Core Java Material

JavaFullstackGuru

By Mr. Laxman

Chapter – 1

- Java Introduction
- Java Features
- Java Installation
- Java Program Execution Flow
- Java Programming Elements
- Java Programs Development
- Compilation
- Execution
- Translators
- JVM Architecture

What is Java

-> Java is a programming language

-> Java developed by James Gosling & his team in 1991 at Sun Microsystem Company

-> Initially they named it as 'OAK' programming language

-> In 1995, OAK language renamed to JAVA

Note: Oracle Corporation acquired Sun Microsystem in 2010. Now java is under license of Oracle corporation

-> Java is a free & open-source software

-> 3 billion devices run on Java language

-> Java is one of the most popular languages in the world

-> Java can run on any platform (It is platform independent)

Java is divided into 3 parts

1) J2SE / JSE (Java Standard Edition) → For stand-alone applications development

2) J2EE / JEE (Java Enterprise Edition) → For Web applications development

3) J2ME / JME (Java Micro / Mobile Edition) → For Mobile applications development

What we can develop using Java

-> Using Java, we can develop several kinds of applications like

1) Stand-alone applications
2) Web applications
3) Mobile Applications
4) Games
5) Servers
6) Databases and much more

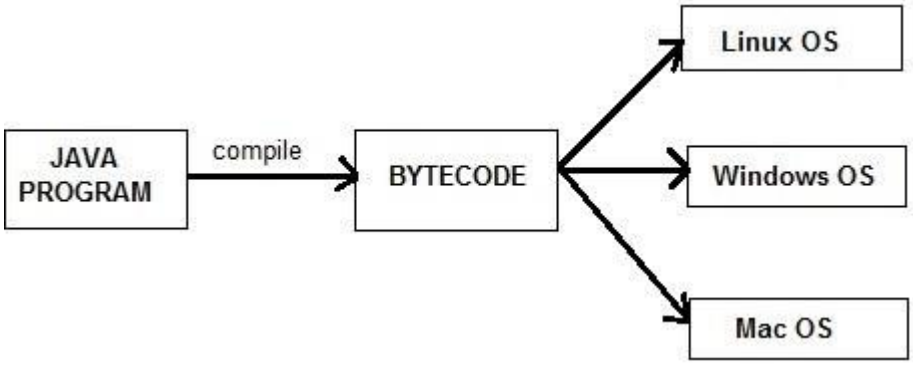
Java Features

1) Simple: Java is easy to learn and its syntax is quite simple, clean and easy to understand. The confusing and ambiguous concepts of C++ are either left out in Java or they have been re-implemented in a cleaner way.

Eg: Pointers and Operator Overloading are not there in java

2) Platform Independent: Unlike other programming languages such as C, C++ etc which are compiled into platform specific machines. Java is guaranteed to be write-once, run-anywhere language.

When we compile java code it will generate bytecode. This bytecode is platform independent and can be run on any machine, plus this bytecode format also provide security. Any machine with Java Runtime Environment can run Java Programs.



4) **OOP Language:** Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporate both data and behaviour.

Object-oriented programming (OOPs) is a methodology that simplifies software development and maintenance by providing some rules.

OOPs Principles are:

- ➔ Object
- ➔ Class
- ➔ Inheritance
- ➔ Polymorphism
- ➔ Abstraction
- ➔ Encapsulation

3) **Secure:** When it comes to security, Java is always the first choice. With java secure features it enables us to develop virus free, temper free system. Java program always runs in Java runtime environment with almost null interaction with system OS, hence it is more secure.

4) Multi-Threading: Java multithreading feature makes it possible to write program that can do many tasks simultaneously. Benefit of multithreading is that it utilizes same memory and other resources to execute multiple threads at the same time, like While typing, grammatical errors are checked along.

5) Architectural Neutral: Compiler generates bytecodes, which have nothing to do with a particular computer architecture, hence a Java program is easy to interpret on any machine.

6) Portable: Java Byte code can be carried to any platform. No implementation dependent features. Everything related to storage is predefined, example: size of primitive data types

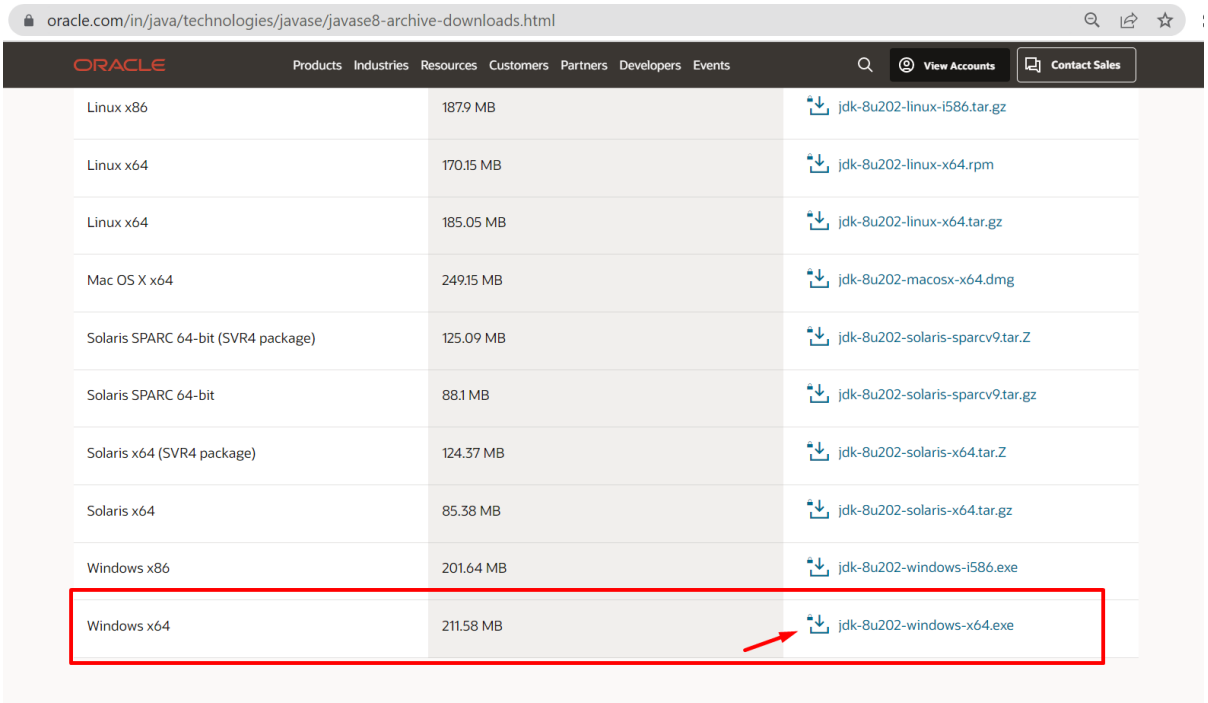
7) High Performance: Java is an interpreted language, so it will never be as fast as a compiled language like C or C++. But Java enables high performance with the use of just-in-time compiler.

8) Distributed: Java is also a distributed language. Programs can be designed to run on computer networks. Java has a special class library for communicating using TCP/IP protocols. Creating network connections is very much easy in Java as compared to C/C++.

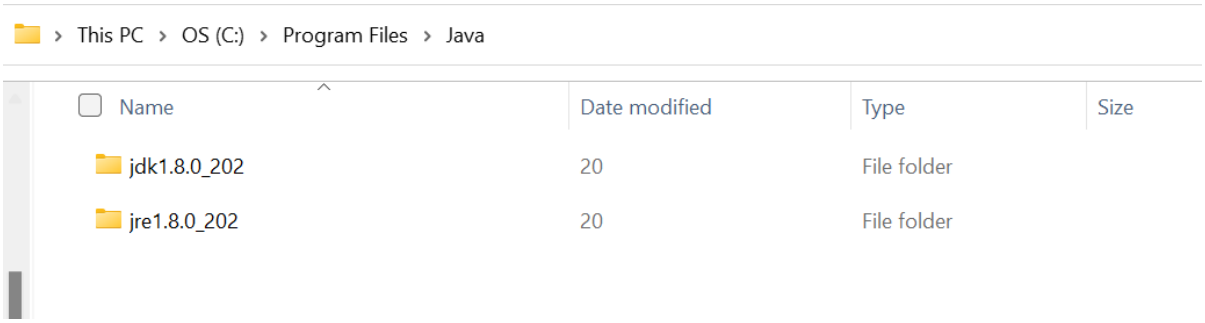
Java Slogan: WORA (Write Once Run Anywhere)

Environment Setup (Java Installation)

Step- 1) Download and Install Java Software



➔ After installing java, we can see below two folders (JDK & JRE)



Note: After installing java software, we need to set PATH for java software to use it.

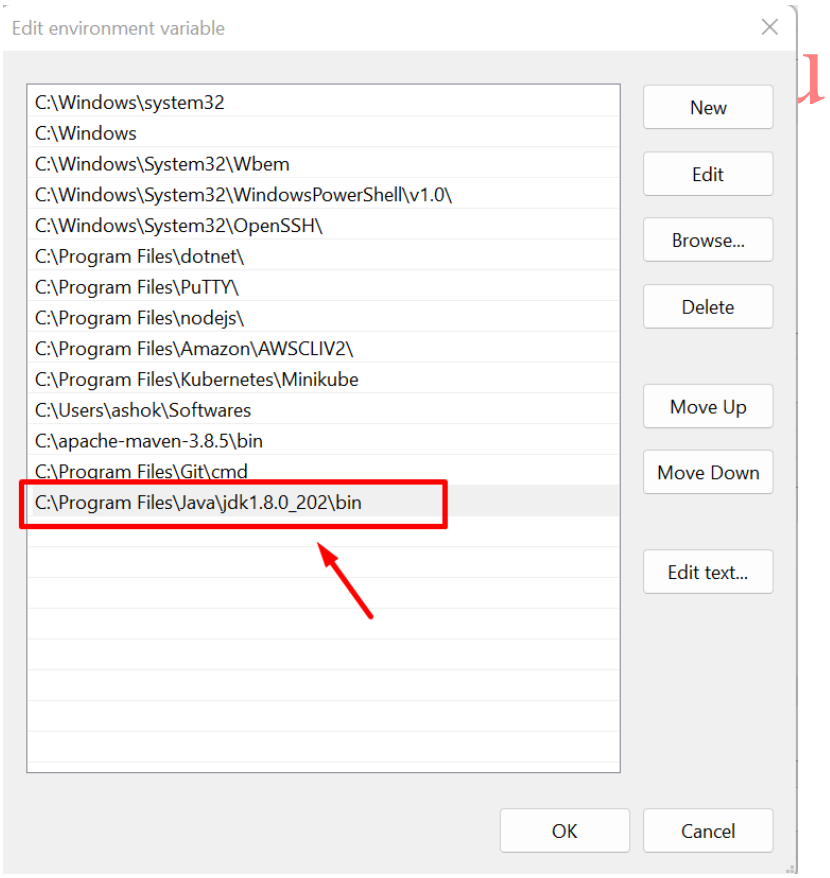
Step-2) Set Path for Java

-> Go To Environment Variables

-> Go To System Environment Variables

-> Edit Path

-> Add path up to JDK bin directory



Path = C:\Program Files\Java\jdk1.8.0_202\bin

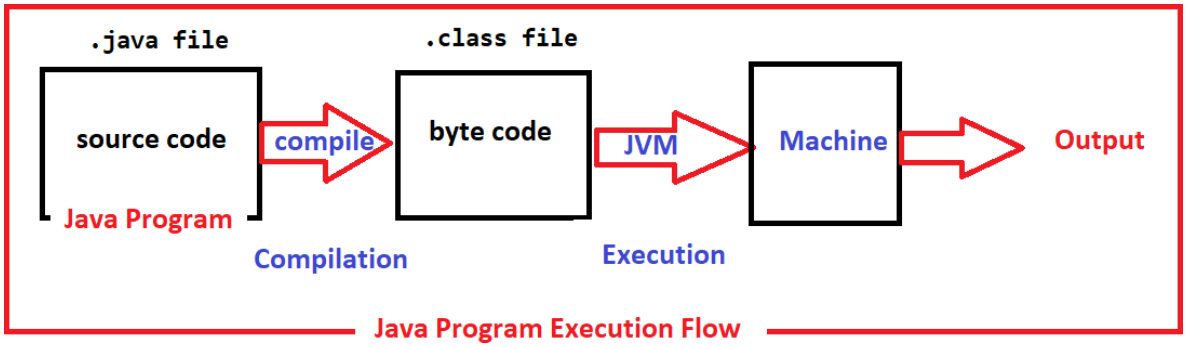
Step-3) Verify PATH Setup (Open command prompt and execute below command)

Note: If you are getting message like above that means, java installation and PATH setup is success.

Q) What is the difference between JDK, JRE & JVM?

- JDK contains set of tools to develop java programs
- JRE providing a platform to run our java programs
- JVM will take care of program execution, memory allocation & de-allocation

Java Program Execution Flow



Step-1: We will write source code and we will save that code in a file using .java extension

Step-2: We will compile source code using java compiler (it will generate byte code)

Step-3: We will execute .class file (JVM will convert byte code into machine code & gives output)

Java Program Structure

```
package statement

import statement(s)

class declaration

variables

methods
```

-> **Package statement** is used to group our classes. At most we can write only one package statement in one java program and package statement should be the first statement in the program.

-> **Import statements** are used to import one program into another program. We can write multiple import statements in one program (It depends on requirement). We should write import statements after package statement only.

-> **Class** is the most import part in java program. Class is a plan to define program elements. Inside class we will write variables and methods.

-> **Variables** are used to store the data. We can write any no. of variables inside a class.

-> **Methods** are used to perform action. Application business logic we will write inside methods. A can class can have any no. of methods.

Developing First Java Program

Step-1: Open any text editor (notepad / notepad ++ / edit plus)

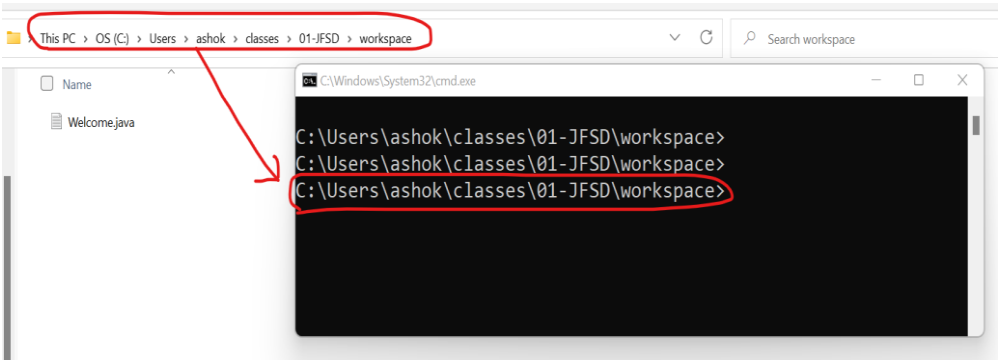
Step-2: Write below java program

```
class Welcome {

 public static void main(String... args){
      System.out.println("Welcome To Ashok IT");
 }
}
```

Welcome.java

Step-3: Save the program with .java extension (Filename: Welcome.java)

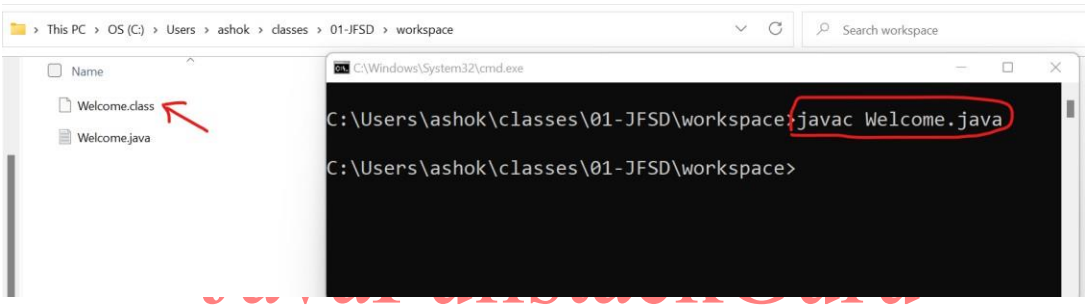
Step-4: Open Command prompt and change directory location to the place where our java program is available.



Step-5: Compile the Java program

Syntax:

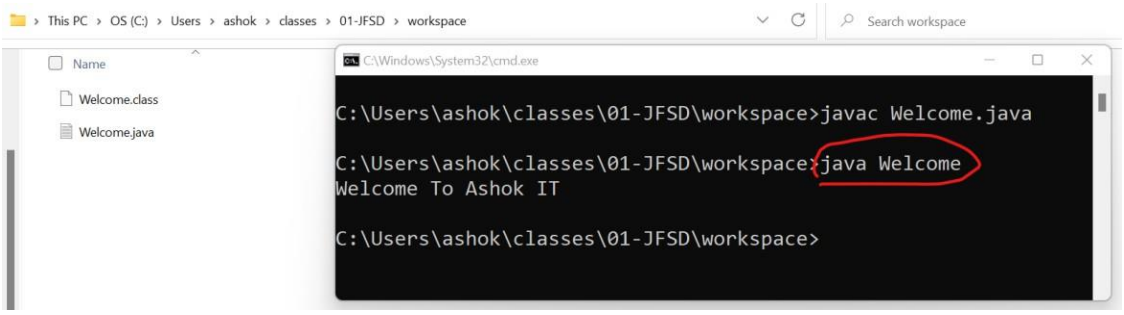
- **javac filename.java**



➔ We are able to see .class file that means our program compilation is successful.

Step-6: Run the java program

- **Java filename**



-> We are able to see “Welcome To Ashok IT” as output of our program that means our program executed successfully.

Note: In Realtime, we will use IDE to develop java programs/ project.

Translators

-> Translators are used to convert from one format to another format

-> We have 3 types of translators

1) Interpreter

2) Compiler

3) Assembler

-> Interpreter will convert the program line by line (performance is slow)

-> Compiler will convert all the lines of program at a time (performance is fast)

-> Assembler is used to convert assembler programming languages into machine language

JVM

-> JVM stands for Java Virtual Machine (We can't see with our eyes)

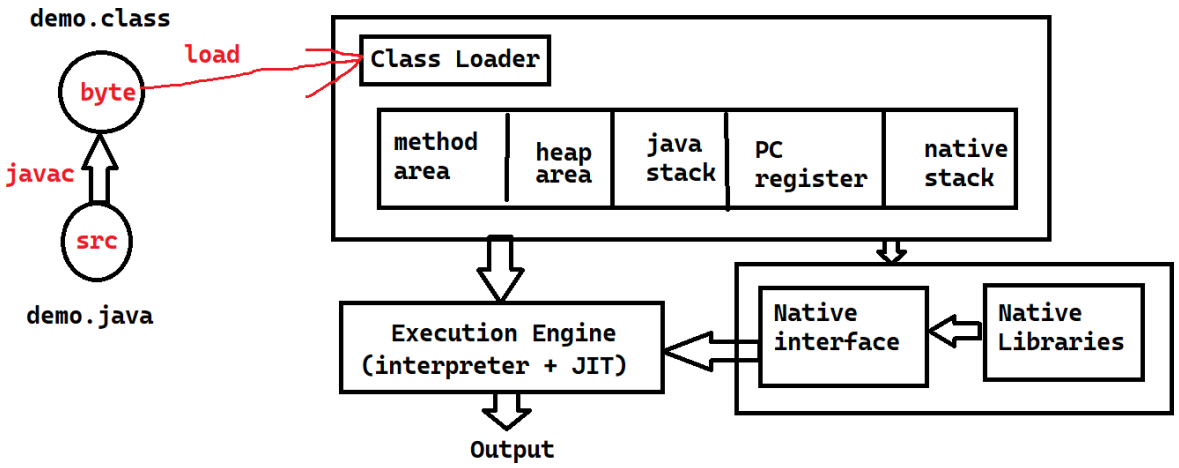
-> JVM will be part of JRE software

-> JVM is responsible for executing java programs

-> JVM will allocate memory required for program execution & de-allocate memory when it is not used

-> JVM will convert byte code into machine understandable format

JVM Architecture



Class loader: It will load .class file into JVM

Method Area: Class code will be stored here

Heap area: Objects will be stored into heap area

Java Stack: Method execution information will be stored here

PC Register: It will maintain next line information to execute

Native Stack: It will maintain non-java code execution information

Native Interface: It will load native libraries into JVM

Native Libraries: Non-java libraries which are required for native code execution

Execution Engine: It is responsible to execute the program and provide output/result. It will use Interpreter and JIT for execution.

Chapter - 2

- Variables
- Data Types
- Identifiers & Rules
- Reserved Words
- Java Coding Standards
- Java comments
- Reading Data From keyboard

Variables

-> variables are used to store the data during program execution

-> We need to specify type of the variable to store the data

-> To specify type of data we will use 'data types'

-> To declare the variable in Java, we can use following syntax

```
datatype variableName;
```

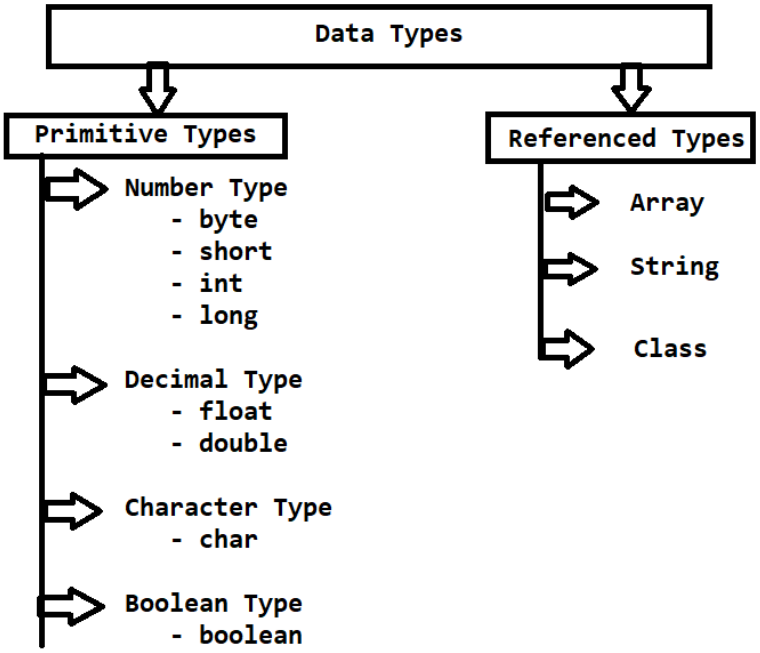
Data types

-> Data types are used to specify type of the data

-> Data types are divided into 2 categories

1) Primitive Data Types

2) Non-Primitive Data Types



Primitive Data Types

Primitive data types are divided into eight types

Primitive Data types							
char	boolean	byte	short	int	long	float	double

Once a primitive data type has been declared its type can never change, although in most cases its value can change. These eight primitive types can be put into four groups.

Integer

This group includes `byte`, `short`, `int`, `long`

byte : It is 1 byte(8-bits) integer data type. Value range from -128 to 127. Default value zero. example: `byte b=10;`

short : It is 2 bytes(16-bits) integer data type. Value range from -32768 to 32767. Default value zero. example: `short s=11;`

int : It is 4 bytes(32-bits) integer data type. Value range from -2147483648 to 2147483647. Default value zero. example: `int i=10;`

long : It is 8 bytes(64-bits) integer data type. Value range from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. Default value zero. example: `long l=100012;`

Floating Data Types

This group includes float and double

float : It is 4 bytes(32-bits) float data type. Default value 0.0f. example: `float ff=10.3f;`

double : It is 8 bytes(64-bits) float data type. Default value 0.0d. example: `double db=11.123;`

Character data type

-> Character data type is used to store single character

-> Any single character (alphabet / digit / special character) we can store using 'char' data type

-> char datatype will occupy 2 bytes of memory

-> When we are storing data into 'char' data type single quote is mandatory

char : It is 2 bytes(16-bits) unsigned unicode character. Range 0 to 65,535. example: `char c='a';`

Boolean data type

-> It is used to store true or false values only

-> It will occupy 1 bit memory

Note: 8 bits = 1 byte

-> default value for boolean is false

Ex: boolean flag = false

Variables

-> Variables are used to store the data / value

-> To store the data into variable we need to specify data type

-> To store data into variables we need to perform 2 steps

1) Variable Declaration (defining variable with data type)

Ex: byte age ;

2) Variable Initialization (storing value into variable)

Ex: age = abc;

-> We can complete declaration and initialization in single line

byte age = 20;

VarDemo.java

```
1 public class VarDemo {
2
3     public static void main(String[] args) {
4         int age = 20;
5         System.out.println(age);
6
7         float a = 25.01f;
8         System.out.println(a);
9
10        double price = 120.87;
11        System.out.println(price);
12
13        char gender = 'm';
14        System.out.println(gender);
15
16        boolean pass = true;
17        System.out.println(pass);
18
19    }
20 }
```

Identifiers in Java

All Java components require names. Name used for classes, methods, interfaces and variables are called Identifier.

Identifier must follow some rules. Here are the rules:

Rule-1 : The only allowed characters in java identifiers are:

- ➔ a to z
- ➔ A to Z
- ➔ 0 to 9
- ➔ $
- ➔ _ (underscore)

name-------> valid

name@----- > invalid

age#--------> invalid

age -----➔ valid

Rule-2 : Identifier should not start with digit (First letter shouldn't be digit)

1age-------> invalid

age2-------> valid

name3------> valid

_name-----> valid

$name------> valid

@name------ > invalid

$_amt ---------> valid

_1bill----------> valid

Rule-3: Java reserved keywords cannot be used as an identifier (53 reserved words available in java)

int byte = 20;--------- > invalid bcz byte is a reserved word

byte for = 25; --------> invalid bcz for is a reserved word

int try = 30;----------> invalid bcz try is a reserved word

long phno = 797979799 -----> valid

Rule-4 : Spaces are not allowed in identifiers

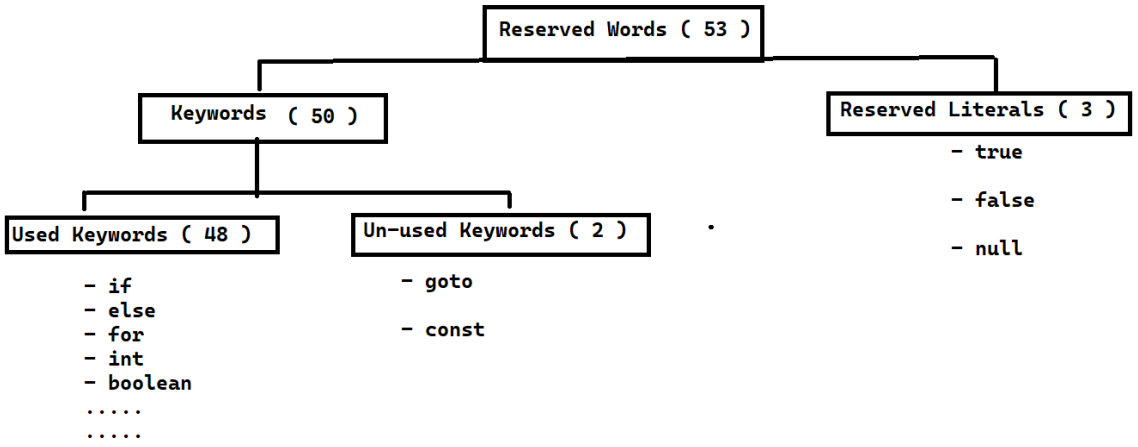
int mobile bill = 400; // invalid

int mobile_bill = 400 ; // valid

Rule - 5: Identifiers in Java are case sensitive; foo and Foo are two different identifiers.

Java Keywords

-> In Java we have total 53 reserved words, those are classified like below



JAVA RESERVED WORDS (53)					
Datatype keywords	Flow control keywords	Modifiers keywords	Exception handling keywords	class related keywords	Object related keywords
1) byte 2) short 3) char 4) int 5) long 6) float 7) double 8) boolean	9) if 10)else 11) do 12) while 13) for 14) switch 15) case 16) default 17) break 18) continue 19) return	20) public 21) private 22) protected 23) static 24) final 25) abstract 26) synchronized 27) strictfp 28) native 29) transient 30) volatile	31) try 32) catch 33) finally 34) throw 35) throws 36) assert	37) package 38) import 39) class 40) interface 41) extends 42) implements Unused keywords 47)const 48)goto	43) super 44) this 45) new 46) instanceof 49) void 50) enum 51) true 52) false 53) null

Java coding/naming conventions

-> Java language followed some standards/conventions for pre-defined packages, classes and methods....

-> Java language suggested java programmers also to follow same standards / conventions

-> Following these standards/conventions is not mandatory but highly recommended.

Following are different java coding conventions

1. coding convention for a class

-> Class name can contain any no.of words without spaces

-> Recommended to write every word first character as uppercase in class name

Sample Class Names	
class Hello { }	class UserManagementService{ }
class HelloWorld { }	class WelcomeRestController { }

2. coding convention for interfaces

-> Class & Interface Naming conventions are same

Sample Interface Names	
interface Demo { }	interface UserService { }
interface Report { }	interface UserDao{ }

3. coding convention for a variables

-> Variable name can have any no. of words without spaces

-> Recommended to start variable name with lowercase letter

-> If variable name contains multiple words, then recommended to write first word all characters in lowercase and from second word onwards every word first character in Uppercase

Sample Variable Names
int age ; int userAge; long creditCardNumber ;

4. coding convention for methods

-> Method name can have any no. of words without spaces

-> Recommended to start method name with lowercase letter

-> If method name contains multiple words, then recommended to write first word all characters in lowercase and from second word onwards every word first character in Uppercase

Sample Method Names	
main ( ) { } 	saveUser( ) { }
save ( ) { } 	getWelcomeMsg ( ) { }

Note: Variables & Methods naming conventions are same. But methods will have parenthesis (( )) variables will not have parenthesis.

5. coding convention for constants

-> Constant means fixed value (value will not change, it is fixed)

-> Recommended to write constant variable all characters in uppercase

-> If constant variable contains multiple words recommended to use _ (underscore) with all uppercase characters

Sample Constant Variable Names

int MIN_AGE = 21;
int MAX_AGE = 60 ;
int PI = 3.14;

6. coding convention for packages

-> Package name can have any no.of characters & any of words

-> Recommended to use only lowercase letters in package names

-> If package name contains multiple words then we will use . (dot) to separate words

Sample package names
java.lang
java.io
java.util
in.ashokit
in.ashokit.service
com.oracle

Java Comments

-> In java we have 3 types of comments

-> Commented code will not participate in compilation & execution

Note: Java compiler will skip commented lines of code

1) single line comments

Syntax:

```
// this is single line comment
```

2) multi line comments (when we don't want to compile multiple lines of code)

```
/*
        commented code
*/
```

3) documentation comments (Used to generate API documentation)

```
/**
 *
 * @author ashok
 *
 */
```

String data type

-> String is a predefined class available in java.lang package

-> String can be used as a data type also

-> String is used to store group of characters

Note: Every java class can be used as a data type (referenced data type)

```
String name = "ashok";

String email = "ashok@gmail.com"

String country = "India";

String pwd = "ashoK@123";
```

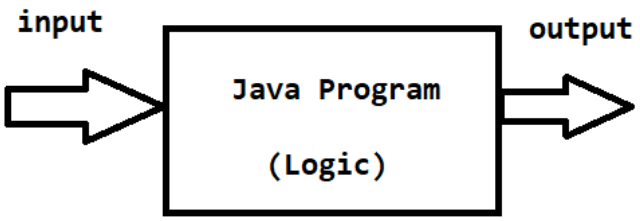
-> To store data in String variable double quotes are mandatory

Reading Data from Keyboard

-> We can pass input for java program to perform some operation. Based on given input we can execute some logic and we can send output.

