B.Tech-CSE(AI&ML)

B.Tech II Year - I Semester	OBJECT-ORIENTED PROGRAMMING THROUGH JAVA (23HPC0504)	L	T	P	C
		3	0	0	3

Course Objectives: The learning objectives of this course are to:

- To understanding the object oriented concepts and problem solving techniques.
- To obtain Knowledge about the principles of Inheritance and polymorphism.
- To introduce the implementation of packages and Exception handling and execution.
- Able to design GUI based application and Java FX.

Course Outcomes: After completion of the course, students will be able to

- Able to solve real world problems using OOP'S techniques.
- Understanding the Syntax, Semantics and Features of Java programming language.
- Demonstrate features of Interfaces to implement Multiple Interfaces.
- Learn when to use Exception handling and how to create user defined exception.
- Able to develop multithreaded applications with Synchronizations, graphical user interface(GUI) java FX.

UNIT I: Object Oriented Programming:

Basic concepts, Principles, Program Structure in Java: Introduction, Writing Simple Java Programs, Elements or Tokens in Java Programs, Java Statements, Command Line Arguments, User Input to Programs, Escape Sequences Comments, Programming Style.
Data Types: Variables, and Operators :Introduction, Data Types in Java, Declaration of Variables, Data Types, Type Casting, Scope of Variable Identifier, Literal Constants, Symbolic Constants, Formatted Output with printf() Method, Static Variables and Methods, Attribute Final, **Introduction to Operators:** Precedence and Associativity of Operators, Assignment Operator (=), Basic Arithmetic Operators, Increment (++) and Decrement (- -) Operators, Ternary Operator, Relational Operators, Boolean Logical Operators, Bitwise Logical Operators.
Control Statements: Introduction, if Expression, Nested if Expressions, if–else Expressions, Ternary Operator?:, Switch Statement, Iteration Statements, while Expression, do–while Loop, for Loop, Nested for Loop, For–Each for Loop, Break Statement, Continue Statement.

UNIT II: Classes and Objects: Introduction, Class Declaration and Modifiers, Class Members, Declaration of Class Objects, Assigning One Object to Another, Access Control for Class Members, Accessing Private Members of Class, Constructor Methods for Class, Overloaded Constructor Methods, Nested Classes, Final Class and Methods, Passing Arguments by Value and by Reference, Keyword this.

Methods: Introduction, Defining Methods, Overloaded Methods, Overloaded Constructor Methods, Class Objects as Parameters in Methods, Access Control, Recursive Methods, Nesting of Methods, Overriding Methods, Attributes Final and Static.

UNIT III: Arrays: Introduction, Declaration and Initialization of Arrays, Storage of Array inComputer Memory, Accessing Elements of Arrays, Operations on Array Elements, Assigning Array to Another Array, Dynamic Change of Array Size, Sorting of Arrays, Search for Values in Arrays, Class Arrays, Two-dimensional Arrays, Arrays of Varying Lengths, Three-dimensional Arrays, Arrays as Vectors.
Inheritance: Introduction, Process of Inheritance, Types of Inheritances, Universal Super Class-Object Class, Inhibiting Inheritance of Class Using Final, Access Control and Inheritance, Multilevel Inheritance, Application of Keyword Super, Constructor Method and Inheritance, Method Overriding, Dynamic Method Dispatch, Abstract Classes, Interfaces and Inheritance.
Interfaces: Introduction, Declaration of Interface, Implementation of Interface, Multiple Interfaces, Nested Interfaces, Inheritance of Interfaces, Default Methods in Interfaces, Static Methods in Interface, Functional Interfaces, Annotations.

UNIT IV: Packages and Java Library: Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path, Access Control, Packages in Java SE, Java.lang Package and its Classes, Class Object, Enumeration, class Math, Wrapper Classes, Auto-boxing and Auto-unboxing, Java util Classes and Interfaces, Formatter Class, Random Class, Time Package, Class Instant (java.time.Instant), Formatting for Date/Time in Java, Temporal Adjusters Class, Temporal Adjusters Class.
Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throwable, Unchecked Exceptions, Checked Exceptions.
Java I/O and File: Java I/O API, standard I/O streams, types, Byte streams, Character streams, Scanner class, Files in Java(Text Book 2)

UNIT V: String Handling in Java: Introduction, Interface Char Sequence, Class String, Methods for Extracting Characters from Strings, Comparison, Modifying, Searching; Class String Buffer.
Multithreaded Programming: Introduction, Need for Multiple Threads Multithreaded Programming for Multi-core Processor, Thread Class, Main Thread-Creation of New Threads, Thread States, Thread Priority-Synchronization, Deadlock and Race Situations, Inter-thread Communication - Suspending, Resuming, and Stopping of Threads.
Java Database Connectivity: Introduction, JDBC Architecture, Installing MySQL and MySQL Connector/J, JDBC Environment Setup, Establishing JDBC Database Connections, Result Set Interface
Java FX GUI: Java FX Scene Builder, Java FX App Window Structure, displaying text and image, event handling, laying out nodes in scene graph, mouse events (Text Book 3)

Text Books:

1. JAVA one step ahead, Anitha Seth, B.L.Juneja, Oxford.
2. Joy with JAVA, Fundamentals of Object Oriented Programming, DebasisSamanta,MonalisaSarma, Cambridge, 2023.
3. JAVA 9 for Programmers, Paul Deitel, Harvey Deitel, 4th Edition, Pearson.

References Books:

1. The complete Reference Java, 11th edition, Herbert Schildt,TMH
2. Introduction to Java programming, 7th Edition, Y Daniel Liang, Pearson

Online Resources:

1. https://nptel.ac.in/courses/106/105/106105191/
2. https://infyspringboard.onwingspan.com/web/en/app/toc/lex_auth_012880464547618816347_shared/overview

UNIT-1

Program Structure in Java: Introduction, Writing Simple Java Programs, Elements or Tokens in Java Programs, Java Statements, Command Line Arguments, User Input to Programs, Escape Sequences Comments, Programming Style.

Data Types, Variables, and Operators :Introduction, Data Types in Java, Declaration of Variables, Data Types, Type Casting, Scope of Variable Identifier, Literal Constants, Symbolic Constants, Formatted Output with printf() Method, Static Variables and Methods, Attribute Final, Introduction to Operators, Precedence and Associativity of Operators, Assignment Operator (=), Basic Arithmetic Operators, Increment (++) and Decrement (- -) Operators, Ternary Operator, Relational Operators, Boolean Logical Operators, Bitwise Logical Operators.

Control Statements: Introduction, if Expression, Nested if Expressions, if–else Expressions, Ternary Operator ?:, Switch Statement, Iteration Statements, while Expression, do–while Loop, for Loop, Nested for Loop, For–Each for Loop, Break Statement, Continue Statement.

INTRODUCTION TO JAVA:

Java is a popular object-oriented programming language, developed by sun micro systems of USA in 1991(which has since been acquired by oracle). originally it was called as oak. the developers of java are james gosling and his team (patrick naughton, chris warth, ed frank, and mike sheridan). this language was renamed as “java” in 1995. java was publicly announced in 1995. and now more than 3 billion devices run java.

Features of Java:

- Platform Independence: Java works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc.) and Programs developed in JAVA are executed anywhere on any system.
- Simple: They made java simple by eliminating difficult concepts of C and C++. example the concept of pointers and Java is simple because it has the same syntax of C and C++
- Object Oriented: Object oriented throughout - no coding outside of class definitions, including main ().

- Compiler/Interpreter Combo: Usually a computer language is either compiled or interpreted. Java combines both these approaches thus making java a two-stage system.First, java compiler translates source code into bytecode. Bytecodes are not machine instructions and therefore, in the second stage, java interpreter generates machine code that can be directly executed by the machine that is running the java program so we can say that java is both compiled and interpreted language.
- Robust: because of the following concept included in java it is called as robust
    1. Exception handling
    2. strong type checking
    3. local variables must be initialized.
- Automatic Memory Management: Automatic garbage collection - memory management handled by JVM.
- Security: Security problems like eavesdropping, tampering, impersonation and virus threats can be eliminated or maintained by using java on internet.
- Dynamic Binding: The linking of data and methods to where they are located, is done at run-time.
- High Performance: The Java language supports many high-performance features such as multithreading, just-in-time compiling, and native code usage.
- Threading: A thread represents an individual process to execute a group of statements. JVM uses several threads to execute different block of code.
- Built-in Networking: Java was designed with networking in mind and comes with many classes to develop sophisticated Internet communications.
- It is open-source and free
- It is secure, fast and powerful

Java is used for:

- Mobile applications (especially Android apps)
- Desktop applications
- Web applications
- Web servers and application servers
- Games
- Database connection and much, much more!

ELEMENTS OR TOKENS IN JAVA PROGRAM:

A token is the smallest element of a program that is meaningful to the compiler. Tokens can be classified as follows:

- Keywords
- Identifiers
- Constants
- Special Symbols
- Operators

1. **Keyword:** Keywords are pre-defined or reserved words in a programming language. Each keyword is meant to perform a specific function in a program. Since keywords are referred names for a compiler, they can't be used as variable names because by doing so, we are trying to assign a new meaning to the keyword which is not allowed. **Java** language supports following keywords:

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

2. Identifiers: Identifiers used for naming classes, methods, variables, objects, labels and interfaces in a program.

   Java identifiers follow the following rules:

   1. Identifier must start with a letter, a currency character ($), or a connecting character such as underscore (_).

2. Identifier cannot start with a number.
3. Uppercase and lowercase letters are distinct.
4. Total, TOTAL and total are different.
5. They can be of any length.
6. It should not be a keyword.
7. Spaces are not allowed.

Examples of legal and illegal identifiers follow, first some legal identifiers:

- int _a;
- int $c;
- int______2_w;
- int _$;
- int this_is_a_very_detailed_name_for_an_identifier;

The following are illegal (Recognize why):

- int :b;
- int -d;
- int e#;
- int .f;

Conventions for identifier names:

1. Names of all public methods and instance variables start with lowercase, for more than one word second word's first character shold be capital.
   Ex:- total, display(), totalMarks, netSalary, getData()

2. All classes and interfaces should start with capital letter, for more then one word the second word's first character should be capital.
   Ex:- Student, HelloJava

3. Constant variables should be in capital letters and for more than one word underscore is used.
   Ex:- MAX, TOTAL_VALUE

4. Package name should be in lowercase only.
   Ex:- mypack

3. Constants: Constants are also like normal variables. But the only difference is, their values cannot be modified by the program once they are defined. Constants refer to fixed values. They are also called as literals. Constants may belong to any of the data type.

Syntax: final data_type variable_name;

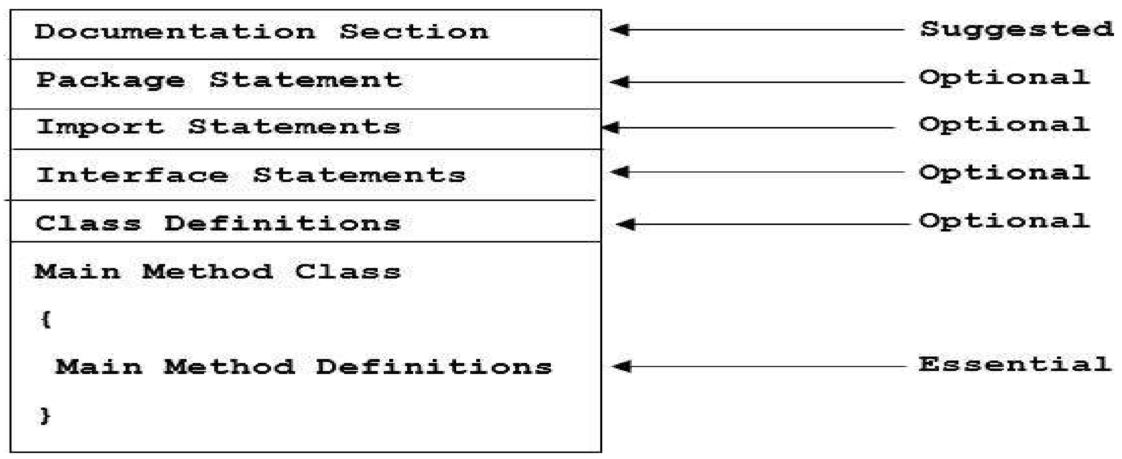
4. **Special Symbols:** The following special symbols are used in Java having some special meaning and thus, cannot be used for some other purpose.

   [] () {}, ; * =

   - Brackets[]: Opening and closing brackets are used as array element reference. These indicate single and multidimensional subscripts.
   - Parentheses(): These special symbols are used to indicate function calls and function parameters.
   - Braces{}: These opening and ending curly braces marks the start and end of a block of code containing more than one executable statement.
   - comma (,): It is used to separate more than one statements like for separating parameters in function calls.
   - semi colon : It is an operator that essentially invokes something called an initialization list.
   - asterick (*): It is used to create pointer variable.
   - assignment operator: It is used to assign values.
5. Operators: Java provides many types of operators which can be used according to the need. They are classified based on the functionality they provide. Some of the types are-
   - Arithmetic Operators
   - Unary Operators
   - Assignment Operator
   - Relational Operators
   - Logical Operators
   - Ternary Operator(conditional operator)
   - Bitwise Operators and bit-shift operators
   - instance of operator

WRITING SIMPLE JAVA PROGRAM:

A java program may contain many classes, of which only one class should contains main() method.



Documentation section:

It consists of comment lines(program name, author, date,…….),

// - for single line comment.

/*--- */ - multiple line comments.

/** --*/ - known as documentation comment, which generated documentation automatically.

Package statement:

This is the first statement in java file. This statement declares a *package name* and informs the compiler that the class defined here belong to this package.

Ex: - package student

Import statements:

This is the statement after the package statement. It is similar to # include in c/c++.

Ex: import java.lang.String

The above statement instructs the interpreter to load the String class from the lang package.

Note:

- Import statement should be before the class definitions.
- A java file can contain N number of import statements.

Interface statements:

An interface is like a class but includes a group of method declarations.

Note: Methods in interfaces are not defined just declared.

Class definitions:

Java is a true oop, so classes are primary and essential elements of java program. A program can have multiple class definitions.

Main method class:

Every stand-alone program required a main method as its starting point. As it is true oop the main method is kept in a class definition. Every stand-alone small java program should contain at least one class with main method definition.

Rules for writing JAVA programs:

1. Every statement ends with a semicolon
2. Source file name and class name must be the same.
3. It is a case-sensitive language.
4. Every thing must be placed inside a class, this feature makes JAVA a true object oriented programming language.

Rules for file names:

- A source code file can have only one public class, and the file name must match the public class name.
- A file can have more than one non-public class.
- Files with no public classes have no naming restrictions.

Simple Java Program:

```
/* This is a simple Java program.
Program name : Sample.java */
class Sample
{
        public static void main(String args[])
        {
                System.out.println("Hello! Vahida Welcomes U to JAVA Class");
         }
}
```

The name of the source file and name of the class name (which contains main method) must be the same, because in JAVA all code must reside inside a class.

Class Sample:

The keyword *class* is used to declare a class and *Sample* is the JAVA identifier ie name of the class.

public static void main(String args[])

public:

The keyword *public* is an access specifier that declares the *main* method as unprotected and therefore making it accessible to all other classes. The opposite of *public* is *private*.

Static:

The keyword static allows main() to be called without creating any instance of that class.

This is necessary because main() is called by java interpreter before creating any object.

Void:

The keyword *void* tells the compiler that main() does not return any value.

main():

main() is the method called when a java application begins.

String args[]:

Any information that you pass to a method is received by variables specified with in the set of parenthesis that follow the name of the method. These variables are called parameters. Here args[] is the name of the parameter of string type.

System.out.println:

It is equal to printf() or cout<<, since java is a true object oriented language, every method must be part of an object.

The *println* method is a member of *out* object, which is a static data member of *System* class.

Note: println always appends a newline character to the end of the string. So the next println prints the statements in next line.

Compiling the program:

```
C:> javac Sample.java
```

Extension is compulsory at the time of compiling.

javac is a compiler, which creates a file called Sample.class(contains the bytecode).

Note: when java source code is compiled each class is put into a separate

.class file.

Executing the program:

C:> java Sample

java is interpreter which accepts .class file name as a command line argument.

That's why the name of the source code file and class name should be equal, which avoids the confusion.

Output:

Hello! Vahida Welcomes U to JAVA Class

Important Note:

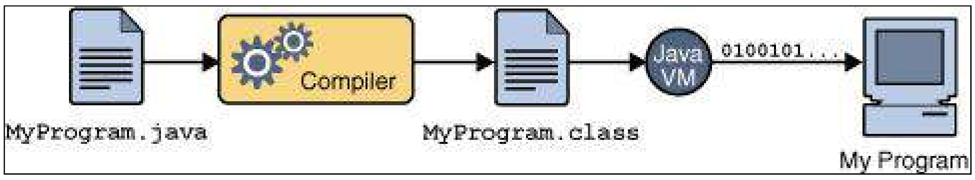
If the source code file name is Test.java and the class name is Hello, then to compile the program we have to give.

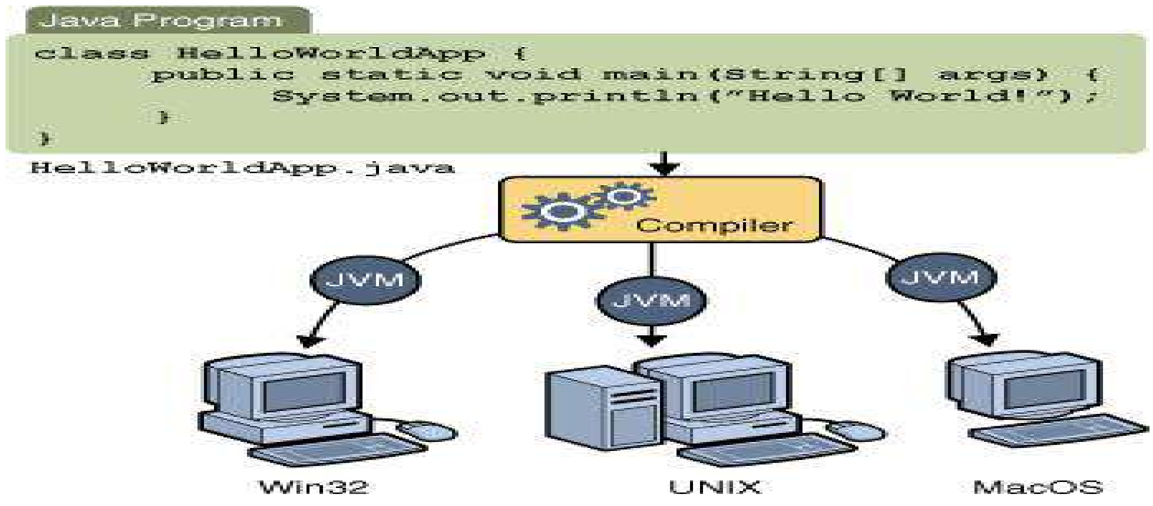
C:> Javac Test.java (which creates a Hello.class file)

To execute the program

C:> Java Hello

The following picture shows the execution of a java program/applet.

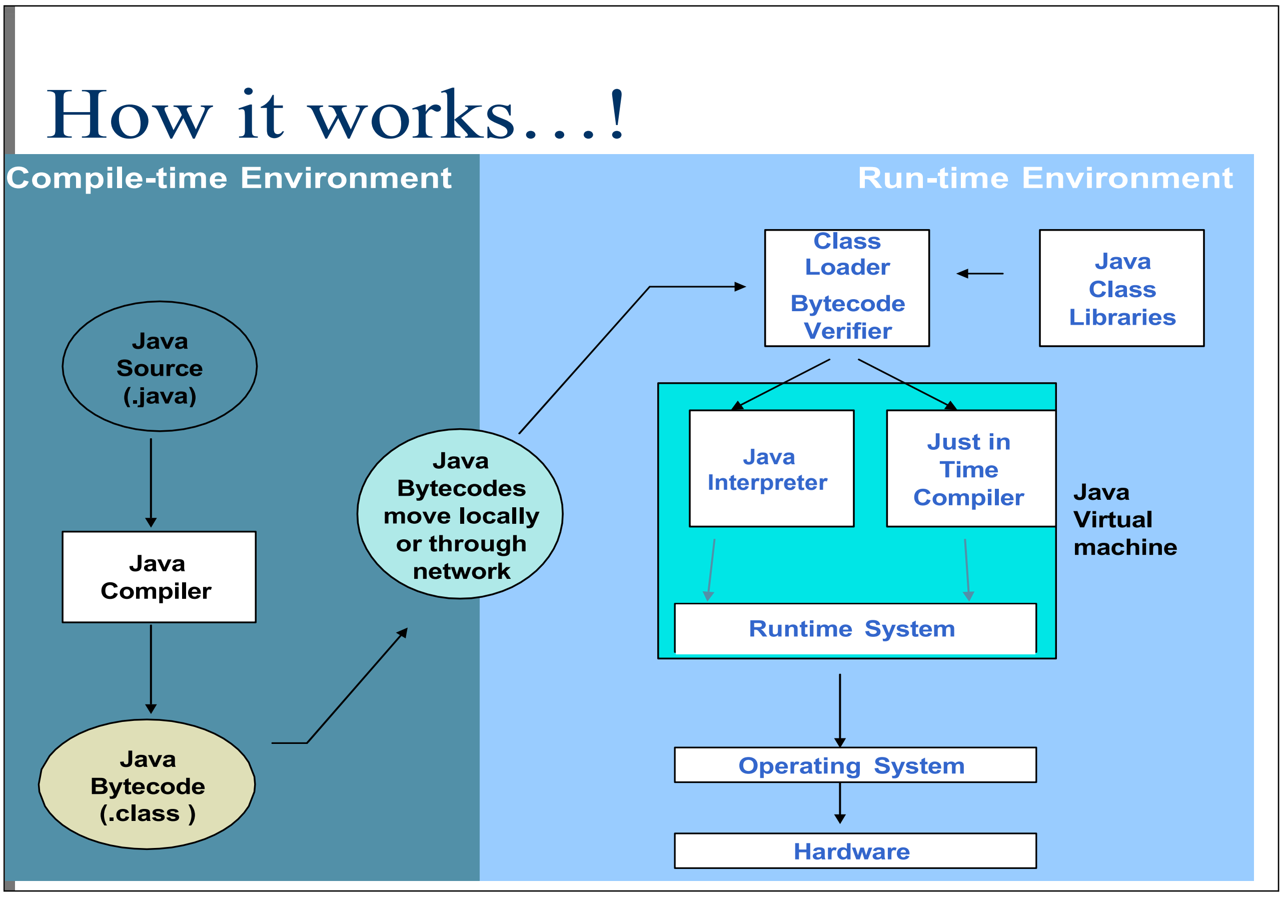




```
class HelloWorldApp {
        public static void main(String[] args) {
                System.out.println("Hello World!");
        }
}
```

Note: JVM is an interpreter for *bytecode.*

***Java interpreter is different from machine to machine



JAVA STATEMENTS:

- A statement specifies an action in a Java program.
- Java statements can be broadly classified into three categories:
    - Declaration statement
    - Expression statement
    - Control flow statement

Java Declaration Statement: A declaration statement is used to declare a variable.

