread-2
It is Threadx class
It is Threadx class
```

Thread Group

- Every thread is in fact a member of a thread group. The thread groups are useful for invoking methods that apply to all the threads.
- The thread is made a member of a group at the time of its creation with constructor.
- The thread remains the member of the designated group throughout its life.
- It cannot become the member of another group.
- If a thread's group is not specified in its constructor, as is the usual case, the thread is entered into the same group as the thread that created it.
- The default thread group for a newly executing Java application is the main group.
- When a thread dies, its thread group becomes null

Methods of Thread Class

- All threads are objects of class Thread.
- The methods of Thread class for manipulating threads, changing their properties, and understanding their behaviour.
- The class Thread contains several methods that control the execution as well as for setting and getting attributes of threads.
- Methods of Thread class are as follows.

Thread S public void **start()**
public void **run()**
public final boolean **isAlive()**
public final String **getName()**
public static Thread **currentThread()**
public final void **setName(**String name**)**
public static void **yield()**
public static void **sleep** (long milliseconds)
public static void **sleep** (long millisecs, int nanosecs)
public final vold **join()**
public final void **join**(long milliseconds)
public final void **join**(long milliseconds, int nanoseconds)

public final int **getPriority**()
public static int **activeCount**()
public final void **setPriority**(int newpriority)
public long **getID**()
public **Thread.State getState**()
public void **interrupt**()
public static boolean **interrupted**()
public boolean **isInterrupted**()
public final void **checkAccess**()
public static int **enumerate**(Thread [] array)
public String **toString**()
public final boolean **isDaemon**()
public final void **setDaemon**(boolean b)
public static boolean **holdstock**(Object obj)
public final ThreadGroup **getThreadGroup**()

Deprecated Methods of Class Thread

The following methods of class Thread have been deprecated because these are either unsafe or are deadlock pruned.

stop() : The invoking thread stops executing with clean-up, and thus, exposes sensitive resources to other threads.
destroy(): It terminates the thread without clean-up. Deadlock pruned method.
suspend(): It temporarily suspends the execution of thread without clean-up. Deadlock pruned method
resume(): It brings back the suspended thread to runnable state. Used only after suspend().
countStackFrames(): It is not well-defined. Returns number of stack frames in this thread.

Example: ThreadNew3.java

```
public class ThreadNew3 extends Thread
{
  public void run()
  {
      System.out.println("In run() Method ");
      System.out.println("The current Thread = " + this.
currentThread());
      System.out.println("Is present thread daemon Thread:"+
this.isDaemon());

      int n = 10;
      System.out.println("Square root of " + n + " = " +
Math.sqrt(n));
      System.out.println("The active count = "+ this.activeCount());
```

```
  }

   public static void main(String args[])
  {
      Thread t1 = new Thread (new ThreadNew3());
      System.out.println("Is Thread ti alive before Start():  " +
t1.isAlive());
      t1.start();
      System.out.println("Is Thread is alive after start(): " +
t1.isAlive());
      System.out.println("ThreadGroup of t1 = " +
t1.getThreadGroup());
      System.out.println("Name of t1 = " + t1.getName());
      t1.setName("SecondThread");
      System.out.println("New name of t1 = " + t1.getName());
  }
}

C:\ >javac ThreadNew3.java

C:\ >java ThreadNew3
Is Thread ti alive before Start():  false
Is Thread is alive after start(): true
In run() Method
ThreadGroup of t1 = java.lang.ThreadGroup[name=main,maxpri=10]
The current Thread = Thread[Thread-1,5,main]
Name of t1 = Thread-1
Is present thread daemon Thread:false
New name of t1 = SecondThread
Square root of 10 = 3.1622776601683795
The active count = 2
```

4. Main Thread

- When we run a program in Java, one thread is automatically created and it executes the main method. The thread is called main thread. This is the thread from which other threads are created.
- The threads created from the main thread are called child threads.
- A program keeps running as long as the user threads are running.
- The main thread also has the status of a user thread.
- The child threads spawned from the main thread also have the status of user thread.
- Therefore, main thread should be the last thread to finish so that the program finishes when the main method finishes.
- A thread is controlled through its reference like any other object in Java.
- The main thread can also be controlled by methods of Thread class through its reference that can be obtained by using the static method current Thread(). Its signature is

```
Thread thread = Thread.currentThread();
```

- **sleep()** method suspend the execution for the given specified time interval.
  Ex:

```
thread.sleep(500);
```

the avbove statement will suspend the execution of main() method for 500 ms, which is the argument of the method.

- setName() method add the new name to the existing thread
  Ex:

```
thread.setName("My Thread")
```

Illustration of main thread and methods setName() and getName()

Example: MyThread.java

```
class MyThread
{
  public static void main (String args[])
  {
      Thread thread = Thread.currentThread(); //Thread reference
      System.out.println("CurrentThread :" + thread);

      System.out.println("Before modification ,Thread Name =" +
thread.getName());
      System.out.println("Change the name to MyThread.");
      thread.setName("MyThread"); //new name for main thread
      System.out.println("After modification ,Thread Name =" +
thread.getName());
      //try block contains code to be executed by main thread
      try
      {
        for (int i=0; i<4; i++)
         {
              boolean B = thread.isAlive();
```