Ex -1: Take 2 numbers as input and provide sum of those numbers

Ex- 2: Take first name & last name and provide full name



-> In java we have 2 classes to read data from Keyboard

1) java.io.BufferedReader

2) java.util.Scanner

// write a java program to perform sum of two numbers Using Scanner

```
4
5  public class Demo {
6
7      public static void main(String[] args) {
8
9          Scanner sc = new Scanner(System.in); // Obj creation
10
11         System.out.println("Enter First Name");
12         int i = sc.nextInt(); // reading first number
13
14         System.out.println("Enter Second Number");
15         int j = sc.nextInt(); // reading second number
16
17         int sum = i + j; // sum of given numbers
18
19         // printing result
20         System.out.print("Result : " + sum);
21
22         sc.close();
23     }
24 }
25
```

// Write a java program to take two strings as input

```
3  import java.util.Scanner;
4
5  public class Demo {
6
7      public static void main(String[] args) {
8
9          Scanner sc = new Scanner(System.in); // Obj creation
10
11         System.out.println("Enter First Name");
12         String fname = sc.next(); // reading first name
13
14         System.out.println("Enter Second Number");
15         String lname = sc.next(); // reading second name
16
17         String fullnae = fname + lname;
18
19         // printing result
20         System.out.print(fullnae);
21
22         sc.close();
23     }
24 }
25
```

Knowledge Check

1) What is full stack development?

2) Explain software project architecture

3) What are the roles & responsibilities of full stack developer

4) What is database and why we need it?

5) What is programming language & why we need programming language?

6) What is JAVA?

7) What are the features of java?

8) What is the difference between C and Java?

9) What type of applications we can develop using java & brief them

10) What is the difference between JDK, JRE and JVM?

11) What is the execution flow of java program?

12) What is the difference between interpreter and compiler?

13) Write JVM architecture & explain JVM components

14) What is JIT?

15) Write Java data types with size, range and default values

16) What is variable and how to create variables

17) Explain Java program elements

18) Write a java program to print welcome message

19) Write a java program on variables declaration, initialization

20) How many types of comments available in java

21) What is identifier and what are rules available for identifier

22) What are the reserved words in java

23) Write Java Naming Conventions for packages, classes, variables and methods

Chapter – 3

- Operators
  - Arithmetic
  - Relational
  - Logical
  - Assignment
  - New
  - Dot (.) Operator

- Control Statements
  - Conditional Statements
  - Looping Statements
  - Transfer Statements

Operators

-> Operator is a symbol which tells to the compiler to perform some operation.

-> Java provides a rich set of operators do deal with various types of operations.

-> Sometimes we need to perform arithmetic operations then we use plus (+) operator for addition, multiply (*) for multiplication etc.

-> Operators are always essential part of any programming language.

-> Java operators can be divided into following categories:

- Arithmetic operators
- Relation operators
- Logical operators
- Assignment operators
- Conditional operators
- Misc. operators

Arithmetic operators

Arithmetic operators are used to perform arithmetic operations like: addition, subtraction etc and helpful to solve mathematical expressions.

The below table contains Arithmetic operators.

Operator	Description
+	adds two operands
-	subtract second operands from first
*	multiply two operand
/	divide numerator by denumerator
%	remainder of division
++	Increment operator increases integer value by one
--	Decrement operator decreases integer value by one

// Write a java program on Athematic Operators

```
public class Demo {

    public static void main(String[] args) {

        int a = 10;
        int b = 2;

        System.out.println(a + b);

        System.out.println(a - b);

        System.out.println(a * b);

        System.out.println(a / b);

        System.out.println(a % b);

    }
}
```

Increment & Decrement Operators

In programming (Java, C, C++, JavaScript etc.), the increment operator ++ increases the value of a variable by 1. Similarly, the decrement operator -- decreases the value of a variable by 1.

```
a = 5

++a ;   // a becomes 6

a++ ;   // a becomes 7

--a ;   // a becomes 6

a-- ;   // a becomes 5
```


Post-Increment (Ex: a ++): Value is first used for computing the result and then incremented.

IncrementOperator.java

```
public class IncrementOperator {

    public static void main(String[] args) {
        // 1. Post-Increment Operator
        int a = 5;
        int b = 7;

        // Here 'a' will not incremented immediately, a++ will still return value 5.
        int c = a++ + b;
        // c = 5 + 7 and this will evaluate to 12.

        System.out.println("Post- Increment \n c = " + c);
    }
}
```

Pre-Increment (Ex: ++ a): Value is incremented first and then the result is computed.

IncrementOperator.java

```java
public class IncrementOperator {

    public static void main(String[] args) {
        // 2. Pre-Increment Operator

        int A = 5;
        int B = 7;

        // Here 'a' will be incremented immediately, and ++a will return value 6.
        int C = ++A + B;
        // C = 6 + 7 and this will evaluate to 13.

        System.out.println("Pre- Increment \n C = " + C);
    }
}

```

Post-decrement (Ex: a--): Value is first used for computing the result and then decremented.

DecrementOperator.java

```java
public class DecrementOperator {

    public static void main(String[] args) {
        // 1. Post-Decrement Operator
        int a = 5;
        int b = 7;

        // Here 'a' will not decremented immediately, a-- will still return value 5.
        int c = a-- + b;
        // c = 5 + 7 and this will evaluate to 12.

        System.out.println("Post- Decrement \n c = " + c);
    }
}

```

Pre-decrement (Ex: --a): Value is decremented first and then the result is computed.

DecrementOperator.java

```java
public class DecrementOperator {

    public static void main(String[] args) {
        // 2. Pre-Decrement Operator

        int A = 5;
        int B = 7;

        // Here 'a' will be decremented immediately, and --a will return value 4.
        int C = --A + B;
        // C = 4 + 7 and this will evaluate to 11.

        System.out.println("Pre- Decrement \n C = " + C);
    }
}

```

Facts about Increment and Decrement operators

- Can only be applied to variables only
- Nesting of both operators is not allowed
- They are not operated over final variables
- Increment and Decrement Operators cannot be applied to boolean.

Relational operators

Relational operators are used to test comparison between operands or values. It can be used to test whether two values are equal or not equal or less than or greater than etc.

The following table shows all relation operators supported by Java.

Operator	Description
==	Check if two operand are equal
!=	Check if two operand are not equal.
>	Check if operand on the left is greater than operand on the right
<	Check operand on the left is smaller than right operand
>=	check left operand is greater than or equal to right operand
<=	Check if operand on left is smaller than or equal to right operand

```
class Operations {

public static void main(String as[]) {
        int a, b;

        a=40;
        b=30;

        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) );

    }
}
```

Operations.java

Logical operators

Logical Operators are used to check conditional expression. For example, we can use logical operators in if statement to evaluate conditional based expression. We can use them into loop as well to evaluate a condition.

Java supports following 3 logical operators. Suppose we have two variables whose values are: a=true and b=false.

Operator	Description	Example
&&	Logical AND	(a && b) is false
||	Logical OR	(a || b) is true
!	Logical NOT	(!a) is false

```
class LogicalOperators {

   public static void main(String as[]) {

        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));

    }
}
```

LogicalOperators.java

Assignment Operators

Assignment operators are used to assign a value to a variable. It can also be used combine with arithmetic operators to perform arithmetic operations and then assign the result to the variable. Assignment operator supported by Java are as follows:

Operator	Description	Example
=	assigns values from right side operands to left side operand	a = b
+=	adds right operand to the left operand and assign the result to left	a+=b is same as a=a+b
-=	subtracts right operand from the left operand and assign the result to left operand	a-=b is same as a=a-b
*=	mutiply left operand with the right operand and assign the result to left operand	a*=b is same as a=a*b
/=	divides left operand with the right operand and assign the result to left operand	a/=b is same as a=a/b
%=	calculate modulus using two operands and assign the result to left operand	a%=b is same as a=a%b

```
1  class AssignmentOperators {
2
3  public static void main(String as[]) {
4
5      int a = 30;
6      int b = 10;
7      int c = 0;
8
9      c = a + b;
10     System.out.println(c);
11
12     c += a ;
13     System.out.println(c);
14
15     c -= a ;
16     System.out.println(c );
17
18     c *= a ;
19     System.out.println(c);
20
21     a = 20;
22     c = 25;
23     c /= a ;
24     System.out.println(c);
25
26     a = 20;
27     c = 25;
28     c %= a ;
29     System.out.println("c %= a  = " + c );
30
31     }
32 }
33
```

Conditional operator

It is also known as ternary operator because it works with three operands. It is short alternate of if-else statement. It can be used to evaluate Boolean expression and return either true or false value

Syntax : epr1 ? expr2 : expr3

In ternary operator, if **epr1** is true then expression evaluates after **question mark (?)** else evaluates **after colon (:)**. See the below example.

```
1
2  class ConditionalOperator {
3
4      public static void main(String as[]) {
5
6          int a, b;
7          a = 20;
8
9          b = (a == 1) ? 30: 40;
10         System.out.println( "Value of b is : " +  b );
11
12         b = (a == 20) ? 30: 40;
13         System.out.println( "Value of b is : " + b );
14     }
15 }
```


instanceOf operator

It is a java keyword and used to test whether the given reference belongs to provided type or not. Type can be a class or interface. It returns either true or false.

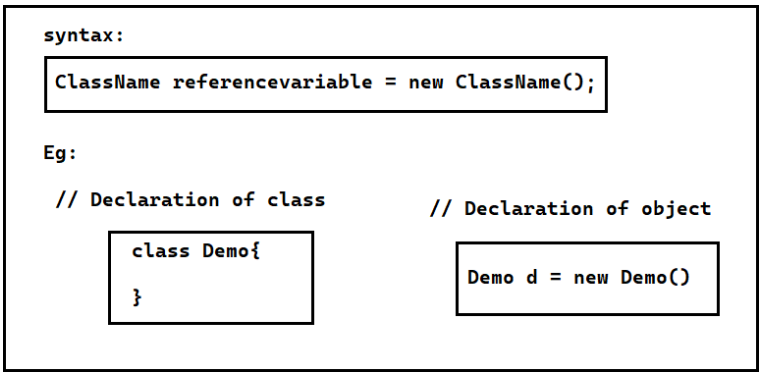
Example:

Here, we created a string reference variable that stores “Ashok IT”. Since it stores string value so we test it using instance operator to check whether it belongs to string class or not. See the below example.

```
1
2  class instanceofOperator {
3
4      public static void main(String as[]) {
5
6          String a = "Ashok IT";
7
8          boolean b = a instanceof String;
9
10         System.out.println( b );
11
12     }
13 }
14
```

new operator

- new is java keyword or operator which is used to create the object
- we can create an object for both user defined classes and predefine classes
- creating an object nothing but allocating the memory so that we can use in the application.
- once an object created that will be located inside the Heap memory of JVM.



. (Dot) operator

-> This operator used to access the members of the class using reference or column like follows

Eg:

reference.variable

reference.method()

ClassName.variable

ClassName.method()

-> This operator can also be used to refer or identify the class from a package

Eg:

java.lang.NoSuchMethodError

java.lang.String

.....

Control statements (flow control)

-> In java we can write any number of statements which are executed in sequence order by default

-> But if we want to execute java statements according to our requirement then we have to use control statements

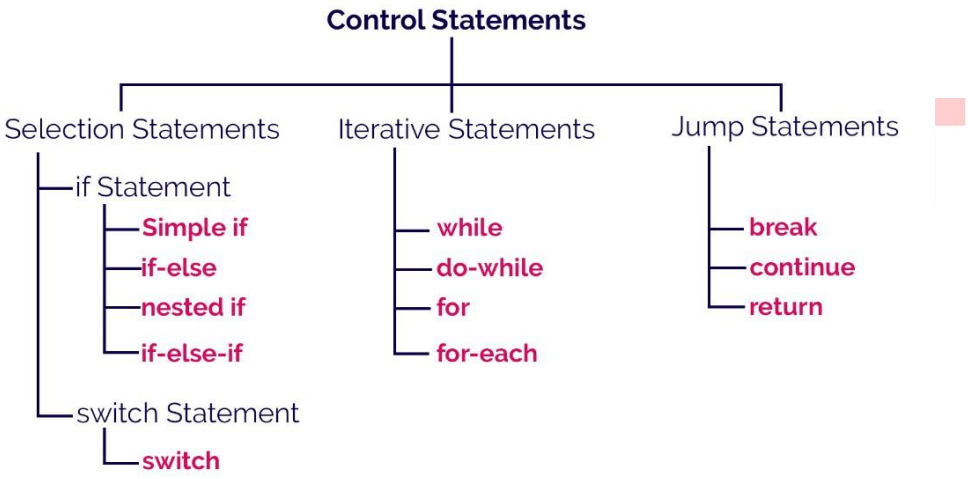
-> These statements decide whether a specific part of the code will be executed or not.

Types of Control Flow Statements

-> There are different types of control statements available in Java for different situations or conditions.

-> In java, we can majorly divide control statements into three major types:

1. Selection / Conditional statements
2. Loop statements
3. Jump / Branching / Transfer statements



Conditional / Selection Statements

-> Conditional statements are used to execute group of statements based on condition

-> Conditional statements will evaluate boolean expression to make the decision

Simple 'if' statement in java

-> In java, we use if statement to test a condition and decide the execution of a block of statements based on that condition result.

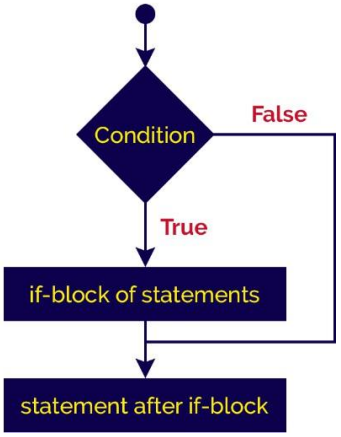
-> If the condition is True, then the block of statements is executed and if it is false, then the block of statements will be ignored.

The syntax and execution flow of if the statement is as follows.

Syntax

```
if(condition){
        if-block of statements;
        ...
}
statement after if-block;
...
```

Flow of execution



Java Program on 'Simple if condition'

SimpleIfDemo.java

```
1  import java.util.Scanner;
2
3  public class SimpleIfDemo {
4
5      public static void main(String[] args) {
6
7          Scanner read = new Scanner(System.in);
8          System.out.print("Enter any number: ");
9
10         int num = read.nextInt();
11
12         if ((num % 5) == 0) {
13             System.out.println("We are inside the if-block!");
14             System.out.println("Given number is divisible by 5!!");
15         }
16         System.out.println("We are outside the if-block!!!");
17     }
18 }
19
```

Console | Problems | Debug Shell

```
<terminated> SimpleIfDemo [Java Application] C:\Program Files\Java\jre1.
Enter any number: 15
We are inside the if-block!
Given number is divisible by 5!!
We are outside the if-block!!!
```

if-else statement in java

-> In java, we use the if-else statement to test a condition and pick the execution of a block of statements out of two blocks based on that condition result.

-> The if-else statement checks the given condition then decides which block of statements to be executed based on the condition result.

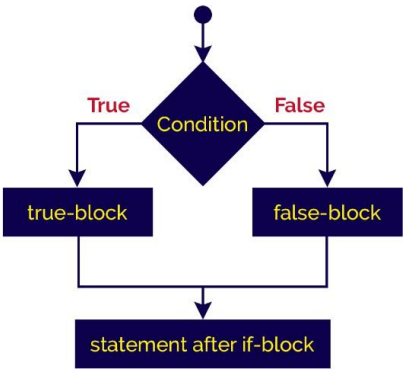
-> If the condition is True, then the true block of statements is executed and if it is False, then the false block of statements is executed.

-> The syntax and execution flow of if-else statement is as follows.

Syntax

```
if(condition){
        true-block of statements;
        ...
}
else{
        false-block of statements;
        ...
}
statement after if-block;
...
```

Flow of execution



Java Program on If – else statements

```
1  import java.util.Scanner;
2
3  public class IfElseDemo {
4
5      public static void main(String[] args) {
6          Scanner read = new Scanner(System.in);
7          System.out.print("Enter any number: ");
8
9          int num = read.nextInt();
10
11         if((num % 2) == 0) {
12             System.out.println("We are inside the true-block!");
13             System.out.println("Given number is EVEN number!!");
14         }
15         else {
16             System.out.println("We are inside the false-block!");
17             System.out.println("Given number is ODD number!!");
18         }
19         System.out.println("We are outside the if-block!!!");
20     }
21 }
```

```
Enter any number: 24
We are inside the true-block!
Given number is EVEN number!!
We are outside the if-block!!!
```

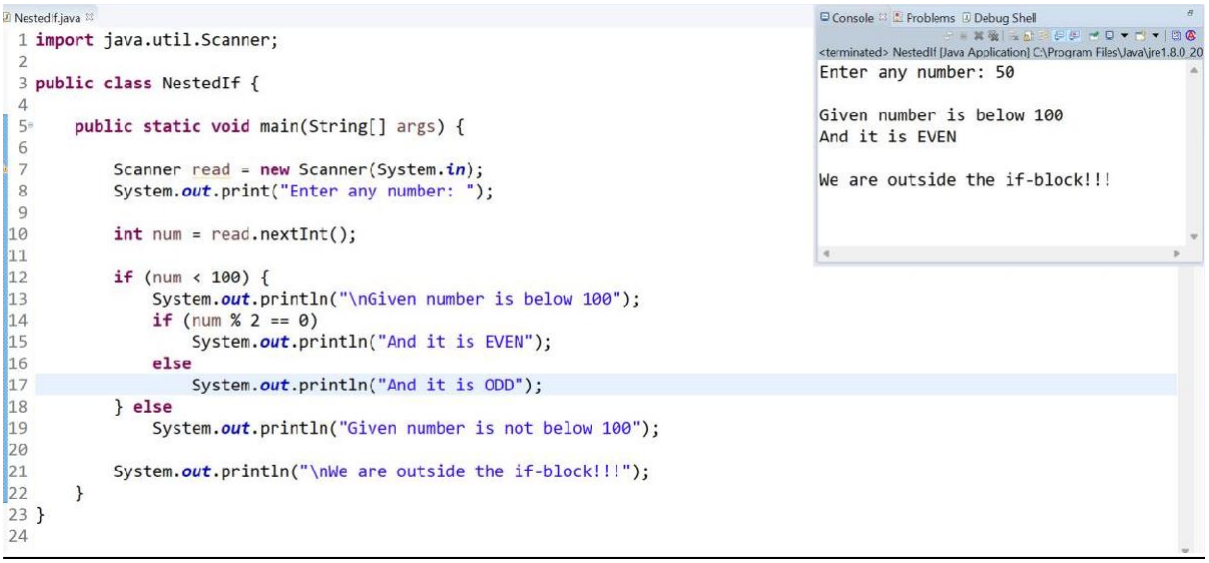
Nested if statement in java

Writing an if statement inside another if-statement is called nested if statement.

The general syntax of the nested if-statement is as follows.

```
if ( condition_1 ) {

    if(condition_2){
        inner if-block of statements;
        ...
    }
    ...
}
```

Java Program on Nested-If statement



```
Nestedif.java
1 import java.util.Scanner;
2
3 public class NestedIf {
4
5     public static void main(String[] args) {
6
7         Scanner read = new Scanner(System.in);
8         System.out.print("Enter any number: ");
9
10        int num = read.nextInt();
11
12        if (num < 100) {
13            System.out.println("\nGiven number is below 100");
14            if (num % 2 == 0)
15                System.out.println("And it is EVEN");
16            else
17                System.out.println("And it is ODD");
18        } else
19            System.out.println("Given number is not below 100");
20
21        System.out.println("\nWe are outside the if-block!!!");
22    }
23 }
24
```

```
Console  Problems  Debug Shell
<terminated> NestedIf [Java Application] C:\Program Files\Java\jre1.8.0_20
Enter any number: 50

Given number is below 100
And it is EVEN

We are outside the if-block!!!
```

if...else if...else statement

if…else if statements will be used when we need to compare the value with more than 2 conditions.

They are executed from top to bottom approach. As soon as the code finds the matching condition, that block will be executed. But if no condition is matching then the last else statement will be executed.

```
if ( condition_1 ) {

   Statement 1; //if condition_1 becomes true then this will be executed

} else if ( condition_2 ) {

   Statement 2; // if condition_2 becomes true then this will be executed

} else {

   Statement 3; //executed when no matching condition found

}
```

Java Program on if-else-if – else statements

IfelseIfDemo.java

```
1
2  public class IfelseIfDemo {
3
4      public static void main(String args[]) {
5          int x = 2;
6
7          if (x > 10) {
8              System.out.println("i is greater than 10");
9          } else if (x < 10) {
10             System.out.println("i is less than 10");
11         } else {
12             System.out.println("i is 10");
13         }
14         System.out.println("After if else if ladder statement");
15     }
16
17 }
```

Console | Problems | Debug Shell

<terminated> IfelseIfDemo [Java Application] C:\Program Files\Java\jre1.8

```
i is less than 10
After if else if ladder statement
```

Assignment

1) Write a java program to check given number is even number or odd number

2) Write a java program to check given number is prime number or not

3) Write a java program to check given number is divisible by 5 or not

4) Write a java program to check given character is alphabet or digit

Switch statement

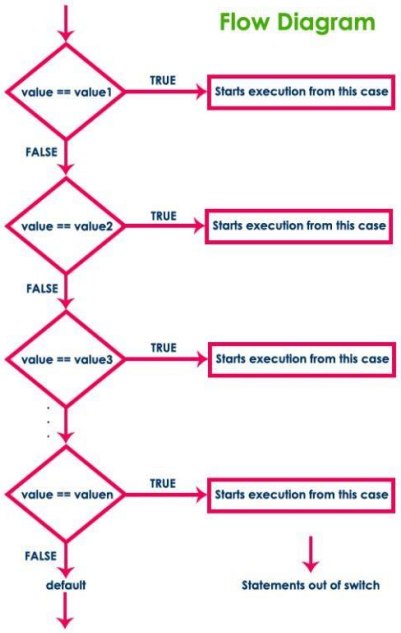
-> Java switch statement compares the value and executes one of the case blocks based on the condition.

-> It is same as if…else if ladder. Below are some points to consider while working with switch statements:

- **There can be one or N number of cases**
- **The values in the case must be unique**
- **case value must be of the same type as expression used in switch statement**
- **each case statement can have break statement (it is optional)**

Syntax

```
switch ( expression or value )
{
      case  value1:  set of statements;
                     ....
      case  value2:  set of statements;
                     ....
      case  value3:  set of statements;
                     ....
      case  value4:  set of statements;
                     ....
      case  value5:  set of statements;
                     ....
      .
      .
      default: set of statements;
}
```




```
switch(expression) {
case value1:
   //code for execution;
   break;  //optional
case value2:
 // code for execution
 break;  //optional
......
......
......
......
Case value n:
// code for execution
 break;  //optional

default:
 code for execution when none of the case is true;
}
```

Write a java program on 'Switch' case

SwitchDemo.java

```
1
2  public class SwitchDemo {
3
4      public static void main(String[] args) {
5          int day = 3;
6          String dayName;
7          switch (day) {
8          case 1:
9              dayName = "Today is Monday"; break;
10         case 2:
11             dayName = "Today is Tuesday"; break;
12         case 3:
13             dayName = "Today is Wednesday"; break;
14         case 4:
15             dayName = "Today is Thursday"; break;
16         case 5:
17             dayName = "Today is Friday"; break;
18         case 6:
19             dayName = "Today is Saturday"; break;
20         case 7:
21             dayName = "Today is Sunday"; break;
22         default:
23             dayName = "Invalid day"; break;
24         }
25         System.out.println(dayName);
26     }
27 }
28
```


Looping Statements in Java

-> Loop is an important concept of a programming that allows to iterate over the sequence of statements.

-> Loop is designed to execute particular code block till the specified condition is true or all the elements of a collection (array, list etc) are completely traversed.

-> The most common use of loop is to perform repetitive tasks. For example, if we want to print table of a number then we need to write print statement 10 times. However, we can do the same with a single print statement by using loop.

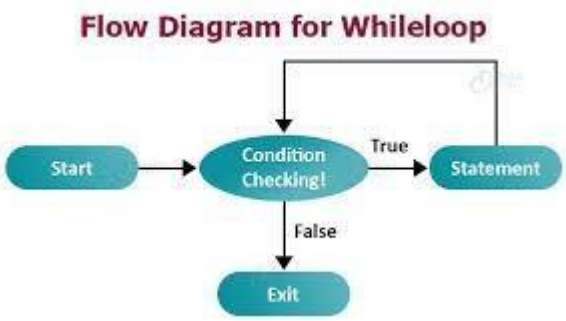
-> Loop is designed to execute its block till the specified condition is true.

Java provides mainly three loops based on the loop structure.

1. **for loop**
2. **while loop**
3. **do while loop**

while statement in java

The while loop is used to execute a single statement or block of statements repeatedly as long as the given condition is TRUE. The while statement is also known as Entry control looping statement. The syntax and execution flow of while statement is as follows.



Write a java program on 'while' loop

WhileDemo.java

```
1  public class WhileDemo {
2
3      public static void main(String[] args) {
4
5          int i = 1;
6
7          while (i <= 10) {
8              System.out.println(i);
9              i++;
10         }
11     }
12 }
13
14
```


do-while statement in java

The do-while loop is used to execute a single statement or block of statements repeatedly as long as given the condition is TRUE. The do-while statement is also known as the Exit control looping statement. The do-while statement has the following syntax.

Write a java program on do-while loop

DoWhileDemo.java

```
1  public class DoWhileDemo {
2
3      public static void main(String[] args) {
4          int i = 1;
5
6          do {
7              System.out.println(i);
8              i++;
9
10         } while (i <= 10);
11     }
12
13 }
14
```

for loop in java

The for loop is used to execute a single statement or a block of statements repeatedly as long as the given condition is TRUE.

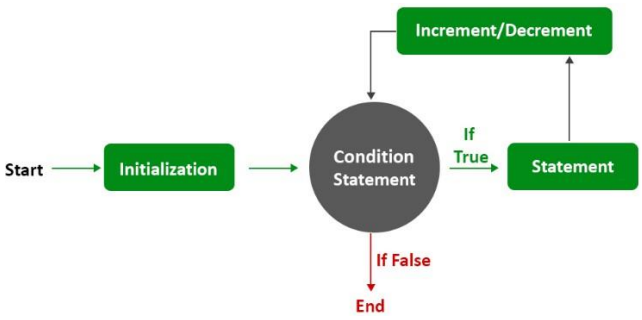
To create a for loop, we need to set the following parameters.

1) Initialization: It is the initial part, where we set initial value for the loop. It is executed only once at the starting of loop. It is optional, if we don't want to set initial value.

2) Condition: It is used to test a condition each time while executing. The execution continues until the condition is false. It is optional and if we don't specify, loop will be infinite.

3) Statement: It is loop body and executed every time until the condition is false.

4) Increment/Decrement: It is used for set increment or decrement value for the loop.

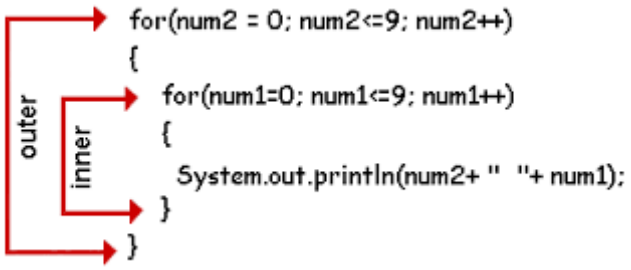


ForLoopDemo.java

```
1  public class ForLoopDemo {
2
3      public static void main(String[] args) {
4
5          int n = 2;
6
7          for (int i = 1; i <= 10; i++) {
8              System.out.println(n + "*" + i + "=" + n * i);
9          }
10     }
11 }
12
```

Nested For Loop

➔ Writing a loop inside another loop is called as Nested For Loop



Write a java program on 'Nested For Loop'

NestedForLoopDemo.java

```
1  public class NestedForLoopDemo {
2
3      public static void main(String[] args) {
4
5          for (int i = 1; i <= 5; i++) {
6
7              for (int j = 1; j <= i; j++) {
8                  System.out.print("* ");
9              }
10
11             System.out.println();
12         }
13     }
14 }
15
```

for-each loop in java

-> The Java for-each statement was introduced since Java 5.0 version.

-> It provides an approach to traverse through an array or collection in Java.

-> The for-each statement also known as enhanced for statement.

-> The for-each statement executes the block of statements for each element of the given array or collection.

ForEachDemo.java

```
1 public class ForEachDemo {
2
3     public static void main(String[] args) {
4         int a[] = { 20, 21, 22, 23, 24 };
5
6         for (int i : a) {
7             System.out.println(i);
8         }
9
10    }
11 }
12
```

Branching / Transfer Statements

-> Transferring statements are the control statements which are used to transfer the control position from 1 location to another location

-> In Java we have following 3 types of transfer statements

1) break

2) continue

3) return

Break statement

-> In Java, break is a statement that is used to break current execution flow of the program.

-> We can use break statement inside loop, switch case etc.

-> If break is used inside loop then it will terminate the loop.

-> If break is used inside the innermost loop then break will terminate the innermost loop only and execution will start from the outer loop.

-> If break is used in switch case then it will terminate the execution after the matched case.

BreakDemo.java

```
1 public class BreakDemo {
2
3     public static void main(String[] args) {
4
5         for (int i = 1; i <= 10; i++) {
6             if (i == 8) {
7                 break;
8             }
9             System.out.println(i);
10        }
11    }
12 }
13
```

Continue Statement

-> In Java, the continue statement is used to skip the current iteration of the loop. It jumps to the next iteration of the loop immediately.

-> We can use continue statement with for loop, while loop and do-while loop as well.

ContinueDemo.java

```java
public class ContinueDemo {

    public static void main(String[] args) {

        for (int i = 1; i <= 10; i++) {

            if (i == 5) {
                continue;
            }

            System.out.println(i);
        }
    }
}
```

Return statement

-> return is a transferring statement which is used to stop the continuity of method execution.