For example,

```
int num;
int num2 = 100;
String str;
```

Java Expression Statement: An expression with a semicolon at the end is called an expression statement. For example,

```
/Increment and decrement expressions
num++;
++num;
num--;
--num;

//Assignment expressions
num = 100;
num *= 10;

//Method invocation expressions
System.out.println("This is a statement");
someMethod(param1, param2);
```

Java Flow Control Statement

By default, all statements in a Java program are executed in the order they appear in the program. Sometimes you may want to execute a set of statements repeatedly for a number of times or as long as a particular condition is true.

All of these are possible in Java using flow control statements. The if statement, while loop statement and for loop statement are examples of control flow statements.

COMMAND LINE ARGUMENTS:

Command line arguments are parameters that are passed to the application program at the time of execution. The parameters are received by args array of string type.

The first argument is stored at args[0]

The second argument is stored at args[1] and so on…..

Example 1:

```
class Commarg
{
 public static void main(String args[])
 {
  int i=0,l;
  l=args.length;
  System.out.println("Number of arguments are:"+l);
  while(i<l)
  {
   System.out.println(args[i]);
   i++;
  }
 }
}
```

Ex:

```
C:..\> java Commarg hello how are u
```

output:

```
       number of arguments are : 4
       hello
       how
       are
       u
```

Note: From the above output it is clear that *hello* is stored at *args[0]* Position and so on…..

Example-2:

Program to add two no’s using command line arguments.

```
class CmdAdd2
{
 public static void main(String args[])
 {
  // if two arguments are not entered then come out
  if(args.length!=2)
  {
          System.out.println("Please enter two values... ");
          return;
  }
  int a=Integer.parseInt(args[0]);
  int b=Integer.parseInt(args[1]);
  int c=a+b;
  System.out.println("The Result is:"+c);
 }
}
```

To run the program:

```
> java CmdAdd2 10 20
```

USER INPUT TO THE PROGRAM:

- The Scanner class is used to get user input, and it is found in the java.util package.
- To use the Scanner class, create an object of the class and use any of the available methods found in the Scanner class.
- For Example, we will use the nextLine() method to read String.

Input Types and their respective methods:

Method	**Description**
nextBoolean()	Reads a boolean value from the user
nextByte()	Reads a byte value from the user
nextDouble()	Reads a double value from the user
nextFloat()	Reads a float value from the user
nextInt()	Reads a int value from the user
nextLine()	Reads a String value from the user
nextLong()	Reads a long value from the user
nextShort()	Reads a short value from the user

Example:

```
import java.util.Scanner;

class Main {

 public static void main(String[] args)

   { Scanner s = new Scanner(System.in);

   System.out.println("Enter name, age and salary:");

   // String input

   String name = s.nextLine();

   // Numerical input

   int age = s.nextInt();

   double salary = s.nextDouble();

   // Output input by user

   System.out.println("Name: " + name);
   System.out.println("Age: " + age);

   System.out.println("Salary: " + salary);

  }

}
```

JAVA COMMENTS:

- Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.
- Single-line comments start with two forward slashes (//).
- Any text between // and the end of the line is ignored by Java (will not be executed).
- This example uses a single-line comment before a line of code:

  Example

  ```
  // This is a comment

  System.out.println("Hello World");
  ```

- Java Multi-line Comments
  - Multi-line comments start with /* and ends with */.
  - Any text between /* and */ will be ignored by Java.
  - This example uses a multi-line comment (a comment block) to explain the code:

    ```
    /* The code below will print the words Hello World

    to the screen, and it is amazing */

    System.out.println("Hello World");
    ```

ESCAPE SEQUENCE CHARACTERS:

A character preceded by a backslash (\) is an escape sequence and has a special meaning to the compiler.

The following table shows the Java escape sequences.

Escape Sequence	**Description**
\t	Inserts a tab in the text at this point.
\b	Inserts a backspace in the text at this point.
\n	Inserts a newline in the text at this point.
\r	Inserts a carriage return in the text at this point.
\f	Inserts a form feed in the text at this point.
\'	Inserts a single quote character in the text at this point.
\"	Inserts a double quote character in the text at this point.
\\\\	Inserts a backslash character in the text at this point.

JAVA VARIABLES:

- Variables are containers for storing data values.
- The Java programming language is strongly-typed, means all variables must be declared before they can be used.

Declaring (Creating) Variables: To create a variable, you must specify the type and assign it a value

Syntax: type variable = value;

Where *type* is one of Java's types (such as int or String), and *variable* is the name of the variable (such as **any identifier**). The equal sign is used to assign values to the variable.

In Java, there are different **types** of variables, for example:

- String - stores text, such as "Hello". String values are surrounded by double quotes
- int - stores integers (whole numbers), without decimals, such as 123 or -123
- float - stores floating point numbers, with decimals, such as 19.99 or -19.99
- char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes
- boolean - stores values with two states: true or false

Example:

```
int num = 5;
float num1 = 5.99f;
char ch = 'D';
boolean status = true;
String name = "sai";
```

Display Variables: The println() method is often used to display variables. To combine both text and a variable , use the + character:

Example

```
int num = 5;
float num1 = 5.99f;
char ch = 'D';
boolean status = true;
String name = "sai";
System.out.println("Value inside integer type variable " + num);
System.out.println("Value inside float type variable " + num1);
System.out.println("Value inside character type variable " + ch);
```

System.out.println("Value inside boolean type variable " + status);

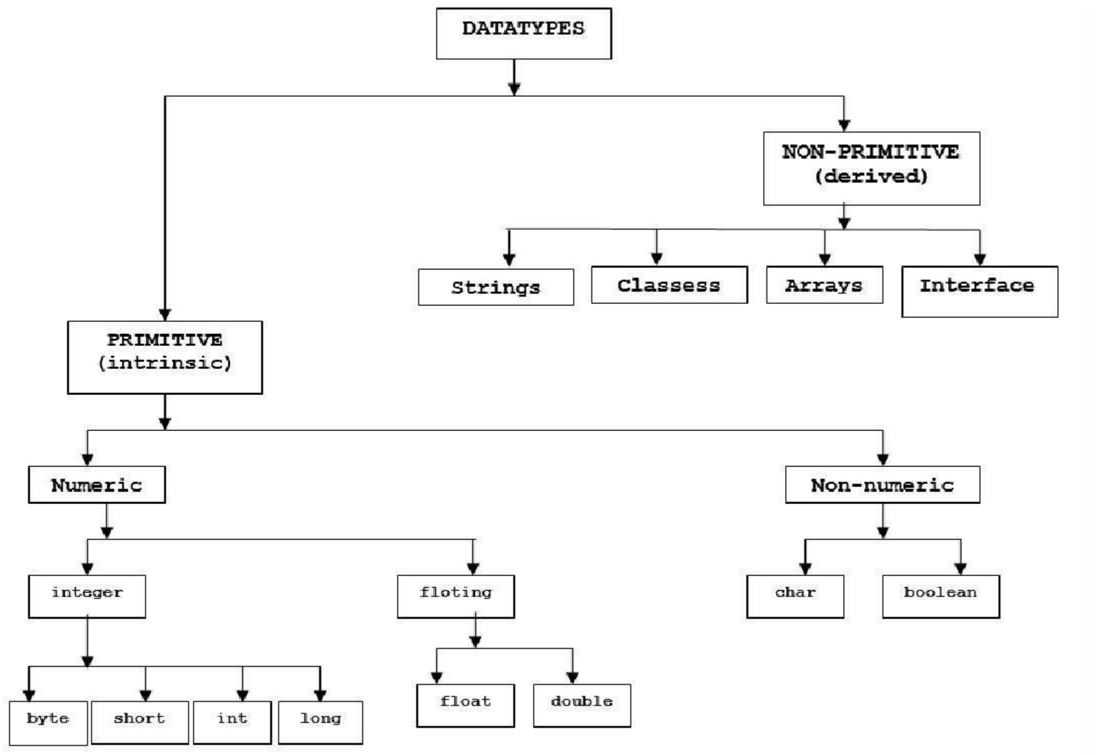
System.out.println("Value inside string type variable " + name);

Declare Many Variables: To declare more than one variable of the **same type**, use a comma-separated list:

int x = 5, y = 6, z = 50;

System.out.println(x + y + z);

DATA TYPES IN JAVA: Datatype specifies the size and types of values that a variable hold. Java is rich in it's data types.



Primitive Data Types: A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data Type	**Size**	**Description**
Byte	1 byte	Stores whole numbers from -128 to 127
Short	2 bytes	Stores whole numbers from -32,768 to 32,767
Int	4 bytes	Stores whole numbers from -2,147,483,648 to 2,147,483,647
Long	8 bytes	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Float	4 bytes	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	8 bytes	Stores fractional numbers. Sufficient for storing 15 decimal digits
boolean	1 bit	Stores true or false values
Char	2 bytes	Stores a single character/letter or ASCII values

Note 1: Note that you should end the value with an "f"

Note 2: Note that you should end the value with a "d"

Example:

```
int num = 5;
float num1 = 5.99f;
double num3=5.7896d;
char ch = 'D';
boolean status = true;
String name = "sai";
System.out.println("Value inside integer type variable " + num);
System.out.println("Value inside float type variable " + num1);
System.out.println("Value inside double type variable " + num1);
System.out.println("Value inside character type variable " + ch);
System.out.println("Value inside boolean type variable " + status);
System.out.println("Value inside string type variable " + name);
```

Non-Primitive Data Types:

- Non-primitive data types are called **reference types** because they refer to objects.
- The main difference between **primitive** and **non-primitive** data types are:
    1. Primitive types are predefined (already defined) in Java. Non-primitive types are created by the programmer and is not defined by Java (except for String).
    2. Non-primitive types can be used to call methods to perform certain operations, while primitive types cannot.
    3. A primitive type has always a value, while non-primitive types can be null.
    4. A primitive type starts with a lowercase letter, while non-primitive types starts with an uppercase letter.
    5. The size of a primitive type depends on the data type, while non-primitive types have all the same size.

6. Examples of non-primitive types are Strings, Arrays, Classes, Interface, etc.

String: The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

Example:

```
String greet = "Hello World";
System.out.println(greet);
```

Note: A String in Java is actually a **non-primitive** data type, because it refers to an object. The String object has methods that are used to perform certain operations on strings.

Type casting: Converting one datatype into another type is called "Type casting".

Data types:

There are two types of data types.

- primitive data types or Fundamental data types.
  - These data types will represent single entity (or value).
  - Ex : int, char, byte,…….
- Referenced Data types or Advanced Data types.
  - These Data types represent several values.
  - Ex: arrays, classes, enums,…

Note: We can convert a primitive type to another primitive type and a Referenced type to another Referenced type, but to convert Primitive type to Referenced type and Referenced type to Primitive type Wrapper classes are used.

Casting primitive Data types:

- It is done in two ways
  - Widening
  - Narrowing
- The primitives are classified into two types lower types and higher types.

byte, short, char, int, long, float double

Lower←.................................→higher

Note:

boolean is not included because it cannot be converted into any type.

Widening primitive data types: Converting lower data type into higher data type is called widening.

Ex: 1

```
char ch='A';
int no=(int)ch;
```

....here no will contain 65 ASCII value of A.

Ex: 2

```
int x=5000;
float sal=(float)x;
```

...here sal will contain 5000.0

Widening is safe because there will not be any loss of data or precision or accuracy. So widening is done by compiler automatically.

Ex 1:

```
 char ch='A';
int  no=ch;
```

Ex 2:

```
 int x=5000;
 float sal=x;
```

widening is also known as *implicit casting.*

Narrowing implicit Data types:

Converting higher data type into lower data type is called narrowing.

Ex 1:

```
  int n=65;
  char ch=(char)n;
```

... here ch will contain A

Ex 2:

```
  double d=12.890;
  int x=(int)d;
```

.. here x will contain 12

narrowing is unsafe because there will be loss of data or precision or accuracy. So narrowing is not done by compiler automatically. The programmer has to use cast operator explicitly.

Narrowing is also called *Explicit casting.*

Static methods:

- A static method is a method that does not act upon instance variables of a class.
- A static method is declared by using the keyword static.
- Static methods are called using classname.methodName()
- The reason why static methods cannot act on instance variables is that JVM first loads static elements of the .class file one specific memory location inside the RAM and then searches for the main method, and then only it creates the objects.
- Since objects are not available at the time calling static methods, the instance variables are also not available at the time of calling the static methods.
- Static methods are also called as *class methods*.
- They cannot refer to this or super keywords (super used in inheritance)

Example:

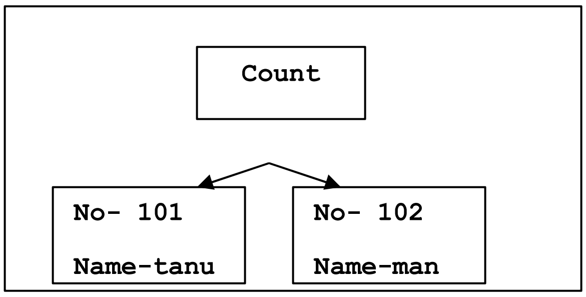
```
class Smethod
{
  static void disp()
  {
  System.out.println("Hi... ");
  }
 public static void main(String args[])
 {
   disp();
   disp();
   disp();
  }
}
```

Static variable:

- The variables that are created comman to all the objects are called static variables/*class variables.*
- The static variables belong to the class as a whole rather than the objects.
- It is mainly used when we need to count no of objects created for the class

Example:

```
class emp
{
 int eno;
 String name;
 static int count;

 emp(int no,String str)
  {
   eno=no;
   name=str;
   count++;
  }
}
class EmpCount
{
  public static void main(String args[])
   {
       emp e1=new emp(101,"tan");
       emp e2=new emp(102,"manu");
       System.out.println("No of objects are:"+e1.count);
   }
}
```



Formatted Output with printf() Method(Displaying output with System.out.printf()):

To format and display the output, printf() method is available in PrintStream class. This method works similar to printf() function in C.

Note: printf() is also available from J2SE 5.0 or later.

The following format characters can be used in printf():

- %s – String
- %c – char
- %d – decimal integer
- %f – float number
- %o – octal number
- %b, %B – Boolean value
- %x, %X – hexadecimal value

- %e, %E – number scientific notation
- %n – new line character

Example :

```
class PrintfDemo
{
 public static void main(String[] args)
 {
  String s="sai";
  int n=65;
  float f=15.45f;
  System.out.printf(" String =%s %n number=%d %n HexaDecimal=%x %n
Float=%f",s,n,n,f);
  }
}
```

Attribute Final:

Final variables: It is just like const in c/c++, which restricts the user to modify. The value of final variable is finalized at the time of declaration it self.

Ex:

```
final double PI=3.14
```

example:

```
class Final
{
	public static void main(String args[])
	{
		final int a=10;
		System.out.println("before modification:"+a);
		a=a+10; //Error: cannot assign a value to final variable a a=a+10;
		System.out.println("after modification:"+a);
	}
}
```

For-each loop:

The enhanced for loop, new to java 5, is a specialized for loop that simplifies looping through an array or a collection.

Syntax:

```
for(declaration: expression)
 {
   ---
   ---
 }
```

Declaration: Its Data type should be same as the current array element.

Expression: It should be the array variable

Example:

```
class Loop
{
	public static void main(String args[])
	{
		int x[]={10,20,30,40,50};
		float y[]={1.5f,2.5f,3.5f,4.5f,5.5f};
		String names[]={"vahida","madhuri","lalitha","anil","manik"};
		for(int i:x)
		{
			System.out.println(i);
		}
		for(float j:y)
		{
			System.out.println(j);
		}
		for(String s:names)
		{
			System.out.println(s);
		}
	}
}
```

UNIT II

Classes and Objects: Introduction, Class Declaration and Modifiers, Class Members, Declaration of Class Objects, Assigning One Object to Another, Access Control for Class Members, Accessing Private Members of Class, Constructor Methods for Class, Overloaded Constructor Methods, Nested Classes, Final Class and Methods, Passing Arguments by Value and by Reference, Keyword this.

Methods: Introduction, Defining Methods, Overloaded Methods, Overloaded Constructor Methods, Class Objects as Parameters in Methods, Access Control, Recursive Methods, Nesting of Methods, Overriding Methods, Attributes Final and Static.

Class:

- A class is the blueprint from which individual objects are created.
- This means the properties and actions of the objects are written in the class.

Class Declaration Syntax:

```
class ClassName
{
   VariableDeclaration-1;
   VariableDeclaration-2;
   VariableDeclaration-3;
    --------------------------
    --------------------------
   VariableDeclaration-n;

   returnType methodname-1([parameters list])
  {
      body of the method…
  }
  returnType methodname-2([parameters list])
  {
      body of the method…
  }
    --------------------------
    --------------------------
  returnType methodname-3([parameters list])
  {
      body of the method…
  }
}// end of class
```

Object:

- Object is an entity of a class.

Object Creation Syntax:

ClassName objectName=new className ();

Class Members:

All the variable declared and method defined inside a class are called class members.

Instance variables: The variables defined within a class are called instance Variables (data members).

Methods: The block in which code is written is called method (member functions).

The Java programming language defines the following kinds of variables:

There are 4 types of java variables

- instance variables
- class variables
- local variables
- Parameters

Instance variables:

Instance variables are declared inside a class. Instance variables are created when the objects are instantiated (created). They take different values for each object.

Class variables:

Class variables are also declared inside a class. These are global to a class. So these are accessed by all the objects of the same class. Only one memory location is created for a class variable.

Local variables:

Variables which are declared and used with in a method are called local variables.

Parameters:

Variables that are declared in the method parenthesis are called parameters.

Class and Object Example:

```
class Test
{       int a; //default access

        void setData(int i)
        {
            a=i;
         }
        int dispData()
        {
          return a;
        }
}
class AccessTest
{
 public static void main(String args[])
 {
  Test ob=new Test(); //object creation
  ob.setData(100);
  System.out.println(" value of a is:-”+ob.dispData());
}
}
```

Access specifiers (Or) Access Control (Or) access Modifiers or Access Control for Class Members (Or) Accessing Private Members of Class:

An access specifier determines which feature of a class (class itself, data members, methods) may be used by another classes.

Java supports four access specifiers:

1. The public access specifier
2. The private access specifier
3. The protected access specifier
4. The Default access specifier

Public:

If the members of a class are declared as public then the members (variables/methods) are accessed by out side of the class.

Private:

If the members of a class are declared as private then only the methods of same class can access private members (variables/methods).

Protected:

Discussed later at the time of inheritance.

Default access:

If the access specifier is not specified, then the scope is friendly. A class, variable, or method that has friendly access is accessible to all the classes of a package (A package is collection of classes).

Example:

```
class Test
{
        int a; //default access
        public int b; // public access
        private int c; // private access
        //methods to access c
        void setData(int i)
        {
                                c=i;
        }
        int dispData()
        {
                                 return c;
        }
}
class AccessTest
{
        public static void main(String args[])
        {
                                Test ob=new Test();
                                //a and b can be accessed directly
                                ob.a=10;
                                ob.b=20;
                                // c can not be accessed directly because it is private
                                //ob.c=100; // error
                                // private data must be accessed with methods of the same class
                                ob.setData(100);
                                System.out.println(" value of a, b and c are:"+ob.a+" "+ob.b+"
"+ob.dispData());
                                ob.dispData();
}
}
```

Assigning One Object to Another or Cloning of objects:

We can copy the values of one object to another using many ways like :

1. Using clone() method of an object class.
2. Using constructor.
3. By assigning the values of one object to another.
4. in this example, we copy the values of object to another by assigning the values of one object to another.

Example:

```
class Copy
{
        int a=10;
}
class CopyObject
{
        public static void main(String args[])
        {
                Copy c1=new Copy();
                Copy c2=c1;
                System.out.println("object c1 value-"+c1.a);
                System.out.println("object c2 value-"+c2.a);
        }
}
```

Constructors:

1. JAVA provides a special method called constructor which enables an object to initialize itself when it is created.
2. Constructor name and class name should be same.
3. Constructor is called automatically when the object is created.
4. Person p1=new Person() → invokes the constructor Person() and Initializes the Person object p1.
5. Constructor does not return any return type (not even void).

There are two types of constructors.

Default constructor: A constructor which does not accept any parameters.

Parameterized constructor:A constructor that accepts arguments is called parameterized constructor.

Default constructor Example:

```
class Rectangle
{
        int length,bredth;  // Declaration of variables
        Rectangle() // Default Constructor method
        {
                System.out.println("Constructing Rectangle..");
                length=10;
                bredth=20;
        }
        int rectArea()
        {
                return(length*bredth);
        }
}
class Rect_Defa
{
        public static void main(String args[])
        {
                Rectangle r1=new Rectangle();
                System.out.println("Area of rectangle="+r1.rectArea());
        }
}
```

Note: If the default constructor is not explicitly defined, then system default constructor automatically initializes all instance variables to zero.