```
            System.out.println("Is the thread alive? " + B);
            System.out.println(Thread.currentThread()+ " i = "+ i);
            Thread.sleep(1000);
          }
           }catch(InterruptedException e)
        {
         System.out.println("Main thread interrupted");
        }
   }
}

Output:
C:\>javac MyThread.java

C:\>java MyThread
CurrentThread :Thread[main,5,main]
Before modification ,Thread Name =main
Change the name to MyThread.
After modification ,Thread Name =MyThread
Is the thread alive? true
Thread[MyThread,5,main] i = 0
Is the thread alive? true
Thread[MyThread,5,main] i = 1
Is the thread alive? true
Thread[MyThread,5,main] i = 2
Is the thread alive? true
Thread[MyThread,5,main] i = 3
```

5. Creation of New Threads

The new thread is created by creating a new instance of Thread class. The enhancements in Java 8 enables us to create and execute new threads in the following ways

i. By extending Thread class
ii. By implementing Runnable interface: This may be carried out in the following four styles:
    a. Conventional code
    b. Lambda expression
    c. Method reference
    d. Anonymous inner class

i. Creation of New Thread by Extending Thread Class

A thread is an object of class Thread that contains methods to control the behaviour of threads. **A class** that **extends** the class **Thread** inherits all methods and constructors

The procedure followed in this case is given as follows:

1) A class is declared that extends Thread class. Since this is a subclass, it inherits the methods of Thread class.
2) This class calls a constructor of Thread class in its constructor.
3) In this class, the method run() is defined to override the run() method of Thread class. The method run() contains the code that the thread is expected to execute.

4) The object of class the method start() inherited from Thread class for the execution of run().

Example: MyClass.java

```
public class MyClass extends Thread
{
      MyClass()
      {
         super("MyThread"); // constructor of Myclass
         start();
      }

      //The run() method is defined below
      public void run()
      {
        System.out.println("It is MyThread.");
      }

      public static void main(String args[])
      {
          new MyClass(); //creates a new instance of Myclass
      }
}
```

Output:

```
C:\>javac MyClass.java

C:\>java MyClass
It is MyThread.
```

ii. Creation of New Threads by Implementing Runnable interface

The runnable is an interface that has only one method in it, that is

```
public interface Runable( public void run());
```

The full definition of method run() is included in the class. The thread begins with execution of run() and ends with end of run() method. The step wise procedure is given here.

1. Declare a class that implements Runnable interface.
2. Define the run() method in the class. This is the code that the thread will execute.
3. Declare an object of Thread class in main() method.
4. Thread class constructor defines the thread declared with operator new and the Runnable object is passed on to the new thread constructor.
5. Method start() is invoked by the thread object created.

The following Program illustrates a simple example of implementing Runnable.

Example: MyThreadClass.java

```
public class MyThreadClass implements Runnable
{
   public void run()
   {
           System.out.println("This is Runnable implementation.");
   }
   public static void main(String args[])
   {
      Thread Th = new Thread(new MyThreadClass());
         Th.start();
   }
}
```

```
Output

This Is Runnable implementation.
```

iii. Creation of Threads by Lambda Expression, Method Reference, and Anonymous Class

The Lambda expression and method references are simplify the code for creating the thread, as illustrated in the following program.

```
public class ThreadMethods
{
      public static void main(String[] args)
      {
      //Method reference

      new Thread (ThreadMethods::Method1).start();

      //The following line is Lambda expression
      new Thread(() -> Method2()).start();

      //The anonymous inner class or conventional method
```

```
      new Thread(new Runnable()
      {
        public void run()
        { Method3();}
      }).start();

      }

      static void Method1()
      {
        System.out.println("It method reference thread.");
      }
      static void Method2()
      {
        System.out.println("It is Lambda expression method
thread.");
      }
      static void Method3()
      {
        System.out.println("It is conventional method thread.");
      }
}
```

Output:

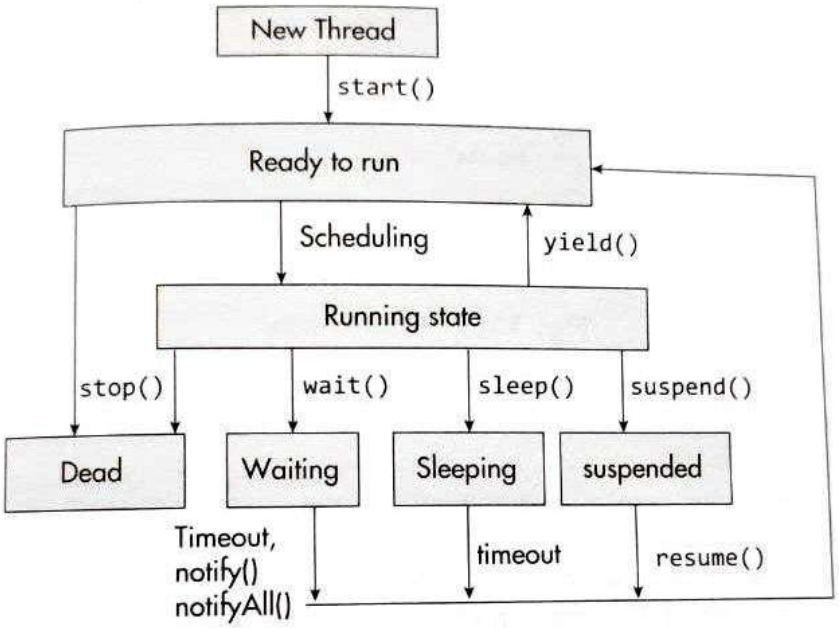
```
C:\>javac ThreadMethods.java