ReturnDemo.java

```java
public class ReturnDemo {

    public static void main(String[] args) {
        System.out.println("Hi");
        if (10 > 2) {
            return;
        }
        // dead code
        System.out.println("Bye!!!");
    }
}
```

Logical Programs - Assignment

Q-1) Write a java program to read shoes brand name from keyboard, based on brand name print brand slogan like below

Nike -> Just do it

Adidas -> Impossible is nothing

Puma -> Forever Faster

Reebok -> I Am What I Am

Q-2) Write a java program to read person basic salary and calculate Provident Fund amount from the basic salary

Formula: Provident Fund is 12 % of Basic Salary

Q-3) Write a java program to read person age and person salary and print his eligibility for marriage

Condition: If person age less than 30 and salary greater than 1 lakh then eligible for marriage

Q-4) Write a java program to find factorial of given number

Q-5) Write a java program to print mathematical table of given number

Q-6) Write a java program to print numbers from 100 to 1

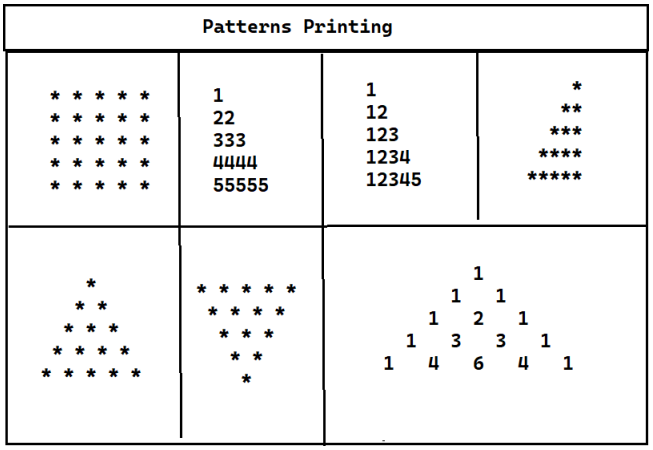
Q-7) Write a java program to find sum of 1 - 100 numbers

Q-8) Write a java program to find sum of digits of given number

Q-9) Write a java program to check given number is Armstrong number or not

Q-10) Write a java program to print first 100 prime numbers

Q-11) Write java programs to print below patterns



Q-12) Write a java program to print Descending order Pattern

5

5 4

5 4 3

5 4 3 2

5 4 3 2 1

Knowledge Check

1) What is Operator and why we need operators?

2) List down all the operators available in Java?

3) What is the difference between Relational & Logical Operators?

4) What is the purpose of 'new' operator?

5) What is the purpose of 'dot(.)' operator?

6) What is the purpose of instance of operator?

7) What is the difference between "=" and "==" operators?

8) What is the purpose of Control Statements & List down all the control statements available in java?

9) Write a java program on if - else if - else ladder

10) Write a java program on 'switch' case

11) What is the difference between while & do-while loops

12) Write a java program on 'while' loop

13) What is the difference between 'while' loop & 'for' loop

14) Write a java program on 'for' loop

15) Write a java program on 'nested for loop'

16) What is the difference between 'break' and 'continue' & 'return' keywords

17) Write a java program using 'break' keyword

18) Write a java program using 'continue' keyword

Chapter – 4

- Arrays
- String
- StringBuffer
- StringBuffer
- Command Line Args

Arrays

-> Array is an object which contains elements of similar data type

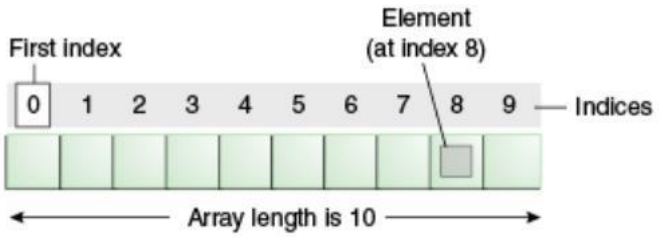
-> Array is a container object that hold values of homogeneous type.

-> Array is also known as static data structure because size of an array must be specified at the time of its declaration.

-> Array starts from zero index and goes to n-1 where n is length of the array.

-> Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on.

-> In Java, array is treated as an object and stores into heap memory. It allows to store primitive values or reference values.



Types of Arrays in java

There are two types of arrays in java

1) Single Dimensional Array

2) Multidimensional Array

Single Dimensional Array in Java

Syntax to Declare an Array in Java

```
dataType[] arr; (or)

dataType []arr; (or)

dataType arr[];
```

Instantiation of an Array in Java

```
arrayRefVar = new datatype[size];
```

-> Instantiation is a process of allocating memory to an array. At the time of instantiation, we specify the size of array to reserve memory area.

ArrayDemo.java

```
1  public class ArrayDemo {
2
3      public static void main(String[] args) {
4
5          int[] arr = new int[5];
6
7          for (int x : arr) {
8              System.out.println(x);
9          }
10     }
11 }
12
```

-> In the above example, we created an array arr of int type and can store 5 elements. We iterate the array to access its elements and it prints five times zero to the console. It prints zero because we did not set values to array, so all the elements of the array initialized to 0 by default.

Set Array Elements

We can set array elements either at the time of initialization or by assigning direct to its index.

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

-> Here, we are assigning values at the time of array creation. It is useful when we want to store static data into the array.

-> We can set value based on index position also

```
arr[1] = 105
```

ArrayDemo.java

```java
public class ArrayDemo {

    public static void main(String[] args) {

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

        for (int x : arr) {
            System.out.println(x);
        }

        // assigning a value
        arr[1] = 105;

        System.out.println("element at first index: " + arr[1]);
    }
}
```

Accessing array element

We can access array elements by its index value. Either by using loop or direct index value. We can use loop like: for, for-each or while to traverse the array elements.

ArrayDemo.java

```java
public class ArrayDemo {

    public static void main(String[] args) {

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

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

        System.out.println("element at first index: " + arr[1]);
    }
}
```

Length Of Array In Java

The length of an array indicates the number of elements present in the array. Unlike C/C++, where we use ‘sizeof’ operator to get the length of the array, Java array has ‘length’ property. We will explore more on this property later.

ArrayDemo.java

```java
public class ArrayDemo {

    public static void main(String[] args) {
        int[] myArray = { 1, 3, 5, 7 };
        System.out.println("Size of the given array : " + myArray.length);
    }
}
```

Multi-Dimensional Array

A multi-dimensional array is very much similar to a single dimensional array. It can have multiple rows and multiple columns unlike single dimensional array, which can have only one row index.

It represents data into tabular form in which data is stored into row and columns.

Multi-Dimensional Array Declaration
datatype[ ][ ] arrayName;

Initialization of Array
datatype[ ][ ] arrayName = new int[no_of_rows][no_of_columns];

ArrayDemo.java

```java
public class ArrayDemo {

    public static void main(String[] args) {

        int arr[][] = { { 1, 2, 3, 4, 5 }, { 6, 7, 8, 9, 10 }, { 11, 12, 13, 14, 15 } };
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 5; j++) {
                System.out.print(arr[i][j] + " ");
            }
            System.out.println();
        }
        // accessing a value
        System.out.println("element at first row and second column: " + arr[0][1]);
    }
}

```

Logical Programs On Arrays

1) Write a java program to print array elements

2) Write a java program to calculate sum of array

3) Write a java program to reverse array elements

4) Write a java program to print min and max elements in array

5) Write a java program to print second min and second max elements in array

Strings

-> String is an object that represents sequence of characters.

Ex: "hello", "ashokit"

-> In Java, String is represented by String class which is available java.lang package

-> One important thing to notice about string object is that string objects are immutable that means once a string object is created it cannot be changed.

How to create String object in Java?

-> To handle string data in Java, we need an object of string class. Basically, there are three ways to create a String object in Java.

1) By string literal.

2) By new keyword.

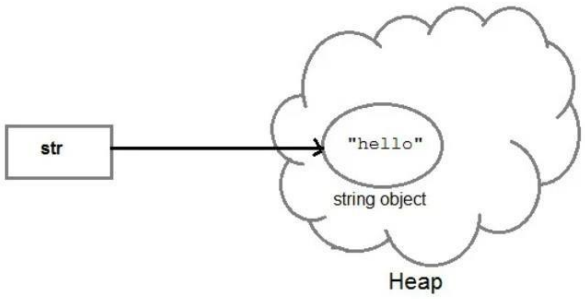
3) By converting character arrays into strings

Working with String literal

-> String literal in Java is created by using double quotes.

String str = "hello";

-> The string literal is always created in the string constant pool.



-> In Java, String constant pool is a special area that is used for storing string objects.

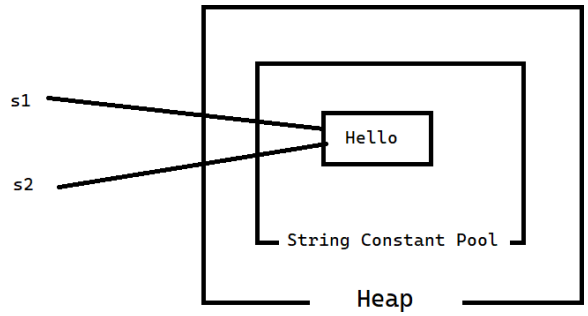
-> Whenever we create a string literal in Java, JVM checks string constant pool first. If the string already exists in string constant pool, no new string object will be created in the string pool by JVM.

-> JVM uses the same string object by pointing a reference to it to save memory. But if string does not exist in the string pool, JVM creates a new string object and placed it in the pool.

For example:

String s1 = "Hello";

String s2 = "Hello";



Creating String Object by using new Keyword

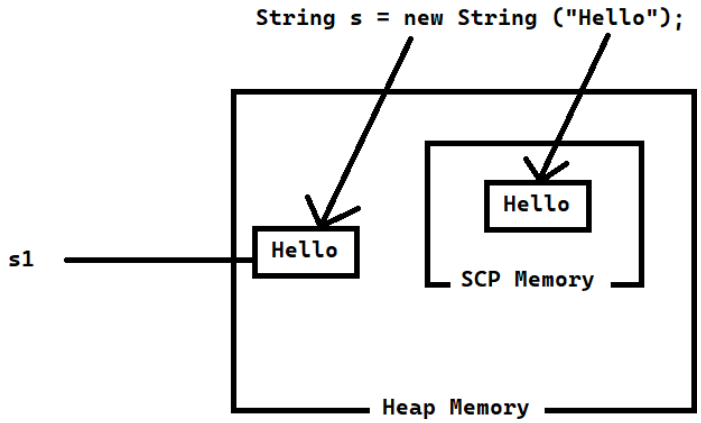
-> The second way of creating an object to string class is by using new operator.

-> It is just like creating an object of any class. It can be declared as follows:

String s = new String("Hello");

-> Whenever we will create an object of string class using the new operator, JVM will create two objects. First, it will create an object in the heap area and store string “Hello” into the object.

-> After storing data into memory, JVM will point a reference variable s to that object in the heap. Allocating memory for string objects is shown in the below figure.



-> Now JVM will create the second object as a copy for literal “Hello” in string constant pool for future purposes. There is no explicit reference variable pointing to the copy object in the pool but internally, JVM maintains the reference variable implicitly.

-> Remember that the object created in the SCP area is not eligible for garbage collection because implicitly, the reference variable will be maintained by JVM itself.

By converting Character Arrays into String

-> The third way to create strings is by converting the character arrays into string. Let’s take a character type array: arr[ ] with some characters as given below:

```
char arr[ ] = {'j','a','v','a'};
```

-> Now create a string object by passing array name to string constructor like this:

```
String s = new String(arr);
```

-> Now string object ‘s’ contains string “java”. It means that all the characters of the array are copied into string.

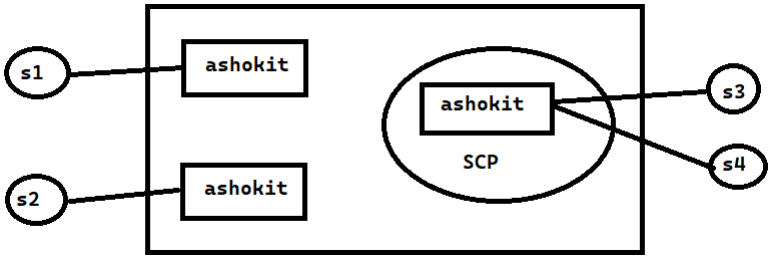
-> If you do not want all the characters of the array into string then you can also mention which character you need, like this:

```
String s = new String(arr, 1,3);
```


-> From the above statement, total of three characters will be copied into string s. Since the original characters are j-a-v-a. So, the counting will start from 0 i.e 0th character in the array is ‘j’ and the first character is ‘a’. Starting from ‘a’, total of three characters ‘aja’ will copy into string s.

How many total objects will be created in memory for following string objects?

```
String s1 = new String("ashokit");

String s2 = new String("ashokit");

String s3 = "ashokit";

String s4 = "ashokit";
```



1. During the execution of first statement using new operator, JVM will create two objects, one with content in heap area and another as a copy for literal “ashokit” in the SCP area for future purposes as shown in the figure.

The reference variable s1 is pointing to the first object in the heap area.

2. When the second statement will be executed, for every new operation, JVM will create again one new object with content “ashokit” in the heap area.

But in the SCP area, no new object for literal will be created by JVM because it is already available in the SCP area. The s2 is pointing to the object in the heap area as shown in the figure.

3. During the execution of third and fourth statements, JVM will not create a new object with content “ashokit” in the SCP area because it is already available in string constant pool.

It simply points the reference variables s3 and s4 to the same object in the SCP. They are shown in the above figure.

Thus, three objects are created, two in the heap area and one in string constant pool.

Why String Objects are given as immutable objects

An object can be referred by multiple reference variables in this case if string objects are mutable objects, then we change the content of object automatically other references get also modified so that string objects are given as immutable objects it means whenever any operation is done on strings it will create new object.

String Manipulations

String class provided several methods to perform operations on Strings

#1) Length

The length is the number of characters that a given string contains. String class has a length() method that gives the number of characters in a String.

Demo.java

```
1 public class Demo {
2
3     public static void main(String[] args) {
4
5         String str = "ashokit";
6         System.out.println(str.length());
7     }
8 }
9
```

#2) concatenation

Although Java uses a '+' operator for concatenating two or more strings. A concat() is an inbuilt method for String concatenation in Java.

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          String str1 = "ashok";
6
7          String str2 = "it";
8
9          String str3 = str1.concat(str2);
10
11         System.out.println(str3);
12
13     }
14 }
```

#3) String to CharArray()

This method is used to convert all the characters of a string into a Character Array. This is widely used in the String manipulation programs.

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          String str = "ashokit";
6
7          char[] charArray = str.toCharArray();
8
9          System.out.println(charArray);
10     }
11 }
12
```

#4) String charAt()

This method is used to retrieve a single character from a given String.

The syntax is given as:

```
char charAt(int i);
```

The value of 'i' should not be negative and it should specify the location of a given String i.e. if a String length is 5, then the value of 'i' should be less than 5.

```java
public class Demo {

    public static void main(String[] args) {

        String str = "ashokit";

        char charAt = str.charAt(5);

        System.out.println(charAt);
    }
}
```

#5) Java String compareTo()

This method is used to compare two Strings. The comparison is based on alphabetical order. In general terms, a String is less than the other if it comes before the other in the dictionary.

```java
public class Demo {

    public static void main(String[] args) {

        String str1 = "ashok";

        String str2 = "it";

        int compareTo = str1.compareTo(str2);

        System.out.println(compareTo);
    }
}
```

#6) String contains()

This method is used to determine whether a substring is a part of the main String or not. The return type is Boolean.

For E.g. In the below program, we will check whether “it” is a part of “ashokit” or not and we will also check whether “blog” is a part of “ashokit”.

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          String str1 = "ashokit";
6
7          boolean contains1 = str1.contains("it");
8          System.out.println(contains1);
9
10         boolean contains2 = str1.contains("blog");
11         System.out.println(contains2);
12     }
13 }
14
```

#7) Java String split()

As the name suggests, a split() method is used to split or separate the given String into multiple substrings separated by the delimiters (“”, “ ”, \\, etc).

In the below example, we will split the String (Thexyzwebsitexyzisxyzashokitxyzhelp) using a chunk of String(xyz) already present in the main String.

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          String str1 = "Thexyzwebsitexyzisxyzashokitxyzhelp";
6
7          String[] split = str1.split("xyz");
8
9          for(String str : split) {
10
11             System.out.println(str);
12         }
13     }
14 }
15
```

#8) Java String indexOf()

This method is used to perform a search operation for a specific character or a substring on the main String. There is one more method known as lastIndexOf() which is also commonly used.

indexOf() is used to search for the first occurrence of the character.

lastIndexOf() is used to search for the last occurrence of the character.

Demo.java

```
public class Demo {

    public static void main(String[] args) {

        String str1 = "ashokit..learn here..lead anywhere..!!";

        System.out.println(str1.indexOf("a"));

        System.out.println(str1.lastIndexOf("e"));
    }
}

```

#9) Java String toString()

This method returns the String equivalent of the object that invokes it. This method does not have any parameters. Given below is the program where we will try to get the String representation of the object.

Demo.java

```
public class Demo {

    public static void main(String[] args) {

        String str1 = "ashokit..learn here..lead anywhere..!!";

        String string = str1.toString();

        System.out.println(string);
    }
}

```

#10) String replace ()

The replace() method is used to replace the character with the new characters in a String.

Demo.java

```
public class Demo {

    public static void main(String[] args) {

        String str1 = "Hello World";

        String replace = str1.replace("World", "Java");

        System.out.println(replace);
    }
}
```

#12) Substring Method ()

The Substring() method is used to return the substring of the main String by specifying the starting index and the last index of the substring.

For Example, in the given String “ashokitjavaclasses”, we will try to fetch the substring by specifying the starting index and the last index.

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          String str = "ashokitjavaclasses";
6
7          String substring = str.substring(0, 7);
8
9          System.out.println(substring);
10     }
11 }
12
```

StringBuffer Class

-> StringBuffer class is used to create a mutable string object. It means, it can be changed after it is created.

-> It is similar to String class in Java both are used to create string, but stringbuffer object can be changed.

-> StringBuffer class is used when we have to make lot of modifications to our string.

-> It is also thread safe i.e multiple threads cannot access it simultaneously.

StringBuffer defines 4 constructors.

StringBuffer(): It creates an empty string buffer and reserves space for 16 characters.

StringBuffer(int size): It creates an empty string and takes an integer argument to set capacity of the buffer.

StringBuffer(String str): It creates a stringbuffer object from the specified string.

StringBuffer(charSequence []ch): It creates a stringbuffer object from the charsequence array.

Creating StringBuffer Object

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          StringBuffer sb = new StringBuffer("Ashok");
6          System.out.println(sb);
7
8          // modifying object
9          sb.append("IT");
10
11         System.out.println(sb); // Output: AshoKIT
12     }
13 }
```

Difference Between String & StringBuffer

Demo.java

```
1  public class Demo {
2
3      public static void main(String args[]) {
4          String str = "ashok";
5          str.concat("it");
6          System.out.println(str); // Output: ashok
7
8          StringBuffer strB = new StringBuffer("ashok");
9          strB.append("it");
10         System.out.println(strB); // Output: ashokit
11     }
12 }
```

Note: In the above program Output is such because String objects are immutable objects. Hence, if we concatenate on the same String object, it won't be altered But StringBuffer creates mutable objects. Hence, it can be altered.

StringBuffer class methods

#1) append()

This method will concatenate the string representation of any type of data to the end of the StringBuffer object

Demo.java

```
1 public class Demo {
2
3     public static void main(String[] args) {
4
5         StringBuffer sb1 = new StringBuffer("ashok");
6
7         sb1.append("it");
8
9         System.out.println(sb1);
10    }
11 }
12
```

#2) insert() :

This method inserts one string into another. Here are few forms of insert() method

```
StringBuffer insert(int index, String str)

StringBuffer insert(int index, int num)

StringBuffer insert(int index, Object obj)
```

Demo.java

```
1 public class Demo {
2
3     public static void main(String args[]) {
4         StringBuffer str = new StringBuffer("test");
5         str.insert(2, 123);
6         System.out.println(str);
7     }
8 }
```


#3) reverse()

This method reverses the characters within a StringBuffer object.

Demo.java

```
1 public class Demo {
2
3     public static void main(String[] args) {
4
5         StringBuffer str = new StringBuffer("ashokit");
6         str.reverse();
7         System.out.println(str);
8
9     }
10 }
11
```

#4) replace()

This method replaces the string from specified start index to the end index.

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          StringBuffer str = new StringBuffer("Hello World");
6          str.replace(6, 11, "java");
7          System.out.println(str);
8
9      }
10 }
11
```

#5) capacity()

This method returns the current capacity of StringBuffer object.

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          StringBuffer str = new StringBuffer();
6          System.out.println( str.capacity() );
7
8      }
9  }
10
```

Java StringBuilder class

-> StringBuilder is identical to StringBuffer except for one important difference that it is not synchronized, which means it is not thread safe.

-> StringBuilder also used for creating string object that is mutable and non synchronized.

-> The StringBuilder class provides no guarantee of synchronization.

-> StringBuffer and StringBuilder both are mutable but if synchronization is not required then it is recommended to use StringBuilder class.

-> This class is located into java.lang package and signature of the class is as:

StringBuilder Constructors

StringBuilder (): creates an empty StringBuilder and reserves room for 16 characters.

StringBuilder (int size): create an empty string and takes an integer argument to set capacity of the buffer.

StringBuilder (String str): create a StringBuilder object and initialize it with string str.

StringBuilder (CharSequence seq): It creates stringbuilder object by using CharSequence object.

Working with StringBuilder class

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          StringBuilder obj = new StringBuilder("Welcome to ");
6          obj.append("ashokit");
7
8          System.out.println(obj);
9      }
10 }
```

-> When we want a mutable String without thread-safety then StringBuilder should be used

-> When we want a mutable String with thread-safety then StringBuffer should be used

-> When we want an Immutable object then String should be used.

Logical Programs on Strings

1) Write a java program to combine two strings

2) Write a java program to reverse a String

3) Write a java program to check given String is a palindrome or not

4) Write a java program remove all occurrences of a given character from String

5) Write a program to count number of words in a String

6) Write a java program to compare two strings

7) Write a java program to check first string present in second string

8) Write a java program to find first non-repeated character in given String

Command Line Arguments

-> Command-line arguments in Java are used to pass arguments to the main program.

-> If you look at the Java main method syntax, it accepts String array as an argument.

Demo.java

```
1 public class Demo {
2
3     public static void main(String[] args) {
4
5         //logic
6     }
7 }
8
```

-> When we pass command-line arguments, they are treated as strings and passed to the main method in the string array argument.

-> The arguments have to be passed as space-separated values.

-> We can pass strings and primitive data types as command-line arguments.

-> The arguments will be converted to strings and passed into the main method string array argument.

Java Program with Command Line Arguments

Demo.java

```
1  public class Demo {
2
3      public static void main(String[] args) {
4
5          // convert into integer type
6          int number1 = Integer.parseInt(args[0]);
7          System.out.println("First Number: " + number1);
8
9          // convert into integer type
10         int number2 = Integer.parseInt(args[1]);
11         System.out.println("Second Number: " + number2);
12
13         int result = number1 + number2;
14         System.out.println("Addition of two numbers is: " + result);
15     }
16 }
17
```

Passing Command Line Arguments to Above Program

C:\Windows\System32\cmd.exe

```
Microsoft Windows [Version 10.0.22000.978]
(c) Microsoft Corporation. All rights reserved.

C:\Users\ashok\classes\01-JFSD\workspace>javac Demo.java

C:\Users\ashok\classes\01-JFSD\workspace>
C:\Users\ashok\classes\01-JFSD\workspace>
C:\Users\ashok\classes\01-JFSD\workspace>java Demo 10 20
First Number: 10
Second Number: 20
Addition of two numbers is: 30

C:\Users\ashok\classes\01-JFSD\workspace>
```

Chapter- 5: OOPS (Part – 1)

- OOPS Introduction
- Classes
- Objects
- Variables
  - Instance
  - Static
  - Local
- Methods
- Constructor

OOPS

-> Programming languages are divided into 2 types

1) Procedure Oriented Languages

Ex: C, Cobol, Pascal etc.....

2) Object Oriented Languages

Ex: Java, C#, Python etc.....

-> In Procedure Oriented programming language, we will develop functions & procedures

-> If we want to add more functionalities then we need to develop more functions

-> Maintaining & Managing more functions is difficult task

-> In PoP, data is exposed globally

-> In Pop, there is no security

-> If we want to develop a project using OOP lanaguage then we have to use Classes & Objects

-> Any language which follows OOPS Principles is called as OOP Language

-> Object Oriented languages provides security for our data

-> The main advantage of OOPS is code re-usability

OOPS Principles

1) Encapsulation

2) Abstraction

3) Polymorphism

4) Inheritance

Encapsulation

-> Encapsulation is used to combine our variables & methods as single entity / unit

-> Encapsulation provides data hiding

-> We can achieve encapsulation using Classes

```
class  Demo {
        //variables
        // methods
}
```

Abstraction

-> Abstraction means hiding un-necessary data and providing only required data

-> We can achieve Abstraction using interfaces & abstract classes

Ex : we will not bother about how laptop working internally

We will not bother about how car engine starting internally

Polymorphism

-> Exhibiting multiple behaviours based on situation is called as Polymorphism

Ex:-1 : in below scenario + symbol having 2 different behaviours

10 + 20 ===> 30 (Here + is adding)

"hi" + "hello" ==> hihello (here + is concatinating)

Ex:-2:

When i come to class i will behave like a trainer

When i go to ofc i will behave like a employee

When i go to home i will behave like a family member

Inheritance

-> Extending the properties from one class to another class is called as Inheritance

Ex: child will inherit the properties from parent

-> The main aim of inheritance is code re-usability

Note: In java, one child can't inherit properties from two parents at a time

Class

-> Class is a plan or model or template

-> Class is a blue print of object

-> Class is used to declare variables & methods

-> Project means collection of classes

-> Once class is created then we can create any no.of objects for a class

-> 'class' keyword is used to create Classes in java

-> Classes will not exist physically

Class Syntax	Class Example

class <ClassName> {
// variables
// methods
}
	class Student {
	int age = 20;
	void printAge ( ) {
	System.out.println (age);
	}
	}

Object

-> Any real-world entity is called as Object

-> Objects exist physically

-> Objects will be created based on the Classes

-> Without having the class, we can' create object (class is mandatory to create objects)

-> Object creation means allocating memory in JVM

-> 'new' keyword is used to create the objects

Syntax :

ClassName refVariable = new ClassName ( );

Example:

User u1 = new User ( );

User u2 = new User ( ) ;

Note: Here 'User' is a class name

-> Objects will be created by JVM in the runtime

-> Objects will be created in heap area.

-> If object is not using then garbage Collector will remove that object from heap

-> Garbage Collector is responsible for memory clean-up activities in JVM heap area.

-> Garbage Collector will remove un-used objects from heap.

-> Garbage Collector will be managed & controlled by JVM only.

Note: Programmer don't have control on Garbage Collector.

What is Hash Code

-> When we create Object for class, then JVM will assign unique hashcode for every object

-> No two objects will have same hashcode

-> We can get hashcode of the object by calling java.lang.Object class hashCode () method.

u1.hashCode ( );

Note: java.lang.Object class is by default parent class for all java classes.

User.java

```
1 public class User {
2
3     public static void main(String[] args) {
4         User u1 = new User();
5         System.out.println(u1.hashCode());
6
7         User u2 = new User();
8         System.out.println(u2.hashCode());
9
10        User u3 = new User();
11        System.out.println(u3.hashCode());
12
13    }
14 }
```

Variables

-> Variables are used to store the data

int a = 10 ;

User u1 = new User ( );

Student s1 = new Student ( );

-> Variables are divided into 3 types

a) Global Variables / instance variables / non-static variables

b) static variables

c) local variables

Instance variables

-> Variables which are declared inside the class and outside the method are called as instance variables

-> instance variables can be accessed by all the methods available in the class thats why the are called as Global Variables.

-> Initialization is optional for instance variables

-> Instance variables are called as Object variables

-> When we create the object, then only memory will be allocated for instance variables

Note: If we create 2 objects, then 2 times memory will be allocated for instance variables

-> If we don't initialize instance variable, it will be initialized with default value based on datatype when the object is created

-> Every Object will maintain its own copy of the instance variable

User.java

```
1  public class User {
2
3      int age; // instance variable
4
5      public static void main(String[] args) {
6          User raju = new User(); // object creation (1)
7
8          raju.age = 20; // initializing instance variable
9          System.out.println(raju.age);
10
11         User rani = new User(); // object creation (2)
12         rani.age = 25;// initializing instance variable
13         System.out.println(rani.age);
14
15         User ashok = new User(); // object creation (3)
16     }
17 }
```


Static Variables

-> The variables which are declared inside the class and outside the method with 'static' keyword are called as static variables

-> Static variables are class level variables

-> When class is loaded into JVM then immediately memory will be allocated for static variables

-> Memory will be allocated for static variables only once when the class is loaded into JVM

-> All objects of the class will maintain same copy of the static variables

-> Static variables we will access using class name

Student.java

```java
public class Student {

    String name; // instance variable
    String email; // instance variable
    static String institute; // static variable

    public static void main(String[] args) {
        Student.institute = "ashokit"; // initializing static variable

        Student ankit = new Student(); // creating object
        ankit.name = "Ankit";// initializing instance variable

        Student goutham = new Student();
        goutham.name = "Goutham";// initializing instance variable
    }
}
```

When to declare variable as static or non-static?

-> If we want to store different value based on object then use instance variable

-> If we want to store same value for all objects then use static variable

Local Variables

-> The variables which are declared inside the method or constructor or block are called as Local Variables

-> If we declare a variable with in the method, then that variable can be used / accessed only with in that method

-> Before using local variables, we have to initialize them

-> If we create a variable with in the method, memory will be allocated for that variable when that method is executing. After that method execution completed, local variable will be deleted from memory

Demo.java

```java
public class Demo {

    public static void main(String[] args) {
        int a = 10; // local variable
        System.out.println(a); // valid

        int b; // local variable
        System.out.println(b); // invalid

    }
}
```

Methods

-> Methods are used to perform some operation / action

-> In a class we can write any no. of methods including main method

-> Every method contains 2 parts

1) Method Declaration

2) Method Body

What is Method Declaration?

Method declaration means we are going to decide what is the name of the method, what are the parameters it will take and what kind of value is return by the method.

Syntax ::	returntype methodname(list of parameters);

What is return type?

-> return type is data type that indicates what type of value is return by the particular method.

-> return type can be any primitive type or array type or reference type

-> if method does not return any value, then return type must be specified using a java keyword called " void ".

-> specifying return type is mandatory

What is method name?

-> To identify and access the method we should give a suitable name for a method which is called as method name.

-> a method name can be any valid java identifier.

-> specifying method name is mandatory

What are method parameters?

-> parameters are the variables that will receive the values that are passed into the. particular method on which data method will perform the operations.

-> we can write 0 or more number of parameters of any primitive type or array type or reference type

-> specifying parameters is optional.

method body/method definition/method implementation

-> method definition means we are going to write the group of statements that are executed by the method.

-> a method definition can be written in between a pair of curry braces syntax:

```
returntype methodName (list of parameters){
        //statements;
        return value;
}
```

-> here we can write 0 or more number of statements in between the pair of curly braces.

-> when we write 0 statements then it is called as null implementation

-> if the return type is specified as other than void then we must return a value from our method using java keyword called " return ".

-> The datatype of the value we return must be match with the datatype that we specify as return type.

-> But if return type specified as void, then we must not write any return value statement.

-> In java we can create any number of methods which are in any of the following 4 combinations of methods

1. method without return type, without parameters
2. method without return type, with parameters
3. method with return type, without parameters
4. method with return type, with parameters

Scenario-1: Method without parameters & without return type

Demo.java

```java
public class Demo {

    // method declaration and definition
    void add() {
        int a = 10;
        int b = 20;
        int c = a + b;
        System.out.println("Addition = " + c);
    }

    public static void main(String args[]) {
        Demo m = new Demo(); // object creation
        m.add(); // method invocation
    }
}
```

Scenario-2: Method with parameters & without return type

Demo.java