Parameterized constructor:

```
class Rectangle
{
        int length,bredth;  // Declaration of variables
        Rectangle(int x,int y) // Constructor method
        {
                length=x;
                bredth=y;
        }
        int rectArea()
        {
                return(length*bredth);
```

```
		}
}
class Rect_Para
{
		public static void main(String args[])
		{
			Rectangle r1=new Rectangle(5,10);
			System.out.println("Area of rectangle="+r1.rectArea());
		}
}
```

Overloaded Constructor Methods:

Writing more than one constructor with in a same class with different parameters is called constructor overloading.

Example:

```
class Addition

{

	int a,b;

	Addition()

	{

	  a=10;

	  b=20;

	}

	Addition(int a1,int b1)

	{

	   a=a1;

	   b=b1;

	}

	void add()

	{

	      System.out.println("Addition of "+a+" and "+b+" is "+(a+b));


	}

}

class ConsoverLoading

{
        public static void main(String args[])
        {
                        Addition obj=new Addition();
                        obj.add(); //output:30
                        Addition obj2=new Addition(2,3);
                        obj2.add(); // output: 5
        }
}
```

Nested Classes: nested class is a class that is declared inside the class or interface. it can access all the members of the outer class, including private data members and methods.

Advantages:

1. Nested classes can access all the members (data members and methods) of the outer class, including private.
2. to develop more readable and maintainable code
3. Code Optimization

Syntax of Inner class

```
class Java_Outer_class
{
        //code
        class Java_Inner_class
        {
                //code
        }
}
```

<u>Types of Nested classes</u>

There are two types of nested classes non-static and static nested classes. The non-static nested classes are also known as inner classes.

- Non-static nested class (inner class)
    1. Member inner class
    2. Anonymous inner class
    3. Local inner class
- Static nested class

<u>Example: local inner class</u>

```
public class localInner1
{
        private int data=30;//instance variable
        void display()
        {
                class Local
                {
                  void msg()
                {
                        System.out.println(data);
                }
            }
            Local l=new Local();
            l.msg();
        }
        public static void main(String
        args[]){ localInner1 obj=new
        localInner1(); obj.display();
        }
}
```

Example: member inner class

```
class Outer
{
        int a=10;
        class Inner // member inner class
        {
                int b=20;
        }
}
class MemberInnerClass
{
        public static void main(String args[])
        {
                Outer m=new Outer();
                Outer.Inner n=m.new Inner();//syntax to create inner class object
                System.out.println(m.a+n.b);
        }
}
```

Example: member inner class

```
class Outer
{
        static int a=10;
        static void sample()
        {
                System.out.println("Hello I am method of outer class");
        }
        static class Inner//static Inner Class
        {
                int b=20;//instance variable
                void display()
                {
                        sample();
                        System.out.println("I am outer class varible:"+a);
                }
        }
}
class StaticInnerClass
{
        public static void main(String args[])
        {
                Outer.Inner m=new Outer.Inner();
                System.out.println("I am Inner class varible:"+m.b);
                m.display();
        }
}
```

Final Class and Methods:

Final Class: Classes declared as final cannot be inherited.

Syntax:

```
 final class A
{
 -----
 -----
 }
 class B extends A // cannot be inherited
{ ---
  -----
}
```

Example: (Error Will get while executing following program)

```
final class A
{
  int a=10;
}
class B extends A //can't inherit because class-A is defined as final class
{
  int b=20;
}
class Final_Class
{
 public static void main(String args[])
  {
        B obj=new B();
        System.out.println(obj.a+obj.b);
   }
}
```

Final Method:

- To prevent method riding final keyword is used.
- Methods declared as final cannot be overridden.

Example: (Attempt to override final methods leads to compile time error.)

```
class A
{
	int a=10;
	final void display()
	{
		System.out.println("I am display() of class-A");
		System.out.println("my value is:"+a);
	}
}
class B extends A
{
	int b=20;
	void display()   //can't be overriden
	{
		System.out.println("I am display() of class-B");
		System.out.println("my value is:"+b);
	}
}
class Final_Methods_Classes
{
	public static void main(String args[])
	{
	B b=new B();
	b.display();
	}
}
```

Passing Arguments by Value and by Reference

- There is only call by value in java, not call by reference but we can pass non-primitive datatype to function to see the changes done by callee function in caller function.
- If we call a method passing a value, it is known as call by value. The changes being done in the called method, is not affected in the calling method.

Example 1: Passing Primitive datatype to function

```
class Example
{
	int a=10;
	void change(int a) //called or callee function
	{
		a=a+100;
	}
}
class CallByValue
```

```
{
        public static void main(String args[]) //calling function
        {
                Example e=new Example();
                System.out.println("a value before calling change() :"+e.a); //10
                e.change(10); //call by value(passing primitive data type)
                System.out.println("a value after calling change() :"+e.a);  //10
        }
}
```

Example 1: Passing Primitive datatype to function

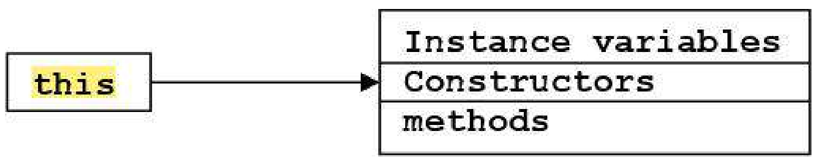
(Or)

Class Objects as Parameters in Methods:

```
class Example
{
        int a=10;
        void change(Example x) //called or callee function
        {
                x.a=x.a+100;
        }
}
class CallByValue
{
        public static void main(String args[]) //calling function
        {
                Example e=new Example();
                System.out.println("a value before calling change() :"+e.a); //10
                e.change(e); //call by value(passing non primitive data type) (or) Class Objects as Parameters in Methods
                System.out.println("a value after calling change() :"+e.a);  //110
        }
}
```

'this' keyword:

- 'this' is a keyword that referes to the object of the class where it is used.
- When an object is created to a class, a default reference is also created internally to the object.

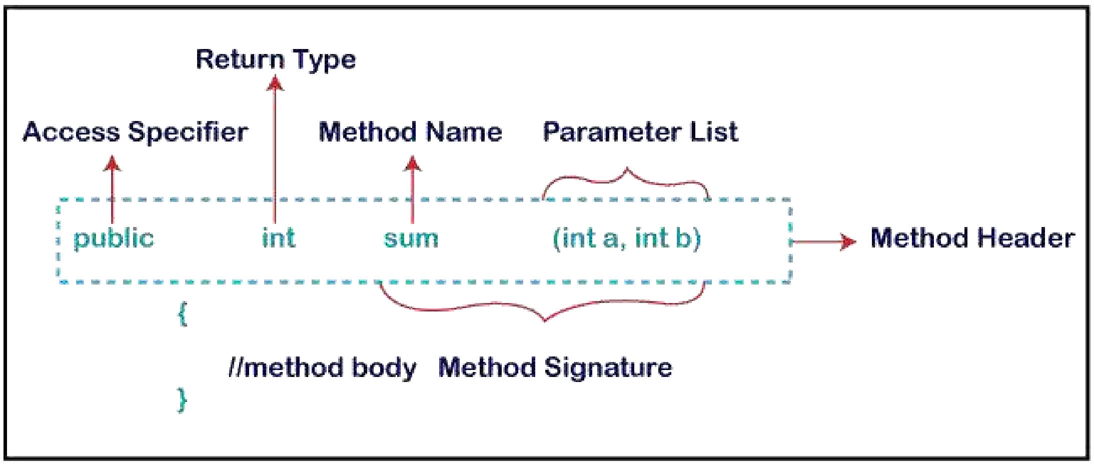


```
class Sample
{
  private int x;

  Sample()
   {
          this(10); // calls parameterized constructor and sends 10
          this.access(); //calls present class method
   }
  Sample(int x)
   {
          this.x=x; // referes present class reference variable
   }
  void access()
   {
          System.out.println("X ="+x);
   }
}
class ThisDemo
{
 public static void main(String[] args)
 {
           Sample s=new Sample();
 }
}
```

Methods in java:

- Method in Java is a collection of instructions that performs a specific task. It provides the reusability of code.
- Syntax of Defining a Method in java:

Method Declaration



Types of Method

There are two types of methods in Java:

- Predefined Method
- User-defined Method

Predefined method:

In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the standard library method or built-in method. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are length(), equals(), compareTo(), sqrt(), etc. When we call any of the predefined methods in our program, a series of codes related to the corresponding method runs in the background that is already stored in the library.

Each and every predefined method is defined inside a class. Such as print() method is defined in the java.io.PrintStream class. It prints the statement that we write inside the method. For example, print("Java"), it prints Java on the console.

Example:

```
public class Demo

{

        public static void main(String[] args)

        {

                // using the max() method of Math class

                System.out.print("The maximum number is: " + Math.max(9,7));

        }

}
```

User-defined Method:

The method written by the user or programmer is known as a user-defined method. These methods are modified according to the requirement.

Example:

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

```
//user defined method
public static void findEvenOdd(int num)
{
        //method body
        if(num%2==0)
                System.out.println(num+" is even");
        else
        System.out.println(num+" is odd");
}
public class EvenOdd
{
        public static void main (String args[])
        {
                //creating Scanner class object
                Scanner scan=new Scanner(System.in);
                System.out.print("Enter the number: ");
                //reading value from the user
                int num=scan.nextInt();
                //method calling
                findEvenOdd(num);
        }
}
```

Overloaded Methods or Method Overloading:

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading.

Example:

```
class Method
{
        int add(int a,int b)
        {
                System.out.println("I am Integer method");
                return a+b;
        }
        float add(float a,float b,float c)
        {
                System.out.println("I am float method");
                return a+b+c;
        }
        int add(int a,int b,int c)
        {
                System.out.println("I am Integer method");
                return a+b+c;
        }
```

```
	float add(float a,float b)
	{
		System.out.println("I am float method");
		return a+b;
	}
}
class MethodOverLoad
{
	public static void main(String args[])
	{
		Method m=new Method();
		System.out.println(m.add(10,20));
		System.out.println(m.add(10.2f,20.4f,30.5f));
		System.out.println(m.add(10,20,30));
		System.out.println(m.add(10.2f,20.4f));

	}
}
```

Recursive Methods:

- A method in java that calls itself is called recursive method.
- It makes the code compact but complex to understand.

Syntax:

```
returntype methodname()
{
	//code to be executed
	methodname();//calling same method
}
```

Example:

```
public class Recursion
{
	static int count=0;
	static void p()
	{
		count++;
		if(count<=5)
		{
			System.out.println("hello "+count);
			p();
		}
	}
	public static void main(String[] args)
	{
		p();
	}
}
```

Nesting of Methods:

A method can be called by using only its name by another method of the same class that is called **Nesting of Methods**.

Syntax:

```
class Main
{
   method1()
  {

       // statements
   }

   method2()
   {
     // statements

    // calling method1() from method2()
     method1();
   }
   method3()
   {
    // statements

    // calling of method2() from method3()
    method2();
   }
}
```

Example:

```
public class NestingMethod
{
   public void a1()
  {
       System.out.println("****** Inside a1 method ******");
      // calling method a2() from a1() with parameters a
      a2();
   }
   public void a2()
   {
     System.out.println("****** Inside a2 method ******");
   }
   public void a3()
   {
     System.out.println("****** Inside a1 method ******");
      a1();
   }
   public static void main(String[] args)
   {
      // creating the object of class
```

```
        NestingMethod n=new NestingMethod();

        // calling method a3() from main() method
        n.a3(a, b);
    }
}
```
Overriding Method: If subclass has the same method as declared in the parent class, it is known as method overriding.

Uses: runtime polymorphism

Rules:

- There must be a IS-A relationship(inheritance).
- The method must have same method signature as in the parent class.

Example:

```
class SuperClass
{
        void calculate(double x)
        {
                        System.out.println("Square value of X is: "+(x*x));
        }
}
class SubClass extends SuperClass
{
        void calculate(double x)
        {
                        super();
                        System.out.println("Square root of X is: "+Math.sqrt(x));
        }
}
class MethodOver
{
        public static void main(String args[])
        {
                        SubClass s=new SubClass();
                        s.calculate(2.5);
        }
}
```

Note: when a super class method is overridden by the sub class method calls only the sub class method and never calls the super class method. We can say that sub class method is replacing super class method.

3.1 ARRAYS

3.1.1 Introduction:

An array is a group of like-typed variables that are referred to by a common name. Arrays of any type can be created and may have one or more dimensions. A specific element in an array is accessed by its index. Arrays offer a convenient means of grouping related information.

3.1.2 Declaration and Initialization of Arrays

A *one-dimensional array* is, essentially, a list of like-typed variables. To create an array, you first must create an array variable of the desired type.

Declaration:

The general form of a one-dimensional array declaration is
type var-name[ ];

Here, *type* declares the element type (also called the base type) of the array. The element type determines the data type of each element that comprises the array. Thus, the element type for the array determines what type of data the array will hold.

For example, the following declares an array named **month_days** with the type "array of int":
int month_days[];

Although this declaration establishes the fact that **month_days** is an array variable, no array actually exists. In fact, the value of **month_days** is set to **null**, which represents an array with no value. To link **month_days** with an actual, physical array of integers, you must allocate one using **new** and assign it to **month_days**. **new** is a special operator that allocates memory.

The general form of **new** as it applies to one-dimensional arrays appears as follows:
array-var = **new** *type* [*size*];

Here, *type* specifies the type of data being allocated, *size* specifies the number of elements in the array, and *array-var* is the array variable that is linked to the array. That is, to use **new** to allocate an array, you must specify the type and number of elements to allocate. The elements in the array allocated by **new** will automatically be initialized to zero (for numeric types), **false** (for **boolean**), or **null**.

This example allocates a 12-element array of integers and links them to **month_days**:
month_days = new int[12];

After this statement executes, **month_days** will refer to an array of 12 integers. Further, all elements in the array will be initialized to zero.

Let’s review: Obtaining an array is a two-step process. First, you must declare a variable of the desired array type. Second, you must allocate the memory that will hold the array, using **new**, and assign it to the array variable. Thus, in Java all arrays are dynamically allocated.

A program that creates an array of the number of days in each month:

```
// Demonstrate a one-dimensional array.
class Array {
public static void main(String args[])
{ int month_days[];
month_days = new int[12];
month_days[0] = 31;  month_days[1] = 28;  month_days[2] = 31;  month_days[3] = 30;
month_days[4] = 31;  month_days[5] = 30;  month_days[6] = 31;  month_days[7] = 31;
month_days[8] = 30; month_days[9] = 31; month_days[10] = 30; month_days[11] = 31;
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, it prints the number of days in April. As mentioned, Java array indexes start with zero, so the number of days in April is month_days[3] or 30.

Initialization:

Arrays can be initialized when they are declared. The process is much the same as that used to initialize the simple types. An *array initializer* is a list of comma-separated expressions surrounded by curly braces. The commas separate the values of the array elements. The array will automatically be created large enough to hold the number of elements you specify in the array initializer. There is no need to use **new**.

For example, to store the number of days in each month, the following code creates an initialized array of integers:

```
// An improved version of the previous program.
class AutoArray {
public static void main(String args[ ]) {
int month_days[ ] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };
                                   or
int month_days[ ] = new int [ ] { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }; //you may use new
System.out.println("April has " + month_days[3] + " days.");
}
}
```

When you run this program, you see the same output as that generated by the previous version.

Java strictly checks to make sure you do not accidentally try to store or reference values outside of the range of the array. The Java run-time system will check to be sure that all array indexes are in the correct range. For example, the run-time system will check the value of each index into **month_days** to make sure that it is between 0 and 11 inclusive. If you try to access elements outside the range of the array (negative numbers or numbers greater than the length of the array), you will cause a run-time error.

Alternative Array Declaration Syntax:

Java also supports the following type of code for array declaration. In this declaration the type is followed by square bracket and the name or identifier of the array follows the square bracket.

type [ ] identifier; //one dimensional

type [ ] [ ] identifier; //two dimensional

Example: *int [ ] num;*

NOTE: Use of a square bracket before and after the identifier will create a multi-dimensional array.

int [ ] num [ ];

the above declaration is equivalent to

int [ ] [ ] num; ***or*** *int num [ ] [ ];*

3.1.3 Storage of Array in Computer Memory

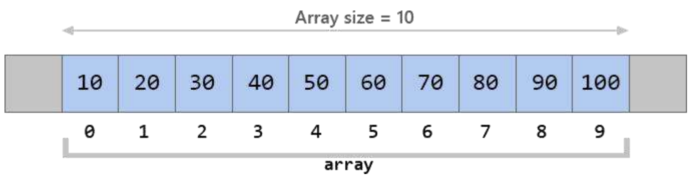
The operator new which is a keyword, allocates memory for storing the array elements.

For example with the following declaration

int [] num = new int [4];

The compiler allocates 4 memory spaces each equal to 4 bytes for storing the int type values of elements of array numbers. When an array is created as above, elements of the array are automatically initialized to 0 by the compiler.

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



A two dimensional array may be declared and initialized as

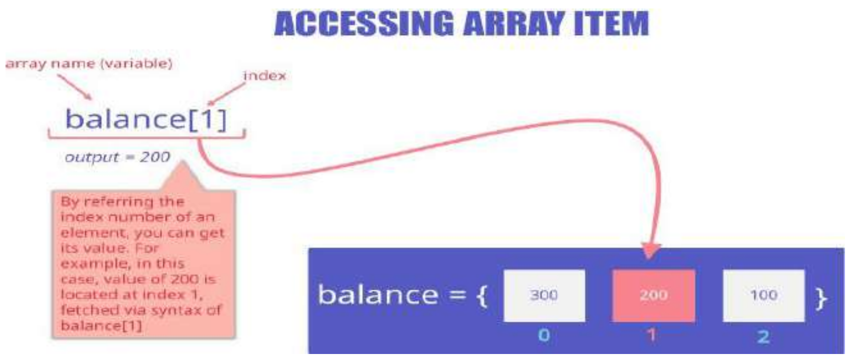
int [ ] [ ] num = new int [ ] [ ] ({1,2,3},{4,5,6}};

or

int [ ] [ ] num = ({1,2,3},{4,5,6}};

3.1.4 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.


ACCESSING ARRAY ITEM
array name (variable)
index
balance[1]
output = 200
By referring the index number of an element, you can get its value. For example, in this case, value of 200 is located at index 1, fetched via syntax of balance[1]
balance = { 300 200 100 }
0
1
2

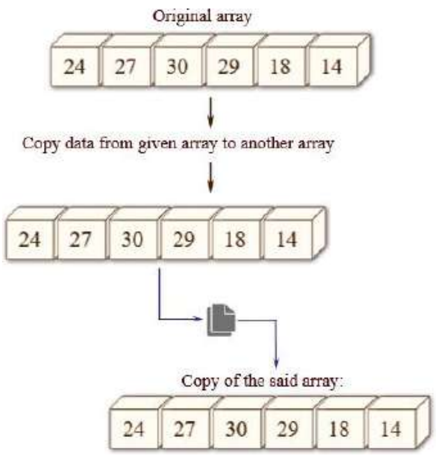
3.1.5 Operations on Array Elements

An array element is a variable of the type declared with the array. All the operations that are admissible for that type of a variable can be applied to an individual array element.

Similar to primitive types or objects of classes, the array may also be declared as a parameter of a method.
Acess_modifier type method_identifier (type array [], type other_parameter, ...)

3.1.6 Assigning Array to another Array

Unlike in C and C++ languages, in Java, an array may be assigned as a whole to another array of same data type. In this process, the second array identifier, in fact, becomes the reference to the assigned array. The second array is not a new array, instead only a second reference is created.