C:\>java ThreadMethods
It method reference thread.
It is Lambda expression method thread.
It is conventional method thread.
```

6. Thread States

Thread states are as follows.

i. New thread state
ii. Ready-to-run state
iii. Running state
iv. Non-runnable state-waiting, sleeping, or blocked state
v. Dead state

The following Figure illustrates the different states of a thread.



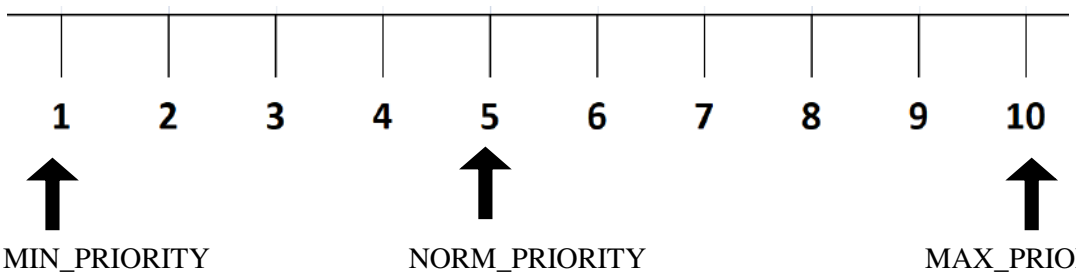
The transition from one state to another is shown in the figure. The different states are as follows

1. New thread state. The thread is just created. It is simply a lifeless object of class Thread.
2. Runnable state: When the thread invokes the method start(), it becomes alive. This state is called runnable state. It is not a running state. It is simply a ready-to-run. The thread has to wait till it is scheduled, that is, selected from the group of threads waiting in running state for dispatch to the CPU by the scheduler
3 Running state. If the scheduler chooses the thread from the waiting list of threads and dispatches it CPU, the thread transits to runnable state, that is, it starts executing the method run() meant for it. This is turning state. If it completes its run() method successfully, its life span is over. The thread is automatically terminated and it goes to **dead state** from which it cannot be revived.
4. Sleep state: The code that a thread is executing may require it to relinquish the CPU for sometime so the other threads can possess CPU. The sleep method may be invoked by the thread. The time period of them is the argument of sleep() method. It is either `long milliseconds`
or `int nanoseconds`. After the sleep the thread returns normally to runnable state.
5. Blocked state. A running thread gets into blocked state if it attempts to execute a task.
6. State wait: A thread gets into wait state on invocation of any of the three methods wait() or wait (long millisees) or wait (long millisecs, int nanosecs).
7. yield. The use of code Thread yield(), is another method besides the sleep for the thread to cease the use of CPU. The thread transits to Runnable state.

8. Suspended. The term belongs to legacy code and is defined in Java 1.1. The suspended thread can be brought back to normal Runnable state only by method resume(). Both the methods suspend() and
resume() are now deprecated, instead one should use wait()and notify().

7. Thread Priority

Thread priority is an important factor among others that helps the scheduler to decide which thread to dispatch the CPU from the group of waiting threads in their runnable state.



All the threads have a priority rating of be When several threads are present, the priority value determines which thread has the chance to possess the Ofest.

The actual allocation is done by the scheduler. Thus, a thread with higher priority has higher chance getting the CPU and also a higher share of CPU time. Keeping equal priority for all threads ensures that each has equal chance to share CPU time, and thus, no thread starves, because when a thread with higher priority enters the Runnable state, the operating system may pre-empt the scheduling by allocating CPU to the thread with higher priority.

When this thread returns from sleep or wait state, the same story may be repeated. When several threads of different priorities are present, it is quite likely that a thread with the lowest priority may not get a chance to possess CPU. This is called starvation.

Thread priority can be changed by method **setPriority (n).**
The priority may be obtained by calling **getPriority()** method.

Example: ThreadPriority2.java

```
class PThread extends Thread
{
      String ThreadName;
      Thread Th;
      int P;
      PThread (String Str, int n)
      {
      ThreadName = Str;
      P=n;
      Th = new Thread(this, ThreadName);
      Th.setPriority(P);
      System.out.println("Particulars of new thread:" + Th);
      }
      public void threadStart()
      {
         Th.start();
      }

      public void run()
      {
```

```
      System.out.println("Priority of new thread: " +
Th.getPriority());
          }
}
public class ThreadPriority2
{
  public static void main (String args[])
  {
      PThread PT1 =  new PThread("First", Thread.MAX_PRIORITY);
      PThread     PT2     =           new     PThread("Second",
Thread.NORM_PRIORITY);
      PThread PT3 =  new PThread("Third", 1);