```java
public class Demo {

    // method declaration and definition
    void add(int a, int b) {
        int c = a + b;
        System.out.println("Addition = " + c);
    }

    public static void main(String args[]) {
        Demo m = new Demo(); // object creation
        m.add(20, 30); // method invocation
    }
}
```

Scenario-3: Method with return type & without parameters

Demo.java

```java
public class Demo {

    // method declaration and definition
    int add() {
        int a = 10;
        int b = 20;
        int c = a + b;

        return c;
    }

    public static void main(String args[]) {
        Demo m = new Demo(); // object creation
        int sum = m.add(); // method invocation
        System.out.println("Sum :: " + sum);
    }
}
```

Scenario-4 Method with return type & with parameters

Demo.java

```java
public class Demo {

    // method declaration and definition
    int add(int a, int b) {
        int c = a + b;
        return c;
    }

    public static void main(String args[]) {
        Demo m = new Demo(); // object creation
        int sum = m.add(10, 20); // method invocation
        System.out.println("Sum :: " + sum);
    }
}
```


Method with String return type & String parameters

Demo.java

```java
public class Demo {

    // method declaration and definition
    String print(String fname, String lname) {
        String name = fname + lname;
        return name;
    }

    public static void main(String args[]) {
        Demo d = new Demo(); // object creation
        String s = d.print("ashok", "it"); // method invocation
        System.out.println("Name :: " + s);
    }
}
```

Method with array as parameter & without return type

Demo.java

```java
import java.util.Arrays;

public class Demo {

    // method declaration and definition
    void print(int[] arr) {
        System.out.println(Arrays.toString(arr));
    }

    public static void main(String args[]) {
        Demo d = new Demo(); // object creation
        int[] arr = { 10, 20, 30 }; // array creation
        d.print(arr); // method invocation with parameter
    }
}
```

Method with int as parameter & boolean return type

Demo.java

```java
public class Demo {

    // method declaration and definition
    boolean check(int age) {
        if (age >= 18) {
            return true;
        } else {
            return false;
        }
    }

    public static void main(String args[]) {
        Demo d = new Demo(); // object creation
        boolean status = d.check(28); // method invocation with parameter
        System.out.println("Eligible For Vote ? :: " + status);
    }
}
```

Method with Object as parameter & without return type

Demo.java

```java
public class Demo {

    // method declaration and definition
    void print(Student s) {
        System.out.println("Id : " + s.id);
        System.out.println("Name : " + s.name);
    }

    public static void main(String args[]) {
        Demo d = new Demo(); // object creation

        Student s = new Student();
        s.id = 101;
        s.name = "Ashok";

        d.print(s); // method invocation
    }
}

class Student {
    int id;
    String name;
}
```


Method with Object as return type & without parameter

Demo.java

```java
public class Demo {

    // method declaration and definition
    Student getStudent() {
        Student s1 = new Student();
        s1.id = 102;
        s1.name = "Ashok";

        return s1;
    }

    public static void main(String args[]) {
        Demo d = new Demo(); // object creation

        Student s = d.getStudent(); // method invocation
        System.out.println("Id :: " + s.id);
        System.out.print("Name :: " + s.name);
    }
}

class Student {
    int id;
    String name;
}
```

Types of Methods

-> Methods are divided into 2 types

1) instance / non – static methods ---> Object level methods

2) static methods ----> Class level methods

In a class, we can write any no. of instance and static methods

Not: To create static method we will use 'static' keyword.

-> When we write methods in java class, by default JVM will not execute them. To execute our methods, we have to call them.

-> instance method will be called by using Object

objReference.methodName(..) ;

-> static method will be called by using Class Name

ClassName.methodName(..);

Instance & Static methods Example

Demo.java

```
1  public class Demo {
2
3      // instance method
4      void sayHello() {
5          System.out.println("Hello My Friend...");
6      }
7
8      // static method
9      static void greet() {
10         System.out.println("Good Morning..");
11     }
12
13     public static void main(String[] args) {
14         Demo d = new Demo(); // obj creation
15         d.sayHello(); // invoking instance method
16
17         Demo.greet(); // invoking static method
18     }
19 }
```


Note: When we have static method in same class then we can invoke/call it directly without using classname.

In above program Line : 17, we are calling static method with classname, as the method is available in same class we can call directly like below.

greet ( ) ;

Explain about main ( ) method

-> Java program execution will be begin from main() method

-> JVM always expect main() method like follows

```
public static void main(String  args [] ){
        //statements;
}
```

1. public:

public is keyword/ modifier / access specifier which indicates the particular entity can be accessed from any location. main() method should be declared as public so that JVM can execute the main() method from any location.

2. static:

static is keyword/ modifier which indicates the particular method can access directly by using class name.

main() method should be declared as static so that JVM can access the main() method directly by using class name.

3. void:

void is keyword/ return type which indicates the particular method does not return any value. JVM is not expecting any value to be return from main() method so that main() method should have the return type as void.

4. main():

main is the name of the method(identifier) which is fixed according to sun micro system's JVM implementation.

5. parameters of main () method (String args[])

parameters of main () method are mainly used to accept the command line arguments to resolve the problem of hard coding.

Explain about System.out.println()

System.out.println() is statement used to display messages on the monitor.

System: System is a predefined class available in java. lang package

Out: out is reference variable of PrintStream class which is holding an object of PrintStream class and declared in

System class as static variable. so that we can access this out variable directly with the help of class name.

Println () : println() is the instance method of PrintStream class. If we want to access this method, we must create an object for PrintStream class but we don't need to create any object for PrintStream class because the object for PrintStream class is already created in System class

Note: System.out is by default connected to monitor. System.in is by default connected to keyboard.

Method Recursion

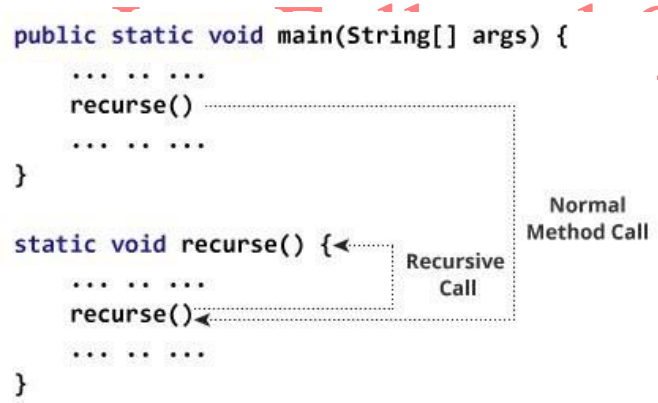
In Java, a method that calls itself is known as a recursive method. And, this process is known as recursion.

Recursion Syntax

Any method that implements Recursion has two basic parts:

a) Method call which can call itself i.e. recursive
b) A precondition that will stop the recursion.

Note that a precondition is necessary for any recursive method as, if we do not break the recursion then it will keep on running infinitely and result in a stack overflow error



Write a java program to find Factorial of a given number

Factorial.java

```
1  class Factorial {
2
3      static int factorial(int n) {
4          if (n != 0) // termination condition
5              return n * factorial(n - 1); // recursive call
6          else
7              return 1;
8      }
9
10     public static void main(String[] args) {
11         int number = 4, result;
12         result = factorial(number);
13         System.out.println(number + " factorial = " + result);
14     }
15 }
16
```

Assignment On Recursion

1) Check If a Number is a Palindrome Using Recursion

2) Reverse a String using Recursion in Java

3) Find Minimum Value in Array Using Recursion

Constructors

-> When we declare instance variables and not initialized with any value, then they automatically initialized with default values.

-> but if we want to initialize instance variables with our own values then we can initialize instance variables in following 2 locations,

1. At the time of declaration
2. Using Constructor

-> A constructor is a special method that is used to initialize an object / instance variables.

-> If we don't declare a constructor in the class then JVM builds a default constructor for that class. This is known as default constructor.

Rules of the constructor

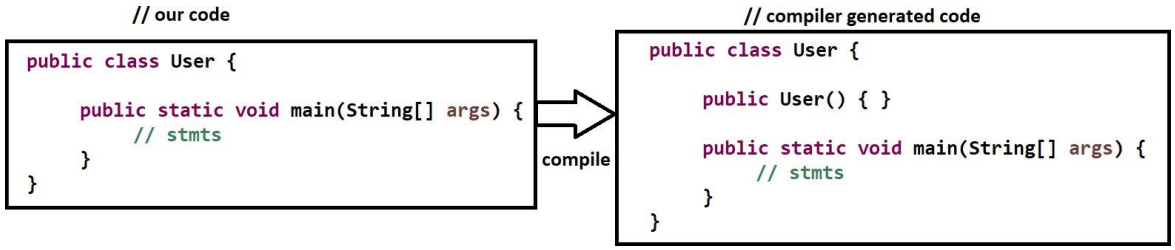
- Constructor name must be same as Class Name
- Constructor cannot take any return type but if we write any return type then the code is valid but it is considered as normal method
- We cannot return any value from the constructor
- Constructors can take one or parameters

Default Constructor

-> In Java, a constructor is said to be default constructor if it does not have any parameter. Default constructor can be either user defined or provided by JVM.

-> If a class does not contain any constructor, then during runtime JVM generates a default constructor which is known as system define default constructor.

-> If a class contain a constructor with no parameter, then it is known as default constructor defined by user. In this case JVM does not create default constructor.



Based on the number of parameters, constructors are classified into following 2 types,

1) 0 Parameterized Constructor
2) Parameterized Constructor

0 parametrized constructor

-> If we declare any constructor without any parameters then it is called as 0 parametrized constructor.

Syntax	Example

class ClassName{
ClassName(){
//statements;
}
}
	public class User {
	User() {
	// statements
	}
	}

-> When we create object for the class then Constructor will be called.

Student.java

```
1  public class Student {
2
3      // 0 - param constructor
4      public Student() {
5          System.out.println("Constructor Called...");
6      }
7
8      public static void main(String[] args) {
9          Student s = new Student(); // creating object
10     }
11 }
12
```

Parametrized constructor

If we declare any constructor with parameters then it is called as parameterized constructor.

Syntax	Example

class ClassName{
ClassName(parameters){
//statements;
}
}
	public class User {
	User(int age) {
	// statements
	}
	}

Note: When the constructor is having parameters, at the time of creating object we have to pass those parameters.

// Program on Parameterized Constructor

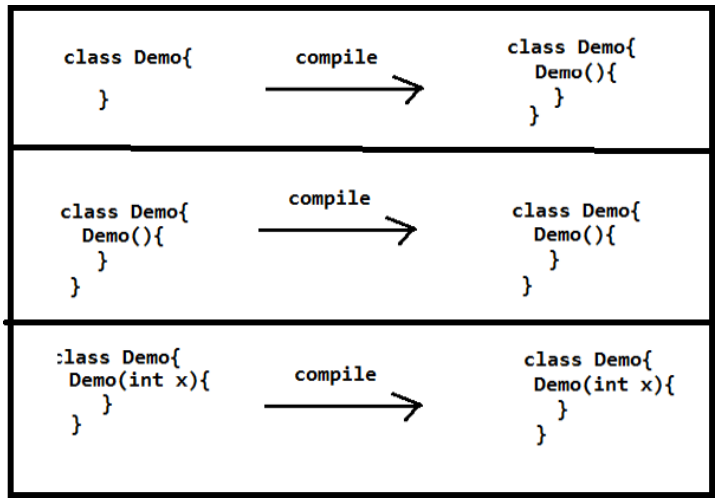
User.java

```
1  public class User {
2
3      // instance variables
4      int age;
5      String name;
6
7      // Parameterized constructor
8      public User(int a, String n) {
9          age = a; // initialization
10         name = n; // initialization
11     }
12
13     public static void main(String[] args) {
14         User u = new User(20, "Ashok"); // creating object
15         System.out.println("Age :" + u.age);
16         System.out.println("Name: " + u.name);
17     }
18 }
19
```

When the default constructor will be added by JVM ?

-> When our class doesn't have any constructor then only JVM will add constructor

-> If our class having any constructor, then JVM will not add any constructor



Constructor Overloading

-> Java Constructor overloading is a technique in which a class can have any number of constructors that differ in parameter list.

In other words, defining two or more constructors with the same name but with different signatures is called constructor overloading in java. It is used to perform different tasks.

-> The compiler differentiates these constructors by taking into account the number of parameters in the list and their data type.


```
Account(int a);

Account (int a, int b);

Account (String a, int b);
```

// Program on Constructor Overloading

User.java

```java
public class User {

    int age;
    String name;

    // Parameterized constructor - 1
    public User(int a, String n) {
        age = a; // initialization
        name = n; // initialization
    }

    // Parameterized constructor - 2
    public User(String n) {
        name = n;
    }

    public static void main(String[] args) {
        User u1 = new User(20, "Ashok"); // creating object
        System.out.println("Age :" + u1.age);
        System.out.println("Name: " + u1.name);

        User u2 = new User("Ashok");
        System.out.println("Age :" + u2.age);
        System.out.println("Name: " + u2.name);
    }
}
```

This keyword

In Java, this is a keyword which is used to refer current object of a class. we can it to refer any member of the class. It means we can access any instance variable and method by using this keyword.

The main purpose of using this keyword is to solve the confusion when we have same variable name for instance and local variables.

We can use this keyword for the following purpose.

- this keyword is used to refer to current object.
- this is always a reference to the object on which method was invoked.
- this can be used to invoke current class constructor.
- this can be passed as an argument to another method.

Let's first understand the most general use of this keyword. As we said, it can be used to differentiate local and instance variables in the class.

Example:

In this example, we have three instance variables and a constructor that have three parameters with same name as instance variables. Now, we will use this to assign values of parameters to instance variables.

Demo.java

```
1  class Demo {
2
3      // instance variables
4      Double width, height, depth;
5
6      // parameterized constructor
7      Demo(double w, double h, double d) {
8          this.width = w;
9          this.height = h;
10         this.depth = d;
11     }
12
13     public static void main(String[] args) {
14
15         Demo d = new Demo(10, 20, 30); //obj creation
16
17         System.out.println("width = " + d.width);
18         System.out.println("height = " + d.height);
19         System.out.println("depth = " + d.depth);
20     }
21 }
```

Knowledge – Check

1) What is Class ?
2) What is Object ?
3) Why we need to Create Object ?
4) What is instance variable and why we need them ?
5) When memory will be allocated for instance variables ?
6) How many times memory will be allocated for instance variables ?
7) What is static variable and why we need them ?
8) When memory will be allocated for static variables ?
9) How many times memory will be allocated for static variables ?
10) What is Local Variable & Why we need Local Variable ?
11) When Memory will be allocated for local variable ?
12) What is Constructor why we need Constructor ?
13) What are the rules to write Constructor ?
14) What is Constructor Overloading & Why we need it ?
15) What is Method and Why we need methods ?
16) When to take method parameter and method return type ?
17) When to use primitive type for method parameters & return types ?
18) When to use Object as method parameter & return type?
19) What is Object Oriented Language ?
20) What are OOPS principles?

Chapter – 6 : OOPS (Part – 2)

- Access Modifiers
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction
- Interfaces
- Abstract Classes
- Java.lang. Object class

Access Modifiers in Java

-> Access modifiers are keywords in Java that are used to set accessibility.

-> An access modifier restricts the access of a class, constructor, data member and method in another class.

Java language has four access modifiers to control access level for classes and its members.

- Default: Default has scope only inside the same package
- Public: Public has scope that is visible everywhere
- Protected: Protected has scope within the package and all sub classes
- Private: Private has scope only within the classes

Java also supports many non-access modifiers, such as static, abstract, synchronized, native, volatile, transient etc.

Inheritance

-> inheritance is the one of most important concepts of OOPS

-> inheritance means taking the properties of on class into another class

-> in inheritance the class which is giving properties is called as parent class or super class or base class

-> in inheritance the class which is taking properties is called as child class or sub class or derived class

-> The main advantage of inheritance is re usability

-> To achieve inheritance we will use 'extends' keyword

```
class Vehicle
{
   ......
}

class Car extends Vehicle
{
   .......   //extends the property of vehicle class
}
```

Now based on above example. In OOPs term we can say that,

- Vehicle is super class of Car.
- Car is sub class of Vehicle.
- Car IS-A Vehicle.

// Inheritance Example – Accessing parent class method in child class

Child.java

```java
class Parent {
    public void p1() {
        System.out.println("Parent method");
    }
}

public class Child extends Parent {

    public void c1() {
        System.out.println("Child method");
    }

    public static void main(String[] args) {
        Child cobj = new Child();
        cobj.c1(); // method of Child class
        cobj.p1(); // method of Parent class
    }
}
```

Note:

1. If we create an object for parent class then we can access only the members of Parent class

2. but if we create an object for child class then we can access the members of Both Parent class and Child class

-> In the code above, we have a class Parent which has a method p1(). We then create a new class Child which inherits the class Parent using the extends keyword and defines its own method c1(). Now by virtue of inheritance the class Child can also access the public method p1() of the class Parent.

// Inheritance Example – Accessing parent class variable in child class

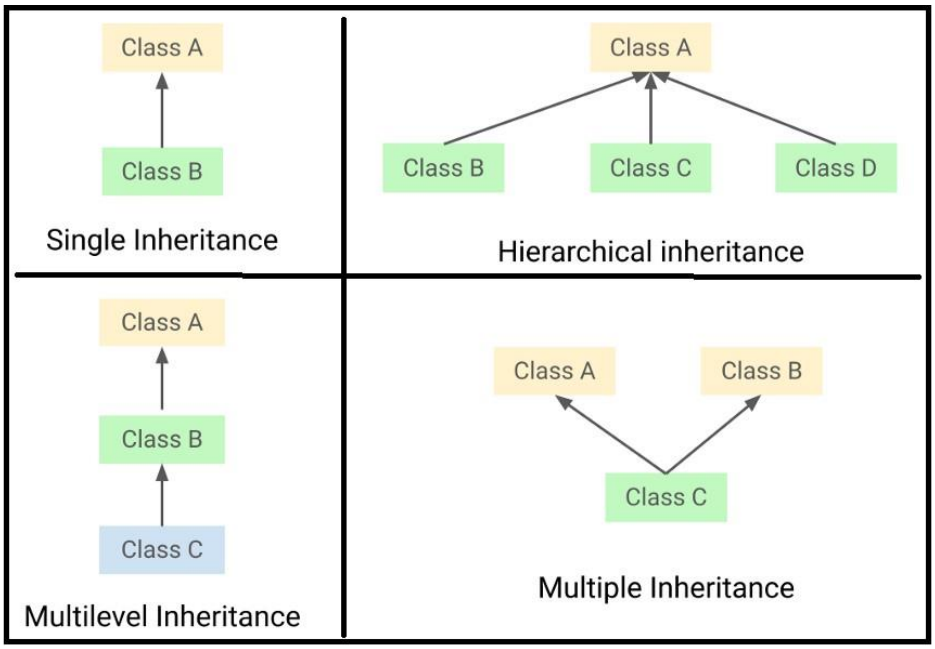
Car.java

```java
class Vehicle {
    // variable defined
    String vehicleType;
}

public class Car extends Vehicle {

    String modelType;

    public void showDetail() {
        vehicleType = "Car"; // accessing Vehicle class member variable
        modelType = "Sports";
        System.out.println(modelType + " " + vehicleType);
    }

    public static void main(String[] args) {
        Car car = new Car();
        car.showDetail();
    }
}
```

Types of Inheritances

Based on the no. of parent classes & child classes we can divide inheritance into below types.

- Single Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Multiple Inheritance



u

Why java does not support multiple inheritance

In multiple inheritance child class is taking properties from multiple parent classes in this case if we have same property in all the parent classes then it is a confusion to child class to decide to take the particular property from which parent class, hence multiple inheritance is not supported in Java.

Inheritance Rules

-> In java we can create a class that extends only one class and we cannot create a class that extends more than one class because java does not support multiple inheritance.

```
class A{

}                       class C extends A, B {
                            // this is invalid
class B{                }

}
```

-> In java cyclic inheritance is not supported.

```
public class Student extends User {

      //stmts
}

public class User extends Student {
      // stmts
}
```
This is invalid

-> In java every predefined class or user defined class are child classes of Object class either directly or indirectly so that all members of the Object class we can directly use in any class.

-> When we declare a class and we are not extending from any class then compiler will create the class by extending from Object class automatically.

but we declare a class by extending from any one class then compiler won't extend from Object class, hence we call Object class as java's super most class

```
class Student {
}
```
⇨
```
class Student extends Object {
}
```

```
class Student  extends User {
}

class User {
}
```
⇨ here Student class extending from User & User class will extend from Object class

ru

super keyword

In Java, super keyword is used to refer to immediate parent class of a child class. In other words super keyword is used by a subclass whenever it need to refer to its immediate super class.

```
class Parent
{
  String name;
}
class Child extends Parent {

  String name;

  void detail()
  {
    super.name = "Parent";
    name = "Child";
  }
}
```

// Example on Super keyword – variable & method invocation

Child.java

```
class Parent {
    String name;

    void details() {
        System.out.println(name.toUpperCase());
    }
}

public class Child extends Parent {
    String name;

    public void details() {
        super.name = "Parent"; // refers to parent class member
        name = "Child";
        System.out.println(super.name + " and " + name);
        super.details(); // refers to parent class method
    }

    public static void main(String[] args) {
        Child cobj = new Child();
        cobj.details();
    }
}
```

// Example of Child class calling Parent class constructor using super keyword

Child.java

```
class Parent {
    String name;

    public Parent(String n) {
        name = n;
    }
}

public class Child extends Parent {
    String name;

    public Child(String n1, String n2) {
        super(n1); // passing argument to parent class constructor
        this.name = n2;
    }

    public void details() {
        System.out.println(super.name + " and " + name);
    }

    public static void main(String[] args) {
        Child cobj = new Child("Parent", "Child");
        cobj.details();
    }
}
```

Note: When calling the parent class constructor from the child class using super keyword, super keyword should always be the first line in the method/constructor of the child class.

Q. Can we use both this () and super () in a Constructor?

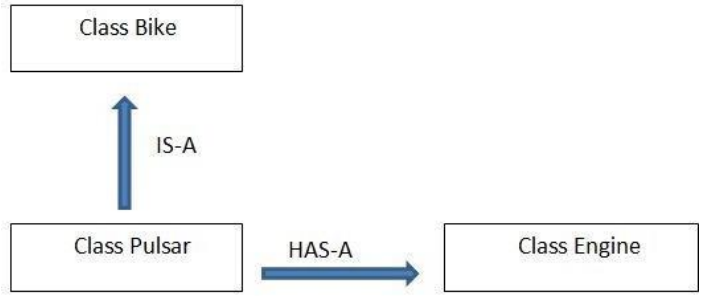
NO, because both super () and this () must be first statement inside a constructor. Hence, we cannot use them together.

Types of Relationships in Java

-> We have below 2 types of relationships

1) Is-A relationship

2) Has-A relationship



IS-A relationship

-> If class is extending from another class then it is called as Is-A relation.

-> Here we can access class1 information inside the class2 directly without creating an object for class1.

Student.java

```
1 class Person {
2 }
3
4 class Employee extends Person {
5 }
6
7 class Student extends Person {
8
9 }
```

Has-A relation

-> If a class contains other class object, then it is called as Has-A relation.

-> Here we can access class1 information inside the class2 only by using object of class1.

Person.java

```
1 class Heart {
2 }
3
4 class Person {
5     Heart h = new Heart();
6 }
7
```

Polymorphism

-> Polymorphism means defining one entity with multiple behaviours

-> In java we have following 2 types of poly morphisms.

1. Compile time polymorphism
2. Runtime Polymorphism

Compile time Polymorphism: If polymorphism is decided at the time of compilation by java compiler, then it is called as compile time polymorphism

Note: By using the concept of **method overloading** we can achieve this compile time poly morphism.

Runtime poly morphism: If polymorphism is decided at the time of execution time by JVM then it is called as run time polymorphism.

Note: By using the concept of **method overriding** we can achieve this runtime poly morphism.

Method overloading

Method overloading means declaring multiple methods with same method name but having different method signature.

In method overloading while writing the method signature we have to follow following 3 Rules

- Method name must be same for all methods
- List of parameters must be different like different type of parameters, different number of parameters, different order of parameters.
- Return type is not considered in method overloading; it means we never decide method overloading with return type

```
void show(int x,int y);
void show(double x,double y);      ✓   // methods are overloaded based on
                                           type of parameters

void show(int x);
void show(int x,int y);            ✓   // methods are overloaded based on
                                           number of parameters

void show(int x,double y);
void show(double x,int y);         ✓   // methods are overloaded based on
                                           order of parameters

void show(int x,int y);
void show(int y,int x);            ✗   // methods are not overloaded because based
                                         on parameter name we can not overload

void show(int x,int y);
int show(int x,int y);             ✗   // methods are not overloaded because based
                                         on return type we can not overload

void show(int x,int y);
int show(double x,double y);       ✓   // methods are overloaded based on type of
                                         parameter but not based on retur type

void show(int x,int y);
void add(double x,double y);       ✓   // methods are not overloaded because 2
                                           method names are different
```

// method overloading example

Calculator.java

```java
class Calculator {

    void sum(int a, int b) {
        System.out.println("sum is" + (a + b));
    }

    void sum(float a, float b) {
        System.out.println("sum is" + (a + b));
    }

    public static void main(String[] args) {
        Calculator cal = new Calculator();
        cal.sum(8, 5); // sum(int a, int b) is method is called.
        cal.sum(4.6f, 3.8f); // sum(float a, float b) is called.
    }
}
```

why should we go for method overloading?

When we want to maintain the flexibility in our application like using one method performing several operations then we can use this method overloading.

Method overriding

-> If we want to achieve the run time polymorphism then we have to use method overriding.

-> Method overriding means declaring 2 methods with same method signature in 2 different classes which are having IS-A relation.

-> While Method overriding and writing the method signature, we must follow following rules.

- Method name must be same
- List of parameters must be same
- Return type must be same
- Private, final and static methods cannot be overridden.
- There must be an IS-A relationship between classes (inheritance).

// Method Overriding Example

Dog.java

```
1  class Animal {
2      public void eat() {
3          System.out.println("Eat all eatables");
4      }
5  }
6
7  class Dog extends Animal {
8      public void eat() // eat() method overridden by Dog class.
9      {
10         System.out.println("Dog like to eat meat");
11     }
12
13     public static void main(String[] args) {
14         Dog d = new Dog();
15         d.eat();
16     }
17 }
```

As you can see here Dog class gives it own implementation of eat() method. For method overriding, the method must have same name and same type signature in both parent and child class.

NOTE: Static methods cannot be overridden because, a static method is bounded with class whereas instance method is bounded with object.

Why should we go for Method overriding?

Whenever child class don't want to use definition written by the Parent class and want to use its own logic then we have to use method overriding it means we have to override the same method with new definition inside the child class.

Final keyword

-> final is a keyword or modifier which can be used at variables, methods & classes.

-> If we declare a variable as final then we can't modify value of the variable. The variable acts like a constant. Final field must be initialized when it is declared.

-> If we declare a method as final then we can't override that method

-> If we declare a class as final then we can't extend from that class. We cannot inherit final class in Java.

// Program on final keyword

FinalDemo.java

```
1  public final class FinalDemo {
2
3      final int a = 10; // valid
4
5      a=20; // invalid
6
7      final int c; // invalid
8
9      final void m1() {
10         System.out.println("Hello");
11     }
12
13     public static void main(String[] args) {
14         FinalDemo d = new FinalDemo();
15         d.m1(); // valid
16     }
17 }
18
19 class Test extends FinalDemo { // invalid
20 }
```

Type casting

-> Type casting is the process of converting value from one type to another type.

-> We always do typecast between 2 different data types which are compatible.

Note: In between 2 same data types typecasting is not required.

-> In java we have following 2 types of type castings

1. type casting w.r.t primitive data types
2. type casting w.r.t reference types

Type casting wrt primitive data types

-> Type casting wrt primitive data types means converting value from one primitive data type to other primitive data type

-> Type casting can be done only between compatible data types

-> Compatible data types are byte, short, int, long, float, double, char

-> we have following 2 types of Type casting w.r.t primitive data types

1) Widening
2) Narrowing

Widening

-> Widening means converting the value from lower data type into higher data type.

syntax:

higher datatype = (higher datatype) lower datatype ;

-> Here data type specified in between the pair of parentheses is called type casting.

-> In widening writing the type casting is optional

-> If we don't write any type casting then compiler will write the type casting automatically hence it is called as implicit type casting.

Main.java

```
class Main {
    public static void main(String[] args) {
        // create int type variable
        int num = 10;
        System.out.println("The integer value: " + num);

        // converting int type to double type
        double data = num;
        System.out.println("The double value: " + data);
    }
}
```

Narrowing

-> narrowing means converting the higher data type value into smaller data type.

syntax:

lowerdatatype = (lowerdatatype) higherdatatype;

-> In narrowing writing the type casting is mandatory

-> In narrowing if we don't write any type casting then compiler wont write any typecasting because there is a chance of loss of some data so that user has to write the typecasting explicitly hence it is called as explicit type casting.

Main.java

```
class Main {
    public static void main(String[] args) {
        // create double type variable
        double num = 10.99;
        System.out.println("The double value: " + num);

        // convert double type to int type
        int data = (int) num;
        System.out.println("The integer value: " + data);
    }
}
```

Type casting w.r.t reference types

-> Type casting w.r.t reference types means converting the object from one reference type to another reference type

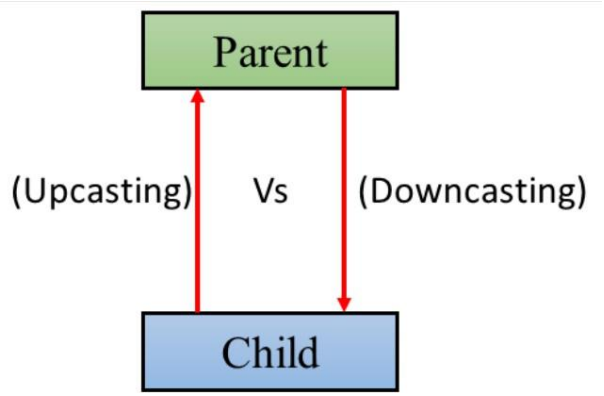
-> But Type casting w.r.t references can be done only between compatible types

-> The two references said to be compatible if and only if its corresponding classes having Is->A relation

-> Type casting w.r.t references is also classified into following 2 types

1) Up Casting

2) Down Casting



Up casting

-> Up casting means storing the child class object into the parent class reference.

syntax:

parentreferencetype = (parentreferencetype) childreferencetype

Note: In up casting writing typecasting is optional

Down casting

-> Down casting means storing the Parent class object into the child class reference.

syntax: childreferencetype = (childreferencetype) parentreferencetype

Note: In down casting writing typecasting is mandatory

// Type Casting Example w.r.t to reference types

TypeCasting.java

```
class Parent {
    void show() {
        System.out.println("This is Parent class Method");
    }
}
class Child extends Parent {
    void show() {
        System.out.println("This is Child class Method");
    }
}

class TypeCasting {
    public static void main(String args[]) {
        Parent p = new Parent();
        p.show();

        p = (Parent) new Child(); // typecasting optional (upcasting)
        p.show();