```
public class Test {
  public static void main(String[] args)
  {
     int a[] = { 1, 8, 3 };                    // Create an array b[] of same size as a[]
     int b[] = new int[a.length];      // Doesn't copy elements of a[] to b[], only makes b refer to same location
     b = a;
               // Change to b[] will also reflect in a[] as 'a' and 'b' refer to same location.
     b[0]++;
      System.out.println("Contents of a[] ");
     for (int i = 0; i < a.length; i++)
        System.out.print(a[i] + " ");
     System.out.println("\n\nContents of b[] ");
     for (int i = 0; i < b.length; i++)
        System.out.print(b[i] + " ");
  }
}
```

Output:

```
Contents of a[ ]
2 8 3
Contents of b[ ]
2 8 3
```

3.1.7 Dynamic Change of Array Size

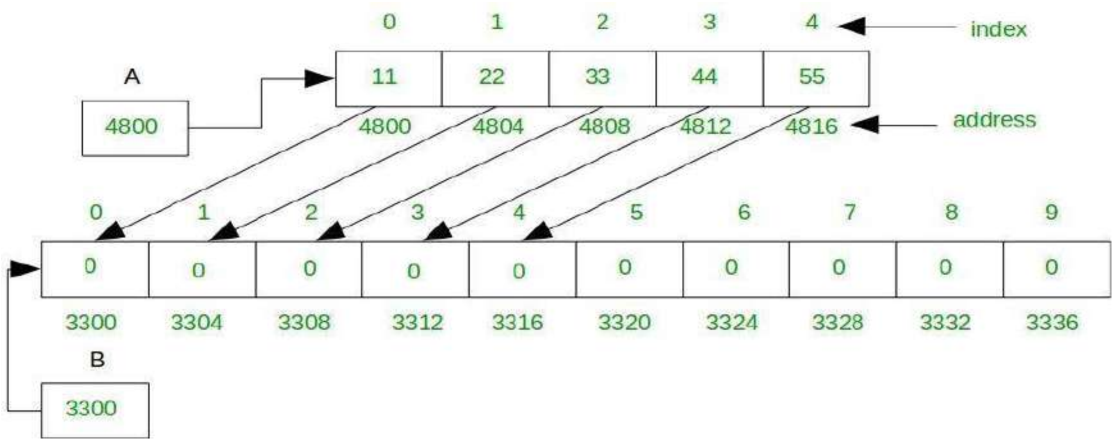
The number of elements (size) of the array may change during the execution of the program. This feature is unlike C and C++ wherein the array once declared is of fixed size, that is, the number of elements cannot be changed.

In Java, however, you may change the number of elements by dynamically retaining the array name. In this process, the old array is destroyed along with the values of elements.

Example:

int [ ] num = new int [5];

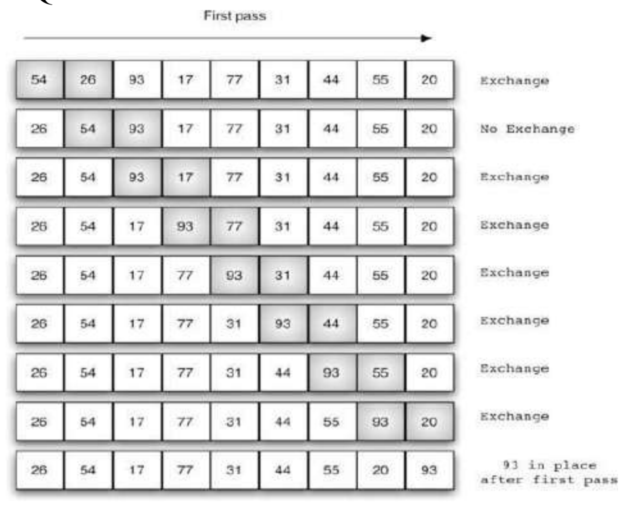
num = new int [10];



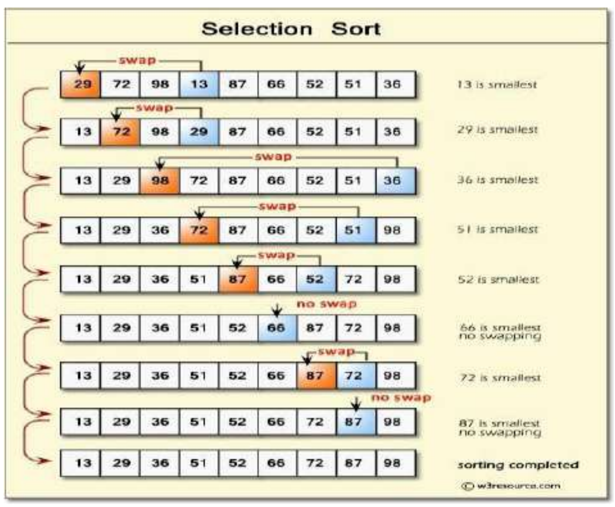
3.1.8 Sorting of Arrays

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

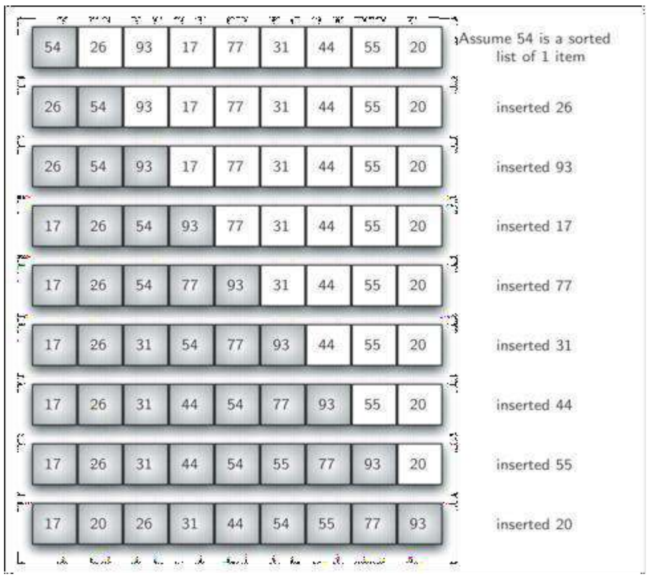
1. Bubble sort
2. Selection sort
3. Sorting by insertion method
4. Quick sort


First pass
54 26 93 17 77 31 44 55 20 Exchange
26 54 93 17 77 31 44 55 20 No Exchange
26 54 93 17 77 31 44 55 20 Exchange
26 54 17 93 77 31 44 55 20 Exchange
26 54 17 77 93 31 44 55 20 Exchange
26 54 17 77 31 93 44 55 20 Exchange
26 54 17 77 31 44 93 55 20 Exchange
26 54 17 77 31 44 55 93 20 Exchange
26 54 17 77 31 44 55 20 93 93 in place after first pass

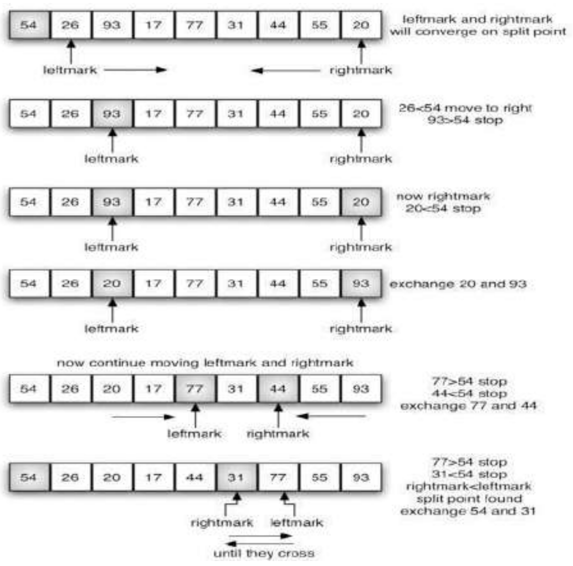
1. Bubble Sort


Selection Sort
swap
29 72 98 13 87 66 52 51 36 13 is smallest
13 72 98 29 87 66 52 51 36 29 is smallest
13 29 98 72 87 66 52 51 36 36 is smallest
13 29 36 72 87 66 52 51 98 51 is smallest
13 29 36 51 87 66 52 72 98 52 is smallest
no swap
13 29 36 51 52 66 87 72 98 66 is smallest no swapping
13 29 36 51 52 66 87 72 98 72 is smallest
13 29 36 51 52 66 72 87 98 87 is smallest no swapping
13 29 36 51 52 66 72 87 98 sorting completed
© w3resource.com

2. Selection Sort



3. Insertion Sort

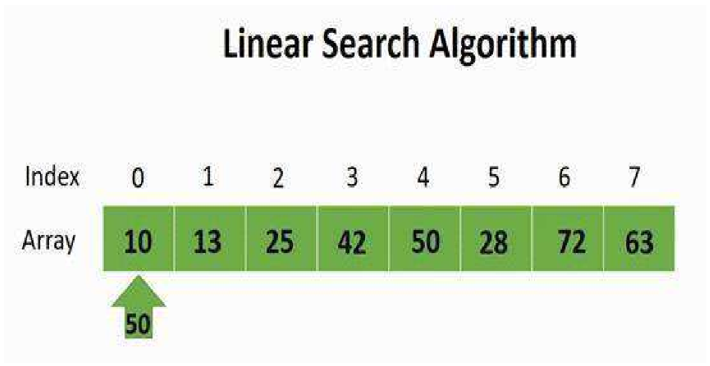


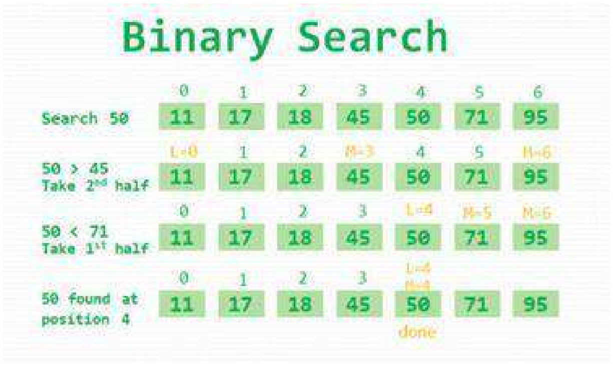
leftmark and rightmark will converge on split point
leftmark
rightmark
26<54 move to right 93>54 stop
now rightmark 20<54 stop
exchange 20 and 93
now continue moving leftmark and rightmark
77>54 stop 44<54 stop exchange 77 and 44
77>54 stop 31<54 stop rightmark<leftmark split point found exchange 54 and 31
until they cross

4. Quick Sort

3.1.9 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 retail reservation chart, air travel reservation chart, searching for a book in a library, and so on.
Two methods are employed: linear search and binary search for sorted arrays.





3.1.10 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 array, for comparing two arrays if they are equal or not, for filling a part of the full array with elements having a specified value, and for copying an array to another array. The sort method of arrays class is based on quick sort technique.

The methods are applicable to all primitive types as well as to class objects.

The class arrays are declared as
public class Arrays extends Object

Methods of Class Arrays

The methods of class arrays are as follows:

- Sort
- Binary Search
- Equals
- Fill
- CopyOf
- AsList
- ToString
- deepToString
- hashCode

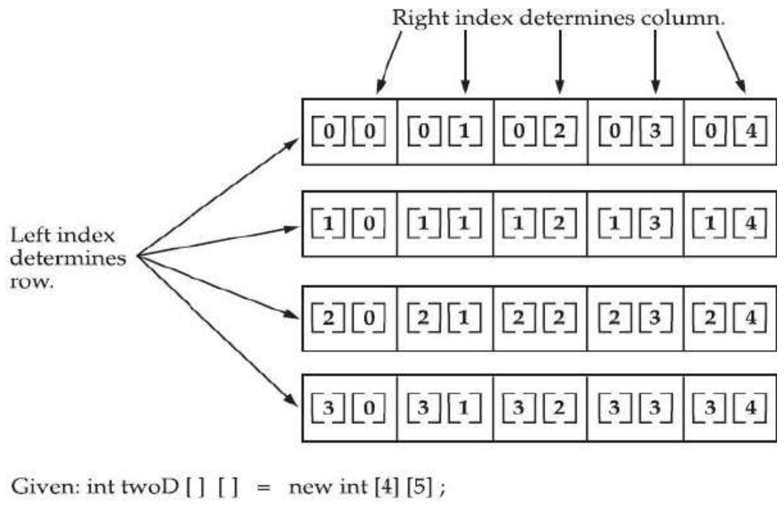
3.1.11 Two-dimensional or multi-dimensional Arrays

In Java, multidimensional arrays are actually arrays of arrays. These, as you might expect, look and act like regular multidimensional arrays. However, as you will see, there are a couple of subtle differences. To declare a multidimensional array variable, specify each additional index using another set of square brackets.

For example, the following declares a twodimensional array variable called twoD:

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

This allocates a 4 by 5 array and assigns it to twoD. Internally this matrix is implemented as an array of arrays of int. Conceptually, this array will look like the one shown in below figure.



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

The following program numbers each element in the array from left to right, top to bottom, and then displays these values:

```
// Demonstrate a two-dimensional array.
class TwoDArray {
        public static void main(String args[])
                { int twoD[][]= new int[4][5];
                int i, j, k = 0;
                        for(i=0; i<4; i++)
                                for(j=0; j<5; j++) {
                                        twoD[i][j] = k;
                k++;
            }
        for(i=0; i<4; i++) {
            for(j=0; j<5; j++)
                System.out.print(twoD[i][j] + " ");
            System.out.println();
        }
    }
}
```

This program generates the following output:

```
0 1 2 3 4
5 6 7 8 9
10 11 12 13 14
15 16 17 18 19
```

When you allocate memory for a multidimensional array, you need only specify the memory for the first (leftmost) dimension. You can allocate the remaining dimensions separately. For example, this following code allocates memory for the first dimension of twoD when it is declared. It allocates the second dimension manually.

```
    int twoD[][] = new int[4][];
    twoD[0] = new int[5];
    twoD[1] = new int[5];
    twoD[2] = new int[5];
    twoD[3] = new int[5];
```

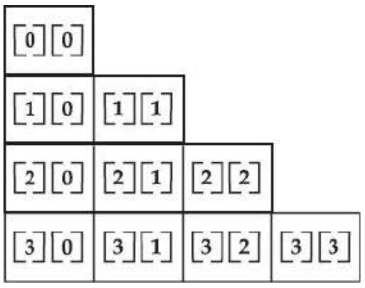
While there is no advantage to individually allocating the second dimension arrays in this situation, there may be in others. For example, when you allocate dimensions manually, you do not need to allocate the same number of elements for each dimension. As stated earlier, since multidimensional arrays are actually arrays of arrays, the length of each array is under your control. For example, the following program creates a two-dimensional array in which the sizes of the second dimension are unequal:

```
// Manually allocate differing size second dimensions.
class TwoDAgain {
    public static void main(String args[])
        { int twoD[][] = new int[4][];
        twoD[0] = new int[1];
        twoD[1] = new int[2];
        twoD[2] = new int[3];
        twoD[3] = new int[4];
            int  i,  j,  k  =  0;
            for(i=0; i<4; i++)
                for(j=0; j<i+1; j++) {
                    twoD[i][j] = k;
                        k++;
                }
            for(i=0; i<4; i++) {
                for(j=0; j<i+1; j++)
                    System.out.print(twoD[i][j] + " ");
                    System.out.println();
                }
        }
}
```

This program generates the following output:

```
0
1 2
3 4 5
6 7 8 9
```

The array created by this program looks like this:



The use of uneven (or irregular) multidimensional arrays may not be appropriate for many applications, because it runs contrary to what people expect to find when a multidimensional array is encountered. However, irregular arrays can be used effectively in some situations. For example, if you need a very large two-dimensional array that is sparsely populated (that is, one in which not all of the elements will be used), then an irregular array might be a perfect solution.

It is possible to initialize multidimensional arrays. To do so, simply enclose each dimension's initializer within its own set of curly braces. The following program creates a matrix where each element contains the product of the row and column indexes. Also notice that you can use expressions as well as literal values inside of array initializers.

```
// Initialize a two-dimensional array.
class Matrix {
    public static void main(String args[])
        { double m[][] = {
                        { 0*0, 1*0, 2*0, 3*0 },
                        { 0*1, 1*1, 2*1, 3*1 },
                        { 0*2, 1*2, 2*2, 3*2 },
                        { 0*3, 1*3, 2*3, 3*3 }
                    };
        int i, j;
            for(i=0; i<4; i++) {
                for(j=0; j<4; j++)
                    System.out.print(m[i][j] + " ");
                    System.out.println();
                }
    }
}
```

This program generates the following output:

```
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 0 0 0 0
0 1 2 3 4
0 2 4 6 8
0 3 6 9 12
0 0 0 0 0
0 2 4 6 8
0 4 8 12 16
0 6 12 18 24
```

Alternative Array Declaration Syntax

There is a second form that may be used to declare an array:

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

Here, the square brackets follow the type specifier, and not the name of the array variable.
For example, the following two declarations are equivalent:

```
int al[] = new int[3];
int[] a2 = new int[3];
```

The following declarations are also equivalent:

```
char twod1[][] = new char[3][4];
char[][] twod2 = new char[3][4];
```

This alternative declaration form offers convenience when declaring several arrays at the same time. For example,

```
int[] nums, nums2, nums3;     // create three arrays
```

creates three array variables of type int. It is the same as writing

```
int nums[], nums2[], nums3[];          // create three arrays
```

The alternative declaration form is also useful when specifying an array as a return type for a method.

3.1.12 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. As we know, an array can be declared in the following way:
int marks[ ] = new int [7];

the basic difference between arrays and vectors is that vectors are dynamically allocated, where as arrays are static. The size of vector can be changed as and when required, but this is not true for arrays.

The vector class is contained in java.util package. Vector stores pointers to the objects and not objects themselves. The following are the vector constructors.

```
Vector vec = new Vector( 5 );    // size of vector is
```

3.2 Inheritance

3.2.1 Introduction:

Inheritance is the technique which allows us to inherit the data members and methods from base class to derived class.

• Base class is one which always gives its features to derived classes.

• Derived class is one which always takes features from base class.

A Derived class is one which contains some of features of its own plus some of the data members from base class.

3.2.2 Process of inheritance

Syntax for INHERITING the features from base class to derived class:

```
class <clsname-2> extends <clsname-1>
{
 Variable declaration;
 Method definition;
};
```

Here, **clsname-1** and **clsname-2** represents derived class and base class respectively.
Extends is a keyword which is used for inheriting the data members and methods from base class to the derived class and it also improves functionality of derived class.

For example:

```
class c1;
{
 int a;
 void f1()
 {
 ………..;
 }
};
class c2 extends c1
{
 int b;
 void f2()
 {
 ………..;
 }
};
```

Whenever we inherit the base class members into derived class, when we creates an object of derived class, JVM always creates the memory space for base class members first and later memory space will be created for derived class members.

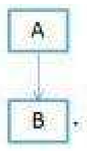
Example:

Write a JAVA program computes sum of two numbers using inheritance?

Answer:

```
class Bc
{
 int a;
};
class Dc extends Bc
{
         int b;
         void set (int x, int y)
         {
            a=x;
             b=y;
          }
         void sum ()
        {
             System.out.println ("SUM = "+(a+b));
         }
};
class InDemo
{
             public static void main (String k [])
            {

                         Dc do1=new Dc ();
                         do1.set (10,12);
                        do1.sum ();
             }
};
```

3.2.3 Types of Inheritances



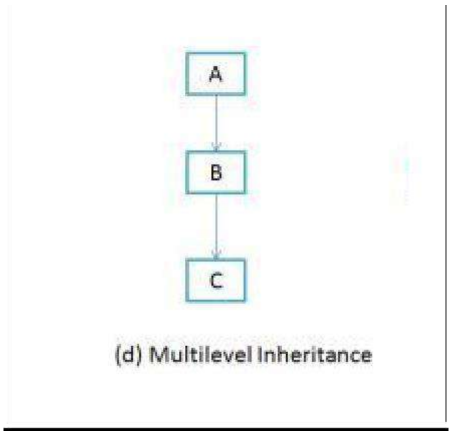
(a) Single Inheritance

<u>Single Inheritance</u>: It means when a base class acquired the properties of super class

```
class Animal
{
void eat()
{
 System.out.println("eating...");
}
}
class Dog extends Animal
{
void bark()
{
System.out.println("barking...");
}
}
class TestInheritance
{
public static void main(String args[])
{
Dog d=new Dog();
d.bark();
d.eat();
}
}
```

In this example base class is Dog and super class is Animal:

Multilevel Inheritance:



(d) Multilevel Inheritance

Multilevel inheritance refers to a mechanism in OO technology where one can inherit from a derived class, thereby making this derived class the base class for the new class. As you can see in below flow diagram C is subclass or child class of B and B is a child class of A

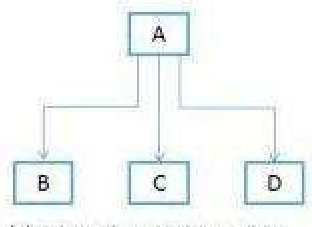
Example:

```
Class X
{
  public void methodX()
  {
    System.out.println("Class X method");
  }
}
```

```
Class Y extends X
{
public void methodY()
{
System.out.println("class Y method");
}
}
Class Z extends Y
{
  public void methodZ()
  {
    System.out.println("class Z method");
  }
  public static void main(String args[])
  {
    Z obj = new Z();
    obj.methodX(); //calling grand parent class method
    obj.methodY(); //calling parent class method
    obj.methodZ(); //calling local method
  }
}
```

Hierarchical Inheritance

In such kind of inheritance one class is inherited by many sub classes. In below example class B,C and D inherits the same class A. A is parent class (or base class) of B,C & D



(c) Hierarchical Inheritance

Example:

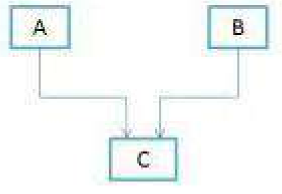
```
class A
{
  public void methodA()
  {
     System.out.println("method of Class A");
  }
}
class B extends A
{
  public void methodB()
  {
      System.out.println("method of Class B");
   }
}
class C extends A
{
  public void methodC()
   {
     System.out.println("method of Class C");
   }
}
class D extends A
{
  public void methodD()
   {
     System.out.println("method of Class D");
   }
}
class JavaExample
{
  public static void main(String args[])
   {
     B obj1 = new B();
     C obj2 = new C();
     D obj3 = new D();
   }
}
```

Output:

```
method of Class A
method of Class A
method of Class A
```

Multiple Inheritance:

Multiple Inheritance" refers to the concept of one class extending (Or inherits) more than one base class. The inheritance we learnt earlier had the concept of one base class or parent. The problem with "multiple inheritance" is that the derived class will have to manage the dependency on two base classes.

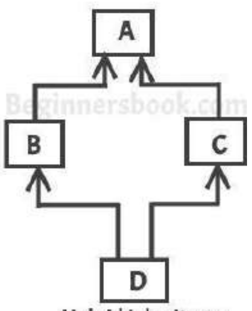


(b) Multiple Inheritance

Note: Multiple Inheritance is very rarely used in software projects. Using Multiple inheritance often leads to problems in the hierarchy. This results in unwanted complexity when further extending the class.

Hybrid Inheritance in Java

A hybrid inheritance is a combination of more than one types of inheritance. For example when class A and B extends class C & another class D extends class A then this is a hybrid inheritance, because it is a combination of single and hierarchical inheritance.



Hybrid Inheritance

The diagram is just for the representation, since multiple inheritance is not possible in java

```
class C
{
  public void disp()
  {
        System.out.println("C");
  }
}

class A extends C
{
  public void disp()
  {
        System.out.println("A");
  }
}

class B extends C
{
  public void disp()
  {
        System.out.println("B");
  }

}

class D extends A
{
   public void disp()
   {
          System.out.println("D");
   }
   public static void main(String args[]){

          D obj = new D();
          obj.disp();
   }
}
```

This example is just to demonstrate the hybrid inheritance in Java. Although this example is meaningless, you would be able to see that how we have implemented two types of inheritance(single and hierarchical) together to form hybrid inheritance.

Class A and B extends class C → Hierarchical inheritance
Class D extends class A → Single inheritance

3.2.4 Inhibiting Inheritance of Class Using Final

The final keyword in java is used to restrict the user. The java final keyword can be used in many context.
Final can be:
1. variable
2. method
3. class

The main purpose of using a class being declared as final is to prevent the class from being subclassed. If a class is marked as final then no class can inherit any feature from the final class.
Example:

```
final class Bike
{
}
 class Honda1 extends Bike
{
 void run()
{
System.out.println("running safely with 100kmph");
}
 public static void main(String args[])
{
 Honda1 honda= new Honda1();
 honda.run();
 }
}
```

Output:
Compile time error

Since class **Bike** is declared as final so the derived class **Honda** cannot extend **Bike**

3.2.5 Access Control and Inheritance

Although a subclass includes all of the members of its superclass, it cannot access those members of the superclass that have been declared as private.

