Object Oriented Programming Using Java

UNIT-I

Java was conceived by James Gosling, Patrick Naughton and their team members at "Sun Micro Systems" in 1991.

⇒ Initially Java was called "Oak".

⇒ The primary motivation behind the invention of Java programing language was to develop a platform independent that could be used to create software to be embedded in various electronic devices.

⇒ The trouble with C/C++ and other languages are that they are designed to be compiled for a specific computer machine.

By overcoming this problem, James Gosling and others concentrated towards the development of platform independent, architectural neutral, portable & secure programing language based on the "Virtual Machine" architectures which is invented by IBM in 1972.

James Gosling's solution "Java" not only providing solution for platform independent language but also creates biggest effect in the era of "Internet"

Scanned with CamScanner

Java programming Language	
Paradigm	Multi paradigm (generic, Object oriented, Imperative)
Designed by	James Gosling
Developer	Sun Microsystems (Initially) Oracle Corporation (Now)
First appeared	May 23, 1995.
stable release	Java SE 16 March 16, 2021
File name extension	.java, .class, .jar
Typing discipline	strong, safe

Scanned with CamScanner

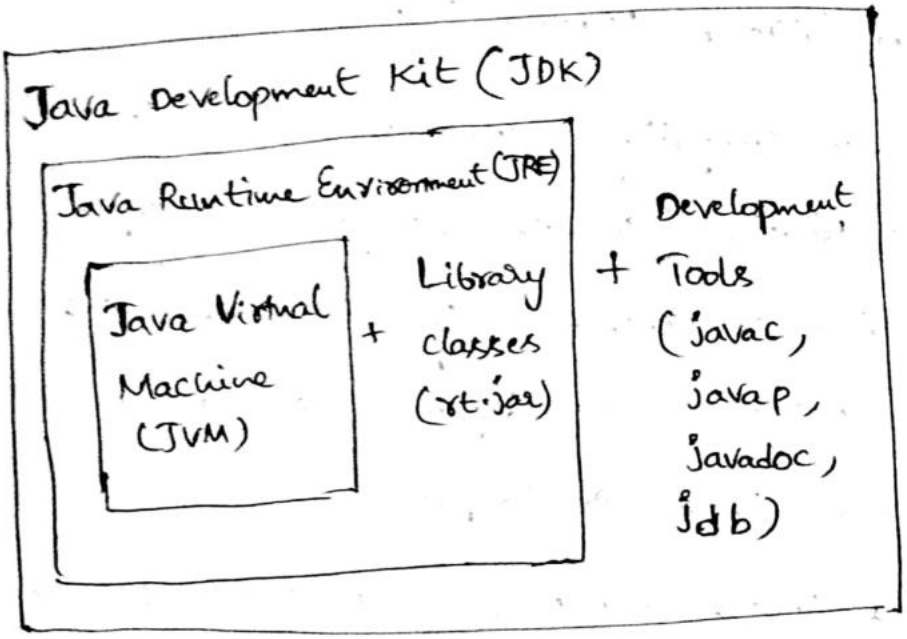
Five primary goals in the creation of Java language

1. It must be simple, Object oriented and familiar
2. It must be robust and secure.
3. It must be architecture - neutral and portable.
4. It must execute with high performance.
5. It must be Interpreted, Multithreaded and dynamic.

Characteristics of Java programming

1. Object orientation
2. platform Independence.
3. Automatic Garbage collection.
4. Java Runtime Environment.

JDK, JRE, JVM and Byte code :



JDK :- JDK (Java Development Kit) is the implementation of Java platform. It may be (i) Java SE
(ii) Java EE
(iii) Java ME

Scanned with CamScanner

(i) Java SE is a computing platform for development and deployment of portable code for desktop and server environments.

(ii) Java EE is a computing platform for development and deployment of web based Enterprise applications.

(iii) Java ME is a computing platform for development and deployment of portable code for embedded and mobile devices.

JRE (Java Runtime Environment) :-

JRE is the implementation of JVM and is defined as a software package that provides Java class libraries along with JVM and other components to run applications written in Java programming.

JVM (Java Virtual Machine) :

In computing Environments, a virtual machine is an emulation (imitate & reproduce) of a computer system which is invented by IBM in 1972.

A Virtual Machine is defined as a computer file which behaves like a computer.

JVM is a Virtual Machine architecture provides seperate computing platform for running java applications on underlying operating system Environments.

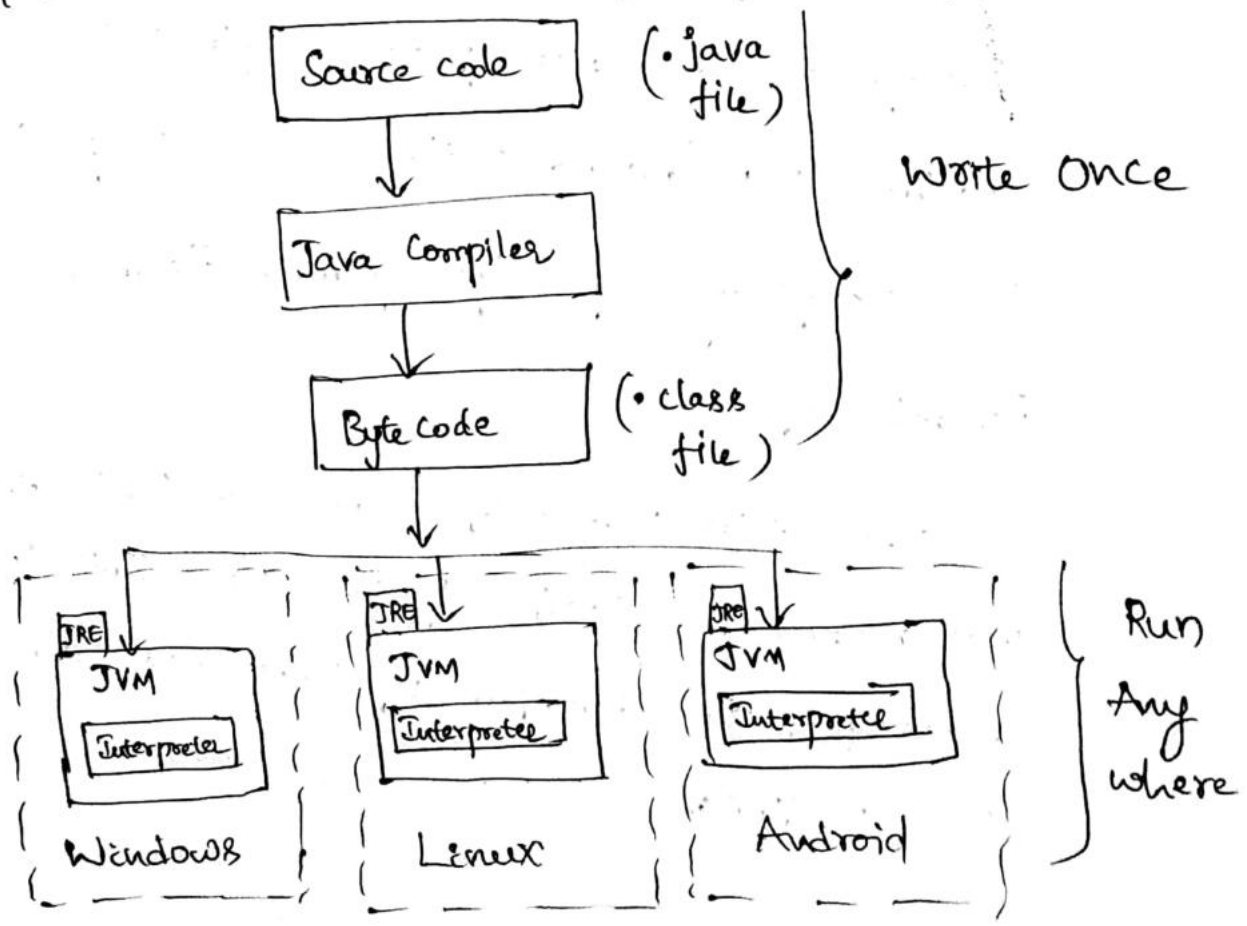
Scanned with CamScanner

Java's Magic : Byte code (.class file)

Byte code is an Intermediate code that has been compiled from source code into low level code designed for software interpreted.

Byte code is a highly optimized set of instructions designed to be executed by JVM's interpreter.

→ Translating a Java file in to byte code makes its easier to run a program in a wide variety of environments, because only the JVM needs to be implemented for each platform.



Scanned with CamScanner

Although the details of JVM will differ from platform to platform all understands the the same byte code.

In brief,

Java program is a platform Independent code that is executed on any operating system and Hardware Environments.

→ when Java compiler translates source code in to byte code which is a portable code to be executable for any platform.

→ JRE accepts byte code and JRE's JVM's Interpreter translates portable byte code in to Machine code (Machine native code) respective to corresponding operating system & Hardware Environments

Evolution of Java

→ The initial version of Java was JDK 1.0 which was released in 1996. By the time JDK 1.0 released, the developers of Java had already created JDK 1.1

Scanned with CamScanner

JDK Versions	Features
JDK 1.0 (1996 Jan 23rd)	Initial version From Sun Micro systems
JDK 1.1 (1997)	Added many new library elements Inner classes, JDBC, AWT, JIT compiler
JDK 1.2 (or) J2SE1.2 (1998)	Unicode support Swing API, collection Framework.
J2SE 1.3 (2000)	RMI, Java Sound
J2SE 1.4 (2002)	IPV6 support, Exception Chaining, working with XML
J2SE 5 (or) Java SE 5.0 (2004)	Generics, Autoboxing & Auto unboxing Enumerations, static import, Formatted I/O (Java programing reconstructed by Java SE 5.0)
Java SE 6 (2006)	JDBC 4.0 scripting languages
Java SE 7 (2011)	Enhanced Exception handling String control a switch statement

From JDK 1.0 to Java SE 6, these versions released from Sun Microsystems. From Java SE7 versions released from Oracle.

Scanned with CamScanner

Version	Features
Java SE 8 (2014)	Lambda expressions, Method references, Default methods, JavaFX.
Java SE 9	Modularity & JShell.
Java SE 10 (2018)	Local variable Type inference performance improvements.
Java SE 11 (2018)	New features are added to String handling.
Java SE 12 (2019)	switch expressions, JVM constants API.
Java SE 13 (2019).	Enhancements to switch expressions Text blocks with """ """.
Java SE 14 (2020)	Packaging Tool (Incubator), JFR (Java Flight recorder) event streaming
Java SE 15 (2020)	Hidden classes, Z Garbage collector.
Java SE 16 (2021)	primitive classes, ~~sealed~~ sealed classes, Vector API
Java SE 17 (2021)	~~sealed~~ sealed classes, Foreign Memory Access API

Scanned with CamScanner

Java Buzzwords:-

The importance and capabilities of java programming language can understand through Java Buzz words.

(i) Simple.
(ii) Secure
(iii) Portable
(iv) Object oriented
(v) Robust
(vi) Multithreaded
(vii) Architecture - Neutral
(viii) Interpreted
(ix) High Performance
(x) Distributed
(xi) Dynamic.

(i) Simple:-

Java was designed to write, learn programs eazily and efficiently. Java inherits syntax from C/C++ programing language and adoption of object oriented features.

(ii) Secure:-

Java programs are prevented from unauthorized users accessing the system resources.

Each Java application can have its own security manager object that acts a "full time security guard"

The JVM performs verification on executable code before running it to prevent the program from performing unsafe operations.

Scanned with CamScanner

(iii) portable :-

portability is a major aspect of the internet; because there are many different types of computers and operating systems connected to it.

The Java program successfully executed in the internet from variety of CPU's, operating systems, and browsers.

The "same code" must work on all computers.

(iv) Object Oriented :-

Java programming language built on the philosophy of "object oriented programming".

The object oriented model in Java is very simple and promises "Every thing is an Object"

(v) Robust :-

Java robust because of 2 reasons.

(a) Efficient Memory Management

– Allocation of memory done by programmer, and programmer need not to think about deallocation of memory – "JVM will take care itself."

(b) Systematic Exception handling mechanism.

– Java provides Object oriented exception handling mechanism.

Scanned with CamScanner

(Vi) Multithreaded :-

Java supports multithreaded programming, which allows us to write programs that do "diffent things simultaneously." One process may create several threads, all threads executed "parallelly", "concurrently"

(Vii) Architectural Neutral :-

Java program can successfully run on any diffent kinds of hardware machines with out writing "machine specific code"

Java is a

Write once ; Run anywhere, anytime, forever

programming language.

(Viii) Interpreted :-

Java is a Interpreted programing language. Java enables the creation of cross-platform programs by compiling into an intermediate representation called "Byte code"

(ix) High Performance :-

Java is faster than other traditional interpreted languages because java byte code is close to native code.

Scanned with CamScanner

The Native code is a computer programing code that is compiled to run with a particular machine. The JVM converts the "platform generalized" byte code into native code that will run in a specific processor.

(X) Distributed:-

Java is designed for the distributed computing (collaborative, shared computation on multiple computers) environment to work across Internet.

(Xi) Dynamic:-

Java is considered as "Dynamic" because of "Byte code". JVM loads java classes at run time.

Scanned with CamScanner

```
//Helloworld - First Java program.
//HelloWorld.java
import java.lang.*;
class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Welcome to Java World");
    }
}
```

Compilization :- D:\>2cse> javac HelloWorld.java

Run : D:\>2cse> java HelloWorld

Welcome to Java World.

The successful compilization of HelloWorld.java using "javac" command generates "HelloWorld.class" file in the directory called as "Byte Code".

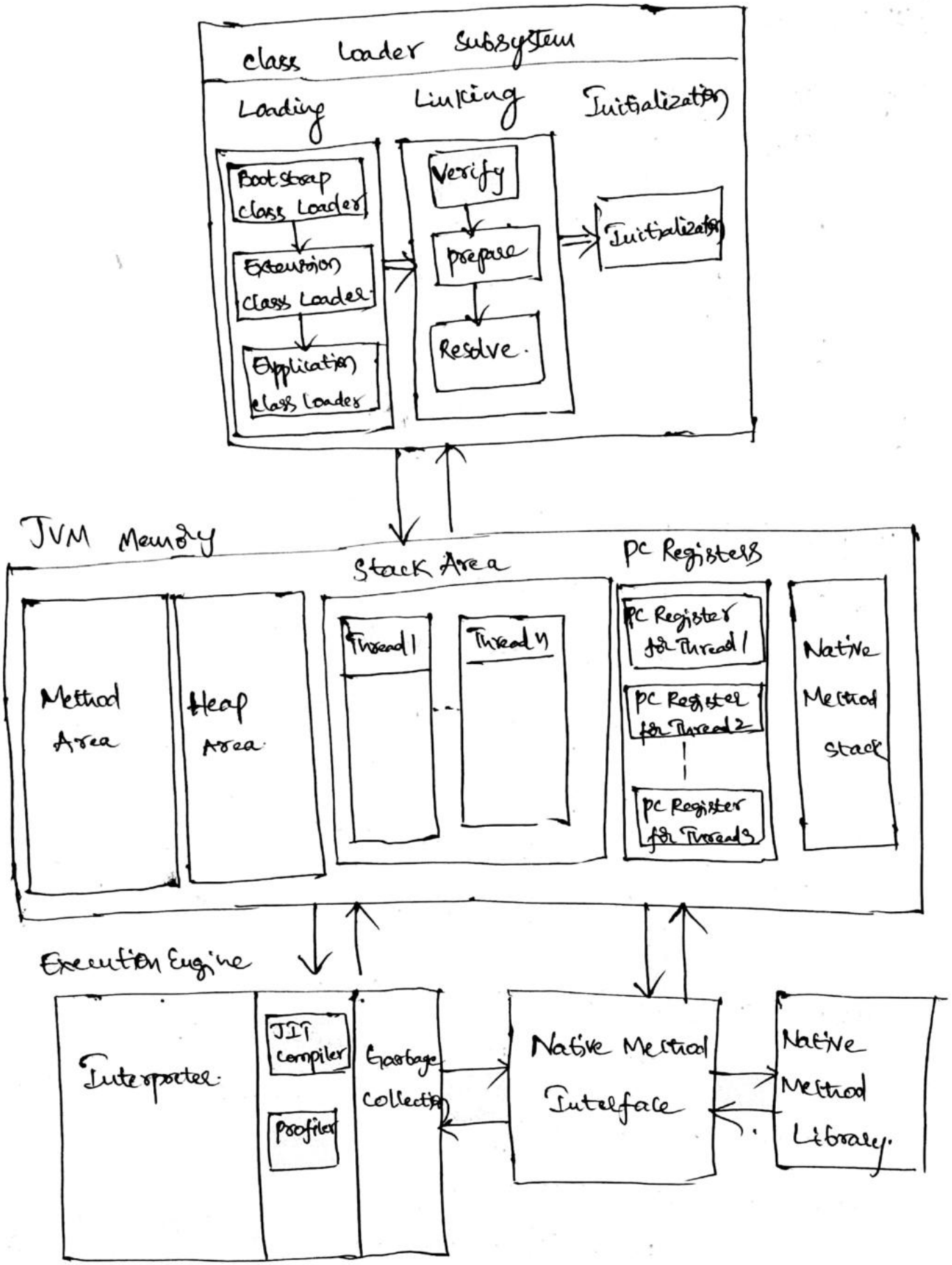
This byte code will be executed by the JRE (Java Runtime Environment).

JRE is the implementation of Java Virtual Machine which analyzes the byte code, interprets the code and executes it.

Every Java developer must know the external architecture of JVM, it enables to write more efficient code.

Scanned with CamScanner

JVM Architecture Diagram :-



Scanned with CamScanner

How does JVM work ?

JVM is divided into three main sub systems.

1. Class Loader Subsystem
2. JVM Memory (or) Runtime Data Area.
3. Execution Engine.

1. Class Loader Subsystem :-

Java's dynamic class loading functionality is handled by the class loader sub system. It loads, links and initializes the class file when it refers to the class for the first time,

Loading :- classes will be loaded by this component.

(i) Bootstrap class loader :-

Responsible for loading class from the bootstrap class path from rt.jar (runtime jar file) → core java classes.

(ii) Extension class loader :-

Responsible for loading classes which are inside the ext folder (jre\lib)

(iii) Application class loader :- Responsible for loading Application classes.

Scanned with CamScanner

Linking :-

(i) Verify ⇒ Byte code verifier will verify whether the generated byte code is proper (or) not.

(ii) prepare ⇒ All static variables, memory will be allocated and assigned with default value.

(iii) Resolve ⇒ All symbolic memory references are replaced with the original references from Method area.

Initialization :-

All static variables will be assigned with the original values and the static block will be executed.

JVM Memory (or) Runtime Data Area :-

1) Method Area :- All the class level data will be stored here including "static variables". There is only one method area per JVM, and it is a "shared resource".

Shared memory multiple threads

2) Heap Area :- → All the objects and their corresponding instance variables will stored here.

Scanned with CamScanner

3. Stack Area:-

For every thread, a seperate runtime stack is created.

4. PC Registers:- Each thread will have seperate PC registers, to holds the address of current executing instruction. once the instruction is executed the PC Register will be updated with the next instruction.

5. Native Method Stack:- it holds native method information. For every thread seperate native method stack created.

Execution Engine:-

The Execution Engine reads the byte code and executes it piece by piece.

The Execution Engine will use Interpreter and JIT (Just Intime) compiler, which compiles the byte code and changes into native code.

This native code will be used directly for repeated method calls, which improve performance of the system.

Scanned with CamScanner

Garbage collector :-

collection and removes unreferenced ~~objects~~ Objects.

Native Method Interface :-

provides Interaction with Native method libraries.

Native Method libraries :-

This is a collection of Native libraries, which are required for execution Engine.

JVM profiler :- JVM profiler is the process of monitoring various JVM Level parameters such as method execution, thread execution, object creation.

How "Hello World" program works

To create a simple Java program, you need to create a class that contains main( ) method.

Requirements to Run Java program

1. Install JDK.
2. Set path for JDK.
3. Create a Java program and save as ".java"
4. Compile the Java program using "javac".

Scanned with CamScanner

5. Run the program using "java" command.

Important Terms (Keywords) used in while writing Java program

class	class keyword is used to declare a class in Java
public	public keyword is an access modifier which represents visibility
static	Static is a keyword. If we declare any method as static, there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it does n't require to create an object to invoke main method.
void	void is the return type of the method. It does n't return any value.
main( )	represents the entry of the program.
String[] args	is used for command arguments.
System.out.println(" ")	This statement used to print statements. System is a class. Out is the object of PrintStream class and declared as static member in System class. println( ) is a method defined in PrintStream class.

Scanned with CamScanner

Valid prototypes for public static void main(...)

1) public static void main(String[] args) ✓
2) static public void main(String[] args) ✓
3) public static void main(String args[]) ✓
4) public static void main(String []args) ✓
5) static public void main(String []args) ✓
6) public static void main(String ... args) ✓
7) public static final void main(String[] args) ✓

Object Oriented Programming Concepts :-

Java is almost purely Object Oriented programming language. Everything in Java program can be defined as Object. Everything in Java program written with in Java classes.

Scanned with CamScanner

Object Oriented Programming :-

Object oriented programming (OOP) is at the core of Java. All Java programs are to atleast some extent of object oriented.

→ All computer programs consists of two elements (i) code (ii) data.

→ Some programs are written around "what is happening" around it's code called "procedure oriented programming".

→ The procedure oriented programming model characterizes the program as a series of linear steps.

Ex:- C Language.

→ Some programs are written around "who is being affected", around its data called "Object oriented programming".

→ In Object oriented programming organizes a program around its data and a set of well defined interfaces to that data.

Abstraction:-

An essential element of Object oriented programming is "abstraction".

Scanned with CamScanner

→ "Representation of essential features without Knowing implementation details called Abstraction".

→ Java programing have well defined mechanism through "interfaces" to define Abstraction.

Three OOp principles :-

1. Encapsulation 2. Inheritance 3. polymorphism.

Encapsulation :- wrapping (or) binding of data and code in to single unit called "Encapsulation".

These encapsulated Entity in Java programing called "class" and its instance called "Object"

→ Encapsulation provides a protective wrapper that prevents code and data from un authorized and unsafe operations.

Inheritance :- Accessing properties from one class to another class called "Inheritance".

Polymorphism :- polymorphism means having many forms.

polymorphism allows you to define a single method in different ways by taking different kinds of inputs.

Scanned with CamScanner

Datatypes, Variables and Arrays

Java is strongly typed language:-

Every variable has a type, and every type is strictly checked and defined.

All assignments to the variables should checked.

The compiler checks all expressions and parameters to ensure that the types are compatible before compiler will finish compiling the class.

Java Primitive Types:-

4 Groups

1 Integers → byte, short, int, long.

2. Floating point Numbers → float & double.

3. Characters → char.

4. Boolean → boolean.

Integers:- Java supports integer datatypes are signed positive & negative values. NO unsigned int datatypes

Name	width	
long	64	
int	32	-2,147,483,648 to 2,147,483,647
short	16	-32,768 to 32767
byte	8	-128 to 127

Scanned with CamScanner

Floating point Numbers :- also known as real numbers are used when evaluating expressions.

Name	width
double	64
float	32

Characters :-

Ezhil - Tamil programing Language influenced by python, Basic

Java use Unicode to represent characters.

Unicode defines a fully international characterset that can represent all of the characters found in all human languages.

Name	width
char	16

c/c++ 8 bit char set only for ISO - Latin - 1.

Achieves global portability

Boolean :-

Java has primitive type called boolean for logical values. It can have only one of two possible values true (or) false.

Expressions to use in conditions

if(b) is similar to if(b == true)

if(!(b)) is similar to if(b == false)

Scanned with CamScanner

Ex:- class Scope {

```
    int x;
    public static void main(String[] args){
        x = 10;
        int y;
        if(x == 10) {
            int z = 20;
            System.out.println("z = " + z);
        }
        z = z+10; // Error
    }
}
```

Life time of a variable:-

The life time of a variable is confined to its scope.

NOTE:- Although blocks can be nested, you can't declare a variable to have the same name as one in enter scope.

```
class ScopeErr{
    public static void main (String[] args){
        int bar = 1;
        {int bar = 2; // compile-time Error, bar already defined.
    }
}
```


Scanned with CamScanner

Variables :-

The variable is the basic unit of storage in java program.

Declaration of variable :-

```
type identifier [=value] [, identifier [= value] . . . ];
```

Dynamic Initialization :-

Java allows variables to be initialized dynamically using any expression at the time the variable declared.

Ex :-
```
double c = Math.sqrt(a*a + b*b);
```

Scope & Life time of variables :-

Java allows variables to be declared with in any block.

A block is begun with an opening curly brace and ended by a closing curly brace.

A block defines a scope.

The scope defined by a method begins with its opening curly brace and closed curly brace.

"Scopes can be nested."

Scanned with CamScanner

Type Conversion and Casting:-

Type casting is to change a variable of one data type to another.

If the types of source variable and destination variable is same then java will perform the conversion automatically.

Java Type conversion
- Automatic Type conversion (Widening conversion)
- Explicit type conversion. (Narrowing conversion)

Automatic Type conversion:- (Widening Conversion)

Java automatic type conversion will takes place if the following conditions are met.

(i) The two types are compatable.

(ii) The destination type is larger than source type.

Ex:-
```
int a;
int b;
b = a; // No Type conversion required.

float x;
int y;
x = y; // No Type conversion.
```


Scanned with CamScanner

Explicit Type Conversion :- (Narrowing Conversion)

Suppose, if you want to assign an 'int' value to a 'byte' variable, the conversion will not be performed automatically. To create a conversion between two incompatable types, Explicit type casting required.

syntax:-

```
(target-type)value;
```

Ex:-

```
int a;
byte b;
b = (byte)a;
```

Type promotion Rules :-

Java type promotion rules :-

(1) If one operand is long, the whole expression is promoted to long.

(2) If one operand is float, the entire expression promoted to float.

(3) If one operand is double, the entire expression is promoted to double.

Scanned with CamScanner

Example program :-

```
class TypePromote {
    public static void main(String[] args) {
        byte b = 42;
        char c = 'a';
        short s = 1024;
        int i = 50000;
        float f = 5.67f;
        double d = .1234;
        double result = (f*b) + (i/c) - (d/s);
        System.out.println("result = " + result);
    }
}
```

Explanation :- In above Expression

f*b promoted to float.

i/c promoted to int

d*s promoted to double

The entire expression promoted to double.

Scanned with CamScanner

Arrays :-

An array is set of data items which belong to similar datatype.

Syntax for one dimensional array :- (Declaration & Initialization)

```
type var_name[];
var_name = new type[size];
```

(or)

```
type var_name[] = new type[size];
```

Declaration

Initialization :-

```
type arr_name[] = { set of values seperated by comma };
```

Sample program for Declaration & Initialization of Array

```
// SampleArray.java
class SampleArray{
    public static void main(String[] args) {
        double numb[] = { 10.1, 11.2, 12.3, 13.4, 15.4};
        double result = 0;
        for(int i=0; i<5; i++)
            result = result + numb[i];
        System.out.println("Average is"+(result/5));
    }
}
```


Scanned with CamScanner

Multidimensional Arrays:-

→ Multi dimensional Array is an "array of array."

Multi dimensional arrays used to represent "Matrices"

Syntax:-

```
datatype mname[][] = new datatype[rows][columns];
```

mname → Matrix name; rows → no of rows; columns → columns.

Example:-

```
int a[][] = new int[5][5];
float b[][] = new float[3][3];
```

```
//matrix.java
class Matrix{
    public static void main(String[] args){
        int a[][] = new int[4][4];
        int i, j, k=0;
        for(i=0; i<4; i++){
            for(j=0; j<i+1; j++){
                a[i][j] = k;
                k++;
            }
        }
        for(i=0; i<4; i++){
            for(j=0; j<i+1; j++){
                System.out.print(a[i][j] + " ");
            }
            System.out.println();
        }
    }
}
```

Output

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


Scanned with CamScanner

Sample program - Reading & displaying Matrix elements (32)

```
// SampleMatrix.java
import java.util.*;
class SampleMatrix {
    public static void main(String[] args){
        int a[][] = new a[10][10];
        int i, j, r, c;
        Scanner sin = new Scanner(System.in);
        System.out.println("Enter rows & columns");
        r = sin.nextInt();
        c = sin.nextInt();
        System.out.println("Enter Matrix elements");
        for(i=0; i<r; i++){
            for(j=0; j<c; j++){
                a[i][j] = sin.nextInt();
            }
        }
        System.out.println("Matrix elements are\n");
        for(i=0; i<r; i++){
            for(j=0; j<c; j++){
                System.out.print(a[i][j]+" ");
            }
            System.out.println();
        }
    }
}
```


Scanned with CamScanner

// Input & output

Enter rows & columns

2 2

Enter Matrix elements 2 3 4 5

Matrix Elements are

2 3

4 5

---

practice programs on Matrices

1. Addition & subtraction of two matrices.
2. Multiplication of two matrices.
3. Find transpose of Matrix.
4. Sum of all elements of given Matrix.

How to find the ~~out~~ Size of the array?

→ The size of the array refers the number of elements that an array can hold.

→ The "length" instance variable on array object returns the number of elements that are actually in use.

Ex:- class ArrayLength{

```
public static void main(String[] args){
    int a1[] = new int[10];
    int a2[] = {4, 3, 2, 1, 5};
    System.out.println("a1 length = "+ a1.length);
    System.out.println("a2 length = "+ a2.length);
}
}
```

output:-
a1.length = 10
a2.length = 5

Scanned with CamScanner

Operators:-

Java provides a rich set of operators.

Java operators grouped in the following way.

(i) Arithmetic operators.

(ii) Bitwise operators

(iii) Relational operators.

(iv) Boolean logical operators.

(v) Assignment operator.

(vi) Ternary operator.

Scanned with CamScanner

JAVA OPERATORS:

Java provides a rich set of operators to manipulate variables. We can divide all the Java operators into the following groups –

- Arithmetic Operators
- Relational Operators
- Bitwise Operators
- Logical Operators
- Assignment Operators
- Misc Operators

1. Arithmetic Operators:

Arithmetic operators are used in mathematical expressions in the same way that they are used in algebra. The following table lists the arithmetic operators –

Assume integer variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
+ (Addition)	Adds values on either side of the operator.	A + B will give 30
- (Subtraction)	Subtracts right-hand operand from left-hand operand.	A - B will give -10
* (Multiplication)	Multiplies values on either side of the operator.	A * B will give 200
/ (Division)	Divides left-hand operand by right-hand operand.	B / A will give 2
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.	B % A will give 0
++ (Increment)	Increases the value of operand by 1.	B++ gives 21
-- (Decrement)	Decreases the value of operand by 1.	B-- gives 19

Scanned with CamScanner

```
public class Test {

   public static void main(String args[]) {
      int a = 10;
      int b = 20;
      int c = 25;
      int d = 25;

      System.out.println("a + b = " + (a + b) );
      System.out.println("a - b = " + (a - b) );
      System.out.println("a * b = " + (a * b) );
      System.out.println("b / a = " + (b / a) );
      System.out.println("b % a = " + (b % a) );
      System.out.println("c % a = " + (c % a) );
      System.out.println("a++   = " +  (a++) );
      System.out.println("a--   = " + (a--) );

      // Check the difference in d++ and ++d
      System.out.println("d++   = " +  (d++) );
      System.out.println("++d   = " +  (++d) );
   }
}
```

```
/* output

a + b = 30
a - b = -10
a * b = 200
b / a = 2
b % a = 0
c % a = 5
a++   = 10
b--   = 11
d++   = 25
++d   = 27
*/
```

2. Relational Operators:

There are following relational operators supported by Java language.

Assume variable A holds 10 and variable B holds 20, then –

Operator	Description	Example
== (equal to)	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.

Scanned with CamScanner

!= (not equal to)	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
> (greater than)	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
< (less than)	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>= (greater than or equal to)	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<= (less than or equal to)	Checks if the value of left operand is less than or equal to the value of right operand, if yes then condition becomes true.	(A <= B) is true

```
public class Test {

   public static void main(String args[]) {
      int a = 10;
      int b = 20;

      System.out.println("a == b = " + (a == b) );
      System.out.println("a != b = " + (a != b) );
      System.out.println("a > b = " + (a > b) );
      System.out.println("a < b = " + (a < b) );
      System.out.println("b >= a = " + (b >= a) );
      System.out.println("b <= a = " + (b <= a) );
   }
}


/*output
a == b = false
a != b = true
a > b = false
a < b = true
b >= a = true
b <= a = false */
```

3. Bitwise Operators:

Java defines several bitwise operators, which can be applied to the integer types, long, int, short, char, and byte.

Bitwise operator works on bits and performs bit-by-bit operation. Assume if a = 60 and b = 13; now in binary format they will be as follows –

a = 0011 1100
b = 0000 1101
---
a&b = 0000 1100

Scanned with CamScanner

a | b = 0011 1101
a^b = 0011 0001
~a = 1100 0011

The following table lists the bitwise operators –

Assume integer variable A holds 60 and variable B holds 13 then –

Operator	Description	Example
& (bitwise and)	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
| (bitwise or)	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^ (bitwise XOR)	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~ (bitwise compliment)	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<< (left shift)	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>> (right shift)	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 1111
>>> (zero fill right shift)	Shift right zero fill operator. The left operands value is moved right by the number of bits specified by the right operand and shifted values are filled up with zeros.	A >>>2 will give 15 which is 0000 1111

```
public class Test {

   public static void main(String args[]) {
      int a = 60;    /* 60 = 0011 1100 */
      int b = 13;    /* 13 = 0000 1101 */
      int c = 0;

      c = a & b;        /* 12 = 0000 1100 */
      System.out.println("a & b = " + c );

      c = a | b;        /* 61 = 0011 1101 */
      System.out.println("a | b = " + c );

      c = a ^ b;        /* 49 = 0011 0001 */
      System.out.println("a ^ b = " + c );
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

Scanned with CamScanner

```
      c = ~a;              /*-61 = 1100 0011 */
      System.out.println("~a = " + c );

      c = a << 2;          /* 240 = 1111 0000 */
      System.out.println("a << 2 = " + c );

      c = a >> 2;          /* 15 = 1111 */
      System.out.println("a >> 2  = " + c );

      c = a >>> 2;         /* 15 = 0000 1111 */
      System.out.println("a >>> 2 = " + c );
   }
}
```

```
/*

Output
a & b = 12
a | b = 61
a ^ b = 49
~a = -61
a << 2 = 240
a >> 2  = 15
a >>> 2 = 15

*/
```

4. Logical Operators:

The following table lists the logical operators –

Assume Boolean variables A holds true and variable B holds false, then –

Operator	Description	Example
&& (logical and)	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false
| | (logical or)	Called Logical OR Operator. If any of the two operands are non-zero, then the condition becomes true.	(A | | B) is true
! (logical not)	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true then Logical NOT operator will make false.	!(A && B) is true

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

Scanned with CamScanner

```
public class Test {

   public static void main(String args[]) {
      boolean a = true;
      boolean b = false;

      System.out.println("a && b = " + (a&&b));
      System.out.println("a || b = " + (a||b) );
      System.out.println("!(a && b) = " + !(a && b));
   }
}
```

This will produce the following result –

```
/*

Output
a && b = false
a || b = true
!(a && b) = true


*/
```

5. Assignment Operators:

Following are the assignment operators supported by Java language –

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator. It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator. It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assign the result to left operand.	C %= A is equivalent to C = C % A

Scanned with CamScanner

<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

```
public class Test {

   public static void main(String args[]) {
      int a = 10;
      int b = 20;
      int c = 0;

      c = a + b;
      System.out.println("c = a + b = " + c );

      c += a ;
      System.out.println("c += a  = " + c );

      c -= a ;
      System.out.println("c -= a = " + c );

      c *= a ;
      System.out.println("c *= a = " + c );

      a = 10;
      c = 15;
      c /= a ;
      System.out.println("c /= a = " + c );

      a = 10;
      c = 15;
      c %= a ;
      System.out.println("c %= a  = " + c );

      c <<= 2 ;
      System.out.println("c <<= 2 = " + c );

      c >>= 2 ;
      System.out.println("c >>= 2 = " + c );

      c >>= 2 ;
      System.out.println("c >>= 2 = " + c );

      c &= a ;
      System.out.println("c &= a  = " + c );

      c ^= a ;
```


Scanned with CamScanner

```
      System.out.println("c ^= a   = " + c );

      c |= a ;
      System.out.println("c |= a   = " + c );
   }
}
```

```
/*
Output
c = a + b = 30
c += a  = 40
c -= a = 30
c *= a = 300
c /= a = 1
c %= a  = 5
c <<= 2 = 20
c >>= 2 = 5
c >>= 2 = 1
c &= a  = 0
c ^= a   = 10
c |= a   = 10
*/
```

6. Miscellaneous Operators:

There are few other operators supported by Java Language.

(i) Conditional Operator (? :)

Conditional operator is also known as the **ternary operator**. This operator consists of three operands and is used to evaluate Boolean expressions. The goal of the operator is to decide, which value should be assigned to the variable. The operator is written as –

variable x = (expression) ? value if true : value if false

Example

```
public class Test {

   public static void main(String args[]) {
      int a, b;
      a = 10;
      b = (a == 1) ? 20: 30;
      System.out.println( "Value of b is : " +  b );

      b = (a == 10) ? 20: 30;
      System.out.println( "Value of b is : " + b );
   }
}
```


Scanned with CamScanner

```
/*Output
Value of b is : 30
Value of b is : 20 */
```

(ii) instanceof Operator

This operator is used only for object reference variables. The operator checks whether the object is of a particular type (class type or interface type). instanceof operator is written as –

(Object reference variable) instanceof (class/interface type)

If the object referred by the variable on the left side of the operator passes the IS-A check for the class/interface type on the right side, then the result will be true. Following is an example –

Example 1)

```
public class Test {

   public static void main(String args[]) {

      String name = "James";

      // following will return true since name is type of String
      boolean result = name instanceof String;
      System.out.println( result );
   }
}
```

```
/*Output
true

 */
```

Example 2)

```
class Vehicle {}

public class Car extends Vehicle {

   public static void main(String args[]) {

      Vehicle a = new Car();
      boolean result =  a instanceof Car;
      System.out.println( result );
   }
}
```

This will produce the following result –

```
/*
Output
true
*/
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

Scanned with CamScanner

Precedence of Java Operators:

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator –

For example, x = 7 + 3 * 2; here x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3 * 2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	expression++ expression--	Left to right
Unary	++expression --expression +expression –expression ~ !	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >> >>>	Left to right
Relational	< > <= >= instanceof	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %= ^= |= <<= >>= >>>=	Right to left

(Courtesy: https://www.tutorialspoint.com/java/java_basic_operators.htm)

Scanned with CamScanner

Control Statements:-

In Any programing language, control statements are used to control the flow of execution based on the changes to the state of the program.

Java's program control statements can be put in to following categories.

- (i) selection
  - if
  - switch
- (ii) Iteration
  - while
  - do-while
  - for
  - Enhanced for loop (called for-each loop Java 5.0 feature)
- (iii) Jump
  - break
  - continue
  - return

Java control statements syntax & usage similar to C (or) C++ programing languages.

Java Selection Statements:-

Selection statements allow the programmer to control the flow of program execution based on conditions, which are known only at runtime.

(i) if statement:-

Syntax:- if(condition){
// true block statements
}

Simple if

Scanned with CamScanner

If-else

syntax:-

```
if(condition){
    //true block statements
}
else{
    //false block statements
}
```

NOTE:-
condition in any 'if' expression returns boolean value.

If condition is true "boolean true" value returned else boolean false returned.

Nested If:-

syntax:-

```
if(condition){
    if(condition){
        //block of statements
        if(condition){
            }
        else{
            //block of statements
        }
    }
}
```

If-else-If ladder:-

syntax:-

```
if(condition){
    --
}
else if(condition){
    --
}
else{ //block of statements
```


Scanned with CamScanner

(ii) Switch statement :-

"switch" statement is Java's multiway branch statement.

⇒ It provides an easy way to dispatch execution to different parts of code based on the value of expression.

Syntax :-

```
switch (expression){
    case value1:
        //statement sequence
        break;
    case value2:
        //statement sequence
        break;
    case value3:
        //statement sequence
        break;
        |
        |
        |
    case valueN:
        //statement sequence
        break;
    default:
        // default statement sequence
}
```

Example program :-

Write a Java program to perform arithmetic operations using switch-case statement.

Scanned with CamScanner

Nested switch statements :-

A switch statement can be defined in another switch statement also.

Syntax:-

```
switch (conditionalexpression){
    case value1:
        switch(expression){
            case value11:
                //block
                break;
            case value12:
                //block
                break;
        }
    case value2:
        }
```

Example:-

```
switch (count){
    case 1:
        switch (target){
            case 0:
                //block
                break;
            case 1:
                //block
                break;
        }
```


Scanned with CamScanner

NOTE:-

In switch statement "break" statement is optional. If programmer omit the break, execution will continue into next case.

```
//TeluguSeason.Java

class TeluguSeason{
    public static void main(String args[]){
        Scanner sin = new Scanner(System.in);
        System.out.println("Enter month number");
        int m = sin.nextInt();
        switch(m){
            case 1:
            case 2:
                System.out.println("Vasantha");
                break;
            case 3:
            case 4:
                System.out.println("Greeshma");
                break;
            case 5:
            case 6:
                System.out.println("varsha");
                break;
            case 7:
            case 8:
                System.out.println("Sharath");
                break;
            case 9:
            case 10:
```


Scanned with CamScanner

```
        System.out.println("Hemantha");
        break;
    Case 11:
    case 12:
        System.out.println("Shishira");
        break;
    }
  }
}
```

Output:- Enter month number 12

Shishira

Iteration Statements:-

Java's iteration statements are for, while and do-while.

Iteration also called as "loop" statement, which executes a loop repeatedly until the termination condition is met.

(i) while loop:-

syntax:-

```
while(condition){
    //body of loop
}
```

(ii) do-while loop:-

syntax:

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


Scanned with CamScanner

NOTE:- The diffence between while loop & do-while loop is, the do-while loop always exeutes its body atleast once, because its conditional expression is at the bottom of the loop. The while loop may not exeutes at least once, if the conditional expression false.

Practice programs:

while loop:

(1) sum of digits of a number.

(2) Armstrong number.

(3) strong number.

Example program for do-while.

```
//DoWhileEx.java
class DoWhileEx {
    public static void main(String[] args) {
        int n=10;
        do {
            System.out.print(n + " ");
            n--;
        } while (n>0);
    }
}
```


Scanned with CamScanner

Output:- 10 9 8 7 6 5 4 3 2 1

(iii) for loop:-

Syntax:- for (initialization ; condition ; iteration) {

//body of the loop

}

Ex:-

```
for (int i=0; i<n; i++){
  //- -
}
```

Infinite for loop:-

```
for( ; ; ){
  //
}
```

For-Each version of for loop:-

A for-each style loop is designed to cycle through a collection of objects such as an array in sequential fashion from start to finish.

For iterating collection of elements, programmers use for-each style of loop.

Scanned with CamScanner

Syntax :-

```
for (type itr_var : collection) {
    //Statement - block
}
```

Ex :-

```
int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int sum = 0;
for (int x : a) {
    sum = sum + x;
}
```

//Sum of array elements using for-each loop.

//SumArray.java.

```
class SumArray {
    public static void main (String[] args) {
        int a[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        int sum = 0;
        for (int x : a) {
            sum = sum + x;
        }
    } System.out.println("Sum is" + sum);
}.
```


Scanned with CamScanner

Iterating Multidimensional Arrays using for-each loop

Multidimensional array called as "Array of Array".

=> when iterating "multidimensional array, each iteration obtains the next array, not an individual element.

This is illustrated with following example using for-each loop.

```
//SumMatrix.java
class SumMatrix{
    public static void main(String[] args){
        int a[][] = new int a[10][10];
        int r, c, i, j, sum = 0;
        Scanner sin = new Scanner(System.in);
        System.out.println("Enter rows & columns");
        r = sin.nextInt();
        c = sin.nextInt();
        System.out.println("Enter Matrix elements");
        for(i=0; i<r; i++){
            for(j=0; j<; j++){
                a[i][j] = sin.nextInt();
            }
        }
        // Using for-each style to find sum of Matrix
        for(int x[] : a){
            for(int y : x){
                sum = sum + y;
            }
        }
```


Scanned with CamScanner

// Display matrix elements using for-each style

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

```
System.out.println("Sum of Matrix elements is " + sum);
}
}
```

output :- Enter rows & columns 2 2

Enter Matrix elements 1 2 3 4

1 2

3 4

Sum of Matrix elements is 10

Practice programs :-

(1) Write a Java program to implement Linear Search using for-each style loop.