        Child c = (Child) p; // typecasting is mandatory ( downcasting)
        c.show();
    }
}
```

Java Abstract class and methods

-> A class which is declared using abstract keyword known as abstract class.

-> An abstract class may or may not have abstract methods.

-> We cannot create object of abstract class.

-> It is used to achieve abstraction but it does not provide 100% abstraction because it can have concrete methods.

- An abstract class must be declared with an abstract keyword.
- It can have abstract and non-abstract methods.
- It cannot be instantiated.
- It is used for abstraction.

Syntax :

abstract class class_name { }

Abstract method

Method that are declared without any body within an abstract class are called abstract method.

The method body will be defined by its subclass.

Abstract method can never be final and static.

Any class that extends an abstract class must implement all the abstract methods.

Syntax:

abstract return_type function_name (); //No definition

// Program on abstract class & abstract method

Car.java

```
1  abstract class Vehicle {
2      public abstract void engine();
3  }
4
5  public class Car extends Vehicle {
6
7      public void engine() {
8          System.out.println("Car engine");
9      }
10
11     public static void main(String[] args) {
12         Vehicle v = new Car();
13         v.engine();
14     }
15 }
16
```

When to use Abstract Methods & Abstract Class?

Abstract methods are usually declared where two or more subclasses are expected to do a similar thing in different ways through different implementations. These subclasses extend the same Abstract class and provide different implementations for the abstract methods.

Abstract classes are used to define generic types of behaviours at the top of an object-oriented programming class hierarchy, and use its subclasses to provide implementation details of the abstract class.

Interfaces in Java

-> An interface is a fully abstract class. It includes a group of abstract methods (methods without a body).

-> We use the interface keyword to create an interface in Java.

Language.java

```
1 interface Language {
2
3     public void getType();
4
5     public void getVersion();
6 }
```

Here, Language is an interface.

It contains abstract methods: getType() and getVersion().

Note:

-> All the variables declared in interface are public static final by default whether we specify or not

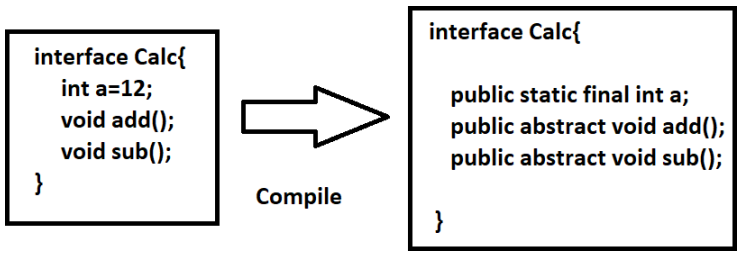
-> All the methods declared in interface are public abstract by default whether we specify or not

public: all the variables and methods declared in interface are public so that they can be accessible from any where.

static: variables declared in interface are by default static so that they can be accessible directly by using the interface name.

final: the variables declared in interface are by default final it means they are constant whose value cannot be changed.

abstract: all the methods declared in interface are abstract because they don't contain any method body



Implementing an Interface

-> Like abstract classes, we cannot create objects of interfaces.

-> To use an interface, other classes must implement it. We use the implements keyword to implement an interface.

Main.java

```java
// create an interface
interface Language {
    void getName(String name);
}

// class implements interface
class ProgrammingLanguage implements Language {

    // implementation of abstract method
    public void getName(String name) {
        System.out.println("Programming Language: " + name);
    }
}

class Main {
    public static void main(String[] args) {
        ProgrammingLanguage language = new ProgrammingLanguage();
        language.getName("Java");
    }
}
```

In the above example, we have created an interface named Language. The interface includes an abstract method getName().

Here, the ProgrammingLanguage class implements the interface and provides the implementation for the method.

Interface Rules

- For an interface we cannot create any object directly but we can create reference variable
- Once an interface is implemented by any class then that class must provide the implementation for all the abstract methods available in the particular interface. This class is also called as implementation class or child class.
- For example, if our class is not providing implementation for at least 1 method then our class must be declared as abstract
- We cannot create an object for abstract class or interface but we can create an object only for implementation class.
- Once an interface is created then any number of classes can implement that interface

-> In Java, a class can implement multiple interfaces. For example,

```
interface A {
  // members of A
}

interface B {
  // members of B
}

class C implements A, B {
  // abstract members of A
  // abstract members of B
}
```


-> Similar to classes, interfaces can extend other interfaces. The extends keyword is used for extending interfaces. For example,

```
interface Line {
  // members of Line interface
}

// extending interface
interface Polygon extends Line {
  // members of Polygon interface
  // members of Line interface
}
```

Here, the Polygon interface extends the Line interface. Now, if any class implements Polygon, it should provide implementations for all the abstract methods of both Line and Polygon.

-> An interface can extend multiple interfaces also

```
interface A {
  ...
}
interface B {
  ...
}

interface C extends A, B {
  ...
}
```

Why do we use an Interface?

- It is used to achieve total abstraction.
- Since java does not support multiple inheritances in the case of class, by using an interface it can achieve multiple inheritances.
- It is also used to achieve loose coupling.
- Interfaces are used to implement abstraction. So the question arises why use interfaces when we have abstract classes?

Default Methods in interfaces

Before Java 8, we could only declare abstract methods in an interface. However, Java 8 introduced the concept of default methods. Default methods are methods that can have a body. The most important use of default methods in interfaces is to provide additional functionality to a given type without breaking down the implementing classes.

Before Java 8, if a new method was introduced in an interface then all the implementing classes used to break. We would need to provide the implementation of that method in all the implementing classes.

However, sometimes methods have only single implementation and there is no need to provide their implementation in each class. In that case, we can declare that method as a default in the interface and provide its implementation in the interface itself.

Bus.java

```java
interface Vehicle {

    void cleanVehicle();

    default void startVehicle() {
        System.out.println("Vehicle is starting");
    }
}

public class Bus implements Vehicle {
    @Override
    public void cleanVehicle() {
        System.out.println("Cleaning the vehicle");
    }

    public static void main(String args[]) {
        Bus bus = new Bus();
        bus.cleanVehicle();
        bus.startVehicle();
    }
}
```

What are static methods in interfaces?

The static methods in interfaces are similar to default methods but the only difference is that you can't override them.

Now, why do we need static methods in interfaces if we already have default methods?

Suppose you want to provide some implementation in your interface and you don't want this implementation to be overridden in the implementing class, then you can declare the method as static.

Bus.java

```java
interface Vehicle {

    static void cleanVehicle() {
        System.out.println("I am cleaning vehicle");
    }
}

public class Bus implements Vehicle {

    public static void main(String args[]) {
        Vehicle.cleanVehicle();
    }
}
```

Marked interface or tagged interface

-> We can also declare an interface without any abstract methods which is called Marked interface or tagged interface

-> the main advantage of Marked interfaces is giving an instruction to JVM to perform a special task.

Eg: Cloneable, Serializable, EventListener, …

Functional interfaces

Functional interfaces are new additions in Java 8. As a rule, a functional interface can contain exactly one abstract method. These functional interfaces are also called Single Abstract Method interfaces (SAM Interfaces).

Apart from one abstract method, a functional interface can also have the following methods that do not count for defining it as a functional interface.

- Default methods
- Static methods
- Public methods inherited from the Object class

// Functional interface example

Example.java

```
1 @FunctionalInterface
2 interface sayable {
3     void say(String msg);
4 }
5
6 class Example implements sayable {
7     public void say(String msg) {
8         System.out.println(msg);
9     }
10
11    public static void main(String[] args) {
12        Example ex = new Example();
13        ex.say("Hello there");
14    }
15 }
```

Note: Functional Interfaces are introduced to invoke Lambda Expressions

Variable Arguments In Java

-> In Java version 1.5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments.

-> A method that takes a variable number of arguments is a var-args method.

Rules for var args

There are some rules for var args that we must follow, otherwise, the code can't compile, They are:

1) There can be only one variable argument in the method

2) Variable arguments (var args) must be the last argument

3) Variable argument should have only three ellipses (...)

// how we will write method before java - 5

BeforeJDK5.java

```
1  package in.ashokit;
2
3  public class BeforeJDK5 {
4
5      public static void main(String[] args) {
6          sum(10, 20);
7          sum(10, 20, 30);
8          sum(10, 20, 30, 40);
9          sum(10, 20, 30, 40, 50);
10     }
11
12     public static void sum(int a, int b) {
13         System.out.println(a + b);
14     }
15
16     public static void sum(int a, int b, int c) {
17         System.out.println(a + b + c);
18     }
19
20     public static void sum(int a, int b, int c, int d) {
21         System.out.println(a + b + c + d);
22     }
23
24     public static void sum(int a, int b, int c, int d, int e) {
25         System.out.println(a + b + c + d + e);
26     }
27 }
28
```

Note: If we observe above program, we are writing sum ( ) method different arguments. If we want to add 5 numbers or 6 numbers or 7 numbers then writing multiple methods will become difficult.

// Writing method with var args from Java 1.5 v

FromJDK5.java

```
1  package in.ashokit;
2
3  public class FromJDK5 {
4
5      public static void main(String[] args) {
6          sum(10, 20);
7          sum(10, 20, 30);
8          sum(10, 20, 30, 40);
9          sum(10, 20, 30, 40, 50);
10     }
11
12     public static void sum(int... x) {
13         int total = 0;
14         for (int i = 0; i < x.length; i++) {
15             total = total + x[i];
16         }
17         System.out.println(total);
18     }
19
20 }
21
```


Java.lang. Object class in Java

-> Object class is present in java.lang package.

-> Every class in Java is directly or indirectly derived from the Object class.

-> If a class does not extend any other class, then it is a direct child class of Object and if extends another class then it is indirectly derived.

-> Object class methods are available to all Java classes. Hence Object class acts as a root of inheritance hierarchy in any Java Program.

-> Below are the methods available in java.lang.Object class

1. String toString()

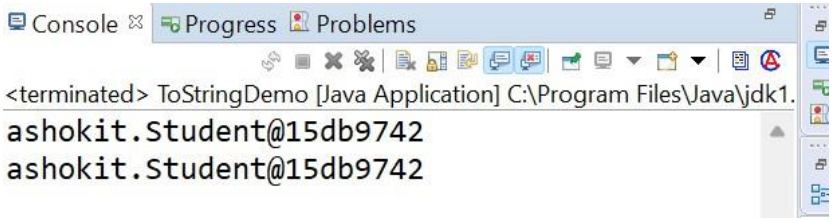
This method used to represent the object in the form of string. when we display any object it will call toString() method automatically whether we specify or not

// Example on Object class toString ( ) method

ToStringDemo.java

```
1  package ashokit;
2
3  public class ToStringDemo {
4
5      public static void main(String[] args) {
6          Student s = new Student(1, "sachin");
7
8          System.out.println(s); // Student@19821f
9          System.out.println(s.toString()); // Student@19821f
10
11         Student s1 = new Student(1, "sachin");
12         System.out.println(s1);// Student@addbf1
13     }
14
15 }
16
17 class Student {
18     int id;
19     String name;
20
21     public Student(int id, String name) {
22         this.id = id;
23         this.name = name;
24     }
25 }
```


// output of Object class toString ( ) method



// Predefined code of toString() method in Object class

```
public String toString() {
  return getClass().getName() + "@" + Integer.toHexString(hashCode());
}
```

when we display the object of the class it will call Object class toString( ) method, if we want to display object data then we have to override the toString() method inside the particular class.

// Example on toString ( ) method - overriding in our class to get object data

ToStringDemo.java

```
1  package ashokit;
2
3  public class ToStringDemo {
4
5      public static void main(String[] args) {
6          Student s = new Student(1, "sachin");
7
8          System.out.println(s);
9          System.out.println(s.toString());
10     }
11
12 }
13
14 class Student {
15     int id;
16     String name;
17
18     public Student(int id, String name) {
19         this.id = id;
20         this.name = name;
21     }
22
23     @Override
24     public String toString() {
25         return "Student [id=" + id + ", name=" + name + "]";
26     }
27 }
```


// Output after overriding toString ( ) method

Console Progress Problems

<terminated> ToStringDemo [Java Application] C:\Program Files\Java\jdk1

```
Student [id=1, name=sachin]
Student [id=1, name=sachin]
```

Note: String, StringBuffer,Wrapper classes are already overriding toString() method

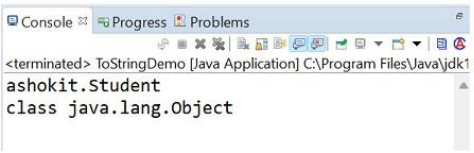
2. Class getClass()

-> This method returns the object in the form of Class using which we can get the information of particular class like class name, belongs what package,.....

-> When ever we want to know what it is underlying class name of the particular object we have to use getClass() method of Object class.

Main.java

```
1  package ashokit;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          Student s = new Student(1, "Ashok");
7
8          System.out.println(s.getClass().getName());
9          System.out.println(s.getClass().getSuperclass());
10     }
11 }
12
13 class Student {
14     int id;
15     String name;
16
17     public Student(int id, String name) {
18         this.id = id;
19         this.name = name;
20     }
21 }
```



3. int hashCode()

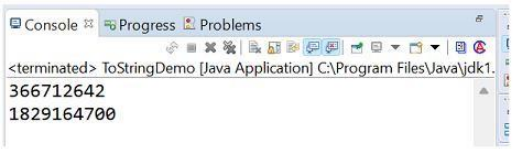
-> This method returns hashcode of the particular object .

-> Hashcode is a unique identification number which holds address of the corresponding object.

Main.java

```
1  package ashokit;
2
3  public class Main {
4
5      public static void main(String[] args) {
6          Student s1 = new Student(1, "Ashok");
7          Student s2 = new Student(2, "Chakravarthy");
8
9          System.out.println(s1.hashCode());
10         System.out.println(s2.hashCode());
11     }
12 }
13
14 class Student {
15     int id;
16     String name;
17
18     public Student(int id, String name) {
19         this.id = id;
20         this.name = name;
21     }
22 }
```

// Output



Note: We can also provide our own hashcode then we have to override the hashcode() method inside corresponding class

Main.java

```java
package ashokit;

public class Main {

    public static void main(String[] args) {
        Student s1 = new Student(1, "Ashok");
        Student s2 = new Student(2, "Chakravarthy");

        System.out.println(s1.hashCode()); // 1
        System.out.println(s2.hashCode()); // 2
    }
}

class Student {
    static int count = 1;
    int rno;
    String name;

    Student(int rno, String name) {
        this.rno = rno;
        this.name = name;
    }

    public int hashCode() {
        return count++;
    }
}
```


4. boolean equals(Object o)

this method compares the 2 references whether they contain same object or not bydefault. but if we want to compare the equality of the content of 2 objects then we have to override equals() method with in the particular class.

Main.java

```java
package ashokit;

public class Main {

    public static void main(String[] args) {

        String str1 = new String("ashok");
        String str2 = new String("ashok");

        System.out.println(str1.equals(str2)); // true

        Student s1 = new Student(1, "Ashok");
        Student s2 = new Student(2, "Chakravarthy");

        System.out.println(s1.equals(s2)); // false (reference checking)

    }
}

class Student {
    int rno;
    String name;

    Student(int rno, String name) {
        this.rno = rno;
        this.name = name;
    }
}
```

Note:

In String class already equals() method is overridden in such a way that it will compare the equality of the content of 2 Strings.

```java
class Student {
    int rno;
    String name;

    Student(int rno, String name) {
        this.rno = rno;
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        Student s = (Student) obj;
        if (this.name.equals(s.name) && this.rno == s.rno) {
            return true;
        } else {
            return false;
        }
    }
}
```

Note: StringBuffer class is not overriding equals() method so that in the above program when we compare 2 StringBuffer objects using equals() method it will check reference of 2 objects but not equality of the content.

Main.java

```
1  package ashokit;
2
3  public class Main {
4
5      public static void main(String[] args) {
6
7          String str1 = new String("ashok");
8          String str2 = new String("ashok");
9
10         System.out.println(str1.equals(str2)); // true
11
12         StringBuffer sb1 = new StringBuffer("ashok");
13         StringBuffer sb2 = new StringBuffer("ashok");
14
15         System.out.println(sb1.equals(sb2)); // false - reference checking
16
17     }
18 }
```

5. Object clone():

This method used to clone or copy the object so that we can take the backup for the object, but in java every object is created in a way that, it can not be copied directly, if we want to perform this special operations we must follow following rules.

1. the cloned object must be type casted to corresponding object

2. we must handle an exception called CloneNotSupportedException

3. corresponding class must implement a marked interface called Cloneable interface

//program without cloning (references are copied)

Student.java

```
1  package ashokit;
2
3  class Student implements Cloneable {
4
5      int rno;
6      String name;
7
8      Student(int rno, String name) {
9          this.rno = rno;
10         this.name = name;
11     }
12
13     public static void main(String args[]) throws CloneNotSupportedException {
14         Student s1 = new Student(1, "sachin");
15         System.out.println(s1.rno + "\t" + s1.name); // 1 sachin
16
17         Student s2 = s1; // shalow cloning or partial cloning
18         System.out.println(s2.rno + "\t" + s2.name); // 1 sachin
19
20         s2.rno = 7;
21         s2.name = "dhoni";
22
23         System.out.println(s1.rno + "\t" + s1.name); // 7 dhoni
24         System.out.println(s2.rno + "\t" + s2.name); // 7 dhoni
25     }
26 }
```

// program with cloning

Student.java

```
1  package ashokit;
2
3  class Student implements Cloneable {
4
5      int rno;
6      String name;
7
8      Student(int rno, String name) {
9          this.rno = rno;
10         this.name = name;
11     }
12
13     public static void main(String args[]) throws CloneNotSupportedException {
14         Student s1 = new Student(1, "sachin");
15         System.out.println(s1.rno + "\t" + s1.name); // 1 sachin
16
17         Student s2 = (Student) s1.clone(); // deep cloning
18         System.out.println(s2.rno + "\t" + s2.name); // 1 sachin
19
20         s1.rno = 7;
21         s1.name = "dhoni";
22
23         System.out.println(s1.rno + "\t" + s1.name); // 7 dhoni
24         System.out.println(s2.rno + "\t" + s2.name); // 1 sachin
25     }
26 }
```

6. finalize() in Java

The finalize() method is called by the Garbage Collector when there are no more references to the object in question. Thus, finalize() is called just before an object is garbage collected.

Demo.java

```
1  package ashokit;
2
3  public class Demo {
4
5      public static void main(String[] args) {
6          Demo d = new Demo();
7          d = null;
8          System.gc();
9      }
10
11     protected void finalize() {
12         System.out.println("finalize( ) method called");
13     }
14
15 }
16
```

7. wait () in Java

The wait () method causes a current thread to wait until another thread invokes the notify () or the notifyAll () method for the object in question.

Syntax:

public final void wait () throws InterruptedException

public final void wait (long timeout) throws InterruptedException

public final void wait (long timeout, int nanos) throws InterruptedException

8. notify()

The notify() method releases the wait on the waiting thread on the invoking object's monitor so that it can continue execution.

Syntax : public final void notify()

9. notifyAll()

The notifyAll() method releases the wait on all the waiting threads on the invoking object's monitor so that they can continue execution.

Syntax : public final void notifyAll()

Note: The notify() can be used to wake up only one thread that is waiting for a particular object whereas notifyAll() can be used to wake up all the threads of a specific object.

Knowledge – Check

1) What are Access Modifiers in Java?
2) When to use public, private, protected?
3) What is Inheritance?
4) What is Single Level and Multi-Level Inheritance?
5) Why Java Doesn't support Multiple Inheritance?
6) What is Encapsulation?
7) Why to declare variables as private?
8) What is the need of setter and getter methods?
9) What is Polymorphism?
10) What is Method Overloading and why we need it?
11) What is Method Overriding and why we need it?
12) What is Abstract Method?
13) What is Interface and why we need interfaces?
14) What is Abstract Class and Why we need them?
15) Difference between Interface and Abstract Classes
16) What is Marker Interface and why we need them?
17) What is difference between this keyword and super keyword?
18) What is final keyword and when to use it?
19) What is the use of var args?
20) What is Object class?
21) What is Cloning?
22) Can you explain all methods of Object class?
23) When to override equals ( ) method ?
24) What is the difference between String class equals ( ) method and Object class equals ( ) method ?
25) What is hashcode ?
26) How to create Object for a class without using new keyword?

Chapter – 7

- **Packages**
- **Wrapper Classes**
- **Exception Handling**

Packages in Java

-> In small projects, all the java files have unique names. So, it is not difficult to put them in a single folder.

-> But, in the case of huge projects where the number of java files is large, it is very difficult to put files in a single folder because the manner of storing files would be disorganized.

-> Moreover, if different java files in various modules of the project have the same name, it is not possible to store two java files with the same name in the same folder because it may occur naming conflict.

This problem of naming conflict can be overcome by using the concept of packages.

A package is nothing but group of related classes, interfaces, and sub-packages according to their functionality.

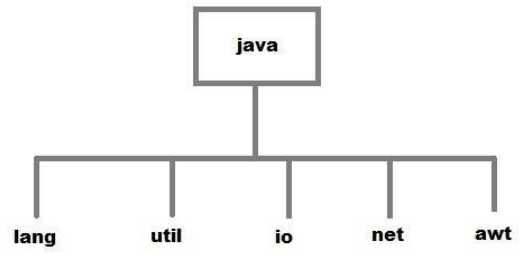
Types of Packages in Java

There are two different types of packages in Java. They are:

1. Predefined Packages in Java (Built-in Packages)

2. User-defined Package

-> Built-in packages are existing java packages that come along with the JDK. For example, java.lang, java.util, java.io, etc.



The package which is defined by the user is called a User-defined package. It is used to create user-defined classes and interfaces.

Creating package in Java

Java supports a keyword called “package” which is used to create user-defined packages in java programming.

Demo.java

```
1 package ashokit;
2
3 public class Demo {
4
5     public static void main(String[] args) {
6         System.out.println("Welcome to Ashok IT...!!");
7     }
8
9 }
10
```

How to compile Java programs inside packages?

This is just like compiling a normal java program. If you are not using any IDE, you need to follow the steps given below to successfully compile your packages:

- javac -d . Demo.java

How to run Java package program?

To run the compiled class that we compiled using above command, we need to specify package name too. Use the below command to run the class file.

- java ashokit.FirstProgram

How to import Java Package

To import java package into a class, we need to use java import keyword which is used to access package and its classes into the java program.

Use import to access built-in and user-defined packages into your java source file so that your class can refer to a class that is in another package by directly using its name.

There are 3 different ways to refer to any class that is present in a different package:

- without import the package
- import package with specified class
- import package with all classes

// Example on Packages importing

Demo.java

```java
package ashokit; // user-defined package

import java.io.*; // importing all classes
import java.util.Arrays; // importing particular class

public class Demo {

    public static void main(String[] args) throws Exception {
        System.out.println("Welcome to Ashok IT...!!");

        int[] arr = { 20, 20, 30 };
        // Arrays class available in java.util package
        System.out.println(Arrays.toString(arr));

        // Date class we are accessing without import keyword
        java.util.Date d = new java.util.Date();
        System.out.println(d);

        // FileReader class available in java.io package
        FileReader fr = new FileReader("abc.txt");
    }
}
```


Static imports

static imports are special kind of import statements given in java 1.5 version which are used to import the static members of any class into the program so that we can access those static members directly without any class name or object.

StaticImport.java

```java
import static java.lang.System.*;

class StaticImportExample {

    public static void main(String args[]) {
        out.println("Hello");// Now no need of System.out
        out.println("Java");
    }
}
```

Java Wrapper Classes

-> In Java, Wrapper Class is used for converting primitive data type into object and object into a primitive data type.

-> For each primitive data type, a pre-defined class is present which is known as Wrapper class.

-> From J2SE 5.0 version the feature of autoboxing and unboxing is used for converting primitive data type into object and object into a primitive data type automatically.

-> All wrapper classes available in java.lang package

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

8 Wrapper Classes in Java

u

Constructors of Wrapper classes

-> to create an object for any Wrapper class we have to use constructors of Wrapper classes.

-> In all most all wrapper classes we have following 2 constructors

-> first one will take primitive value directly

-> and second one will take primitive value in the form of String

Byte b = new Byte(byte);

Byte b = new Byte(String);

Short s = new Short(short);

Short s = new Short(String);

Integer i = new Integer(int);

Integer i = new Integer(String);

Long l = new Long(long);

Long l = new Long(String);

Float f = new Float(float);

Float f = new Float(double);

Float f = new Float(String);

Double d = new Double(double);

Double d = new Double(String);

Character c = new Character(char);

Character c = new Character(String); // Not available

Boolean b = new Boolean(boolean);

Boolean b = new Boolean(String);

// example on Wrapper classes

WrapperDemo.java

```
1  package ashokit;
2
3  public class WrapperDemo {
4
5      public static void main(String args[]) {
6          int a = 10;
7          Integer p = new Integer(a); // boxing
8          System.out.println(p);
9
10         String b = "120";
11         Integer q = new Integer(b); // boxing
12         System.out.println(q);
13     }
14
15 }
16
```

-> While creating an object for Boolean Wrapper class if constructor takes primitive boolean value as true it stores true otherwise if it takes primitive boolean value as false it stores false.

-> While creating an object for Boolean Wrapper class if constructor takes primitive boolean value in the form of string then if string value is true in any case (true, True,TRUE,...) then it stores true otherwise if string value is other than true it stores false.

Eg:

```
Boolean b1 = new Boolean(); //CTE

Boolean b2 = new Boolean(true); //true

Boolean b3 = new Boolean(false); //false

Boolean b4 = new Boolean(True); //CTE

Boolean b5 = new Boolean("TRUE"); //true

Boolean b6 = new Boolean("false"); //false

Boolean b7 = new Boolean("suresh"); //false

Boolean b8 = new Boolean(null); //false

Boolean b9 = new Boolean(1); //CTE
```

Methods of Wrapper classes

1. valueOf() : : This methd is also used to convert the primitive value into object(boxing)

syntax1: public static WrapperClass valueOf(primitive)

-> This method is available in all the 8 wrapper classes.

syntax2: public static WrapperClass valueOf(String)

-> This method is available in all the wrapper classes except Character class.

2. primitivetype xxxValue() :: This method return the primitive value available in the respective object, it means we are converting object into primitive value this procedure is called as unboxing.

3. static primitiveType parseXXX(String) : This method converts the primitive value available in the form of String into required primitive type.

-> This procedure is called as parsing.(kind of unboxing)

-> This method is available in all wrapper classes except Character class.

// Example on valueOf ( ) method

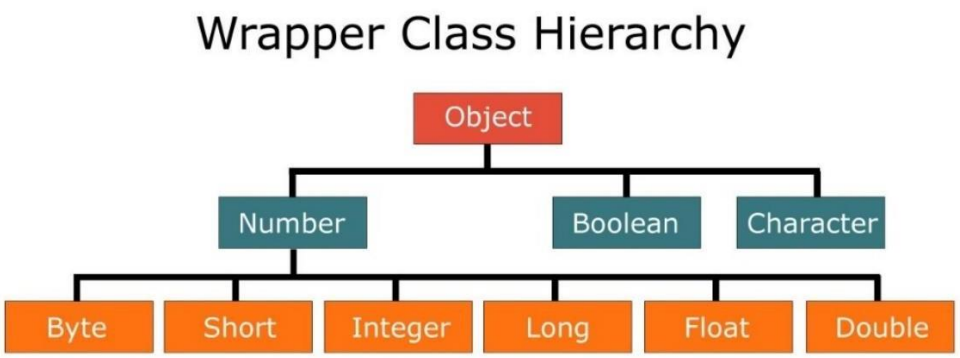
WrapperDemo.java

```
1  package ashokit;
2
3  public class WrapperDemo {
4
5      public static void main(String args[]) {
6
7          int a = 10;
8          Integer p = new Integer(a); // boxing using constructor
9          Integer q = Integer.valueOf(a);// boxing using valueOf()
10
11         System.out.println(p);
12         System.out.println(q);
13
14         String b = "120";
15         Integer x = new Integer(b); // boxing using constructor
16         Integer y = Integer.valueOf(b); // boxing using valueOf()
17
18         System.out.println(x);
19         System.out.println(y);
20     }
21 }
22
```

// example on static valueOf ( ) method

WrapperDemo.java

```
1  package ashokit;
2
3  public class WrapperDemo {
4
5      public static void main(String args[]) {
6
7          int a = 1000;
8
9          Integer i1 = Integer.valueOf(a);// boxing
10         int v1 = i1.intValue();// unboxing
11         byte v2 = i1.byteValue(); // unboxing
12
13         System.out.println(v1);
14         System.out.println(v2);
15
16         // parsing
17         String b = "20";
18         int v3 = Integer.parseInt(b);
19         System.out.println(v3);
20     }
21 }
22
```



Typecasting: typecasting is a process of converting one primitive type to another primitive type or one reference to another reference type

Parsing: parsing is a process of converting String into corresponding primitive type

Boxing: converting a primitive value into object is called boxing. from java 1.5 onwards this procedure is done automatically by compiler hence it is called auto boxing.

Unboxing: converting an object value into primitive type is called un boxing. from java 1.5 onwards this procedure is automatically done by compiler hence it is called auto un boxing

WrapperDemo.java