```
class A
 {
int i; // public by default
private int j; // private to A
          void setij(int x, int y)
         {
                    i = x;
                    j = y;
          }
}
// A's j is not accessible here.
class B extends A
 {
          int total;
          void sum()
           {
                    total = i + j; // ERROR, j is not accessible here
          }
}
class Access
{
          public static void main(String args[])
           {
                    B subOb = new B();
                    subOb.setij(10,  12);
                    subOb.sum();
                    System.out.println("Total is " + subOb.total);
          }
}
```

This program will not compile because the reference to j inside the sum( ) method of B causes an access violation. Since j is declared as private, it is only accessible by other members of its own class. Subclasses have no access to it.

3.2.6 Application of Keyword Super

Super keyword is used for differentiating the base class features with derived class features. Super keyword is placing an important role in three places.

- variable level
- method level
- constructor level

Super at variable level:
Whenever we inherit the base class members into derived class, there is a possibility that base class members are similar to derived class members.

In order to distinguish the base class members with derived class members in the derived class, the base class members will be preceded by a keyword super.

For example:

```
class Bc
{
 int a;
};
class Dc extends Bc
{
 int a;
 void set (int x, int y)
 {
 super.a=x;
 a=y; //by default 'a' is preceded with 'this.' since 'this.' represents current class
 }
 void sum ()
 {
 System.out.println ("SUM = "+(super.a+a));
 }
};
class InDemo1
{
 public static void main (String k [])
 {

 Dc do1=new Dc ();
 do1.set (20, 30);
 do1.sum ();
 }
};
```

Super at method level:
Whenever we inherit the base class methods into the derived class, there is a possibility that base class methods are similar to derived methods.

To differentiate the base class methods with derived class methods in the derived class, the base class methods must be preceded by a keyword super.

Syntax for super at method level: super. base class method name

For example:
```
class Bc
{
 void display ()
 {
 System.out.println ("BASE CLASS - DISPLAY...");
 }
};
class Dc extends Bc
{
 void display ()
 {
super.display (); //refers to base class display method
 System.out.println ("DERIVED CLASS- DISPLAY...");
 }
};
class InDemo2
{
public static void main (String k [])
 {
 Dc do1=new Dc ();
 do1.display ();
 }
};
```

3.2.7 Constructor Method and Inheritance

Super at constructor level:
super() can be used to invoke immediate parent class constructor.

```
class A
{
   int i,j;
   A(int a,int b)
   {
    i=a;
    j=b;
   }
   void show()
   {
    System.out.println("i and j values are"+i+" "+j);
   }
}
class B extends A
{
  int k;
   B(int a, int b, int c)
   {
    super(a, b);//super class constructor
    k = c;
   }
    // display k – this overrides show() in A
   void show()
    {
    super.show();
    System.out.println("k: " + k);
    }

}

class Override
{

public static void main(String args[])
 {
   B subOb = new B(1, 2, 3);
   subOb.show(); // this calls show() in B

}
}
```

3.2.8 Method Overriding // Refer 2nd unit

3.2.9 Dynamic Method Dispatch // Refer 2nd unit

3.2.10 Abstract Classes:

In JAVA we have two types of classes. They are concrete classes and abstract classes.

• A concrete class is one which contains fully defined methods. Defined methods are also known as implemented or concrete methods. With respect to concrete class, we can create an object of that class directly.

- **An abstract class is one which contains some defined methods and some undefined methods. Undefined methods are also known as unimplemented or abstract methods. Abstract method is one which does not contain any definition.**

To make the method as abstract we have to use a keyword called abstract before the function declaration.

Syntax for ABSTRACT CLASS: abstract return_type method_name (parameters list);

```
// A Simple demonstration of abstract.
abstract class A
{
        abstract void callme();

// concrete methods are still allowed in abstract classes
        void callmetoo()
        {
                System.out.println("This is a concrete method.");
        }
}
class B extends A
 {
        void callme()
        {
                System.out.println("B's implementation of callme.");
        }
}
class AbstractDemo
 {
        public static void main(String args[])
        {
                B b = new B();
                b.callme();
                b.callmetoo();
        }
}
```

Notice that no objects of class A are declared in the program. As mentioned, it is not possible to instantiate an abstract class.

One other point: class A implements a concrete method called callmetoo( )

3.3 Interface in Java

3.3.1 Introduction

An interface in java is a blueprint of a class.
It has static constants and abstract methods. The interface in java is a mechanism to achieve abstraction. There can be only abstract methods in the java interface not method body.
It is used to achieve abstraction and multiple inheritance in Java.

- Using the keyword interface, you can fully abstract a class' interface from its implementation.
- That is, using interface, you can specify what a class must do, but not how it does it.
- Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body
- Variables can be declared inside of interface declarations.
- They are implicitly final and static, meaning they cannot be changed by the implementing class.
- They must also be initialized. All methods and variables are implicitly public.

3.3.2 Declaring an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name {
return-type method-name1(parameter-list);
return-type method-name2(parameter-list);
type final-varname1 = value;
type final-varname2 = value;
// ...
return-type method-nameN(parameter-list);
type final-varnameN = value;
}
```

Here is an example of an interface definition. It declares a simple interface that contains one method called callback( ) that takes a single integer parameter.

```
interface Callback
 {
void callback(int param);
}
```

3.3.3 Implementing Interfaces

Once an interface has been defined, one or more classes can implement that interface. To implement an interface, include the implements clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
class classname [extends superclass] [implements interface [,interface...]] {
// class-body
}
```

If a class implements more than one interface, the interfaces are separated with a comma.

Here is a small example class that implements the Callback interface shown earlier.

```
class Client implements Callback
 {
// Implement Callback's interface
public void callback(int p)
 {
System.out.println("callback called with " + p);
}
}
```

Notice that callback( ) is declared using the public access specifier.

It is both permissible and common for classes that implement interfaces to define additional members of their own. For example, the following version of Client implements callback( ) and adds the method nonIfaceMeth( ):

```
class Client implements Callback
 {
// Implement Callback's interface
public void callback(int p)
 {
System.out.println("callback called with " + p);
}
void nonIfaceMeth()
 {
System.out.println("Classes that implement interfaces " +
"may also define other members, too.");
}
}
```

Accessing Implementations Through Interface References:

You can declare variables as object references that use an interface rather than a class type. Any instance of any class that implements the declared interface can be referred to by such a variable. When you call a method through one of these references, the correct version will be called based on the actual instance of the interface being referred to.

The following example calls the callback( ) method via an interface reference variable:

```
class TestIface
 {
public static void main(String args[])
 {
Callback c = new Client();
c.callback(42);
}
}
```

The output of this program is shown here:
callback called with 42

3.3.4 Multiple Interfaces

Multiple inheritance in java can be achieved through interfaces:
Example:

```
interface Printable
{
void print();
}

interface Showable
{
void show();
}

class A7 implements Printable,Showable
{
        public void print()
        {
                System.out.println("Hello");
        }
        public void show()
        {
                System.out.println("Welcome");
        }
        public static void main(String args[])
        {
                A7 obj = new A7();
                obj.print();
                obj.show();
        }
}
```

Output: Hello
Welcome

3.3.5 Nested Interfaces:

An interface i.e. declared within another interface or class is known as nested interface. The nested interfaces are used to group related interfaces so that they can be easy to maintain. The nested interface must be referred by the outer interface or class. It can't be accessed directly.

There are given some points that should be remembered by the java programmer.

- Nested interface must be public if it is declared inside the interface but it can have any access modifier if declared within the class.
- Nested interfaces are declared static implicitly.

Example:

```
interface Showable
{
  void show();
          interface Message
          {
            void msg();
          }
}
class TestNestedInterface1 implements Showable.Message
{
          public void msg()
          {
          System.out.println("Hello nested interface");
          }

 public static void main(String args[])
{
  Showable.Message message=new TestNestedInterface1();//upcasting here
  message.msg();
 }
}
```

As you can see in the above example, we are acessing the Message interface by its outer interface Showable because it cannot be accessed directly. It is just like almirah inside the room, we cannot access the almirah directly because we must enter the room first.

3.3.6 Inheritance of Interfaces

One interface can inherit another by use of the keyword extends. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
// One interface can extend another.
interface A
 {
        void meth1();
        void meth2();
}
// B now includes meth1() and meth2() -- it adds meth3().
interface B extends A
{
        void meth3();
}
// This class must implement all of A and B
class MyClass implements B
 {
        public void meth1()
        {
                System.out.println("Implement meth1().");
        }
        public void meth2()
         {
                System.out.println("Implement meth2().");
        }
        public void meth3()
        {
                System.out.println("Implement meth3().");
        }
}
class IFExtend
 {
        public static void main(String arg[])
         {
                MyClass ob = new MyClass();
                ob.meth1();
                ob.meth2();
                ob.meth3();
        }
}
```

3.3.7 Default methods in interfaces

Before Java 8, interfaces could have only abstract methods. The implementation of these methods has to be provided in a separate class. So, if a new method is to be added in an interface, then its implementation code has to be provided in the class implementing the same interface. To overcome this issue, Java 8 has introduced the concept of default methods which allow the interfaces to have methods with implementation without affecting the classes that implement the interface

Example:

```
interface TestInterface
{
   // abstract method
   public void square(int a);

   // default method
   default void show()
   {
     System.out.println("Default Method Executed");
   }
}

class TestClass implements TestInterface
{
   // implementation of square abstract method
   public void square(int a)
   {
       System.out.println(a*a);
   }

   public static void main(String args[])
   {
       TestClass d = new TestClass();
       d.square(4);

       // default method executed
       d.show();
   }
}.
```

```
Output:
16
 Default Method Executed
```

Default methods are also known as defender methods or virtual extension methods.

3.3.8 Static methods in interfaces

The interfaces can have static methods as well which is similar to static method of classes.

Example:

```
interface TestInterface
{
   // abstract method
```

```
    public void square (int a);

    // static method
    static void show()
    {
        System.out.println("Static Method Executed");
    }
}

class TestClass implements TestInterface
{
    // Implementation of square abstract method
    public void square (int a)
    {
        System.out.println(a*a);
    }

    public static void main(String args[])
    {
        TestClass d = new TestClass();
        d.square(4);

        // Static method executed
        TestInterface.show();
    }
}
```

Output:

```
16
 Static Method Executed
```

3.3.9 Functional Interfaces

A functional interface is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface.

A functional interface can have any number of default methods. Runnable, ActionListener, Comparable are some of the examples of functional interfaces.

3.3.10 FunctionalInterface annotation is used to ensure that the functional interface can’t have more than one abstract method. In case more than one abstract methods are present, the compiler flags an ‘Unexpected @FunctionalInterface annotation’ message.

```
@FunctionalInterface
interface Square
{
    int calculate(int x);
}

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

     // lambda expression to define the calculate method
     Square s = (int x)->x*x;

     // parameter passed and return type must be
     // same as defined in the prototype
     int ans = s.calculate(a);
     System.out.println(ans);
   }
}
```

Output:

```
25
```

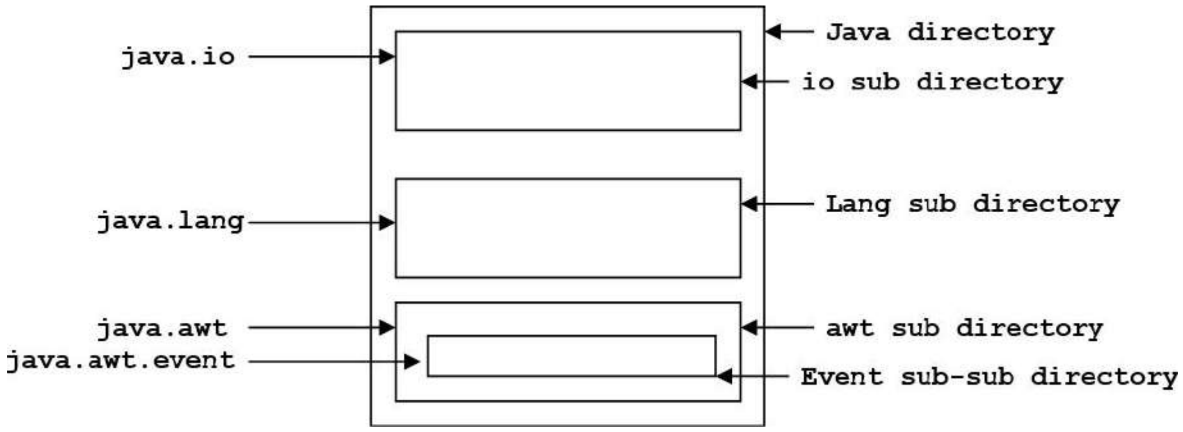
UNIT IV

Packages and Java Library: Introduction, Defining Package, Importing Packages and Classes into Programs, Path and Class Path, Access Control, Packages in Java SE, Java.lang Package and its Classes, Class Object, Enumeration, class Math, Wrapper Classes, Auto-boxing and Auto-unboxing, Java util Classes and Interfaces, Formatter Class, Random Class, Time Package, Class Instant (java. time. Instant), Formatting for Date/Time in Java, Temporal Adjusters Class, Temporal Adjusters Class.

Exception Handling: Introduction, Hierarchy of Standard Exception Classes, Keywords throws and throw, try, catch, and finally Blocks, Multiple Catch Clauses, Class Throwable, Unchecked Exceptions, Checked Exceptions, try-with-resources, Catching Subclass Exception, Custom Exceptions, Nested try and catch Blocks, Rethrowing Exception, Throws Clause.

Packages and Java Library: Introduction

- It is necessary in software development to create several classes and interfaces.
- After creating these classes and interfaces, it is better if they are divided into some groups depending on their relationship.
- So these classes and interfaces are stored in some directory.
- This directory or folder is also known as package.



Advantages of packages:

- packages hide the classes and interfaces in a separate sub directory, so accidental deletion of classes and interfaces will not take place.
- Two classes in two different packages can have the same name.
- A group of packages is called a library. The reusability nature of packages makes programming easy.

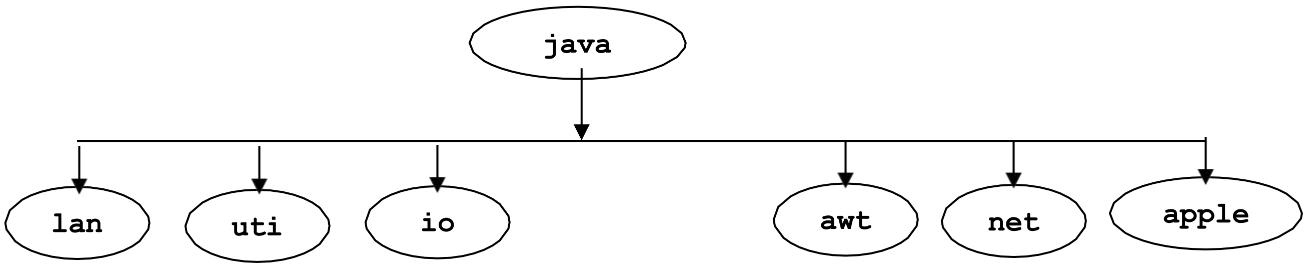
There are two types of packages.

- o Built-in packages/Java API packages.
- o User Defined Packages.

Java API Packages:

- o Java API provides a large number of classes grouped into different packages according functionality.

Packages from java API are:



Package name	Contents
java.lang	lang package contains language support classes. There classes are imported by java compiler automatically for its usage. lang package contains classes for primitive types, string, threads, exceptions, etc…….
java.util	Utility classes such as vectors, hash tables, etc…….
java.io	Contains classes for input/output of data.
java.awt	Contains for graphical user interface, classes for windows, buttons, lists, menus, etc…..
java.net	Classes for networking, contains classes for communicating local computers and internet servers.
Java.applet	Classes for creating and implementing applets

Defining Package

- A package is created using keyword package
    - Syntax:

```
package <package>;
```

- Package statement must be the first statement in a java source file.
    - Ex:

```
package studentspack;  // package declaration
class Student          // class declaration
{
 -----
 -----
}
```

Create a package which contans Addition class in that.

```
package pack;

public class Addition

{

 private double a,b;
 public Addition(double a,double b)
 {
  this.a=a;
  this.b=b;
 }
 public void sum()
 {
  System.out.println("Sum="+(a+b));
 }
}
```

<u>Compiling the program:</u>

```
> javac –d . Addition.java
```

- -d tells java compiler to create a separate sub directory and place the .class file there.
- Dot (.) indicates that the package should be created in the current directory.

<u>Program to use Addition.class from mypack:</u>

```
class UsePack
{
 public static void main(String args[])
 {
  pack.Addition obj=new pack.Addition(10,20.5);
  obj.sum();
 }
}
```

<u>Note:</u>

- Instead of referring the package name every time, we can import the package like this
    - import pack.Addition;
- Then the program can be written asS....

```
import pack.Addition;
class UsePack
{
 public static void main(String args[])
 {
  Addition obj=new Addition(10,20.5);
  obj.sum();
 }
}
```

Importing Packages and Classes into Programs

There are two ways of accessing classes stored in a package.

1. Fully qualified class name
2. Using Import Statement

1. Fully qualified class name:

```
double y=java.lang.Math.sqrt(x);
```

2. Using Import Statement:

syntax:-

```
        import packagename.classname;

                or

        import packagename.*;
```

```
import java.awt.Font;
```

- imports Font class in our program.

(or)

```
import java.awt.*;
```

- imports all classes of awt package in our program.

Path and Class Path

PATH	CLASSPATH
PATH is an environment variable.	CLASSPATH is an environment variable that tells the java compailer where to look for class files to import. Generally CLASSPATH is set to a directory or JAR file.
It is used by the operating system to find the executable files (.exe).	It is used by Application ClassLoader to locate the .class file.
You are required to include the directory which contains .exe files.	You are required to include all the directories which contain .class and JAR files.
PATH environment variable once set, cannot be overridden.	The CLASSPATH environment variable can be overridden by using the command line option -cp or -CLASSPATH to both javac and java command.

How to Set CLASSPATH in Windows Using Command Prompt

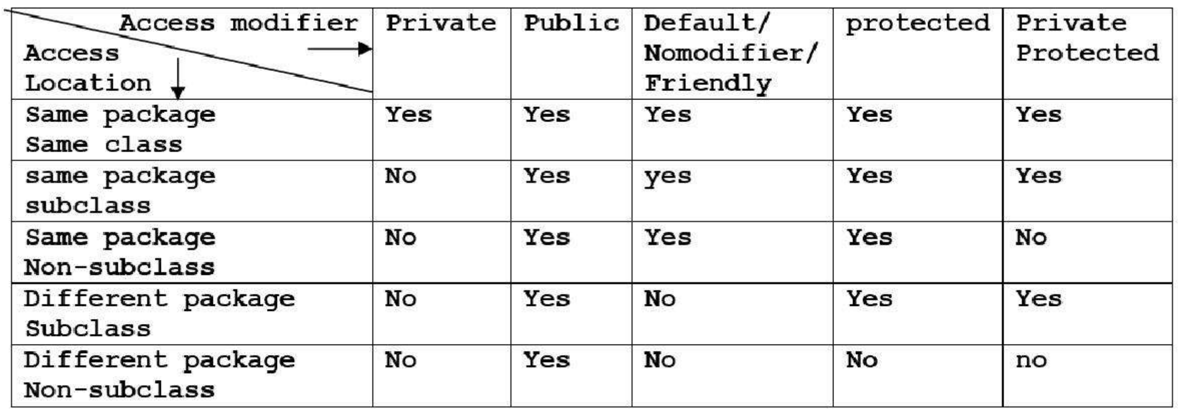
CLASSPATH is an environment variable that tells the java compailer where to look for class files to import. Generally CLASSPATH is set to a directory or JAR file.

Type the following command in your Command Prompt and press enter.

```
set CLASSPATH=%CLASSPATH%;C:\Program Files\Java\jre1.8\rt.jar;
```

In the above command, the set is an internal DOS command that allows the user to change the variable value. CLASSPATH is a variable name. The variable enclosed in percentage sign (%) is an existing environment variable. The semicolon is a separator, and after the (;) there is the PATH of rt.jar file.

Access Control:



Access modifier → / Access Location ↓	Private	Public	Default/ Nomodifier/ Friendly	protected	Private Protected
Same package Same class	Yes	Yes	Yes	Yes	Yes
same package subclass	No	Yes	yes	Yes	Yes
Same package Non-subclass	No	Yes	Yes	Yes	No
Different package Subclass	No	Yes	No	Yes	Yes
Different package Non-subclass	No	Yes	No	No	no

- **This table only applies to members of classes**
- **A class has only two access levels**
    - **Public - accessible any where**
    - **Default - accessible with in the same package.**

Packages in Java SE:

Java Standard Edition provides 14 packages namely –

- **applet** – This package provides classes and methods to create and communicate with the applets.
- **awt**– This package provides classes and methods to create user interfaces.
- **io**– This package contains classes and methods to read and write data standard input and output devices, streams and files.
- **lang**– This package contains the fundamental classes, methods, and, interfaces of Java language.
- **math**– This package contains classes and methods which helps you to perform arithmetic operations using the Java language.
- **net**– This package provides classes to implement networking applications.
- **rmi**– This package provides classes, methods, and interfaces for Remote Method Invocation.
- **security**– This package provides classes and interfaces for security framework.
- **sql**– This package provides classes and methods to access and process the data stored in a data source.
- **text**– This package provides classes and interfaces to handle text.
- **time**– This package provides API for dates, times, instants, and durations.
- **util**– This package contains collection framework, collection classes, classes related to date and time, event model, internationalization, and miscellaneous utility classes.