(2) Write a Java program to implement Selection sort using for-each style loop.

(3) Write a Java program to implement Binary Search using for-each style loop.

Scanned with CamScanner

Jump Statements

Java supports three jump statements

(i) break (ii) continue (iii) return.

These statements transfer control to another part of program.

(i) break statement :-

Uses of break statement

1) Terminates a 'case' block statements in switch statement (already discussed in switch-case)

2) Exit from a loop.

3) civilized form of "goto"

Exit from a loop using break :-

By using break, programmer can terminated immediately from the loop.

```
//BreakLoop.java.

class BreakLoop{
    public static void main(String[] args){
        for(int i=0; i<100; i++){
            if(i==10){
                break;
            }
            System.out.println(i+" ");
        }
        System.out.println("Loop terminates at "+i+" th iteration");
    }
}
```


Scanned with CamScanner

Output :-

0 1 2 3 4 5 6 7 8 9

Loop terminates at 10 th iteration

NOTE:- when used set of nested loops, the break statement will only break innermost loop only.

Civilized form of goto :-

Java directly does n't have "goto" statement because of not supporting unconditional jumping.

But in someplaces, goto may be useful. To overcome this situation java supported "goto" type of functionality using "break" statement

syntax :-

```
break label;
```

```
//CivilBreak.java
class CivilBreak {
    public static void main(String args[]) {
        boolean t = true;
        first: {
            second: {
                third: {
                    System.out.println("Before break");
                    if(t){
                        break second;
```


Scanned with CamScanner

```
        System.out.println("This won't execute");
    }
    System.out.println("This won't execute");
}
System.out.println("This is after second block");
}
}
```

Output :- Before break
This is after second block.

Example :- Use of labeled break statement to exit from nested loop.

```
//BreakOuter.java
class BreakOuter {
    public static void main(String[] args) {
        outer: for(int i=0; i<10; i++) {
            System.out.println("Iteration" + i + ":");
            for(int j=0; j<100; j++) {
                if (j == 10)
                    break outer;
                System.out.print(j + " ");
            }
            System.out.println("This won't print");
        }
        System.out.println("Loop complete");
    }
}
```


Scanned with CamScanner

Output :-

Iteration 0: 0 1 2 3 4 5 6 7 8 9

Loop complete.

continue :-

In Java programming "continue" used in two cases.

(i) skipping the current iteration execution and directly transfer the control to next iteration.

(2) civilized form of "goto" using "continue"

```
continue label;
```

Example program :- Skipping current iteration

```
//OddSeriex.java
class OddSeries{
    public static void main(String[] args){
        for(int i=0; i<10; i++){
            if(i%2==0){
                continue;
            }
            else{
                System.out.println(i+" ");
            }
        }
    }
}
```

Output :- 1 3 5 7 9

Scanned with CamScanner

Example: Using continue with a label:

```
//ContinueLabel.java
class ContinueLabel{
    public static void main(String[] args){
        outer: for(int i=0; i<10; i++){
            for(int j=0; j<10; j++){
                if(j>i){
                    System.out.println(i);
                    continue outer;
                }
                System.out.println(" " + (i*j));
            }
        }
    }
}
```

output:-

```
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```


Scanned with CamScanner

(iii) return statement :-

The return statement is used to explicitly return from a method.

This statement causes program control to transfer back to the caller of the method.

```
//ReturnEx.java
class ReturnEx{
    public static void main(String[] args){
        boolean t = true;
        System.out.println("Before return");
        if(t){
            return;
        }
        System.out.println("This won't execute");
    }
}
```

Output :- Before return.

Scanned with CamScanner

Introducing Classes :-

Class :-

"Class is a logical template, which it defines the shape and nature of the object".

Class forms the basis for object oriented programming in Java.

> "Class is a template for an object, an object is an instance of a class."

General form of a class :-

when you define a class, programmer should declare its exact form and nature by "specifying data members and methods that operate on data members."

Syntax :-

```
class classname{
    type instance-variable1;
    type instance-variable2;
    //.
    type instance-variablen;
    type methodname1(parameter-list){
        // body
    }
    //-
    type methodname(parameter-list){
        // body
}}
```


Scanned with CamScanner

→ The data or variable with in class are called "instance variables" and the code operates on data called "methods".

Adding Methods to a class:-

"Method" referred as a function like in c programming.

Syntax:-

```
returntype  methodName(parameter-list){
    // body of method
    //return value;
}
```

→ returntype specifies the data returned by the method. If the method does n't return a value, its return type is "void"

Declaring Object reference and creating Objects:-

Creating objects in java is a two step process.

(i) Declaring reference (declaring a variable of class type)

(ii) creating physical object through "new" keyword and assigning to class variable (object reference).

Syntax:-

```
ClassName obj;
obj = new ClassName();
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

Scanned with CamScanner

```
ClassName obj = new ClassName();
```

Example:-

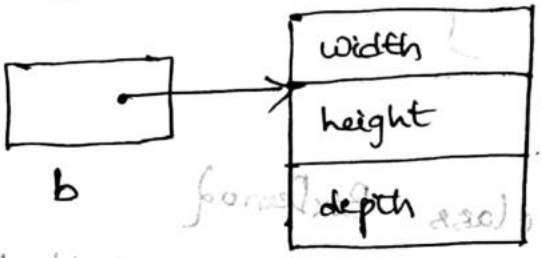
```
class Box{
    double width, height, depth;
    double volume(){
        double v;
        v = width * height * depth;
        return v;
    }
}
class BoxDemo{
    public static void main(String[] args){
        Box b = new Box();
        b.width = 20;
        b.height = 10;
        b.depth = 15;
        double v;
        v = b.volume();
        System.out.println("volume of Box is " + v);
    }
}
```

Output:- Volume of Box is 3000

Scanned with CamScanner

In the above program Box object 'b' created in a single step process. You can also create object in the following way also.


```
Box b;
b = new Box();
```

Statement	Effect
Box b;	
b = new Box();	

The new operator dynamically allocates memory for an object.

Box object created in JVM.

Adding a method that takes parameters

→ Some methods don't need parameters.

→ parameters allow a method to be generalized.

A parameter is a variable defined by a method that receives a value when a method is called.

→ parameterized method can operate on variety of data which is sent by programmer at run time.

Scanned with CamScanner

```
//BoxDemo.java

class Box{
    double width, height, depth;
    void setDim(double w, double h, double d){
        width = w;
        height = h;
        depth = d;
    }
    double volume(){
        return width*height*depth;
    }
}
class BoxDemo{
    public static void main(String args[]){
        Box b1 = new Box();
        Box b2 = new Box();
        double v;
        b1.setDim(10,20,15);
        b2.setDim(3,6,9);
        v = b1.volume();
        System.out.println("Box b1 volume is " + v);
        v = b2.volume();
        System.out.println("Box b2 volume is " + v);
    }
}
```


Scanned with CamScanner

Output:- Box b1 volume is 3000
Box b2 volume is 162.

Constructors:-

A constructor initializes an object immediately upon object creation.

"constructor has same name as the class and syntactically similar to method, does n't have return type and does n't return value".

constructor properties:-

(i) constructor name is same to class name
(ii) constructor have no return type, not even "void".
(iii) constructor does n't return value.
(iv) constructor initializes the internal state of an object.
(v) when a class not explicitly define a constructor for a class, then ~~Java Virtual M~~ Java compiler automatically creates a default constructor for the class.

NOTE:- If the programmer does n't add any constructor to class, Java compiler automatically creates one default constructor and initializes the members of the class

Scanned with CamScanner

Example:- // BoxDemo.java


```
class Box {
    double width, height, depth;
    Box() {
        width = 10;
        height = 10;
        depth = 10;
    }
    Box( double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    double volume() {
        return width * height * depth ;
    }
}
class BoxDemo {
    public static void main(String[] args) {
        Box b = new Box();
        Box b1 = new Box(10, 20, 30);
        double vol;
        vol = b.volume();
        System.out.println("volume of Default box is" +vol);
        vol = b1.volume();
        System.out.println("volume dimensions specified at
                                  runtime is" + vol);
```

⇒ Default constructor

Parameterized constructor.

Scanned with CamScanner

```
    }
}
```

Output :- Volume of default box is 1000.0
Volume of dimensions specified at run time is 6000.0

Constructor Overloading :-

Constructor overloading is a form of "polymorphism"
In Java, it is possible to define two (or) more constructors with in the same class and they differ from parameter declarations.

When JVM encounters a call to an overloadable constructor, it executes the appropriate constructor whose parameters match the arguments with constructor.

Example :-

```
// Constructor overloading
// ConOverload.java
class Box{
    double width, height, depth;
    Box () {
        width = -1;
        height = -1;
        depth = -1;
    }
```


Scanned with CamScanner

```
Box(double len){
    width = height = depth = len;
}
Box(double w, double h, double d) {
    width = w;
    height = h;
    double = d;
}
Box(Box ob) {
    width = ob.width;
    height = ob.height;
    depth = ob.depth;
}
double volume(){
    return width * height * depth;
}
}
class CorOverload{
    public static void main(String[] args){
        Box b1 = new Box();
        Box b2 = new Box(2);
        Box b3 = new Box(2,3,4);
        Box b4 = new Box(b3);
```


Scanned with CamScanner

```
double vol;
vol = b1.volume();
System.out.println("volume of default Box is" + vol);
vol = b2.volume();
System.out.println("volume of cube is" + vol);
vol = b3.volume();
System.out.println("volume of shape is" + vol);
vol = b4.volume();
System.out.println("volume of shape duplicated
                    from b3 is" + vol);
}
}
```

output :-

Volume of default Box is 1

volume of cube is 8.

volume of shape is 24

volume of shape duplicated from b3 is 24

Scanned with CamScanner

Method Overloading :-

- In Java, it is possible to define two (or) more methods with in the same class that share the same name and they differ from parameter declarations.

This is said to be "Method overloading"

Method overloading is a form of "polymorphism"

when JVM encounters a call to an overloaded method, it executes the appropriate method, whose parameters match with arguments from calling method.

Example : // MethodOversload.java

```
class Sum{
    int addition( int a){
        return (a+0);
    }
    int addition( int a, int b){
        return (a+b);
    }
    float addition( float a, float b, float c){
        return (a+b+c);
    }
```


Scanned with CamScanner

```
int addition(int a, int b, int c){
        return (a+b+c);
    }
}

class MethodOverload{
    public static void main(String[] args){
        Sum s = new Sum();
        int add;
        System.out.println("calling from single argument method");
        add = s.addition(2);
        System.out.println(add);
        System.out.println("calling from two arguments method");
        add = s.addition(2,5);
        System.out.println(add);
        System.out.println("calling from three arguments
                                float method");
        float fadd = s.addition(2.5, 2.7, 2.3);
        System.out.println(fadd);
        System.out.println("calling from three arguments
                                int method");
        add = s.addition(2,3,5);
        System.out.println(add);
    }
}
```


Scanned with CamScanner

Method overloading valid & Invalid combinations :-

case 1)

```
int method (int a, int b);
int method (float v1, float v2);
```

✓

case 2)

```
int method (int a, int b);
int method (int num);
```

✓

case 3)

```
float method (int a, float b);
float method (float v, int b);
```

✓

case 4)

```
int method (int a, int b)
float method (int v, int v2)
```

✗

Scanned with CamScanner

Understanding "static" keyword :-

- static members
- static methods
- static blocks.

→ Sometimes, you would like to define a class member that will be used independently of an object of that class without reference to specific instance (object)

→ To create such member (or) method declare with "static" keyword.

→ Instance variables declared as "static" are essentially "global variables".

→ Static methods have following restrictions.

(i) They can call only static methods.

(ii) They must only access static data.

(iii) They can n't refer to "this" (or) "super" keyword.

```
//StaticDemo.java
class StaticDemo {
        static int a = 3;   // static members
        static int b;
        static void meth(int x) {   // static method
                System.out.println("x=" + x);
                System.out.println("a=" + a);
                System.out.println("b=" + b);
        }
```


Scanned with CamScanner

```
static {                    // Static block
    System.out.println("Static block Initialized");
    b = a*4;
}
public static void main(String[] args) {
        meth(43);
}
}
```

/* Output:

Static block Initialized

x = 43

a = 3

b = 12

*/

NOTE: Static members & static methods called using class name

Syntax:-

ClassName.method(arglist);

ClassName.member

Scanned with CamScanner

"this" keyword :-

"this" keyword can be used inside any constructor(s) method to refer to the object of the self class.

"this" keyword is similar to "self" in python programing.

Example:-

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

Instance Variable hiding:-

→ Java doesn't allow to declare the local variables and class members with same name inside the same enclosing scope.

→ when a local variable has the same name with instance variable (member), the local variable hides the "instance variable". This is called "Instance variable hiding".

→ To avoid name collisions between instance variables and local variables (those refers with same name) explained in following example.

Scanned with CamScanner

// Use in following way to resolving name-space collisions

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


Passing Objects as parameters :-

In Java programming objects also passed as arguments to methods.

```
// PassObjectEx.java
class Test {
    int a,b;
    Test(int i, int j) {
        a = i;
        b = j;
    }
    boolean equalTo(Test o) {
        if (o.a == a && o.b == b) {
            return true;
        }
        else
        return false;
    }
}
```


Scanned with CamScanner

```
class PassObjectEx{
    public static void main(String[] args){
        Test ob1 = new Test(100, 22);
        Test ob2 = new Test(100, 22);
        Test ob3 = new Test(-1, -1);
        System.out.println("ob1 == ob2:" + ob1.equalsTo(ob2));
        System.out.println("ob1 == ob3:" + ob1.equalsTo(ob3));
    }
}
/* output
  ob1 == ob2 : true
  ob1 == ob3 : false  */
```

Argument Passing mechanisms :-

In general, there are two techniques available to pass an argument to method.

(i) call-by-value (or) pass-by-value

(ii) call-by-reference (or) pass-by reference.

call-by-value :- call-by-value copies the value of an argument in to the formal parameter of the method.

call-by-reference :- In call-by-reference a reference to an argument is passed to the parameter.

Scanned with CamScanner

Call-by-value:

In Java, when you pass a primitive type to a method, it is pass by value.

```
// primitive types are passed by value
// CallByValue.java

class Test {
    void meth(int i, int j) {
        i* = 2;
        j* = 2;
    }
}

class CallByValue {
    public static void main(String[] args) {
        Test ob = new Test();
        int a = 15, b = 20;
        System.out.println("a and b before call:" +
                                    a + " " + b);
        ob.meth(a,b);
        System.out.println("a and b after call:" +
                                    a + " " + b);
    }
}
/* output:
    a and b before call: 15 20
    a and b after call: 15 20
```


Scanned with CamScanner

Call-By-Reference:-

when you pass an object to a method, this is automatically referred as call-by-reference.

→ when you pass reference to a method, the parameter receives it will refer to the same object as that referenced to by the argument.

→ This means, that objects are passed to methods by use reference.

→ Changes to the objects inside the method do affect the object used as an argument.

```
//Objects are passed by reference
//CallByRef.java
class Test{
    int a,b;
    Test(int i, int j){
        a = i;
        b = j;
    }
    void meth(Test o){
        o.a *= 2;  ob.b /= 2;
    }
}
```


Scanned with CamScanner

```
class CallByRef {
    public static void main(String[] args) {
        Test ob = new Test(15, 20);
        System.out.println("ob.a and ob.b before call: "+
                                ob.a + " " +ob.b);
        ob.meth(ob);
        System.out.println("ob.a and ob.b after call: "+
                                ob.a + " " +ob.b);
    }
}

/* Output
  ob.a and ob.b before call: 15 20
  ob.a and ob.b after call: 30 10.  */
```

Returning Objects :-

→ A method can return any type of data, including class types (objects) that created.

```
// Returning an object
// RetOb.java
class Test {
    int a;
    public Test(int i) {
        a = i;
    }
```


Scanned with CamScanner

```
//Method returns Object
Test incByTen() {
    Test temp = new Test(a+10);
    return temp;
  }
}

class RetOb {
    public static void main(String[] args) {
        Test ob1 = new Test(2);
        Test ob2;
        ob2 = ob1.incByTen();
        System.out.println("ob1.a:" + ob1.a);
        System.out.println("ob2.a:" + ob2.a);
        ob2 = ob2.incByTen();
        System.out.println("ob2.a after second
                                increase:" + ob2.a);
    }
}

/* output:
   ob1.a : 2
   ob2.a : 12
   ob2.a after second increase : 22.
*/
```


Scanned with CamScanner

Recursion:

→ Java supports recursion.

→ Recursion is the process of called itself until certain condition satisfied.

```
// A Simple example of Recursion

// RecFact.java

class Factorial {

    long fact(int n) {

        long result;

        if(n==1) return 1;

        result = fact(n-1) * n;

        return result;

    }

}

class RecFact {

    public static void main(String[] args) {

        Factorial f = new Factorial();

        System.out.println(" Factorial of 3 is " + f.fact(3));

        System.out.println(" Factorial of 4 is " + f.fact(4));

        System.out.println(" Factorial of 5 is " + f.fact(5));

    }

}
```


Scanned with CamScanner

```
/* Output
Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120.  */

// Printing Array elements from 1 to 10 using Recursion
// PrintArray.java
class RecPrint {
    int a[];
    RecPrint(int i) {
        a = new int[i];
    }
    // display array recursively
    void printArr(int i) {
        if (i==0) return;
        else printArr(i-1);
        System.out.println("[" + (i-1) + "]" +
                                a[i-1]);
    }
}

class PrintArray {
    public static void main(String[] args) {
        RecPrint ob = new RecPrint(10);
        int i;
```


Scanned with CamScanner

```
    for (i=0; i<10; i++)
        ob.a[i] = i;
    ob.printArr(10);
  }
}

/* output
  [0]  0
  [1]  1
  [2]  2
  [3]  3
  [4]  4
  [5]  5
  [6]  6
  [7]  7
  [8]  8
  [9]  9
*/
```

Access Specifiers:

→ Encapsulation links data with code that manipulates it.

→ Encapsulation provides another important attribute called access control.

→ By controlling access of the members & methods you can prevent misuse.

Scanned with CamScanner

Access level	No Modifier	private	public	protected
Same class	Yes	Yes	Yes	Yes
Sub class	Yes	NO	Yes	Yes
Non subclass	Yes	NO	Yes	Yes


```
// This program demonstrate the diffence between public & private

// AccessTest.java
class Test {
    int a; // default access - No Modifier;
    public int b; // public access
    private int c; // private access
    void setC(int c) {
        c = c;
    }
    int getC() {
        return c;
    }
}

class AccessTest {
    public static void main(String [] args) {
        Test ob = new Test();
        ob.a = 10; // No modifier member accessed
        ob.b = 20; // public member accessed
```


Scanned with CamScanner

```
ob.c = 100; // Error! private member not accessed
ob.setC(100);
System.out.println("a, b and c: " + ob.a + " " +
                    ob.b + " " + ob.getC());
}
}
```


'final' keyword :-

→ A field can be declared as "final".

→ when any field declared as "final", it is prevented from modifying of content.

→ "final" keyword is similar to "const" keyword in 'C' programing.

Ex:- 1) final int FILE_NEW = 1;

2) final int FILE_OPEN = 2;

3) final int FILE_SAVE = 3;

→ In general, final field names referred by Uppercase letters.

Scanned with CamScanner

Introducing Nested and Inner classes :-

In Java programming it is possible to define a class within another classes. such classes are known as "nested classes."

Two different types of nested classes.

(i) static nested classes.

(ii) Non static nested classes ⇒ Inner classes.

(i) static nested classes :-

If a class is defined in another class and that is declared as a static class, then that class is called "static nested class."

> NOTE:- A nested class has access to the members, including private members of the class in which it is nested.

```
// Static nested classes
// StaticNestedDemo.java
class Outer {
    static private int x = 10;
    static class Inner {
        static void display() {
            System.out.println("called from
                                Static Nested class:" +x);
        }
    }
}
```


Scanned with CamScanner

```
    void callStaticInner() {
        Inner.display();
    }
}

class StaticNestedDemo {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.callStaticInner();
    }
}

/* output
   called from static Nested class : 10
*/
```

(ii) <u>Inner classes (or) Non static nested classes</u> :

An Inner class is a non static nested class. It has access to all of the variables and members of it's outer class and may refer to them directly.

<u>Syntax</u>:-

```
class Outer {
    // members
    // methods
    class Inner {
        // members
        // methods
    }
}
```

Outer class members directly accessible to Inner class

Inner class members & methods not directly accessible to Outer class.

Scanned with CamScanner

```
//InnerOuterDemo.java
class Outer{
    private int x = 100;
    class Inner {
        void display ( ){
            System.out.println("Outer class private
                                members accessible :"+x);
        }
    }
    void callInnerDisplay ( ){
        Inner i = new Inner ( );
        i.display ( );
    }
}
class InnerOuterDemo {
    public static void main (String[] args) {
        Outer o = new Outer( );
        o.callInnerDisplay ( );
    }
}
/* output
Outer class private members accessible : 100 */
```


Scanned with CamScanner

Defining an inner class within a for loop

An inner class is also defined within for loop also.

```
//ForInnerDemo.java
class Outer {
    int x = 100;
    void test() {
        for (int i=0; i<10; i++) {
            class Inner {
                void display() {
                    System.out.println("outer class x = "+
                                        x);
                }
            }
            Inner i = new Inner();
            i.display();
        }
    }
}
class ForInnerDemo {
    public static void main(String[] args) {
        Outer o = new Outer();
        o.test();
    }
}
```


Scanned with CamScanner

/* output

```
Outer class  x = 100
outer class  x = 100
outer class  x = 100
outer class  x = 100
outer class  x = 100
outer class  x = 100
outer class  x = 100
outer class  x = 100
outer class  x = 100
outer class  x = 100
```

*/

Garbage Collection:-

In general, Objects are created dynamically, and such objects are destroyed dynamically and their memory released for later reallocation.

→ In C++ programing objects dynamically allocated through "new" keyword and manually released through "delete" keyword.

→ In python programming also, objects released through "del" keyword.

→ Java follows diffent approach.

→ In Java programming, programmer takes the responsibility to create objects.

Scanned with CamScanner

→ But the programmer is not responsible for deallocation of objects.

→ JVM's "Garbage Collector" takes the responsibility to de allocate the objects, when objects no longer needed (or) when no references to an object exist.

How can an object be unreferenced ?

(i) By nulling the reference.

(ii) By assigning a reference to another.

(iii) By anonymous object.

(i) By nulling the reference :-

Ex:-

```
Sample S = new Sample();
S.meth1();
S.meth2();
S = null;
```

→ S is eligible for Garbage collection

(ii) By assigning a reference to another :-

```
Sample s1 = new Sample();
Sample s2 = new Sample();
s1 = s2; // Now the first object referred by s1
         is available for garbage collection.
```


Scanned with CamScanner

(iii) By anonymous object :-

```
new Sample();
```

Anonymous objects automatically deallocated when out of scope of the object creation statement occured.

The "finalize()" method :-

→ Sometimes an object will need to perform some action when it is destroyed.

→ For example,

* If an object is holding some non-java resources, such as opening of external files, (or) connectivity with external databases, (or) opened images.

* In this situation, the programmer takes the responsibility to detach the external resources before an object is destroyed.

→ The finalize() method of Object class called by the "Garbage Collector" before the deletion / destroying the object which is eligible for "Garbage collection" to perform clean-up activity.

Scanned with CamScanner

→ clean-up activity means to close the resources associated with the object.

→ To initiate finalize() method activities, Override finalize() method from java.lang.Object class which is by default parent class for all Java classes in Java world.

Syntax:-

→ Defined in java.lang.Object class

```
protected void finalize( )
```

→ The keyword "protected" is an access specifier that prevents access to finalize() by code defined outside its class.

```
//GarbageDemo.java
class Sample{
    int a[] = new int[20];
    void display(){
        a[0] = 2;
        a[1] = 5;
        System.out.println(" a[0] = " + a[0]);
        System.out.println(" a[1] = " + a[1]);
    }
    protected void finalize( ){ // Override finalize method
        a = null;
        System.out.println(" Calling finalize method ");
    }
}
```


Scanned with CamScanner

```
class GarbageDemo{
    public static void main(String[] args){
        Sample s = new Sample();
        s.display();
        Sample s1 = new Sample();

        s1.display();
        s = null;
        s1 = null;
        System.gc();
    }
}
```


Scanned with CamScanner

Exploring the String class:

- Strings are a sequence of characters. In Java programming language.
- Strings are treated as objects.
- The Java programming provides the **String class** to create and manipulate strings.
- Strings created from String class are mutable. This means where original created string never changed.

Creating Strings:

The most direct way to create a string is to write −

String greeting = "Hello Java World";

Whenever it encounters a string literal in your code, the compiler creates a String object with its value in this case, "Hello Java world!'.

String class constructors:

Sr.No.	Constructor & Description
1	**String()** This initializes a newly created String object so that it represents an empty character sequence.
2	**String(byte[] bytes)** This constructs a new String by decoding the specified array of bytes using the platform's default charset.
3.	**String(byte[] bytes, int offset, int length)** This constructs a new String by decoding the specified subarray of bytes using the platform's default charset
4	**String(char[] value)** This allocates a new String so that it represents the sequence of characters currently contained in the character array argument.
5	**String(char[] value, int offset, int count)** This allocates a new String that contains characters from a subarray of the character array argument.
6	**String(int[] codePoints, int offset, int count)** This allocates a new String that contains characters from a subarray of the Unicode code point array argument.

7	**String(String original)** This initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a copy of the argument string.
8	**String(StringBuffer buffer)** This allocates a new string that contains the sequence of characters currently contained in the string buffer argument.
9	**String(StringBuilder builder)** This allocates a new string that contains the sequence of characters currently contained in the string builder argument.

String class Methods:

Sr.No.	Method & Description
1	**char charAt(int index)** **This method returns the char value at the specified index.**
2	**int compareTo(String anotherString)** **This method compares two strings lexicographically.**
3	**int compareToIgnoreCase(String str)** **This method compares two strings lexicographically, ignoring case differences.**
4	**String concat(String str)** **This method concatenates the specified string to the end of this string.**
5	**boolean contains(CharSequence s)** **This method ceturns true if and only if this string contains the specified sequence of char values.**
6	boolean contentEquals(CharSequence cs) This method compares this string to the specified CharSequence.
7	boolean contentEquals(StringBuffer sb) This method compares this string to the specified StringBuffer.
8	boolean endsWith(String suffix) This method tests if this string ends with the specified suffix.

9	**boolean equals(Object anObject)** **This method compares this string to the specified object.**
10	**boolean equalsIgnoreCase(String anotherString)** **This method compares this String to another String, ignoring case considerations.**
11	static String format(String format, Object... args) This method returns a formatted string using the specified format string and arguments.
12	**byte[] getBytes()** **This method encodes this String into a sequence of bytes using the platform's default charset, storing the result into a new byte array.**
13	void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin) This method copies characters from this string into the destination character array.
14	**int hashCode()** **This method returns a hash code for this string.**
15	**int indexOf(int ch)** **This method returns the index within this string of the first occurrence of the specified character.**
16	int indexOf(int ch, int fromIndex) This method returns the index within this string of the first occurrence of the specified character, starting the search at the specified index.
17	int indexOf(String str) This method returns the index within this string of the first occurrence of the specified substring.
18	int indexOf(String str, int fromIndex) This method returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
19	int lastIndexOf(int ch) This method returns the index within this string of the last occurrence of the specified character.

20	int lastIndexOf(int ch, int fromIndex) This method returns the index within this string of the last occurrence of the specified character, searching backward starting at the specified index.
21	int lastIndexOf(String str) This method returns the index within this string of the rightmost occurrence of the specified substring.
22	int lastIndexOf(String str, int fromIndex) This method returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.
23	**int length()** **This method returns the length of this string.**
24	String replace(char oldChar, char newChar) This method returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.
25	String replaceFirst(String regex, String replacement) This method replaces the first substring of this string that matches the given regular expression with the given replacement.
26	boolean startsWith(String prefix) This method tests if this string starts with the specified prefix.
27	boolean startsWith(String prefix, int toffset) This method tests if the substring of this string beginning at the specified index starts with the specified prefix.
28	String substring(int beginIndex) This method returns a new string that is a substring of this string.
29	String substring(int beginIndex, int endIndex) This method returns a new string that is a substring of this string.
30	**char[] toCharArray()** **This method converts this string to a new character array.**
31	String toLowerCase() This method converts all of the characters in this String to lower case using the rules of the default locale.

32	String toLowerCase(Locale locale) This method converts all of the characters in this String to lower case using the rules of the given Locale.
33	String toString() This method returns the string itself.
34	String toUpperCase() This method converts all of the characters in this String to upper case using the rules of the default locale.
35	String toUpperCase(Locale locale) This method converts all of the characters in this String to upper case using the rules of the given Locale.
36	**String trim()** **This method returns a copy of the string, with leading and trailing whitespace omitted.**
37	static String valueOf(boolean b) This method returns the string representation of the boolean argument.
38	static String valueOf(char c) This method returns the string representation of the char argument.
39	static String valueOf(char[] data) This method returns the string representation of the char array argument.
40	static String valueOf(char[] data, int offset, int count) This method Returns the string representation of a specific subarray of the char array argument.
41	static String valueOf(double d) This method returns the string representation of the double argument.
42	static String valueOf(float f) This method returns the string representation of the float argument.
43	static String valueOf(int i) This method returns the string representation of the int argument.

44	static String valueOf(long l) This method returns the string representation of the long argument.
45	static String valueOf(Object obj) This method returns the string representation of the Object argument.

To concatenate two String objects '+" operator is used.

Example:

String str ="I " + " Like "+"Java Programming";

Which results str as "I Like Java Programming".

```
//Demonstration of Strings creation
//StringEx.java
//Strings Creation
class StringEx {
	public static void main(String args[]){
		String s1="Java Programming";
		String s2=new String("James Gosling");
		char ch[]={'J','a','v','a'};
		String s3=new String(ch);
		byte b[]={78,79,80,81};
		String s4=new String(b);
		StringBuffer sb=new StringBuffer("Java String Buffer");
		String s5=new String(sb);
		char ch1[]={'v','i','n','o','d','k','h','o','s','l','a'};
		String s6=new String(ch1,0,5);
		System.out.println("s1="+s1);
		System.out.println("s2="+s2);
		System.out.println("s3="+s3);
		System.out.println("s4="+s4);
		System.out.println("s5="+s5);
		System.out.println("s6="+s6);
	}
}
/*
F:\BHAVAJAVA\BN>javac StringEx.java
F:\BHAVAJAVA\BN>java StringEx
```

```
s1=Java Programming
s2=James Gosling
s3=Java
s4=NOPQ
s5=Java String Buffer
s6=vinod
*/
```

//Demonstration of selected String methods.

//StringDemo.java

```
class StringDemo {
        public static void main(String[] args) {
                String str1="Hello";
                String str2="Java";
                String str3="World";
                String str4=new String(str2);
                System.out.println("str1 length="+str1.length());
                System.out.println("Character at index 3 in str1="+str1.charAt(1));
                System.out.println("Upper case str2="+str2.toUpperCase());
                System.out.println("Lower case str2="+str2.toLowerCase());
                System.out.println("str1 concatenates str2="+str1.concat(str2));
                if(str1.equals(str2))
                        System.out.println("str1==str2");
                else
                        System.out.println("str1!=str2");
                if(str2.equals(str4))
                        System.out.println("str2==str4");
                else
```

```
                System.out.println("str2!=str4");
        }
}


/*
F:\BHAVAJAVA\BN>javac StringDemo.java
F:\BHAVAJAVA\BN>java StringDemo
str1 length=5
Character at index 3 in str1=e
Upper case str2=JAVA
Lower case str2=java
str1 concatenates str2=HelloJava
str1!=str2
str2==str4
*/
```

Java Program to sort strings in ascending and descending order.

//AscDesStr.java

```
import java.util.*;
class AscDscStr {
        public static void main(String[] args) {
                Scanner sin=new Scanner(System.in);
                System.out.println("Enter no of strings to sort=");
                int n=sin.nextInt();
                String str[]=new String[n];
                System.out.println("Enter array of Strings=");
                for(int i=0;i<n;i++)
                        str[i]=sin.next();
                //sort the strings in Ascending Order
                for(int i=0;i<n;i++) {
                        for(int j=i+1;j<n;j++) {
                                if((str[i].compareTo(str[j]))>0){
```

```
                                String temp=str[i];
                                str[i]=str[j];
                                str[j]=temp;
                        }
        }
    }
            System.out.println("Ascending Order:");
            for(int i=0;i<n;i++)
                    System.out.println(str[i]);
            System.out.println("Descending Order:");
            for(int i=n-1;i>=0;i--)
                    System.out.println(str[i]);
        }
}

/*
F:\BHAVAJAVA\BN>javac AscDscStr.java
F:\BHAVAJAVA\BN>java AscDscStr
Enter no of strings to sort=
5
Enter array of Strings=
Ganga
Krishna
Godhavari
Narmada
Saraswathi

Ascending Order:
Ganga
Godhavari
Krishna
Narmada
Saraswathi
Descending Order:
Saraswathi
Narmada
Krishna
Godhavari
Ganga
*/
```

//Java Program for Searching Strings
// Demonstrate indexOf() and lastIndexOf().

```
class StrIndexDemo {
        public static void main(String args[]) {
                String s = "James Arthur Gosling, often referred to as "+
                        "Dr. Java, OC (born May 19, 1955) is a Canadian"+
                                "computer scientist, best known as the "+
                                "founder and lead designer behind the "+
                                "Java programming language.";
                System.out.println(s);
                System.out.println("indexOf(t) = " +s.indexOf('t'));
                System.out.println("lastIndexOf(t) = " +s.lastIndexOf('t'));
                System.out.println("indexOf(the) = " +s.indexOf("the"));
                System.out.println("lastIndexOf(the) = " +s.lastIndexOf("the"));
                System.out.println("indexOf(t, 10) = " +s.indexOf('t', 10));
                System.out.println("lastIndexOf(t, 60) = " +s.lastIndexOf('t', 60));
                System.out.println("indexOf(the, 10) = " +s.indexOf("the", 10));
  }
}

/*
F:\BHAVAJAVA\BN>javac StrIndexDemo.java
F:\BHAVAJAVA\BN>java StrIndexDemo
James Arthur Gosling, often referred to as Dr. Java, OC
(born May 19, 1955) is a Canadiancomputer scientist,
best known as the founder and lead designer behind the
Java programming language.
indexOf(t) = 8
lastIndexOf(t) = 160
indexOf(the) = 123
lastIndexOf(the) = 160
indexOf(t, 10) = 24
lastIndexOf(t, 60) = 37
indexOf(the, 10) = 123
*/
```

Note: For More relevant information regarding String class refer
https://docs.oracle.com/javase/7/docs/api/java/lang/String.html

ANNAMACHARYA INSTITUTE OF TECHNOLGY AND SCIENCES::RAJAMPET
(AUTONOMOUS)
DEPARTMENT OF CSE

YEAR & SEM: II B.TECH II SEM **ACADEMIC YEAR: 2020-21**
BRANCH : CSE
SUBJECT: OBJECT ORIENTED PROGRAMMING USING JAVA
ASSIGNMENT NO: 1 **TOPICS COVERED: UNIT-I**
****

1.	List and explain the Principles of OOP paradigm. Explain any three object oriented programming features?
2.	Differentiate Procedure Oriented Programming (POP) with Object Oriented Programming (OOP)
3.	What are the advantages of Object Oriented Programming?
4.	List and describe Java Buzzwords.
5.	What is byte code in Java? Explain the working of JVM.
6..	Explain clearly how the following terms are related to Java. i. Architecture-Neutral ii. Robust iii. High-performance iv. Dynamic
7.	Describe Primitive, Boolean data Types?
8.	List and classify the operators used in java. Explain any two operator classifications.
9.	Differentiate between right shift and unsigned right shift operator.
10.	a) Explain different types of control statements available in Java with examples. b) write a java program to print pascal triangle.
11.	Explain jumping statements in java.
12.	Java does not support goto statement. Why? Can you explain how goto type of functionality supported in java.
13.	Explain about type conversion and casting in java
14.	Explain declaration and initialization of single and multidimensional arrays.
15.	Explain the working style of foreach loop with the help of two dimensional array matrix elements accessing.
16.	Define a class? What is the general form of a class? How objects are declared explain with an example?
17.	Explain constructors with an example. Illustrate one scenario where constructors are used? How a constructor is different from other Methods in Java?
18.	Define recursion. Write a java program to find the factorial of a given number using recursion.
19.	Explain method overloading and constructor overloading with an example?
20.	Discuss the necessity of Garbage collection in Java? Explain?
21.	Compare and contrast java's access specifiers with examples.
22.	Explore the importance of static and this keyword.
23.	Write a Java program, which creates String object, store a numerical value in that object, and display the value in words. Ex; 120 should be displayed as ONE TWO ZERO
24.	Explore String class.
25.	Write a Java program to reverse the given long integer.

UNIT-II

INHERITANCE, INTERFACES, PACKAGES

INHERITANCE

- Inheritance is a mechanism in which one object acquires the properties and behaviors of another parent.
- Acquiring properties and methods from parent class to child class called "Inheritance".
- From which class properties and methods acquired called base class or parent class or **super class.**
- To which class properties and methods acquired called derived class or child class or **sub class.**

Advantages:

1. Reusability
2. Extensibility

Syntax:

To support inheritance in Java programming we use 'extends' keyword.

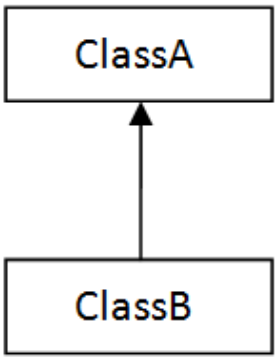
```
class SubClassName extends SuperClassName{
        //Inherits Members and Methods from Super class
          //Members declaration
         //Methods definition
}
```

Example Program:

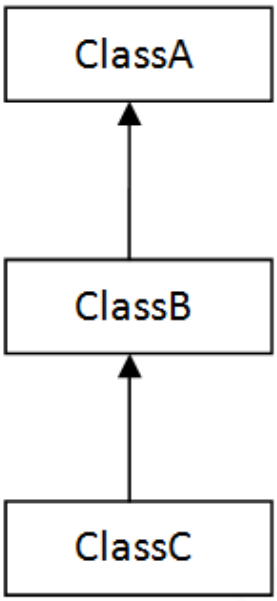
```
class Person{
          protected String name;
          protected String address;
          public Person(){
               System.out.println("From Person class constructor");
               name="Rajesh";
               address="Rajampet";
      }
          public void displayPerson(){
               System.out.println("Name="+name);
               System.out.println("Address="+address);
     }
}
```

```
class Student extends Person {
          private String roll;
          private String branch;
          public Student() {
               System.out.println("From Student  class constructor");
               roll="20701A0597";
               branch="CSE";
     }
    public void displayStudent() {
               displayPerson();
               System.out.println("Roll="+roll);
               System.out.println("Branch="+branch);
   }
}
class PerStu{
          public static void main(String args[]){
               Student s=new Student();
               s.displayStudent();
     }
}
/*
F:\BHAVAJAVA\UNIT-II>javac PerStu.java
F:\BHAVAJAVA\UNIT-II>java PerStu
From Person class constructor
From Student class constructor
Name=Rajesh
Address=Rajampet
Roll=20701A0597
Branch=CSE
*/
```

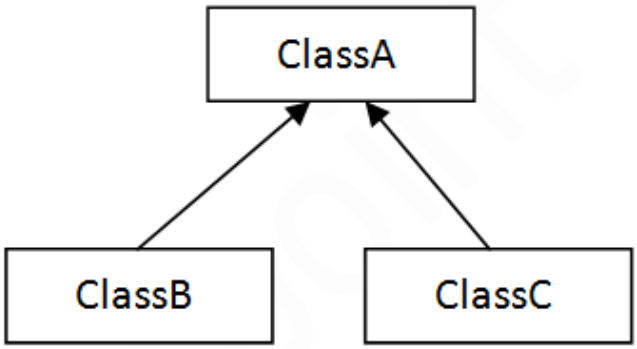
Different types of Inheritance supported in Java:



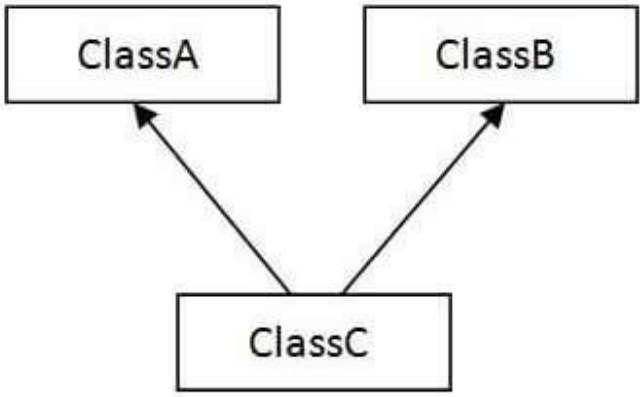
1) Single



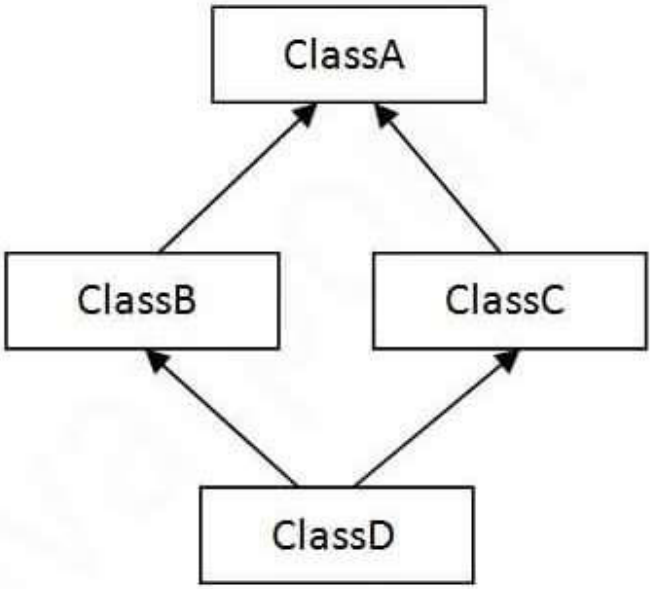
2) Multilevel



3) Hierarchical



4) Multiple



5) Hybrid

1. Single Inheritance

2. Multiple Inheritance (Java supported Multiple Inheritance with interfaces, not with classes)

3. Multi Level Inheritance

4. Hierarchal Inheritance

5. Hybrid Inheritance (Java supported Hybrid Inheritance with interfaces not with classes)

1. Single Inheritance:

```
     Acquiring properties from one super class to another class called
Single Inheritance.
//Single Inheritance
class Person{
     public void displayPerson() {
          System.out.println("I am Indian");
     }
}
class Programmer extends Person {
     public void displayProgrammer(){
          System.out.println("I am Computer Programmer");
     }
}
class SingleInheritance {
     public static void main(String args[]) {
          Programmer p=new Programmer();
          p.displayPerson();
          p.displayProgrammer();
     }
}

/*
F:\BHAVAJAVA\UNIT-II>javac SingleInheritance.java
F:\BHAVAJAVA\UNIT-II>java SingleInheritance
I am Indian
I am Computer Programmer
*/
```

2. Multi Level Inheritance:

Acquiring properties and functionality from one super class to another sub class and another sub class to another sub class in the chain manner called "Multi Level Inheritance".

```
//Multi Level Inheritance
class Person{
     public void displayPerson() {
          System.out.println("I am Indian");
     }
}
class Programmer extends Person {
     public void displayProgrammer(){
          System.out.println("I am Computer Programmer");
     }
}
class Engineer extends Programmer{
     public void displayEngineer(){
          System.out.println("I am Software Engineer");
     }
}
class MultiLevelInheritance {
     public static void main(String args[]) {
          Engineer p=new Engineer();
          p.displayPerson();
          p.displayProgrammer();
          p.displayEngineer();
     }
}

/*
F:\BHAVAJAVA\UNIT-II>javac MultiLevelInheritance.java
F:\BHAVAJAVA\UNIT-II>java MultiLevelInheritance
I am Indian
I am Computer Programmer
I am Software Engineer
*/
```

3. Hierarchical Inheritance:

When two or more classes inherited from a single class, it is known as hierarchical inheritance.

```
//Heirarichal Inheritance
class Person{
     public void displayPerson() {
          System.out.println("I am Indian");
     }
}
class Programmer extends Person {
     public void displayProgrammer(){
          System.out.println("I am Computer Programmer");
     }
}
class Tester extends Person{
     public void displayTester(){
          System.out.println("I am Computer Software Tester");
     }
}
class HeirarichalInheritance {
     public static void main(String args[]) {
          System.out.println("Calling Programmer object...");
          Programmer p=new Programmer();
          p.displayPerson();
          p.displayProgrammer();
          System.out.println("Calling Tester object...");
          Tester t=new Tester();
          t.displayPerson();
          t.displayTester();
     }
}

/*
F:\BHAVAJAVA\UNIT-II>javac HeirarichalInheritance.java
F:\BHAVAJAVA\UNIT-II>java HeirarichalInheritance
Calling Programmer object...
I am Indian
I am Computer Programmer
Calling Tester object...
I am Indian
I am Computer Software Tester
*/
```

Why multiple inheritance is not supported in java?