```
1  package ashokit;
2
3  public class WrapperDemo {
4
5      public static void main(String args[]) {
6          Integer a, b;
7          a = new Integer(10); // mannual boxing
8          a = 10; // auto boxing
9          b = 20; // auto boxing
10
11         int c = a.intValue() + b.intValue(); // manual unboxing
12         int d = a + b; // auto unboxing
13     }
14 }
15
```

Exception Handling

Whenever we develop any application there is a chance of getting some errors in the application. When exception occurs in the program then our program will be terminated abnormally.

Exception handling is a mechanism to catch and throw Java exceptions so that the execution of the program will not get disturbed.

What is an Exception?

Exception is an unwanted or unexpected event, which occurs during the execution of a program, i.e. at run time, that disrupts the normal flow of the program's instructions.

Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception, such as the name and description of the exception and the state of the program when the exception occurred.

An exception can occur for many reasons. Some of them are:

- Invalid user input
- Device failure
- Loss of network connection
- Physical limitations (out of disk memory)
- Code errors
- Opening an unavailable file
- resource exhaustion

What is Error?

Errors represent irrecoverable conditions such as Java virtual machine (JVM) running out of memory, memory leaks, stack overflow errors, library incompatibility, infinite recursion, etc. Errors are usually beyond the control of the programmer, and we should not try to handle errors.

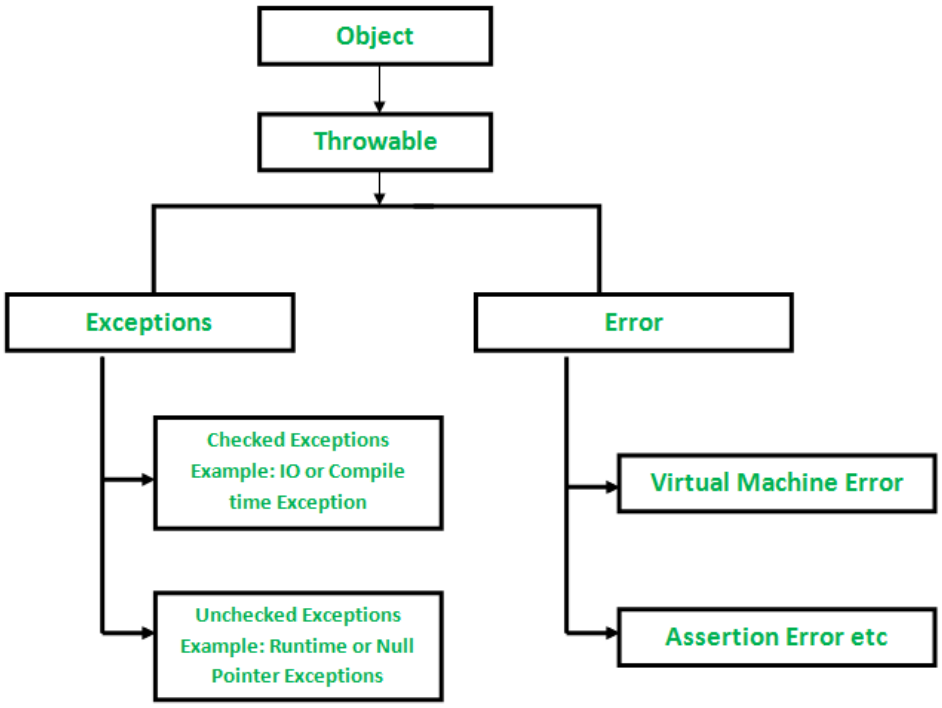
What is the difference between Exception and Error?

Error: An Error indicates a serious problem that a reasonable application should not try to catch.

Exception: Exception indicates conditions that a reasonable application might try to catch.

Exception Hierarchy

All exception types are subclasses of class Throwable, which is at the top of exception class hierarchy.



Checked Exceptions

These are the exceptions that are checked at compile time. If some code within a method throws a checked exception, then the method must either handle the exception or it must specify the exception using the throws keyword.

Note: Checked exceptions are checked by the Java compiler so they are called compile time exceptions.

Note that all checked exceptions are subclasses of Exception class. For example,

- ClassNotFoundException
- IOException
- SQLException etc..

// Example on Checked Exception

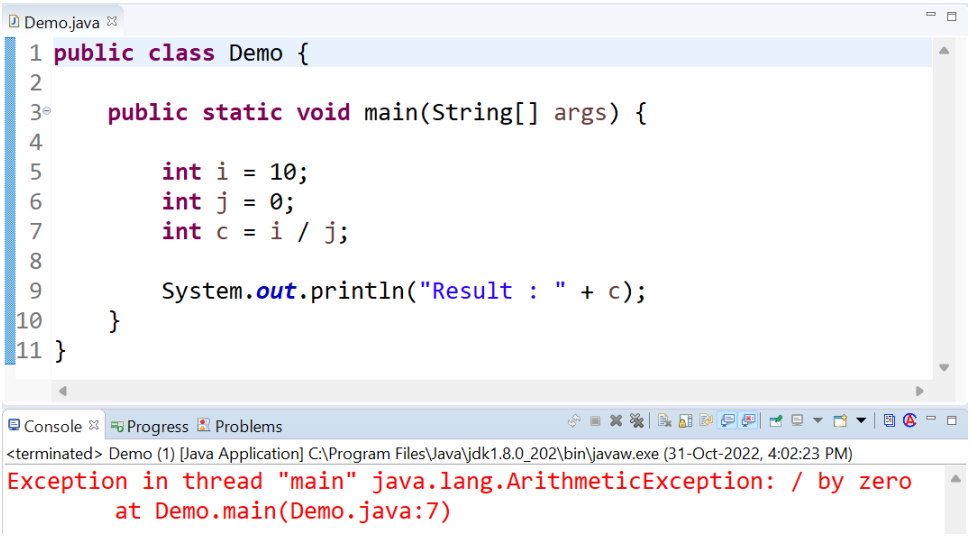
Demo.java

```
1 public class Demo {
2
3     public static void main(String[] args) {
4         FileReader file = new FileReader("somefile.txt");
5     }
6
7 }
```

Un Checked Exceptions

Unchecked exceptions are not checked by the compiler. When the buggy code is executed then we will get exception at runtime, these are called runtime exceptions.

//example on Unchecked exception



Exception Handling Keywords

In java, we can handle exceptions using below 5 keywords

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

try: The try block is used to enclose the suspected code. Suspected code is a code that may raise an exception during program execution.

For example, if a code raise arithmetic exception due to divide by zero then we can wrap that code into the try block.

Syntax :

```
try {
  //group of statements which may generate the exception
}
```

Example:

```
try {
 int a = 10;
 int b = 0
 int c = a/b; // exception
}
```

catch : The catch block also known as handler is used to handle the exception. It handles the exception thrown by the code enclosed into the try block. Try block must provide a catch handler or a finally block.

The catch block must be used after the try block only. We can also use multiple catch block with a single try block.

Syntax :

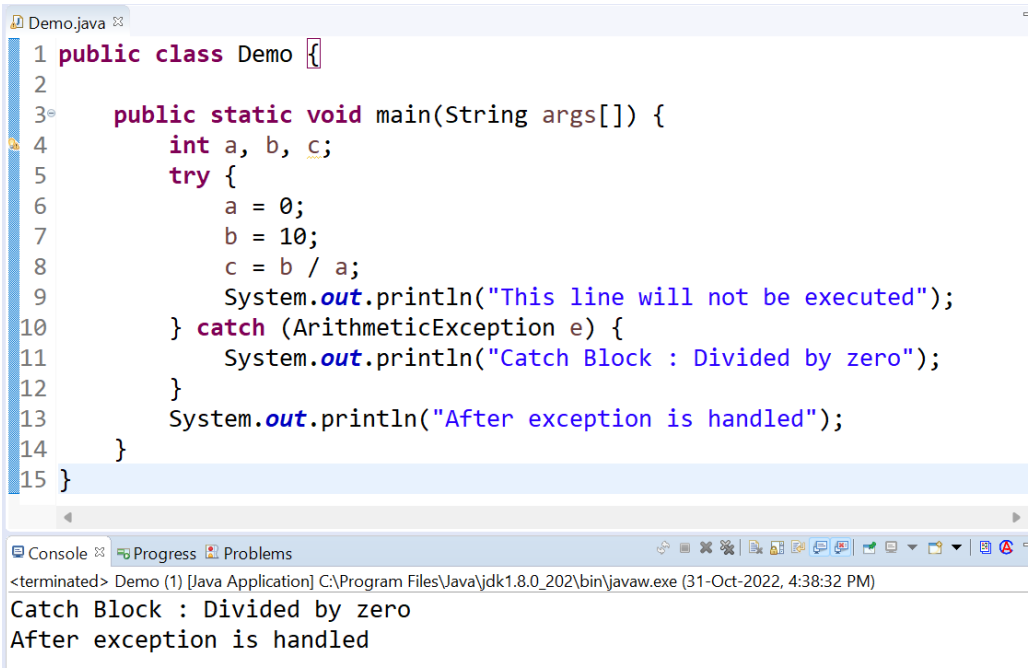
```
catch (AnyException ae) {
   // group of exception handling statements
}
```

Example:

```
try {
 int a = 10;
 int b = 0 ;
 int c = a / b;  // exception
} catch(ArithmeticException e){
 System.out.println(e);
}
```

Note: Exception handling is done by transferring the execution of a program to an appropriate exception handler (catch block) when exception occurs.

// write a java program on try-catch blocks



```
Demo.java
1  public class Demo {
2
3      public static void main(String args[]) {
4          int a, b, c;
5          try {
6              a = 0;
7              b = 10;
8              c = b / a;
9              System.out.println("This line will not be executed");
10         } catch (ArithmeticException e) {
11             System.out.println("Catch Block : Divided by zero");
12         }
13         System.out.println("After exception is handled");
14     }
15 }
```

```
Console  Progress  Problems
<terminated> Demo (1) [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (31-Oct-2022, 4:38:32 PM)
Catch Block : Divided by zero
After exception is handled
```

Exception handling is done by transferring the execution of a program to an appropriate exception handler (catch block) when exception occurs.

An exception will throw by this program as we are trying to divide a number by zero inside try block. The program control is transferred outside try block. Thus, the line "This line will not be executed" is never parsed by the compiler. The exception thrown is handled in catch block. Once the exception is handled, the program control is continued with the next line in the program i.e., after catch block. Thus, the line "After exception is handled" is printed.

Java try with Resource Statement

-> try with resource is a new feature of Java that was introduced in Java 7 and further improved in Java 9. This feature add another way to exception handling with resources management. It is also referred as automatic resource management. It close resources automatically by using AutoCloseable interface..

-> Resource can be any like: file, connection etc and we don't need to explicitly close these, JVM will do this automatically.

-> Suppose, we run a JDBC program to connect to the database then we have to create a connection and close it at the end of task as well. But in case of try-with-resource we don’t need to close the connection, JVM will do this automatically by using AutoCloseable interface.

```
try ( resource-specification )
{
   //use the resource
}
catch()
{
   // handler code
}
```

-> This try statement contains a parenthesis in which one or more resources is declared. Any object that implements java.lang.AutoCloseable or java.io.Closeable, can be passed as a parameter to try statement. A resource is an object that is used in program and must be closed after the program is finished. The try-with-resources statement ensures that each resource is closed at the end of the statement of the try block. We do not have to explicitly close the resources.

Points to Remember

-> A resource is an object in a program that must be closed after the program has finished.

-> Any object that implements java.lang.AutoCloseable or java.io.Closeable can be passed as a parameter to try statement.

-> All the resources declared in the try-with-resources statement will be closed automatically when the try block exits. There is no need to close it explicitly.

Demo.java

```
1  import java.io.*;
2
3  class Demo {
4      public static void main(String[] args) {
5          try (BufferedReader br = new BufferedReader(new FileReader("d:\\myfile.txt"))) {
6              String str;
7              while ((str = br.readLine()) != null) {
8                  System.out.println(str);
9              }
10         } catch (IOException ie) {
11             System.out.println("I/O Exception " + ie);
12         }
13     }
14 }
15
```

-> We can write more than one resources in the try statement.

-> In a try-with-resources statement, any catch or finally block is run after the resources declared have been closed.

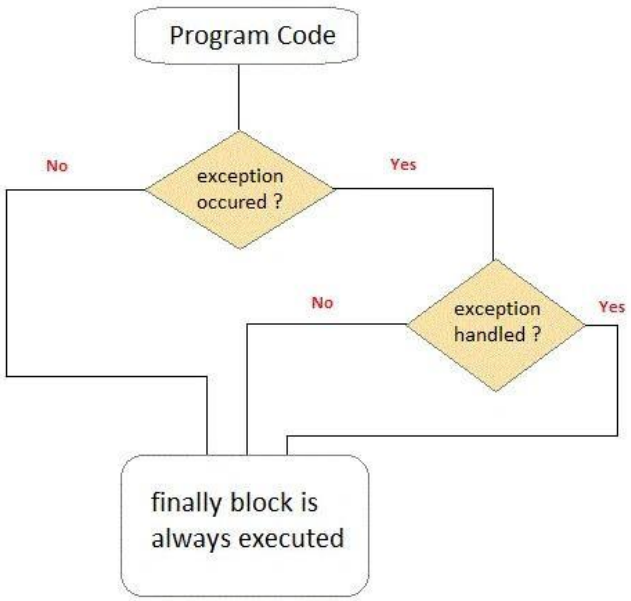
finally block :

-> A finally keyword is used to create a block of code that follows a try block.

-> A finally block of code is always executed whether an exception has occurred or not.

-> Using a finally block, it lets you run any clean up type statements that you want to execute, no matter what happens in the protected code.

-> A finally block appears at the end of catch block.



uru

Demo.java

```java
public class Demo {

    public static void main(String[] args) {
        int a[] = new int[2];
        System.out.println("out of try");
        try {
            System.out.println("Access invalid element" + a[3]);
            /* the above statement will throw ArrayIndexOutOfBoundException */
        } finally {
            System.out.println("finally is always executed.");
        }
    }
}
```

Rules for writing try-catch-finally

1. try must followed by either catch block or finally block or both
2. catch must be followed by either catch block or finally block
3. catch must be preceded by either catch block or try block
4. finally must preceded by either catch block or try block
5. we can write only one finally statement and one try block and many catch blocks in try-catch-finally chain.
6. nesting try-catch-finally is also possible
7. when we write multiple catch blocks if Exceptions are not having any Is-A relation then we can write catch block in any order otherwise we must write catch blocks in a order like first child class and followed by parent class.
8. we can not write 2 catch blocks which are going to catch same exceptions.

How JVM execute try-catch-finally

-> First it will execute the statements written before try-catch blocks next JVM Will enter into try block and if no exception occurred then it will execute all the statements of try block and it will skip all the catch block and control will be given after try-catch blocks

-> JVM Will enter into try block and if exception occurred then it will skip remaining statements of try block and control will be given any one of the matching catch blocks and executes the statements available in the catch block and next control will be given after try-catch blocks

Note:

1. when Exception occurred then control will be given any of the matching catch blocks and the remaining catch blocks will be skipped. and control will be given after try-catch blocks
2. If no catch block is matching then JVM will call or invoke " DefaultExceptionHandler" and which always cause for abnormal termination.
3. If Exception Occurred or not in both the cases finally block will be executed. `

Working with Multiple Catch Blocks

-> For single try block we can write multiple catch blocks

=> When we are writing multiple catch blocks, the hierarchy should be child to parent.

// invalid code because second catch block is un-reachable

```
try{
        int a = 10 / 0;
} cath (Exception e) {
        s.o.p("catch-1");
} catch (ArithematicException e) {
        s.o.p("catch-2");
}
```

// valid code

```
try{
        int a = 10 / 0;
} cath (ArithematicException e) {
        s.o.p("catch-1");
} catch (Exception e) {
        s.o.p("catch-2");
}
```


=> We can handle multiple Exceptions in Single Catch block also like below

```
 9 public class Test {
10
11     public static void main(String[] args) {
12
13         try (FileReader fr = new FileReader(new File("abc.txt"))) {
14             DriverManager.getConnection("url", "uname", "pwd");
15         } catch (IOException | SQLException e) {
16             System.out.println("Catch-1");
17         }
18     }
19 }
20
```

Throw :

-> The throw keyword is used to throw an exception explicitly.

-> By default all predefined exceptions are created and thrown implicitly and identified by JVM

-> If we want to throw the exceptions explicitly then we have to use throw keyword.

Demo.java

```
1  import java.io.*;
2
3  public class Demo {
4      public static void main(String args[]) throws IOException {
5          System.out.println("first line in main()");
6          int a, b, c = 0;
7          a = Integer.parseInt(args[0]);
8          b = Integer.parseInt(args[1]);
9          try {
10             if (b == 0) {
11                 ArithmeticException ae1 = new ArithmeticException("Division by by 0 is not Possible");
12                 throw ae1;
13             }
14             c = a / b;
15             System.out.println("Division=" + c);
16         } catch (ArithmeticException ae) {
17             System.out.println(ae.getMessage());
18         }
19         System.out.println("last line in main()");
20     }
21 }
22
```


Java throws Keyword:

The throws keyword is used to declare the list of exception that a method may throw during execution of program. Any method that is capable of causing exceptions must list all the exceptions possible during its execution, so that anyone calling that method gets a prior knowledge about which exceptions are to be handled. A method can do so by using the throws keyword.

```
type method_name(parameter_list) throws exception_list
{
  // definition of method
}
```

Demo.java

```java
public class Demo {
    static void check() throws ArithmeticException {
        System.out.println("Inside check function");
        throw new ArithmeticException("demo");
    }

    public static void main(String args[]) {
        try {
            check();
        } catch (ArithmeticException e) {
            System.out.println("caught" + e);
        }
    }
}
```

User defined Exceptions

throw	throws
throw keyword is used to throw an exception explicitly.	throws keyword is used to declare an exception possible during its execution.
throw keyword is followed by an instance of Throwable class or one of its sub-classes.	throws keyword is followed by one or more Exception class names separated by commas.
throw keyword is declared inside a method body.	throws keyword is used with method signature (method declaration).
We cannot throw multiple exceptions using throw keyword.	We can declare multiple exceptions (separated by commas) using throws keyword.

Java provides rich set of built-in exception classes like: ArithmeticException, IOException, NullPointerException etc. all are available in the java.lang package and used in exception handling. These exceptions are already set to trigger on pre-defined conditions such as when you divide a number by zero it triggers ArithmeticException.

Apart from these classes, Java allows us to create our own exception class to provide own exception implementation. These type of exceptions are called as user-defined exceptions or custom exceptions.

-> To create the user defined exceptions, we have to follow the following procedure

procedure:

1. All exception classes are extending from Exception class. So that to create user defined exception we should create a class that extends any one of the following 2 classes

a) Exception (Checked Exceptions)

b) RuntimeException (Unchecked Exceptions)

2. Create a parameterized constructor which calls super class parameterized constructor to set the Exception Message.

3. JVM don't know when to generate the user defined exception and how to create and throw the object for User defined exception so that we should explicitly create the object for User defined exception and throw manually using throw keyword.

Vote.java

```
1  import java.io.*;
2
3  class AgeException extends Exception {
4      AgeException(String msg) {
5          super(msg);
6      }
7  }
8
9  public class Vote {
10     public static void main(String args[]) throws IOException {
11         BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
12         System.out.println("Enter Your Age");
13         int age = Integer.parseInt(br.readLine());
14         try {
15             if (age < 18) {
16                 AgeException ae = new AgeException("Sorry You have to wait for " + (18 - age) + " Years");
17                 throw ae;
18             }
19             System.out.println("Congratulations you are eligible for Voting...");
20         } catch (AgeException ae) {
21             System.out.println(ae);
22         }
23     }
24 }
```

Knowledge – Check

1) What is package and why we need packages?
2) What is Import in java?
3) What is static import?
4) What is Wrapper class?
5) When we have primitive types why to go for Wrapper Classes?
6) What is Auto Boxing and Auto Unboxing?
7) What is Exception Hierarchy in Java?
8) Checked Exceptions Vs Un-Checked Exceptions?
9) How to handle Exceptions?
10) What is try-catch-finally?
11) What is the difference between throws and throw?
12) What is try with resources?
13) Why to write Multiple Catch blocks & do you any alternate?
14) How to create and throw User Defined Exception?
15) What is JVM's behaviour when Runtime Exception occurred?
16) What is Exception re-throw ?
17) Can you explain few checked exceptions?
18) Can you explain few Un-Checked exceptions?
19) Can you write a program which gives Stack Overflow Error?
20) What is the difference between final, finalize ( ) and finally block ?

Chapter - 8

Collections Framework

COLLECTIONS

-> Array is an collection of fixed number of homogeneous data elements

OR

-> An array represents a group of elements of same data type.

-> The main advantage of array is we can represent huge number of elements by using single variable. So, the readability of the code is improved.

Limitations of array:

-> Arrays are fixed in size that is once we create an array there is no chance of increasing or decreasing the size of array based on our requirement. Hence to use array concept we must know the size in advance which may not possible every time.

-> Arrays can hold only homogeneous data elements.

Example:

```
Car c = new Car [100];
c[0] = new Car(); // Valid
c[1] = new Bus(); //Invalid(Compile time Error)
```


-> We can resolve this problem by using Object type Array (Object [ ])

Example:

```
Object[] o=new Object[100];
o[0]=new Car();
o[1]=new Bus();
```

-> Arrays concept is not implemented based on some data structure hence we cannot expect ready-made method support. For every requirement we have to write the code explicitly. To overcome the above limitations, we should go for collection concept.

-> Collections are growable in nature that is based on our requirement we can increase or decrease the size hence memory point of view collections concept is recommended to use.

-> Collections can hold both homogeneous and heterogeneous objects.

-> Every Collection class is implemented based on some standard data structure hence for every requirement ready-made method support is available. As a programmer we can use these methods directly without writing the functionality on our own.

#	Array	Collection
1	Arrays are fixed in size	Collections are growable in nature
2	With respect to memory, Arrays are not recommended to use	With respect to memory, Collections are recommended to use
3	With respect to performance Arrays are recommended to use	With respect to performance Collections are not recommended to use
4	Arrays can hold only homogeneous datatype elements	Collections can hold homogeneous & heterogeneous datatype elements
5	No underlying data structure. Hence no readymade method support	Based on standard data structure. Hence it has readymade method support
6	It can hold both Primitives and Objects	It can hold only Objects

Collection: If we want to represent a group of objects as single entity then we should go for Collections.

Collection framework: It defines several classes and interfaces to represent a group of Objects as single entity.

Q) What is the difference between Collection and Collections?

Ans) Collection is an interface which can be used to represent a group of Objects as a single entity whereas Collections is an utility class present in java.util package to define several utility methods for Collection Objects.

- Collection is an interface
- Collections is a Class

-> All the collection classes are available in “java.util” (utility) package.

-> All the collection interfaces and collection class and together as collection frame work.

-> All the collection classes are classified into three categories

1) List

2) Set

3) Queue

4) Map

1. List:

- This category is used to store group of individual elements where the elements can be duplicated.

- List is an Interface whose object can not be created directly.

- To work with this category we have to use following implementations class of list interface

Ex: ArrayList, Linked list, Vector, Stack

2. Set:

- This category is used to store a group of individual elements. But they elements can't be duplicated.

- Set is an interface whose object cannot be created directly.

- To work with this category, we have to use following implementations class of Set interface

Ex: HashSet, LinkedHashSet and TreeSet

3. Queue

-> This category is used to hold the elements about to be processed in FIFO(First In First Out) order.

-> It is an ordered list of objects with its use limited to inserting elements at the end of the list and deleting elements from the start of the list, (i.e.), it follows the FIFO or the First-In-First-Out principle

Ex: PriorityQueue, BlockingQueue

4. Map:

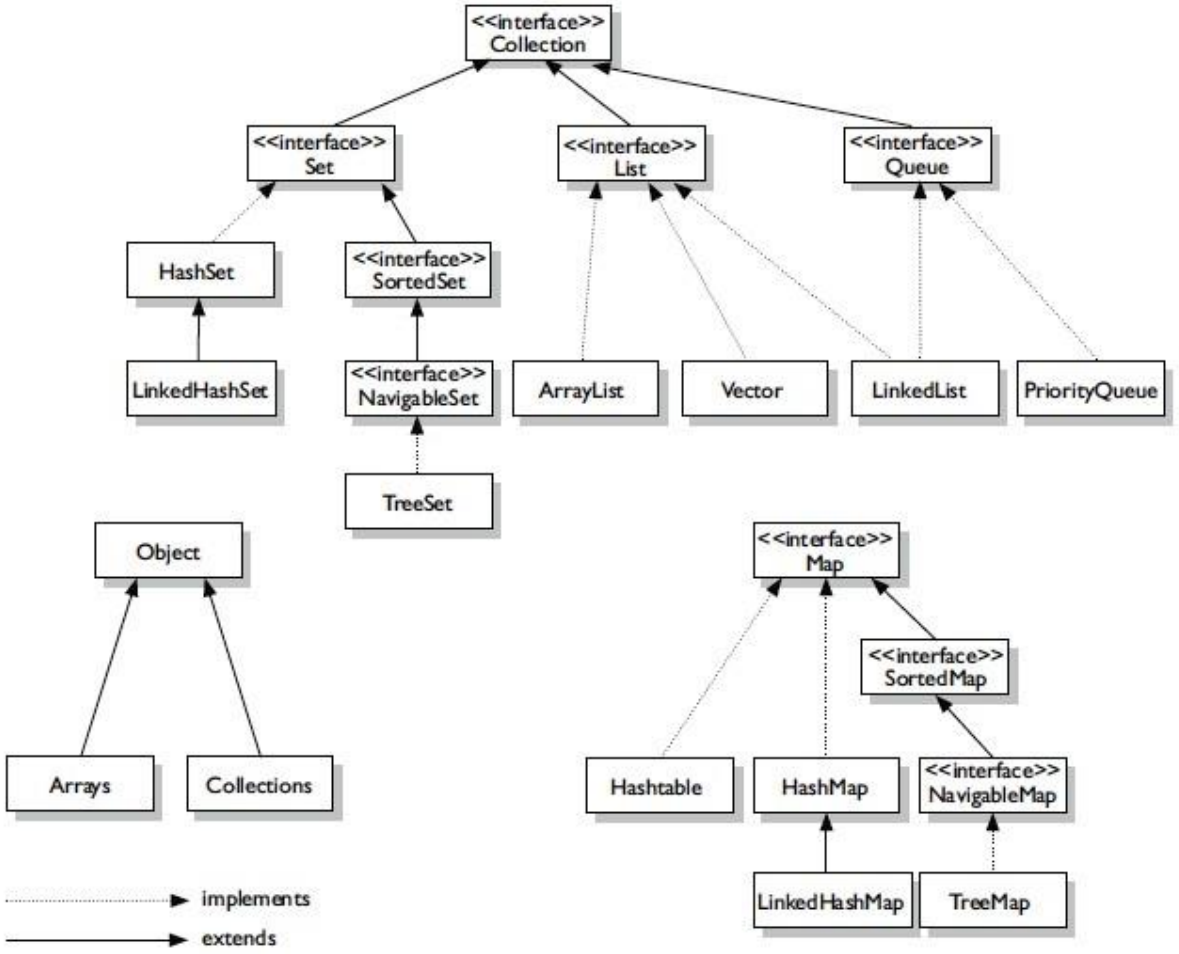
- This category is used to store the element in the form key value pairs where the keys can't be duplicated, values can be duplicated.

- Map is an interface whose object cannot be created directly.

- To work with this category, we have to use following implementation classes of Map interface

Ex: HashMap, LinkedHashMap, TreeMap, Hashtable

Collections Hierarchy



List interface

-> It is the child interface of Collection

-> If we want to represent a group of individual objects where duplicates are allowed and insertion order is preserved. Then we should go for List.

-> We can differentiate duplicate Objects and we can maintain insertion order by means of index hence "index plays important role in List"

List interface defines the following specific methods.

1) boolean add(int index,Object o);

2) boolean addAll(int index,Collection c);

3) Object get(int index);

4) Object remove(int index);

5) Object Set(int index,Object o);//to replace

6) int indexOf(Object o); //Returns index of first occurance of o

7) int lastIndexOf(Object o);

8)ListIterator listIterator();

ArrayList:

-> ArrayList is an implementation class of Collection interface

->The underlying data structure is resizable (Internally it will use Array to store data)

-> Duplicate Objects are allowed

-> Insertion order is preserved

-> Heterogeneous Objects are allowed

-> Null insertion is possible

ArrayList Constructors:

1) ArrayList al=new ArrayList();

-> It creates an empty ArrayList Object with default initial capacity 10

-> If ArrayList reaches its maximum capacity then a new ArrayList Object will be created with

NewCapacity=(Current Capacity*3/2)+1

2) ArrayList al=new ArrayList(int initialCapacity);

-> Creates an empty ArrayList Object with the specified initial capacity.

3) ArrayList al=new ArrayList(Collection c);

-> Creates an equivalent ArrayList Object for the given Collection that is this constructor meant for inter conversation between Collection Objects.

-> ArrayList and Vector classes implements RandomAccess interface so that any random element we can access with the same speed.

-> RandomAccess interface present in util package and doesn't contain any methods. It is a marker interface.

ArrayListDemo.java

```java
package in.ashokit;

import java.util.ArrayList;

public class ArrayListDemo {

    public static void main(String[] args) {

        ArrayList al = new ArrayList();
        al.add("ashokit");
        al.add(101);
        al.add(202.05);
        System.out.println(al); // [ashokit, 101, 202.05]
        al.remove(1);
        System.out.println(al); // [ashokit, 202.05]
        al.add("hyd");
        al.add("java");
        System.out.println(al); // [ashokit, 202.05, hyd, java]
    }
}
```

LinkedList

-> LinkedList is one of the implementation classes of Collection interface

-> The underlying data structure is double LinkedList

-> If our frequent operation is insertion or deletion in the middle then LinkedList is the best choice

-> If our frequent operation is retrieval then LinkedList is not best option

-> Duplicate Objects are allowed

-> Insertion order is preserved

-> Heterogeneous Objects are allowed

-> NULL insertion is possible

-> Implements Serializable and Cloneable interfaces but not RandomAccess

Note: Usually we can use linked list to implement Stacks and Queues to provide support for this requirement LinkedList class defines the following 6 specific methods.

1) void addFirst(Object o);

2) void addLast(Object o);

3) Object getFirst();

4) Object getLast();

5) Object removeFirst();

6) Object removeLast();

LinkedList Constructors:

1) LinkedList l=new LinkedList();

It creates an empty LinkedList Object.

2) LinkedList l=new LinkedList(Collection c);

To create an equivalent LinkedList Object for the given Collection.

IQ: what is the diff b/w ArrayList and LinkedList

-> ArrayList is slower in insertion and deletion of elements because it internally requires shifting operations, But faster in accessing the elements because ArrayList use index position for every element.

-> LinkedList is faster in insertion and deletion of elements because it just require modifying the links of nodes instead of shifting operations, But slower in accessing the elements because LinkedList does not use any index position.

LinkedListDemo.java

```
1  package in.ashokit;
2
3  import java.util.LinkedList;
4
5  public class LinkedListDemo {
6      public static void main(String[] args) {
7          LinkedList ll = new LinkedList();
8          ll.add("ashokit");
9          ll.add(40);
10         ll.add(null);
11         ll.add("ashokit");
12         System.out.println(ll);// [ashokit,40,null,ashokit]
13         ll.add(0, "java");
14         System.out.println(ll);// [java,ashokit,40,null,ashokit]
15         ll.set(0, "oracle");
16         System.out.println(ll);// [oracle,ashokit,40,null,ashokit]
17         ll.removeLast();
18         System.out.println(ll);// [oracle,java,40,null]
19         ll.addFirst("ashok");
20         System.out.println(ll);// [ashok,oracle,java,40,null]
21     }
22 }
23
```

Vector

-> Vector is the implementation class of List interface which is also used to store group of individual objects where duplicate values are allowed

-> Vector is exactly similar to ArrayList but ArrayList is not a synchronized class where Vector is a synchronized class

-> Vector is also called legacy class because it is available from java 1.0 version.

Vector Class Constructors

1) Vector<E> v = new Vector<E>();
2) Vector<E> v = new Vector<E>(int capacity);
3) Vector<E> v = new Vector<E>(Collection obj);

VectorDemo.java

```
1  package in.ashokit;
2
3  import java.util.Vector;
4
5  public class VectorDemo {
6
7      public static void main(String[] args) {
8          Vector<String> v = new Vector<String>();
9          v.add("sachin");
10         v.add(new String("sehwagh"));
11         v.add("kohli");
12         v.add("dhoni");
13         v.add("suresh");
14         System.out.println(v);
15         System.out.println(v.size());
16     }
17 }
18
```

Stack

-> Stack is a child class of Vector and implements List interface

-> Stack stores a group of objects b using a mechanism called LIFO

-> LIFO stands for Last in first out , it means last inserted element deleted first.

Stack Class Constructor:

Stack<E> s = new Stack<E>( );

Methods:

-> We can use all collection Methods

-> We can also use legacy methods of Vector class like addElement(), removeElement(), setElementAt(),.....

-> But if we want to follow the LIFO mechanism, we should use Stack methods like follows

1. E push(E obj) : this method will add new element into the Stack

2. E pop() : this method deletes the top element available on Stack

3. E peek() : this method just returns the top element available on Stack

StackDemo.java

```
1  package in.ashokit;
2
3  import java.util.Stack;
4
5  public class StackDemo {
6
7      public static void main(String[] args) {
8          Stack<Double> s = new Stack<Double>();
9          s.push(10.2);
10         s.push(50.2);
11         s.push(30.2);
12         s.push(40.2);
13         s.push(70.2);
14         System.out.println(s); // [10.2, 50.2, 30.2, 40.2, 70.2]
15         System.out.println(s.pop()); // 70.2
16         System.out.println(s); // [10.2, 50.2, 30.2, 40.2]
17         System.out.println(s.peek()); // 40.2
18         System.out.println(s); // [10.2, 50.2, 30.2, 40.2]
19     }
20 }
21
```

Cursors of Collection Framework

-> cursors are mainly used to access the elements of any collection

-> we have following 3 types of cursors in Collection Framework

1.Iterator

2.ListIterator

3.Enumeration

Iterator

-> this cursor is used to access the elements in forward direction only

-> this cursor can be applied Any Collection (List, Set)

-> while accessing the methods we can also delete the elements

-> Iterator is interface and we cannot create an object directly.

-> if we want to create an object for Iterator, we have to use iterator () method

Creation of Iterator:

Iterator it = c.iterator();

here iterator() method internally creates and returns an object of a class which implements Iterator interface.

Methods

1. boolean hasNext()
2. Object next()
3. void remove()

ArrayListDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListDemo {

    public static void main(String[] args) {

        ArrayList<String> al = new ArrayList<>();
        al.add("ashok");
        al.add("it");
        al.add("java");
        al.add("training");

        Iterator<String> iterator = al.iterator();

        while (iterator.hasNext()) {
            String next = iterator.next();
            System.out.println(next);
        }
    }
}
```

2. ListIterator

-> This cursor is used to access the elements of Collection in both forward and backward directions

-> This cursor can be applied only for List category Collections

-> While accessing the methods we can also add,set,delete elements

-> ListIterator is interface and we can not create object directly.