Java. lang Package and its Classes:

The most important classes are of lang are

- Object, which is the root of the class hierarchy, and Class, instances of which represent classes at runtime.
    - protected Object clone()
    - boolean equals(Object obj)
    - protected void finalize()
    - Class getClass()
    - int hashCode()
    - void notify()
    - void notifyAll()
    - void wait()
    - String toString()
- The wrapper classes
    - Boolean
    - Character
    - Integer
    - Short
    - Byte
    - Long
    - Float
    - Double
- The classes String, StringBuffer, and StringBuilder similarly provide commonly used operations on character strings.
- Class Throwable encompasses objects that may be thrown by the throw statement. Subclasses of Throwable represent errors and exceptions.

Enumeration in java:

```
class EnumExample
{
        //defining enum within class
        public enum Season { WINTER, SPRING, SUMMER, FALL }
        public static void main(String[] args)
        {
                //printing all enum
                for (Season s : Season.values())
                {
                        System.out.println(s);
                }
                System.out.println("Value of WINTER is: "+Season.valueOf("WINTER"));
                System.out.println("Index of WINTER is: "+Season.valueOf("WINTER").ordinal());
                System.out.println("Index of SUMMER is: "+Season.valueOf("SUMMER").ordinal());

}}
```

class Math:

```
class MathClass
{
	public static void main(String args[])
	{
		System.out.println("Absolute value-"+Math.abs(-90));
		System.out.println("Minimum value-"+Math.min(90,20));
		System.out.println("Maximum value-"+Math.max(90,20));
		System.out.println("round value-"+Math.round(79.52));
		System.out.println("sqrtroot value-"+Math.sqrt(25));
		System.out.println("cuberoot value-"+Math.cbrt(125));
		System.out.println("power value-"+Math.pow(2,5));
		System.out.println("ceil value-"+Math.ceil(2.2));
		System.out.println("ceil value-"+Math.floor(2.8));
		System.out.println("floorDiv value-"+Math.floorDiv(25,3));
		System.out.println("random value-"+Math.random());
		System.out.println("rint value-"+Math.rint(81.68));
		System.out.println("subtractExact value-"+Math.subtractExact(732, 190));
		System.out.println("multiplyExact value-"+Math.multiplyExact(732, 190));
		System.out.println("incrementExact value-"+Math.incrementExact(732));
		System.out.println("decrementExact value-"+Math.decrementExact(732));
		System.out.println("negateExact value-"+Math.negateExact(90));
	}
}
```

Wrapper Classes:

Primitive data types can be converted into object types by using the wrapper classes contained in java.lang package.

Below table shows the simple data types and their corresponding wrapper classes:

Simple type	Wrapper class
int	Integer
char	Character
float	Float
double	Double
long	Long
boolean	Boolean

The wrapper classes have a number of unique methods for handling primitive data types and objects.

Converting primitive type to object type using Constructor methods:

Constructor	Conversion Action
Integer x=new Integer(i)	Primitive integer to Integer object
Float x=new Float(f)	Primitive float to Float object
Double x=new Double(d)	Primitibe double to Double object
Long x=new Long(l)	Primitive long to Long object

Converting object numbers to Primitive numbers using typeValue() method:

Method	Conversion Action
int i=x.intValue()	Object to primitive integer
float f=x.floatValue()	Object to primitive float
long l=x.longValue()	Object to primitive long
double d=x.doubleValue()	Object to primitive double

Converting numbers to string using toString() method:

Method	Conversion Action
str=Integer.toString(i)	primitive integer to String
str=Float.toString(f)	Primitive float to String
str=Double.toString(d)	Primitive double to String
str=Long.toString(l)	Primitive long to String

Converting string Objects to Numeric objects using static method valueOf().

Method	Conversion Action
X=Double.valueOf(str)	Converts string to Double object
X=Float.valueOf(str)	Converts string to Float object
X=Integer.valueOf(str)	Converts string to Integer Object
X=Long.valueOf(str)	Converts string to Long object

Converting Numeric string to primitive type using parsing methods:

Method	Conversion Action

int i=parseInt(Str)	Converts string to integer
long l=parseLong(str)	Converts string to long

Ex:-

A simple integer can be changed to object type:

```
int x=100;
Integer ob1=new Integer(x);
```

Now ob1 is an integer object which contains the 100 as its value.

Object can be converted into primitive type:

```
int x=ob1.intValue();
```

now x will contain the value 100.

Byte class:

The following are the Byte class constructures

```
Byte(byte N);
Byte(String str);
```

Ex:

```
Byte b1=new Byte(100);
Byte b2=new Byte("Hello");
```

Note:

Byte class constructor accepts number as well as string as its parameters.

The following are the methods of Byte class:

byte byteValue() – returns byte value of the object.

Int compareTo(Byte b) – it accepts byte object and compares with invoking object.

static byte parseByte(String str) throws NumberFormatException

- it accepts string object and converts into byte. It is static so it can be called directly using the class name.

static Byte valueOf(String str)throws NumberFormatException

- it accepts a String object and converts into Byte class object.

Integer class:

The following are the Integer class constructures

Integer(int N);

Integer(String str) throws NumberFormatException

The following are the methods of Integer class:

int intValue() – To get integer from Integer object.

int compareTo() -

Auto-boxing and Auto-unboxing:

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing. This is the new feature of Java5. So java programmer doesn't need to write the conversion code.

Advantage of Autoboxing and Unboxing:

No need of conversion between primitives and Wrappers manually so less coding is required.

AutoBoxing Example:

```
class BoxingExample1
{
        public static void main(String args[])
        {
          int a=50;
          Integer a2=new Integer(a);//Boxing
         Integer a3=5;//Boxing
         System.out.println(a2+" "+a3);
        }
}
```

UnBoxing Example

```
class UnboxingExample1
{
        public static void main(String args[])
        {
           Integer i=new Integer(50);
           int a=i;
           System.out.println(a);
        }
}
```

Java util Classes and Interfaces:
Classes of util package:

- ArrayDeque
- **ArrayList**
- **Arrays**
- BitSet
- Calendar
- **Collections**
- Currency
- Date
- Dictionary
- EnumMap
- EnumSet
- **Formatter**
- GregorianCalendar
- HashMap
- HashSet
- Hashtable
- IdentityHashMap
- LinkedHashMap
- LinkedHashSet
- LinkedList
- ListResourceBundle
- Locale
- Observable
- PriorityQueue
- Properties
- PropertyPermission
- PropertyResourceBundle
- **Random**
- ResourceBundle
- ResourceBundle.Control
- Scanner
- ServiceLoader
- SimpleTimeZone

- Stack
- StringTokenizer
- Timer
- TimerTask
- TimeZone
- TreeMap
- TreeSet
- UUID
- Vector
- WeakHashMap

Interfaces of util package

- Collection<E>
- Comparator<T>
- Deque<E>
- Enumeration<E>
- EventListener
- Formattable
- Iterator<E>
- List<E>
- ListIterator<E>
- Map<K,V>
- Map.Entry<K,V>
- NavigableMap<K,V>
- NavigableSet<E>
- Observer
- Queue<E>
- RandomAccess()
- Set<E>
- SortedMap<K,V>
- SortedSet<E>

Formatter Class:
The java.util.Formatter class provides support for layout justification and alignment, common formats for numeric, string, and date/time data, and locale-specific output.

Once the Formatter object is created, it may be used in many ways. The format specifier specifies the way the data is formatted.

A few common format specifiers are:

- **%S or %s:** Specifies String
- **%X or %x:** Specifies hexadecimal integer
- **%o:** Specifies Octal integer
- **%d:** Specifies Decimal integer
- **%c:** Specifies character

- **%T or %t:** Specifies Time and date
- **%n:** Inserts newline character
- **%B or %b:** Specifies Boolean
- **%A or %a:** Specifies floating point hexadecimal
- **%f:** Specifies Decimal floating point

Example:

```
Formatter f=new Formatter();
f.format("%3$3s %2$3s %1$3s", "fear","strengthen", "weakness");
System.out.println(f);
```

Random Class:

```
//all the functions of Random class will generate Pseudo random numbers
import java.util.Random;
public class Test
{
   public static void main(String[] args)
   {
      Random random = new Random();
      System.out.println(random.nextInt(10));
      System.out.println(random.nextBoolean());
      System.out.println(random.nextDouble());
      System.out.println(random.nextFloat());
      System.out.println(random.nextGaussian());
  }
}
```

Time Package

Class Instant (java. time. Instant)

Formatting for Date/Time in Java:

```
//Formatting for Date/Time in Java
import java.text.SimpleDateFormat;
import java.util.Date;
class SimpleDateFormatExample2
{
        public static void main(String[] args)
        {
                Date date = new Date();
                SimpleDateFormat formatter = new SimpleDateFormat("MM/dd/yyyy");
                String strDate = formatter.format(date);
                System.out.println("Date Format with MM/dd/yyyy : "+strDate);

                formatter = new SimpleDateFormat("dd-M-yyyy hh:mm:ss");
                strDate = formatter.format(date);
                System.out.println("Date Format with dd-M-yyyy hh:mm:ss : "+strDate);

                formatter = new SimpleDateFormat("dd MMMM yyyy");
                strDate = formatter.format(date);
                System.out.println("Date Format with dd MMMM yyyy : "+strDate);

                formatter = new SimpleDateFormat("dd MMMM yyyy zzzz");
                strDate = formatter.format(date);
```

```
                    System.out.println("Date Format with dd MMMM yyyy zzzz : "+strDate);

                    formatter = new SimpleDateFormat("E, dd MMM yyyy HH:mm:ss z");
                    strDate = formatter.format(date);
                    System.out.println("Date Format with E, dd MMM yyyy HH:mm:ss z : 
"+strDate);
}
}
```

Temporal Adjusters Class:

```
ixmport java.time.DayOfWeek;
import java.time.LocalDate;
import java.time.temporal.TemporalAdjusters;
public class TemporalAdjusterExample
{
    public static void main(String[] args)
    {

        LocalDate now = LocalDate.now();
        System.out.println("Current date : " + now);

        LocalDate output = null;

        output = now.with(TemporalAdjusters.firstDayOfMonth());
        System.out.println("firstDayOfMonth :: " + output);

        output = now.with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println("firstDayOfNextMonth :: " + output);

        output = now.with(TemporalAdjusters.firstDayOfNextYear());
        System.out.println("firstDayOfNextYear :: " + output);

        output = now.with(TemporalAdjusters.firstDayOfYear());
        System.out.println("firstDayOfYear :: " + output);

        output = now.with(TemporalAdjusters.lastDayOfYear());
        System.out.println("lastDayOfYear :: " + output);

        output = now.with(TemporalAdjusters.dayOfWeekInMonth(2, DayOfWeek.FRIDAY));
        System.out.println("dayOfWeekInMonth(1, DayOfWeek.FRIDAY) :: " + output);

        output = now.with(TemporalAdjusters.lastDayOfMonth());
        System.out.println("lastDayOfMonth :: " + output);

        output = now.with(TemporalAdjusters.lastDayOfYear());
        System.out.println("lastDayOfYear :: " + output);
    }

}
```

Exception Handling: Introduction

- **An exception is an abnormal condition that arises in a code sequence at run time. Exception is a run time error.**
- **Java and other programming languages have mechanisms for handling exceptions that you can use to keep your program from crashing. In Java, this is known as *catching an exception/exception handling*.**
- **When java interpreter encounters an error, it creates an exception object and throws it(informs us that an error occurred).**
- **If the exception object is not caught and handled properly, the interpreter will display a message and stops the program execution.**
- **If we want the program to continue with the execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions.**

Errors are broadly classified into two categories.

1. **Compile time exception(error)**
2. **Run Time exception(error)**

Compile-time errors:

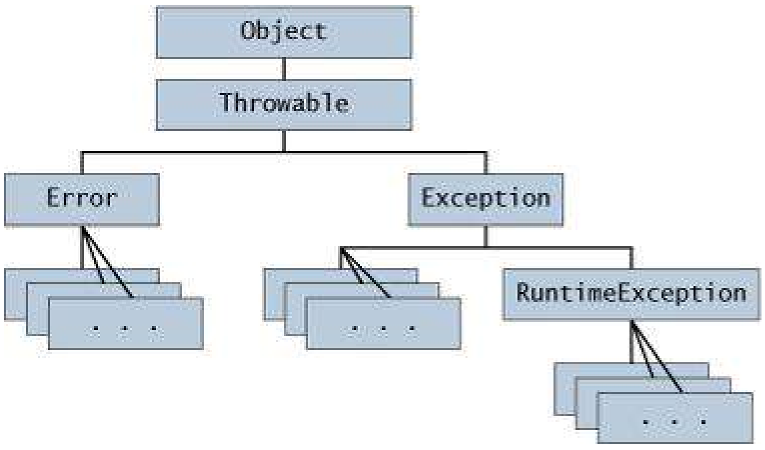
- **All syntax errors will be detected and displayed by the java compiler and therefore these errors are known as compile-time errors.**
- **Whenever the compiler displays an error, it will not create the .class file.**

Run-time errors:

- **Sometimes a program may compile successfully creating the .class file but may not run properly because of abnormal conditions.**
- **Common run-time errors are**
  - **The file you try to open may not exist.**
  - **Dividing an integer by Zero.**
  - **Accessing an element that is out of the bonds of an array type.**
  - **Trying to store a value into an array of an incompatiable class or type.**
  - **Trying to cast an instance of a class to one of its subclasses.**
  - **And many more……………..**

Each exception is a class, part of java.lang package and it is derived from Throwable class.

Hierarchy of Standard Exception Classes:



There are two types of exceptions in Java:

- Unchecked Exceptions
- Checked Exceptions

Unchecked Exceptions:

- Unchecked exceptions are **RuntimeExceptions** and any of its subclasses.
- **Error** class and its subclasses are also called unchecked exceptions.
- Compiler does not force the program to catch the exception or declare in a throws clause.
  - Ex: ArithMeticException
- Unchecked exceptions can occur anywhere in a program and in a typical program can be very numerous.

Checked Exceptions:

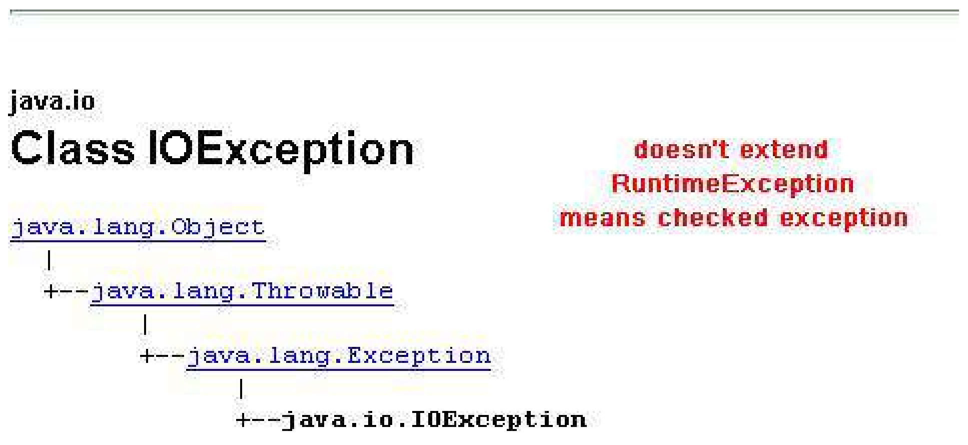
- A checked exception is any subclass of **Exception** (or **Exception class** itself), excluding class **RuntimeException** and its **subclasses**.
- Checked exceptions must be handled by the programmer to avoid a compile-time error.
- There are two ways to handle checked exceptions.
  - Declare the exception using a throws clause.
  - catch the exception.

The compiler requires a throws clause or a try-catch statement for any call to a method that may cause a checked exception to occur.

- Checked Exceptions are checked at compile time, where as unchecked exceptions are at runtime.

We can know whether the exception is Checked or Unchecked in two ways:

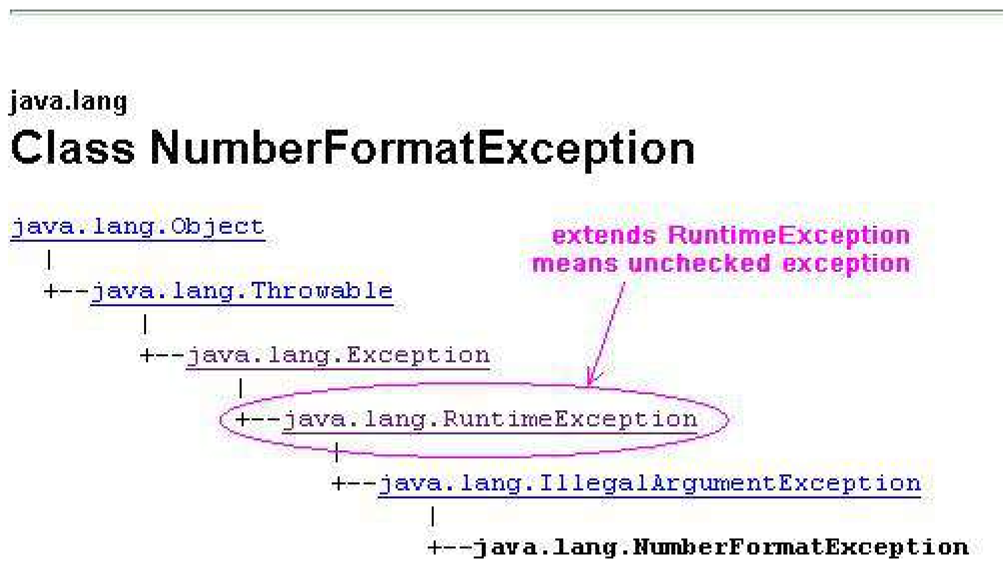
- Using Java API
- By Experience


java.io

Class IOException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.io.IOException
```

doesn't extend RuntimeException means checked exception


java.lang

Class NumberFormatException

```
java.lang.Object
  |
  +--java.lang.Throwable
        |
        +--java.lang.Exception
              |
              +--java.lang.RuntimeException
                    |
                    +--java.lang.IllegalArgumentException
                          |
                          +--java.lang.NumberFormatException
```

extends RuntimeException means unchecked exception

Some common exceptions Unchecked (Runtime Exceptions):

Exception type	Cause of exception
Arithmetic Exception	Caused by math errors such as division by zero
ArrayIndexOutOfBoundsException	Caused by bad array indxes
ArrayStoreException	Caused when a program tries to store the wrong type of data in an array
ClassCastException	Invalid cast
NullPointerException	Invalid use of a null reference

NumberFormatException	Invalid conversion of a string to a numeric format
StringIndexOutOfBounds	Attempt to index outside the bounds of a string

Some common exceptions Checked (Compiletime Exceptions):

Exception type	Cause of exception
ClassNotFound	Class not found
IllegalAccessException	Access to a class denied
InstantiationException	Attempt to create an object of an abstract class or interface
NoSuchFieldException	A requested field does not exist
NoSuchMethodException	A requested method does not exist

Java Exception handling is managed by 5 keywords:

1. try
2. catch
3. throw
4. throws
5. finally

If these abnormal conditions are not handled properly, either the program will be aborted or the incorrect result will be carried on causing more and more abnormal conditions.

Example to see what happens if the exceptions are not handled:

```
class NoException

{


      public static void main(String[] args)

      {

       int no=10;

       int r=0;

       r=no/0;

       System.out.println("Result is:"+r);

      }

}
```

The above program gives the following error:

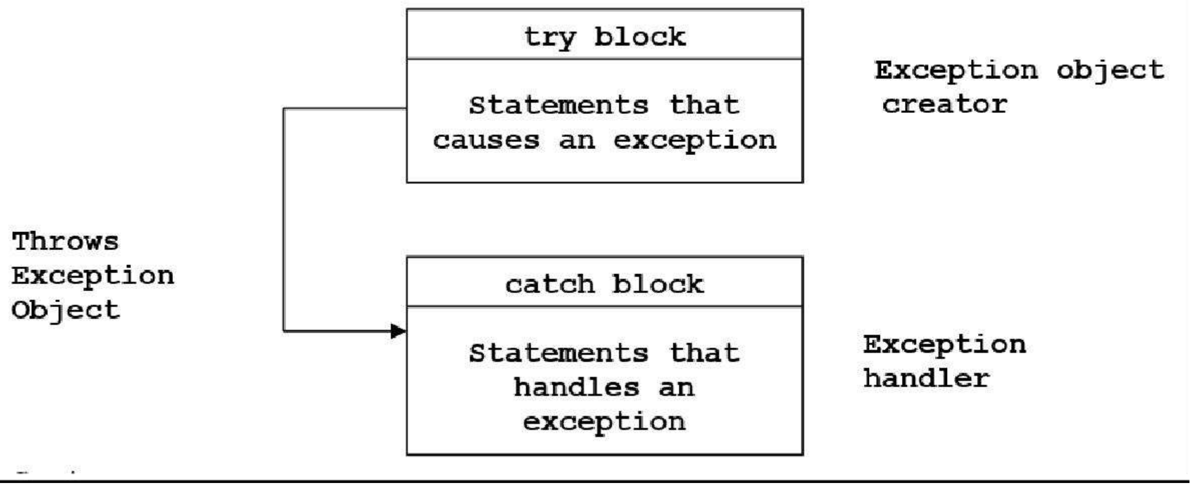
```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at NoException.main(NoException.java:8)
```

In the above program the error is handled by the default exception handler which is provided by java run-time system.

If the exception is handled by the user then it gives two benefits.

1. Fixes the error.
2. Avoids automatic termination of program.

Syntax for exception handling code:



```
try
       {
        statement;   // generates exception
       }
       catch(Exception-type e)
       {
        statement;   // process the exception
       }
```

- Java uses a keyword try to preface a block of code that is likely to cause an error and "throw" an exception.
- A catch block is defined by the keyword catch "catches" the exception "thrown" by the exception block.
- Catch block should be immediately after the try block.
- try block can have one or more statements that could generate an exception.

- If any one statement generates an exception, the remaining statements in the block are skipped and control jumps to the catch block that is placed next to the try block.
- The catch block too can have one or more statements that are necessary to process the exception.
- Every try block should be followed by at least one catch statement.
- Catch statement works like a method definition, contains one parameter, which is reference to the exception object thrown by the try block.

Program:

```
class TryCatch
{
  public static void main(String[] args)
  {
    int a=10;
    int b=5;
    int c=5;
    int r;

    try
    { r=a/(b-c); // exception here
      System.out.println("This will not be executed............");
    }
    catch(ArithmeticException e)
    { System.out.println("Cannot divide by 0....."); }
    System.out.println("After catch statement.........");
  }
}
```

Note: Once an exception is thrown, control is transferred to catch block and never returns to try block again.

***Statements after the exception statement(try block) never get executed.

Multiple Catch Clauses:

Syntax:

```
try
{
 statement;
}
catch(Exception-type 1 e)
{
 statement;  // process exception type 1
}
catch(Exception-type 2 e)
{
 statement;  // process exception type 2
}
catch(Exception-type N e)
{
 statement;  // process exception type N
}
```

- When an exception in a try block is generated, java treats the multiple catch statements like cases in switch statement.
- The first statement whose parameter matches with the exception object will be executed, and the remaining statements will be skipped.
- Code in the catch block is not compulsory.
    - catch (Exception e){}
    - the catch statement simply ends with a curly braces({}), which does nothing, this statement will catch an exception and then ignore it.