- To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So whether you have same method or different, there will be compile time error.

```
//Assumption Scenario
class A{
     void msg(){System.out.println("Hello");}
}
class B{
     void msg(){System.out.println("Welcome");}
}
class C extends A,B{//suppose if it were, but not possible practically
      public static void main(String args[]){
            C obj=new C();
            obj.msg();//Now which msg() method would be invoked?
     }
 }
```

Access Specifiers:

There are 4 different access specifiers supported in Java Programming.

i. private ii. no modifier iii. protected iv. public

Access Level	Private	No modifier or default	Protected	public
Same class	YES	YES	YES	YES
Sub class	NO	YES	YES	YES
Non sub class (through object)	NO	YES	YES	YES

```
//Access Specifiers
class Super{
      private int a=2;//private member
      int b=3;//no modifier member
      protected int c=4;//protected member
      public int d=5;//public member
      void displayPrivate(){//No Modifier Method
            System.out.println("Private Member accessed through nomodifier method="+a);
      }
}
class Sub extends Super{
      public void displayFromSuper(){//public method
            System.out.println("Access From Super class...............");
            System.out.println("Private from Super not accessed");
            displayPrivate();
            System.out.println("no modifier b="+b);
            System.out.println("protected  c="+c);
            System.out.println("public d="+d);
      }
}
class NonSub{
      public void displayFromSuper(){//public method
            Super s=new Super();
            System.out.println("Accessed from Super class in NonSub class
                              through object 's'");
            System.out.println("Private from Super not accessed");
            s.displayPrivate();
            System.out.println("Super members accessed through object");
            System.out.println("no modifier b="+s.b);
            System.out.println("protected  c="+s.c);
            System.out.println("public d="+s.d);
      }
}
class AccessSpecifier {
      public static void main(String args[]){
            Sub s=new Sub();
```

```
            s.displayFromSuper();
            NonSub ns=new NonSub();
            s.displayFromSuper();
      }
}

/*
F:\BHAVAJAVA\UNIT-II>javac AccessSpecifier.java

F:\BHAVAJAVA\UNIT-II>java AccessSpecifier
Access From Super class...............
Private from Super not accessed
Private Member accessed through nomodifier method=2
no modifer b=3
protecetd  c=4
public d=5
Accessed from Super class in NonSub class through object 's'
Private from Super not accessed
Private Member accessed through nomodifier method=2
Super members accessed through object
no modifer b=3
protecetd  c=4
public d=5
*/
```

A Superclass Variable Can Reference a Subclass Object:

- A reference variable of a superclass can be assigned an object of subclass derived from its superclass.
- A reference of super class can hold the object of sub class.
- All the overridden methods of a sub class can accessible through super class reference when sub class object is assigned to super class reference.
  "Super class Reference can refer sub class object."
- This is demonstrated in following example.

```
//Super Class Reference can hold sub class object
class GrandParent {
     //.....
}
class Parent extends GrandParent{
     .....
}
class Child extends Parent {
     .....
}
class SupRef {
     public static void main(String[] args) {
          //GrandParent refers GrandParent object and also refers Parent and Child object
          GrandParent g;
          g=new GrandParent();
          g=new Parent();
          g=new Child();
          //Parent refers Parent object and also refers Child object
          Parent p;
          p=new Parent();
          p=new Child();
     }
}
```

Method Overriding:

- In a class hierarchy, when a method in a subclass has the same name and type signature as a method in its superclass, then the method in the subclass is said to **override the method in the superclass**.
- When an overridden method is called from within its subclass, it will always refer to the version of that method defined by the subclass. **The version of the method defined by the superclass will be hidden.**
- Method overriding refines the functionality of super class method in sub class.

```
//MethodOverriding
//MethodOverride.java
//Demonstrates Multi Level Inheritance also
class Person{
      public void display(){
             System.out.println("I am Indian");
      }
}
class Programmer extends Person{
      //display() method overridden
      public void display(){
             System.out.println("I am Computer Programmer");
      }
}
class Engineer extends Programmer{
      //display() method overriden
      public void display(){
             System.out.println("I am Software Engineer");
      }
}
class MethodOverride {
      public static void main(String[] args){
             Person p;
             p=new Person();
             p.display();
             p=new Programmer();
             p.display();
             p=new Engineer();
             p.display();
               }
}
/*
F:\BHAVAJAVA\UNIT-II>javac MethodOverride.java

F:\BHAVAJAVA\UNIT-II>java MethodOverride
I am Indian
I am Computer Programmer
I am Software Engineer
*/
```

Using super keyword:

super keyword is used to

i. To call the immediate super class constructors in sub class.

Syntax:

super (arg_list)

ii. To call immediate super class members hidden by sub class.

Syntax:

super.member_name

iii. To call immediate super class methods hidden by sub class.

Syntax:

super.method_name (arg_list);

```
//Demonstration of super keyword to access super class constructor
// Demonstration of super keyword to access overriding methods
//PerStuSuper.java
class Person{
      protected String name;
      protected String address;
      public Person(String name,String address){
            this.name=name;
            this.address=address;
      }
      public void display(){
            System.out.println("Name="+name);
            System.out.println("Address="+address);
      }
}
class Student extends Person {
      private String roll;
      private String branch;
      public Student(String roll,String name,String branch,String address) {
            super(name,address);
            this.roll=roll;
            this.branch=branch;
      }
     public void display() {
```

```
            super.display();
            System.out.println("Roll Number="+roll);
            System.out.println("Branch="+branch);
      }
}
class PerStuSuper{
      public static void main(String args[]){
            Student s=new Student("20701A0599","Rajesh","CSE","Rajampet");
            s.display();
      }
}
/*
F:\BHAVAJAVA\UNIT-II>javac PerStuSuper.java

F:\BHAVAJAVA\UNIT-II>java PerStuSuper
Name=Rajesh
Address=Rajampet
Roll Number=20701A0599
Branch=CSE
*/
```

//Demonstration of super keyword to access super class members in sub class
//Demonstration for invoking super class constructor
//SuperMember.java

```
class Father {
      String hname;
      Father(String hname){
            this.hname=hname;
      }
}
class Son extends Father{
      String hname;
      Son(String fhname,String shname){
            super(fhname);
            this.hname=shname;
      }
      public void displayHouseName(){
```

```
            System.out.println("Father House Name="+super.hname);
            System.out.println("Son House Name="+hname);
        }
    }
    class SuperMember {
        public static void main(String args[]){
            Son s=new Son("Brindavanam","Nandanavanam");
            s.displayHouseName();
        }
    }

    /*
    F:\BHAVAJAVA\UNIT-II>javac SuperMember.java

    F:\BHAVAJAVA\UNIT-II>java SuperMember
    Father House Name=Brindavanam
    Son House Name=Nandanavanam
    */
```

The super keyword is demonstrated in following program.

```
//SuperDemo.java
class Box {
    private double width;
    private double height;
    private double depth;
    // constructor used when no dimensions specified
    Box() {
        width = -1; // use -1 to indicate
        height = -1; // an uninitialized
        depth = -1; // box
    }
    // constructor used when cube is created
    Box(double len) {
        width = height = depth = len;
    }
    // constructor used when all dimensions specified
```

```
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // construct clone of an object
    Box(Box ob) { // pass object to constructor
        width = ob.width;
        height = ob.height;
        depth = ob.depth;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
// BoxWeight now fully implements all constructors.
class BoxWeight extends Box {
    double weight; // weight of box
    // default constructor
    BoxWeight() {
        super();
        weight = -1;
    }
    // constructor used when cube is created
    BoxWeight(double len, double m) {
        super(len);
        weight = m;
    }
    // constructor when all parameters are specified
    BoxWeight(double w, double h, double d, double m) {
        super(w, h, d); // call superclass constructor
        weight = m;
    }
    // construct clone of an object
    BoxWeight(BoxWeight ob) { // pass object to constructor
```

```
        super(ob);
        weight = ob.weight;
    }

}
class SuperDemo {
    public static void main(String args[]) {
        BoxWeight mybox1 = new BoxWeight(10, 20, 15, 34.3);
        BoxWeight mybox2 = new BoxWeight(2, 3, 4, 0.076);
        BoxWeight mybox3 = new BoxWeight(); // default
        BoxWeight mycube = new BoxWeight(3, 2);
        BoxWeight myclone = new BoxWeight(mybox1);
        double vol;
        vol = mybox1.volume();
        System.out.println("Volume of mybox1 is " + vol);
        System.out.println("Weight of mybox1 is " + mybox1.weight);
        System.out.println();
        vol = mybox2.volume();
        System.out.println("Volume of mybox2 is " + vol);
        System.out.println("Weight of mybox2 is " + mybox2.weight);
        System.out.println();
        vol = mybox3.volume();
        System.out.println("Volume of mybox3 is " + vol);
        System.out.println("Weight of mybox3 is " + mybox3.weight);
        System.out.println();
        vol = myclone.volume();
        System.out.println("Volume of myclone is " + vol);
        System.out.println("Weight of myclone is " + myclone.weight);
        System.out.println();
        vol = mycube.volume();
        System.out.println("Volume of mycube is " + vol);
        System.out.println("Weight of mycube is " + mycube.weight);
        System.out.println();
    }
}
```

```
/*
F:\BHAVAJAVA\UNIT-II>javac SuperDemo.java

F:\BHAVAJAVA\UNIT-II>java SuperDemo
Volume of mybox1 is 3000.0
Weight of mybox1 is 34.3

Volume of mybox2 is 24.0
Weight of mybox2 is 0.076

Volume of mybox3 is -1.0
Weight of mybox3 is -1.0

Volume of myclone is 3000.0
Weight of myclone is 34.3

Volume of mycube is 27.0
Weight of mycube is 2.0

*/
```

Dynamic Method Dispatch:

- Dynamic method dispatch is the mechanism by **which a call to an overridden method is resolved at run time, rather than compile time**.
- Dynamic method dispatch is important because this is **how Java implements run-time polymorphism**.
- "A superclass reference variable can refer to a subclass object".
- **Java uses this fact to resolve calls to overridden methods at run time.**
- **When an overridden method is called through a superclass reference, Java determines which version of that method to execute based upon the type of the object being referred to at the time the call occurs.**
- Therefore, if a superclass contains a method that is overridden by a subclass, then when different types of objects are referred to through a superclass reference variable, different versions of the method are executed.
- This is demonstrated in following program.

```
class Figure {
     double dim1,dim2;
     Figure(){
          this.dim1=0;
          this.dim2=0;
     }
    Figure(double dim1,double dim2){
          this.dim1=dim1;
          this.dim2=dim2;
     }
    void display(){
          System.out.println("Dimensions are");
          System.out.println("Dim1="+dim1+"\t"+"Dim2="+dim2);
     }
     double area() {
          System.out.println("Figure Unknown");
          return 0;
     }
}
class Rectangle extends Figure{
     Rectangle(){
          super();
     }
    Rectangle(double dim1,double dim2){
          super(dim1,dim2);
     }
    double area() {
          double ar=dim1*dim2;
          return ar;
     }
}
class Traingle extends Figure {
     Traingle() {
          super();
     }
    Traingle(double dim1,double dim2){
```

```
            super(dim1,dim2);
        }
       double area() {
            double ar=0.5*dim1*dim2;
            return ar;
        }
}
class DynMethDispatch{
        public static void main(String[] args){
            Figure f;
            double ar;
            System.out.println("Figure.....");
            //Figure reference f now holding Figure class object.
            f=new Figure(2,5);
            f.display();
            ar=f.area();
            System.out.println("Rectangle.....");
            //Figure reference f now holding Rectangle class object.
            f=new Rectangle(5,7);
            f.display();
            ar=f.area();
            System.out.println("Area of Rectangle="+ar);
            System.out.println("Traingle.....");
            //Figure reference f now holding Traingle class object.
            f=new Traingle(8,9);
            f.display();
            ar=f.area();
            System.out.println("Area of Traingle ="+ar);
        }
}
/*
F:\BHAVAJAVA\UNIT-II>javac DynMethDispatch.java
F:\BHAVAJAVA\UNIT-II>java DynMethDispatch
Figure.....
Dimensions are
Dim1=2.0        Dim2=5.0
```

```
Figure Unknown
Rectangle.....
Dimensions are
Dim1=5.0        Dim2=7.0
Area of Rectangle=35.0
Traingle.....
Dimensions are
Dim1=8.0        Dim2=9.0
Area of Traingle =36.0
*/
```

Abstract Classes:

- There are situations in which you will want to define a superclass that declares the structure of a given abstraction without providing a complete implementation of every method.
- That is, sometimes you will want to create a superclass that only defines a generalized form that will be shared by all of its subclasses, leaving it to each subclass to fill in the details.
- Such a class determines the nature of the methods that the subclasses must implement.
- One way this situation can occur is when a superclass is unable to create a meaningful implementation for a method.
- Java's solution to this problem is the abstract methods and abstract classes.
- Syntax for abstract method:
  abstract return_type method_name(parameter-list);
- Any class that contains one or more abstract methods must also be declared abstract.
- To declare a class abstract, you simply use the abstract keyword in front of the class keyword at the beginning of the class declaration.
- There can be no objects of an abstract class. That is, **an abstract class cannot be directly instantiated with the new operator.**
- **You cannot declare abstract constructors, or abstract static methods.**
- Any subclass of an abstract class must either implement all of the abstract methods in the superclass, or be declared abstract itself.
- The syntax of abstract class demonstrated in following example:

```
// A Simple demonstration of abstract methods and abstract classes.
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();
     }
}
/*
B's implementation of callme.
This is a concrete method.
*/
```

Example Program:

```
//Abstract Methods and Abstract Classes
abstract class Figure {
     double dim1,dim2;
     Figure(){
          this.dim1=0;
          this.dim2=0;
     }
    Figure(double dim1,double dim2){
          this.dim1=dim1;
          this.dim2=dim2;
```

```
        }
    void display(){//Concrete method
            System.out.println("Dimensions are");
            System.out.println("Dim1="+dim1+"\t"+"Dim2="+dim2);
        }
        abstract double area(); //Abstract method
}
class Rectangle extends Figure{
        Rectangle(){
            super();
        }
    Rectangle(double dim1,double dim2){
            super(dim1,dim2);
        }
    double area() {
            double ar=dim1*dim2;
            return ar;
        }
}
class Traingle extends Figure {
        Traingle() {
            super();
        }
    Traingle(double dim1,double dim2){
            super(dim1,dim2);
        }
    double area() {
            double ar=0.5*dim1*dim2;
            return ar;
        }
}
class AbstractDemo{
        public static void main(String[] args){
            Figure f;
            double ar;
            System.out.println("Figure.....");
```

```
            //f=new Figure(2,5);
            //Object can n't instantiated for Figure Abstract class
            //f.display();
            //ar=f.area(); //Error
            System.out.println("Rectangle.....");
            f=new Rectangle(5,7);
            f.display();
            ar=f.area();
            System.out.println("Area of Rectangle="+ar);
            System.out.println("Traingle.....");
            f=new Traingle(8,9);
            f.display();
            ar=f.area();
            System.out.println("Area of Traingle ="+ar);
        }
}

/*
F:\BHAVAJAVA\UNIT-II>javac AbstractDemo.java

F:\BHAVAJAVA\UNIT-II>java AbstractDemo
Figure.....
Rectangle.....
Dimensions are
Dim1=5.0        Dim2=7.0
Area of Rectangle=35.0
Traingle.....
Dimensions are
Dim1=8.0        Dim2=9.0
Area of Traingle =36.0
*/
```

Constructors Execution Order:

- When an object is created for sub class, constructors are executed from super class to sub class.

```
//MethodOverriding
//MethodOverride.java
//Multi Level Inheritance
class Person{
      Person(){
            System.out.println("From Person class Constructor.........");
            System.out.println("I am Indian");
      }
}
class Programmer extends Person{
      Programmer(){
            System.out.println("From Programmer class Constructor.........");
            System.out.println("I am Computer Programmer");
      }
}
class Engineer extends Programmer{
      Engineer(){
            System.out.println("From Engineer class Constructor.........");
            System.out.println("I am Software Engineer");
      }
}
class ConstructorOrder {
      public static void main(String[] args){
            new Engineer();
      }
}
/*
F:\BHAVAJAVA\UNIT-II>javac ConstructorOrder.java
F:\BHAVAJAVA\UNIT-II>java ConstructorOrder
From Person class Constructor.........
I am Indian
From Programmer class Constructor.........
I am Computer Programmer
From Engineer class Constructor.........
I am Software Engineer
*/
```

"final" Keyword:

"final" keyword is used in three different situations.

1. To fix the variable value as constant.
2. To prevent method to be overridden in sub class.
3. To prevent the class to be inherited in sub class.

1. To fix the variable value as constant:

Syntax:

final [access_specifier] data_type var_name=value;

Ex-: final public int a=2;

public final String college="AITS";

Note: final and public keywords interchangeably we can use in syntax.

2. To prevent method to be overridden in sub class:

- To disallow a method from being overridden, specify final as a modifier at the start of its declaration.
- Methods declared as final cannot be overridden.
- The following fragment illustrates final:

//FinalMethod.java

```
class Parent{
   final void display(){ //Error! Not Executed
        System.out.println("This method not overriden");
   }
}
class Child extends Parent{
   void display(){
        System.out.println("From Child class");
   }
}
class FinalMethod {
   public static void main(String[] args){
        Parent p;
        p=new Parent();
        p.display();
        p=new Child();
        p.display();
   }
}
```

```
/*
F:\BHAVAJAVA\UNIT-II>javac FinalMethod.java
FinalMethod.java:7: error: display() in Child cannot override
display() in Parent
        void display(){
             ^
  overridden method is final
1 error
*/
```

Note:

- Methods declared as final can sometimes provide a performance enhancement: The compiler is free to inline calls to them because it "knows" they will not be overridden by a subclass.
- When a small final method is called, **often the Java compiler can copy the bytecode for the subroutine directly inline with the compiled code of the calling method**, thus eliminating the costly overhead associated with a method call. Inlining is an option only with final methods.
- Normally, Java resolves calls to methods dynamically, at run time. This is called late binding. However, since final methods cannot be overridden, a call to one can be resolved at compile time. This is called early binding

3. To prevent the class to be inherited in sub class:

- Sometimes you will want to prevent a class from being inherited. To do this, precede the class declaration with final.
- Declaring a class as final implicitly declares all of its methods as final, too.
- You can n't declare **abstract classes as final classes**. Because abstract classes let completely implemented in derived classes.

```
//FinalClass.java
final class A{
    void display(){
        System.out.println("This class not Inherited");
  }
}
class B extends A{
  void display(){
        System.out.println("From Child class");
  }
}
class FinalClass {
  public static void main(String[] args){
```

```
        Parent p;
        p=new Parent();
        p.display();
        p=new Child();
        p.display();
    }
}

/*
F:\BHAVAJAVA\UNIT-II>javac FinalClass.java
FinalClass.java:6: error: cannot inherit from final A
class B extends A{
                ^
FinalClass.java:16: error: cannot find symbol
                p=new Child();
                      ^
  symbol:   class Child
  location: class FinalClass
2 errors
*/
```

The Object class:

- There is one special class, Object, defined by Java.
- All other classes are subclasses of Object. That is, **Object is a superclass of all other classes.**
- This means that a reference variable of type Object can refer to an object of any other class.

Method
public final Class getClass() returns the Class class object of this object. The **Class** class can further be used to get the metadata of this class.
public int hashCode() returns the hashcode number for this object.
public boolean equals(Object obj) compares the given object to this object.
protected Object clone() throws CloneNotSupportedException creates and returns the exact copy (clone) of this object.
public String toString() returns the string representation of this object.

public final void notify() wakes up single thread, waiting on this object's monitor.
public final void notifyAll() wakes up all the threads, waiting on this object's monitor.
public final void wait(long timeout) throws InterruptedException causes the current thread to wait for the specified milliseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait(long timeout,int nanos) throws InterruptedException causes the current thread to wait for the specified milliseconds and nanoseconds, until another thread notifies (invokes notify() or notifyAll() method).
public final void wait() throws InterruptedException causes the current thread to wait, until another thread notifies (invokes notify() or notifyAll() method).
protected void finalize() throws Throwable is invoked by the garbage collector before object is being garbage collected.

```
//ObjectEx.java
class Sample {
  void display() {
       System.out.println("From Sample class");
  }
    public String toString() {
       return "AITS is a Great College";
  }
}
class ObjectEx{
  public static void main(String[] args){
       Sample s=new Sample();
       System.out.println("Sample Object 's' hashcode="+s.hashCode());
       System.out.println("S object described="+s);
  }
}
/*
F:\BHAVAJAVA\UNIT-II>javac ObjectEx.java
F:\BHAVAJAVA\UNIT-II>java ObjectEx
Sample Object 's' hashcode=2018699554
S object described=AITS is a Great College
*/
```

String class

String is a sequence of characters. In java, objects of String are immutable which means a constant and cannot be changed once created.

Creating a String
There are two ways to create string in Java:

- ***String literal***
  String s = "GeeksforGeeks";
- **Using *new* keyword**
  String s = new String ("GeeksforGeeks");

Constructors

1. **String(byte[] byte_arr)** – Construct a new String by decoding the *byte array*. It uses the platform's default character set for decoding.
   Example:
   byte[] b_arr = {71, 101, 101, 107, 115};

   String s_byte =new String(b_arr); //Geeks
2. **String(byte[] byte_arr, Charset char_set)** – Construct a new String by decoding the *byte array*. It uses the *char_set* for decoding.
   Example:
   byte[] b_arr = {71, 101, 101, 107, 115};

   Charset cs = Charset.defaultCharset();

   String s_byte_char = new String(b_arr, cs); //Geeks
3. **String(byte[] byte_arr, String char_set_name)** – Construct a new String by decoding the *byte array*. It uses the *char_set_name* for decoding. It looks similar to the above constructs and they appear before similar functions but it takes the *String(which contains char_set_name)* as parameter while the above constructor takes *CharSet.*
   Example:
   byte[] b_arr = {71, 101, 101, 107, 115};

   String s = new String(b_arr, "US-ASCII"); //Geeks
4. **String(byte[] byte_arr, int start_index, int length)** – Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location).*
   Example:
   byte[] b_arr = {71, 101, 101, 107, 115};

   String s = new String(b_arr, 1, 3); // eek
5. **String(byte[] byte_arr, int start_index, int length, Charset char_set)** – Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.Uses *char_set* for

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

decoding.
Example:
```
byte[] b_arr = {71, 101, 101, 107, 115};

Charset cs = Charset.defaultCharset();

String s = new String(b_arr, 1, 3, cs); // eek
```

6. **String(byte[] byte_arr, int start_index, int length, String char_set_name)** – Construct a new string from the *bytes array* depending on the *start_index(Starting location)* and *length(number of characters from starting location)*.Uses *char_set_name* for decoding.
Example:
```
byte[] b_arr = {71, 101, 101, 107, 115};

String s = new String(b_arr, 1, 4, "US-ASCII"); // eeks
```
7. **String(char[] char_arr)** – Allocates a new String from the given *Character array*
Example:
```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};

String s = new String(char_arr); //Geeks
```
8. **String(char[] char_array, int start_index, int count)** – Allocates a String from a given *character array* but choose *count* characters from the *start_index*.
Example:
```
char char_arr[] = {'G', 'e', 'e', 'k', 's'};

String s = new String(char_arr , 1, 3); //eek
```
9. **String(int[] uni_code_points, int offset, int count)** – Allocates a String from a *uni_code_array* but choose *count* characters from the *start_index*.
Example:
```
int[] uni_code = {71, 101, 101, 107, 115};

String s = new String(uni_code, 1, 3); //eek
```
10. **String(StringBuffer s_buffer)** – Allocates a new string from the string in *s_buffer*
Example:
```
StringBuffer s_buffer = new StringBuffer("Geeks");

String s = new String(s_buffer); //Geeks
```
11. **String(StringBuilder s_builder)** – Allocates a new string from the string in *s_builder*
Example:
```
StringBuilder s_builder = new StringBuilder("Geeks");

String s = new String(s_builder); //Geeks
```

<u>String Methods</u>

1. **int length():** Returns the number of characters in the String.

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

"GeeksforGeeks".length(); // returns 13

2. Char charAt(int i)**:** Returns the character at i^{th} index.
   "GeeksforGeeks".charAt(3); // returns 'k'
3. String substring (int i)**:** Return the substring from the ith index character to end.
   "GeeksforGeeks".substring(3); // returns "ksforGeeks"
4. String substring (int i, int j)**:** Returns the substring from i to j-1 index.
   "GeeksforGeeks".substring(2, 5); // returns "eks"
5. String concat(String str)**:** Concatenates specified string to the end of this string.
   String s1 = "Geeks";

   String s2 = "forGeeks";

   String output = s1.concat(s2); // returns "GeeksforGeeks"
6. int indexOf (String s)**:** Returns the index within the string of the first occurrence of the specified string.
   String s = "Learn Share Learn";

   int output = s.indexOf("Share"); // returns 6
7. int indexOf (String s, int i)**:** Returns the index within the string of the first occurrence of the specified string, starting at the specified index.
   String s = "Learn Share Learn";

   int output = s.indexOf("ea",3);// returns 13
8. Int lastIndexOf(String s)**:** Returns the index within the string of the last occurrence of the specified string.
   String s = "Learn Share Learn";

   int output = s.lastIndexOf("a"); // returns 14
9. **boolean equals(Object otherObj):** Compares this string to the specified object.
   Boolean out = "Geeks".equals("Geeks"); // returns true

   Boolean out = "Geeks".equals("geeks"); // returns false
10. boolean equalsIgnoreCase (String anotherString)**:** Compares string to another string, ignoring case considerations.
    Boolean out= "Geeks".equalsIgnoreCase("Geeks"); // returns true

    Boolean out = "Geeks".equalsIgnoreCase("geeks"); // returns true
11. int compareTo(String anotherString)**:** Compares two string lexicographically.
    int out = s1.compareTo(s2); // where s1 and s2 are

    // strings to be compared

    This returns difference s1-s2. If :

    out < 0 // s1 comes before s2

    out = 0 // s1 and s2 are equal.

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

out > 0 // s1 comes after s2.

12. **int compareToIgnoreCase(String anotherString):** Compares two string lexicographically, ignoring case considerations.
int out = s1.compareToIgnoreCase(s2);

// where s1 and s2 are

// strings to be compared

This returns difference s1-s2. If :

out < 0 // s1 comes before s2

out = 0 // s1 and s2 are equal.

out > 0 // s1 comes after s2.

Note- In this case, it will not consider case of a letter (it will ignore whether it is uppercase or lowercase).

13. String toLowerCase()**:** Converts all the characters in the String to lower case.
String word1 = "HeLLo";

String word3 = word1.toLowerCase(); // returns "hello"

14. String toUpperCase()**:** Converts all the characters in the String to upper case.
String word1 = "HeLLo";

String word2 = word1.toUpperCase(); // returns "HELLO"

15. String trim()**:** Returns the copy of the String, by removing whitespaces at both ends. It does not affect whitespaces in the middle.
String word1 = " Learn Share Learn ";

String word2 = word1.trim(); // returns "Learn Share Learn"

16. String replace (char oldChar, char newChar)**:** Returns new string by replacing all occurrences of *oldChar* with *newChar.*
String s1 = "feeksforfeeks";

String s2 = "feeksforfeeks".replace('f' ,'g'); // returns "geeksgorgeeks"

Note:- s1 is still feeksforfeeks and s2 is geeksgorgeeks

Program to illustrate all string methods:

```
// Java code to illustrate different constructors and methods
// String class.
 import java.io.*;
import java.util.*;
class Test
{
   public static void main (String[] args)
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
{
    String s= "GeeksforGeeks";
    // or String s= new String ("GeeksforGeeks");
      // Returns the number of characters in the String.
    System.out.println("String length = " + s.length());
      // Returns the character at ith index.
    System.out.println("Character at 3rd position = "+ s.charAt(3));
      // Return the substring from the ith  index character to end of string
    System.out.println("Substring " + s.substring(3));
      // Returns the substring from i to j-1 index.
    System.out.println("Substring  = " + s.substring(2,5));
      // Concatenates string2 to the end of string1.
    String s1 = "Geeks";
    String s2 = "forGeeks";
    System.out.println("Concatenated string  = " +s1.concat(s2));
      // Returns the index within the string of the first occurrence of the specified string.
    String s4 = "Learn Share Learn";
    System.out.println("Index of Share " +s4.indexOf("Share"));
// Returns the index within the string of the  first occurrence of the specified string, starting at the specified index.
    System.out.println("Index of a  = " +s4.indexOf('a',3));
      // Checking equality of Strings
    Boolean out = "Geeks".equals("geeks");
    System.out.println("Checking Equality  " + out);
    out = "Geeks".equals("Geeks");
    System.out.println("Checking Equality  " + out);
      out = "Geeks".equalsIgnoreCase("gEeks ");
    System.out.println("Checking Equality " + out);
      //If ASCII difference is zero then the two strings are similar
    int out1 = s1.compareTo(s2);
    System.out.println("the difference between ASCII value is="+out1);
    // Converting cases
    String word1 = "GeeKyMe";
    System.out.println("Changing to lower Case " +word1.toLowerCase());
      // Converting cases
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
        String word2 = "GeekyME";
        System.out.println("Changing to UPPER Case " +word2.toUpperCase());
          // Trimming the word
        String word4 = " Learn Share Learn ";
        System.out.println("Trim the word " + word4.trim());
          // Replacing characters
        String str1 = "feeksforfeeks";
        System.out.println("Original String " + str1);
        String str2 = "feeksforfeeks".replace('f' ,'g') ;
        System.out.println("Replaced f with g -> " + str2);
    }
}
```

Output :

String length = 13

Character at 3rd position = k

Substring ksforGeeks

Substring = eks

Concatenated string = GeeksforGeeks

Index of Share 6

Index of a = 8

Checking Equality false

Checking Equality true

Checking Equality false

the difference between ASCII value is=-31

Changing to lower Case geekyme

Changing to UPPER Case GEEKYME

Trim the word Learn Share Learn

Original String feeksforfeeks

Replaced f with g -> geeksgorgeeks

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

This Key Word:

In Java, the this keyword is a reference to the current object — the instance of the class in which it is used. It is primarily used to differentiate between class attributes (fields) and parameters with the same name, but it has other useful applications as well. Let's explore the common use cases of this:

1. Referring to Instance Variables

If a constructor or method parameter has the same name as an instance variable, this is used to differentiate between them.

```
class Person {
    String name;

    Person(String name) {
        this.name = name;  // 'this.name' refers to the instance variable, 'name' refers to the parameter
    }

    void display() {
        System.out.println("Name: " + this.name);  // 'this.name' refers to the instance variable
    }
}
```

2. Invoking a Constructor from Another Constructor

The this() keyword can be used to call one constructor from another within the same class. This is known as constructor chaining.

```
class Person {
    String name;
    int age;

    Person(String name) {
        this(name, 0);  // Calls the other constructor with the 'age' parameter
    }

    Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void display() {
        System.out.println("Name: " + name + ", Age: " + age);
    }
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
}
```

3. Passing the Current Object as an Argument

You can use this to pass the current object as a parameter to another method.

```
class Person {
    String name;

    Person(String name) {
        this.name = name;
    }

    void display() {
        System.out.println("Name: " + name);
    }

    void callDisplay() {
        helperMethod(this);  // 'this' passes the current object to the helperMethod
    }

    void helperMethod(Person p) {
        p.display();  // Calls the display method on the current object
    }
}
```

4. Returning the Current Object

You can use this to return the current object from a method. This is often used in method chaining (e.g., in builder patterns).

```
class Person {
    String name;

    Person setName(String name) {
        this.name = name;
        return this;  // Returns the current object
    }

    void display() {
        System.out.println("Name: " + name);
    }
}

class Main {
    public static void main(String[] args) {
        Person p = new Person().setName("Alice");
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
        p.display();  // Name: Alice
    }
}
```

5. Calling Instance Methods

You can use this to call other instance methods within the same class. It's optional, as the method call can be implied, but it can sometimes be useful for clarity.

```
class Person {
    String name;

    void setName(String name) {
        this.name = name;
    }

    void printName() {
        this.display();  // 'this' explicitly refers to the current object
    }

    void display() {
        System.out.println("Name: " + this.name);
    }
}
```

Summary:

- this refers to the **current object** in a class.
- It is used to **differentiate** between instance variables and parameters.
- Can be used to **invoke** constructors, return the current object, or pass the current object as a method argument.

Final Key Word:

In Java, the final keyword can be applied to classes, methods, and variables. It is used to restrict modification in various ways.

1. final Classes:

When a class is marked as final, it cannot be subclassed (inherited). This is useful when you want to prevent inheritance to ensure the behavior of the class remains consistent and unmodified.

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
final class FinalClass {
    // class body
    public void display() {
        System.out.println("This is a final class.");
    }
}

// Example of attempted inheritance (will cause a compilation error)
class SubClass extends FinalClass {
    // Compilation error: cannot inherit from final class FinalClass
}
```

2. final Methods:

When a method is declared as final, it cannot be overridden by subclasses. This ensures that the behavior of the method remains the same in all subclasses.

```
class ParentClass {
    public final void display() {
        System.out.println("This is a final method.");
    }
}

class ChildClass extends ParentClass {
    // Compilation error: cannot override final method from ParentClass
    public void display() {
        System.out.println("Trying to override a final method.");
    }
}
```

3. final Variables:

When a variable is declared as final, its value cannot be changed once it has been initialized. This is particularly useful for constants or variables that should not change after their initial assignment.

- **Final primitive variables:** Their value cannot be changed after initialization.

```
final int MAX_VALUE = 100;
MAX_VALUE = 200;  // Compilation error: cannot assign a value to final variable MAX_VALUE
```

- **Final reference variables:** The reference cannot be changed, but the object itself can still be modified.

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
final Person person = new Person("John");
person = new Person("Jane");  // Compilation error: cannot assign a new reference to final variable 'person'

// But you can modify the object's internal state
person.setName("Jane");  // This is allowed
```

Summary:

- **final class**: Cannot be subclassed.
- **final method**: Cannot be overridden.
- **final variable**: Once assigned, cannot be reassigned.

This is useful when you want to create immutable objects, prevent method overriding, or enforce certain behaviors in your code.

Static Key Word:

In Java, the static keyword is used to indicate that a member (variable, method, block, or nested class) belongs to the **class** rather than instances of the class. This means that static members are shared across all instances of the class and can be accessed without creating an object of the class.

Here's how static is used in various contexts:

1. Static Variables

A static variable is shared by all instances of a class. It is initialized only once when the class is loaded into memory. Since it belongs to the class rather than any instance, it can be accessed using the class name.

Example:

```
class Counter {
    static int count = 0;  // static variable

    Counter() {
        count++;  // increments the static variable
    }

    void displayCount() {
        System.out.println("Count: " + count);
    }
}

public class Main {
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        c1.displayCount();  // Output: Count: 2
        c2.displayCount();  // Output: Count: 2
    }
}
```

Here, the count variable is shared by all instances of the Counter class.

2. Static Methods

A static method belongs to the class and can be called without creating an object. It can access only other static members (variables and methods) of the class and cannot access instance variables or methods directly.

Example:

```
class MathUtility {
    static int add(int a, int b) {
        return a + b;  // static method
    }
}

public class Main {
    public static void main(String[] args) {
        int sum = MathUtility.add(10, 20);  // Calling static method without creating an object
        System.out.println("Sum: " + sum);  // Output: Sum: 30
    }
}
```

In this example, the add() method is static, so it can be called directly using the class name.

3. Static Blocks

A static block is used to initialize static variables. It is executed when the class is loaded, before any objects are created or any static methods are called.

Example:

```
class StaticBlockExample {
    static int value;

    // Static block
    static {
        value = 10;
        System.out.println("Static block executed.");
```


Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

```
    }

    static void displayValue() {
        System.out.println("Value: " + value);
    }
}

public class Main {
    public static void main(String[] args) {
        StaticBlockExample.displayValue();  // Output: Static block executed. Value: 10
    }
}
```

In this example, the static block initializes the value and is executed only once when the class is loaded.

4. Static Nested Classes

A nested class can be declared as static. Unlike non-static nested classes (inner classes), a static nested class does not need a reference to an instance of the enclosing class. It can only access the static members of the outer class.

Example:

java
Copy code

```
class OuterClass {
    static class NestedClass {
        void display() {
            System.out.println("This is a static nested class.");
        }
    }
}

public class Main {
    public static void main(String[] args) {
        OuterClass.NestedClass nested = new OuterClass.NestedClass();  // Create an object of static nested class
        nested.display();  // Output: This is a static nested class.
    }
}
```

Here, the NestedClass is static and can be instantiated without needing an instance of OuterClass.

Summary:

- **Static variables**: Shared across all instances, accessed using the class name.

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

- **Static methods**: Can be called without an object, can only access static members.
- **Static blocks**: Used for static initialization, run when the class is loaded.
- **Static nested classes**: Do not require an instance of the outer class.

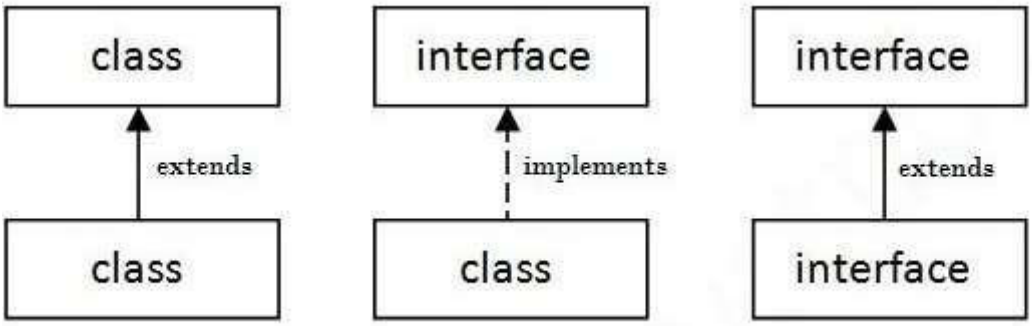
This allows static members to be used efficiently for common, shared logic across a class without needing multiple instances.

Prepared by A Ramesh Babu, Assistant Professor, Dept. AI&DS, Annamacharya University, Rajampet

INTERFACE

INTERFACES:

- Interfaces support pure abstract classes.
- Interfaces are defined using "interface" keyword.
- Interfaces specify the responsibilities, which are implemented by classes.
- **Java allows you to fully utilize the "one interface, multiple methods aspect of polymorphism".**
- One class can implement any number of interfaces using **implements** keyword. Through this way we can implement multiple inheritance in java.
- Interfaces contain instant variables, but they are static and final by default.
- **Interface reference can hold the reference of implemented class object. In this way we can achieve dynamic method dispatch using interfaces.**
- All interface methods are implemented in implemented classes as **public methods.**
- The relationship between classes and interfaces defined as, a class extends another class, an interface extends another interface, but a class implements an interface.



Syntax:

1. Defining Interfaces:

```
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;
}
```

2. Implementing Interfaces:

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

}
```

EX:

JAVA UNIT-III

```
interface Shapes{
   double pi=3.14;
   double areaCircle(int r);
   double areaRectangle(int l,int b);
}
class ShapesImpl implements Shapes{
   public double areaCircle(int r){
        return pi*r*r;
   }
     public double areaRectangle(int l,int b){
         return l*b;
    }
}

class ShapesDemo{
   public static void main(String args[]){
        Shapes s=new ShapesImpl();
        double ac=s.areaCircle(3);
        System.out.println("Area of Circle="+ac);
        double ar=s.areaRectangle(5,4);
        System.out.println("Area of rectangle="+ar);
   }
}
/*
F:\BHAVAJAVA\UNIT-II>javac ShapesDemo.java
F:\BHAVAJAVA\UNIT-II>java ShapesDemo
Area of Circle=28.259999999999998
Area of rectangle=20.0
*/
```

Note: In above program Shapes interface reference "s" holding ShapeImpl class object. Through interface reference "s" we invoked the methods of implemented class.

How can we implement interfaces partially?

If a class includes an interface but does not fully implement the methods required by that interface, then that class must be declared as abstract.

Ex:

```
interface Sample{
     void abstractMethod();
     void concreteMethod();
}
abstract class AbstractSample implements Sample{
     public void concreteMethod(){
          System.out.println("Interface Concrete Method Implemented
                                 in AbstractSample class");
     }
    public abstract void abstractMethod();
}
class ExampleSample extends AbstractSample{
     public void abstractMethod(){
          System.out.println("Interface Abtstact Method implemented
                                in ExampleSample class");
     }
}
class InterAbstract{
     public static void main(String[] args){
          Sample s=new ExampleSample();
          s.concreteMethod();
          s.abstractMethod();
     }
}
/*
F:\BHAVAJAVA\UNIT-II>javac InterAbstract.java
F:\BHAVAJAVA\UNIT-II>java InterAbstract
Interface Concrete Method Implemented in AbstractSample class
Interface Abtstact Method implemented in ExampleSample class
*/
```

Dynamic Method Dispatch with Interfaces:

- Dynamic method dispatch is the mechanism by **which a call to an overridden method is resolved at run time, rather than compile time**.
- Dynamic method dispatch can be implemented through interfaces also.
- An Interface reference variable can refer the appropriate overridden method of implemented classes at run time.

```
//Interfaces, Abstract Methods and Abstract Classes
interface Figure{
     void display();
     double area();
}
abstract class AbstractFigure implements Figure{
     double dim1,dim2;
     AbstractFigure(){
          this.dim1=0;
          this.dim2=0;
     }
    AbstractFigure(double dim1,double dim2){
          this.dim1=dim1;
          this.dim2=dim2;
     }
    public void display(){//Concrete method
          System.out.println("Dimensions are");
          System.out.println("Dim1="+dim1+"\t"+"Dim2="+dim2);
     }
     public abstract double area(); //Abstract method
}
class RectangleFigure extends AbstractFigure{
     RectangleFigure(){
          super();
     }
    RectangleFigure(double dim1,double dim2){
          super(dim1,dim2);
     }
    public double area() {
          double ar=dim1*dim2;
          return ar;
     }
}
class TraingleFigure extends AbstractFigure {
     TraingleFigure() {
          super();
     }
    TraingleFigure(double dim1,double dim2){
          super(dim1,dim2);
```

```
     }
    public double area() {
          double ar=0.5*dim1*dim2;
          return ar;
     }
}
class InterfaceDemo{
     public static void main(String[] args){
          Figure f;
          double ar;
          System.out.println("Figure. ....");
          pSystem.out.println("RectangleFigure. ....");
          f=new RectangleFigure(5,7);
          f.display();
          ar=f.area();
          System.out.println("Area of RectangleFigure="+ar);
          System.out.println("TraingleFigure. ....");
          f=new TraingleFigure(8,9);
          f.display();
          ar=f.area();
          System.out.println("Area of TraingleFigure ="+ar);
     }
}

/*
F:\BHAVAJAVA\UNIT-II>javac InterfaceDemo.java

F:\BHAVAJAVA\UNIT-II>java InterfaceDemo
Figure.....
RectangleFigure.....
Dimensions are
Dim1=5.0        Dim2=7.0
Area of RectangleFigure=35.0
TraingleFigure.....
Dimensions are
Dim1=8.0        Dim2=9.0
Area of TraingleFigure =36.0
*/
```

Nested Interfaces

- An interface can be declared a member of a class or another interface. Such an interface is called a member interface or a nested interface.
- Here is an example that demonstrates a nested interface:

```
// A nested interface example.
// This class contains a member interface.
class A {
     // this is a nested interface
     public interface NestedIF {
     boolean isNotNegative(int x);
}
}
// B implements the nested interface.
class B implements A.NestedIF {
     public boolean isNotNegative(int x) {
     return x < 0 ? false: true;
     }
}
class NestedIFDemo {
     public static void main(String args[]) {
     // use a nested interface reference
     A.NestedIF nif = new B();
     if(nif.isNotNegative(10))
          System.out.println("10 is not negative");
     if(nif.isNotNegative(-12))
          System.out.println("this won't be displayed");
     }
}

/*
F:\BHAVAJAVA\UNIT-II>javac  NestedIFDemo.java

F:\BHAVAJAVA\UNIT-II>java NestedIFDemo
10 is not negative
*/
```

Interfaces Can Be Extended

- 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 required by the interface Inheritance chain.