-> If we want to create an object for ListIterator we have to use listIterator() method

creation of ListIterator:

ListIterator<E> it = l.listIterator();

here listIterator() method internally creates and returns an object of a class which implements ListIterator interface.

Methods

1. boolean hasNext();
2. Object next();
3. boolean hasPrevious();
4. Object previous();
5. int nextIndex();
6. int previousIndex();
7. void remove();
8. void set(Object obj);
9. void add(Object obj);

ArrayListDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.ListIterator;

public class ArrayListDemo {

    public static void main(String[] args) {

        ArrayList<String> al = new ArrayList<>();
        al.add("ashok");
        al.add("it");
        al.add("java");
        al.add("training");

        ListIterator<String> listIterator = al.listIterator();

        while (listIterator.hasNext()) {
            String next = listIterator.next();
            System.out.println(next);
        }
    }
}
```


3. Enumeration

-> this cursor is used to access the elements of Collection only in forward direction

-> this is legacy cursor can be applied only for legacy classes like Vector,Stack,Hashtable.

-> Enumeration is also an interface and we can not create object directly.

-> If we want to create an object for Enumeration we have to use a legacy method called elements() method

Creation of Enumeration:

Enumeration e = v.elements();

here elements() method internally creates and returns an object of a class which implements Enumeration interface.

Methods

1. boolean hasMoreElements()

2. Object nextElement();

VectorDemo.java

```java
package in.ashokit;

import java.util.Enumeration;
import java.util.Vector;

public class VectorDemo {

    public static void main(String[] args) {
        Vector<Integer> v = new Vector<Integer>();
        v.add(10);
        v.add(25);
        v.add(50);
        v.add(20);
        v.add(25);
        v.add(23);
        v.add(25);
        System.out.println(v);

        Enumeration e = v.elements();
        while (e.hasMoreElements()) {
            System.out.print(e.nextElement() + " ");
        }
    }
}
```

	Iterator	ListIterator	Enumeration
Applicable to	List, Set	List implemented clases	legacy classes
Accessing Direction	only forward	both forward & backward	only forward
Operations	access, remove	access,remove, add, set	only accessing
method to create	iterator()	listIterator()	elements()
legacy	✗	✗	✓
No.of Methods	3	9	2

Set category

HashSet

-> HashSet is the implementation class of Set interface which is also used to store group of individual objects but duplicate values are not allowed

-> HashSet internally follows hashtable structure where all the elements are stored using hashing technique which will improve the performance by reducing the waiting time.

-> HashSet is not a synchronized class

-> HashSet supports only one null value.

-> HashSet is called unordered Collection because it is not guarantee for insertion order of elements.

creation of HashSet:

HashSet<E> hs = new HashSet<E>();

HashSet<E> hs = new HashSet<E>(int capacity);

HashSet<E> hs = new HashSet<E>(int capacity,float loadfactor);

HashSet<E> hs = new HashSet<E>(Collection obj);

Methods

1. boolean add(E obj)

2. boolean remove(E obj)

3. int size()

4. void clear()

5. boolean contains(E obj)

6. boolean isEmpty()

HashSetDemo.java

```
1  package in.ashokit;
2
3  import java.util.HashSet;
4  import java.util.Iterator;
5
6  public class HashSetDemo {
7
8      public static void main(String ar[]) {
9          HashSet<Integer> hs = new HashSet<Integer>();
10         hs.add(27);
11         hs.add(12);
12         hs.add(null);
13         hs.add(9);
14         hs.add(21);
15         System.out.println(hs.size());
16         for (Integer e : hs) {
17             System.out.println(e);
18         }
19         Iterator it = hs.iterator();
20         while (it.hasNext()) {
21             System.out.println(it.next());
22         }
23     }
24 }
```

LinkedHashSet

-> LinkedHashSet is the implementation class of Set interface which is also used to store group of individual objects but duplicate values are not allowed

-> LinkedHashSet internally follows hashtable + doubly linked list structures

-> LinkedHashSet is not a synchronized class

-> LinkedHashSet supports only one null value.

-> LinkedHashSet is called as ordered Collection because it is guarantee for insertion order of elements.

Creation of LinkedHashSet:

LinkedHashSet<E> hs = new LinkedHashSet<E>();

LinkedHashSet<E> hs = new LinkedHashSet<E>(int capacity);

LinkedHashSet<E> hs = new LinkedHashSet<E>(int capacity,float loadfactor);

LinkedHashSet<E> hs = new LinkedHashSet<E>(Collection obj);

LinkedHashSetDemo.java

```java
package in.ashokit;

import java.util.Iterator;
import java.util.LinkedHashSet;

public class LinkedHashSetDemo {

    public static void main(String ar[]) {
        LinkedHashSet<Integer> hs = new LinkedHashSet<Integer>();
        hs.add(27);
        hs.add(12);
        hs.add(null);
        hs.add(9);
        hs.add(21);
        System.out.println(hs.size());
        for (Integer e : hs) {
            System.out.println(e);
        }
        Iterator it = hs.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

SortedSet:

1) It is child interface of Set

2) If we want to represent a group of "unique Objects" according to some sorting order then we should go for SortedSet interface.

3) That sorting order can be either default natural sorting order OR customized sorting order.

SortedSet interface defines the following 6 specific methods

1) Object first();

2) Object last();

3) SortedSet headSet(Object o);

It returns the elements whose elements are < o

4) SortedSet tailSet(Object o);

It returns the elements whose elements are >= o

5) SortedSet subSet(Object o1,Object o2);

It returns the elements whose elements are >=o1 and < o2

6) Comparator comparator();

Returns the comparator Object that describes under lying sorting technique. If we follow default natural sorting order then this method returns null.

```
import java.util.SortedSet;
import java.util.TreeSet;
public class SortedSetDemo {
        public static void main(String[] args) {
        SortedSet ss=new TreeSet();
        for(int i=10;i<=20;i++)
        ss.add(i);
        System.out.println(ss.first());//10
        System.out.println(ss.last());//20
        System.out.println(ss.headSet(16));//[10, 11, 12, 13, 14, 15]
        System.out.println(ss.tailSet(18));//[18,  19,  20]
```

```
        System.out.println(ss.subSet(12,17));//[12, 13, 14, 15, 16]
        System.out.println(ss.comparator());//null
        }
}
```

TreeSet

-> TreeSet is the implementation class of Set interface which is also used to store group of individual objects but duplicate values are not allowed

-> TreeSet internally follows tree structure

-> TreeSet is not a synchronized class

-> TreeSet is called as unordered Collection because it is not guarantee for insertion order of elements but all elements are stored in sorted order(bydefault ascending order)

-> TreeSet supports only one null value if TreeSet is Empty otherwise TreeSet does not support null values because it internally perform comparision operations but we never compare a null value. with any object and it will throw a RuntimeException saying NullPointerException

-> TreeSet does not allow to store different types of objects because it internally perform comparison.operations but we never compare a 2 different types of it will throw a RuntimeException saying ClassCastException

Creation of TreeSet

TreeSet<E> ts = new TreeSet<E>();

TreeSet<E> ts = new TreeSet<E>(SortedSet);

TreeSet<E> ts = new TreeSet<E>(Comparator);

TreeSet<E> ts = new TreeSet<E>(Collection obj)

TreeSetDemo.java

```java
package in.ashokit;

import java.util.TreeSet;

public class TreeSetDemo {

    public static void main(String[] args) {
        TreeSet<String> ts = new TreeSet<>();

        ts.add("java");
        ts.add("programming");
        ts.add("language");

        // ts.add(10); // invalid

        System.out.println(ts);
    }
}

```

NULL acceptance:

For the empty TreeSet as the first element "null" insertion is possible but after inserting that null if we try to insert any other value then we will get NullPointerException.

For the non-empty TreeSet if we try to insert null then we will get NullPointerException.

TreeSetDemo.java

```java
package in.ashokit;

import java.util.TreeSet;

public class TreeSetDemo {

    public static void main(String[] args) {
        TreeSet ts = new TreeSet();
        ts.add(new StringBuffer("Java"));// ClassCastException
        ts.add(new StringBuffer("DevOps"));
        ts.add(new StringBuffer("MySQL"));
        ts.add(new StringBuffer("Python"));
        System.out.println(ts);
    }
}

```

String class and all wrapper classes implements Comparable interface but StringBuffer class does not implement Comparable interface hence in the above program we will get ClassCastException.

Comparable interface:

Comparable interface present in java.lang package and contains only one method compareTo() method.

public int compareTo(Object obj);

Example: obj1.compareTo(obj2);

It returns -ve if obj1 comes before obj2

It returns +ve if obj2 comes before obj1

It returns 0 if obj1 and obj2 are equal

CompareTo.java

```
1 package in.ashokit;
2
3 public class CompareTo {
4     public static void main(String[] args) {
5         System.out.println("A".compareTo("Z"));// -25
6         System.out.println("z".compareTo("a"));// 25
7         System.out.println("A".compareTo("A"));// 0
8     }
9 }
```


If we depend on default natural sorting order then internally JVM will use compareTo() method to arrange Objects in sorting order.

CompareTo.java

```
1 package in.ashokit;
2
3 import java.util.TreeSet;
4
5 public class CompareTo {
6     public static void main(String[] args) {
7         TreeSet<Integer> ts = new TreeSet<>();
8         ts.add(21);
9         ts.add(12);
10        ts.add(1);
11        ts.add(0);
12        ts.add(15);
13        ts.add(17);
14        System.out.println(ts); // [0, 1, 12, 15, 17, 21]
15    }
16 }
```

We can define our own customized sorting by Comparator Object.

Comparable meant for default natural sorting order.

Comparator meant for customized sorting order.

Comparator interface:

Comparator interface present in java.util package. It defines the

following two methods.

1) public int compare(Object o1,Object o2);

It returns -ve if o1 comes before o2

It returns +ve if o1 comes after o2

It returns 0 if o1 and o2 are equal

2) public boolen equals(Object o);

Whenever we are implementing comparator interface we have to provide implementation only for compare() method.

Implementing equals() method is optional because it is already available from Object class through inheritance.

Requirement:- Write a program to insert integer Objects into the TreeSet where the

sorting order is descending order.

Example:

```
import java.util.Comparator;
import java.util.TreeSet;
public class TreeSetComparator {
        public static void main(String[] args) {
        TreeSet ts=new TreeSet(new MyComparator());// --- >1
        ts.add(10);
        ts.add(0);
        ts.add(15);
        ts.add(5);
        ts.add(20);
        System.out.println(ts);//[20, 15, 10, 5, 0]
        }
```

```
}
class MyComparator implements Comparator {
        public int compare(Object o1,Object o2) {
        Integer i1=(Integer)o1;
        Integer i2=(Integer)o2;
        if(i1<i2)
                return 1;
        else if(i1>i2)
                return -1;
        else
                return 0;
        }
}
```


Comparable vs Comparator

For predefined Comparable classes default natural sorting order is already available, if we are not satisfied with default natural sorting order then we can define our own customized sorting order by Comparator.

For predefined non Comparable classes (Like StringBuffer) default natural sorting order is not available, we have to define our own sorting order by using Comparator Object.

For our own classes(Like Customer,Student and Employee) we can define default natural sorting order by using Comparable interface. The person who is using our class, if he is not satisfied with default natural sorting order then he can define his own sorting order by using Comparator Object.

Example:

```
import java.util.Comparator;
import java.util.TreeSet;
class Employee implements Comparable {
String name;
int eid;
```

```
Employee(String name,int eid) {
this.name=name;
this.eid=eid;
}
public String toString(){
        return name+"-"+eid;
}
public int compareTo(Object o) {
int eid1=this.eid;
int eid2=((Employee)o).eid;
if(eid1 < eid2)
        return -1;
else if(eid1>eid2)
        return 1;
else
        return 0;
}
}
public class Comp {
        public static void main(String[] args) {
        Employee e1=new Employee("Raju",100);
        Employee e2=new Employee("Rani",101);
        Employee e3=new Employee("John",102);
        Employee e4=new Employee("Smith",103);
        Employee e5=new Employee("Ashok",104);
        TreeSet t=new TreeSet();
        t.add(e1);
        t.add(e2);
        t.add(e3);
```

```
        t.add(e4);
        t.add(e5);

        System.out.println(t);
        TreeSet t2=new TreeSet(new MyComparator());
        t2.add(e1);
        t2.add(e2);
        t2.add(e3);
        t2.add(e4);
        t2.add(e5);
        System.out.println(t2);
  }
}

class  MyComparator  implements  Comparator {
        public int compare(Object o1,Object o2) {
                Employee e1=(Employee)o1;
                Employee e2=(Employee)o2;
                String s1=e1.name;
                String          s2=e2.name;
                return s1.compareTo(s2);
  }
}
```

Comparable in Java	Comparator in Java
Comparable interface is used to sort the objects with natural ordering.	Comparator in Java is used to sort attributes of different objects.
Comparable interface compares "this" reference with the object specified.	Comparator in Java compares two different class objects provided.
Comparable is present in java.lang package.	A Comparator is present in the java.util package.
Comparable affects the original class, i.e., the actual class is modified.	Comparator doesn't affect the original class
Comparable provides compareTo() method to sort elements.	Comparator provides compare() method, equals() method to sort elements.

Map category

Map interface is not child interface of Collection and hence we cannot apply Collection interface methods. Map interface defines the following specific methods.

1) Object put(Obect key,Object value);

2) void putAll(Map m);

3) Object get(Object key);

4) Object remove(Object key);

5) boolean containsKey(Object key);

6) boolean containsValue(Object value);

7) boolean isEmpty();

8) int size();

9) void clear();

10) Set keySet(); //We will get the set of keys

11) Collection values(); //We will get the set of values

12) Set entrySet(); //We will get the set of entryset

HashMap

-> HashMap is the implementation class of Map interface which is used to store group of objects in the form of Key-Value pairs where but Keys cannot be duplicated but values can be duplicated

-> HashMap internally follows hashtable data structure

-> HashMap is not a synchronized class

-> HashMap supports only one null value for Key Objects but we can store multiple null values for Value Object

-> HashMap is called unordered Map because it is not guarantee for insertion order of elements.

creation of HashMap:

HashMap<K,V> hm = new HashMap<K,V>();

HashMap<K,V> hm = new HashMap<K,V>(int capacity);

HashMap<K,V> hm = new HashMap<K,V>(int capacity, float loadfactor);

HashMap<K,V> hm = new HashMap<K,V>(Map obj);

MapDemo.java

```
4
5  public class MapDemo {
6
7      public static void main(String[] args) {
8
9          HashMap<Integer, String> hm = new HashMap<Integer, String>();
10
11         hm.put(11, "sachin");
12         hm.put(25, "kohli");
13         hm.put(12, "yuvaraj");
14
15         System.out.println(hm);
16         System.out.println(hm.size());
17
18         Set<Integer> ks = hm.keySet();
19         System.out.println(ks);
20
21         Collection<String> cv = hm.values();
22         System.out.println(cv);
23
24         Set<?> entry = hm.entrySet();
25         System.out.println(entry);
26
27         System.out.println(hm.containsKey(12));
28
29         System.out.println(hm);
30     }
31 }
32
```

LinkedHashMap

-> LinkedHashMap is the implementation class of Map interface which is also used to store group of3 objects in the form of Key-Value pairs where Keys can't be duplicated but values can be duplicated

-> LinkedHashMap internally follows Hashtable + doubly linked list structures

-> LinkedHashMap is not a synchronized class

-> LinkedHashMap supports only one null value for Key Objects but we can store multiple null values for Value Object

-> LinkedHashMap is called as ordered Map because it is guarantee for insertion order of elements.

SortedMap:

-> It is the child interface of Map.

-> If we want to represent a group of key-value pairs according to some sorting order of keys then we should go for SortedMap.

-> Sorting is possible only based on the keys but not based on values.

-> SortedMap interface defines the following 6 specific methods.

1) Object firstKey();

2) Object lastKey();

3) SortedMap headMap(Object key);

4) SortedMap tailMap(Object key);

5) SortedMap subMap(Object key1,Object key2);

6) Comparator comparator();

TreeMap

-> TreeMap is the implementation class of Map interface which is also used to store group of objects in the form of Key-Value pairs where Keys can't be duplicated but values can be duplicated.

-> TreeMap internally follows tree structure

-> TreeMap is not a synchronized class

-> TreeMap is called as unordered Map because it is not guarantee for insertion order of elements, but. all elements are

Hashtable

-> Hashtable is the implementation class of Map interface which is also used to store group of objects in the form of Key-Value pairs where Keys can't be duplicated but values can be duplicated

-> Hashtable is exactly similar to HasMap but Hashtable is a synchronized class where HashMap is not a synchronized class

-> Hashtable does not support null values for both Keys and Values

Constructors:

1) Hashtable ht=new Hashtable();

It creates an empty Hashtable Object with default initial capacity 11 and default fill ratio 0.75

2) Hashtable ht=new Hashtable(int initialCapacity);

3) Hashtable ht=new Hashtable(int initialCapacity,float fillRatio);

4) Hashtable ht=new Hashtable(Map m);

HashTable.java

```
1  package in.ashokit;
2
3  import java.util.Enumeration;
4  import java.util.Hashtable;
5
6  public class HashTable {
7
8      public static void main(String[] args) {
9          Hashtable<String, Integer> ht = new Hashtable<String, Integer>();
10         ht.put("sachin", 200);
11         ht.put("rohit", 264);
12         ht.put("sehwagh", 219);
13         ht.put("ganguly", 183);
14         ht.put("dhoni", 183);
15         System.out.println(ht);
16         Enumeration e = ht.keys();
17         while (e.hasMoreElements()) {
18             System.out.println(e.nextElement());
19         }
20
21     }
22 }
23
```

IdentityHashMap

It is exactly same as HashMap except the following differences.

1) In the case of HashMap JVM will always use equals() method to identify duplicate keys.

2) But inn the case of IdentityHashMap JVM will use == (double equal to operator) to identify duplicate keys.

Example:

```
import java.util.HashMap;

public class HashMapExample {

        public static void main(String[] args) {

                HashMap hm=new HashMap();

                Integer i1=new Integer(10);

                Integer i2=new Integer(10);

                hm.put(i1,"John");

                hm.put(i2,"Smith");

                System.out.println(hm);//{10=Smith}

        }

}
```


In the above program i1 and i2 are duplicate keys because i1.equals(i2) returns true.

In the above proggram if we replace HashMap with IdentityHashMap then i1 and i2 are not duplicate keys because i1==i2 returns false. In this case the output is as below.

        Here 10==i1 is false

        Here 10==i2 is false

WeakHashMap:

-> It is exactly same as HashMap except the following differences.

-> In the case of normal HashMap, an Object is not eligible for Garbage Collection even though it does not have any references if it is associated with HashMap. That means HashMap dominates Garbage Collector.

-> But in the case of WeakHashap if an Object does not have any references, then it is always eligible GC even though if it is associated with WeakHashMap that means GC dominates WeakHashMap.

WeakHashMapDemo.java

```java
package in.ashokit;

import java.util.WeakHashMap;

public class WeakHashMapDemo {
    public static void main(String[] args) throws Exception {
        WeakHashMap<Person, String> whm = new WeakHashMap<>();
        Person t = new Person();
        whm.put(t, "Ashok");
        System.out.println(whm);// {Person = Ashok}
        t = null;
        System.gc();// finalize() method is called
        Thread.sleep(500);
        System.out.println(whm);// {}
    }

}

class Person {
    public String toString() {
        return "Temp";
    }

    public void finalize() {
        System.out.println("finalize() method is called");
    }
}
```

In the above program if we replace WeakHashMap with normal HashMap then Object won't be destroyed by the garbage collector.

Properties:

1) Properties class is the child class of Hashtable.

2) If anything changes frequently such type of values not recommended to hardcode in java application because for every change we have to recompile, rebuild and redeploy the application and even server restart also required. Sometimes it creates big business impact to the client.

3) Such type of variables we have to hardcode in property files and we have to read the values from the property files.

4) The main advantage in this approach is if there is any change in property files automatically those changes will be available to java application just redeployment is enough.

5) By using Properties Object, we can read and hold properties from property files into java application.

Constructor:

Properties p=new Properties ();

In Properties both key and value should be String type only.

Methods:

1) String getProperty (String propertyName);

2) String setProperty(String propertyName,String propertyValue);

3) Enumeration propertyNames();

4) void load(InputStream is);

5) void store(OutputStream os,String comment);

db.properties

```
uname=ashokit
pwd=india@123
```


PropertiesDemo.java

```java
package in.ashokit;

import java.io.FileInputStream;

public class PropertiesDemo {

    public static void main(String[] args) throws IOException {
        Properties p = new Properties();

        FileInputStream fis = new FileInputStream("db.properties");
        p.load(fis);

        System.out.println(p); // {uname=system, pwd=manager}
    }
}

```

Collections

-> Collections class is one of the utility classes in Java Collections Framework.

-> The java.util package contains the Collections class.

-> Collections class is basically used with the static methods that operate on the collections or return the collection.

-> All the methods of this class throw the NullPointerException if the collection or object passed to the methods is null.

Methods

```
1) public static  Collection synchronizedCollection(Collection c);

2) public static  Set synchronizedSet(Set s);

3) public static  List synchronizedList(List list);

4) public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);

5) public static  SortedSet synchronizedSortedSet(SortedSet s);

6) public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```


Java String tokenizer

-> StringTokenizer is a class present in the java.util package and it is used to break a String into tokens based on provided delimiter.

-> Delimiter can be specified either at the time of object creation or on a per-token basis.

Following are the constructors in StringTokenizer class

```
1. StringTokenizer(String str)

2. StringTokenizer(String str, String delim)

3. StringTokenizer(String str, String delim, booleanreturnValue)
```

Following are the methods in StringTokenizer class

1. booleanhasMoreTokens()
2. String nextToken()
3. String nextToken(String delim)
4. booleanhasMoreElements()
5. Object nextElement()
6. intcountTokens()

// Java program to split a String using space as delimiter

Demo.java

```
1  package in.ashokit;
2
3  import java.util.StringTokenizer;
4
5  public class Demo {
6
7      public static void main(String[] args) throws Exception {
8          StringTokenizer obj = new StringTokenizer("Ashok IT Java", " ");
9          while (obj.hasMoreTokens()) {
10             System.out.println(obj.nextToken());
11         }
12     }
13 }
```

// java program to split a String using colon as delimiter

Demo.java

```
1  package in.ashokit;
2
3  import java.util.StringTokenizer;
4
5  public class Demo {
6
7      public static void main(String[] args) throws Exception {
8          String a = " : ";
9          String b = "Welcome : to : ashokit : . : How : are : You : ?";
10         StringTokenizer c = new StringTokenizer(b, a);
11         int count1 = c.countTokens();
12         for (int i = 0; i < count1; i++)
13             System.out.println("token [" + i + "] : " + c.nextToken());
14         StringTokenizer d = null;
15         while (c.hasMoreTokens()) {
16             System.out.println(d.nextToken());
17         }
18     }
19 }
```

Knowledge – Check

1) What is Collection and why we need it?
2) What is Collection framework?
3) What is List and when to use it?
4) How ArrayList works internally?
5) How LinkedList works internally?
6) When to use ArrayList and When to Use LinkedList?
7) What is Cursor and How many cursors available?
8) What is Set?
9) How HashSet works internally?
10) What is TreeSet and when to use it?
11) What is Comparable?
12) What is Comparator?
13) How to sort Objects?
14) What is Map and when to use it?
15) What is Hash Map?
16) How HashMap works internally?
17) How to iterate a Map?
18) What is Weak HashMap?
19) What is Identity HashMap?
20) What is Collections?
21) What is Properties Class?
22) What are Fail-fast collection and Fail-Safe Collection?

Chapter – 9

Multi - Threading

Single tasking:

Single tasking means executing 1 task at a time. Here much of the processor time will be wasted

because waiting time very high and processor gives response late.

Eg: DOS OS

Multi-tasking:

Multi-tasking means executing multiple tasks at the same time simultaneously. Here processor time utilized in a proper manner and performance of the application by reducing waiting time and processor gives response faster.

Eg: Windows OS, Student can listen and can write at a time

We can achieve multitasking in following 2 ways

1. Processor based multi-tasking

-> Processor based multi-tasking means executing multiple tasks at the same time simultaneously where each task is independent of other tasks having separate memory and resources

-> Processor based multi-tasking is an operating system approach

Eg:

java program,

listen to music,

download songs,

copy softwares,

chating,

....

2. Thread Based multi tasking

-> Thread based multi tasking means executing different parts of the same program at the same time simultaneously where each task is independent sometimes or dependent sometimes of other tasks which are having common memory and resources.

-> Thread based multi tasking is programming approach

-> Each different of the program is called Trheads

Eg: games, web based apps,....

Thread vs Process :

Process : a program execution is often referred as process.

Thread : a Thread is subset(part) or sub process of the process.

(or)

A thread is flow of execution and for every thread separate independent job(task).

NOTE: weather it is process based or thread based the main objective multitasking is reduce responsive time of the system and improve the performance.

Advantage of Multithreading

Multithreading reduces the CPU idle time that increase overall performance of the system. Since thread is lightweight process then it takes less memory and perform context switching as well that helps to share the memory and reduce time of switching between threads.

Main Thread

-> For every java program there will be a default thread by JVM which is called as Main Thread.

-> The entry point for Main Thread is main() method

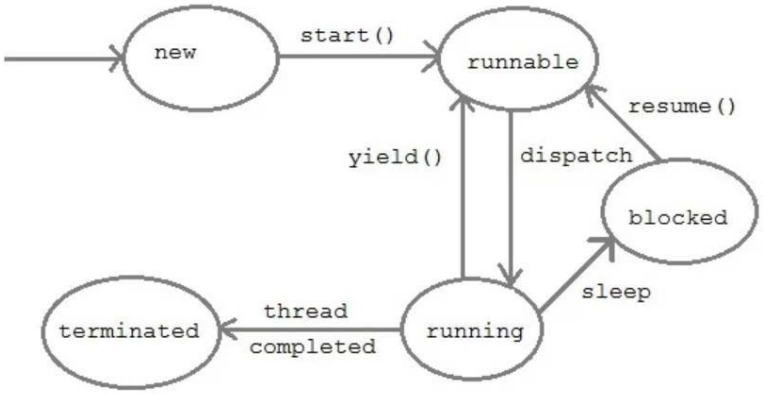
// wap to display the info of currently Running Thread

ThreadDemo.java

```
1  package in.ashokit;
2
3  public class ThreadDemo {
4
5      public static void main(String args[]) {
6
7          Thread ct = Thread.currentThread();
8          System.out.println(ct);
9          System.out.println(ct.getName());
10         System.out.println(ct.getPriority());
11         System.out.println(ct.getThreadGroup());
12
13     }
14 }
15
```

Life cycle of a Thread

Like process, thread have its life cycle that includes various phases like: new, running, terminated etc. we have described it using the below image.



New: A thread begins its life cycle in the new state. It remains in this state until the start() method is called on it.

Runnable: After invocation of start () method on new thread, the thread becomes runnable.

Running: A thread is in running state if the thread scheduler has selected it.

Waiting: A thread is in waiting state if it waits for another thread to perform a task. In this stage the thread is still alive.

Terminated: A thread enter the terminated state when it completes its task.

What is Thread Priority?

-> When we create a thread, we can assign its priority. We can set different priorities to different Threads but it doesn't guarantee that a higher priority thread will execute first than a lower priority thread.

-> The thread scheduler is the part of Operating System implementation and when a Thread is started, its execution is controlled by Thread Scheduler and JVM doesn't have any control over its execution.

-> Thread priorities are integer numbers.

-> Using thread priority, we can decide when we should switch from one thread in running state to another. This is the context switching process wherein we switch contexts of the threads.

Anytime a thread can voluntarily release its control over CPU. Then the thread with the highest priority can take over.

-> Similarly, a higher priority thread can preempt any other thread with lower priority.

-> Thread class provides a setPriority () method that is used to set the priority for the thread.

-> We can also use constants MIN_PRIORITY, MAX_PRIORITY, or NORM_PRIORITY in the place of integers.

User defined threads

-> In java we can also create user defined Threads by using any one of the following 2 approaches.

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

Creating the user defined Thread by extending the Thread class

if we want to create the user defined Threads using this approach we have to follow following 5 steps

Step-1: creating a class that extends a Thread class

```
class MyThread extends Thread{
}
```

Step-2: overriding the run() method

```
public void run(){
    //statements;
}
```


Note: Here run() method is called entry-point for the user defined thread and all the job done by the user defined thread will be written here itself.

Step-3: creating an object for user defined Thread class

```
MyThread mt = new MyThread();
```

Step-4: attaching user defined thread with main ThreadGroup

```
Thread t = new Thread(mt);
```

Step-5: starting the execution of the Thread by calling start()

```
t.start();
```

-> When we call start() method the thread will be registered with ThreadScheduler and next will invoke the run() method.

MyThread.java

```
1  package in.ashokit;
2
3  class MyThread extends Thread {
4      public void run() {
5          for (int i = 1; i <= 10; i++) {
6              System.out.println("User Thread Value:" + i);
7          }
8      }
9
10     public static void main(String args[]) {
11         MyThread mt = new MyThread();
12         Thread t = new Thread(mt);
13         t.start();
14     }
15 }
```

Creating user defined Thread by implementing Runnable interface

-> if we want to create the user defined Threads using this approach, we have to follow following 5 steps

Step1: creating a class that implements Runnable interface

class MyThread implements Runnable {

}

Step2: overriding the run() method

public void run(){

//statements;

}

Step3: creating an object for user defined Thread class

MyThread mt = new MyThread();

Step4: attaching user defined Thread with main ThreadGroup

Thread t = new Thread(mt);

Step5: starting user defined Thread by calling start() method

t.start();

MyThread.java