      PT1.threadStart();
      PT2.threadStart();
      PT3.threadStart();
      try
      {
        Thread.sleep(50);
      } catch(InterruptedException e)
      {
      System.out.println("Main thread sleep interrupted");
      }
  }
}
```

Output:

```
C:\>javac ThreadPriority2.java

C:\>java ThreadPriority2
Particulars of new thread:Thread[First,10,main]
Particulars of new thread:Thread[Second,5,main]
Particulars of new thread:Thread[Third,1,main]
Priority of new thread: 10
Priority of new thread: 5
Priority of new thread: 1
```

8. Synchronization

Synchronization in java is the capability *to control the access of multiple threads to any shared resource*. Java Synchronization is better option where we want to allow only one thread to access the shared resource.

In several multithreaded programs, the different threads are required to access the same resource, for example, a memory object during the execution. One thread may attempt to update it, while the other wants to read it and a third may try to alter the content of the memory.

Such a situation may give rise to **race condition** and the result may be quite different if the sequence of operations actually followed is different from the desired one.

Proper execution requires that at any point of time, only one thread be allowed to use the critical resource till it finishes its task.

We are all aware of the computerized banking system. In one branch, money may be deposited in your account, while you are withdrawing money at another branch. There can be a problem if one thread has partly finished its task, while the other gets in. It will corrupt the data.

Synchronization solves this problem by allowing only one thread can access the resource. The second thread should be allowed only when the first has finished its task.

Example: SyncDemo.java

```
class Counter
{
      int count=0;
      public  void increment()
      {
        count++;
      }
}

public class SyncDemo
{
  public static void main(String[] args) throws Exception
  {
      Counter c = new Counter();

      Thread t1 = new Thread(new Runnable()
      {
        public void run()
        {
                  for(int i=1;i<=5000; i++)
            {
               c.increment();
```

```
            }
        }
      });

      Thread t2 = new Thread(new Runnable()
      {
        public void run()
        {
                  for(int i=1;i<=5000; i++)
            {
               c.increment();
            }
        }
      });


            t1.start();
            t2.start();
      t1.join();
      t2.join();

            System.out.println("Count = "+ c.count);
 }
```

Output:

```
C:\>javac SyncDemo.java

C:\>java SyncDemo
Count = 8343

C:\>java SyncDemo
Count = 9998

C:\>java SyncDemo
Count = 9865

C:\>java SyncDemo
Count = 9989

C:\>java SyncDemo
Count = 9790

C:\>java SyncDemo
Count = 9954

C:\>java SyncDemo
Count = 9799
```

Here the count should be 10000, but it is printing less than 10000. To solve this problem, add synchronized keyword to the increment() method as shown in the

following code.

```
class Counter
{
      int count=0;
      public synchronized void increment()
      {
        count++;
      }
}
```

Output: After adding synchronized keyword to increment() method

```
C:\>javac SyncDemo.java

C:\>java SyncDemo
Count = 10000

C:\>java SyncDemo
Count = 10000

C:\>java SyncDemo
Count = 10000
```

9. Deadlock and Race Situations

In a multithreaded program, the race and deadlock conditions are common when improperly synchronized threads have a shared data source as their target.

A race condition may occur when more than one thread tries to execute the code concurrently. A thread partly does the job when the second thread enters.

If the process is **atomic**, that is, it cannot be subdivided the effect of race condition will be limited.
However, when the process is **not atomic** and can be subdivided into two or more sub-processes, it may occur that a thread has done subprocess, and the second thread enters and starts executing. In such cases, the results can be far from the desired ones.

Therefore, a race condition is a situation in which two or more threads try to execute code and their actions get interleaved. The solution for such condition is the synchronization, and therefore, only one thread should enter code at a time and othersshould wait until the first has done its job.

In multithreaded programs, the deadlock conditions are also common

A deadlock condition may lead to program crash. Such a situation occurs when two threads are attempting to carry out synchronized hods that are inter-dependent, that is, the completion of Method1 needs Method2 and completion of Method2 needs Method1.

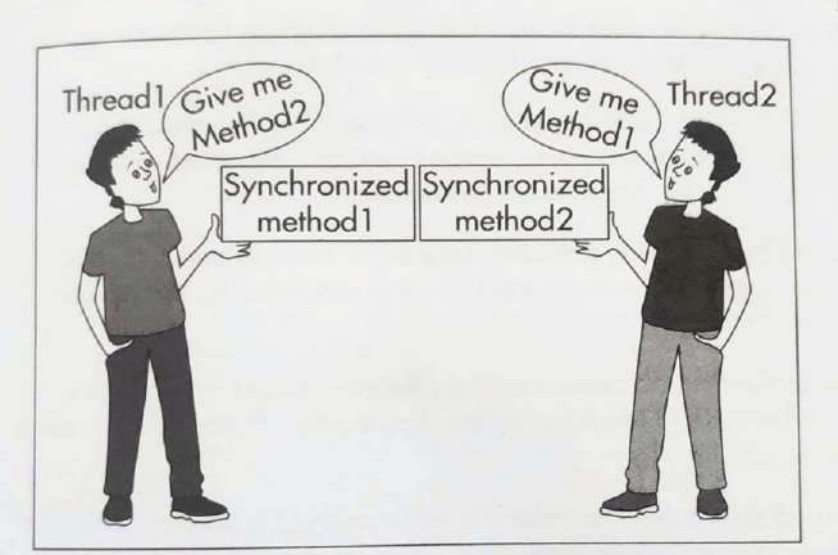


Fig. 13.8 Example of deadlock situation

10.Inter-thread Communication

- Iter-thread communication involves synchronized threads to communicate with each other to pass information. In this mechanism, one of the threads is paused in its critical section to run and another thread is allowed to enter in the same critical section to be executed
- The inter-communication between threads is carried out by three methods, wait(), notify(), and notifyAll(), which can be called within the synchronized context.

Methods for Inter-Thread communication are as follows

i. final void wait()
ii. final void wait(long time milliseconds)
iii. final void wait (long time milliseconds, int nanoseconds)
iv. final void notify()
v. final void notifyAll()

Example: ThreadDemo2.java