```
//ExtendInterface.java
interface A{
     void methodA1();
     void methodA2();
}
interface B extends A{
     void methodB();
}
class ABImpl implements A,B{
     public void methodA1(){
          System.out.println("From methodA1-Interface A");
     }
     public void methodA2(){
          System.out.println("From methodA2-Interface A");
     }
     public void methodB(){
          System.out.println("From methodB-Interface B");
     }
}
class ExtendInterface{
     public static void main(String[] args) {
          B bobj=new ABImpl();
          bobj.methodA1();
          bobj.methodA2();
        bobj.methodB();
     }
}
/*
F:\BHAVAJAVA\UNIT-II>javac ExtendInterface.java
F:\BHAVAJAVA\UNIT-II>java ExtendInterface
From methodA1-Interface A
From methodA2-Interface A
From methodB-Interface B        */
```

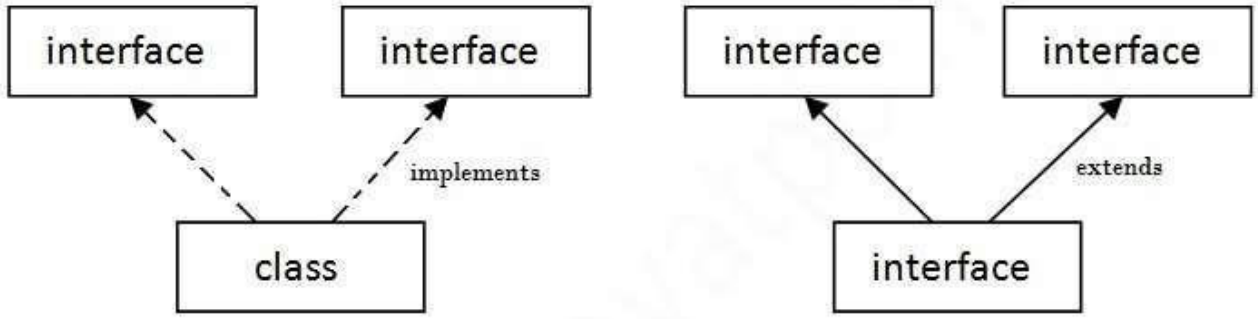
Variables in Interfaces:

- You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values.
- When you include that interface in a class (that is, when you "implement" the interface), all of those variable names will be in scope as constants.
- This is similar to using a header file in C/C++ to create a large number of #defined constants or const declarations.
- If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is as if that class were importing the constant fields into the class name space as final variables.
- This is demonstrated in following program.

```
//BoundaryEx.java
interface Boundary{
     int UPPER_LIMIT=100;
     int LOWER_LIMIT=0;
}
class BoundaryEx{
     public static void main(String args[]){
          System.out.println("Upper Limit="+Boundary.UPPER_LIMIT);
          System.out.println("Lower Limit="+Boundary.LOWER_LIMIT);
     }
}

/*
F:\BHAVAJAVA\UNIT-II>javac BoundaryEx.java
F:\BHAVAJAVA\UNIT-II>java BoundaryEx
Upper Limit=100
Lower Limit=0
*/
```

Multiple Inheritance using interfaces

JAVA UNIT-III



Multiple Inheritance in Java

```
//Multiple Inheritance
interface Programmer{
       void displayP();
}
interface Engineer{
       void displayE();
}
interface Innovator{
       void displayI();
}
class Student implements Programmer,Engineer,Innovator{
       public void displayP(){
          System.out.println("I am Good Computer Programmer");
       }
       public void displayE(){
          System.out.println("I can built Software Product");
       }
       public void displayI(){
          System.out.println("I have  Innovative Skills");
       }
       public void displayCEO(){
          System.out.println("I can start My own Startup");
       }
}
class MultipleSkills{
       public static void main(String args[]){
          Student s=new Student();
          s.displayP();
          s.displayE();
          s.displayI();
          s.displayCEO();
       }
}
```

Default Methods in Interfaces(JDK 8 feature): JAVA UNIT-III

- The release of JDK 8 has changed this by adding a new capability to interface called the default method.
- A default method lets you define a default implementation for an interface method.
- In other words, by use of a default method, it is possible for an interface method to provide a body, rather than being abstract.
- During its development, the default method was also referred to as an extension method.

Syntax:

```
interface interface_name{
        // declaration of interface variables
        //declaration of interface methods
        //definition of default methods
        default return_type method_name(arg_list){
                ...........
        }
        ....................
}
```

```
//DefaultInterMethod.java
interface Arithemetic{
     int add(int a,int b);
     int sub(int a,int b);
     int mul(int a,int b);
     float div(int a,int b);
     //default method
     default void show(){
          System.out.println("Default Method from Arithemetic Interface");
     }
}
class ArithemeticImpl implements Arithemetic{
     public int add(int a,int b){
          return (a+b);
     }
     public int sub(int a,int b){
          return (a-b);
     }
    public int mul(int a,int b){
          return (a*b);
     }
    public float div(int a,int b){
          return (float)a/b;
     }
}
```

```
class DefaultInterMethod{
     public static void main(String[] args){
          Arithemetic ar=new ArithemeticImpl();
          //Calling Default method
          ar.show();
          System.out.println("Addition="+ar.add(2,5));
          System.out.println("Subtraction="+ar.sub(5,2));
          System.out.println("Multiplication="+ar.mul(9,9));
          System.out.println("Division="+ar.div(9,7));
          }
}
/*
F:\BHAVAJAVA\UNIT-II>javac DefaultInterMethod.java

F:\BHAVAJAVA\UNIT-II>java DefaultInterMethod
Default Method from Arithemetic Interface
Addition=7
Subtraction=3
Multiplication=81
Division=1.2857143
*/
```

Use static Methods in an Interface

- JDK 8 added another new capability to interface: the ability to define one or more static methods.
- Syntax:

```
interface interface_name{
     // declaration of interface variables
     //declaration of interface methods
     //definition of default methods
     default return_type method_name(arg_list){
              ...........
     }
     ....................
     //definition of static methods
     static return_type method_name(arg_list){
              ...........
     }
     ....................

}
```

- Like static methods in a class, a static method defined by an interface can be called independently of any object.
- Thus, no implementation of the interface is necessary, and no instance of the interface is required, in order to call a static method.

- Instead, a static method is called by specifying the interface name, followed by a period, followed by the method name.
- Here is the general form:

InterfaceName.staticMethodName

Example Program:

```
//StaticInterfaceMethod.java
interface Sample{
 //static method
 static void staticMethod(){
      System.out.println("We are AITS students");
 }
 String getMyBranch();
}
class SampleImpl implements Sample{
 public String getMyBranch(){
      return "CSE";
 }
}
class StaticInterfaceMethod{
 public static void main(String args[]){
      Sample s=new SampleImpl();
      //Call static method
      Sample.staticMethod();
      System.out.println("My Branch="+s.getMyBranch());
 }
}

/*
F:\BHAVAJAVA\UNIT-II>javac StaticInterfaceMethod.java

F:\BHAVAJAVA\UNIT-II>java StaticInterfaceMethod
We are AITS students
My Branch=CSE
*/
```

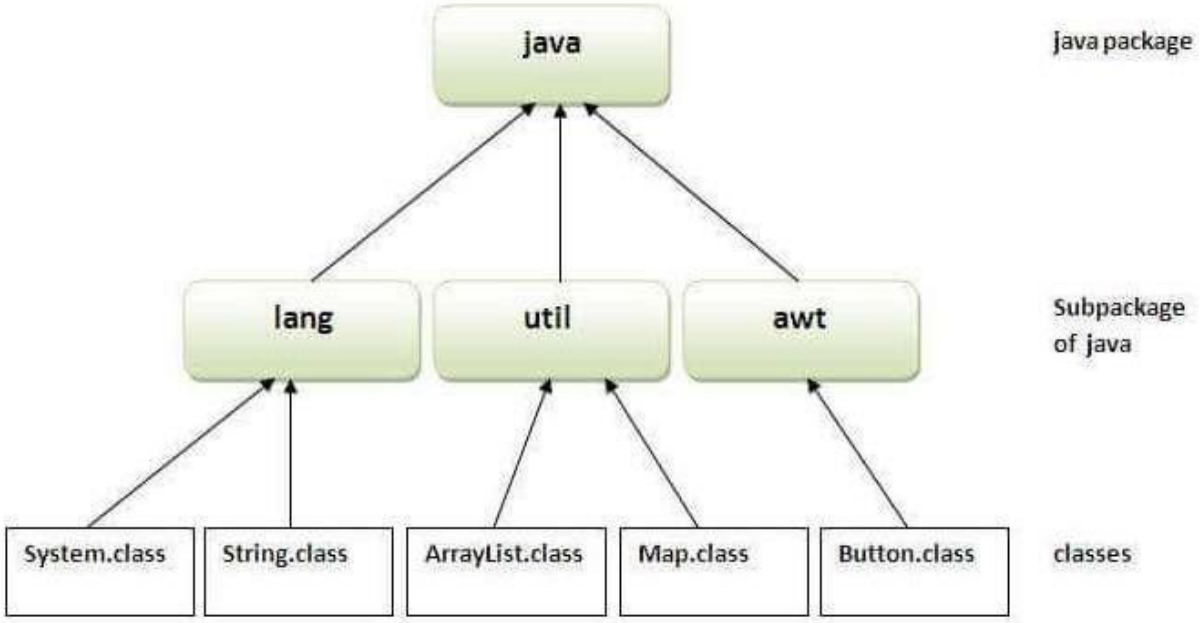
PACKAGES

- Packages are containers for java classes, interfaces, abstract classes and sub packages.
- Package is a mechanism to group related classes ,interfaces and enums and sub packages in to a single module.
- Package can be declared using the following statement :
  Syntax :

  package <package-name>;

- Package name should be declared in small letters.
- Package statement defines the namespace.
- Packages are similar to modules in python programming.
- The main use of package is
  1) To resolve naming conflicts
  2) For visibility control: We can define classes and interfaces that are not accessible outside the package.
- Java packages are two types.
  1. Built-in packages
     Some of the existing packages in Java are –
     java.lang – bundles the fundamental classes
     java.io – classes for input, output functions are bundled in this package
  2. User defined packages
     Programmer can create packages like built in packages.

Example for built-in packages:



JAVA UNIT-III

Defining Packages:

- The package statement defines a name space in which classes are stored.
- If you omit the package statement, the class names are put into the default package, which has no name.
- Syntax for defining package:
  package pkgname;

  Ex: package aits;
- You can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period.
- Syntax of a multileveled package statement is shown here:
  package pkg1[.pkg2[.pkg3]];

  Ex: package aits.cse.cse2;

Importing Package:

- Java includes the import statement to bring certain classes, or entire packages, into visibility.
- Once imported, a class can be referred to directly, using only its name.
- Syntax:
  i. Importing all classes in root package named by pkg:
  import pkg.*;
  Ex: import aits.*;
  ii. Importing only a class from a package:
  import pkg.ClassName;
  Ex: import aits.Results;
  iii. I mporting sub package classes:
  import pkg.[pkg2.[pkg3]...].*;
  Ex: import aits.cse.cse2.*;

Access Protection:

- Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages.
- Java addresses five categories of visibility for class members:
  - Same class
  - Subclasses in the same package
  - Non-subclasses in the same package
  - Subclasses in different packages
  - Classes that are neither in the same package nor subclasses

Access Level	Private	No modifier or default	Protected	public
Same class	YES	YES	YES	YES
Same package Sub class	NO	YES	YES	YES
Same package Non sub class (through object)	NO	YES	YES	YES
Different package sub class	NO	NO	YES	YES
Different package Non sub class	NO	NO	NO	YES

Note: In order to access package classes outside define package class as **public** class, define constructors inside class as **public** and methods as **public**.

Step by Step development and execution of Package Application:

1. Create or define a class .

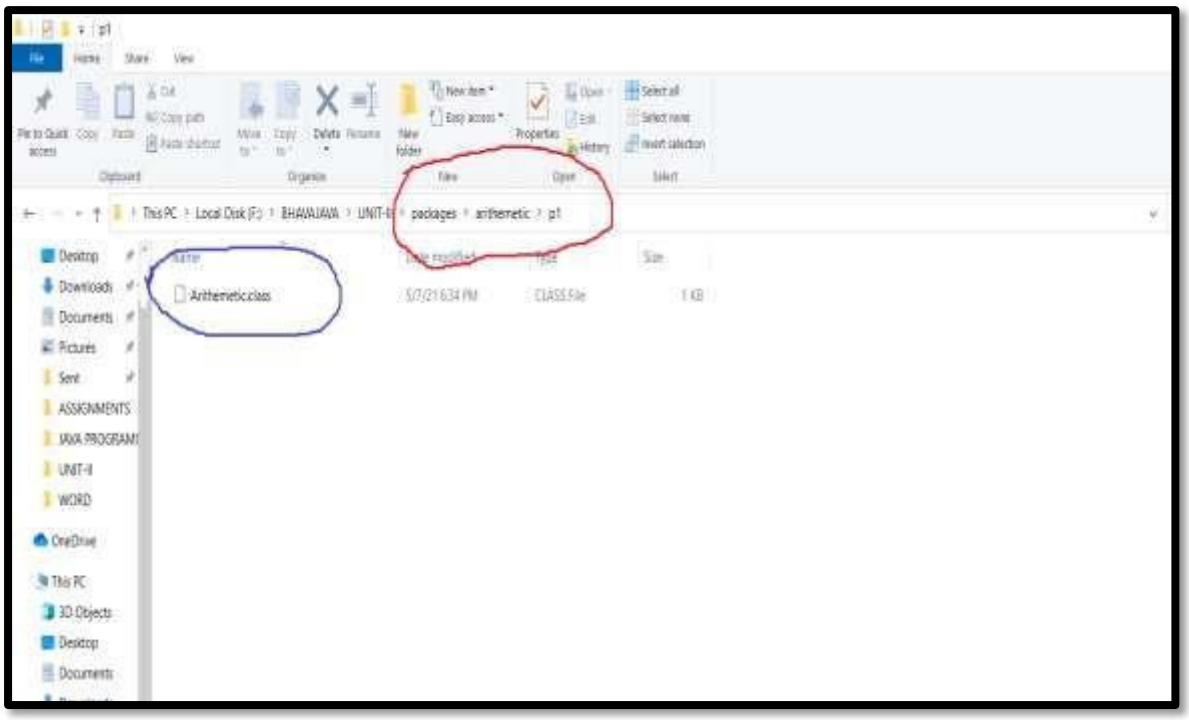
```
//Arithemetic.java
package p1;
public class Arithemetic{
 public double addition(double a,double b){
      return (a+b);
 }
 public double subtraction(double a,double b){
      return (a-b);
 }
 public double multiplication(double a,double b){
      return (a*b);
 }
 public double division(double a,double b){
      return (a/b);
 }
}
```

2. Do package compilization with following command.

javac -d . PackageClass.java

F:\BHAVAJAVA\UNIT-II\packages\arithemetic>javac -d . Arithemetic.java

- With the above command, Java Compiler creates a folder in respective application folder.
- The name of the folder same as package name.
- Inside the newly created folder **Arithemetic.class** file is saved.
- This complete execution is invoked by **Java compiler.**



3. Create a class which imports package class in application folder.

```
//PackageDemo.java
import p1.Arithemetic;
class PackageDemo{
      public static void main(String[] args){
         Arithemetic ar=new Arithemetic();
         System.out.println("Invoke Arithemetic class methods from Package p1");
         double a=ar.addition(2.7,9.9);
         System.out.println("Addition="+a);
         double s=ar.subtraction(2.7,9.9);
         System.out.println("Subtraction="+s);
         double m=ar.multiplication(2.7,9.9);
         System.out.println("Multiplication="+m);
         double d=ar.division(2.7,9.9);
         System.out.println("Division="+d);
      }
}
```

4. Compile and execute above class with following commands.

```
/* F:\BHAVAJAVA\UNIT-II\packages\arithemetic>javac PackageDemo.java
   F:\BHAVAJAVA\UNIT-II\packages\arithemetic>java PackageDemo
   Invoke Arithemetic class methods from Package p1
   Addition=12.600000000000001
   Subtraction=-7.2
   Multiplication=26.730000000000004
   Division=0.27272727272727276
```

```
*/
```


Example Program on Packages:

Step by Step development and execution of Package Application:

1. **Create package classes.**
   - **i) Figure.java**
   - **ii) AbstractFigure.java**
   - **iii) TraingleFigure.java**
   - **iv) RectangleFigure.java**

```
//Packages, Interfaces, Abstract Methods and Abstract Classes
//Figure.java
package p1;
public interface Figure{
      void display();
      double area();
}


//AbstractFigure.java
package p1;
public abstract class AbstractFigure implements Figure{
      public double dim1,dim2;
      public AbstractFigure(){
         this.dim1=0;
         this.dim2=0;
      }
    public AbstractFigure(double dim1,double dim2){
         this.dim1=dim1;
         this.dim2=dim2;
      }
    public void display(){//Concrete method
         System.out.println("Dimensions are");
         System.out.println("Dim1="+dim1+"\t"+"Dim2="+dim2);
      }
      public abstract double area(); //Abstract method
}

//TriangleFigure.java
package p1;
public class TraingleFigure extends AbstractFigure {
      public TraingleFigure() {
         super();
      }
    public TraingleFigure(double dim1,double dim2){
         super(dim1,dim2);
      }
    public double area() {
         double ar=0.5*dim1*dim2;
         return ar;
      }
```

```
}
//RectangleFigure.java
package p1;
public class RectangleFigure extends AbstractFigure{
      public RectangleFigure(){
          super();
      }
    public RectangleFigure(double dim1,double dim2){
          super(dim1,dim2);
      }
    public double area() {
          double ar=dim1*dim2;
          return ar;
      }
}
```

2. Do package compilization with following command.

```
F:\BHAVAJAVA\UNIT-II\packages\figures>javac -d . *.java
```

- With the above command, Java Compiler creates a folder in respective application folder.
- The name of the folder same as package name.
- Inside the newly created folder **Figure.class, AbstractFigure.class, TraingleFigure.class and RectangleFigure.class** file is saved.
- This complete execution is invoked by **Java compiler.**

3. Create a class which imports package class in application folder.

```
import p1.*;
class InterfaceDemo{
   public static void main(String[] args){
      Figure f;
      double ar;
      System.out.println("Accessing from Package classes P1");
      System.out.println("Figure. ....");
      System.out.println("RectangleFigure. ....");
      f=new RectangleFigure(5,7);
      f.display();
      ar=f.area();
      System.out.println("Area of RectangleFigure="+ar);
      System.out.println("TraingleFigure. ....");
      f=new TraingleFigure(8,9);
      f.display();
      ar=f.area();
      System.out.println("Area of TraingleFigure ="+ar);
   }
}
```

4. Compile and execute above class with following commands.

```
F:\BHAVAJAVA\UNIT-II\packages\figures>javac  InterfaceDemo.java

F:\BHAVAJAVA\UNIT-II\packages\figures>java InterfaceDemo
Accessing from Package classes P1
Figure.....
RectangleFigure.....
Dimensions are
Dim1=5.0        Dim2=7.0
Area of RectangleFigure=35.0
TraingleFigure.....
Dimensions are
Dim1=8.0        Dim2=9.0
Area of TraingleFigure =36.0
```

ANNAMACHARYA INSTITUTE OF TECHNOLGY AND SCIENCES::RAJAMPET
(AUTONOMOUS)
DEPARTMENT OF CSE

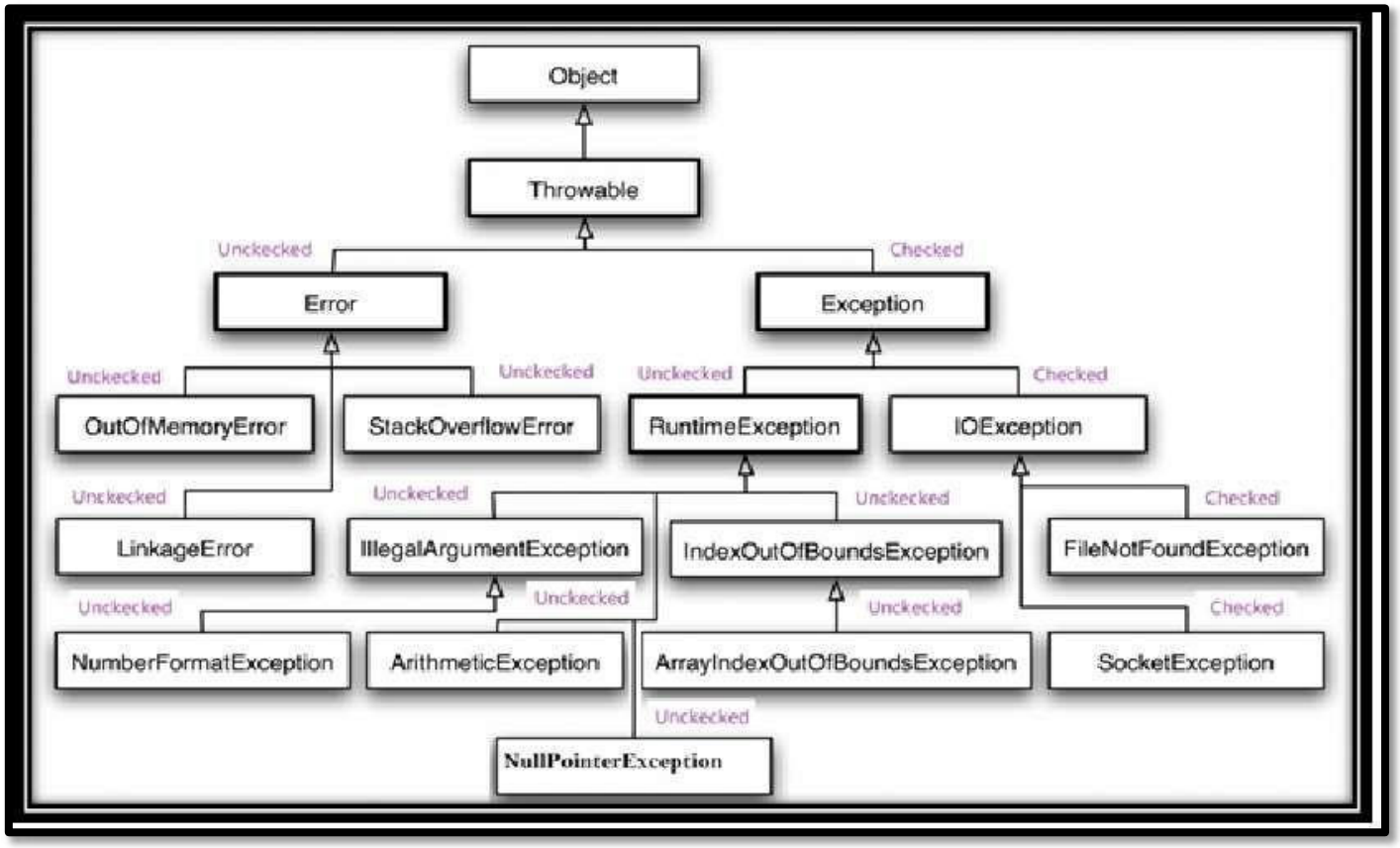
YEAR & SEM: II B.TECH II SEM **ACADEMIC YEAR: 2020-21**
BRANCH &SEC: CSE
SUBJECT: OBJECT ORIENTED PROGRAMMING USING JAVA
ASSIGNMENT NO: 2 **TOPICS COVERED: UNIT-II**
****

1.	Define inheritance. Write a java program to implement multi-level inheritance. 1
2.	What is an interface? Explain the definition and implementation of interface in Java.
3.	What is meant by access protection? Explain different access specifiers supported by Java with an example of each.
4.	a) Define Inheritance. Explain how one class inherits another class with an example. b) Define package. Write a simple java program to implement package.
5.	a) When a class called as abstract classes? Explain b) Is interfaces can be extended? Explain with an example
6..	a) What is meant by method overriding? Illustrate with an example. b) How packages are imported? Explain with suitable example.
7.	What are the uses of final keyword in inheritance? Explain with suitable examples.
8.	a) Differentiate between classes and interfaces? b) How Interfaces can be extended? Explain with an example?
9.	Write a sample program to illustrate packages.
10.	a) What is the difference between an Interface, class and an Abstract Class? b) How can you declare variables in interfaces? Explain with example.
11.	Define package? Explain four categories of visibility for class members with simple example.
12.	a) Explain with suitable example, how super class variable can refer subclass objects? b) "Interface variables are static and final by default in Java" - Support this statement with proper explanation
13.	a) With an example explain the effect of using final keyword in inheritance. b) Write a program to read two numbers in one class and do the arithmetic operations on these two numbers in another class, which is stored in another package
14.	How java supports multiple inheritances?
15.	Explain static members and default interface methods in interfaces.
16.	Explain method overriding with an example.
17.	Explain dynamic method dispatch using interfaces.
18.	Demonstrate Dynamic stacks using packages.
19.	How to implement multiple inheritance in java. Explain with Example.
20.	Explore Object class.

OHM SRI SAIRAM

UNIT-III
CHAPTER-I
EXCEPTION HANDLING

- Java Exception handling provides systematic approach for handling abnormal conditions during compilzation and execution of program.
- Java Exception handling mechanism is object oriented.
- The **java.lang** package provides well defined classes to handle exception handling



- **Exception:**
  - An Exception is an abnormal condition that arises during execution of program.
  - An Exception is run time error.
  - If Java program unable to execute any piece of code at runtime simply it returns the problem in terms of exception object.
  - Java exceptions are classified as checked exceptions and un-checked exceptions.

- Checked Exception:

  Checked Exceptions are exceptions, that abnormal condition or exception **checked at compile time rather than at run-time**.

- Un-Checked Exception:

  Un-Checked Exceptions are exceptions, that abnormal condition or exception **checked at run time rather than at compile-time**.

Exception handling mechanism:

- A **Java exception is an object** that describes an exceptional (that is, error) condition that has occurred in a piece of code.
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is caught and processed.
- Java Exception handling managed through five keywords:
  - try
  - throw
  - catch
  - throws
  - finally
- **try** keyword: Which block of statements raised exceptions, that will kept in try block.
- **throw keyword:** If an exception is raised in try block, you can explicitly throw the exception using throw keyword.
- **catch keyword:** The thrown exception from try block handled in **catch** block.
- **throws keyword:** Any exception not handled by respective method and passed to caller of the method, then it is thrown through **throws** keyword.
- **finally keyword:**

  A **finally** is a keyword used to create a block of code that follows a try/catch block.

OHM SRI SAIRAM

A **finally** block contains all the crucial codes such as closing connections, stream, etc that is always executed whether an exception occurs within a try block or not.

- Syntax:

```
try {
// block of code to monitor for errors
}
catch (ExceptionType1 exOb) {
// exception handler for ExceptionType1
}
catch (ExceptionType2 exOb) {
// exception handler for ExceptionType2
}
// ...
finally {
// block of code to be executed after try block ends
}
```

- The following example demonstrates Exception handling

```
//ExceptionDemo.java
import java.util.*;
class ExceptionDemo{
      public static void main(String[] args){
         Scanner sin=new Scanner(System.in);
         System.out.println("Enter a,b");
         int a=sin.nextInt();
         int b=sin.nextInt();
         try{
              int c=a/b;
              System.out.println("c="+c);
         }
           catch(ArithmeticException e){
                   System.out.println("Exception occured="+e);
           }
      }
}
```

```
/*
F:\BHAVAJAVA\UNIT-III>javac ExceptionDemo.java

F:\BHAVAJAVA\UNIT-III>java ExceptionDemo
Enter a,b
2
0
Exception occured=java.lang.ArithmeticException: / by zero
F:\BHAVAJAVA\UNIT-III>javac ExceptionDemo.java

F:\BHAVAJAVA\UNIT-III>java ExceptionDemo
Enter a,b
5
2
c=2
*/
```

Multiple catch Clauses

- In some cases, more than one exception could be raised by a single piece of code.
- To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception.
- When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed.
- After one catch statement executes, the others are bypassed, and execution continues after the try /catch block.

// Demonstrate the multiple-catch feature.

```
//MultipleCatch.java
import java.util.*;
class MultipleCatch {
    public static void main(String args[]) {
        Scanner sin=new Scanner(System.in);
        int a=10,b=0;
        int arr[]={1,2,3};
        try {
            System.out.println("1 for Division 2 for Array Accessing");
            System.out.println("Enter Choice c=");
            int c=sin.nextInt();
            if(c==1){
                //Generates an ArithmeticException
```

```
                    c=a/b;
                   System.out.println("Division="+c);
                }
                else if(c==2){
                     //Generates an ArrayIndexOutOfBoundsException
                     arr[10]=19;
                }
            } catch(ArithmeticException  e) {
                 //The below method prints Exception Information
                   e.printStackTrace();
             }
                catch(ArrayIndexOutOfBoundsException e){
                     e.printStackTrace();//prints Exception Information
                }
             System.out.println("After multiple catch...");
        }
}

/*
F:\BHAVAJAVA\UNIT-III>javac MultipleCatch.java

F:\BHAVAJAVA\UNIT-III>java MultipleCatch
1 for Division 2 for Array Accessing
Enter Choice c=
1
java.lang.ArithmeticException: / by zero
        at MultipleCatch.main(MultipleCatch.java:15)
After multiple catch...

F:\BHAVAJAVA\UNIT-III>java MultipleCatch
1 for Division 2 for Array Accessing
Enter Choice c=
2
java.lang.ArrayIndexOutOfBoundsException: 10
        at MultipleCatch.main(MultipleCatch.java:20)
After multiple catch...
*/
```

Nested try Statements

- The try statement can be nested. That is, a try statement can be inside the block of another try.

OHM SRI SAIRAM

- Each time a try statement is entered, the context of that exception is pushed on the stack.
- If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement's catch handlers are inspected for a match.
- This continues until one of the catch statements succeeds, or until the entire nested try statements are exhausted.
- If no catch statement matches, then the Java run-time system will handle the exception.

```
//Demonstration of Nested Try
//NestedTryDemo.java
class NestedTryDemo {
   public static void main(String args[]) {
      try {
         int a[]=new int[10];
         System.out.println(a[12]);
         try {
            System.out.println("Division");
            int res = 100/ 0;
         }
         catch (ArithmeticException e) {
            System.out.println("Excception from Inner Try Block");
            e.printStackTrace();
         }
      }
      catch (ArrayIndexOutOfBoundsException e) {
         System.out.println("Excception from Outer Try Block");
         e.printStackTrace();
  }
   }
}
```

```
/*
F:\BHAVAJAVA\UNIT-III>javac NestedTryDemo.java

F:\BHAVAJAVA\UNIT-III>java NestedTryDemo
Excception from Outer Try Block
java.lang.ArrayIndexOutOfBoundsException: 12
        at NestedTryDemo.main(NestedTryDemo.java:7)
*/
```

Using "throw" keyword:

- To throw an exception explicitly, using the throw statement.
- Syntax:

  throw ThrowableInstance;

  Here, ThrowableInstance must be an object of type **Throwable** or a **subclass of Throwable.**
- Primitive types, such as int or char, as well as non-Throwable classes, such as String and Object, cannot be used as exceptions.
- There are two ways you can obtain a Throwable object:

  i) Using a parameter in a catch clause

  Ex: catch(ArithmeticException e) { …}

  Here **e** is parameter.

  ii) Creating one with the new operator

  Ex: throw **new ArithemeticException ();**

```
//ThrowDemo.java
import java.util.*;
class ThrowDemo{
    public static void main(String[] args){
        Scanner sin=new Scanner(System.in);
        System.out.println("Enter a,b=");
        int a=sin.nextInt();
        int b=sin.nextInt();
        try{
```

```
                if(b!=0){
                     int c=a/b;
                     System.out.println("c="+c);
                }
                else
                     throw new ArithmeticException();
          }
              catch(ArithmeticException e){
                     e.printStackTrace();
                }
     }
}
/*
F:\BHAVAJAVA\UNIT-III>javac ThrowDemo.java

F:\BHAVAJAVA\UNIT-III>java ThrowDemo
Enter a,b=
10
2
c=5

F:\BHAVAJAVA\UNIT-III>java ThrowDemo
Enter a,b=
2
0
java.lang.ArithmeticException
        at ThrowDemo.main(ThrowDemo.java:15)
*/
```

OHM SRI SAIRAM

Using throws keyword:

- If a method is capable of causing an exception that it does not handle, it must specify this by including a throws clause in the method's declaration.
- A throws clause lists the types of exceptions that a method might throw.
- This is necessary for all exceptions, except those of type Error or RuntimeException, or any of their subclasses.
- All other exceptions that a method can throw must be declared in the throws clause.
- This is the general form of a method declaration that includes a throws clause:

  type method-name(parameter-list) throws exception-list
  {
  // body of method
  }

Here, exception-list is a comma-separated list of the exceptions that a method can throw.

```
//ThrowsDemo.java
import java.util.*;
class Division{
     int divide(int a,int b)throws ArithmeticException{
          int c=a/b;
         return c;
     }
}

class ThrowsDemo{
     public static void main(String[] args){
          Division d=new Division();
          Scanner sin=new Scanner(System.in);
          System.out.println("Enter a,b");
          int a=sin.nextInt();
          int b=sin.nextInt();
          try{
               int div=d.divide(a,b);
               System.out.println("Division Result="+div);
             }
              catch(Exception e){
```

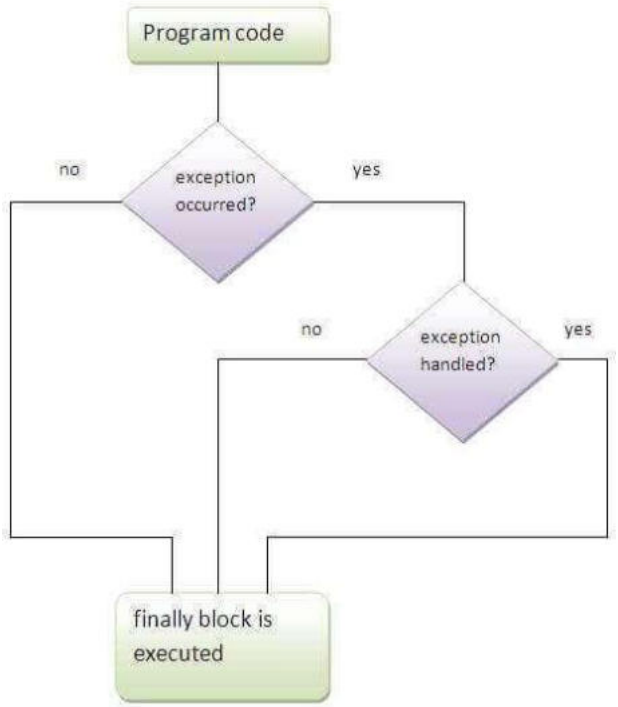
```
                    e.printStackTrace();
                }
        }
    }
    /*
    F:\BHAVAJAVA\UNIT-III>javac ThrowsDemo.java

    F:\BHAVAJAVA\UNIT-III>java ThrowsDemo
    Enter a,b
    10
    2
    Division Result=5

    F:\BHAVAJAVA\UNIT-III>java ThrowsDemo
    Enter a,b
    2
    0
    java.lang.ArithmeticException: / by zero
            at Division.divide(ThrowsDemo.java:5)
            at ThrowsDemo.main(ThrowsDemo.java:18)
    */
```

Using finally keyword:

- When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method.
- Depending upon how the method is coded, it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods.
- The **finally** keyword is designed to address this problem.
- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.

OHM SRI SAIRAM

- This can be useful for closing file handles and freeing up any other resources that might have been allocated at the beginning of a method with the intent of disposing of them before returning.
- The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.
- **For each try block there can be zero or more catch blocks, but only one finally block.**

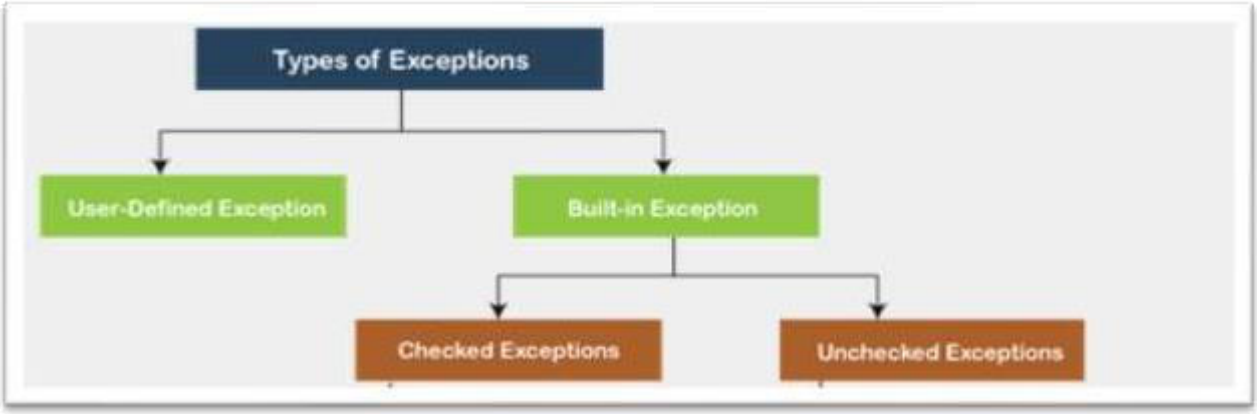



```
//Finally.java
import java.io.*;
import java.util.*;
class FinallyDemo{
     public static void main(String[] args)throws IOException{
          Scanner sin=new Scanner(System.in);
          FileWriter fw=new FileWriter("divfile.txt");
          System.out.println("Enter a,b");
          int a=sin.nextInt();
          int b=sin.nextInt();
          try{
               int c=a/b;
               System.out.println("c="+c);
               fw.write("Division Successful..."+a+"/"+b+"="+c);
```

```
            }
                catch(ArithmeticException e){
                      System.out.println("Exception occured="+e);
                 }
                 finally{
                      System.out.println("File closed...");
                      fw.close();
                 }
      }
}
/*
F:\BHAVAJAVA\UNIT-III>javac FinallyDemo.java

F:\BHAVAJAVA\UNIT-III>java FinallyDemo
Enter a,b
10
2
c=5
File closed...

F:\BHAVAJAVA\UNIT-III>java FinallyDemo
Enter a,b
2
0
Exception occured=java.lang.ArithmeticException: / by zero
File closed...
*/
```

Java Built-in Exceptions:

- All Java built-in exceptions are defined in java.lang library.
- Exceptions can be categorized into two ways:



OHM SRI SAIRAM

Sr.No.	Unchecked Exceptions & Description
1	**ArithmeticException** Arithmetic error, such as divide-by-zero.
2	**ArrayIndexOutOfBoundsException** Array index is out-of-bounds.
3	**ArrayStoreException** Assignment to an array element of an incompatible type.
4	**IllegalMonitorStateException** Illegal monitor operation, such as waiting on an unlocked thread.
5.	**IndexOutOfBoundsException** Some type of index is out-of-bounds.
6.	**NullPointerException** Invalid use of a null reference.
7.	**NumberFormatException** Invalid conversion of a string to a numeric format.

Sr.No.	Checked Exceptions & Description
1	**ClassNotFoundException** Class not found.
2	**CloneNotSupportedException** Attempt to clone an object that does not implement the Cloneable interface.
3	**IllegalAccessException** Access to a class is denied.
4	**InstantiationException** Attempt to create an object of an abstract class or interface.
5	**InterruptedException** One thread has been interrupted by another thread.
6	**NoSuchFieldException** A requested field does not exist.

OHM SRI SAIRAM

7	**NoSuchMethodException** A requested method does not exist.

```
//Checked Exception
import java.io.*;
class CheckedExceptionDemo{
      public static void main(String args[]){
            File file = new File("not_existing_file.txt");
            FileInputStream stream = new FileInputStream(file);
    }
}
/*

F:\BHAVAJAVA\UNIT-III>javac CheckedExceptionDemo.java
CheckedExceptionDemo.java:5: error: unreported exception FileNotFoundException;
must be caught or declared to be thrown
                FileInputStream stream = new FileInputStream(file);
                                         ^
1 error
*/
//Unchecked Exception
import java.util.*;
class UnCheckedExceptionDemo{
      public static void main(String[] args){
            Scanner sin=new Scanner(System.in);
            System.out.println("Enter a,b=");
            int a=sin.nextInt();
            int b=sin.nextInt();
            int c=a/b;
            System.out.println("c="+c);
    }
}
/*
F:\BHAVAJAVA\UNIT-III>javac UnCheckedExceptionDemo.java
F:\BHAVAJAVA\UNIT-III>java UnCheckedExceptionDemo
Enter a,b=
10
2
c=5
```

OHM SRI SAIRAM

```
F:\BHAVAJAVA\UNIT-III>java UnCheckedExceptionDemo
Enter a,b=
2
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at UnCheckedExceptionDemo.main(UnCheckedExceptionDemo.java:8)
*/
```

Difference between Checked and Unchecked Exception

S.No	Checked Exception	Unchecked Exception
1.	These exceptions are checked at compile time. These exceptions are handled at compile time too.	These exceptions are just opposite to the checked exceptions. These exceptions are not checked and handled at compile time.
2.	These exceptions are direct subclasses of exception but not extended from RuntimeException class.	They are the direct subclasses of the RuntimeException class.
3.	It is required to provide the try-catch and try-finally block to handle the checked exception.	In the case of unchecked exception it is not mandatory.
4.	Common checked exceptions include IOException, DataAccessException, InterruptedException, etc.	Common unchecked exceptions include ArithmeticException, InvalidClassException, NullPointerException, etc.
5.	These exceptions are propagated using the throws keyword.	These are automatically propagated.

OHM SRI SAIRAM

Creating Your Own Exception Subclasses or User Defined Exceptions

- Although Java's built-in exceptions handle most common errors, you will probably want to create your own exception types to handle situations specific to your applications.
- In Java, we can write our own exception class by extends the Exception class. We can throw our own exception on a particular condition using the throw keyword.
- The Exception class does not define any methods of its own. These methods are inherited from **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them. Exception defines four public constructors.

Exception( )

The first form creates an exception that has no description.

Exception(String msg)

The second form lets you specify a description of the exception.

Throwable class

- The **java.lang.Throwable** class is the superclass of all errors and exceptions in the Java language.

Sr.No.	Throwable class methods & Description
1	Throwable fillInStackTrace() This method fills in the execution stack trace.
2	Throwable getCause() This method returns the cause of this throwable or null if the cause is nonexistent or unknown.
3	String getMessage() This method returns the detail message string of this throwable.
4	StackTraceElement[] getStackTrace() This method provides programmatic access to the stack trace information printed by printStackTrace().

OHM SRI SAIRAM

5	void printStackTrace() This method prints this throwable and its backtrace to the standard error stream.
6	void setStackTrace(StackTraceElement[] stackTrace) This method sets the stack trace elements that will be returned by getStackTrace() and printed by printStackTrace() and related methods.
7	String toString() This method returns a short description of this throwable.

```
//Creating Own Exception
//UserDefinedException.java
import java.util.*;
class InvalidAgeException extends Exception{
   InvalidAgeException(String s){
         super(s);
   }
    public String toString(){
         return "User Defined InvalidAgeException";
    }
}
class UserDefinedException{
   public static void main(String args[]){
         Scanner sin=new Scanner(System.in);
         System.out.println("Enter Age");
         int age=sin.nextInt();
         try{
               if(age<18)
                     throw new InvalidAgeException("Not valid Age");
               else
                     System.out.println("You are eligible for voting");
         }
         catch(InvalidAgeException e){
               e.printStackTrace();
         }
    }
}
/*
F:\BHAVAJAVA\UNIT-III>javac UserDefinedException.java

F:\BHAVAJAVA\UNIT-III>java UserDefinedException
Enter Age
27
You are eligible for voting
```

```
F:\BHAVAJAVA\UNIT-III>java UserDefinedException
Enter Age
12
User Defined InvalidAgeException
        at UserDefinedException.main(UserDefinedException.java:19)
*/
```

Three Recently Added Exception Features(Beyond Syllabus)

- Beginning with JDK 7, three interesting and useful features have been added to the exception system.

1. Automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the try statement called try-with-resources.
2. The second feature is called multi-catch, and the third is sometimes referred to as final rethrow or more precise rethrow.
3. The multi-catch feature allows two or more exceptions to be caught by the same catch clause. Instead of having to catch each exception type individually, you can use a single catch clause to handle all of the exceptions without code duplication. To use a multi-catch, separate each exception type in the catch clause with the OR operator.