```
import java.util.*;

class MultiCatch
{
 public static void main(String args[])
 {
  int no,d,r;
  no=d=r=0;

 try
 {
  no=Integer.parseInt(args[0]);
  d=Integer.parseInt(args[1]);
  r=no/d;
  System.out.println("Result :"+r);
 }
 catch (NumberFormatException nf)
 {
   System.out.println(nf);
 }
 catch (ArrayIndexOutOfBoundsException ai)
 {
 System.out.println(ai);
 }
 catch (ArithmeticException ae)
 {
  System.out.println(ae);
 }
 }
}
```

<u>try-with-resources</u>:

In Java, the try-with-resources statement is a try statement that declares one or more resources. The resource is as an object that must be closed after finishing the program. The try-with-resources statement ensures that each resource is closed at the end of the statement execution.

<u>Syntax:</u>

```
try(resource code)
{
        use resources
}
catch()
{
  handle exceptions
}
```

<u>Example:</u>

```
import java.io.*;
class TryWithRes
{
        public static void main(String args[])
        {

                try (BufferedReader br=new BufferedReader(new FileReader("Sample.java")))
                {
                        String line;
                        while((line=br.readLine())!=null)
                        {
                                System.out.println(line);
                        }
                }
                catch(IOException e)
                {
                        System.out.println(e);
                }
        }
}
```

Custom Exceptions:

we can create our own exceptions that are derived classes of the Exception class. Creating our own Exception is known as custom exception or user-defined exception. Basically, Java custom exceptions are used to customize the exception according to user need.

Example:

```
public class WrongFileNameException extends Exception
   { public WrongFileNameException(String errorMessage)
   { super(errorMessage);
   }
}
```

Nested try and catch Blocks:

In Java, using a try block inside another try block is permitted. It is called as nested try block.

Syntax:

```
....
//main try block
try
{
   statement 1;
   statement 2;
//try catch block within another try block
   try
   {
      statement 3;
      statement 4;
//try catch block within nested try block
      try
      {
         statement 5;
         statement 6;
```

```
        }
          catch(Exception e2)
          {
    //exception message
          }

       }
       catch(Exception e1)
       {
    //exception message
       }
    }
    //catch block of parent (outer) try block
    catch(Exception e3)
    {
    //exception message
    }
    ....
```

Example:

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

                        try
                        {

                                System.out.println("going to divide");
                                int b =39/0;
                        }
                catch(ArithmeticException e)
                {
                        System.out.println("A number can't be divide with zero");
                }
```

```
                        try
                        {
                        int a[]=new int[5];
                                a[5]=4;
                        }
                catch(ArrayIndexOutOfBoundsException e)
                {
                        System.out.println(e);
                }

                }
                catch(Exception e)
                {
                        System.out.println("handeled");
                }

        }
}
```

Throws Clause or Rethrowing Exception

```
import java.io.*;
 class Test
 {
  void doWork() throws IOException
  {
   throw new IOException();
  }
 }

 class ThrowsTest1
 {
  public static void main(String args[])  throws IOException
  {
   Test t1=new Test();
   t1.doWork();
  }
  }
```

UNIT V

String Handling in Java: Introduction, Interface Char Sequence, Class String, Methods for Extracting Characters from Strings, Methods for Comparison of Strings, Methods for Modifying Strings, Methods for Searching Strings, Data Conversion and Miscellaneous Methods, Class String Buffer, Class String Builder.

Multithreaded Programming: Introduction, Need for Multiple Threads Multithreaded Programming for Multi-core Processor, Thread Class, Main Thread- Creation of New Threads, Thread States, Thread Priority-Synchronization, Deadlock and Race Situations, Inter-thread Communication - Suspending, Resuming, and Stopping of Threads.

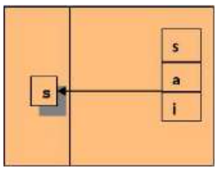
Java Database Connectivity: Introduction, JDBC Architecture, Installing MySQL and MySQL Connector/J, JDBC Environment Setup, Establishing JDBC Database Connections, ResultSet Interface, Creating JDBC Application, JDBC Batch Processing, JDBC Transaction Management.

String Handling in Java:

String: String is a group of characters surrounded by double quotes, in java String class is used to represent these group of characters.

Ex:

String s="sai"



1. Address of the object is assigned to the variable s. This object will be like a character array.
2. Once a string object is created, the data inside the object cannot be modified, that’s why we say strings are immutable.

Different ways of creating String objects:

String s1=new String();

String s2=new String("sai");

String s3="sai";

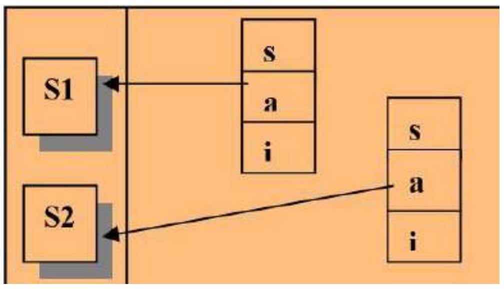
Difference between new and " "

1. Using new keyword any number of objects are created for the same class.

String s1=new String("sai");

String s2=new String("sai");

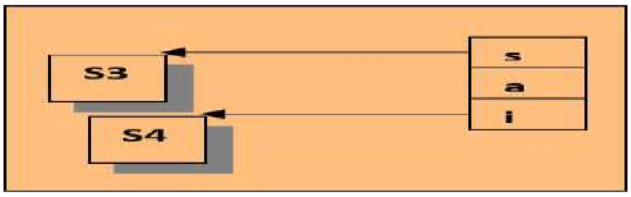
The above statements will create two different objects, with different address(same contents) assigned to different variable s1 and s2 respectively.



2. Assignment operator is used to assign the string to the variable s3. in this case, JVM first of all checks whether the same object is already available in the string constant pool or not. If it is available, then it creates another reference to it. If same object is not available, then it creates another object with the with the content "PREM" and stores it into the string constant pool.

String s3="sai";

String s4="sai";

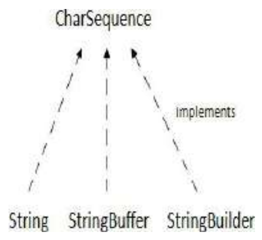


```
class StringDiff
{
 public static void main(String args[])
 {
  String s1=new String("sai");
  String s2=new String("sai");
  String s3="sai";
  String s4="sai";
  if(s1==s2)
            System.out.println("Equal..");
  else
            System.out.println("NOT Equal..");
  if(s3==s4)
            System.out.println("Equal..");
  else
            System.out.println("NOT Equal..");
 }
}
```

Note: The java. lang. String class implements *Serializable*, *Comparable* and *Char Sequence* interfaces.

Interface Char Sequence:

The CharSequence interface is used to represent the sequence of characters. String, StringBuffer and StringBuilder classes implement it. It means, we can create strings in Java by using these three classes.



The Java String is immutable which means it cannot be changed. Whenever we change any string, a new instance is created. For mutable strings, you can use StringBuffer and StringBuilder classes.

Class String:

The java. lang. String class provides many useful methods to perform operations on sequence of char values.

No.	Method	Description
1	char charAt(int index)	returns char value for the particular index
2	int length()	returns string length
3	static String format(String format, Object... args)	returns a formatted string.
4	static String format(Locale l, String format, Object... args)	returns formatted string with given locale.
5	String substring(int beginIndex)	returns substring for given begin index.
6	String substring(int beginIndex, int endIndex)	returns substring for given begin index and end index.
7	boolean contains(CharSequence s)	returns true or false after matching the sequence of char value.
8	static String join(CharSequence delimiter, CharSequence... elements)	returns a joined string.
9	boolean equals(Object another)	checks the equality of string with the given object.
10	boolean isEmpty()	checks if string is empty.
11	String concat(String str)	concatenates the specified string.

12	String replace(char old, char new)	replaces all occurrences of the specified char value.
13	String replace(CharSequence old, CharSequence new)	replaces all occurrences of the specified CharSequence.
14	static String equalsIgnoreCase(String another)	compares another string. It doesn't check case.
15	String[] split(String regex)	returns a split string matching regex.
16	int indexOf(int ch)	returns the specified char value index.
17	int indexOf(int ch, int fromIndex)	returns the specified char value index starting with given index.
18	int indexOf(String substring)	returns the specified substring index.
19	int indexOf(String substring, int fromIndex)	returns the specified substring index starting with given index.
20	String toLowerCase()	returns a string in lowercase.
21	String toLowerCase(Locale l)	returns a string in lowercase using specified locale.
22	String toUpperCase()	returns a string in uppercase.
23	String toUpperCase(Locale l)	returns a string in uppercase using specified locale.
24	String trim()	removes beginning and ending spaces of this string.
25	static String valueOf(int value)	converts given type into string. It is an overloaded method.

String concatenation:

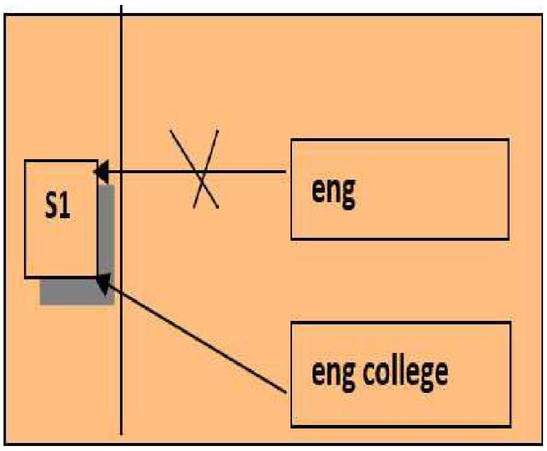
Appending a string to another String is known as String concatenation.
The + symbol acts as concatenation operator.

```
class SCat
{
 public static void main (String args[])
 {
  String s1="eng";
  s1=s1+" college";
  System.out.println(s1);
 }
}
```

We said that String objects immutable(contents of the object are not changed once created).

```
s1=s1+" Computers";
```

when this statement is executed a new string is created with the contents of LHS and RHS of the + operator. And the old object will eligible for garbage collection.



Example to work with String functions:

```
class StringClass
{
        public static void main(String args[])
        {
        String name="ADITYA";
        String name1="college";
        String name3="";
        String name4="Aditya College of Engineering and Technology";
        String name5="    Aditya College of Engineering and Technology     ";
        char ch=name.charAt(4);//returns the char value at the 4th index
        System.out.println(ch);
        System.out.println("string length is: "+name.length());
        System.out.println("sub string example: "+name.substring(2,4));
        System.out.println("contains example: "+name.contains("IT"));
        System.out.println("join example: "+String.join("-","02","07","2021"));
        System.out.println("equals example: "+name.equals(name1));
        System.out.println("concat example: "+name.concat(name1));
        System.out.println("replace example: "+name.replace('a','A'));
        System.out.println("isEmpty example: "+name3.isEmpty());
        System.out.println("isEmpty example: "+name1.isEmpty());
        System.out.println("name 4 before spliting: "+name4);
        String s[]=name4.split("\\s");
        for(String word:s)
                        System.out.println(word);
        System.out.println("indexof example: "+name.indexOf('t'));
        System.out.println("tolowercase example: "+name.toLowerCase());
        System.out.println("touppercase example: "+name1.toUpperCase());
        System.out.println("name 5 is:-"+name5);
        System.out.println("trim example: "+name5.trim());
        }
}
```

Methods for Extracting Characters from Strings

The following methods are used to extract Characters from Strings

1. charAt()
2. getChars()
3. toCharArray()
4. getBytes()

Example:

```
class Extract
{
        public static void main(String args[])
        {
        String s=new String("Aditya College of Engineering and technology");
        System.out.println("character at 7th index is-"+s.charAt(7));
        System.out.println(s.length());
        char[] des=new char[44];
        s.getChars(1,21,des,0);
        System.out.println("Extracted characters-"+des);
        String s1="ABCDEF";
        byte b[]=s1.getBytes();
        for(byte a:b)
        System.out.println(a);
        String s2="Aditya";
        char[] ch=s2.toCharArray();
        for(char c:ch)
            System.out.print(c);
        }
}
```

Methods for Comparison of Strings

The following methods are used to compare Strings

1. equals()
2. compareTo()

Example:

```
class Compare
{
        public static void main(String args[])
        {
                        String s1="Aditya";
                        String s2="Sai";
                        String s3="aditya";
                        String s4="Aditya";
                        System.out.println(s1.compareTo(s4));
                        System.out.println(s1.compareTo(s2));
                        System.out.println(s2.compareTo(s1));
                        if(s1.equals(s4))
                           System.out.println("Equals");
                        else
                          System.out.println("Not Equals");
                        if(s1==s4)
                           System.out.println("Equals");
```

```
                    else
                        System.out.println("Not Equals");
        }
}
```

Methods for Modifying Strings

The following methods are used to modify Strings

1. concat()
2. replace()
3. trim()
4. substring()

Example:

```
class StringClass
{
        public static void main(String args[])
        {
        String name="ADITYA";
        String name1="college";
        String name3="";
        String name4="Aditya College of Engineering and Technology";
        String name5="     Aditya College of Engineering and Technology      ";
        System.out.println("sub string example: "+name.substring(2,4));
        System.out.println("concat example: "+name.concat(name1));
        System.out.println("replace example: "+name.replace('a','A'));
        System.out.println("trim example: "+name5.trim());
        }
}
```

Methods for Searching Strings

The following methods are used searching Strings

1. indexOf()
2. lastIndex()
3. charAt()
4. contains()

Example:

```
class StringClass
{
        public static void main(String args[])
        {
        String name="ADITYA";
        String name1="college";
        String name3="";
        String name4="Aditya College of Engineering and Technology";
        String name5="     Aditya College of Engineering and Technology       ";
        char ch=name.charAt(4);//returns the char value at the 4th index
        System.out.println("contains example: "+name.contains("IT"));
        System.out.println("indexof example: "+name.indexOf('t'));


        }
}
```

Data Conversion and Miscellaneous Methods

Example:

```
class StringClass
{
        public static void main(String args[])
        {
        String name="ADITYA";
        String name1="college";
        String name3="";
        String name4="Aditya College of Engineering and Technology";
        String name5="     Aditya College of Engineering and Technology       ";
        char ch=name.charAt(4);//returns the char value at the 4th index
        System.out.println(ch);
        System.out.println("string length is: "+name.length());
        System.out.println("sub string example: "+name.substring(2,4));
        System.out.println("contains example: "+name.contains("IT"));
        System.out.println("join example: "+String.join("-","02","07","2021"));
        System.out.println("equals example: "+name.equals(name1));
        System.out.println("concat example: "+name.concat(name1));
        System.out.println("replace example: "+name.replace('a','A'));
        System.out.println("isEmpty example: "+name3.isEmpty());
        System.out.println("isEmpty example: "+name1.isEmpty());
        System.out.println("name 4 before spliting: "+name4);
        String s[]=name4.split("\\s");
        for(String word:s)
                        System.out.println(word);
        System.out.println("indexof example: "+name.indexOf('t'));
        System.out.println("tolowercase example: "+name.toLowerCase());
        System.out.println("touppercase example: "+name1.toUpperCase());
        System.out.println("name 5 is:-"+name5);
        System.out.println("trim example: "+name5.trim());
        }
}
```

StringBuffer class:

1. String buffer is a peer class of string. The string buffer class creates mutable strings of flexible length which can be modified in terms of both length and content.
2. StringBuffer reserves space for 16 characters.
3. EnsureCapacity () adds double+2 spaces.
4. The following are methods of StringBuffer Class
    1. length()
    2. capacity()
    3. charAt()
    4. append()
    5. toString()
    6. reverse()
    7. ensureCapacity()
    8. setCharAt()

9. delete();
10. deleteCharAt();
11. replace();
12. substring();
13. insert()

Example 1:

```
class StringBufferDemo
{
 public static void main(String args[])
 {
  StringBuffer sb=new StringBuffer("Engineering Student");
  System.out.println("Buffer:"+sb);
  System.out.println("Length:"+sb.length());
  System.out.println("Capacity:"+sb.capacity());
  // Displaying all the characters of the string
  for(int i=0;i<s.length();i++)
  {
        System.out.println("Character at position:" +i+" is "+s.charAt(i));
  }
   // Appending at the end
   s.append(" CSE");
   System.out.println("New string is:"+s);
 }
}
```

Example 2:

```
class StringBufferDemo1
{
public static void main(String args[])
{
 StringBuffer sb1=new StringBuffer();
 StringBuffer sb2=new StringBuffer(10);
 StringBuffer sb3=new StringBuffer("Prem");

 System.out.println("sb3 toString:"+sb3.toString());
 System.out.println("sb1 length is:"+sb1.length());
 System.out.println("sb2 length is:"+sb2.length());
 System.out.println("sb3 length is:"+sb3.length());
 System.out.println("sb1 capacity is:"+sb1.capacity());
 System.out.println("sb2 capacity is:"+sb2.capacity());
 System.out.println("sb3 capacity is:"+sb3.capacity());
 sb1.ensureCapacity(50);
 System.out.println("sb1 capacity is:"+sb1.capacity());
 StringBuffer sb4=new StringBuffer("hi are u");
 System.out.println("sb4 character is:"+sb4.charAt(0));
 System.out.println("sb4 character is:"+sb4.charAt(4));
 sb4.setCharAt(0,'H');
 sb4.setCharAt(4,'H');
 sb4.reverse();
 System.out.println(sb4);

 sb4.insert(4,"  Prem");
 String s="Good Bye";
 boolean b=true;
 int i=99;
 double d=99.99;

 StringBuffer sb5=new StringBuffer();

 sb5.append('r');
 sb5.append('a');
 sb5.append('j');
```

```
 System.out.println(sb5);
 StringBuffer sb6=new StringBuffer("Raj Jain");
 sb6.insert(4," Kumar ");
 System.out.println(sb6);
 sb6.delete(1,3);
 sb6.deleteCharAt(3);
 sb6.replace(0,4,"Hello");
 String s1=sb6.substring(4);
 String s2=sb6.substring(0,4);
 System.out.println(s1);
 System.out.println(s2);
 }
};
```

equals() and ==:

equals() and == performs two different operations. equals() method compares the characters inside a String object.

The == operator compares two object references to see whether they refer to the same instance.

StringBuilder class:

StringBuilder class has been added in JDK1.5 which has same features like StringBuffer class. StringBuilder class objects are also mutable as the stringBuffer Objects.

Difference:

StringBuffer is class is synchronized and StringBuilder is not.

Multithreaded Programming:

Introduction, Need for Multiple Threads Multithreaded Programming for Multi-core Processor

1. Multithreading in Java is a process of executing multiple threads simultaneously.
2. A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Advantages of Java Multithreading

1. It **doesn't block the user** because threads are independent and you can perform multiple operations at the same time.
2. You **can perform many operations together, so it saves time**.

3. Threads are **independent**, so it doesn't affect other threads if an exception occurs in a single thread.

Multitasking

Multitasking is a process of executing multiple tasks simultaneously. We use multitasking to utilize the CPU. Multitasking can be achieved in two ways:

- Process-based Multitasking (Multiprocessing)
- Thread-based Multitasking (Multithreading)

1) Process-based Multitasking (Multiprocessing)

- Each process has an address in memory. In other words, each process allocates a separate memory area.
- A process is heavyweight.
- Cost of communication between the process is high.
- Switching from one process to another requires some time for saving and loading registers, memory maps, updating lists, etc.

2) Thread-based Multitasking (Multithreading)

- Threads share the same address space.
- A thread is lightweight.
- Cost of communication between the thread is low.

Thread Class

Thread class provide constructors and methods to create and perform operations on a thread.Thread class extends Object class and implements Runnable interface.

Commonly used Constructors of Thread class:

- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r,String name)

Commonly used methods of Thread class:

- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void sleep(long miliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** waits for a thread to die.
- **public void join(long miliseconds):** waits for a thread to die for the specified miliseconds.
- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.