```
class BankAccount
{
      int accountNumber;
      static double balance;

      BankAccount(int n, double y)
      {
         accountNumber = n;
         balance = y;
      }

      synchronized void withDraw(int wd)
      {
            if(balance < wd)
                System.out.println("Less balance " + + balance + " is
available; waiting to deposit more " );
```

```
            if(balance>= wd)
            {
               System.out.println("balance is available : "+ balance);
               balance = balance -wd;
               System.out.println("balance after withdrawal : " +
balance);
            }
            try{
                 wait();
            }catch(Exception e)
            {
            }
            if(balance> wd)
            {
               System.out.println("balance is available : "+ balance);
               balance = balance -wd;
               System.out.println("balance after withdrawal : " +
balance);
            }

      }

      synchronized double deposit(int dp)
      {
            System.out.println("Going to deposit:  " + dp );
            balance = balance + dp;
            System.out.println("Balance after deposit = "+ balance);
            notify();
            return(balance);
      }
}

public class ThreadDemo2
{
      public static void main(String[] args)
      {
      final BankAccount ba = new BankAccount(2345, 1000.0);
      new Thread()
      {
            public void run()
            {
              ba.withDraw(5000);
            }
      }.start();


      new Thread()
      {
            public void run()
            {
            ba.deposit(15000);
            }
      }.start();