```
// Demonstrate the multi-catch feature.
//MultiException.java
import java.util.*;
class MultiException {
   public static void main(String args[]) {
        Scanner sin=new Scanner(System.in);
        int a=10,b=0;
        int arr[]={1,2,3};
        try {
              System.out.println("1 for Division 2 for Array Accessing");
              System.out.println("Enter Choice c=");
              int c=sin.nextInt();
              if(c==1){
                 //Generate an ArithmeticException
                 c=a/b;
                 System.out.println("Division="+c);
```

```
                }
                else if(c==2)
                    //Generate an ArrayIndexOutOfBoundsException
                    arr[10]=19;
            }catch(ArithmeticException | ArrayIndexOutOfBoundsException e)
             {
              System.out.println("Exception caught="+e);
             }
      System.out.println("After multi-catch...");
  }
}

/*
F:\BHAVAJAVA\UNIT-III>javac MultiException.java

F:\BHAVAJAVA\UNIT-III>java MultiException
1 for Division 2 for Array Accessing
Enter Choice c=
1
Exception caught=java.lang.ArithmeticException: / by zero
After multi-catch...

F:\BHAVAJAVA\UNIT-III>java MultiException
1 for Division 2 for Array Accessing
Enter Choice c=
2
Exception caught=java.lang.ArrayIndexOutOfBoundsException: 10
After multi-catch...
*/
```

OHM SRI SAIRAM

UNIT-III
CHAPTER-II
MULTI THREADING

- Java provides built-in support for multithreaded programming. A multithreaded program contains two or more parts that can run concurrently.
- Each part of such a program is called a thread, and each thread defines a separate path of execution.
- Thus, multithreading is a specialized form of multitasking.
- Multi-tasking supported by all different types of Operating systems.
- There are two different types of multi-tasking.
  1. Process based multi-tasking
  2. Thread based multi-tasking

Process based Multitasking Vs Thread based Multi-tasking

Process based Multitasking:

- process-based multitasking is the feature that allows your computer to run two or more programs concurrently.
- For example, process based multitasking enables you to run the Java compiler at the same time that you are using a text editor or visiting a web site.
- In process-based multitasking, a program is the smallest unit of code that can be dispatched by the scheduler.

Thread based Multi-tasking:

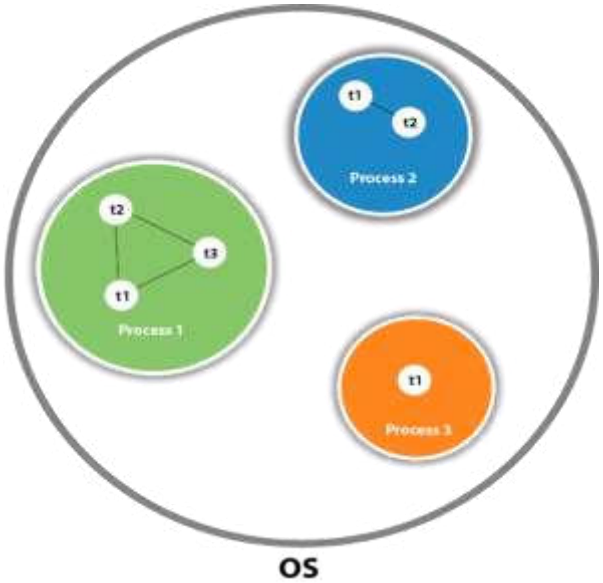
- In a thread-based multitasking environment, the thread is the smallest unit of dispatchable code.
- This means that a single program can perform two or more tasks simultaneously.
- For instance, a text editor can format text at the same time that it is printing, as long as these two actions are being performed by two separate threads.

The differences between process based multithreading and Thread based multi-threading demonstrated in following table.

Sr. No.	Process Based Multitasking	Thread Based Multitasking
1	Two or more programs run concurrently. E.g. web browser , text editor, music player etc.	Two or more parts of the same program run concurrently.
2	Each running program is called a process.	Each independent part is called a thread.
3	Program is the smallest unit of code that can be dispatched by the scheduler.	Thread is the smallest unit of code that can be dispatched by the scheduler.
4	It deals with the big picture.	It deals with the details.
5	It requires more overheads.	It requires less overheads.
6	Processes are heavyweight.	Threads are lightweight.
7	Each process require its own address space.	Threads belonging to same program share address space.
8	Context switching is expensive.	Context switching is cheap.
9	Inter-process communication is costly and limited.	Inter-thread communication is inexpensive.
10.	It is not under the control of Java.	It is under the control of Java.

The Java Thread Model

- A thread is a lightweight sub process, the smallest unit of processing. It is a separate path of execution.
- Threads are independent. If there occurs exception in one thread, it doesn't affect other threads. It uses a shared memory area.

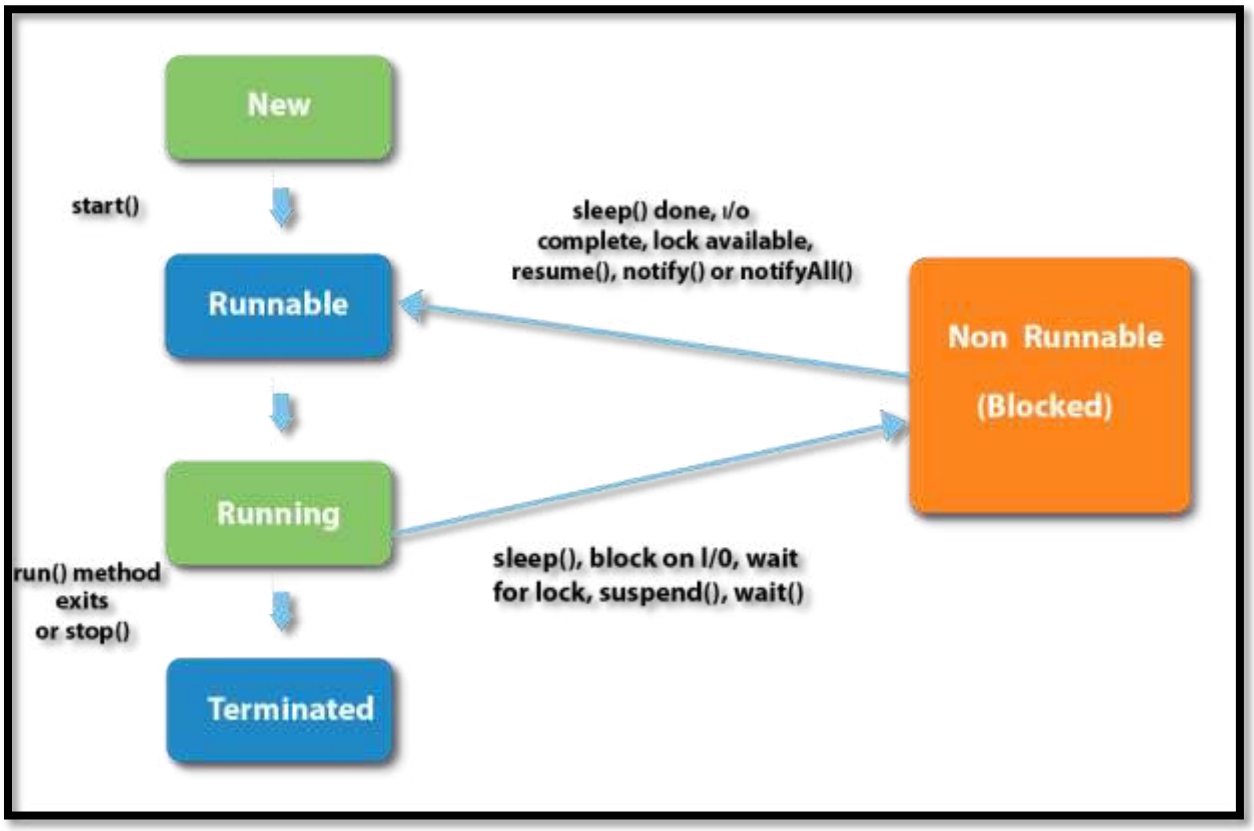


OHM SRI SAIRAM

- The Java run-time system depends on threads for many things, and all the class libraries are designed with multithreading in mind.
- Java uses threads to enable the entire environment to be asynchronous.
- This helps reduce inefficiency by preventing the waste of CPU cycles.

Thread Life Cycle:

- A thread can be in one of the five states.
- The life cycle of the thread in java is controlled by JVM.
- The java thread execution refers 5 states are as follows:
  New
  Runnable
  Running
  Non-Runnable (Blocked)
  Terminate



1) New: The thread is in new state if you create an instance of Thread class but before the Invocation of start() method.

2) Runnable: The thread is in runnable state after invocation of start() method, but the thread scheduler has not selected it to be the running thread.

OHM SRI SAIRAM

3) Running: The thread is in running state if the thread scheduler has selected it.

4) Non-Runnable (Blocked) : This is the state when the thread is still alive, but is currently not eligible to run.

5) Terminated: A thread is in terminated or dead state when its run() method exits.

Java Multithreading Model: Runnable Interface and Thread class

- Java's multithreading system is built upon the Thread class, its methods, and its parent interface, Runnable.
- Thread encapsulates a thread of execution.
- To create a new thread, java program will either **extend Thread class or implement the Runnable interface.**
- Java multithreading corresponding interfaces and classes defined in **java.lang package.**

Runnable Interface: The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread.

Syntax:

```
public interface Runnable{
        void run();
}
```

- When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run() method to be called in that separately executing thread.

Thread class:

Syntax:

```
public class Thread extends Object implements Runnable{
  //....
}
```

- A thread is a thread of execution in a program.
- The Java Virtual Machine allows an application to have multiple threads of execution running concurrently.

OHM SRI SAIRAM

Thread class fields, Constructors and Methods

Thread class Fields:

Following are the fields for **java.lang.Thread** class −

- **static int MAX_PRIORITY** − This is the maximum priority that a thread can have.
- **static int NORM_PRIORITY** − This is the default priority that is assigned to a thread.

Thread Class constructors

Constructor & Description
Thread() This allocates a new Thread object.
Thread(Runnable target) This allocates a new Thread object.
Thread(Runnable target, String name) This allocates a new Thread object.
Thread(String name) This constructs allocates a new Thread object.
Thread(ThreadGroup group, Runnable target) This allocates a new Thread object.
Thread(ThreadGroup group, Runnable target, String name) This allocates a new Thread object so that it has target as its run object, has the specified name as its name, and belongs to the thread group referred to by group.
Thread(ThreadGroup group, String name) This allocates a new Thread object.

OHM SRI SAIRAM

Thread Class methods

Method & Description
public static int activeCount() This method returns the number of active threads in the current thread's thread group.
public static Thread currentThread() This method returns a reference to the currently executing thread object.
public String getName() This method returns this thread's name.
public int getPriority() This method Returns this thread's priority.
public ThreadGroup getThreadGroup() This method returns the thread group to which this thread belongs.
public void interrupt() This method interrupts this thread.
public static boolean interrupted() This method tests whether the current thread has been interrupted.
public boolean isAlive() This method tests if this thread is alive.
public boolean isDaemon() This method tests if this thread is a daemon thread.
public boolean isInterrupted() This method tests whether this thread has been interrupted.
public void join() Waits for this thread to die.
public void join(long millis) Waits at most millis milliseconds for this thread to die.

OHM SRI SAIRAM

public void join(long millis, int nanos) Waits at most millis milliseconds plus nanos nanoseconds for this thread to die.
public void run() If this thread was constructed using a separate Runnable run object, then that Runnable object's run method is called; otherwise, this method does nothing and returns
public void setDaemon(boolean on) This method marks this thread as either a daemon thread or a user thread.
public void setName(String name) This method changes the name of this thread to be equal to the argument name.
public void setPriority(int newPriority) This method changes the priority of this thread.
public static void sleep(long millis) This method causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds, subject to the precision and accuracy of system timers and schedulers.
public static void sleep(long millis, int nanos) This method causes the currently executing thread to sleep (cease execution) for the specified number of milliseconds plus the specified number of nanoseconds, subject to the precision and accuracy of system timers and schedulers.
public void start() This method causes this thread to begin execution; the Java Virtual Machine calls the run method of this thread.
public String toString() This method Returns a string representation of this thread, including the thread's name, priority, and thread group.
public static void yield() This method causes the currently executing thread object to temporarily pause and allow other threads to execute.

OHM SRI SAIRAM

Java Main Thread:

- When a Java program starts up, one thread begins running immediately.
- This is usually called the main thread of your program, because it is the one that is executed when your program begins.
- The main thread is important for two reasons:
  1. It is the thread from which other "child" threads will be spawned.
  2. Often, it must be the last thread to finish execution because it performs various shutdown actions.
- Although the main thread is created automatically when your program is started, it can be controlled through a Thread object.
- To do so, you must obtain a reference to it by calling the method currentThread( ), which is a public static member of Thread.
- Syntax:

  public **static Thread currentThread( )**

  This method returns a reference to the thread in which it is called. Once you have a reference to the main thread, you can control it just like any other thread.

```
//Demonstration of Main Thread
class MainThreadDemo{
   public static void main(String args[]){
      Thread t=Thread.currentThread();
        System.out.println("Current Thread="+t);
        t.setName("AITS-II CSE");
        System.out.println("After Name Changing="+t.getName());
        System.out.println("Thread Priority="+t.getPriority());
        System.out.println("Thead is alive="+t.isAlive());
        System.out.println("Thread Description="+t);
        try{
            for(int i=1;i<=5;i++){
                System.out.println("i value is="+i);
                 Thread.sleep(10000);
             }
          }catch(InterruptedException e){
              e.printStackTrace();
           }
     System.out.println("HAVE A NICE DAY");
    }
}
/*
```

OHM SRI SAIRAM

```
F:\BHAVAJAVA\MultiThreading>javac MainThreadDemo.java
F:\BHAVAJAVA\MultiThreading>java MainThreadDemo
Current Thread=Thread[main,5,main]
After Name Changing=AITS-II CSE
Thread Priority=5
Thead is alive=true
Thread Description=Thread[AITS-II CSE,5,main]
i value is=1
i value is=2
i value is=3
i value is=4
i value is=5
HAVE A NICE DAY
*/
```

Creating Threads:

Java defines two ways to create threads.

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

1. Implementing **Runnable** interface:

- The easiest way to create a thread is to create a class that implements the **Runnable interface**. Runnable abstracts a unit of executable code.
- You can construct a thread on any object that implements Runnable.
- To implement Runnable, a class need only implement a single method called **run( )**.
- Inside run ( ), you will define the code that constitutes the new thread.
- The run( ) establishes the entry point for another, concurrent thread of execution within your program. This thread will end when run( ) returns.
- After you create a class that implements Runnable, you will instantiate an object of type Thread from within that class.
- Thread defines several constructors.
- The one that we will use is shown here:
  Thread(Runnable threadOb, String threadName)
- In this constructor, threadOb is an instance of a class that implements the Runnable interface. This defines where execution of the thread will begin. The name of the new thread is specified by threadName.
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within Thread.
- In essence, start ( ) executes a call to run( ).
- The start( ) method is shown here:
  void start( )

OHM SRI SAIRAM

```
//Thread Creation by implementing Runnable interface
class RunnableThread implements Runnable{
      String name;
      RunnableThread(String name){
           this.name=name;
      }
      public void run(){
           System.out.println("I am Child Thread,created from Main Thread");
           System.out.println("Child Thread Prints from 10.....1");
           try{
                for(int i=10;i>=1;i--){
                      System.out.println(name+":"+i);
                     Thread.sleep(1000);
               }
          }catch(InterruptedException e){
               e.printStackTrace();
           }
          System.out.println("Child Thread Exited");
       }
}
class MainThreadR{
   public static void main(String args[]){
        System.out.println("From Main Thread");
        RunnableThread r=new RunnableThread("Child Thread");
       Thread t=new Thread(r);
        t.start();
       System.out.println("Main Thread Prints from 1.....10");
        try{
            for(int i=1;i<=10;i++){
                  System.out.println("Main Thread:"+i);
                  Thread.sleep(1000);
            }
         }catch(InterruptedException e){
             e.printStackTrace();
           }
         System.out.println("Main Thread Completed");
     }
 }
/*
F:\BHAVAJAVA\MultiThreading>javac MainThreadR.java
F:\BHAVAJAVA\MultiThreading>java MainThreadR
From Main Thread
Main Thread Prints from 1.....10
```

```
I am Child Thread,created from Main Thread
Child Thread Prints from 10.....1
Main Thread:1
Child Thread:10
Child Thread:9
Main Thread:2
Main Thread:3
Child Thread:8
Main Thread:4
Child Thread:7
Main Thread:5
Child Thread:6
Main Thread:6
Child Thread:5
Main Thread:7
Child Thread:4
Main Thread:8
Child Thread:3
Child Thread:2
Main Thread:9
Main Thread:10
Child Thread:1
Main Thread Completed
Child Thread Exited
*/
```

Extending Thread:

- The second way to create a thread is to create a new class that extends Thread
- The extending class must override the run ( ) method, which is the entry point for the new thread. It must also call start( ) to begin execution of the new thread.

//MainThreadEx.java

```
class ThreadEx extends Thread{
   String str;
   ThreadEx(String str){
      this.str=str;
   }
   public void run(){
       System.out.println("I am Child Thread,created from Main Thread");
       System.out.println("Child Thread Prints from 10.....1");
```

```
        try{
             for(int i=10;i>=1;i--){
                  System.out.println("Child Thread="+i);
                  Thread.sleep(1000);
            }
       }catch(Exception e){e.printStackTrace();}
        System.out.println("Child Thread Completed");

    }
}
class MainThreadEx{
    public static void main(String args[]){
            System.out.println("Main Thread Running");
            ThreadEx t=new ThreadEx("Child");
            t.start();
            try{
               for(int i=1;i<=10;i++){
                  System.out.println("Main Thread="+i);
                  Thread.sleep(1000);
              }
          }catch(Exception e){e.printStackTrace();}
          System.out.println("Main Thread Completed");
  }
}

/*
F:\BHAVAJAVA\MultiThreading>javac MainThreadEx.java
F:\BHAVAJAVA\MultiThreading>java MainThreadEx
Main Thread Running
Main Thread=1
I am Child Thread,created from Main Thread
Child Thread Prints from 10.....1
Child Thread=10
Main Thread=2
Child Thread=9
Child Thread=8
Main Thread=3
Main Thread=4
Child Thread=7
Main Thread=5
Child Thread=6
Child Thread=5
Main Thread=6
```

```
Main Thread=7
Child Thread=4
Main Thread=8
Child Thread=3
Main Thread=9
Child Thread=2
Main Thread=10
Child Thread=1
Main Thread Completed
Child Thread Completed

*/
```

Choosing an Approach

- Java programming is free to override non final methods of super class methods in sub class. If you will override the Thread class methods in extended class extending Thread class is best approach.
- If you will not be overriding any of Thread's other methods, it is probably best simply to implement Runnable.
- Also, by implementing Runnable, your thread class does not need to inherit Thread, making it free to inherit a different class.

Creating Multiple Threads

```
//Creating Multiple Threads
import java.awt.*;
class ThreadEx extends Thread{
   String name;
   ThreadEx(String name){
        this.name=name;
   }
   public void run(){
       try{
            if(name=="s1"){
                 for(int i=1;i<=10;i++){
                 System.out.println("s1:"+i);
                 Thread.sleep(1000);
            }
        }
        else if(name=="s2"){
            for(int i=1;i<=10;i++){
                  System.out.println("s2:Disturbed: creates beep sound ");
                  Toolkit.getDefaultToolkit().beep();
                  Thread.sleep(1000);
```

OHM SRI SAIRAM

```
            }
        }
        else if(name=="s3"){
            for(int i=10;i>=1;i--){
                  System.out.println("s3:"+i);
                  Thread.sleep(1000);
            }
        }
       }catch(Exception e){e.printStackTrace();}
    }
}
class MultipleThreads{
    public static void main(String args[]){
            System.out.println("Main Thread Running");
            ThreadEx t1=new ThreadEx("s1");
            ThreadEx t2=new ThreadEx("s2");
            ThreadEx t3=new ThreadEx("s3");
            System.out.println("Main Thread Exiting");
   }
}

/*
F:\BHAVAJAVA\MultiThreading>javac MultipleThreads.java

F:\BHAVAJAVA\MultiThreading>java MultipleThreads
Main Thread Running
Main Thread Exiting
s2:Disturbed: creates beep sound
s3:10
s1:1
s1:2
s3:9
s2:Disturbed: creates beep sound
s1:3
s3:8
s2:Disturbed: creates beep sound
s1:4
s3:7
s2:Disturbed: creates beep sound
s3:6
s1:5
s2:Disturbed: creates beep sound
s1:6
```

```
s3:5
s2:Disturbed: creates beep sound
s1:7
s3:4
s2:Disturbed: creates beep sound
s3:3
s1:8
s2:Disturbed: creates beep sound
s1:9
s3:2
s2:Disturbed: creates beep sound
s3:1
s1:10
s2:Disturbed: creates beep sound
*/
```

Thread Priorities:

- Each thread has a priority. Priorities are represented by a number between 1 and 10.
- In practice, the amount of CPU time that a thread gets often depends on several factors besides its priority.
- A higher-priority thread can also preempt a lower-priority one.
- For instance, when a lower-priority thread is running and a higher-priority thread resumes (from sleeping or waiting on I/O, for example), it will preempt the lower-priority thread.3 constants defined in Thread class:
  1. public static int MIN_PRIORITY
  2. public static int NORM_PRIORITY
  3. public static int MAX_PRIORITY
- Default priority of a thread is 5 (NORM_PRIORITY). The value of MIN_PRIORITY is 1 and the value of MAX_PRIORITY is 10.

```
//PriorityDemo.java
class ThreadPriority extends Thread{
    public void run(){
          System.out.println("Running Thread Name="+Thread.currentThread().getName());
         System.out.println("Running Thread Priority="+Thread.currentThread().getPriority());
     }
}
class PriorityDemo{
      public static void main(String args[]){
           ThreadPriority t1=new ThreadPriority();
           ThreadPriority t2=new ThreadPriority();
           t1.setPriority(Thread.MIN_PRIORITY);
```

```
            t2.setPriority(Thread.MAX_PRIORITY);
            t1.start();
            t2.start();
        }
    }
    /*
    F:\BHAVAJAVA\MultiThreading>javac PriorityDemo.java
    F:\BHAVAJAVA\MultiThreading>java PriorityDemo
    Running Thread Name=Thread-1
    Running Thread Name=Thread-0
    Running Thread Priority=10
    Running Thread Priority=1
    */
```

Using isAlive( ) and join( )

isAlive():

- To determine whether a thread has finished or not call isAlive( ) on the thread.
- This method is defined by Thread class.
- Syntax:
  final boolean isAlive( )
- The isAlive( ) method returns true if the thread upon which it is called is still running. It returns false otherwise.

join() method:

- This method is used to wait a calling thread until its spawned threads completes their execution or the calling thread waiting until the specified thread joins it.
- Syntax:
  final void join( ) throws InterruptedException
  final void join(long milliseconds) throws InterruptedException
  final void join(lng milliseconds,long nanoseconds) throws InterruputedException

```
 //Demonstration of join() and isAlive() methods
//ThreadJoin.java
class ThreadEx extends Thread{
     String name;
    ThreadEx(String name){
       this.name=name;
    }
    public void run(){
      try{
        if(name=="s1"){
```

OHM SRI SAIRAM

```
            for(int i=1;i<=10;i++){
                System.out.println("s1:"+i);
                Thread.sleep(1000);
            }
        }
        else if(name=="s2"){
            for(int i=1;i<=10;i++){
               System.out.println("s2:"+i);
               Thread.sleep(1000);
           }
        }
        else if(name=="s3"){
            for(int i=1;i<=10;i++){
            System.out.println("s3:"+i);
            Thread.sleep(1000);
          }
        }
       }catch(Exception e){e.printStackTrace();}
 }
}
class ThreadJoin{
    public static void main(String args[]){
        System.out.println("Main Thread Running");
        ThreadEx t1=new ThreadEx("s1");
        ThreadEx t2=new ThreadEx("s2");
        ThreadEx t3=new ThreadEx("s3");
        t1.start();
        t2.start();
        t3.start();
        System.out.println("Thread s1 is alive="+t1.isAlive());
        System.out.println("Thread s2 is alive="+t2.isAlive());
        System.out.println("Thread s3 is alive="+t3.isAlive());
        try{
             t1.join();
             t2.join();
             t3.join();
        }catch(InterruptedException e){e.printStackTrace();}
       System.out.println("Thread s1 is alive="+t1.isAlive());
        System.out.println("Thread s2 is alive="+t2.isAlive());
        System.out.println("Thread s3 is alive="+t3.isAlive());
        System.out.println("Main Thread Exiting");
      }
}
```

OHM SRI SAIRAM

```
/*
F:\BHAVAJAVA\MultiThreading>javac ThreadJoin.java
F:\BHAVAJAVA\MultiThreading>java ThreadJoin
Main Thread Running                s2:6
Thread s1 is alive=true            s3:6
s1:1                               s1:7
Thread s1 is alive=true            s3:7
s3:1                               s2:7
s2:1                               s1:8
Thread s1 is alive=true            s2:8
s1:2                               s3:8
s2:2                               s1:9
s3:2                               s3:9
s1:3                               s2:9
s2:3                               s1:10
s3:3                               s2:10
s1:4                               s3:10
s2:4                               Thread s1 is alive=false
s3:4                               Thread s2 is alive=false
s1:5                               Thread s3 is alive=false
s2:5                               Main Thread Exiting
s3:5                               */
s1:6
```

Suspending, Resuming and Stopping Threads:

Thread class provides following methods to suspend, resuming and stopping the execution of threads.

Sr.No.	Method & Description
1	**public void suspend()** This method puts a thread in the suspended state and can be resumed using resume() method.
2	**public void stop()** This method stops a thread completely.
3	**public void resume()** This method resumes a thread, which was suspended using suspend() method.

OHM SRI SAIRAM

```
//Suspendresume.java
import java.awt.*;
class ThreadEx extends Thread{
   String name;
   ThreadEx(String name){
        this.name=name;
   }
   public void run(){
      try{
           for(int i=1;i<=30;i++){
                System.out.println(name+":"+i);
                Thread.sleep(1000);
           }
      }catch(Exception e){
        e.printStackTrace();
      }
      }
}
class SuspendResume{
    public static void main(String args[]){
        System.out.println("Main Thread Running");
        ThreadEx t1=new ThreadEx("s1");
        ThreadEx t2=new ThreadEx("s2");
        ThreadEx t3=new ThreadEx("s3");
        t1.start();
        t2.start();
        t3.start();
        try{
           Thread.sleep(500);
           t2.suspend();
           t3.suspend();
           Thread.sleep(5000);
           t2.resume();
           Thread.sleep(500);
           t3.resume();
           t1.suspend();
           Thread.sleep(500);
           t1.resume();
          t1.join();
          t2.join();
          t3.join();
        }catch(InterruptedException e){e.printStackTrace();}
        System.out.println("Main Thread Exiting");
```

```
        }
    }
    /*

    F:\BHAVAJAVA\MultiThreading>javac SuspendResume.java
    Note: SuspendResume.java uses or overrides a deprecated API.
    Note: Recompile with -Xlint:deprecation for details.
    F:\BHAVAJAVA\MultiThreading>java SuspendResume
    Main Thread Running
    s1:1
    s3:1
    s2:1
    s1:2
    s1:3
    s1:4
    s1:5
    s1:6
    s2:2
    s3:2
    s2:3
    s1:7
    s3:3
    s2:4
    s1:8
    s3:4
    s2:5
    s1:9
    s3:5
    s2:6
    s1:10
    s3:6
    s2:7
    s1:11
    s3:7
    s2:8
    s1:12
    s3:8
    s2:9
    s1:13
    s3:9
    s2:10
    s1:14
    s3:10
```

```
s2:11
s1:15
s3:11
s2:12
s1:16
s3:12
s2:13
s1:17
s3:13
s2:14
s1:18
s3:14
s2:15
s1:19
s3:15
s2:16
s1:20
s3:16
s2:17
s1:21
s3:17
s2:18
s1:22
s3:18
s2:19
s1:23
s3:19
s2:20
s1:24
s3:20
s2:21
s1:25
s3:21
s2:22
s1:26
s3:22
s2:23
s1:27
s3:23
s2:24
s1:28
s3:24
s2:25
```

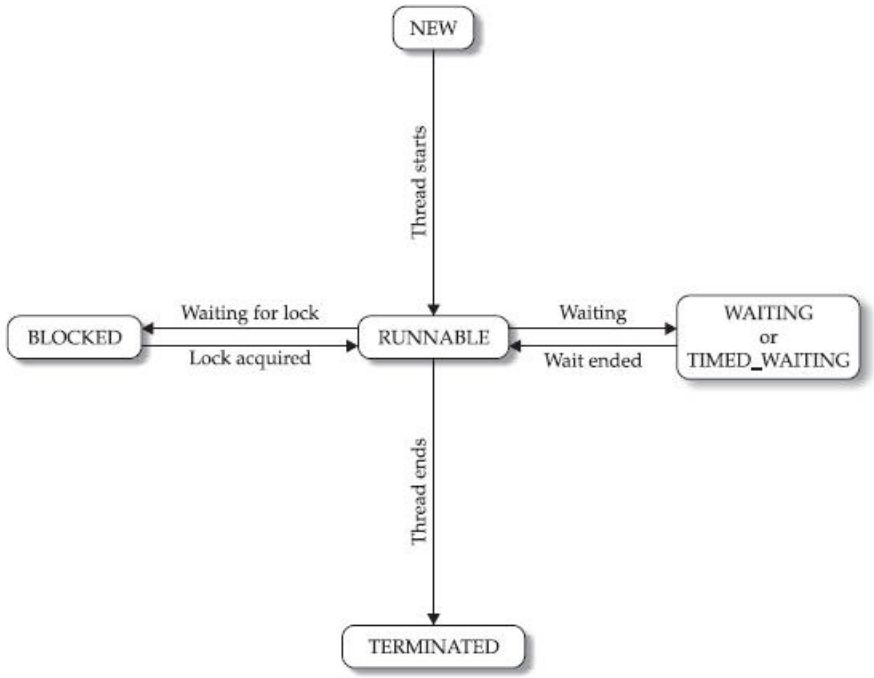
```
s1:29
s3:25
s2:26
s1:30
s3:26
s2:27
s3:27
s2:28
s3:28
s2:29
s3:29
s2:30
s3:30
Main Thread Exiting
*/
```

Obtaining a Thread state:

- A thread can exist in a number of different states.
- You can obtain the current state of a thread by calling the getState( ) method defined by Thread. It is shown here:
  Thread.State getState( )
- It returns a value of type Thread. State that indicates the state of the thread at the time at which the call was made. State is an enumeration defined by Thread.
- The Thread.State enum defined in java.lang package as
  Syntax:
  public static enum Thread.State extends Enum<Thread.State>
- **Constants defined in Thread.State enum:**

Value	State
BLOCKED	A thread that has suspended execution because it is waiting to acquire a lock.
NEW	A thread that has not begun execution.
RUNNABLE	A thread that either is currently executing or will execute when it gains access to the CPU.
TERMINATED	A thread that has completed execution.
TIMED_WAITING	A thread that has suspended execution for a specified period of time, such as when it has called **sleep**( ). This state is also entered when a timeout version of **wait**( ) or **join**( ) is called.
WAITING	A thread that has suspended execution because it is waiting for some action to occur. For example, it is waiting because of a call to a non-timeout version of **wait**( ) or **join**( ).

- The thread states how they are executed in life cycle of the thread demonstrated in following figure.



Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time.
- The process called **synchronization**.
- Synchronization is the concept of the monitor.
- A monitor is an object that is used as a mutually exclusive lock.
- Only one thread can own a monitor at a given time.
- When a thread acquires a lock, it is said to have entered the monitor. All other threads attempting to enter the locked monitor will be suspended until the first thread exits the monitor. These other threads are said to be waiting for the monitor.
- A thread that owns a monitor can reenter the same monitor if it so desires.
- In Java programming synchronization achieved using synchronized keyword.
- In Java programming synchronization achieved in two ways.

1. **synchronized** blocks

Syntax:

```
synchronized(objref){
        //The statements to be synchronized
}
```

OHM SRI SAIRAM

2. **synchronized** methods

```
synchronized return_type method_name(arg_list){
                //block of staments
}
```

The following program demonstrates execution of threads without synchronization.

```
// Without Synchronization
class Resource{
   public void display(String str){
     try{
        System.out.print("["+str);
         Thread.sleep(1000);
        }
         catch(InterruptedException e){
            e.printStackTrace();
          }
          System.out.println("]");
     }
}
class ResourceCaller extends Thread{
      String str;
      Resource r;
      ResourceCaller(String str,Resource r){
         this.str=str;
         this.r=r;
      }
       public void run(){
           r.display(str);
      }
}
class SynchronousNot{
      public static void main(String args[]){
           Resource r=new Resource();
           ResourceCaller t1=new ResourceCaller("Hello",r);
           ResourceCaller t2=new ResourceCaller("Java",r);
           ResourceCaller t3=new ResourceCaller("world",r);
           t1.start();
           t2.start();
           t3.start();
           try{
                t1.join();
                t2.join();
                t3.join();
          }
           catch(InterruptedException e){
                     e.printStackTrace();
           }
```

```
            System.out.println("Main Thread Exiting");
      }
}
/*
F:\BHAVAJAVA\MultiThreading>javac SynchronousNot.java

F:\BHAVAJAVA\MultiThreading>java SynchronousNot
[Java[world[Hello]
]
]
Main Thread Exiting

F:\BHAVAJAVA\MultiThreading>java SynchronousNot
[Java[Hello[world]
]
]
Main Thread Exiting

F:\BHAVAJAVA\MultiThreading>java SynchronousNot
[Hello[world[Java]
]
]
Main Thread Exiting
*/
```

The above program changed by applying synchronization technique:

1. Synchronized block:

```
//Synchronization-Using Synchronized Block
class Resource{
   public void display(String str){
     try{
        System.out.print("["+str);
         Thread.sleep(1000);
        }
         catch(InterruptedException e){
            e.printStackTrace();
          }
          System.out.println("]");
     }
}
class ResourceCaller extends Thread{
      String str;
      Resource r;
      ResourceCaller(String str,Resource r){
```

OHM SRI SAIRAM

```
        this.str=str;
        this.r=r;
    }
     public void run(){
        synchronized(r){
            r.display(str);
        }
    }
}
class SynchronousExBlock{
    public static void main(String args[]){
        Resource r=new Resource();
        ResourceCaller t1=new ResourceCaller("Hello",r);
        ResourceCaller t2=new ResourceCaller("Java",r);
        ResourceCaller t3=new ResourceCaller("world",r);
        t1.start();
        t2.start();
        t3.start();
        try{
            t1.join();
            t2.join();
            t3.join();
        }
        catch(InterruptedException e){
                e.printStackTrace();
        }
         System.out.println("Main Thread Exiting");
    }
}
/*
F:\BHAVAJAVA\MultiThreading>javac SynchronousExBlock.java

F:\BHAVAJAVA\MultiThreading>java SynchronousExBlock
[Hello]
[world]
[Java]
Main Thread Exiting
*/
```

OHM SRI SAIRAM

2. synchronized methods:

```
//Synchronization Methods
class Resource{
   synchronized public void display(String str){
     try{
        System.out.print("["+str);
         Thread.sleep(1000);
        }
         catch(InterruptedException e){
            e.printStackTrace();
          }
          System.out.println("]");
   }
}
class ResourceCaller extends Thread{
      String str;
      Resource r;
      ResourceCaller(String str,Resource r){
         this.str=str;
         this.r=r;
      }
       public void run(){
           r.display(str);
         }
}
class SynchronousMethod{
      public static void main(String args[]){
           Resource r=new Resource();
           ResourceCaller t1=new ResourceCaller("Hello",r);
           ResourceCaller t2=new ResourceCaller("Java",r);
           ResourceCaller t3=new ResourceCaller("world",r);
           t1.start();
           t2.start();
           t3.start();
           try{
                t1.join();
                t2.join();
                t3.join();
          }
           catch(InterruptedException e){
                     e.printStackTrace();
           }
```

```
                System.out.println("Main Thread Exiting");
        }
    }
    /*
    F:\BHAVAJAVA\MultiThreading>javac SynchronousMethod.java
    F:\BHAVAJAVA\MultiThreading>java SynchronousMethod
    [Hello]
    [world]
    [Java]
    Main Thread Exiting
    */
```

Inter-thread Communication:

- Java provides an efficient mechanism for inter process communication through **implicit monitor support.**
- Java includes an elegant inter process communication mechanism via the wait( ), notify( ), and notifyAll( ) methods.
- These methods are implemented as final methods in Object class, so all classes have them.
- All three methods can be called only from within a synchronized context.
- **Syntax:**

  final public void wait( ) throws InterruptedException
    - Tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls notify( ) or notifyAll( ).

  final public void notify( )
    - Wakes up a thread that called wait( ) on the same object.

  final public void notifyAll( )
    - Wakes up all the threads that called wait( ) on the same object. One of the threads will be granted access.

```
//Solving of Producer Consumer Problem
//ProducerConsumer.java
class Resource{
   int n;
   boolean flag=false;
   synchronized int get(){
      while(!flag){
           try{
                wait();
           }
           catch(InterruptedException e){
                e.printStackTrace();
```

```
            }
        }
      System.out.println("Consumer="+n);
      flag=false;
      notify();
      return n;
    }
    synchronized void put(int n){
          while(flag){
               try{
                    wait();
               }
               catch(InterruptedException e){
                    e.printStackTrace();
               }
          }
          this.n=n;
          flag=true;
          System.out.println("Producer="+n);
          notify();
   }
}
class Producer implements Runnable{
      Resource r;
      Producer(Resource r){
               this.r=r;
      }
      public void run(){
          int i=0;
          while(true){
               r.put(i++);
          }
      }
}
class Consumer implements Runnable{
      Resource r;
      Consumer(Resource r){
          this.r=r;
      }
        public void run(){
          while(true){
                    r.get();
          }
```

OHM SRI SAIRAM

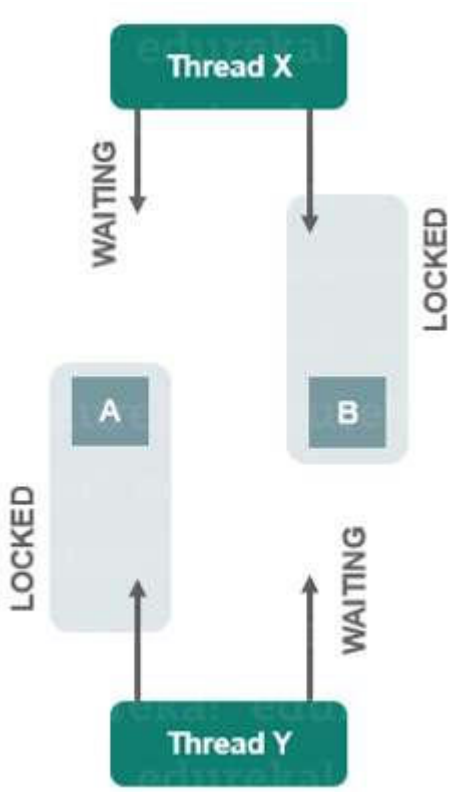
```
        }
    }
    class ProducerConsumer{
        public static void main(String args[]){
            Resource r=new Resource();
            Producer t=new Producer(r);
            Consumer s=new Consumer(r);
            Thread pt=new Thread(t,"Producer");
            Thread ct=new Thread(s,"Consumer");
            pt.start();
           ct.start();
            System.out.println("Press ctrl+C to stop");
        }
    }
    /*
    F:\BHAVAJAVA\MultiThreading>javac ProducerConsumer.java

    F:\BHAVAJAVA\MultiThreading>java ProducerConsumer
    Press ctrl+C to stop
    Producer=0
    Consumer=0
    Producer=1
    Consumer=1
    Producer=2
    Consumer=2
    Producer=3
    Consumer=3
    Producer=4
    Consumer=4
    Producer=5
    Consumer=5
    Producer=6
    Consumer=6
    Producer=7
    Consumer=7
    Producer=8
    Consumer=8
    Producer=9
    Consumer=9
    Producer=10
    Consumer=10
    Producer=11
    Consumer=11
```

```
Producer=12
Consumer=12
Producer=13
Consumer=13
Producer=14
Consumer=14
Producer=15
Consumer=15
Producer=16
Consumer=16
Producer=17
Consumer=17
Producer=18
Consumer=18
*/
```

Deadlock:

- Deadlock is an exceptional condition in multi-tasking or multi-threading execution.
- When two processes or two threads circularly dependent on pair of synchronized objects deadlock is occurred.
- For example, suppose X thread enters the monitor on object A and Y thread enters the monitor on object B. If the thread X tries to call a synchronized object B, it was locked by Thread Y. However, if the thread in Y, in turn, tries to call any synchronized object already locked by Thread X, the thread waits forever, because to access A, it would have to release its own lock on B so that the first thread could complete.

OHM SRI SAIRAM



Using Multithreading:

- The key to utilizing Java's multithreading features effectively is to think concurrently rather than serially.
- For example, when you have two subsystems within a program that can execute concurrently, make them individual threads.

OHM SRI SAIRAM

UNIT-IV

PART-1

GENERICS

- Before JDK 5.0, every property and method defined in the class should be declared with exact type.
- An operation (method) defined in the class applied to specific type of data declared only.
- In JDK 5.0, Sun micro systems introduced a new concept called **Generics**, which fundamentally reconstructs the base JDK libraries and adds mode expensive power.
- With generics, we can define algorithm once, independently of any specific type of data and then the algorithm applied to any wide variety data types at run time of execution of the class without modifying the class definition.
- Generics changed Java in two important ways.
  1. **It added a new syntactical element to the language.**
  2. **It caused changes to many of the classes and methods in the core API.**

What are Generics?

- Generics are parameterized types.
- Parameterized types are important, because they enable you to create classes, interfaces, and methods in which the type of data which they operate is specified as a parameter.
- Using generics, it is possible to create a single class, for example, that automatically works with different types of data.
- A class, interface, or method that operates on a parameterized type is called generic or generic class or generic method.

How Generics adds type safety?

- Pre-generics code, generalized classes, interfaces, and methods used Object references to operate on various types of objects. The problem was that they could not do so with type safety.

OHM SRI SAIRAM

- Generics added the type safety. **It is no longer necessary to explicitly employ casts to translate between Object and the type of data that is actually being operated upon**.
- With generics, all casts are automatic and implicit through **Auto boxing and Auto Unboxing**.
- With Generics, Reusability concept well expanded.

General Form of a Generic Class and its Creation Syntax:

Syntax for declaring a generic class:

```
class class-name<type-param-list > {
    //Members and Methods
}
```

Syntax for creating object for Generic class:

```
class-name<type-arg-list> var-name = new class-name<type-arg-list >(arg-list);
```

- Before practically understanding Generics, we should think about **Autoboxing and Auto Unboxing** which are introduced in JDK 5.0.

Autoboxing and Auto Unboxing:

- **Autoboxing**

  In Autoboxing primitive type is **automatically encapsulated** into equivalent wrapper class object.

  - Wrapper class object is object representation of primitive variable data.
  - Wrapper class object encapsulates primitive java types.

Primitive Type	Wrapper class(included in java.lang package)
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

OHM SRI SAIRAM

- **Auto Unboxing**

  In Auto Unboxing object type is **automatically extracted** into primitive type.

  Example for Autoboxing

```
int i=2;
Integer iob=i;
```

  Example for Auto Unboxing:

```
Integer iob=100;
int i=iob;
```

//Demonstration of Autoboxing and Auto Unboxing

```
class AutoBoxUnBox{
   public static void main(String[] args){
       int ip=9;
       //Auto Boxing
       Integer iob=ip;
       System.out.println("Primitive i="+ip);
       System.out.println("Integer Object iob="+iob);
       //Auto Unboxing
       int xp=iob;
       System.out.println("xp value from iob Auto Unboxing="+xp);
   }
}

/*
F:\BHAVAJAVA\GENERICS>javac AutoBoxUnBox.java
F:\BHAVAJAVA\GENERICS>java AutoBoxUnBox
Primitive i=9
Integer Object iob=9
xp value from iob Auto Unboxing=9
*/
```

OHM SRI SAIRAM

First Generics Sample Program:

```
//GenDemo.java
// A simple generic class.
// Here, T is a type parameter that will be replaced by a real
// type   when an object of type Gen is created.
class Gen<T>{
    T ob;
    Gen(T ob){
         this.ob=ob;
    }
    T getOb(){
         return ob;
    }
    String showType(){
         String type=ob.getClass().getName();
         return type;
    }
}
class GenDemo{
    public static void main(String[] args){
         Gen<Integer> iob=new Gen<Integer>(7);
         System.out.println("Generic is Integer Type...");
         System.out.println("iob value="+iob.getOb());
         System.out.println("iob type="+iob.showType());
         Gen<Double> dob=new Gen<Double>(7.979);
         System.out.println("Generic is Double Type...");
         System.out.println("dob value="+dob.getOb());
         System.out.println("dob type="+dob.showType());
         System.out.println("Generic is String Type...");
         Gen<String> sob=new Gen<String>("AITS is Great College");
         System.out.println("sob value="+sob.getOb());
         System.out.println("sob type="+sob.showType());
    }
}
/*
F:\BHAVAJAVA\GENERICS>javac GenDemo.java
F:\BHAVAJAVA\GENERICS>java GenDemo
Generic is Integer Type...
iob value=7
iob type=java.lang.Integer
Generic is Double Type...
dob value=7.979
```

OHM SRI SAIRAM

```
dob type=java.lang.Double
Generic is String Type...
sob value=AITS is Great College
sob type=java.lang.String
*/
```

Generic Class with Two Type Parameters:

- You can declare more than one type parameter in a generic type.
- To specify two or more type parameters, simply use a comma-separated list.
- Syntax:

```
class ClassName<T1,T2,…> {
    //….
}
```

```
//TwoGenDemo.java
class TwoGen<T1,T2>{
    T1 ob1;
    T2 ob2;
    TwoGen(T1 ob1,T2 ob2){
         this.ob1=ob1;
         this.ob2=ob2;
    }
    T1 getOb1(){
         return ob1;
    }
    T2 getOb2(){
         return ob2;
    }
    String showType1(){
         String type=ob1.getClass().getName();
         return type;
    }
    String showType2(){
         String type=ob2.getClass().getName();
         return type;
    }
}
class TwoGenDemo{
    public static void main(String[] args){
         TwoGen<Integer,String> ob=new TwoGen<Integer,String>(70,"AITS");
         System.out.println("Two Argument Generic ...");
         System.out.println("College Code="+ob.getOb1());
```

OHM SRI SAIRAM

```
            System.out.println("Collge Code type="+ob.showType1());
            System.out.println("College Name="+ob.getOb2());
            System.out.println("Collge Name Type="+ob.showType2());
        }
    }
    /*
    F:\BHAVAJAVA\GENERICS>javac TwoGenDemo.java
    F:\BHAVAJAVA\GENERICS>java TwoGenDemo
    Two Argument Generic ...
    College Code=70
    Collge Code type=java.lang.Integer
    College Name=AITS
    Collge Name Type=java.lang.String
    */
```

Bounded Generics:

- Sometimes there may be helpful to limit the types that can passed through generic type parameter.
- In order to handle such situation java defines boundary for generic type parameter as shown in following way.