- **public int getId():** returns the id of the thread.
- **public Thread.State getState():** returns the state of the thread.
- **public boolean isAlive():** tests if the thread is alive.
- **public void yield():** causes the currently executing thread object to temporarily pause and allow other threads to execute.
- **public void suspend():** is used to suspend the thread(depricated).
- **public void resume():** is used to resume the suspended thread(depricated).
- **public void stop():** is used to stop the thread(depricated).
- **public boolean isDaemon():** tests if the thread is a daemon thread.
- **public void setDaemon(boolean b):** marks the thread as daemon or user thread.
- **public void interrupt():** interrupts the thread.
- **public boolean isInterrupted():** tests if the thread has been interrupted.
- **public static boolean interrupted():** tests if the current thread has been interrupted.

Main Thread:

- A Thread represents a separate path of execution.
- Group of statements executed by JVM one by one.

Program to find the thread used by JVM to execute the statements(Main Thread):

```
class ThreadName
{
   public static void main(String args[])
   {
      System.out.println("Welcome ");
      Thread t=Thread.currentThread();
      System.out.println("Current Thread is : "+t);
      System.out.println("Current Thread Name is : "+t.getName());
   }
}
```

In the above program currentThread() is a static method of Thread Class, which returns the reference of the current running thread.

```
Thread[main,5,main]
```

- Here Thread indicates that t is thread class object.
- The First main indicates the name of the thread that is executing the current code.
- 5 the is priority of thread.

The next main indicates the thread group name to which this thread belongs.

Creation of New Threads

Creating threads in java is simple. Threads are created in the form of objects that contain a method called run(). The run() is heart and soul of any thread. The whole code that constitutes a new thread is written inside the run() method.

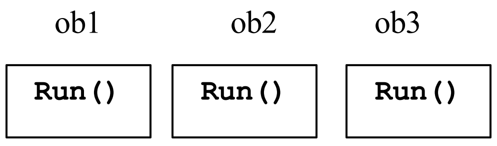
Threads can be created by the user in two ways

1. By extending Thread class
2. By implementing Runnable interface

Creating thread by extending Thread class

Syntax:-

```
class <class name> extends Thread
{
 public void run()
 {
  -----------
  -----------override run() method
  -----------
 }
}
```



Note:

- Methods of Thread class are inherited in our own class, so every object created for our class is a thread.
- Every object(thread) has its own start(),run() and many more methods.
- Whenever start() method is called run() is excuted.
- The run() method is just like your main() method where you can declare variables, use other classes, and can call other methods.
- The run() is an entry point for another thread. This thread will end when run() returns.

Example:

```
class NewThread1 extends Thread
{
 public void run()
 {
  try
  {
  for(int i=1;i<=5;i++)
  { System.out.println(Thread.currentThread().getName()+" : "+i);
```

```
   Thread.sleep(1000);
  }
  }
  catch(InterruptedException e){}
  System.out.println("Child thread exiting... ");
};
}
class NewThreadDemo1
{
 public static void main(String args[])throws InterruptedException
 {
  NewThread1 t1=new NewThread1();
  t1.setName("Child Thread");
  t1.start();
}
};
```

<u>By implementing Runnable interface:</u>

- We can’t extend more than one class to our class because multiple inheritance is not possible, in such situations we can implement runnable interface.
- Runnable implementation is slightly less simple. To run a separate thread, you still need a thread instance.
- To implement Runnable interface, a class need to implement a single method called run().

Syntax: Public void run()

<u>Example:</u>

```
class NewThread3 implements Runnable
{
 public void run()
 {
  try
  {
  for(int i=1;i<=5;i++)
  {
   System.out.println(Thread.currentThread().getName()+" : "+i);
   Thread.sleep(1000);
  }
  }
  catch(InterruptedException e){}
  System.out.println("Child thread exiting... ");
};
}
class NewThreadDemo3
{
 public static void main(String args[])throws InterruptedException
 {
  NewThread3 obj=new NewThread3();
  Thread t1=new Thread(obj);
  t1.setName("Child Thread"); t1.start();
}
};
```

Thread Priorities:

When the threads are created and started, a 'thread scheduler' program in JVM will load them into memory and execute them. This scheduler will allot more JVM time to those thread which are having priority.

The priority numbers will change from 1 to 10.
Thread.MAX_PRIORITY – 10
Thread.MIN_PRIORITY – 1
Thread.NORM_PRIORITY - 5

Example:

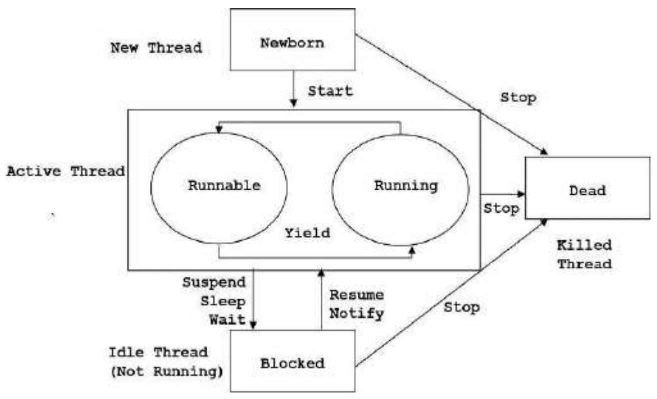
```
class MyClass extends Thread
{
 int count=0;
 public void run()
 {
  for(int i=1;i<=10000;i++)
   count++;

  System.out.println("Completed Thread:"+Thread.currentThread().getName());
  System.out.println("It's Priority:"+Thread.currentThread().getPriority());
 }
}
class Prior
{
 public static void main(String args[])
 {
  MyClass m=new MyClass();
  Thread t1=new Thread(m,"One");
  Thread t2=new Thread(m,"Two");
  t1.setPriority(2);
  t2.setPriority(Thread.NORM_PRIORITY);

  t1.start();
  t2.start();
 }
}
```

<u>Thread States (OR) Life Cycle of the Thread:</u>

During the life time of a thread, there aremany states it can enter.They are

1. Newborn State
2. Runnable State
3. Running State
4. Blocked State
5. Dead Sate

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

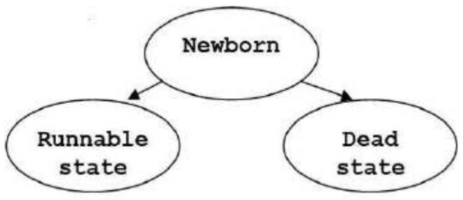


<u>Newborn state:</u>

When a thread object is created it is said to be in newborn state. It is not yet scheduled for running. At this stage we can do only following things.

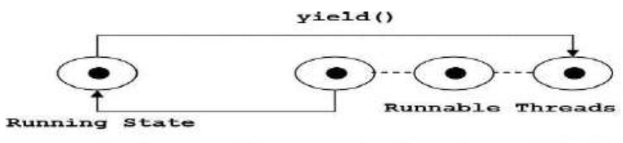
- Schedule it for running using start() method.
- Kill it using Stop() method.

If scheduled, it moves to the runnable state. If any other method is used an exception will be thrown.



<u>Runnable state:</u>

- Runnable state means the thread is ready for execution and is waiting for the availability of the processor. i.e The thread has joined the queue of threads that are waiting for execution.
- If all threads have equal priority, they are given time slots for execution in round robin fashion. i.e first come first serve manner.
- The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is knows as time slicing.



(Relinquishing control using yield() method)

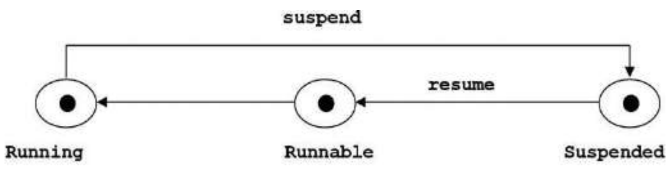
If we want a thread to relinquish control to another thread of equal priority before its turn comes, we can do so by using the yield().

Running State:

- Running means the processor has given its time to the thread for execution.
- A running thread may relinquish its control in one of the following situtations.

Suspend:

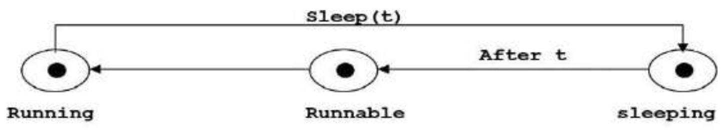
It has been suspended using suspend() method. A suspended thread can be revived using resume() method. This is useful when we want to suspend a thread for some time due to certain reason. But do not want to kill.



(Relinquishing control using suspend() method)

Sleep:

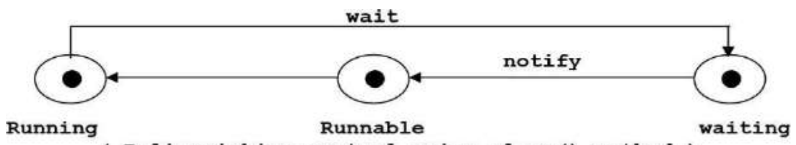
It has been made to sleep. We put a thread to sleep for a specified time period using the method sleep(time) where time is in milliseconds. This means that the thread is out of the queue during this time period. It re-enters the runnable state as soon as time period is elapsed.



(Relinquishing control using sleep() method)

Wait:

It had been told to wait until some event occurs. This is done using wait() method. The thread can be scheduled to run again using notify() method.



(Relinquishing control using sleep() method)

Blocked State:

- A thread is said to be blocked when it is prevented from entering into the Runnable state. Subsequently the running state.

- This happens when the thread is suspended, sleeping or waiting in order to satisfy certain requirements.
- A blocked thread is considered "NOT RUNNABLE" but not dead and therefore fully qualified to run again.

Dead state:

- Every thread has a life cycle.
- A running thread ends its life when it has completed executing its run() method.
- It is a natural death.
- We can kill a thread by sending stop message to it at any state thus causing a premature death.
- A thread can be killed as soon as it is born, or while it is running or even when it is in "NOT RUNNABLE"(blocked) condition.

Daemon Threads:

- Threads that work continuously without any interruption to provide services to other threads (works in the background to support the runtime environment) are called *daemon threads*.
  Eg: the clock handler thread, the idle thread, the garbage collector thread, the screen updater thread etc.
- By default a thread is not a daemon thread.
  - *setDaemon(true)* turns a thread into deamon thread.
  - *isDaemon()* to know whether a thread is Daemon or not.

Synchronization

- When a thread is already acting on an object, preventing any other thread from acting on the same object is called "Thread synchronization" or "Thread safe".
- Synchronized object is like a locked object, When a thread enters the object, it locks it, so that the next thread cannot enter till it comes out.
- This means the object is locked mutually on threads, so this object is called mutex(Mutually exclusive lock).

How to synchronize the object:

There are two ways

- Synchronized block
- Synchronized keyword

Synchronized block:

Here we can embed a group of statements of the object(inside run() method) within a synchronized block.

```
synchronized(this)

{

 Statements;

}
```

Synchronized keyword:

We can synchronize an entire method my using synchronized keyword.

```
synchronized void display()

{

 Statements;

}
```

Example for Synchronized block can be executed only by one thread:

```
class Reserve implements Runnable
{
  int avaliable=1;
  int wanted;
  Reserve(int i)
   {
           wanted=i;
    }
  public void run()
   {
            synchronized(this)
             {
              System.out.println("Avaliable Berths= " +avaliable);
              if(avaliable>=wanted)
               {
                String name=Thread.currentThread().getName();
                System.out.println(wanted+"  berths reserved for "+name);
                try
                {
                  Thread.sleep(1000);
                  avaliable=avaliable-wanted;
                 }
                catch(InterruptedException ie)
                { ie.printStackTrace(); }
                  }
                  else
                    System.out.println("Sorry no berths....");
           }
   }
}
class Sync
{
          public static void main(String args[])
          {
            Reserve obj=new Reserve(1);
            Thread t1=new Thread(obj);
            Thread t2=new Thread(obj);

            t1.setName("First person...");
            t2.setName("Second Person. .. ");
            t1.start();
            t2.start();
          }
};
```

Deadlock and Race Situations

- Deadlock in Java is a condition where two or more threads are blocked forever, waiting for each other.
- This usually happens when multiple threads need the same locks but obtain them in different orders.
- Multithreaded Programming in Java suffers from the deadlock situation because of the synchronized keyword.

Example:

```
public class Example
{
    public static void main(String[] args)
    {
       final String r1 = "surya";
       final String r2 = "java";
       Thread t1 = new Thread()
       {
            public void run()
            {
               synchronized(r1)
               {
                    System.out.println("Thread 1: Locked r1");
                    try
                    {
                         Thread.sleep(100);
                    }
                    catch(exception e)
                    {
                    }
               }
            }
};
Thread t2 = new Thread()
  {
     public void run()
      {
     synchronized(r1)
      {
      System.out.println("Thread 2: Locked r1");
      try{ Thread.sleep(100);} catch(exception e) {}
      }
    }
};
t1.start();
t2.start();
}
}
```

Inter-thread Communication:

In some cases, two or more threads should communicate with each other. For example, a consumer thread is waiting for a producer to produce the data. When the producer completes the production of data, then the consumer thread should take that data and use it.

Methods used for inter -thread communication:

obj.wait() : This method makes a thread wait for the object(obj) till it receives a notification from a notify() or notifyAll() method.

obj.notify() : This method releases an object(obj) and sends a notification to a waiting thread that the object is available.

obj.notifyAll() : This method is useful to send notification to all waiting threads at once that the object is available.

Example:

```
class Communicate1

{

 public static void main(String args[])throws Exception

 {

  Producer obj1=new Producer();

  // pass producer object to Consumer

  Consumer obj2=new Consumer(obj1);

  //Create two threads and attach to producer and consumer

  Thread t1=new Thread(obj1);

  Thread t2=new Thread(obj2);

  t2.start();//consumer waits

  t1.start();//producer starts production

 }

}
```

```
class Producer extends Thread
{
 StringBuffer sb;
 Producer()
 {
  sb=new StringBuffer();
 }
 public void run()
 {
  synchronized(sb)
  {
  for(int i=1;i<=10;i++)
  {
            try{
                        sb.append(i+":");
                        Thread.sleep(100);
                        System.out.println("Appending...");
               }
              catch(Exception e){}
   }
   //Production is over, so it is sending a notification to consumer
   //thread that sb object is avaliable
   sb.notify();
  }
 }//end of run
}//end of producer class
class Consumer extends Thread
{
 Producer prod;
 Consumer(Producer p)
{
 prod=p;
}
public void run()
{
 synchronized(prod.sb)
  {
   // wait till a notification is received from producer thread. Here
   //there is no wastage of time of even a single millisecond
   try{
            prod.sb.wait();
      }
      catch(Exception e){}
      System.out.println(prod.sb);
  }//end of sync
 }//end of run
}//end of consumer class
```

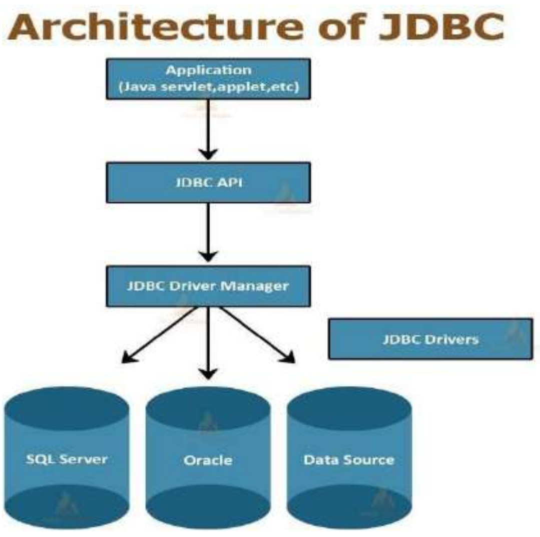
Java Database Connectivity:

Introduction

1. The term JDBC stands for Java Database Connectivity and it is a specification from Sun microsystems.
2. JDBC is an API (Application programming interface) in Java that helps users to interact or communicate with various databases.
3. The classes and interfaces of JDBC API allow the application to send the request to the specified database.
4. Using JDBC, we can write **programs required to access databases**. JDBC and the database driver are capable of accessing databases and spreadsheets.
5. Interacting with the database requires efficient database connectivity, which we can achieve using ODBC (Open database connectivity) driver. We can use this ODBC Driver with the JDBC to interact or communicate with various kinds of databases, like Oracle, MS Access, MySQL, and SQL, etc.

6. The **java.sql** package contains classes and interfaces for JDBC API. A list of popular *interfaces* of JDBC API are given below:
    - Driver interface
    - Connection interface
    - Statement interface
    - PreparedStatement interface
    - CallableStatement interface
    - ResultSet interface
    - ResultSetMetaData interface
    - DatabaseMetaData interface
    - RowSet interface

    A list of popular classes of JDBC API are given below:
    - DriverManager class
    - Blob class
    - Clob class
    - Types class
7. We can use JDBC API to handle database using Java program and can perform the following activities:
    1. Connect to the database
    2. Execute queries and update statements to the database
    3. Retrieve the result received from the database.



Java Database Connectivity with MySQL

To connect Java application with the MySQL database, we need to follow 5 following steps. In this example we are using MySql as the database. So we need to know following informations for the mysql database:

1. **Driver class:** The driver class for the mysql database is **com.mysql.jdbc.Driver**.
2. **Connection URL:** The connection URL for the mysql database is **jdbc:mysql://localhost:3306/aditya** where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and aditya is the database name. We may use any database, in such case, we need to replace the aditya with our database name.
3. **Username:** The default username for the mysql database is **root**.
4. **Password:** It is the password given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

Example:

Id	Name	City
2415	Vahida	KKD
117	Ail	RJY

Example:

```
import java.sql.*;
class MysqlCon
{
        public static void main(String args[])
        {
                try
                {
                        Class.forName("com.mysql.jdbc.Driver");
                        Connection con=DriverManager.getConnection( "jdbc:mysql://localho
                st:3306/aditya","root","root");
                        //here sai is database name, root is username and password
                        Statement stmt=con.createStatement();
                        ResultSet rs=stmt.executeQuery("select * from emp");
                        while(rs.next())
                        System.out.println(rs.getInt(1)+" "+rs.getString(2)+" "+rs.get
                    String(3));
                    con.close();
                }
                catch(Exception e)
                {
                    System.out.println(e);
            }
}
}
```

JDBC Batch Processing:

- Batch Processing allows you to group related SQL statements into a batch and submit them with one call to the database.
- When you send several SQL statements to the database at once, you reduce the amount of communication overhead, thereby improving performance.
- JDBC drivers are not required to support this feature. You should use the DatabaseMetaData.supportsBatchUpdates() method to determine if the target database supports batch update processing. The method returns true if your JDBC driver supports this feature.
- The **addBatch()** method of Statement, PreparedStatement, and CallableStatement is used to add individual statements to the batch. The **executeBatch()** is used to start the execution of all the statements grouped together.
- The **executeBatch()** returns an array of integers, and each element of the array represents the update count for the respective update statement.
- Just as you can add statements to a batch for processing, you can remove them with the **clearBatch()** method. This method removes all the statements you added with the addBatch() method. However, you cannot selectively choose which statement to remove.

Step to Batching with Statement Object

Here is a typical sequence of steps to use Batch Processing with Statement Object −

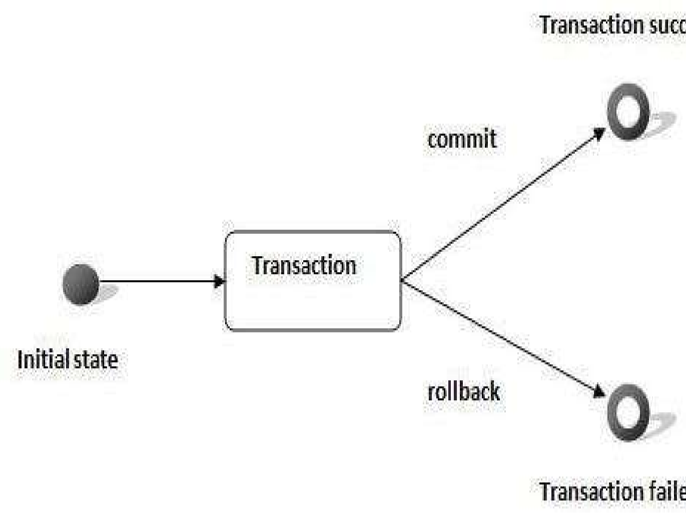
- Create a Statement object using either createStatement() methods.
- Set auto-commit to false using setAutoCommit().
- Add as many as SQL statements you like into batch using addBatch() method on created statement object.
- Execute all the SQL statements using executeBatch() method on created statement object.
- Finally, commit all the changes using commit() method.

Example:

```
// Create statement object
Statement stmt = conn.createStatement();
// Set auto-commit to false
conn.setAutoCommit(false);
// Create SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " + "VALUES(200,'ravi', 'Ali', 30)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
// Create one more SQL statement
String SQL = "INSERT INTO Employees (id, first, last, age) " +  "VALUES(201,'Raj', 'Kumar', 35)";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
// Create one more SQL statement
String SQL = "UPDATE Employees SET age = 35 " +"WHERE id = 100";
// Add above SQL statement in the batch.
stmt.addBatch(SQL);
// Create an int[] to hold returned values
int[] count = stmt.executeBatch();
//Explicitly commit statements to apply changes
conn.commit();
```

Transaction Management in JDBC

- Transaction represents **a single unit of work**.
- The ACID properties describes the transaction management well. ACID stands for Atomicity, Consistency, isolation and durability.



In JDBC, **Connection interface** provides methods to manage transaction.

Method	**Description**
void setAutoCommit(boolean status)	It is true bydefault means each transaction is committed bydefault.
void commit()	commits the transaction.
void rollback()	cancels the transaction.

Example:

```
import java.sql.*;
class FetchRecords
{
  public static void main(String args[])throws Exception
  {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        Connection con=DriverManager.getConnection("jdbc:oracle:thin:@localhost:
1521:xe","system","oracle");
        con.setAutoCommit(false);
        Statement stmt=con.createStatement();
        stmt.executeUpdate("insert into user420 values(190,'abhi',40000)");
        stmt.executeUpdate("insert into user420 values(191,'umesh',50000)");
        con.commit();
con.close();
   }}
```