      }
}
```

```
C:\ >javac ThreadDemo2.java

C:\ >java ThreadDemo2
Less balance 1000.0 is available; waiting to deposit more
Going to deposit:  15000
Balance after deposit = 16000.0
balance is available : 16000.0
balance after withdrawal : 11000.0
```

11.Suspending Resuming and Stopping of Threads.

- In Java 1.1, the following three methods are defined
    - suspend() - pause the operation of thread
    - resume() - resume the operation of thread
    - stop() - to terminate the thread

- These direct methods are very convenient to control the operation of threads in a multithread environment.
- However, in some situations, they may crash the program or cause serious damage to critical data.
- For example, if a thread has got the lock to a critical synchronized section of code and gets suspended, it will not release the lock for which other threads may be waiting.
- Instead of methods suspend() and resume(), the methods wait() and notify() are used.

Example: ThreadDemo2.java

III. Java Database Connectivity

1. Introduction

JDBC stands for Java Database Connectivity and has been developed by Sun Microsystems.

It is a standard Java API that defines how the front-end application (that may be web application or simple Java application) may access the database.

It provides a standard library for accessing a wide variety of database systems. Previously, Open Database Connectivity (ODBC) API used to be the database API for connecting and executing query with the database.

JDBC applications are platform independent, and thus, they can be used for connecting with Internet applications. JDBC applications are simpler and easier to develop.

The following Figure illustrates the connectivity model of JDBC.

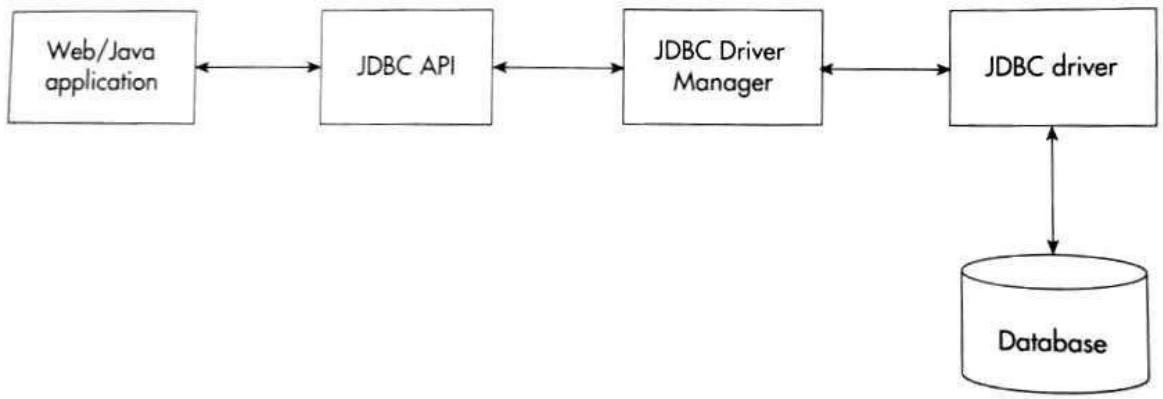


Fig. 27.1 JDBC connectivity model

JDBC API and JDBC driver form the important components in order to fetch/store the information to the database.

JDBC API is installed at the client side. Therefore, when the user wants to fetch some data from the database, the user sets up the connection to JDBC manager using JDBC API through the Java application.

JDBC manager needs a medium in order to communicate with the database. JDBC driver provides this medium and the required information to JDBC manager, JDBC library includes APIs that define interfaces and classes for writing database applications in Java.

Through the JDBC API, we can access a wide variety of database systems including relational as well as non-relational database system. This chapter focuses on using JDBC to access data in Relational Database

Relational Database Management System (RDBMS). It is a database system that is based on relational model. Data in RDBMS is stored in the form of database objects called tables.

Table is a collection of related data entries and consists of columns and rows.

Some of the most widely used relational database systems include Microsoft MySQL server, Oracle, and IBM's DB2.

Any Java-based application can access a relational database. For this, any RDBM system needs to be installed that provides driver conforming to Java Database Connectivity.

JDBC API enables the programmers to change the underlying DBMS (for example, from MySQL to Oracle), without changing the Java code that accesses the database.

2. JDBC Architecture

i. Two-tier Architecture for Data Access

JDBC API supports both two-tier and three-tier processing models for database access. This implies that a Java application can communicate directly with the database or through a middle-tier element.

In this model, Java application communicates directly with the database. For this, JDBC driver is required and it can establish direct communication with the database.

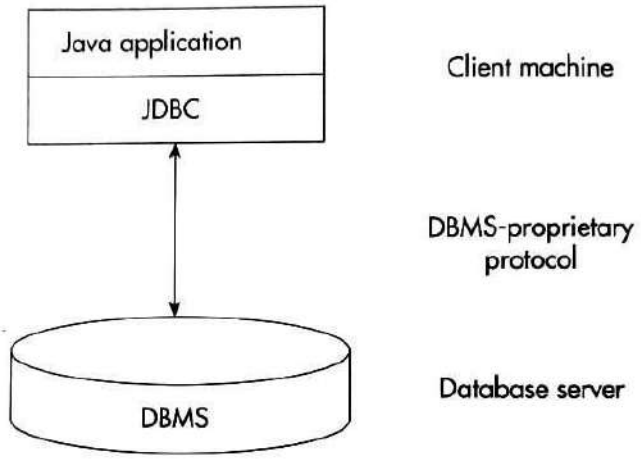


Fig. 27.2 Two-tier architecture