```
class ClassName<GenericType extends SuperGenericClassName>{
    //Body of Generic Class
}
```

```
//BoundedGenDemo.java
class Gen<T extends Number>{
    T ob;
    Gen(T ob){
        this.ob=ob;
    }
     T getOb(){
        return ob;
    }
     String showType(){
        String type=ob.getClass().getName();
        return type;
    }
}
class BoundedGenDemo{
    public static void main(String[] args){
        Gen<Integer> iob=new Gen<Integer>(7);
```

OHM SRI SAIRAM

```
        System.out.println("Generic is Integer Type...");
        System.out.println("iob value="+iob.getOb());
        System.out.println("iob type="+iob.showType());
        Gen<Double> dob=new Gen<Double>(7.979);
        System.out.println("Generic is Double Type...");
        System.out.println("dob value="+dob.getOb());
        System.out.println("dob type="+dob.showType());
    }
}
/*
F:\BHAVAJAVA\GENERICS>javac BoundedGenDemo.java
F:\BHAVAJAVA\GENERICS>java BoundedGenDemo
Generic is Integer Type...
iob value=7
iob type=java.lang.Integer
Generic is Double Type...
dob value=7.979
dob type=java.lang.Double

*/
```

The following program demonstrates to find the sum of Generic array which is upper bounded to Number.

```
//BoundedArrayDemo.java
class Gen<T extends Number>{
    T[] nums;
    Gen(T[] nums){
        this.nums=nums;
    }
     //Return Type double works for all Number types
    double getSum(){
        double sum=0;
        for( T t:nums){
            sum=sum+t.doubleValue();
        }
        return sum;
    }
}
class BoundedArrayDemo{
    public static void main(String[] args){
        Integer ia[]={1,3,5,7,9};
        Gen<Integer> gia=new Gen<Integer>(ia);
        System.out.println("Generic is Integer Type...");
        System.out.println("Sum of Integer Array="+gia.getSum());
```

OHM SRI SAIRAM

```
            Double da[]={7.979,5.97,8.9,3.37,4.57};
            Gen<Double> gda=new Gen<Double>(da);
            System.out.println("Generic is Double Type...");
            System.out.println("Sum of Double Array="+gda.getSum());
            Float fa[]={1.2F,5.7F,9.5F,9.9F,2.7F};
            Gen<Float> gfa=new Gen<Float>(fa);
            System.out.println("Generic is Float Type...");
            System.out.println("Sum of Float Array="+gfa.getSum());
        }
    }
    /*
    F:\BHAVAJAVA\GENERICS>javac BoundedArrayDemo.java
    F:\BHAVAJAVA\GENERICS>java BoundedArrayDemo
    Generic is Integer Type...
    Sum of Integer Array=25.0
    Generic is Double Type...
    Sum of Double Array=30.789
    Generic is Float Type...
    Sum of Float Array=28.999999523162842

    */
```

Using Wildcard arguments

- Wildcard generally refers unknown type.
- Wildcards in generics represent an unknown type, that the wildcard argument is specified by the '**?**', and it represents an unknown type.

```
//WildcardDemo.java
class AvgCompare<T extends Number>{
    T[] nums;
    AvgCompare(T[] nums){
        this.nums=nums;
    }
    double average(){
        double sum=0.0;
        for(T x:nums){
            sum=sum+x.doubleValue();
        }
        return (sum/nums.length);
    }
    boolean sameAvg(AvgCompare<?> ob){
        if(average()==ob.average())
            return true;
        else
```

OHM SRI SAIRAM

```
                return false;
        }
    }
    class WildcardDemo{
        public static void main(String[] args){
            Integer inums[]={1,2,3,4,5};
            AvgCompare<Integer> ac=new AvgCompare<Integer>(inums);
            Float fnums[]={0.0F,3.0F,2.5F,3.5F,6.0F};
            AvgCompare<Float> fc=new AvgCompare<Float>(fnums);
            if(ac.sameAvg(fc))
                System.out.println("Same Average...");
            else
                System.out.println("Not Same Average...");
        }
    }

    /*
    F:\BHAVAJAVA\GENERICS>javac WildcardDemo.java

    F:\BHAVAJAVA\GENERICS>java WildcardDemo
    Same Average...

    */
```

Bounded Wildcard Generics

- A bounded wildcard is important when you are creating a generic type that will operate on a class hierarchy.
- Wildcard generics are two types.

  1. Upper boundary wildcard generics
  2. Lower boundary wildcard generics
- **Upper boundary wildcard generics:**

  To establish an upper bound for a wildcard, use the following type of wildcard expression:

  <? extends superclass>

  where **superclass** is the name of the class that serves as the upper bound.
- **Lower boundary wildcard generics:**

  You can also specify a lower bound for a wildcard by adding a **super** clause to a wildcard declaration. Here is its general form:

  <? super subclass>

OHM SRI SAIRAM

In this case, only classes that are super classes of subclass are acceptable arguments. This is an inclusive clause.

- The following program demonstrates Upper boundary wildcard arguments.

```
//BoundedWildCardDemo.java
class WildGen<T>{
   T ob;
   WildGen(T ob){
        this.ob=ob;
   }
    String showType(){
        String type=ob.getClass().getName();
        return type;
   }
   boolean sameType(WildGen<? extends Number> wob){
        if(this.showType()==wob.showType())
             return true;
        else
             return false;
    }
}
class BoundedWildCardGenDemo{
   public static void main(String[] args){
        WildGen<Integer> iob=new WildGen<Integer>(7);
        WildGen<Integer> iob1=new WildGen<Integer>(7);
        WildGen<Double> dob=new WildGen<Double>(7.979);
        WildGen<String> sob=new WildGen<String>("AITS is Great College");
        System.out.println("Types of iob,iob1 are equal?="+iob.sameType(iob1));
        System.out.println("Types of iob,dob are equal?="+iob.sameType(dob));
        System.out.println("Types of sob,dob are equal?="+sob.sameType(dob));
   }
}
/*
F:\BHAVAJAVA\GENERICS>javac BoundedWildCardGenDemo.java

F:\BHAVAJAVA\GENERICS>java BoundedWildCardGenDemo
Types of iob,iob1 are equal?=true
Types of iob,dob are equal?=false
Types of sob,dob are equal?=false
*/
```

OHM SRI SAIRAM

Creating a Generic Method:

- **To define a generic method in non-generic class, specially we create generic methods using following syntax.**
- **Syntax:**

```
class Nongen{
    <T1>returnType methodName(genericArgList) {
            //…
    }
     <T2>returnType methodName(genericArgList) {
            //…
    }
    returnType methodName(argList) {
            //Non generic methods also possible
    }
 }
```

```
//Demonstration of Generic Method in Non Generic class
class NonGen{
    <T>void swap(T a,T b){
        System.out.println("Before Swapping a and b="+a+","+b);
        T temp;
        temp=a;
        a=b;
        b=temp;
        System.out.println("After Swapping a and b="+a+","+b);
    }
}
class GenMethodDemo{
    public static void main(String[] args){
        NonGen ng=new NonGen();
        System.out.println("Invoking 3 forms of swap()generic method");
        System.out.println("Swapping of Strings...");
        ng.swap(new String("Virat"),new String("Dhoni"));
        System.out.println("Swapping of Integers...");
        ng.swap(new Integer(5),new Integer(9));
        System.out.println("Swapping of Floating Point Values...");
        ng.swap(new Float(9.79F),new Float(7.99F));
    }
}
```

OHM SRI SAIRAM

```
/*
F:\BHAVAJAVA\GENERICS>javac GenMethodDemo.java
F:\BHAVAJAVA\GENERICS>java GenMethodDemo
Invoking 3 forms of display()generic method
Swapping of Strings...
Before Swapping a and b=Virat,Dhoni
After Swapping a and b=Dhoni,Virat
Swapping of Integers...
Before Swapping a and b=5,9
After Swapping a and b=9,5
Swapping of Floating Point Values...
Before Swapping a and b=9.79,7.99
After Swapping a and b=7.99,9.79
*/
```

Generic Constructors:

- To define a generic constructor in non-generic class, especially we create generic constructor using following syntax.
- **Syntax:**

```
class Nongen{
        <T1>ConstructorName(genericArgList) {
                //...
        }
    <T2>ConstructorName(genericArgList) {
                //...
        }
   ConstructorName(argList) {
                //Non generic constructor also possible
        }
 }
```

```
//Demostration of Generic Constructor in Non-Generic Constructor
class NonGen{
      double val;
      <T extends Number>NonGen(T ob){
           val=ob.doubleValue();
      }
      void showVal(){
           System.out.println("Double Value val="+val);
      }
}
```

OHM SRI SAIRAM

```
class GenConstructorDemo{
      public static void main(String[] args){
            System.out.println("Calling Generic Constructor as Integer type...");
            NonGen ng1=new NonGen(9);
            ng1.showVal();
            System.out.println("Calling Generic Constructor as Float type...");
            NonGen ng2=new NonGen(9.99F);
            ng2.showVal();
      }
}
/*
F:\BHAVAJAVA\GENERICS>javac GenConstructorDemo.java

F:\BHAVAJAVA\GENERICS>java GenConstructorDemo
Calling Generic Constructor as Integer type...
Double Value val=9.0
Calling Generic Constructor as Float type...
Double Value val=9.989999771118164
*/
```

Generic Class Hierarchies (Generics on Inheritance)

- Generic classes are also used to define inheritance hierarchy as like as non generic classes.
- A generic class can act as a superclass or be a subclass.
- The key difference between generic and non-generic hierarchies is that in a generic hierarchy, any type arguments needed by a generic superclass must be passed up the hierarchy by all subclasses.
- Generic class hierarchies two types.
  1. Generic Super class - Generic Sub class
  2. Non Generic Super class - Generic Sub class

```
//Demonstration of Inheritance from Generic class to Generic class
class GenSuper<T>{
  T ob;
  GenSuper(T ob){
        this.ob=ob;
  }
    T getOb(){
        return ob;
  }
```

OHM SRI SAIRAM

```
}
class GenSub<T,V> extends GenSuper<T>{
  V ob1;
  GenSub(T ob,V ob1){
        super(ob);
        this.ob1=ob1;
  }
  V getOb1(){
        return ob1;
  }
}
class GenGenDemo{
  public static void main(String[] args){
        GenSub<Integer,String> gs=new GenSub<Integer,String>(70,"AITS");
        System.out.println("College Code="+gs.getOb());
        System.out.println("College Name="+gs.getOb1());
  }
}

/*
F:\BHAVAJAVA\GENERICS>javac GenGenDemo.java

F:\BHAVAJAVA\GENERICS>java GenGenDemo
College Code=70
College Name=AITS
*/
//Demonstration of Inheritance from Non Generic class to Generic class
class GenSuper{
  int ob;
  GenSuper(int ob){
        this.ob=ob;
  }
    int getOb(){
        return ob;
  }
}
class GenSub<V> extends GenSuper{
  V ob1;
  GenSub(int ob,V ob1){
        super(ob);
        this.ob1=ob1;
  }
  V getOb1(){
        return ob1;
```

OHM SRI SAIRAM

```
  }
}
class GenNonGenDemo{
  public static void main(String[] args){
       GenSub<String> gs=new GenSub<String>(70,"AITS");
       System.out.println("College Code="+gs.getOb());
       System.out.println("College Name="+gs.getOb1());
  }
}

/*
F:\BHAVAJAVA\GENERICS>javac GenNonGenDemo.java

F:\BHAVAJAVA\GENERICS>java GenNonGenDemo
College Code=70
College Name=AITS
*/
```

Run-Time Type Comparisons With in a Generic Hierarchy

- Run-time type information operator **instanceof** determines if an object is an instance of a class.
- This returns true if an object is of the specified type or can be cast to the
- specified type. The **instanceof** operator **can also be applied to objects of generic classes**.

```
//Demonstration of Inheritance from Generic class to Generic class
class GenSuper<T>{
  T ob;
  GenSuper(T ob){
       this.ob=ob;
  }
    T getOb(){
       return ob;
  }
}
class GenSub<T> extends GenSuper<T>{
  T ob;
  GenSub(T ob){
       super(ob);
  }
  T getOb(){
       return ob;
  }
```

OHM SRI SAIRAM

```
}
class GenGenHeirarchyDemo{
  public static void main(String[] args){
        GenSuper<Integer> gsupi=new GenSuper<Integer>(70);
        GenSub<Integer> gsubi=new GenSub<Integer>(90);
        GenSub<String>  gsubs=new GenSub<String>("AITS");
        System.out.print("gsupi is instance of GenSuper?=");
        System.out.println((gsupi instanceof GenSuper<?>));
        System.out.print("gsubi is instance of GenSuper?=");
        System.out.println((gsubi instanceof GenSuper<?>));
        System.out.print("gsubs is instance of GenSuper?=");
        System.out.println((gsubs instanceof GenSuper<?>));
        System.out.print("gsubi is instance of GenSub?=");
        System.out.println((gsubi instanceof GenSub<?>));
        System.out.print("gsubs is instance of GenSub?=");
        System.out.println((gsubs instanceof GenSub<?>));
  }
}

/*
F:\BHAVAJAVA\GENERICS>javac GenGenHeirarchyDemo.java
F:\BHAVAJAVA\GENERICS>java GenGenHeirarchyDemo
gsupi is instance of GenSuper?=true
gsubi is instance of GenSuper?=true
gsubs is instance of GenSuper?=true
gsubi is instance of GenSub?=true
gsubs is instance of GenSub?=true
*/
```

OHM SRI SAIRAM

Generic Interfaces

- Generic interfaces are specified just like generic classes.
- Syntax:

interface InterfaceName<GenericArgList>{
//Generic and Non Generic method declarations
}

```
//Demonstration of Generic Interfaces
//GenInterDemo.java
interface GenInter<T>{
  void setOb(T ob);
  T getOb();
  String showType();
}

class Gen<T> implements GenInter<T>{
  T ob;
  public void setOb(T ob){
       this.ob=ob;
  }
    public T getOb(){
       return ob;
  }
    public String showType(){
       String type=ob.getClass().getName();
       return type;
  }
}
class GenInterDemo{
  public static void main(String[] args){
       Gen<Integer> iob=new Gen<Integer>();
       System.out.println("Generic is Integer Type...");
       iob.setOb(7);
       System.out.println("iob value="+iob.getOb());
       System.out.println("iob type="+iob.showType());
       Gen<Double> dob=new Gen<Double>();
       dob.setOb(7.979);
       System.out.println("Generic is Double Type...");
       System.out.println("dob value="+dob.getOb());
       System.out.println("dob type="+dob.showType());
       System.out.println("Generic is String Type...");
       Gen<String> sob=new Gen<String>();
       sob.setOb("AITS is Great College");
       System.out.println("sob value="+sob.getOb());
       System.out.println("sob type="+sob.showType());
  }
}
```

OHM SRI SAIRAM

```
/*
F:\BHAVAJAVA\GENERICS>javac GenInterDemo.java

F:\BHAVAJAVA\GENERICS>java GenInterDemo
Generic is Integer Type...
iob value=7
iob type=java.lang.Integer
Generic is Double Type...
dob value=7.979
dob type=java.lang.Double
Generic is String Type...
sob value=AITS is Great College
sob type=java.lang.String
*/
```

Overriding Methods in a Generic Class

- A method in a generic class can be overridden just like any other method.

```
//Demonstration of Inheritance from Generic class to Generic class
class GenSuper<T>{
  T ob;
  GenSuper(T ob){
       this.ob=ob;
  }
    T getOb(){
       return ob;
  }
}
class GenSub<T> extends GenSuper<T>{
  GenSub(T ob){
       super(ob);
  }
  T getOb(){
       return ob;
  }
}
class GenGenOverrideDemo{
  public static void main(String[] args){
       Gen<Integer> gs;
       //Super class Generic reference refering super class generic object
       gs=new Gen<Integer>(70);
       System.out.println("Super class ob="+gs.getOb());
       //Super class Generic reference refering sub class generic object
       gs=new GenSub<Integer>(79);
       System.out.println("Calling sub class ob="+gs.getOb());
  }
}
```

OHM SRI SAIRAM

```
/*
F:\BHAVAJAVA\GENERICS>javac GenGenOverrideDemo.java

F:\BHAVAJAVA\GENERICS>java GenGenOverrideDemo
Super class ob=70
Calling sub class ob=79
*/
```

Raw Types and Legacy Code:

- Generics did not exist prior to JDK 5, it was necessary to provide some transition path from old, pre-generics code.
- At the time of this writing, there is still pre-generics legacy code that must remain both functional and compatible with generics.
- Pre-generics code must be able to work with generics, and generic code must be able to work with pre-generics code.
- **To handle the transition to generics, Java allows a generic class to be used without any type arguments. This creates a raw type for the class. This raw type is compatible with legacy code, which has no knowledge of generics.**
- The main drawback to **using the raw type is that the type safety of generics is lost**.

```
//GenDemo.java
class Gen<T>{
  T ob;
  Gen(T ob){
       this.ob=ob;
  }
    T getOb(){
       return ob;
  }
 }
class GenRawTypeDemo{
  public static void main(String[] args){
       Gen<Integer> iob=new Gen<Integer>(70);
       System.out.println("Calling Generic Type...");
       System.out.println("Generic is Integer Type...");
       System.out.println("iob value="+iob.getOb());
       System.out.println("Generic is String Type...");
       Gen<String> sob=new Gen<String>("AITS is Great College");
       System.out.println("sob value="+sob.getOb());
```

OHM SRI SAIRAM

```
        System.out.println("Calling RawType...");
        Gen gi=new Gen(new Integer(70));
        int i=(Integer)gi.getOb();
        System.out.println("Calling gi in Non generic way i="+i);
        Gen gs=new Gen(new String("AITS is Great College"));
        String s=(String)gs.getOb();
        System.out.println("Calling gs in Non generic way s="+s);
    }
}
/*
F:\BHAVAJAVA\GENERICS>javac GenRawTypeDemo.java
Note: GenRawTypeDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.

F:\BHAVAJAVA\GENERICS>java GenRawTypeDemo
Calling Generic Type...
Generic is Integer Type...
iob value=70
Generic is String Type...
sob value=AITS is Great College
Calling RawType...
Calling gi in Non generic way i=70
Calling gs in Non generic way s=AITS is Great College
*/
```

Erasure

- Generics were added to Java was the need for compatibility with previous versions of Java.
- Generic code had to be compatible with preexisting, non-generic code.
- Any changes to the syntax of the Java language, or to the JVM, had to avoid breaking older code. The way Java implements generics while satisfying this constraint is through the use of erasure.

How erasure works?

1. When your Java code is compiled, all generic type information is removed (erased).
2. Replacing type parameters with their bound type, which is **Object** type if no explicit bound is specified.
3. Applying the appropriate casts (as determined by the type arguments) to maintain type compatibility with the types specified by the type arguments.

OHM SRI SAIRAM

```
//GenErasure.java
class GenErasure<T>{
   T ob;
   GenErasure(T ob){
        this.ob=ob;
   }
    T getOb(){
        return ob;
   }
 }
/*
F:\BHAVAJAVA\GENERICS>javac GenErasure.java

F:\BHAVAJAVA\GENERICS>javap -s GenErasure.class
Compiled from "GenErasure.java"
class GenErasure<T> {
  T ob;
    descriptor: Ljava/lang/Object;
  GenErasure(T);
    descriptor: (Ljava/lang/Object;)V

  T getOb();
    descriptor: ()Ljava/lang/Object;
}
*/
```

Bridge Methods

- The compiler will need to add a bridge method to a class to handle situations in which the type erasure of an overriding method in a subclass does not produce the same erasure as the method in the superclass.
- In this case, a method is generated that uses the type erasure of the superclass and this method calls the method that has the type erasure specified by the subclass. Of course, bridge methods only occur at the bytecode level, are not seen by you, and are not available for your use.

```
//Demonstration of Bridge Methods
class GenSuper<T>{
  T ob;
  GenSuper(T ob){
       this.ob=ob;
  }
```

OHM SRI SAIRAM

```
 T getOb(){
      return ob;
  }
}
class GenSub extends GenSuper<String>{
  GenSub(String ob){
      super(ob);
  }
  String getOb(){
      return ob;
  }
}
class GenBridgeDemo{
  public static void main(String[] args){
      // Create a Gen2 object for Strings.
      GenSub sob = new GenSub("AITS is Great College");
      System.out.println(sob.getOb());
  }
}

/*
F:\BHAVAJAVA\GENERICS>javac GenBridgeDemo.java

F:\BHAVAJAVA\GENERICS>java GenBridgeDemo
AITS is Great College
*/
```

Type Inference with Generics

- Beginning with JDK 7, it is possible to shorten the syntax used to create an instance of a generic type. To begin, consider the following generic class:

```
class MyClass<T, V> {
     T ob1;
     V ob2;
    MyClass(T o1, V o2) {
        ob1 = o1;
        ob2 = o2;
    }
    // ...
}
```

OHM SRI SAIRAM

- Prior to JDK 7, to create an instance of MyClass, you would have needed to use a statement similar to the following:

  MyClass<Integer, String> mcOb =new MyClass<Integer, String>(98, "A String");
- JDK 7 added a syntactic element that lets you avoid the second specification.

  MyClass<Integer, String> mcOb = new MyClass<>(98, "A String");
- When type inference is used, the declaration syntax for a generic reference and instance creation has this general form:

  class-name<type-arg-list > var-name = new class-name <>(cons-arg-list);

```
//GenTypeDemo.java
class Gen<T,V>{
   T ob1;
   V ob2;
   Gen(T ob1,V ob2){
        this.ob1=ob1;
        this.ob2=ob2;
   }
    T getOb1(){
        return ob1;
   }
    V getOb2(){
        return ob2;
   }
}
class GenTypeDemo{
   public static void main(String[] args){
        Gen<Integer,String> gis=new Gen<>(70,"AITS");
        System.out.println("College Code="+gis.getOb1());
        System.out.println("College Name="+gis.getOb2());
   }
}
/*
F:\BHAVAJAVA\GENERICS>javac GenTypeDemo.java

F:\BHAVAJAVA\GENERICS>java GenTypeDemo
College Code=70
College Name=AITS

*/
```

OHM SRI SAIRAM

Some generic Restrictions

There are a few restrictions that you need to keep in mind when using generics.

1. Type Parameters Can't Be Instantiated

```
// Can't create an instance of T.
class Gen<T> {
    T ob;
    Gen() {
     ob = new T(); // Illegal!!!
    }
}
```

2. Restrictions on Static Members

static members can n't be declared with generic type.

```
class Wrong<T> {
    // Wrong, no static variables of type T.
    static T ob;
    // Wrong, no static method can use T.
    static T getob() {
     return ob;
    }
}
```

3. Generic Array Restrictions

There are two important generics restrictions that apply to arrays.

1. You cannot instantiate an array whose element type is a type parameter.
2. You cannot create an array of type-specific generic references.

```
// Generics and arrays.
class Gen<T extends Number> {
      T ob;
      T vals[]; // OK
      Gen(T o, T[] nums) {
          ob = o;
          // This statement is illegal.
          // vals = new T[10]; // can't create an array of T
          // But, this statement is OK.
          vals = nums; // OK to assign reference to existent array
      }
}
class GenArrays {
     public static void main(String args[]) {
```

OHM SRI SAIRAM

```
        Integer n[] = { 1, 2, 3, 4, 5 };
        Gen<Integer> iOb = new Gen<Integer>(50, n);
        // Can't create an array of type-specific generic references.
        // Gen<Integer> gens[] = new Gen<Integer>[10]; // Wrong!
        // This is OK.
        Gen<?> gens[] = new Gen<?>[10]; // OK
    }
}
```

4. Generic Exception Restriction

A generic class cannot extend Throwable. This means that you cannot create generic exception classes.

5. Generics Work Only with Reference Types.

```
Gen<int> intOb = new Gen<int>(53); // Error, can't use primitive type
```

OHM SRI SAIRAM

UNIT-IV

Chapter-II

PART-I

Lambda Expressions

Two Standout features profoundly reshaped the Java Programming language.

1. Generics
2. Lambda Expressions

Lambda expressions added in JDK 8.0.

Lambda expressions (and their related features) significantly enhance Java because of the following primary reasons.

1. They add new syntax elements that increase the expressive power of the language.
2. The addition of lambda expressions resulted in new capabilities being incorporated into the API library. Among these are the parallel processing capabilities of multi-core environments.
3. The syntax of the lambda expressions not only become powerful in java but also added in many other prominent programming languages like Python, C++, C#, Java script languages…

Lambda Expressions Constructs:

Java's implementation of lambda expressions have two constructs.

1. Lambda Expression:
   - A lambda expression is, essentially, an anonymous (that is, unnamed) method.
   - This method is not executed on its own. It is used to implement a method defined by a functional interface.
   - Thus, a lambda expression results in a form of anonymous class.
   - Lambda expressions are also commonly referred to as closures.
2. Functional Interface:
   - A functional interface is an interface that contains one and only one abstract method. This method specifies the intended purpose of the interface.
   - A functional interface typically represents a single action.
   - A functional interface is also referred to as a SAM type, where SAM stands for Single Abstract Method.

Lambda Expression Fundamentals

- The lambda expression introduces a new syntax element and operator into the Java language.
- The new operator, sometimes referred to as the lambda operator or the arrow operator , is ' –>' .

- It divides a lambda expression into two parts.
  1. The left side specifies any parameters required by the lambda expression. (If no parameters are needed, an empty parameter list is used.)
  2. On the right side is the lambda body, which specifies the actions of the lambda expression.
- The –> can be pronounced as "becomes" or "goes to."

Java defines two types of lambda bodies.

1. Single expression Lambda bodies
2. Block of code Lambda bodies.

1. Functional Interfaces and Single expression Lambda bodies:

- A functional interface is an interface that specifies only one abstract method.
- The Syntax for defining functional interface is:

```
interface interface_name{
    returntype  method_name(arg_list);
}
```

- A lambda expression is not executed on its own.
- It forms the implementation of the abstract method defined by the functional interface that specifies its target type.
- When a lambda expression occurs in a target type context, an instance of a class is automatically created that implements the functional interface, with the lambda expression defining the behavior of the abstract method declared by the functional interface.
- When that method is called through the target, the lambda expression is executed.

OHM SRI SAIRAM

```
interface MyNumber{
      double getValue();
}
class SBLambda{
   public static void main(String args[]){
        MyNumber myNum;
        //Here, the lambda expression is simply a constant expression.
         // When it is assigned to myNum, a class instance is constructed.
        // Lambda expression implements the getValue() method in MyNumber.
        myNum = () -> 123.45;
        //Call getValue(),provided by assigned lambda expression.
       System.out.println("A fixed value: " + myNum.getValue());
       myNum = () -> Math.random() * 100;
       System.out.println("A random value: " + myNum.getValue());
       System.out.println("Another random value: " + myNum.getValue());
      // Error due to not compatible with functional interface!
      // myNum = () -> "123.03";
   }
}
/*Input and Output
F:\BHAVAJAVA\Lambda>javac SBLambda.java
F:\BHAVAJAVA\Lambda>java SBLambda
A fixed value: 123.45
A random value: 62.432241993471585
Another random value: 6.632275991721492
*/
```

```
// Demonstrate a lambda expression that takes a parameter.
//LambdaArgument.java
interface NumericTest {
      boolean test(int n);
}
class LambdaArgument {
      public static void main(String args[])
      {
            // A lambda expression that tests if a number is even.
            NumericTest isEven = (n) -> (n % 2)==0;
            if(isEven.test(10)) System.out.println("10 is even");
            if(!isEven.test(9)) System.out.println("9 is not even");
            //Use a lambda expression that tests if a number is non-negative.
            NumericTest isNonNeg = (n) -> n >= 0;
            if(isNonNeg.test(1)) System.out.println("1 is non-negative");
            if(!isNonNeg.test(-1)) System.out.println("-1 is negative");
      }
}
/*Input and Output
F:\BHAVAJAVA\Lambda>javac LambdaArgument.java
F:\BHAVAJAVA\Lambda>java LambdaArgument
10 is even
9 is not even
1 is non-negative
-1 is negative
*/
```

OHM SRI SAIRAM

2. Functional interfaces and Block Lambda Expressions:

In single lambda expressions an expression body, the code on the right side of the lambda operator must consist of a single expression. But in Block lambda expressions, an expression body, the code on the right side of the lambda operator must consist of more than a single expression.

```
// A block lambda that computes the factorial of an int value.
interface NumericFunc {
        int func(int n);
}
class BlockLambda1 {
        public static void main(String args[])
        {
                // This block lambda computes the factorial of an int value.
                NumericFunc factorial = (n) -> {
                        int result = 1;
                        for(int i=1; i <= n; i++)
                                result = i * result;
                        return result;
                 };
                System.out.println("The factorial of 3 is " + factorial.func(3));
                System.out.println("The factorial of 5 is " + factorial.func(5));
        }
}
/*Input and Output
F:\BHAVAJAVA\Lambda>javac BlockLambda1.java
F:\BHAVAJAVA\Lambda>java BlockLambda1
The factoral of 3 is 6
The factoral of 5 is 120
*/
```

In the above program,

- The block lambda declares a variable called result, uses a for loop, and has a return statement.
- When a return statement occurs within a lambda expression, it simply causes a return from the lambda.

OHM SRI SAIRAM

Example Program for one functional interface with multiple Lambda expressions:

```
interface StringFunc {
        String func(String n);
}
class BlockLambda2 {
        public static void main(String args[])
        {
                // This block lambda reverses the characters in a string.
                StringFunc reverse = (str) -> {
                        String result = "";
                        int i;
                        for(i = str.length()-1; i >= 0; i--)
                                result += str.charAt(i);
                        return result;
                };
                System.out.println("Lambda reversed is " +reverse.func("Lambda"));
                StringFunc upper = (str) -> {
                        String result=str.toUpperCase();
                        return result;
                };
                System.out.println("Uppercase String  is " +
                upper.func("annamacharya"));
                StringFunc lower = (str) -> {
                                String result=str.toLowerCase();
                                return result;
                };
                System.out.println("Lowercase String  is " +lower.func("JAVA PROGRAMMING"));
     }
}

/*Input and Output
F:\BHAVAJAVA\Lambda>javac BlockLambda2.java
F:\BHAVAJAVA\Lambda>java BlockLambda2
Lambda reversed is adbmaL
Uppercase String is ANNAMACHARYA
Lowercase String is java programming
*/
```

OHM SRI SAIRAM

Generic Functional interfaces and Lambda Expressions:

The functional interface associated with a lambda expression can be generic. In this case, the target type of the lambda expression is determined, by the type argument or arguments specified when a functional interface reference is declared.

Example Program:

```
interface GenAdd<T>{
      T add(T a,T b);
}
class GenLambda{
      public static void main(String args[])
      {
            // This block lambda reverses the characters in a string.
         GenAdd<Integer> iadd = (a,b) -> {
                                            return (a+b);
                                          };
         System.out.println("Integers addition from Lambda is " +iadd.add(2,3));
         GenAdd<Double> dadd = (a,b) -> {
                                          return (a+b);
                                         };
         System.out.println("Double addition from Lambda is " +dadd.add(2.5,3.7));
         GenAdd<String> sadd = (a,b) -> {
                                          return (a+b);
                                        };
         System.out.println("Strings addition” +sadd.add("India","America"));

   }
}
/*Input and Output
F:\BHAVAJAVA\Lambda>javac GenLambda.java
F:\BHAVAJAVA\Lambda>java GenLambda
Integers addition from Lambda is 5
Double addition from Lambda is 6.2
Strings addition from Lambda is IndiaAmerica
*/
```

OHM SRI SAIRAM

Passing Lambda Expressions as Arguments:

- A lambda expression is passed as an argument to the function.
- To pass a lambda expression as an argument, the type of the parameter receiving the lambda expression argument must be of a functional interface type compatible with the lambda.

```
interface StringFunc {
      String func(String n);
}
class LambdaFunArg {
      static String stringOp(StringFunc sf, String s) {
        return sf.func(s);
      }
      public static void main(String args[])
      {
            // This block lambda reverses the characters in a string.
        String instr="Lambda Expressions are Java 8.0 Feature";
        String outstr;
        outstr = stringOp((str) -> {
                                        String result = "";
                                        int i;
                                        for(i = str.length()-1; i >= 0; i--)
                                              result += str.charAt(i);
                                         return result;
                                  },instr);
         System.out.println("Lambda reversed is: " +outstr);

         outstr = stringOp((str) -> {
                                         String result=str.toUpperCase();
                                         return result;
                                  },instr);
         System.out.println("Uppercase String  is: "+outstr);

         outstr = stringOp((str) -> {
                                         String result=str.toLowerCase();
                                       return result;
                                  },instr);
         System.out.println("Lowercase String  is: "+outstr);
     }
}

/*Input and Output
F:\BHAVAJAVA\Lambda>javac LambdaFunArg.java
F:\BHAVAJAVA\Lambda>java LambdaFunArg
Lambda reversed is: erutaeF 0.8 avaJ era snoisserpxE adbmaL
Uppercase String  is: LAMBDA EXPRESSIONS ARE JAVA 8.0 FEATURE
Lowercase String  is: lambda expressions are java 8.0 feature
*/
```

OHM SRI SAIRAM

Lambda Expressions and Exceptions:

- A lambda expression can throw an exception.
- If it throws a checked exception, then that exception must be compatible with the exception(s) listed in the **throws** clause of the abstract method in the functional interface.

This is illustrated in following example.

- It computes the average of an array of **double** values.
- If a zero-length array is passed, it throws the custom exception **EmptyArrayException**.
- As the example shows, this exception is listed in the **throws** clause of **func( )** declared inside the **DoubleNumericArrayFunc** functional interface.

```
// Throw an exception from a lambda expression.
interface DoubleNumericArrayFunc {
 double func(double[] n) throws EmptyArrayException;
}
class EmptyArrayException extends Exception {
  EmptyArrayException() {
    super("Array Empty");
  }
}
class LambdaExceptionDemo {
 public static void main(String args[]) throws EmptyArrayException
 {
  double[] values = { 1.0, 2.0, 3.0, 4.0 };
  // This block lambda computes the average of an array of doubles.
  DoubleNumericArrayFunc average = (n) -> {
                double sum = 0;
                if(n.length == 0)
                  throw new EmptyArrayException();
                for(int i=0; i < n.length; i++)
                     sum += n[i];
                return sum / n.length;
               };
  System.out.println("The average is " + average.func(values));
  // This causes an exception to be thrown.
  System.out.println("The average is " + average.func(new double[0]));
 }
}

/*Input and Output
F:\BHAVAJAVA\Lambda>java LambdaExceptionDemo
The average is 2.5
Exception in thread "main" EmptyArrayException: Array Empty
        at LambdaExceptionDemo.lambda$main$0(LambdaExceptionDemo.java:18)
        at LambdaExceptionDemo.main(LambdaExceptionDemo.java:25)
*/
```

OHM SRI SAIRAM

Lambda Expressions and Variable Capture:

- Variables defined by the enclosing scope of a lambda expression are accessible within the lambda expression.
- A lambda expression can use an instance or **static** variable defined by its enclosing class.
- When a lambda expression uses a local variable from its enclosing scope, a special situation is created that is referred to as a *variable capture*. Here, a lambda expression may only use local variables that are *effectively final*.
- An effectively final variable is one whose value does not change after it is first assigned. There is no need to explicitly declare such a variable as **final**, although doing so would not be an error.

```
// An example of capturing a local variable from the enclosing scope.
interface MyFunc {
      int func(int n);
}
class VarCapture {
      public static void main(String args[])
      {
            // A local variable that can be captured.
        int num = 10;
         MyFunc myLambda = (n) -> {
         int v = num + n;
        //num++ is illegal.'num' variable captured as final.
        // num++;
            return v;
        };
        // The following line would also cause an error,
        //because it would try to remove the effectively final status from num.
        num++;
     }
}


/*Input and Output
F:\BHAVAJAVA\Lambda>javac VarCapture.java
VarCapture.java:12: error: local variables referenced from a lambda expression must be
final or effectively final
         int v = num + n;
                 ^
1 error
*/
```

OHM SRI SAIRAM

PART-II
METHOD REFERENCES

- An important feature related to lambda expressions is "Method References".
- A method reference provides a way to refer to a method without executing it.
- Method References relates to lambda expressions.

Different types of method references:

1. Method references to static methods
2. Method references to instance methods
3. method references with generics
4. Constructor references

1. Method references to static methods:

- To create a **static** method reference, use this general syntax:

ClassName::methodName

- The class name is separated from the method name by a double colon.
- The **::** is a new separator that has been added to Java by JDK 8 expressly for this purpose.

```
// Demonstrate a method reference for a static method.
interface StringFunc {
      String func(String n);
}
// This class defines a static method called strReverse().
class MyStringOps {
      // A static method that reverses a string.
      static String strReverse(String str) {
            String result = "";
            int i;
            for(i = str.length()-1; i >= 0; i--)
                  result += str.charAt(i);
            return result;
      }
}
class StaticMethRef {
      static String stringOp(StringFunc sf, String s) {
            return sf.func(s);
      }
      public static void main(String args[])
      {
            String inStr = "Lambdas add power to Java";
            String outStr;
            // Here, a method reference to strReverse is passed to stringOp().
            outStr = stringOp(MyStringOps::strReverse, inStr);
            System.out.println("Original string: " + inStr);
            System.out.println("String reversed: " + outStr);
      }
}
```

OHM SRI SAIRAM

/*Input and Output

F:\BHAVAJAVA\Lambda>javac StaticMethRef.java

F:\BHAVAJAVA\Lambda>java StaticMethRef

Original string: Lambdas add power to Java

String reversed: avaJ ot rewop dda sadbmaL

*/

2. Method References to instance methods:

- To create a Method reference to instance methods (means non static methods), use this general syntax:

objRef::methodName

- The objRef is separated from the method name by a double colon.

```
// Demonstrate a method reference to an instance method
interface StringFunc {
      String func(String n);
}
class MyStringOps {
      String strReverse(String str) {
            String result = "";
            int i;
            for(i = str.length()-1; i >= 0; i--)
                  result += str.charAt(i);
                  return result;
      }
}
class MethodRef {
      static String stringOp(StringFunc sf, String s) {
            return sf.func(s);
      }
      public static void main(String args[])
      {
            String inStr = "Lambdas add power to Java";
            String outStr;
            // Create a MyStringOps object.
            MyStringOps strOps = new MyStringOps( );
            outStr = stringOp(strOps::strReverse, inStr);
            System.out.println("Original string: " + inStr);
            System.out.println("String reversed: " + outStr);
      }
}

/*Input and Output
F:\BHAVAJAVA\Lambda>javac MethodRef.java
F:\BHAVAJAVA\Lambda>java MethodRef
Original string: Lambdas add power to Java
String reversed: avaJ ot rewop dda sadbmaL
*/
```

OHM SRI SAIRAM

Method References with Generics:

- We can use method references for generic methods and/or generic classes also.

```
interface MyFunc<T> {
      int func(T[] vals, T v);
}
class MyArrayOps {
      static <T> int countMatching(T[] vals, T v) {
            int count = 0;
            for(int i=0; i < vals.length; i++)
                  if(vals[i] == v) count++;
            return count;
      }
}
class GenMethRef{
      static <T> int myOp(MyFunc<T> f, T[] vals, T v) {
            return f.func(vals, v);
      }
      public static void main(String args[])
      {
            Integer[] vals = { 1, 2, 3, 7, 2, 3, 7, 7, 5 };
            String[] strs = { "One", "Two", "Three", "Two" };
            int count;
            count = myOp(MyArrayOps::<Integer>countMatching, vals,7);
            System.out.println("No of 7's in vals:" + count);
            count = myOp(MyArrayOps::<String>countMatching, strs,"Two");
            System.out.println("No of Two's in vals:" + count);
      }
}

/*Input and Output
F:\BHAVAJAVA\Lambda>javac GenMethRef.java
F:\BHAVAJAVA\Lambda>java GenMethRef
No of 7's in vals:3
No of Two's in vals:2
*/
```

3. Constructor References:

- To create constructor references, use this general syntax:

```
ClassName::new
```

- The ClassName is separated from the new keyword by a double colon.
- Constructor reference and Generic constructor references will understand through following programs.

i) Constructor Reference Example Program:

```
// Demonstrate a Constructor reference.
interface MyFunc {
      MyClass func(int n);
}
class MyClass {
      private int val;
      MyClass() {
            val = 0;
      }
      MyClass(int v) {
            val = v;
      }
      int getVal() {
            return val;
      }
}
class ConRef {
      public static void main(String args[])
      {
            MyFunc myClassCons = MyClass::new;
            MyClass mc = myClassCons.func(100);
            System.out.println("val in mc is " + mc.getVal( ));
      }
}

/*Input and Output
F:\BHAVAJAVA\Lambda>javac ConRef.java
F:\BHAVAJAVA\Lambda>java ConRef
val in mc is 100
*/
```

OHM SRI SAIRAM

ii) Generic Constructor References:

```
// Demonstrate a constructor reference with a generic class.
interface MyFunc<T> {
      MyClass<T> func(T n);
}
class MyClass<T> {
      private T val;
      MyClass(T v) {
            val = v;
      }
      MyClass( ) {
            val = null;
      }
      T getVal() {
            return val;
      }
}
class GenConRef {
      public static void main(String args[])
      {
            // Create a reference to the MyClass<T> constructor.
            MyFunc<Integer> conRef1 = MyClass<Integer>::new;
            // Create an instance of MyClass<T> through constructor reference
            MyClass<Integer> mi = conRef1.func(70);
            System.out.println("College Code: " + mi.getVal( ));
           // Create a reference to the MyClass<T> constructor.
            MyFunc<String> conRef2 = MyClass<String>::new;
            // Create an instance of MyClass<T> through constructor reference
            MyClass<String> ms = conRef2.func("AITS");
            System.out.println("College Name: " + ms.getVal( ));
      }
}

/*Input and Output
F:\BHAVAJAVA\Lambda>javac GenConRef.java
F:\BHAVAJAVA\Lambda>java GenConRef
College Code: 70
College Name: AITS
*/
```

OHM SRI SAIRAM

Predefined Functional Interfaces:

The **java.util.function** package defines several predefined functional interfaces that you can use when creating lambda expressions or method references. They are also widely used throughout the Java API.

Interface	Purpose
UnaryOperator<T>	Apply a unary operation to an object of type **T** and return the result, which is also of type **T**. Its method is called **apply( )**. Syntax: T apply(T val)
BinaryOperator<T>	Apply an operation to two objects of type **T** and return the result, which is also of type **T**. Its method is called **apply( )**. Syntax: T apply(T val)
Consumer<T>	Apply an operation on an object of type **T**. Its method is called **accept( )**. Syntax: void accept(T val)
Supplier<T>	Return an object of type **T**. Its method is called **get( )**. Syntax: T get(T val)
Function<T, R>	Apply an operation to an object of type **T** and return the result as an object of type **R**. Its method is called **apply( )**. Syntax: T apply(T val)
Predicate<T>	Determine if an object of type **T** fulfills some constraint. Return a **boolean** value that indicates the outcome. Its method is called **test( )**. Syntax: boolean apply(T val)

```
// Use the Function built-in functional interface.
// Import the Function interface.
import java.util.function.Function;
class PreFunInt {
      public static void main(String args[])
      {
            // This block lambda computes the factorial of an int value.
            Function<Integer, Integer> factorial = (n) -> {
            int result = 1;
            for(int i=1; i <= n; i++)
                  result = i * result;
                  return result;
            };
            System.out.println("The factoral of 3 is " + factorial.apply(3));
            System.out.println("The factoral of 5 is " + factorial.apply(5));
      }
}

/*
Input and Output
F:\BHAVAJAVA\Lambda>javac PreFunInt.java
F:\BHAVAJAVA\Lambda>java PreFunInt
The factoral of 3 is 6
The factoral of 5 is 120
*/
```

OHM SRI SAIRAM

UNIT-V
java.util package
(The Collection Framework)
PART-I

Java.util contains a large number of classes and interfaces that support well-defined functionality in order to work with data structures. All these well-defined, interfaces and classes called "Collection Framework".

The collections framework was designed to meet several goals, such as –

- The framework had to be high-performance. The implementations for the fundamental collections (dynamic arrays, linked lists, trees, and hashtables) were to be highly efficient.
- The framework had to allow different types of collections to work in a similar manner and with a high degree of interoperability.
- The framework had to extend and/or adapt a collection easily.

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following –

- **Interfaces** – These are abstract data types that represent collections. Interfaces allow collections to be manipulated independently of the details of their representation
- **Implementations, i.e., Classes** – These are the concrete implementations of the collection interfaces. These are reusable data structure classes.
- **Algorithms** – These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

How JDK5 changed the collection framework:

When JDK 5 was released, some fundamental changes were made to the Collections Framework that increased its power and streamlined its use.

1. **Generics Fundamentally Changed the Collections Framework**
   - The addition of generics caused a significant change to the Collections Framework because the entire Collections Framework was reconstructed for it.
   - All collections are now generic, and many of the methods that operate on collections take generic type parameters.
   - Simply put, the addition of generics affected every part of the Collections Framework.
2. **Autoboxing Facilitates the Use of Primitive Types**
   - Autoboxing/unboxing facilitates the storing of primitive types in collections.
   - Before generics, if you wanted to store a primitive value, such as an **int**, in a collection, you had to manually construct corresponding object it into its type wrapper.

When the value was retrieved, it needed to be manually reconstruct primitive type (by using an explicit cast).

- Through autoboxing/unboxing, Java can automatically perform the proper boxing and unboxing needed when storing or retrieving primitive types. There is no need to manually perform these operations.

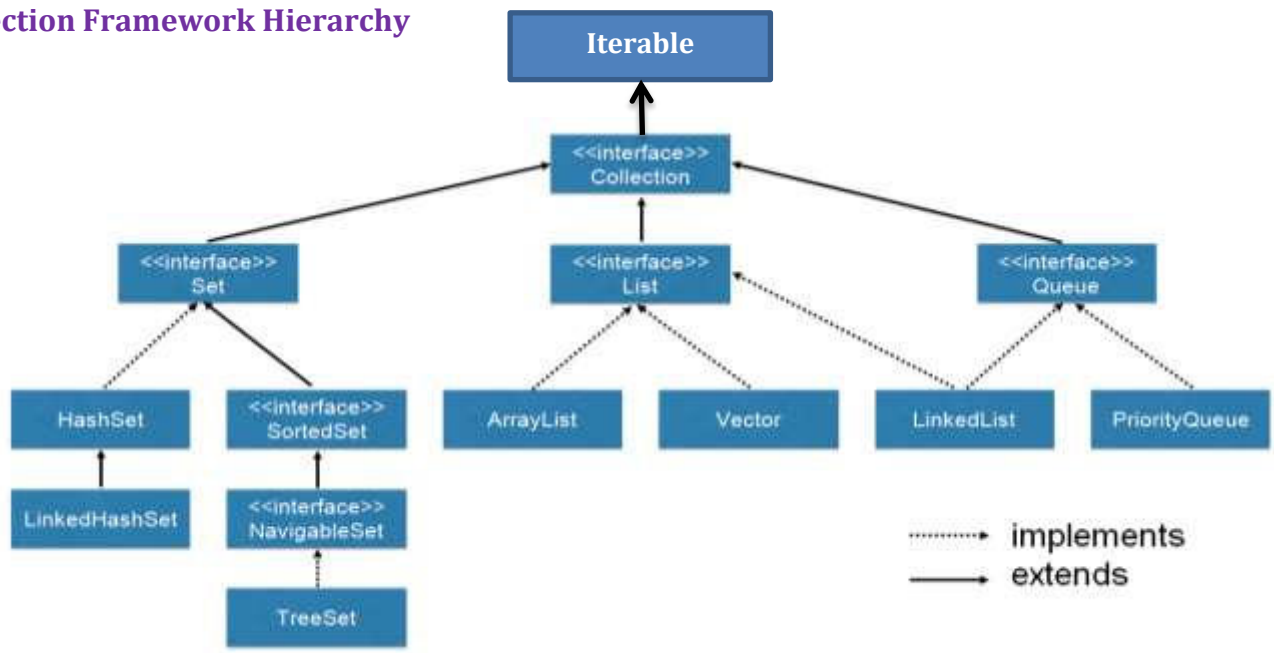
3. The For-Each Style for Loop

- Before Generics, In general elements stored in collections (Built-in data structures in java) are accessed through iterator.
- With the invention of for-each style of for loop in java, it facilitates accessing of collection using for-each style of for loop without need of iterator.

The Collection framework will be understandable, by practicing

- Collection interfaces and their implemented classes,
- Accessing elements of Collections through Itearators and for-each loop
- Collection Algorithms

Collection Framework Hierarchy



Iterable Interface:

The Iterable interface is the root interface for all the collection classes. The Collection interface extends the Iterable interface and therefore all the subclasses of Collection interface also implement the Iterable interface. It contains only one abstract method. i.e.,

```
public interface   Iterable<E> {
    Iterator<E> iterator();
}
```

It returns the iterator over the elements of type T.

Collection Interface:

The Collection interface is a foundation interface for Collection Framework. This Collection interface is extends Iterable interface.

Collection is a generic interface that has this declaration:

```
public interface   Collection<E> extends Iterable<E>{
    //Collection interface methods
}
```

Here, E specifies the type of objects that the collection will hold.

Collection declares the core methods that all collections will have. These methods are summarized in following Table. All collections implement Collection interface, familiarity with its methods is necessary for a clear understanding of the collection framework.

No.	Method	Description
1	public boolean add(E e)	It is used to insert an element in this collection.
2	public boolean addAll(Collection<? extends E> c)	It is used to insert the specified collection elements in the invoking collection.
3	public boolean remove(Object element)	It is used to delete an element from the collection.
4	public boolean removeAll(Collection<?> c)	It is used to delete all the elements of the specified collection from the invoking collection.
5	public boolean retainAll(Collection<?> c)	It is used to delete all the elements of invoking collection except the specified collection.
6	public int size()	It returns the total number of elements in the collection.
7	public void clear()	It removes the total number of elements from the collection.
8	public boolean contains(Object element)	It is used to search an element.
9	public boolean containsAll(Collection<?> c)	It is used to search the specified collection in the collection.
10	public Iterator iterator()	It returns an iterator.
11	public Object[] toArray()	It converts collection into array.
12	public <T> T[] toArray(T[] a)	It converts collection into array. Here, the runtime type of the returned array is that of the specified array.
13	public boolean isEmpty()	It checks if collection is empty.
14	default Spliterator<E> spliterator()	It generates a Spliterator over the specified elements in the collection.

15	public boolean equals(Object element)	It matches two collections.
16	public int hashCode()	It returns the hash code number of the collection.

List interface:

- The **List** interface extends **Collection** and declares the behavior of a collection that stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list, using a zero-based index.
- A list may contain duplicate elements.
- **List** is a generic interface that has this declaration:

```
public interface   List<E> extends Collection<E>{
    //List interface methods
}
```

. Here, **E** specifies the type of objects that the List will hold.

No.	Method	Description
1	void add(int *index*, E *obj*)	Inserts *obj* into the invoking list at the index passed in *index*. Any preexisting elements at or beyond the point of insertion are shifted up. Thus, no elements are overwritten.
2	boolean addAll(int *index*,Collection<? extends E> *c*)	Inserts all elements of *c* into the invoking list at the index passed in *index*. No elements are overwritten. Returns **true** if the invoking list changes and returns **false** otherwise.
3	E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
4	int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
5	int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
6	ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
7	ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified *index*.
8	E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element.
9	E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list. Returns the old value.
10	List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list.

Set Interface:

The **Set** interface defines a set. It extends **Collection** and specifies the behavior of a collection that does not allow duplicate elements. Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set. **It does not specify any additional methods of its own.**

Set is a generic interface that has this declaration:

```
public interface   Set<E> extends Collection<E>{
    //Methods imported from Collection
}
```

. Here, **E** specifies the type of objects that the Set will hold.

SortedSet Interface:

- The **SortedSet** interface extends **Set .**
- This SortedSet keeps elements in sorted ascending order.
- **SortedSet** is a generic interface that has this declaration.

```
public interface   SortedSet<E> extends Set<E>{
    //SortedSet interface methods
}
```

No.	Method	Description
1	E first( )	Returns the first element in the invoking sorted set.
2	SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
3	E last( )	Returns the last element in the invoking sorted set.
4	SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
5	SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

NavigableSet Interface:

- The **NavigableSet** interface extends **SortedSet.**
- **The NavigableSet** declares the behavior of a collection that supports the retrieval of elements based on the closest match to a given value or values.
- **NavigableSet** is a generic interface that has this declaration:

```
public interface   NavigableSet<E> extends SortedSet<E>{
    // NavigableSet interface methods
}
```

Here, **E** specifies the type of objects that the set will hold.

No.	Method	Description
1	E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*.If such an element is found, it is returned. Otherwise, **null** is returned.
2	E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*.If such an element is found, it is returned. Otherwise, **null** is returned.
3	E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
4	E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
5	E pollFirst( )	Returns the first element, removing the element in the process.Because the set is sorted, this is the element with the least value.**null** is returned if the set is empty.
6	E pollLast( )	Returns the last element, removing the element in the process.Because the set is sorted, this is the element with the greatest value. **null** is returned if the set is empty.
7	NavigableSet<E>subSet(E *lowerBound*, boolean *lowIncl*, E *upperBound*, boolean *highIncl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are greater than *lowerBound* and less than *upperBound*. If *lowIncl* is **true**, then an element equal to *lowerBound* is included. If *highIncl* is **true**, then an element equal to *upperBound* is included. The resulting set is backed by the invoking set.
8	NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set.The resulting set is backed by the invoking set.
9	E pollFirst( )	Returns the first element, removing the element in the process.Because the set is sorted, this is the element with the least value.**null** is returned if the set is empty.
10	E pollLast( )	Returns the first element, removing the element in the process.Because the set is sorted, this is the element with the least value.**null** is returned if the set is empty.

OHM SRI SAIRAM

Queue Interface:

- The **Queue** interface extends **Collection.**
- **Queue often follows and acts as** a first-in, first-out list.
- **Queue** is a generic interface that has this declaration:

```
public interface   Queue<E> extends Collection<E>{
    //Queue interface methods
}
```

No.	Method	Description
1	E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty
2	boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
3	E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
4	E poll( )	Returns the element at the head of the queue, and removing the element from queue. It returns **null** if the queue is empty.
5	E remove( )	Removes the element at the head of the queue, returning the element from queue. It throws **NoSuchElementException** if the queue is empty.

Deque Interface:

The **Deque** interface extends **Queue** and declares the behavior of a double-ended queue.

- Double-ended queues can function as standard, first-in, first-out queues or as last-in, firstout stacks.
- **Deque** is a generic interface that has this declaration:

```
public interface   DeQue<E> extends Queue<E>{
    //Queue interface methods
}
```

Here, E specifies the type of objects that the deque will hold.

No.	Method	Description
1.	void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
2.	void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.

3.	E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty
4.	E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
5.	boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
6.	boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
7.	E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
8.	E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
9.	E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
10.	E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
11.	E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.
12.	void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
13.	E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
14.	boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque.Returns **true** if successful and **false** if the deque did not contain *obj*.
15.	E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
16.	boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque.Returns **true** if successful and **false** if the deque did not contain *obj*.

OHM SRI SAIRAM

Collection Classes:

- Collection classes are standard classes that implement class interfaces.
- Some of the classes provide full implementations, some other provide partial implementations as abstract classes.
- Collection classes are not synchronized.

No.	Method	Description
1	AbstractCollection	Implements most of the **Collection** interface.
2	AbstractList	Extends **AbstractCollection** and implements most of the **List** interface.
3	AbstractQueue	Extends **AbstractCollection** and implements parts of the **Queue** interface
4	AbstractSequentialList	Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements.
5	LinkedList	Implements a linked list by extending **AbstractSequentialList**.
6	ArrayList	Implements a dynamic array by extending **AbstractList**.
7	AbstractSet	Extends **AbstractCollection** and implements most of the **Set** interface.
8	EnumSet	Extends **AbstractSet** for use with **enum** elements.
9	HashSet	Extends **AbstractSet** for use with a hash table.
10	LinkedHashSet	Extends **HashSet** to allow insertion-order iterations.
11	PriorityQueue	Extends **AbstractQueue** to support a priority-based queue.
12	TreeSet	Implements a set stored in a tree. Extends **AbstractSet**

Note: For every Collection class you must practice list of constructors, methods and atleast one program.

ArrayList:

- **ArrayList** class extends **AbstractList** and implements the **List** interface
- **ArrayList** is a generic class that has this declaration:

```
class ArrayList<E> implements List<E>{
  //Constructors
  //Methods implemented from List
}
```

- **ArrayList** supports **dynamic arrays** that can grow as needed.In Java, standard arrays are of a fixed length. After arrays are created, they cannot grow or shrink, which means that you must know in advance how many elements an array will hold.
- An **ArrayList** is a variable-length array of object references. That is, an **ArrayList** can dynamically increase or decrease in size.
- Array lists are created with an initial size. When this size is exceeded, the collection is automatically enlarged. When objects are removed, the array can be shrunk(reduced).

OHM SRI SAIRAM

Constructors:

1. ArrayList( )
2. ArrayList(Collection<? extends E> *c*)
3. ArrayList(int *capacity*)

Sample Program demonstrates ArrayList

```
//ArrayListDemo.java
import java.util.*;
class ArrayListDemo {
   public static void main(String args[]) {
         // Create an array list.
         ArrayList<String> al = new ArrayList<String>();
         System.out.println("Initial size of al: " +al.size());
         // Add elements to the array list.
          al.add("C");
         al.add("A");
         al.add("E");
         al.add("B");
         al.add("D");
         al.add("F");
         al.add(1, "A2");
         System.out.println("Size of al after additions: " +al.size());
         // Display the array list.
         System.out.println("Contents of al: " + al);
         // Remove elements from the array list.
         al.remove("F");
         al.remove(2);
         System.out.println("Size of al after deletions: " +al.size());
         System.out.println("Contents of al: " + al);
    }
}
/*
Output
********
F:\Collections>javac ArrayListDemo.java
F:\Collections>java ArrayListDemo
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
*/
```

Although the capacity of an **ArrayList** object increases automatically as objects are stored in it, you can increase the capacity of an **ArrayList** object manually by calling **ensureCapacity( )** method.

Syntax:

void ensureCapacity(int cap)

If you want to reduce the size of the array of an **ArrayList** object so that it is precisely as large as the number of items that it is currently holding, call **trimToSize( )**

Syntax:

void trimToSize()

LinkedList Class:

The **LinkedList** class extends **AbstractSequentialList** and implements the **List**, **Deque**, and **Queue** interfaces. It provides a linked-list data structure.

LinkedList is a generic class that has this declaration:

```
class LinkedList<E> extends AbstractSequentialList<E> implements
                            List<E>,Queue<E>,Deque<E>{
    //Constrctors
   //Methods
}
```

Constructors:

1. LinkedList( )
2. LinkedList(Collection<? extends E> *c*)

```
//LinkedListDemo.java
import java.util.*;
class LinkedListDemo {
   public static void main(String args[]) {
         // Create a linked list.
         LinkedList<String> ll = new LinkedList<String>();
         // Add elements to the linked list.
         ll.add("F");
         ll.add("B");
         ll.add("D");
         ll.add("E");
         ll.add("C");
         ll.addLast("Z");
         ll.addFirst("A");
         ll.add(1, "A2");
         System.out.println("Original contents of ll: " + ll);
         // Remove elements from the linked list.
         ll.remove("F");
         ll.remove(2);
         System.out.println("Contents of ll after deletion: "+ll);
         // Remove first and last elements.
         ll.removeFirst();
         ll.removeLast();
         System.out.println("ll after deleting first and last: "+ll);
   }
}
```

```
/*Output
 ***********
F:\Collections>javac LinkedListDemo.java
F:\Collections>java LinkedListDemo
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
*/
```

HashSet class:

- **HashSet** extends **AbstractSet** and implements the **Set** interface.
- It creates a collection that uses a hash table for storage.
- **HashSet** is a generic class that has this declaration:

class HashSet<E> extends **AbstractSet<E> implements Set<E>{**
//Constrctors
//Methods
}

Constructors :

1. HashSet( )
2. HashSet(Collection<? extends E> c)
3. HashSet(int *capacity*)
4. HashSet(int *capacity*, float *fillRatio*)

The fourth form initializes both the capacity and the fill ratio (also called *load capacity*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded.
For constructors that do not take a fill ratio, 0.75 is used.

```
// HashSetDemo.java
import java.util.*;
class HashSetDemo {
   public static void main(String args[]) {
         HashSet<String> hs = new HashSet<String>();
         // Add elements to the hash set.
         hs.add("Beta");
         hs.add("Alpha");
         hs.add("Eta");
         hs.add("Gamma");
         hs.add("Epsilon");
         hs.add("Omega");
         System.out.println(hs);
   }
}
```

```
/*
Output
***********
F:\Collections>javac HashSetDemo.java
F:\Collections>java HashSetDemo
[Gamma, Eta, Alpha, Epsilon, Omega, Beta]
*/
```

Note: It is important to note that **HashSet** does not guarantee the order of its elements, because the process of hashing doesn't usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as **TreeSet**, is a better choice.

LinkedHashSet Class:

- The **LinkedHashSet** class extends **HashSet** and adds no members of its own.
- It is a generic class that has this declaration.

```
class LinkedHashSet<E> extends HashSet<E> {
    //Constrctors
   //Methods
}
```

Here, **E** specifies the type of objects that the set will hold.

Constructors:

1. LinkedHashSet( )
2. LinkedHashSet(Collection<? extends E> c)
3. LinkedHashSet(int *capacity*)
4. LinkedHashSet(int *capacity*, float *fillRatio*)

```
// LinkedHashSetDemo.java
import java.util.*;
class LinkedHashSetDemo {
      public static void main(String args[]) {
            // Create a Linked hash set.
            LinkedHashSet<String> lhs = new LinkedHashSet<String>();
            // Add elements to the Linked hash set.
            lhs.add("Beta");
            lhs.add("Alpha");
            lhs.add("Eta");
            lhs.add("Gamma");
            lhs.add("Epsilon");
            lhs.add("Omega");
            System.out.println(lhs);
      }

}
```

OHM SRI SAIRAM

```
/*
Output
***********
F:\Collections>javac LinkedHaslhsetDemo.java

F:\Collections>java LinkedHaslhsetDemo

[Beta, Alpha, Eta, Gamma, Epsilon, Omega]

*/
```

Note: **LinkedHashSet** maintains a linked list of the entries in the set, in the order in which they were inserted. This allows insertion-order iteration over the set.

TreeSet Class:

- **TreeSet** extends **AbstractSet** and implements the **NavigableSet** interface.
- It creates a collection that uses a tree for storage. Objects are stored in sorted, ascending order.
- Access and retrieval times are quite fast, which makes **TreeSet** an excellent choice when storing large amounts of sorted information that must be found quickly.
- **TreeSet** is a generic class that has this declaration:

class TreeSet<E> extends **AbstractSet<E> implements NavigableSet<E> {**
//Constrctors
//Methods
}

Constructors:
TreeSet( )
TreeSet(Collection<? extends E> *c*)
TreeSet(SortedSet<E> *ss*)

```
// Demonstrate TreeSet.
import java.util.*;
class TreeSetDemo {
      public static void main(String args[]) {
            // Create a TreeSet.
            TreeSet<String> ts = new TreeSet<String>();
            // Add elements to the Linked hash set.
            ts.add("Beta");
            ts.add("Alpha");
            ts.add("Eta");
            ts.add("Gamma");
            ts.add("Epsilon");
            ts.add("Omega");
            System.out.println(ts);
      }
}
```

```
/*
Output
***********
F:\Collections>javac TreeSetDemo.java
F:\Collections>java TreeSetDemo
[Alpha, Beta, Epsilon, Eta, Gamma, Omega]
*/
```

EnumSet Class:

- **EnumSet** extends **AbstractSet** and implements **Set**.
- It is specifically for use with elements of an **enum** type.
- EnumSet does n't define any constructors.
- It is a generic class that has this declaration.

```
public abstract class EnumSet<E extends Enum<E>> extends AbstractSet<E>{
    //Methods
}
```

Here E specifies the elements.

No.	Method
1	static <E extends Enum<E>>EnumSet<E> allOf(Class<E> *t*) Creates an **EnumSet** that contains the elements in the enumeration specified by *t*.
2	static <E extends Enum<E>>EnumSet<E> of(E *v*) Creates an **EnumSet** that contains *v*.
3	static <E extends Enum<E>>EnumSet<E> of(E *v1*, E *v2*) Creates an **EnumSet** that contains *v1* and *v2*.
4	static <E extends Enum<E>>EnumSet<E> copyOf(Collection<E> *c*) Creates an **EnumSet** from the elements stored in *c*.
5	static <E extends Enum<E>>EnumSet<E> of(E *v1*, E *v2*, E *v3*) Creates an **EnumSet** that contains *v1* through *v3*.
6	static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> *e*) Creates an **EnumSet** that is comprised of those elements not stored in *e*.
7	static <E extends Enum<E>> EnumSet<E> range(E *start*, E *end*) Creates an **EnumSet** that contains the elements in the range specified by *start* and *end*.

```
import java.util.EnumSet;
enum Rainbow{
    RED,ORANGE,YELLOW,GREEN,BLUE,INDIGO,VIOLET
};
public class EnumSetDemo{
    public static void main(String[] args){
        // Creating a set
        EnumSet<Rainbow> set1, set2, set3, set4;
        // Adding elements
        set1 = EnumSet.of(Rainbow.RED, Rainbow.GREEN,
                          Rainbow.BLUE);
        set2 = EnumSet.complementOf(set1);
        set3 = EnumSet.allOf(Rainbow.class);
        set4 = EnumSet.range(Rainbow.RED, Rainbow.INDIGO);
        System.out.println("Set 1: " + set1);
        System.out.println("Set 2: " + set2);
        System.out.println("Set 3: " + set3);
        System.out.println("Set 4: " + set4);
    }
}
```

```
/*
Output
********
F:\Collections>javac EnumSetDemo.java
F:\Collections>java EnumSetDemo
Set 1: [RED, GREEN, BLUE]
Set 2: [ORANGE, YELLOW, INDIGO, VIOLET]
Set 3: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET]
Set 4: [RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO]
*/
```

OHM SRI SAIRAM

UNIT-V
java.util package
(The Collection Framework)

In Part-I Introduction to Collection framework, Collection interfaces and Collection classes covered.

PART-II

Iterators:

Elements in the Collection are accessed through Iterators.
In java.util package two different iterators available.

1. **Iterator**
   Iterator enables you to traversal of a collection for obtaining or removing elements.
2. **ListIterator**
   ListIterator extends Iterator to allow bidirectional traversal of a list, and the modification of elements.

Iterator and **ListIterator** are generic interfaces which are declared as shown here:

Iterator :

Syntax:

```
interface Iterator<E>{
    //Methods
}
```

Here, **E** specifies the type of objects being iterated.

Iterator Methods:

No.	Method	Description
1	boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
2	E next( )	Returns the next element. Throws **NoSuchElementException.**if there is not a next element.
3	default void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**. The default version throws an **UnsupportedOperationException**.

OHM SRI SAIRAM

How to use Iterator:

In general, to use an iterator to cycle through the contents of a collection,follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.
2. Define a loop that makes a call to **hasNext( ) method**. This loop iterates as long as **hasNext( )** returns **true**.
3. Within the loop, obtain each element by calling **next( )**.

Example Program:

```
// Demonstrate iterators.
import java.util.*;
class IterDemo{
      public static void main(String args[]){
            // Create an array list.
            ArrayList<String> al = new ArrayList<String>();
            // Add elements to the array list.
            al.add("C");
            al.add("A");
            al.add("E");
            al.add("B");
            al.add("D");
            al.add("F");
            // Use iterator to display contents of al.
            System.out.print("Original contents of al: ");
            Iterator<String> itr = al.iterator();
            while(itr.hasNext()) {
                  String element = itr.next();
                  System.out.print(element + " ");
            }
          System.out.println();
      }
}

/*
Input and Output
****************
F:\BHAVAJAVA\Collections\ITERATORS>javac IterDemo.java
F:\BHAVAJAVA\Collections\ITERATORS>java IterDemo
Original contents of al: C A E B D F
*/
```

OHM SRI SAIRAM

ListIterator:

Syntax:

```
interface ListIterator<E> extends Iterator{
    //Methods
}
```

Here, **E** specifies the type of objects being iterated.

Methods:

No.	Method	Description
1	boolean hasNext( )	Returns **true** if there is a next element. Otherwise, returns **false**.
2	boolean hasPrevious( )	Returns **true** if there is a previous element. Otherwise,returns **false**.
3	E next( )	Returns the next element. A **NoSuchElementException** is thrown if there is not a next element.
4	E previous( )	Retu[rns the previous element. A **NoSuchElementException** is thrown if there is not a previous element.
5	int nextIndex( )	Returns the index of the next element. If there is not a next element, returns the size of the list.
6	int previousIndex( )	Returns the index of the previous element. If there is not a previous element, returns –1.
7	void add(E *obj*)	Inserts *obj* into the list in front of the element that will be returned by the next call to **next( )**.
8	void remove( )	Removes the current element from the list. An **IllegalStateException** is thrown if **remove( )** is called before **next( )** or **previous( )** is invoked.
9	void set(E *obj*)	Assigns *obj* to the current element. This is the element last returned by a call to either **next( )** or **previous( )**.

OHM SRI SAIRAM

```
// Demonstrate ListIterators.
import java.util.*;
class ListIterDemo{
      public static void main(String args[]){
            // Create an array list.
            ArrayList<String> al = new ArrayList<String>();
            // Add elements to the array list.
            al.add("C");
            al.add("A");
            al.add("E");
            al.add("B");
            al.add("D");
            al.add("F");
            //create ListIterator object
            ListIterator<String> litr = al.listIterator();
            System.out.print("List Elements:" );
            while(litr.hasNext()) {
                   String element = litr.next();
                   System.out.print(element + " ");
            }
            System.out.println();
            litr = al.listIterator();
            //Modifify ArrayList elements using set method
            while(litr.hasNext()) {
                  String element = litr.next();
                  litr.set(element + "+");
            }
            litr = al.listIterator();
            System.out.print("Modified contents of al: ");
            while(litr.hasNext()) {
                  String element = litr.next();
                  System.out.print(element + " ");
            }
            System.out.println();
            // Now, display the list backwards.
            System.out.print("Modified list backwards: ");
            while(litr.hasPrevious()) {
                  String element = litr.previous();
                  System.out.print(element + " ");
            }
            System.out.println();
      }
}
```

OHM SRI SAIRAM

```
/*
F:\BHAVAJAVA\Collections\ITERATORS>javac ListIterDemo.java
F:\BHAVAJAVA\Collections\ITERATORS>java ListIterDemo
List Elements:C A E B D F
Modified contents of al: C+ A+ E+ B+ D+ F+
Modified list backwards: F+ D+ B+ E+ A+ C+
*/
```

For-Each Alternative to Iterators:

If you won't be modifying the contents of a collection or obtaining elements in reverse order, **then the for-each version of the for loop is often a more convenient alternative to traverse a collection than is using an iterator.**

```
// Demonstrate Iterating ArrayList elements Using ForEach Loop.
import java.util.*;
class ForDemo{
     public static void main(String args[]){
                // Create an array list for integers.
                 ArrayList<Integer> al = new ArrayList<Integer>();
                 // Add values to the array list.
                 al.add(1);
                 al.add(2);
                 al.add(3);
                 al.add(4);
                 al.add(5);
                 // Use for loop to display the values.
                 System.out.print("Contents of al: ");
                 for(int ele : al)
                 System.out.print(ele + " ");
                 System.out.println();
                int sum = 0;
               for(int v : al)
                    sum += v;
               System.out.println("Sum of values: " + sum);
     }
}
/*
Input and Output
F:\BHAVAJAVA\Collections\ITERATORS>javac ForDemo.java
F:\BHAVAJAVA\Collections\ITERATORS>java ForDemo
Contents of al: 1 2 3 4 5
Sum of values: 15
*/
```

OHM SRI SAIRAM

Storing User-Defined Classes in Collections:

Collections are not limited to the storage of built-in objects. .The power of collections is that they can store any type of object, including objects of classes that you create.

For example, consider the following example that uses a **Linked List** to store **student** information.

```
import java.util.*;
class Student{
      String roll;
      String name;
      int age;
      String gender;
      String address;
      Student(String roll,String name,int age,String gender,String address){
            this.roll=roll;
            this.name=name;
            this.age=age;
            this.gender=gender;
            this.address=address;
   }
   public String toString(){
            return
      ("Roll Number:"+roll+"\n"+"Name:"+name+"\n"+"Age:"+age+"\n"+"Gender:"
             +gender+"\n"+"Address:"+address);
   }
}
class StudentList{
      public static void main(String args[]){
            ArrayList<Student> as=new ArrayList<Student>();
        Student s1,s2,s3,s4,s5;
            s1=new Student("501","RAJESH",20,"MALE","RAJAMPET");
            s2=new Student("502","RAJESWARI",18,"FEMALE","RAJAMPET");
            s3=new Student("503","ESWAR",18,"MALE","VARANASI");
            s4=new Student("504","ESWARI",18,"FEMALE","VARRANASI");
            s5=new Student("505","COMPUTER",18,"MALE","EVERY WHERE");
            as.add(s1);
            as.add(s2);
            as.add(s3);
            as.add(s4);
            as.add(s5);
            for(Student s:as){
                  System.out.println("-------------------------------");
                  System.out.println(s);
          }
      }
}
```

OHM SRI SAIRAM

/*Output

```
F:\BHAVAJAVA\Collections>javac StudentList.java

F:\BHAVAJAVA\Collections>java StudentList
-------------------------------
Roll Number:501
Name:RAJESH
Age:20
Gender:MALE
Address:RAJAMPET
-------------------------------
Roll Number:502
Name:RAJESWARI
Age:18
Gender:FEMALE
Address:RAJAMPET
-------------------------------
Roll Number:503
Name:ESWAR
Age:18
Gender:MALE
Address:VARANASI
-------------------------------
Roll Number:504
Name:ESWARI
Age:18
Gender:FEMALE
Address:VARRANASI
-------------------------------
Roll Number:505
Name:COMPUTER
Age:18
Gender:MALE
Address:EVERY WHERE
*/
```

OHM SRI SAIRAM

Working with Maps:

- Maps similar to dictionaries behavior, in which each element associated with kay-value pair.
- A *map* is an object that stores associations between keys and values, or *key/value pairs*.
- Given a key, you can find its value. Both keys and values are objects.
- The keys must be unique, but the values may be duplicated.
- Maps don't implement the Iterable interface. This means that you cannot cycle through a map using a for-each style for loop.

Map interfaces:

Interface Name	Description
Map	Maps unique keys to values.
Map.Entry	Describes an element (a key/value pair) in a map. This is an inner class of **Map**.
NavigableMap	NavigableMap Extends **SortedMap** to handle the retrieval of entries based on closest-match searches.
SortedMap	Extends **Map** so that the keys are maintained in ascending order.

1. Map Interface

- The **Map** interface maps unique keys to values. A *key* is an object that you use to retrieve a value.
- Given a key and a value, you can store the value in a **Map** object. After the value is stored, you can retrieve it by using **its** key.
- **Map** is generic and is declared as shown here:

```
interface Map<K,V>{
    //Methods
}
```

Here, K specifies the type of keys, and V specifies the type of values.

No.	Method	Description
1	void clear( )	Removes all key/value pairs from the invoking map.
2	boolean containsKey(Object *k*)	Returns **true** if the invoking map contains *k* as a key. Otherwise, returns **false**.
3	boolean containsValue(Object *v*)	Returns **true** if the map contains *v* as a value. Otherwise, returns **false**.

OHM SRI SAIRAM

4	Set<Map.Entry<K, V>> entrySet( )	Returns a Set that contains the entries in the map. The set contains objects of type Map.Entry. Thus, this method provides a set-view of the invoking map.
5	boolean equals(Object *obj*)	Returns true if obj is a Map and contains the same entries. Otherwise, returns false.
6	V get(Object *k*)	Returns the value associated with the key k. Returns null if the key is not found.
7	int hashCode( )	Returns the hash code for the invoking map.
8	boolean isEmpty( )	Returns true if the invoking map is empty. Otherwise, returns false.
9	Set<K> keySet( )	Returns a Set that contains the keys in the invoking map. This method provides a set-view of the keys in the invoking map.
10	V put(K *k*, V *v*)	Puts an entry in the invoking map, overwriting any previous value associated with the key. The key and value are k and v, respectively. Returns null if the key did not already exist. Otherwise, the previous value linked to the key is returned.
11	void putAll(Map<? extends K,? extends V> *m*)	Puts all the entries from m into this map.
12.	V remove(Object *k*)	Removes the entry whose key equals k
13.	int size( )	Returns the number of key/value pairs in the map.
14.	Collection<V> values( )	Returns a collection containing the values in the map. This method provides a collection-view of the values in the map.

2. Map.Entry Interface

- The **Map.Entry** interface enables you to work with a map entry.
- Recall that the **entrySet( )** method declared by the **Map** interface returns a **Set** containing the map entries.
- Each of these set elements is a Map.Entry object.

Map.Entry is generic and is declared like this:

```
interface Map.Entry<K,V>{
    //Methods
}
```

Here, K specifies the type of keys, and V specifies the type of values.

No.	Method	Description
1	boolean equals(Object *obj*)	Returns true if obj is a Map.Entry whose key and value are equal to that of the invoking object.
2	K getKey( )	Returns the key for this map entry.
3	V getValue( )	Returns the value for this map entry.
4	int hashCode( )	Returns the hash code for this map entry.
5	V setValue(V *v*)	Sets the value for this map entry to v. A ClassCastException is thrown if v is not the correct type for the map. An IllegalArgumentException is thrown if there is a problem with v.

3. SortedMap Interface

- The SortedMap interface extends Map.
- It ensures that the entries are maintained in ascending order based on the keys.
- Sorted maps allow very efficient manipulations of submaps (or also called subsets of a map).
- SortedMap is generic and is declared as shown here:

```
interface SortedMap<K,V> extends Map{
    //Methods
}
```

Here, K specifies the type of keys, and V specifies the type of values.

- Several methods throw a NoSuchElementException when no items are in the invoking map.
- A ClassCastException is thrown when an object is incompatible with the elements in a map.
- A NullPointerException is thrown if an attempt is made to use a null object when null is not allowed in the map. An IllegalArgumentException is thrown if an invalid argument is used.

No.	Method	Description
1	SortedMap<K, V> subMap(K *start*, K *end*)	Returns a map containing those entries with keys that are greater than or equal to start and less than end.
2	K firstKey( )	Returns the first key in the invoking map.

3	K lastKey( )	Returns the last key in the invoking map.
4	SortedMap<K, V> headMap(K end)	Returns a sorted map for those map entries with keys that are less than end.
5	SortedMap<K, V> tailMap(K start)	Returns a map containing those entries with keys that are greater than or equal to *start*.

4. NavigableMap interface

- The NavigableMap interface extends SortedMap and declares the behavior of a map that supports the retrieval of entries based on the closest match to a given key or keys.
- NavigableMap is a generic interface that has this declaration:

```
interface NavigableMap<K,V> extends SortedMap{
    //Methods
}
```

Here, K specifies the type of the keys, and V specifies the type of the values associated with the keys.

No.	Method	Description
1	**Map.Entry<K,V> firstEntry( )**	Returns the first entry in the map. This is the entry with the least key.
2	**Map.Entry<K,V> lastEntry( )**	Returns the last entry in the map. This is the entry with the largest key.
3	**Map.Entry<K,V> ceilingEntry(K *obj*)**	Searches the map for the smallest key *k* such that *k >= obj*. If such a key is found, its entry is returned.Otherwise, **null** is returned.
4	**Map.Entry<K,V> floorEntry(K *obj*)**	Searches the map for the largest key k such that k <= obj. If such a key is found, its entry is returned.Otherwise, null is returned.
5	**Map.Entry<K,V> higherEntry(K *obj*)**	Searches the set for the largest key *k* such that *k > obj*. If such a key is found, its entry is returned.Otherwise, **null** is returned.
6	**NavigableMap<K,V> headMap(K *upperBound*, boolean *incl*)**	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included.
7	**NavigableMap<K,V> tailMap(K lowerBound,boolean incl)**	Returns a **NavigableMap** that includes all entries from the invoking map that have keys that are greater than *lowerBound*.If *incl* is **true**, then an element equal to *lowerBound* is included.
8	**NavigableMap<K,V> subMap(K lowerBound,boolean lowIncl,K upperBound, boolean highIncl)**	

Returns a NavigableMap that includes all entries from the invoking map that have keys that are lowerBound is included. If highIncl is true,then an greater than lowerBound and less than upperBound. If lowIncl is true, then an element equal to element equal to highIncl is included..

Map classes

- Several classes provide implementations of the map interfaces.
- **AbstractMap** is a superclass for all concrete map implementations.

No.	Class Name	Description
1	AbstractMap	Implements most of the **Map** interface.
2	HashMap	Extends **AbstractMap** to use a hash table.
3	LinkedHashMap	Extends **HashMap** to allow insertion-order iterations.
4	TreeMap	Extends **AbstractMap** to use a tree.
5	EnumMap	Extends **AbstractMap** for use with **enum** keys.

1. HashMap Class

- **HashMap** class extends **AbstractMap** and implements the **Map** interface.
- It uses a hash table to store the map.
- This allows the execution time of **get( )** and **put( )** to remain constant even for large sets.
- **HashMap** is a generic class that has this declaration:

```
class  HashMap<K,V> extends AbstractMap implements Map{
    //Methods
}
```

Here, **K** specifies the type of keys, and **V** specifies the type of values.

Constructors:

HashMap( )
HashMap(Map<? extends K, ? extends V> m)
HashMap(int capacity)
HashMap(int capacity, float fillRatio)

Note: 1. The default capacity is 16. The default fill ratio is 0.75.
2. You should note that a hash map does not guarantee the order of its elements.
3. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

OHM SRI SAIRAM

```
import java.util.*;
class HashMapDemo {
      public static void main(String args[]) {
            // Create a hash map.
            HashMap<String, Double> hm = new HashMap<String, Double>();
            // Put elements to the map
            hm.put("Rajesh", new Double(100.00));
            hm.put("Eswar", new Double(99.99));
            hm.put("Hari", new Double(97.27));
            hm.put("Siva", new Double(75.22));
            hm.put("Satya", new Double(100.100));
            // Get a set of the entries.
            Set<Map.Entry<String, Double>> set = hm.entrySet();
            // Display the set.
            System.out.println("Display Elements from HashMap");
            for(Map.Entry<String, Double> me : set) {
                  System.out.print(me.getKey() + ": ");
                  System.out.println(me.getValue());
      }
      System.out.println();
    }
}
/*
Input and Output
****************
F:\BHAVAJAVA\Collections>javac HashMapDemo.java
F:\BHAVAJAVA\Collections>java HashMapDemo
Display Elements from HashMap
Satya: 100.1
Hari: 97.27
Siva: 75.22
Eswar: 99.99
Rajesh: 100.0
*/
```

2. TreeMap Class

- The **TreeMap** class extends **AbstractMap** and implements the **NavigableMap** interface.
- It creates maps stored in a tree structure.
- A **TreeMap** provides an efficient means of storing key/value pairs in sorted order and allows rapid retrieval.
- Unlike a hash map, a tree map guarantees that its elements will be sorted in ascending key order.

TreeMap is a generic class that has this declaration:

```
class  TreeMap<K,V> extends AbstractMap implements NavigableMap{
    //Methods
}
```

TreeMap constructors:

TreeMap( )
TreeMap(Comparator<? super K> comp)
TreeMap(Map<? extends K, ? extends V> m)
TreeMap(SortedMap<K, ? extends V> sm)

```
import java.util.*;
class TreeMapDemo {
      public static void main(String args[]) {
            // Create a tree map.
            TreeMap<String, Double> tm = new TreeMap<String, Double>();
            // Put elements to the map.
            tm.put("Rajesh", new Double(100.00));
             tm.put("Eswar", new Double(99.99));
             tm.put("Hari", new Double(97.27));
             tm.put("Siva", new Double(75.22));
             tm.put("Satya", new Double(100.100));
             // Get a set of the entries.
              Set<Map.Entry<String, Double>> set = tm.entrySet();
              // Display the elements.
              for(Map.Entry<String, Double> me : set) {
                        System.out.print(me.getKey() + ": ");
                        System.out.println(me.getValue());
              }
      }
}
/*Input and Output
F:\BHAVAJAVA\Collections>javac TreeMapDemo.java
F:\BHAVAJAVA\Collections>java TreeMapDemo
Eswar: 99.99
Hari: 97.27
Rajesh: 100.0
Satya: 100.1
Siva: 75.22
*/
```

3. LinkedHashMap Class

- **LinkedHashMap** extends **HashMap**.
- It maintains a linked list of the entries in the map, in the order in which they were inserted.
- This allows insertion-order iteration over the map.
- That is, when iterating through a collection-view of a **LinkedHashMap**, the elements will be returned in the order in which they were inserted.

OHM SRI SAIRAM

- **LinkedHashMap** is a generic class that has this declaration:

```
class  LinkedHashMap<K,V> extends HashMap{
    //Methods
}
```

LinkedHashMap constructors:
LinkedHashMap( )
LinkedHashMap(Map<? extends K, ? extends V> *m*)
LinkedHashMap(int *capacity*)
LinkedHashMap(int *capacity*, float *fillRatio*)
LinkedHashMap(int *capacity*, float *fillRatio*, boolean *Order*)

```
import java.util.*;
class LinkedHashMapDemo {
     public static void main(String args[]) {
          // Create a hash map.
          LinkedHashMap<String, Double> lhm =
                                   new LinkedHashMap<String, Double>();
          // Put elements to the map
          lhm.put("Rajesh", new Double(100.00));
          lhm.put("Eswar", new Double(99.99));
          lhm.put("Hari", new Double(97.27));
          lhm.put("Siva", new Double(75.22));
          lhm.put("Satya", new Double(100.100));
          // Get a set of the entries.
          Set<Map.Entry<String, Double>> set = lhm.entrySet();
          // Display the set.
          System.out.println("Display Elements from Haslhmap");
          for(Map.Entry<String, Double> me : set) {
                System.out.print(me.getKey() + ": ");
                System.out.println(me.getValue());
          }
          System.out.println();
 }
}
/*Input and Output
****************
F:\BHAVAJAVA\Collections>javac LinkedHashMapDemo.java
F:\BHAVAJAVA\Collections>java LinkedHashMapDemo
Display Elements from Haslhmap
Rajesh: 100.0
Eswar: 99.99
Hari: 97.27
Siva: 75.22
Satya: 100.1   */
```

OHM SRI SAIRAM

4. EnumMap Class

- **EnumMap** extends **AbstractMap** and implements **Map**.
- It is specifically for use with keys of an **enum** type.
- EnumMap defines no methods of its own.
- It is a generic class that has this declaration:

```
class EnumMap<K extends Enum<K>, V>{
    //Methods
 }
```

Here, K specifies the type of key, and V specifies the type of value. Notice that K must extend Enum<K>, which enforces the requirement that the keys must be of an enum type.

EnumMap class constructors:

EnumMap(Class<K> *kType*)
EnumMap(Map<K, ? extends V> *m*)
EnumMap(EnumMap<K, ? extends V> *em*)

```
import java.util.*;
enum Rainbow{
    RED,ORANGE,YELLOW,GREEN,BLUE,INDINGO,VIOLET
};
public class EnumMapDemo{
    public static void main(String[] args){
        // Creating a set
        EnumMap<Rainbow, String> rmap = new
                     EnumMap<Rainbow, String>(Rainbow.class);
        rmap.put(Rainbow.RED,"VIBRANCY");
        rmap.put(Rainbow.ORANGE,"CREATIVITY");
        rmap.put(Rainbow.YELLOW,"WISDOM");
        rmap.put(Rainbow.GREEN,"NATURE");
        rmap.put(Rainbow.BLUE,"DIVINITY");
        rmap.put(Rainbow.INDINGO,"INFINITY");
        rmap.put(Rainbow.VIOLET,"SPIRTUALITY");
        // Printing size of EnumMap in java
        System.out.println("Size of EnumMap in java: " +
                                    rmap.size());
        System.out.println("EnumMap: " + rmap);
      }
}
```

OHM SRI SAIRAM

```
/*Output

F:\BHAVAJAVA\Collections>javac EnumMapDemo.java
F:\BHAVAJAVA\Collections>java EnumMapDemo
Size of EnumMap in java: 7
EnumMap: {RED=VIBRANCY, ORANGE=CREATIVITY, YELLOW=WISDOM, GREEN=NATURE,
BLUE=DIVINITY, INDINGO=INFINITY, VIOLET=SPIRTUALITY}

*/
```

5. IdentityHashMap Class

- **IdentityHashMap** extends **AbstractMap** and implements the **Map** interface.
- IdentityHashMap uses **reference equality** to comapare keys and values while HashMap uses **object equality** to compare keys and values.
- It has a generic class that has this declaration:

```
class IdentityHashMap extends AbstractMap implements Map{
    //Methods
 }
```

Here, **K** specifies the type of key, and **V** specifies the type of value.