As can be seen from the figure, both Java application and JDBC API are located at the client machine and the DBMS and database are located at the database server.

User sends commands to the database. The commands processed and the results of these statements are sent to the user.

Java application and the database may reside on the same machine. Alternatively, database may be on the server machine, while the Java application may be on the client machine, which may be connected via the network.

ii. Three-tier Architecture for Data Access

In this model, user's commands are first sent to the application server forming the middle tier. Application server containing the JDBC API sends the SQL statements to the database located on the database server. The commands are processed and the result is sent to the middle tier, which then sends it to the user.

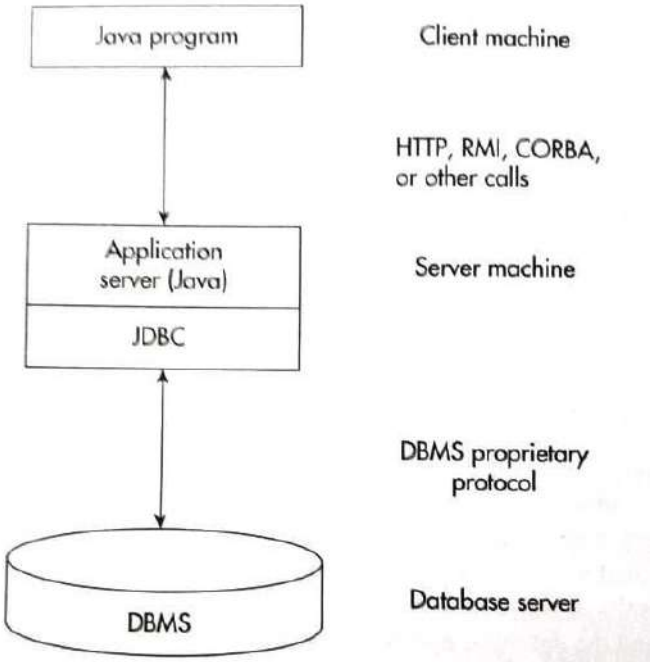


Fig. 27.3 Three-tier architecture

The above Figure depicts the basic three-tier model.

Middle tier has often been written in languages such as C or C++ that provide the fast performance.

However, Java platform has become the standard platform for middle-tier development with the advancements made in the optimizing compilers. These compilers translate Java byte code into an efficient machine specific code.

This model is usually common in web applications in which the client tier is implemented in the web browser. Web server forms the middle tier and the database management system runs on database server. This model provides better performance and simplifies the deployment of applications.

3. Installing MySQL and MySQL Connector

- MySQL is the most widely used open-source relational database management system.
- It is considered to be one of the best RDBMS systems for developing web-based software applications as it provides speed, flexi bility, and reliability.

- Before we can use MySQL with JDBC, we first need to install MySQL.
- **"MySQL community edition"** freely available and can be downloaded from the **MySQL website**

http://www.mysql.com .

Steps to install MySQL database system on Windows platform are as follows:

i. On successful download, click the mySQL icon. MySQL Server 5.6 setup. Click on the Next button.
ii. License agreement page would appear next. Read the terms and conditions and click on I accept the wizard window would terms in the license Agreement checkbox.
iii. Next, it will ask for the set up type that suits your needs. Click on Typical button in Choose set up Type screen.
iv. Then click on install button for starting the installation process wizard.
v. Completed MySQL Server 5.6 Setup Wizard would appear. Click on finish to exit the
vi. MySQL Server Instance Configuration screen would appear, and select the Detailed Configuration.
vii. Following this, MySQL Server Instance Configuration Wizard would open up and choose the server as Developer Machine.
viii. Now click on Next button, and select the Dedicated MySQL Server Machine or you may choose Developer Machine if other applications and tools are being run on your system.
ix. Thereafter, select multifunctional database for general purpose database, when MySQL Server Instance Configuration Wizard screen appears. Then, select the drive where the database files would be stored.
x. Keep the default port as 3306 and specify the root password.

After the successful installation of MySQL, we need to install **MySQL Connector/J** (where J stands for Java). **This is the JDBC driver that allows Java applications to interact with MySQL.**

MySQL connector/J can be downloaded from

dev.mysql.com/downloads/connector/j/3,1.html.

SQL Statements

SQL statements are used for performing various actions on the database.

i. **SQL select statements :**
The select statements are used to select data from the database.

Syntax :