```
import java.util.*;
public class IdentityHashMapDemo {
    public static void main(String[] args) {
        // Created HashMap and IdentityHashMap objects
        Map hmo = new HashMap();
        Map io = new IdentityHashMap();
        // Putting  keys and values in HashMap and IdentityHashMap Object
        hmo.put(new String("key") ,"Google");
        hmo.put(new String("key") ,"Microsoft");
        io.put(new String("identityKey") ,"Google");
        io.put(new String("identityKey") ,"Microsoft");
        // Print HashMap and IdentityHashMap Size : After adding  keys
        System.out.println("HashMap after adding key :"+ hmo);
        System.out.println("IdentityHashMap after adding key :"+ io);

  }
 }
```

```
/*Output
F:\BHAVAJAVA\Collections>javac IdentityHashMapDemo.java
Note: IdentityHashMapDemo.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
F:\BHAVAJAVA\Collections>java IdentityHashMapDemo
HashMap after adding key :{key=Microsoft}
IdentityHashMap after adding key :{identityKey=Google, identityKey=Microsoft}
*/
```

(Continued in PART-III)

OHM SRI SAIRAM

UNIT-V

java.util package
(The Collection Framework)

In Part-I Introduction to Collection framework, Collection interfaces and Collection classes covered.
In Part-II iterators and maps covered.

PART-III

Comparators:

Java Comparator interface is used to order the objects of a user-defined class. This interface is found in java.util package and contains two methods compare (Object obj1,Object obj2) and equals(Object element).

Comparator is a generic interface that has this declaration:

```
interface Comparator<T>{
   //Methods
}
```

Here, **T** specifies the type of objects being compared.

Method of Java Comparator interface:

Method	Description
public int compare(Object obj1, Object obj2)	It compares the first object with the second object.
public boolean equals(Object obj)	It is used to compare the current object with the specified object.

Example Program will understood through following "Collection algortihms concept."

Collection Algorithms:

- The Collections Framework defines several algorithms that can be applied to collections and maps.
- These algorithms are defined as static methods within the **Collections** class.
- Several of the methods can throw a **ClassCastException**, which occurs when an attempt is made to compare incompatible types, or an **UnsupportedOperationException**,
- All Collections class methods are static methods. So all of these methods called directly through class name with out creating object.

OHM SRI SAIRAM

Collections class important methods:

(Note: Practice any ten methods from below)

Modifier and Type	Method and Description
static <T> boolean	**addAll**(**Collection**<? super T> c, T... elements) Adds all of the specified elements to the specified collection.
static <T> Queue**<T>**	**asLifoQueue**(**Deque**<T> deque) Returns a view of a **Deque** as a Last-in-first-out (Lifo) **Queue**.
static <T> int	**binarySearch**(**List**<? extends **Comparable**<? super T>> list, T key) Searches the specified list for the specified object using the binary search algorithm.
static <T> int	**binarySearch**(**List**<? extends T> list, T key, **Comparator**<? super T> c) Searches the specified list for the specified object using the binary search algorithm.
static <T> void	**copy**(**List**<? super T> dest, **List**<? extends T> src) Copies all of the elements from one list into another.
static boolean	**disjoint**(**Collection**<?> c1, **Collection**<?> c2) Returns true if the two specified collections have no elements in common.
static <T> Enumeration**<T>**	**emptyEnumeration**() Returns an enumeration that has no elements.
static <T> Iterator**<T>**	**emptyIterator**() Returns an iterator that has no elements.
static <T> List**<T>**	**emptyList**() Returns the empty list (immutable).
static <T> ListIterator**<T>**	**emptyListIterator**() Returns a list iterator that has no elements.
static <K,V> Map**<K,V>**	**emptyMap**() Returns the empty map (immutable).
static <T> Set**<T>**	**emptySet**() Returns the empty set (immutable).
static <T> Enumeration**<T>**	**enumeration**(**Collection**<T> c) Returns an enumeration over the specified collection.

OHM SRI SAIRAM

static <T> void	**fill**(**List**<? super T> list, T obj) Replaces all of the elements of the specified list with the specified element.
static int	**frequency**(**Collection**<?> c, **Object** o) Returns the number of elements in the specified collection equal to the specified object.
static int	**indexOfSubList**(**List**<?> source, **List**<?> target) Returns the starting position of the first occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static int	**lastIndexOfSubList**(**List**<?> source, **List**<?> target) Returns the starting position of the last occurrence of the specified target list within the specified source list, or -1 if there is no such occurrence.
static <T> ArrayList**<T>**	**list**(**Enumeration**<T> e) Returns an array list containing the elements returned by the specified enumeration in the order they are returned by the enumeration.
static <T> T	**max**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the maximum element of the given collection, according to the order induced by the specified comparator.
static <T> T	**min**(**Collection**<? extends T> coll, **Comparator**<? super T> comp) Returns the minimum element of the given collection, according to the order induced by the specified comparator.
static <E> Set**<E>**	**newSetFromMap**(**Map**<E,**Boolean**> map) Returns a set backed by the specified map.
static <T> boolean	**replaceAll**(**List**<T> list, T oldVal, T newVal) Replaces all occurrences of one specified value in a list with another.
static void	**reverse**(**List**<?> list) Reverses the order of the elements in the specified list.
static <T> Comparator**<T>**	**reverseOrder**() Returns a comparator that imposes the reverse of the *natural ordering* on a collection of objects that implement the Comparable interface.
static <T> Comparator**<T>**	**reverseOrder**(**Comparator**<T> cmp) Returns a comparator that imposes the reverse ordering of the specified comparator.

OHM SRI SAIRAM

static void	**rotate**(**List**<?> list, int distance) Rotates the elements in the specified list by the specified distance.
static void	**shuffle**(**List**<?> list) Randomly permutes the specified list using a default source of randomness.
static void	**shuffle**(**List**<?> list, **Random** rnd) Randomly permute the specified list using the specified source of randomness.
static <T extends Comparable**<? super T>> void**	**sort**(**List**<T> list) Sorts the specified list into ascending order, according to the **natural ordering** of its elements.
static <T> void	**sort**(**List**<T> list, **Comparator**<? super T> c) Sorts the specified list according to the order induced by the specified comparator.
static void	**swap**(**List**<?> list, int i, int j) Swaps the elements at the specified positions in the specified list.

OHM SRI SAIRAM

```
// Demonstrate various algorithms from Collections class.
import java.util.*;
class AlgorithmsDemo {
     public static void main(String args[]) {
          // Create and initialize linked list.
          LinkedList<Integer> ll = new LinkedList<Integer>();
          ll.add(7);
          ll.add(777);
          ll.add(77777);
          ll.add(77);
          // Create a reverse order comparator.
          Comparator<Integer> r = Collections.reverseOrder();
          // Sort list by using the comparator.
          Collections.sort(ll, r);
          System.out.print("List sorted in reverse: ");
          for(int i : ll)
               System.out.print(i+ " ");
          System.out.println();
          // Shuffle list.
          Collections.shuffle(ll);
          // Display randomized list.
          System.out.print("List shuffled: ");
          for(int i : ll)
               System.out.print(i + " ");
          System.out.println();
          System.out.println("Minimum: " + Collections.min(ll));
          System.out.println("Maximum: " + Collections.max(ll));
   }
}


/*Input and Output
F:\BHAVAJAVA\Collections>java AlgorithmsDemo
List sorted in reverse: 77777 777 77 7
List shuffled: 777 7 77777 77
Minimum: 7
Maximum: 77777
*/
```

OHM SRI SAIRAM

Arrays:

- The Arrays class provides various methods that are useful when working with arrays.
- These methods help bridge the gap between collections and arrays.
- Some of the important methods included in following table.(Practice at least one method for each colored group)

Modifier and Type	Method and Description
static <T> **List**<T>	**asList**(T... a) Returns a fixed-size list backed by the specified array.
static int	**binarySearch**(byte[] a, byte key) Searches the specified array of bytes for the specified value using the binary search algorithm.
static int	**binarySearch**(byte[] a, int fromIndex, int toIndex, byte key) Searches a range of the specified array of bytes for the specified value using the binary search algorithm.
static int	**binarySearch**(char[] a, char key) Searches the specified array of chars for the specified value using the binary search algorithm.
static int	**binarySearch**(char[] a, int fromIndex, int toIndex, char key) Searches a range of the specified array of chars for the specified value using the binary search algorithm.
static int	**binarySearch**(double[] a, double key) Searches the specified array of doubles for the specified value using the binary search algorithm.
static int	**binarySearch**(double[] a, int fromIndex, int toIndex, double key) Searches a range of the specified array of doubles for the specified value using the binary search algorithm.
static int	**binarySearch**(float[] a, float key) Searches the specified array of floats for the specified value using the binary search algorithm.
static int	**binarySearch**(float[] a, int fromIndex, int toIndex, float key) Searches a range of the specified array of floats for the specified value using the binary search algorithm.
static int	**binarySearch**(int[] a, int key) Searches the specified array of ints for the specified value using the binary search algorithm.
static int	**binarySearch**(int[] a, int fromIndex, int toIndex, int key) Searches a range of the specified array of ints for the specified value using the binary search algorithm.

OHM SRI SAIRAM

static int	**binarySearch**(long[] a, int fromIndex, int toIndex, long key) Searches a range of the specified array of longs for the specified value using the binary search algorithm.
static int	**binarySearch**(long[] a, long key) Searches the specified array of longs for the specified value using the binary search algorithm.
static int	**binarySearch**(**Object**[] a, int fromIndex, int toIndex, **Object** key) Searches a range of the specified array for the specified object using the binary search algorithm.
static int	**binarySearch**(**Object**[] a, **Object** key) Searches the specified array for the specified object using the binary search algorithm.
static int	**binarySearch**(short[] a, int fromIndex, int toIndex, short key) Searches a range of the specified array of shorts for the specified value using the binary search algorithm.
static int	**binarySearch**(short[] a, short key) Searches the specified array of shorts for the specified value using the binary search algorithm.
static boolean[]	**copyOf**(boolean[] original, int newLength) Copies the specified array, truncating or padding with false (if necessary) so the copy has the specified length.
static byte[]	**copyOf**(byte[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static char[]	**copyOf**(char[] original, int newLength) Copies the specified array, truncating or padding with null characters (if necessary) so the copy has the specified length.
static double[]	**copyOf**(double[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static float[]	**copyOf**(float[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static int[]	**copyOf**(int[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static long[]	**copyOf**(long[] original, int newLength) Copies the specified array, truncating or padding with zeros (if necessary) so the copy has the specified length.
static short[]	**copyOf**(short[] original, int newLength) Copies the specified array, truncating or padding with zeros (if

OHM SRI SAIRAM

	necessary) so the copy has the specified length.
static <T> T[]	**copyOf**(T[] original, int newLength) Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static <T,U> T[]	**copyOf**(U[] original, int newLength, **Class**<? extends T[]> newType) Copies the specified array, truncating or padding with nulls (if necessary) so the copy has the specified length.
static boolean[]	**copyOfRange**(boolean[] original, int from, int to) Copies the specified range of the specified array into a new array.
static byte[]	**copyOfRange**(byte[] original, int from, int to) Copies the specified range of the specified array into a new array.
static char[]	**copyOfRange**(char[] original, int from, int to) Copies the specified range of the specified array into a new array.
static double[]	**copyOfRange**(double[] original, int from, int to) Copies the specified range of the specified array into a new array.
static float[]	**copyOfRange**(float[] original, int from, int to) Copies the specified range of the specified array into a new array.
static int[]	**copyOfRange**(int[] original, int from, int to) Copies the specified range of the specified array into a new array.
static long[]	**copyOfRange**(long[] original, int from, int to) Copies the specified range of the specified array into a new array.
static short[]	**copyOfRange**(short[] original, int from, int to) Copies the specified range of the specified array into a new array.
static <T> T[]	**copyOfRange**(T[] original, int from, int to) Copies the specified range of the specified array into a new array.
static boolean	**equals**(boolean[] a, boolean[] a2) Returns true if the two specified arrays of booleans are *equal* to one another.
static boolean	**equals**(byte[] a, byte[] a2) Returns true if the two specified arrays of bytes are *equal* to one another.
static boolean	**equals**(char[] a, char[] a2) Returns true if the two specified arrays of chars are *equal* to one another.
static boolean	**equals**(double[] a, double[] a2) Returns true if the two specified arrays of doubles are *equal* to one another.
static boolean	**equals**(float[] a, float[] a2) Returns true if the two specified arrays of floats are *equal* to one another.

OHM SRI SAIRAM

static boolean	**equals**(int[] a, int[] a2) Returns true if the two specified arrays of ints are *equal* to one another.
static boolean	**equals**(long[] a, long[] a2) Returns true if the two specified arrays of longs are *equal* to one another.
static boolean	**equals**(**Object**[] a, **Object**[] a2) Returns true if the two specified arrays of Objects are *equal* to one another.
static boolean	**equals**(short[] a, short[] a2) Returns true if the two specified arrays of shorts are *equal* to one another.
static void	**fill**(boolean[] a, boolean val) Assigns the specified boolean value to each element of the specified array of booleans.
static void	**fill**(boolean[] a, int fromIndex, int toIndex, boolean val) Assigns the specified boolean value to each element of the specified range of the specified array of booleans.
static void	**fill**(byte[] a, byte val) Assigns the specified byte value to each element of the specified array of bytes.
static void	**fill**(byte[] a, int fromIndex, int toIndex, byte val) Assigns the specified byte value to each element of the specified range of the specified array of bytes.
static void	**fill**(char[] a, char val) Assigns the specified char value to each element of the specified array of chars.
static void	**fill**(char[] a, int fromIndex, int toIndex, char val) Assigns the specified char value to each element of the specified range of the specified array of chars.
static void	**fill**(double[] a, double val) Assigns the specified double value to each element of the specified array of doubles.
static void	**fill**(double[] a, int fromIndex, int toIndex, double val) Assigns the specified double value to each element of the specified range of the specified array of doubles.
static void	**fill**(float[] a, float val) Assigns the specified float value to each element of the specified array of floats.
static void	**fill**(float[] a, int fromIndex, int toIndex, float val) Assigns the specified float value to each element of the specified range of the specified array of floats.

OHM SRI SAIRAM

static void	**fill**(int[] a, int val) Assigns the specified int value to each element of the specified array of ints.
static void	**fill**(int[] a, int fromIndex, int toIndex, int val) Assigns the specified int value to each element of the specified range of the specified array of ints.
static void	**fill**(long[] a, int fromIndex, int toIndex, long val) Assigns the specified long value to each element of the specified range of the specified array of longs.
static void	**fill**(long[] a, long val) Assigns the specified long value to each element of the specified array of longs.
static void	**fill**(**Object**[] a, int fromIndex, int toIndex, **Object** val) Assigns the specified Object reference to each element of the specified range of the specified array of Objects.
static void	**fill**(**Object**[] a, **Object** val) Assigns the specified Object reference to each element of the specified array of Objects.
static void	**fill**(short[] a, int fromIndex, int toIndex, short val) Assigns the specified short value to each element of the specified range of the specified array of shorts.
static void	**fill**(short[] a, short val) Assigns the specified short value to each element of the specified array of shorts.
static int	**hashCode**(int[] a) Returns a hash code based on the contents of the specified array.
static void	**sort**(byte[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(byte[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(char[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(char[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(double[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(double[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(float[] a) Sorts the specified array into ascending numerical order.

OHM SRI SAIRAM

static void	**sort**(float[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(int[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(int[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(long[] a) Sorts the specified array into ascending numerical order.
static void	**sort**(long[] a, int fromIndex, int toIndex) Sorts the specified range of the array into ascending order.
static void	**sort**(**Object**[] a) Sorts the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static void	**sort**(**Object**[] a, int fromIndex, int toIndex) Sorts the specified range of the specified array of objects into ascending order, according to the **natural ordering** of its elements.
static **String**	**toString**(char[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(double[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(float[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(int[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(long[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(**Object**[] a) Returns a string representation of the contents of the specified array.
static **String**	**toString**(short[] a) Returns a string representation of the contents of the specified array.

OHM SRI SAIRAM

```
// Demonstrate various algorithms.
import java.util.*;
class ArraysDemo {
      public static void main(String args[]) {
        // Allocate and initialize array.
         int array[] = new int[10];
        for(int i = 0; i < 10; i++)
          array[i] = 3 * i;
        // Display, sort, and display the array.
        System.out.print("Original contents: ");
        display(array);
        Arrays.sort(array);
        System.out.print("Sorted: ");
        display(array);
        // Fill and display the array.
        Arrays.fill(array, 2, 6, 99);
        System.out.print("After fill(): ");
        display(array);
        // Sort and display the array.
        Arrays.sort(array);
        System.out.print("After sorting again: ");
        display(array);
        // Binary search for 9.
        System.out.print("The value 9 is at location ");
        int index =Arrays.binarySearch(array,18);
        System.out.println(index);
      }
       static void display(int array[]) {
            for(int i: array)
               System.out.print(i + " ");
            System.out.println();
      }
}
/*
Input and Output
F:\BHAVAJAVA\Collections>javac ArraysDemo.java
F:\BHAVAJAVA\Collections>java ArraysDemo
Original contents: 0 3 6 9 12 15 18 21 24 27
Sorted: 0 3 6 9 12 15 18 21 24 27
After fill(): 0 3 99 99 99 99 18 21 24 27
After sorting again: 0 3 18 21 24 27 99 99 99 99
The value 9 is at location 2
*/
```

OHM SRI SAIRAM

The Legacy Classes and Interfaces:

- Java Collection framework added in J2SE 1.2.Before J2SE 1.2 collection framework was not included.
- Instead java.util package defines set of interfaces and classes to support operations for storing and accessing objects.
- These classes and interfaces called **Legacy classes and interfaces**.
- The main difference between legacy classes and collection framework is "All collection framework interfaces, classes and their functionality not synchronized by default. But Legacy classes and interfaces are synchronized by default. "

List of Legacy classes and interfaces are

Legacy interface	
1	Enumeration Interface
Legacy classes	
2	Vector
3	Stack
4	Dictionary
5	HashTable
6	Properties

Enumeration Interface

The **Enumeration** interface defines the methods by which you can *enumerate* (obtain one at a time) the elements in a collection of objects.

It has this declaration:

```
interface Enumeration<E>{
        boolean hasMoreElements( );
        E nextElement( );
}
```

where **E** specifies the type of element being enumerated.

- **hasMoreElements( )** must return **true** while there are still more elements to extract, and **false** when all the elements have been enumerated.
- **nextElement( )** returns the next object in the enumeration. That is, each call to **nextElement( )** obtains the next object in the enumeration. It throws **NoSuchElementException** when the enumeration is complete.

Vector

- Vector implements a dynamic array.
- It is similar to ArrayList except Vector is synchronized.
- With the advent of collections and generics, Vector was reengineered to extend AbstractList and to implement the List interface and Itearable.
- Vector is declared like this:

```
Class Vector<E>{
      //constructors
     //Methods
}
```

OHM SRI SAIRAM

Here, E specifies the type of element that will be stored.

Vector Constructors and Methods:

No.	Constructor & Description
1	**Vector( )** This constructor creates a default vector, which has an initial size of 10.
2	**Vector(int size)** This constructor accepts an argument that equals to the required size, and creates a vector whose initial capacity is specified by size.
3	**Vector(int size, int incr)** This constructor creates a vector whose initial capacity is specified by size and whose increment is specified by incr. The increment specifies the number of elements to allocate each time that a vector is resized upward.
4	**Vector(Collection c)** This constructor creates a vector that contains the elements of collection c.

No	Method & Description
1	**void add(int index, Object element)** Inserts the specified element at the specified position in this Vector.
2	**boolean add(Object o)** Appends the specified element to the end of this Vector.
3	**boolean addAll(Collection c)** Appends all of the elements in the specified Collection to the end of this Vector, in the order that they are returned by the specified Collection's Iterator.
4	**boolean addAll(int index, Collection c)** Inserts all of the elements in in the specified Collection into this Vector at the specified position.
5	**void addElement(Object obj)** Adds the specified component to the end of this vector, increasing its size by one.
6	**int capacity()** Returns the current capacity of this vector.
7	**void clear()** Removes all of the elements from this vector.

OHM SRI SAIRAM

8	**Object clone()** Returns a clone of this vector.
9	**boolean contains(Object elem)** Tests if the specified object is a component in this vector.
10	**boolean containsAll(Collection c)** Returns true if this vector contains all of the elements in the specified Collection.
11	**Object elementAt(int index)** Returns the component at the specified index.
12	**Enumeration elements()** Returns an enumeration of the components of this vector.
13	**void ensureCapacity(int minCapacity)** Increases the capacity of this vector, if necessary, to ensure that it can hold at least the number of components specified by the minimum capacity argument.
14	**boolean equals(Object o)** Compares the specified Object with this vector for equality.
15	**Object firstElement()** Returns the first component (the item at index 0) of this vector.
16	**Object get(int index)** Returns the element at the specified position in this vector.
17	**int hashCode()** Returns the hash code value for this vector.
18	**int indexOf(Object elem)** Searches for the first occurence of the given argument, testing for equality using the equals method.
19	**int indexOf(Object elem, int index)** Searches for the first occurence of the given argument, beginning the search at index, and testing for equality using the equals method.
20	**void insertElementAt(Object obj, int index)** Inserts the specified object as a component in this vector at the specified index.

OHM SRI SAIRAM

21	**boolean isEmpty()** Tests if this vector has no components.
22	**Object lastElement()** Returns the last component of the vector.
23	**int lastIndexOf(Object elem)** Returns the index of the last occurrence of the specified object in this vector.
24	**int lastIndexOf(Object elem, int index)** Searches backwards for the specified object, starting from the specified index, and returns an index to it.
25	**Object remove(int index)** Removes the element at the specified position in this vector.
26	**boolean remove(Object o)** Removes the first occurrence of the specified element in this vector, If the vector does not contain the element, it is unchanged.
27	**boolean removeAll(Collection c)** Removes from this vector all of its elements that are contained in the specified Collection.
28	**void removeElementAt(int index)** removeElementAt(int index).
29	**void setElementAt(Object obj, int index)** Sets the component at the specified index of this vector to be the specified object.
30	**void setSize(int newSize)** Sets the size of this vector.
31	**int size()** Returns the number of components in this vector.
32	**List subList(int fromIndex, int toIndex)** Returns a view of the portion of this List between fromIndex, inclusive, and toIndex, exclusive.
33	**Object[] toArray()** Returns an array containing all of the elements in this vector in the correct order.

OHM SRI SAIRAM

34	**String toString()** Returns a string representation of this vector, containing the String representation of each element.
35	**void trimToSize()** Trims the capacity of this vector to be the vector's current size.

```
// Demonstrate various Vector operations.
import java.util.*;
class VectorDemo {
      public static void main(String args[]) {
            // initial size is 3, increment is 2
         Vector<String> v = new Vector<String>(3, 2);
         System.out.println("Initial size: " + v.size());
         System.out.println("Initial capacity: " +v.capacity());
         v.addElement("Sachin");
         v.addElement("Dhoni");
        v.addElement("Kohli");
        v.addElement("Kartheek");
        System.out.println("Capacity after four additions:" +v.capacity());
        v.addElement("Saurabh");
        System.out.println("Current capacity:"+v.capacity());
        v.addElement("Lakshman");
        v.addElement("Dravid");
        System.out.println("Current capacity: " +v.capacity());
        v.addElement("Mithali Raj");
        v.addElement("Julan Goswami");
        System.out.println("Current capacity:"+v.capacity());
        v.addElement("Poonam");
        v.addElement("Kapil Dev");
        System.out.println("First Player in the Team:" + v.firstElement());
        System.out.println("Last Player in the Team: " + v.lastElement());
        // Enumerate the elements in the vector.
        Enumeration<String> vEnum = v.elements();
        System.out.println("\nIndian Favorite Cricket Players:");
        while(vEnum.hasMoreElements())
            System.out.println(vEnum.nextElement() + " ");
        System.out.println();
         if(v.contains("Sachin"))
            System.out.println("Yes,Indian Team has Sachin");
       }
}
```

```
/*Input and Output
F:\BHAVAJAVA\Collections>javac VectorDemo.java
F:\BHAVAJAVA\Collections>java VectorDemo
Initial size: 0
Initial capacity: 3
Capacity after four additions:5
Current capacity:5
Current capacity: 7
Current capacity:9
First Player in the Team: Sachin
Last Player in the Team: Kapil Dev

Indian Favorite Cricket Players:
Sachin
Dhoni
Kohli
Kartheek
Saurabh
Lakshman
Dravid
Mithali Raj
Julan Goswami
Poonam
Kapil Dev

Yes,Indian Team has Sachin
*/
```

Dictionary:

Dictionary is an abstract class that represents a key/value storage repository and operates much like Map. Given a key and value, you can store the value in a Dictionary object. Once the value is stored, you can retrieve it by using its key. Thus, like a map, a dictionary can be thought of as a list of key/value pairs.

Syntax:

```
abstract class Dictionary<K,V>{
    //Methods
}
```

Here, K specifies the type of keys, and V specifies the type of values

Sr.No.	Method & Description
1	**Enumeration elements( )** Returns an enumeration of the values contained in the dictionary.
2	**Object get(Object key)** Returns the object that contains the value associated with the key. If the key is not in the dictionary, a null object is returned.
3	**boolean isEmpty( )** Returns true if the dictionary is empty, and returns false if it contains at least one key.
4	**Enumeration keys( )** Returns an enumeration of the keys contained in the dictionary.
5	**Object put(Object key, Object value)** Inserts a key and its value into the dictionary. Returns null if the key is not already in the dictionary; returns the previous value associated with the key if the key is already in the dictionary.
6	**Object remove(Object key)** Removes the key and its value. Returns the value associated with the key. If the key is not in the dictionary, a null is returned.
7	**int size( )** Returns the number of entries in the dictionary.

Hashtable

- **Hashtable** was part of the original **java.util** and is a concrete implementation of a **Dictionary**.
- **Hashtable** was reengineered to also implement the **Map** interface.
- **Hashtable** is integrated into the Collections Framework. It is similar to **HashMap**,
- Hashtable is synchronized.
- **Hashtable** stores key/value pairs in a hash table.

Syntax:

```
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>{
   //Methods
}
```

OHM SRI SAIRAM

Hashtable Constructors:

Constructor	Description
Hashtable()	It creates an empty hashtable having the initial default capacity and load factor.
Hashtable(int capacity)	It accepts an integer parameter and creates a hash table that contains a specified initial capacity.
Hashtable(int capacity, float loadFactor)	It is used to create a hash table having the specified initial capacity and loadFactor.
Hashtable(Map<? extends K,? extends V> t)	It creates a new hash table with the same mappings as the given Map.

Hashtable methods

Method	Description
void clear()	It is used to reset the hash table.
Object clone()	It returns a shallow copy of the Hashtable.
Enumeration elements()	It returns an enumeration of the values in the hash table.
Set<Map.Entry<K,V>> entrySet()	It returns a set view of the mappings contained in the map.
boolean equals(Object o)	It is used to compare the specified Object with the Map.
int hashCode()	It returns the hash code value for the Map
Enumeration<K> keys()	It returns an enumeration of the keys in the hashtable.
Set<K> keySet()	It returns a Set view of the keys contained in the map.
V put(K key, V value)	It inserts the specified value with the specified key in the hash table.
void putAll(Map<? extends K,? extends V> t))	It is used to copy all the key-value pair from map to hashtable.
boolean remove(Object key, Object value)	It removes the specified values with the associated specified keys from the hashtable.
V replace(K key, V value)	It replaces the specified value for a specified key.
boolean replace(K key, V oldValue, V newValue)	It replaces the old value with the new value for a specified key.
String toString()	It returns a string representation of the Hashtable object.

OHM SRI SAIRAM

boolean contains(Object value)	This method returns true if some value equal to the value exists within the hash table, else return false.
boolean containsKey(Object key)	This method return true if some key equal to the key exists within the hash table, else return false.
boolean isEmpty()	This method returns true if the hash table is empty; returns false if it contains at least one key.
V get(Object key)	This method returns the object that contains the value associated with the key.
V remove(Object key)	It is used to remove the key and its value. This method returns the value associated with the key.
int size()	This method returns the number of entries in the hash table.

```
import java.util.*;
class HashtableDemo{
     public static void main(String[] args){
          Hashtable<String,Double> gdp=new Hashtable<String,Double>();
          gdp.put("USA",2.00);
          gdp.put("China",6.00);
          gdp.put("Japan",0.70);
          gdp.put("Germany",1.10);
          gdp.put("India",5.80);
          Enumeration<String> country=gdp.keys();
          System.out.println("GDP statistics of 5 countries");
        while(country.hasMoreElements()) {
            String str = country.nextElement();
            System.out.println(str + ": " +gdp.get(str));
       }
       System.out.println();
    }
}
/*Input and Output
GDP statistics of 5 counries
China: 6.0
USA: 2.0
Germany: 1.1
India: 5.8
Japan: 0.7
*/
```

OHM SRI SAIRAM

Stack:

Stack is a subclass of **Vector** that implements a standard last-in, first-out stack.

Syntax:

```
class Stack<E> extends Vector<E>{
  //Methods
}
```

Stack constructor:

```
Stack()
```

No.	Method & Description
1	**boolean empty()** Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements.
2	**Object peek( )** Returns the element on the top of the stack, but does not remove it.
3	**Object pop( )** Returns the element on the top of the stack, removing it in the process.
4	**Object push(Object element)** Pushes the element onto the stack. Element is also returned.
5	**int search(Object element)** Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned.

OHM SRI SAIRAM

```
import java.util.*;
class StackDemo{
      public static void main(String args[]){
            Stack<Integer> st=new Stack<Integer>();
            Scanner sin=new Scanner(System.in);
            int item;
            char ch;
            do{
                System.out.println("Enter choice to perform stack operations");
                System.out.println("1-push 2-Pop 3-Peek 4-Check Empty 5-Search");
                int choice=sin.nextInt();
                switch(choice){
                       case 1:System.out.println("Enter Item");
                              item=sin.nextInt();
                              st.push(item);
                              break;
                       case 2:item=st.pop();
                              System.out.println("Popped item is:"+item);
                              break;
                       case 3:item=st.peek();
                              System.out.println("Top of the Stack:"+item);
                              break;
                       case 4:if(st.empty())
                                  System.out.println("Stack Empty");
                              else
                                  System.out.println("Stack not Empty");
                              break;
                    case 5:System.out.println("Enter Element to Search");
                           item=sin.nextInt();
                           int pos=st.search(item);
                           if(pos==-1)
                              System.out.println("Element not found");
                           else
                              System.out.println("Element Found at:"+pos);
                }
                System.out.println("Do You want to continue Press Y");
                ch=sin.next().charAt(0);
              }while(ch=='Y');
      }
}
```

OHM SRI SAIRAM

```
/*Input and Output
F:\BHAVAJAVA\Collections>javac StackDemo.java
F:\BHAVAJAVA\Collections>java StackDemo
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
1
Enter Item
7
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
1
Enter Item
25
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
1
Enter Item
34
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
1
Enter Item
43
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
1
Enter Item
44
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
2
Popped item is:44
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
3
Top of the Stack:43
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
5
Enter Element to Search
25
Element Found at:3
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
4
Stack not Empty
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
3
Top of the Stack:43
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
1
Enter Item
77
Do You want to continue Press Y
Y
Enter choice to perform stack operations
1-push 2-Pop 3-Peek 4-Check Empty 5-Search
1
Enter Item
99
Do You want to continue Press Y
N
*/
```

OHM SRI SAIRAM

Properties class:

Properties is a subclass of Hashtable. It is used to maintain lists of values in which the key is a String and the value is also a String.

The Properties class is used by many other Java classes. For example, it is the type of object returned by System.getProperties( ) when obtaining environmental values.

```
class Properties<E> extends Hashtable<E>{
 //Methods
}
```

Properties Constructors:

Sr.No.	Constructor & Description
1	**Properties( )** This constructor creates a Properties object that has no default values.
2	**Properties(Properties propDefault)** Creates an object that uses propDefault for its default values. In both cases, the property list is empty.

Methods

Sr.No.	Method & Description
1	**String getProperty(String key)** Returns the value associated with the key. A null object is returned if the key is neither in the list nor in the default property list.
2	**String getProperty(String key, String defaultProperty)** Returns the value associated with the key; defaultProperty is returned if the key is neither in the list nor in the default property list.
3	**void load(InputStream streamIn) throws IOException** Inputs a property list from the input stream linked to streamIn.
4	**Enumeration propertyNames( )** Returns an enumeration of the keys. This includes those keys found in the default property list, too.

OHM SRI SAIRAM

5	**Object setProperty(String key, String value)** Associates value with the key. Returns the previous value associated with the key, or returns null if no such association exists.
6	**void store(OutputStream streamOut, String description)** After writing the string specified by description, the property list is written to the output stream linked to streamOut.

```
import java.util.*;
public class PropertiesDemo {
   public static void main(String args[]) {
      Properties capitals = new Properties();
      Set states;
      String str;
      capitals.put("Andhra Pradesh", "Amaravathi");
      capitals.put("Telangana", "Hyderabad");
      capitals.put("Tamil Nadu", "Chennai");
      capitals.put("Karnataka", "Bengaluru");
      capitals.put("Kerala", "Tiruvanantapuramu");
      // Show all states and capitals in hashtable.
      states = capitals.keySet();   // get set-view of keys
      Iterator itr = states.iterator();
      while(itr.hasNext()) {
         str = (String) itr.next();
         System.out.println("The capital of " + str + " is " +
            capitals.getProperty(str) + ".");
      }
      System.out.println();
   }
}
/*Input and Output
F:\BHAVAJAVA\Collections>javac PropertiesDemo.java
F:\BHAVAJAVA\Collections>java PropertiesDemo
The capital of Karnataka is Bengaluru.
The capital of Tamil Nadu is Chennai.
The capital of Telangana is Hyderabad.
The capital of Andhra Pradesh is Amaravathi.
The capital of Kerala is Tiruvanantapuramu.
*/
```

OHM SRI SAIRAM

Collection Framework-Concluding Remarks:

- The Collections Framework gives you, the programmer, a powerful set of well-engineered solutions to some of programming's most common tasks.
- Collections will applicable to manage corporate databases, mailing lists, or inventory systems in the program.
- Finally, the collection framework provides an excellent platform to manage large amount of data in systematic manner inside the program.

StringTokenizer

- StringTokenizer is a class defined in java.util package to split the input string into discrete parts or tokens.
- StringTokenizer implements the Enumeration interface. Therefore, given an input string, you can enumerate the individual tokens contained in it using StringTokenizer.
- To use StringTokenizer, you specify an input string and a string that contains delimiters.Delimiters are characters that separate tokens.
- The StringTokenizer constructors are shown here:

StringTokenizer(String str) **creates StringTokenizer with specified string.**
StringTokenizer(String str, String delim) **creates StringTokenizer with specified string and delimeter**.
StringTokenizer(String str, String delim, boolean returnValue) **creates StringTokenizer with specified string, delimeter and returnValue. If return value is true, delimiter characters are considered to be tokens. If it is false, delimiter characters serve to separate tokens.**

Methods of StringTokenizer class

Public method	Description
boolean hasMoreTokens()	checks if there is more tokens available.
String nextToken()	returns the next token from the StringTokenizer object.
String nextToken(String delim)	returns the next token based on the delimeter.
boolean hasMoreElements()	same as hasMoreTokens() method.
Object nextElement()	same as nextToken() but its return type is Object.
int countTokens()	returns the total number of tokens.

OHM SRI SAIRAM

```
import java.util.*;
class StringTokenizerDemo{
      public static void main(String args[]){
        Scanner sin=new Scanner(System.in);
        System.out.println("Read line of integers separated by , symbol");
        String line=sin.next();
        int sum=0;
        StringTokenizer st=new StringTokenizer(line,",");
        while(st.hasMoreTokens()){
            String str=st.nextToken();
            int ele=Integer.parseInt(str);
            sum=sum+ele;
        }
       System.out.println("Sum of integers="+sum);
      }
}
/*Input and Output
F:\BHAVAJAVA\Collections>javac StringTokenizerDemo.java
F:\BHAVAJAVA\Collections>java StringTokenizerDemo
Read line of integers separated by , symbol
1,2,3,4,5,6,7,8,9
Sum of integers=45
*/
```

Scanner

- **Scanner** is a well-defined class defined in java.util package.
- It reads formatted input and converts it into its binary form.
- **Scanner** can be used to read input from the console, a file, a string, or any source that implements the **Readable** interface.

Constructors:

Constructor and Description
Scanner(File **source**) **Constructs a new Scanner that produces values scanned from the specified file.**
Scanner(InputStream **source**) **Constructs a new Scanner that produces values scanned from the specified input stream.**
Scanner(Path **source**) **Constructs a new Scanner that produces values scanned from the specified file.**
Scanner(Readable **source**) **Constructs a new Scanner that produces values scanned from the specified source.**
Scanner(String **source**) **Constructs a new Scanner that produces values scanned from the specified string.**

OHM SRI SAIRAM

Methods:

Return Type	Method name
boolean	**hasNext**() Returns true if this scanner has another token in its input.
boolean	**hasNextBoolean**() Returns true if the next token in this scanner's input can be interpreted as a boolean value using a case insensitive pattern created from the string "true|false".
boolean	**hasNextByte**() Returns true if the next token in this scanner's input can be interpreted as a byte value in the default radix using the **nextByte()** method.
boolean	**hasNextByte**(int radix) Returns true if the next token in this scanner's input can be interpreted as a byte value in the specified radix using the **nextByte()** method.
boolean	**hasNextDouble**() Returns true if the next token in this scanner's input can be interpreted as a double value using the **nextDouble()** method.
boolean	**hasNextFloat**() Returns true if the next token in this scanner's input can be interpreted as a float value using the **nextFloat()** method.
boolean	**hasNextInt**() Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the **nextInt()** method.
boolean	**hasNextInt**(int radix) Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the **nextInt()** method.
boolean	**hasNextLine**() Returns true if there is another line in the input of this scanner.
boolean	**hasNextLong**() Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the **nextLong()** method.
boolean	**hasNextLong**(int radix) Returns true if the next token in this scanner's input can be interpreted as a long value in the specified radix using the **nextLong()** method.
boolean	**hasNextShort**()

OHM SRI SAIRAM

	Returns true if the next token in this scanner's input can be interpreted as a short value in the default radix using the **nextShort()** method.
boolean	**hasNextShort**(int radix) Returns true if the next token in this scanner's input can be interpreted as a short value in the specified radix using the **nextShort()** method.
String	**next**() Finds and returns the next complete token from this scanner.
String	**next**(**String** pattern) Returns the next token if it matches the pattern constructed from the specified string.
BigDecimal	**nextBigDecimal**() Scans the next token of the input as a **BigDecimal**.
BigInteger	**nextBigInteger**() Scans the next token of the input as a **BigInteger**.
BigInteger	**nextBigInteger**(int radix) Scans the next token of the input as a **BigInteger**.
boolean	**nextBoolean**() Scans the next token of the input into a boolean value and returns that value.
byte	**nextByte**() Scans the next token of the input as a byte.
byte	**nextByte**(int radix) Scans the next token of the input as a byte.
double	**nextDouble**() Scans the next token of the input as a double.
float	**nextFloat**() Scans the next token of the input as a float.
int	**nextInt**() Scans the next token of the input as an int.
int	**nextInt**(int radix) Scans the next token of the input as an int.
String	**nextLine**() Advances this scanner past the current line and returns the input that was skipped.

OHM SRI SAIRAM

long	**nextLong**() Scans the next token of the input as a long.
long	**nextLong**(int radix) Scans the next token of the input as a long.
short	**nextShort**() Scans the next token of the input as a short.
short	**nextShort**(int radix) Scans the next token of the input as a short.
void	**remove**() The remove operation is not supported by this implementation of Iterator.
Scanner	**reset**() Resets this scanner.
void	**close**() Closes this scanner.

ANNAMACHARYA INSTITUTE OF TECHNOLGY AND SCIENCES::RAJAMPET
(AUTONOMOUS)
DEPARTMENT OF CSE

YEAR & SEM: II B.TECH II SEM **ACADEMIC YEAR: 2020-21**
BRANCH &SEC: CSE
SUBJECT: OBJECT ORIENTED PROGRAMMING USING JAVA
ASSIGNMENT NO: 5 **TOPICS COVERED: UNIT-V**

****

1.	Narrate the importance of collection framework. Explain how JDK5 changed collection framework
2.	What is List interface.Explain important methods in List interface.
3.	a)Compare and contrast between Set and List. b)How Set, SoretedSet and NavigableSet are differed.
4.	Demonstrate LinkedList and ArrayList with example programs.
5.	Differentiate between HashSet Vs LinkedHashSet Vs TreeSet Vs EnumSet.
6.	Explain LinkedHashSet with an example program.
7.	Explain Queue interface.
8.	How for each loop used to iterate elements from collection. Explain with an example
9.	What is the difference between Iterator and ListIterator. Demonstrate ListIterator with example program.
10.	Breifly explain Map classes.
11.	Demonstrate HashMap and TreeMap.
12.	Demonstrate Collection algorithms with example program.
13.	What is Arrays class? Demonstrate with example program.
14.	. Explain any two Legacy classes
15.	Write a Java program to read line of integers separated with comma symbol. Extract integer numbers into array(Hint: StringTokenizer). Find sorting of array elements using Arrays method.
16.	Demonstrate stack operations using Stack legacy class.
17.	Explain Properties class.
18.	How Scanner class helpful to read input of various types data. Explain with example program.