```
select column_name1,  column_name2  from  tablename;
```

Example-1:

```
select id, stdName from Student;
```

Here :

id and stdName are the column names

Student is the table name;

Example-2:

```
Select * from Student;
```

The above statement selects all the columns from Student table.

ii. SQL insert into Statement:

SQL insert into statement It is used to insert new records in a table.

```
insert into tablename values (valuel, value2, value3_);
```

Alternatively, we can also write,

```
insert into tablename (calumn1, column2...)
       Values (valuel, value2; value3.);
```

For instance,

```
insert into Student values (Riya, Sharma, 60);
```

iii. SQL where clause

SQL where clause It is used to extract only those records that satisfy a particular criterion.
The syntax for this statement is:

```
select column_namel, column_name2 from table_name
where column_name operator value;

select from * Student where id=347;
```

iv. SQL delete statement

SQL delete statement It is used to delete the records in the table.
The syntax for this statement

```
delete from table_name where column_name = value;
```

example:

```
delete from Student where firstname="Riya";
```

4. JDBC Environment Setup

This section focuses on how to set up the connection to MySQL database from NetBeans IDE. NetBeans IDE supports MySQL RDBMS. In order to access MySQL database server in NetBeans IDE, we have to first configure the MySQL Server properties. This involves the following steps:

1. Open the NetBeans IDE, right click the database node in the services window. Select Register MySQL Server MySQL server properties dialog box would open. Confirm that the server host name and port are correct. The default server host name is localhost and 3306 is the default server port name.

2. Enter the administrator password. You can give any password and default is set to blank password.

3. At the top of dialog box, click on the Admin properties tab. Here, you can enter the information for controlling the MySQL server.

4. In the admin properties dialog box, enter the path/URL to admin tool. This is the location of MySQL Administration.

5. Next, you have to enter the path to start the command. For this, look for mysqld in the bin folder MySQL installation directory of

6. In the path to stop the command field, type or browse to the location of MySQL stop command. This is the path where mysqladmin the bin folder of MySQL installation directory is located.

Additional steps that must be checked before starting with the involvement of Java-based database connectivity:

1. Before starting, make sure that MySQL Database server is running. If database server is not connected, it will show 'disconnected'. For connecting it, right click the Database, and choose 'connect". This may also prompt to give password to connect to the database server

2. When a new project is created in NetBeans, copy the mysql-connector/java JAR file into the library folder.

JDBC Connectivity Model and API

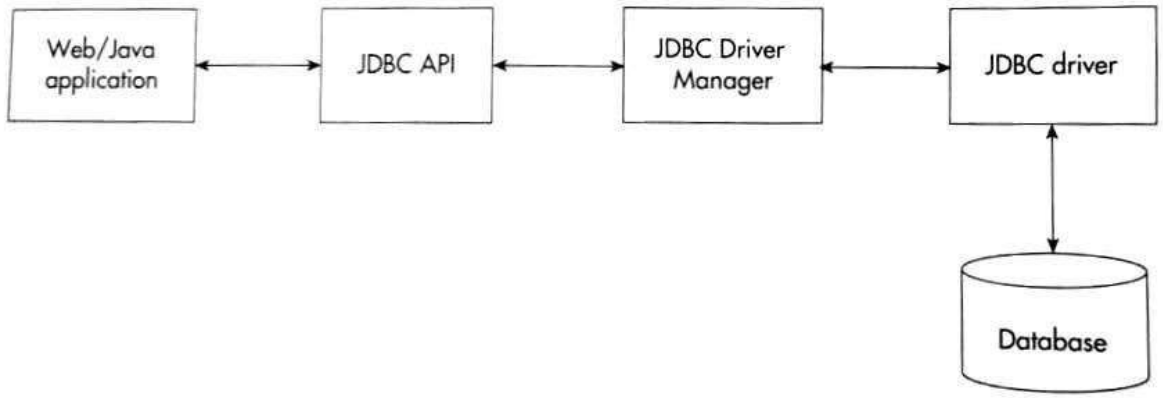


Fig. 27.1 JDBC connectivity model

Fig. 27.1 depicts the JDBC connectivity model. JDBC API enables Java applications to be connected to relational databases through standard API. This makes possible for the user to establish a connection to a database, create SQL or MySQL statements, execute queries in the database, and so on. JDBC API comprises the following interfaces and classes:

Driver manager

This class manages a list of database drivers. It matches the connection request from the Java application with the database driver using communication sub-protocol. **It acts as an interface between the user and the driver and is used to get a Connection object.** Commonly used methods of this class are as follows

Method

`Connection getConnection(String url)` :
It is used to establish the connection with the specified URL

`Connection getConnection(String url, String username, String password)`
It is used to establish the connection with the specified URL, username, and password

Driver

It handles communication with the database server. JDBC drivers are written in Java language in order to connect with the database. JDBC driver is a software component comprising a set of Java classes that provides linkage between Java program running on Java platform and RDBM system that is residing on the operating system.

There are basically four different types of JDBC drivers and these implementations vary because of the wide variety of operating systems and hardware platforms available in which Java operates

Type 1 JDBC-ODBC bridge driver

Type 1 JDBC driver provides a standard API for accessing SQL on Windows platform. In this type of the driver, JDBC bridge is used to access ODBC drivers installed on the client machine. For using ODBC, Data Source Name (DSN) on the client machine is required to be configured.
The driver converts JDBC interface calls into ODBC calls. It is, therefore, the least efficient driver of the four types. These drivers were mostly used in the beginning and now it is usually used for experimental purposes when no other alternative is available

Type 2 driver (also known as Native API driver)

In Type 2 driver, Java interface for vendor-specific API is provided and it is implemented in native code. It includes a set of Java classes that make use of **Java Native Interface (JNI)** and acts as bridge between Java and the native code.

JNI is a standard programming interface that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications (these include the programs that are specific to a particular hardware and operating system like C/C++).

Thus, the driver converts JDBC method calls into native calls of the database API. For using this driver, it is required that RDBMS system must reside in the same host as the client program.

The Type 2 driver provides more functionality and performance than Type 1 driver. For using this driver in a distributed environment, it is required that all the classes that operate on the database should reside on the database host system.

Type 3 driver (also known as Network-Protocol driver)

It is similar to Type 2 driver but in this case, the user accesses the database through TCP/IP connection. The driver sends JDBC interface calls to an inter mediate server, which then connects to the database on behalf of the JDBC driver.

Type 3 and 4 drivers are preferably used if the program application does not exist on the same host as the database. It requires data base-specific coding to be done in the middle tier.

Type 4 driver (also known as Native-Protocol driver)

Type 4 JDBC driver is completely written in Java, and thus, it is platform independent. The driver converts JDBC calls directly into vendor-specific database protocol.

It is installed inside the Java Virtual Machine of the client and most of the JDBC drivers are of Type 4.

It provides better performance when compared to Type 1 and 2 drivers as it does not have the over head of conversion of calls into ODBC or database API calls, However, at the client side, a separate driver is required for each database.

Packages:

There are two packages that make up the JDBC API.
They are

i. **java.sql**
ii. **javax.sql**

iii. java. sql package provides API for accessing and processing data that is stored in a data source. This API comprises framework whereby different drivers can be installed dynamically in order to access different data sources. For instance, it comprises API for establishing a connection with a database via the Driver manager facility, sending SQL statements a database, retrieving and updating the results of the query, and so on.

iv. javax.sql package provides the required API for server-side data source access and processing. This package supplements the java.sql package. It is included in the Java Platform Standard Edition (Java SETM).

Connection:

This interface comprises methods for making a connection with the database. All types of communication with the database is carried out through the connection object only.

Statement

Object created from this interface is used to submit the SQL statements to the database.

ResultSet

It acts as an iterator that enables to move through the data. Object created from interface is used to data received from the database after executing SQL query.

SQL Exception

This class handles errors that may occur in a database application.