```java
package in.ashokit;

class MyThread implements Runnable {

    public void run() {
        for (int i = 1; i <= 10; i++) {
            System.out.println("User Thread Value:" + i);
        }
    }

    public static void main(String args[]) {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        t.start();
    }
}
```

Q) What is the difference between extending Thread class and implementing Runnable interface

-> If we create any thread by extending Thread class then we have no chance for extending from any other class.

-> But if we create any thread by implementing Runnable interface then we have a chance for extending from any one class.

-> It is always recommended to create the user defined threads by implementing Runnable interface only.

What is the difference between calling t.run() and t.start() ?

-> We can call run() method directly but no thread will be created / registered with Thread scheduler but run()method executes like a normal method by Main Thread.

-> But if we call start() method thread will be registered with thread scheduler and it calls run() method.

MyThread.java

```
1  package in.ashokit;
2
3  class MyThread implements Runnable {
4
5      public void run() {
6          Thread t = Thread.currentThread();
7          for (int i = 1; i <= 5; i++) {
8              System.out.println(t.getName()+ " Thread Value:" + i);
9          }
10     }
11
12     public static void main(String args[]) {
13         MyThread mt = new MyThread();
14         Thread t = new Thread(mt);
15         t.start();
16         //t.run();
17     }
18 }
```

Note: When we call start ( ) method it is creating thread and printing output like below

Thread-0 Thread Value:1

Thread-0 Thread Value:2

Thread-0 Thread Value:3

Thread-0 Thread Value:4

Thread-0 Thread Value:5

MyThread.java

```
1  package in.ashokit;
2
3  class MyThread implements Runnable {
4
5      public void run() {
6          Thread t = Thread.currentThread();
7          for (int i = 1; i <= 5; i++) {
8              System.out.println(t.getName() + " Thread Value:" + i);
9          }
10     }
11
12     public static void main(String args[]) {
13         MyThread mt = new MyThread();
14         Thread t = new Thread(mt);
15         // t.start();
16         t.run();
17     }
18 }
```

Note: When we call run ( ) method it is not creating thread and printing output like below with main thread

main Thread Value:1

main Thread Value:2

main Thread Value:3

main Thread Value:4

main Thread Value:5

Creating Multiple Threads

When we execute multiple threads at the same time simultaneously then we never get same output for every execution because Thread Scheduler will decide which thread should execute in next step.

MyThread.java

```
1  package in.ashokit;
2
3  class MyThread implements Runnable {
4
5      public void run() {
6          Thread t = Thread.currentThread();
7          for (int i = 1; i <= 5; i++) {
8              System.out.println(t.getName() + " Thread Value:" + i);
9          }
10     }
11
12     public static void main(String args[]) {
13         MyThread mt1 = new MyThread();
14         MyThread mt2 = new MyThread();
15         MyThread mt3 = new MyThread();
16
17         Thread t1 = new Thread(mt1);
18         Thread t2 = new Thread(mt2);
19         Thread t3 = new Thread(mt3);
20
21         t1.start();
22         t2.start();
23         t3.start();
24     }
25 }
```

Data in consistency problem

-> When we execute multiple Threads which are acting on same object at the same time simultaneously then there is chance of occurring data inconsistency problem in the application

-> Data inconsistency problem will occur because one thread is updating the value at the same time other thread is using old value.

-> To resolve this data inconsistency problem, we have to **synchronize** the object on which multiple Threads are acting.

Thread synchronization

-> Thread synchronization is a process of allowing only one thread to use the object when multiple

Threads are trying to use the particular object at the same time.

-> To achieve this Thread synchronization, we have to use a java keyword or modifier called

“synchronized ".

-> We can achieve Thread synchronization in following 2 ways

1. synchronized blocks

If we want to synchronize a group of statements then we have to go for synchronized blocks

syntax:

```
synchronized(object) {
        //statements
}
```

2. synchronized methods

If we want to synchronize all the statements of the particular method then we have to go for synchronized methods.

syntax:

```
synchronized returntype methodname(parameters){
        //statements;
}
```

Note: Thread synchronization concept is recommended to use only when we run multiple threads which are acting on same object otherwise this concept is not required.

Q) Which is more preferred - Synchronized method or Synchronized block

-> In Java, synchronized keyword causes a performance cost.

-> A synchronized method in Java is very slow and can degrade performance. So we must use synchronization keyword in java when it is necessary else, we should use Java synchronized block that is used for synchronizing critical section only.

Deadlock

-> When we execute multiple Threads which are acting on same object that is synchronized at the same time simultaneously then there is another problem may occur called deadlock

-> Dead lock may occur if one thread holding resource1 and waiting for resource2 release by the Thread2, at the same time Thread2 is holding on resource2 and waiting for the resource1 released by the Thread1 in this case 2 Threads are continuously waiting and no thread will execute this situation is called as deadlock.

-> To resolve this deadlock situation there is no any concept in java, programmer only responsible for writing the proper logic to resolve the problem of deadlock.

```
// below is the java program which gives dead lock
package in.ashokit;
class MyThread {
	public static void main(String[] args) {
		final String resource1 = "Ashok IT";
		final String resource2 = "Java Training";
		// t1 tries to lock resource1 then resource2
		Thread t1 = new Thread() {
			public void run() {
				synchronized (resource1) {
					System.out.println("Thread 1: locked resource 1");
					try {
						Thread.sleep(100);
					} catch (Exception e) {
					}

					synchronized (resource2) {
						System.out.println("Thread 1: locked resource
2");
					}
				}
			}
		};
		// t2 tries to lock resource2 then resource1
		Thread t2 = new Thread() {
			public void run() {
				synchronized (resource2) {
					System.out.println("Thread 2: locked resource 2");
					try {
						Thread.sleep(100);
					} catch (Exception e) {
```

```
                                        }
                                        synchronized (resource1) {
                                                System.out.println("Thread 2: locked resource
1");
                                        }
                                }
                        }
                };
                t1.start();
                t2.start();
        }
}
```

Daemon Thread

-> Daemon threads is a low priority thread that provide supports to user threads. These threads can be user defined and system defined as well.

-> Garbage collection thread is one of the systems generated daemon thread that runs in background.

-> When a JVM founds daemon threads it terminates the thread and then shutdown itself, it does not care Daemon thread whether it is running or not.

MyThread.java

```java
package in.ashokit;

class MyThread extends Thread {

    public void run() {
        if (Thread.currentThread().isDaemon()) {// checking for daemon thread
            System.out.println("daemon thread work");
        } else {
            System.out.println("user thread work");
        }
    }

    public static void main(String[] args) {
        MyThread t1 = new MyThread();// creating thread
        MyThread t2 = new MyThread();
        MyThread t3 = new MyThread();

        t1.setDaemon(true);// now t1 is daemon thread

        t1.start();
        t2.start();
        t3.start();
    }
}
```

Methods of Object class which are related to threads

1. wait(): : This method used to make the particular Thread wait until it gets a notification.

2. notify(): This method used to send the notification to one of the waiting thread so that thread enter into running state and execute the remaining task.

3. notifyAll(): This method used to send the notification to all the waiting threads so that all thread enter into running state and execute simultaneously.

-> All these 3 methods are available in Object class which is super most class so that we can access all these 3 methods in any class directly without any reference.

-> These methods are mainly used to perform Inner thread communication

-> By using these methods we can resolve the problems like Producer-Consumer problem, Reader-Writer problem, ...

MyThread.java

```
class MyThread extends Thread {
    int total;
    public void run() {
        for (int i = 1; i <= 1000; i++) {
            total = total + i;
            if (total > 10000) {
                synchronized (this) {
                    notify();
                }
            }
            try {
                Thread.sleep(5);
            } catch (InterruptedException ie) { }
        }
        System.out.println("User Thread total:" + total);
    }

    public static void main(String args[]) {
        MyThread mt = new MyThread();
        Thread t = new Thread(mt);
        t.start();
        try {
            synchronized (mt) {
                mt.wait();
            }
        } catch (InterruptedException ie) { }
        System.out.println("Main Thread total:" + mt.total);
    }
}
```

Note: We must call wait(), notify(), notifyAll() methods in side the synchronized blocks or synchronized methods otherwise it will throw a runtime exception called IllegalMonitorStateException.

```
// Write a java program on Produce – Consumer

package in.ashokit;

public class Store {

	int value;
	boolean pro_thread = true;

	synchronized void produce(int i) {
		if (pro_thread == true) {
			value = i;
			System.out.println("Procuder Has Produced Product " + value);
			pro_thread = false;
			notify();
		}
		try {
			wait();
		} catch (InterruptedException ie) {
		}
	}

	synchronized int consume() {
		if (pro_thread == true) {
			try {
				wait();
			} catch (InterruptedException ie) {
			}
		}
		pro_thread = true;
		notify();
		return value;
	}

}

class Producer implements Runnable {
	Store sr;

	Producer(Store sr) {
		this.sr = sr;
	}

	public void run() {
		int i = 1;
		while (true) {
			sr.produce(i);
```

```
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException ie) {
                    }
                    i++;
                }
            }
}

class Consumer implements Runnable {

        Store sr;

        Consumer(Store sr) {
            this.sr = sr;
        }

        public void run() {
            while (true) {
                int res = sr.consume();
                System.out.println("Consumer Has Taken Product " + res);
                try {
                    Thread.sleep(1000);
                } catch (InterruptedException ie) {
                }
            }
        }
}

class ProducerConsumer {
        public static void main(String args[]) {
            Store sr = new Store();
            Producer p = new Producer(sr);
            Consumer c = new Consumer(sr);
            Thread t1 = new Thread(p);
            Thread t2 = new Thread(c);
            t1.start();
            t2.start();
        }
}
```

Knowledge – Check

1) What is Multi - Threading?
2) Why we need multi-Threading?
3) What is Thread Schedular?
4) What is Thread Priority?
5) What is Thread Life Cycle?
6) How to create Thread?
7) Difference between start () method and run ( ) method ?
8) Thread vs Runnable
9) Runnable Vs Callable
10) What is Synchronization
11) Synchronized Method Vs Synchronized Block
12) What is Dead Lock
13) Write a java program which gives Dead Lock
14) What is Inter - Thread Communication
15) What is Producer – Consumer problem
16) What is Executor Framework

Chapter – 10

- File Handling Introduction
- File
- FineReader
- FileWriter
- Serialization
- De-Serialization

File Handling

-> File handling is a crucial part of any programming language.

-> File handling means performing various operations on a file, like reading, writing, editing, etc.

Common file handling operations

- Creating a new file.
- Writing data in aa file.
- Reading an existing file data.
- Deleting a file.

-> Java provides us library classes and various methods to perform file handling easily.

-> All these methods are present in the File Class of the java.io package.

So, first of all, before starting these programs, we need to import this package and the file class.

```
import java.io.File;  // Importing the File class

File obj = new File("filename.txt"); // Specify the name of the file
```

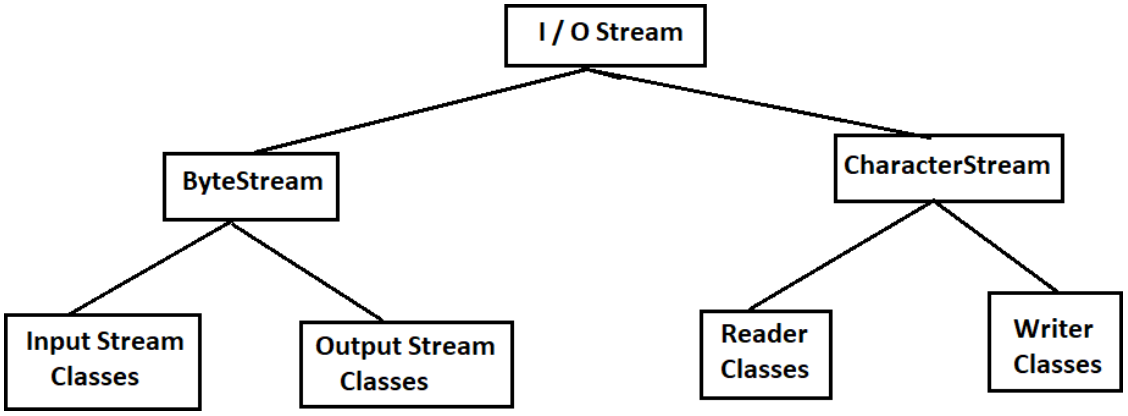

-> Java uses i/o stream to perform file-related operations. Let us understand the concept of stream first.

Stream in Java

-> Stream is a concept of java that pipelines a sequence of objects to obtain the desired result.

-> A stream cannot be called to be a data structure, rather it just takes input from the collection of I/O.

-> A stream can be classified into two types: Byte Stream and Character Stream.



Byte Stream:

The byte stream deals with mainly byte data. We know that one byte is equal to eight-bit. Thus, this stream mainly deals with 8bit of data. This stream performs an Input-output operation per 8bit of data.

The byte stream contains two stream classes, Input Stream classes and Output Stream Classes.

1. Input Stream Classes: This stream helps take input(Read Data) from the collection in I/O File.

2. Output Stream Classes: This stream helps to give output(Write Data) into the collection in I/O File.

The most commonly used Input and Output Stream Classes are FileInputStream and FileOutputStream.

Character Stream:

There is also Character Stream which allows I/O operation on 16bit of Unicode data at a time. Character Stream takes 2 bytes of data at a time. It is faster as it can take double the intake as compared to a byte stream. Character streams usually use Byte Stream classes to implement operations.

The two main classes used in Character Stream are FileReader and FileWriter.

File and Directory

-> A file is a named location that can be used to store related information. For example,

-> main.java is a Java file that contains information about the Java program.

-> A directory is a collection of files and subdirectories. A directory inside a directory is known as subdirectory.

Note: Using java program we can create both files and directories

-> The File class has many useful methods for creating and getting information about files. For example:

Method	Type	Description
canRead()	Boolean	Tests whether the file is readable or not
canWrite()	Boolean	Tests whether the file is writable or not
createNewFile()	Boolean	Creates an empty file
delete()	Boolean	Deletes a file
exists()	Boolean	Tests whether the file exists
getName()	String	Returns the name of the file
getAbsolutePath()	String	Returns the absolute pathname of the file
length()	Long	Returns the size of the file in bytes
list()	String[]	Returns an array of the files in the directory
mkdir()	Boolean	Creates a directory

// java program to create and write data to a file

WriteData.java

```java
package in.ashokit;

import java.io.*;

public class WriteData {

    public static void main(String[] args) {
        File myFile = new File("ashokit.txt");
        try {
            FileWriter fileWriter = new FileWriter(myFile);
            fileWriter.write("Ashok IT - Learn Here.. Lead Anywhere..!!");
            fileWriter.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
        System.out.println("Done....");
    }
}
```

// java program to read data from file

ReadData.java

```
1  package in.ashokit;
2
3  import java.io.File;
6
7  public class ReadData {
8
9      public static void main(String[] args) {
10         File myFile = new File("ashokit.txt");
11         try {
12             Scanner sc = new Scanner(myFile);
13             while (sc.hasNextLine()) {
14                 String line = sc.nextLine();
15                 System.out.println(line);
16             }
17             sc.close();
18         } catch (FileNotFoundException e) {
19             e.printStackTrace();
20         }
21     }
22 }
```

// java program to delete file

DeleteFile.java

```
1  package in.ashokit;
2
3  import java.io.File;
4
5  public class DeleteFile {
6      public static void main(String[] args) {
7
8          File myFile = new File("ashokit.txt");
9          if (myFile.delete()) {
10             System.out.println("I have deleted: " + myFile.getName());
11         } else {
12             System.out.println("Some problem occurred");
13         }
14     }
15 }
16
```

Java BufferedReader Class

-> Java BufferedReader class is used to read the text from a character-based input stream.

-> It can be used to read data line by line by readLine( ) method. It makes the performance fast.

-> It inherits Reader class.

ReadData.java

```java
package in.ashokit;

import java.io.BufferedReader;

public class ReadData {

    public static void main(String[] args) {
        File myFile = new File("ashokit.txt");
        try {
            FileReader fr = new FileReader(myFile);
            BufferedReader br = new BufferedReader(fr);
            String line = br.readLine();
            while (line != null) {
                System.out.println(line);
                line = br.readLine();
            }
            br.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Scanner Class

-> Scanner class is mostly used to scan the input and read the input of primitive (built-in) data types like int, decimal, double, etc.

-> Scanner class basically returns the tokenized input based on some delimiter pattern.

-> A Scanner class implements Iterator (string), Closeable, and AutoCloseable interfaces.

ScannerDemo.java

```java
package in.ashokit;

import java.util.Scanner;

public class ScannerDemo {

    public static void main(String args[]) {
        Scanner sc = new Scanner(System.in);
        System.err.println("Enter Any String");
        String str = sc.next();
        System.err.println("str=" + str);
        System.err.println("Enter Any Integer");
        int i = sc.nextInt();
        System.err.println("i=" + i);
        System.err.println("Enter Any Double");
        double d = sc.nextDouble();
        System.err.println("d=" + d);
        System.err.println("Enter Any Character");
        char ch = sc.next().charAt(0);
        System.err.println("ch=" + ch);
        sc.close();
    }

}
```

Serialization and De-Serialization

-> Serialization is a process of converting an object into a sequence of bytes which can be persisted to a disk or database or can be sent through streams.

-> The reverse process of creating object from sequence of bytes is called deserialization.

-> A class must implement Serializable interface present in java.io package in order to serialize its object successfully. Serializable is a marker interface that adds serializable behaviour to the class implementing it.

-> Java provides Serializable API encapsulated under java.io package for serializing and deserializing objects which include

- java.io.serializable
- java.io.Externalizable
- ObjectInputStream
- ObjectOutputStream

Java Marker interface

Marker Interface is a special interface in Java without any field and method.

Marker interface is used to inform compiler that the class implementing it has some special behaviour or meaning. Some examples of Marker interface are,

- java.io.serializable
- java.lang.Cloneable
- java.rmi.Remote
- java.util.RandomAccess

All these interfaces does not have any method and field. They only add special behaviour to the classes implementing them.

To implement serialization and deserialization, Java provides two classes ObjectOutputStream and ObjectInputStream.

ObjectOutputStream class

It is used to write object states to the file. An object that implements java.io.Serializable interface can be written to strams. It provides various methods to perform serialization.

Methods of ObjectOutputStream Class

writeObject() method: Serializes the object, and writes it to ObjectOutputStream.

public final void writeObject(Object obj) throws IOException {}

close() : Closes a current stream.

public void close() throws IOException {}

flush(): Flushes the current output stream.

public void flush() throws IOException { }

ObjectInputStream class

An ObjectInputStream deserializes objects and primitive data written using an ObjectOutputStream.

Methods of ObjectInputStream Class

readObject(): This method converts the stream of bytes to the state of the object.

public final Object readObject() throws IOException, ClassNotFoundException{}

close(): As the name suggests, this method basically closes the Input stream.

public void close() throws IOException {}

// Serialization Example

Employee.java

```
1  package in.ashokit;
2
3  import java.io.Serializable;
4
5  public class Employee implements Serializable {
6
7      String name;
8      int id;
9
10     public Employee(String name, int id) {
11         this.name = name;
12         this.id = id;
13     }
14 }
15
```

Employee.java SerializationDemo.java

```java
package in.ashokit;

import java.io.FileOutputStream;
import java.io.ObjectOutputStream;

public class SerializationDemo {

    public static void main(String args[]) {

        try {
            Employee emp = new Employee("Ashok", 132);

            FileOutputStream fout = new FileOutputStream("ashokit.txt");
            ObjectOutputStream out = new ObjectOutputStream(fout);

            out.writeObject(emp);
            out.flush();
            out.close();
            System.out.println("Serialization successful!");
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

// De-Serialization Java Program

Employee.java SerializationDemo.java DeSerializationDemo.java

```java
package in.ashokit;

import java.io.FileInputStream;
import java.io.ObjectInputStream;

public class DeSerializationDemo {

    public static void main(String[] args) {
        try {
            FileInputStream fin = new FileInputStream("ashokit.txt");

            ObjectInputStream in = new ObjectInputStream(fin);

            Employee emp1 = (Employee) in.readObject();
            in.close();

            System.out.println("Deserialization successful");
            System.out.println("Name: " + emp1.name);
            System.out.println("id: " + emp1.id);
        } catch (Exception ex) {
            System.out.println("Exception");
        }
    }

}
```

Java Transient Keyword

-> Let's consider a case where you are serializing an object and want that a certain field of the object doesn't get serialized.

Ex: I don't want to serialize sensitive or secret data like my password, atm pin, ssn etc..

-> To achieve this, we can take the help of the transient keyword. So, the Java transient keyword helps to prevent a particular field from being serialized.

```
import java.io.Serializable;

public class Employee implements Serializable{

        private static final long serialVersionUID = 123L;
        String name;
        int id;
        transient int age;
}
```

What is the serialVersionUID?

-> The serialization runtime associates with each serializable class a version number, called a serialVersionUID, which is used during deserialization to verify that the sender and receiver of a serialized object have loaded classes for that object that are compatible with respect to serialization.

-> If the receiver has loaded a class for the object that has a different serialVersionUID than that of the corresponding sender's class, then deserialization will result in an InvalidClassException.

-> A serialzable class can declare its own serialVersionUID explicitly by declaring a field named serialVersionUID that must be static, final, and of type long

Note: If we don't specify serialVersionUID then JVM will assign one randome serialVersionUID. Its always better to mention our serialVersionUID.

```
public class AppleProduct implements Serializable {

  private static final long serialVersionUID = 1234567L;

   public String headphonePort;
   public String thunderboltPort;

}
```

Points to Remember while Implementing the Serializable Interface in Java

-> Serializable Interface must be implemented by all the associated objects.

-> If the parent class has already implemented the Serializable interface, in that case, a child class doesn't have to implement it.

-> During the serialization process, only non-static data members are saved.

-> While converting the byte stream back to the original object, the constructor of the object is not called.

-> Serializable Interface has no methods or fields of its own.

Knowledge – Check

1) What is File Handling
2) What is File class
3) What are the methods available in File class
4) What is FileReader and FileWriter
5) How to read data from a File
6) How to write data into File
7) BufferedReader class vs FileReader class
8) Copy data from One file into another file
9) Read data from two files and write into another file
10) Write a java program to count no.of words in a file
11) What is Scanner class
12) What is Serializable interface ?
13) What is Serialization ?
14) What is De-Serialization ?
15) What is serialVersionUID
16) What is transient keyword

Chapter – 11

- Generics
- Garbage Collection
- Reflection API
- Inner Classes

Java Generics

-> Generics was first introduced in Java 1.5. Now it is one of the most profound features of java programming language.

-> Generic programming enables the programmer to create classes, interfaces and methods in which type of data is specified as a parameter.

-> Generics provide type safety. Type safety means ensuring that an operation is being performed on the right type of data.

Note: Before Generics was introduced, generalized classes, interfaces or methods were created using references of type Object because Object is the super class of all classes in Java, but this way of programming did not ensure type safety.

```
Class_name <data type>  reference_name  =  new Class_name <data type> ( );

OR

Class_name <data type> reference_name  =  new Class_name <> ( );
```

Note: This is also known as Diamond Notation of creating an object of Generic type.

Generic class

Generic class is a class which can hold any kind of objects, to create Generic class we have to specify generic type <T> after the class name.

syntax:

```
class ClassName<T>{

        //members
}
```

-> We can take multiple Generic Types also for class like below

```
class ClassName<T1,T2,T3,...> {

 }
```

Generic interfaces

We can also create a generic interface by specifying the <T> after the interface name.

syntax:

```
interface InterfaceName <T> {
        //members
}
```

// Java program with Generics

Demo.java

```
1  package in.ashokit;
2
3  class Gen<T> {
4      T ob; // an object of type T is declared<
5
6      Gen(T o) { // constructor
7          ob = o;
8      }
9
10     public T getOb() {
11         return ob;
12     }
13 }
14
15 public class Demo {
16
17     public static void main(String[] args) {
18         Gen<Integer> iob = new Gen<>(100); // instance of Integer type Gen Class
19         int x = iob.getOb();
20         System.out.println(x);
21         Gen<String> sob = new Gen<>("Hello"); // instance of String type Gen Class
22         String str = sob.getOb();
23         System.out.println(str);
24     }
25 }
```

In the above program, we first passed an Integer type parameter to the Generic class. Then, we passed a String type parameter to the same Generic class. Hence, we reused the same class for two different data types. Thus, Generics helps in code reusability with ease.

Java Reflection API

-> Reflection means ability of a software to analyse itself. In Java, Reflection API provides facility to analyse and change runtime behaviour of a class at runtime.

-> For example, using reflection at the runtime you can determine what method, field, constructor or modifiers a class supports.

-> The java.lang.Class class provides methods that are used to get metadata and manipulate the run time behaviour of a class.

-> The java.lang and java.lang.reflect packages provide many classes for reflection and get metadata of a particular class.

-> One of the advantages of reflection API is, we can manipulate private members of the class too.

What is reflect package?

java.lang.reflect package encapsulates several important interfaces and classes. These classes and interface define methods which are used for reflection.

Some Important Classes of java.lang.reflect package

Class	What it does ?
Array	allow you to dynamically create and manipulate arrays
Constructor	gives information about constructor of a class
Field	provide information about field
Method	provide information about method
Modifier	provide information about class and member access modifier
Proxy	supports dynamic proxy classes

Uses of Reflection

- Developing IDE
- Debugging and Test tools
- Loading drivers and providing dynamic information

Disadvantages of Reflection

- Low performance
- Security risk
- Violation of Oops concept

// java program to get Class object

Demo.java

```
1  package in.ashokit;
2
3  class Employee {
4      int empId;
5      String name;
6  }
7
8  public class Demo {
9
10     public static void main(String[] args) throws ClassNotFoundException {
11         Employee employee = new Employee();
12         // employee object
13         Class name = employee.getClass();
14         System.out.println(name);
15         // string object
16         name = "hello".getClass();
17         System.out.println(name);
18     }
19
20 }
```

Class.forName ( )

-> Class is a predefined class available in java.lang package.

-> Inside Class we have a static method which is Class.forName(String cls). This method is used to load classes into JVM using fully qualified name.

Demo.java

```
1  package in.ashokit;
2
3  class Employee {
4      int empId;
5      String name;
6  }
7
8  public class Demo {
9
10     public static void main(String[] args) throws Exception {
11         // Employee class
12         Class name = Class.forName("in.ashokit.Employee");
13         System.out.println(name);
14
15         // creating object
16         Object instance = name.newInstance();
17         Employee emp = (Employee) instance;
18         System.out.println(emp);
19
20         // String class
21         name = Class.forName("java.lang.String");
22         System.out.println(name);
23     }
24
25 }
```

// Java program to access private variable outside the class using Reflection

-> By default private variables can't be accessed outside of the class. By using Reflection api we can access them outside of the class like below

Demo.java

```java
package in.ashokit;

import java.lang.reflect.Field;

class Employee {
    private int empId;
    String name;
}

public class Demo {

    public static void main(String[] args) throws Exception {
        // Employee class
        Class clz = Class.forName("in.ashokit.Employee");
        System.out.println(clz);

        // creating object
        Object instance = clz.newInstance();
        Employee emp = (Employee) instance;

        // accessing private variable in another class
        Field field = clz.getDeclaredField("empId");
        field.setAccessible(true);
        field.set(emp, 10);
        System.out.println(field.get(emp));

    }
}
```


// Java program to get Variables of class using Reflection

```java
public class Demo {

    public static void main(String[] args) throws Exception {
        // Employee class
        Class name = Class.forName("in.ashokit.Employee");
        // Get Fields metadata
        Field[] field = name.getDeclaredFields();
        for (Field f : field) {
            System.out.print(f.getType() + " ");
            System.out.println(f.getName());
            System.out.println(f.getModifiers());
        }
    }
}
```

// java program to get methods metadata using Reflection

```
14 public class Demo {
15
16     public static void main(String[] args) throws Exception {
17         // Employee class
18         Class name = Class.forName("in.ashokit.Employee");
19         // Get methods metadata
20         Method[] method = name.getDeclaredMethods();
21         for (Method m : method) {
22             System.out.println(m);
23             System.out.println(m.getDefaultValue());
24             System.out.println(m.getModifiers());
25             System.out.println(m.getName());
26             System.out.println(m.getParameterCount());
27             System.out.println(m.getReturnType());
28         }
29     }
30 }
```

Note: Like fields and methods we can get Constructors info also using Reflection API.

Garbage Collection in Java

-> The garbage collector in Java is solely responsible for deleting un-used / un-referenced objects.

-> In other languages, such as C or C++, the programmer is solely responsible for creating and deleting objects. This may result in memory depletion if the programmer forgets to dereference the objects.

-> In Java, programmers do not have to work on this. The JVM automatically destroys objects which have lost their reference.

Now, what do we mean when we say "lost their reference"?

Let us find out through a simple example.

Let us say we have a class called human which has a constructor Human (String name). This constructor initializes the current value of the string to the class variable name.

Now if we create an object of the class Human as follows.

Human object = new Human("Ashok");

The JVM creates a reference of the name "object" and points it to the data Ashok.

Now if I write

object=null;

The pointer to the value "Ashok" is now nullified. I cannot access the value anymore as there was only one reference pointing to it. This is unreachability of objects in memory.

This object of human named “Ashok” is now eligible for garbage collection.

An object is eligible for garbage collection if there are no references to it in the heap memory.

There are a few ways to make an object eligible for garbage collection. They are:

- You can nullify the reference variable.
- You can assign the same pointer to a different object.
- All objects created inside a method lose their reference outside the method and are thus eligible for garbage collection.
- Using Island of Isolation.

What is Island Isolation

When two objects ‘a’, and ‘b’ reference each other, and they are not referenced by any other object, it is known as island of isolation.

Demo.java

```
1  package in.ashokit;
2
3  public class Demo {
4
5      Demo i;
6
7      public static void main(String[] args) {
8          Demo obj1 = new Demo();
9          System.out.println("Demo object one has been created");
10         Demo obj2 = new Demo();
11         System.out.println("Demo object two has been created");
12         obj1.i = obj2;
13         obj2.i = obj1;
14         obj1 = null;
15         obj2 = null;
16         System.gc();
17     }
18
19     @Override
20     protected void finalize() throws Throwable {
21         System.out.println("The finalize method has been called on the object");
22     }
23 }
```

Finalization in Java

-> As soon as the garbage collector clears the object, the compiler executes the finalize() method.

-> This method generally contains actions which the JVM performs just before the object gets deleted.

-> The Object class contains the finalize() method. Remember to override the finalize method in the class whose objects will be garbage collected.

-> The finalize method throws a checked exception called “Throwable”.

Demo.java

```
1  package in.ashokit;
2
3  public class Demo {
4
5      @Override
6      protected void finalize() throws Throwable {
7          System.out.println("finalize method - called");
8      }
9
10     public static void main(String[] args) throws Exception {
11         Demo d = new Demo();
12         d = null; // nullifying reference
13         System.gc(); // calling GC
14     }
15 }
```

Methods for calling the Garbage Collector in Java

There are 2 ways to call the garbage collector in java

-> We can use the Runtime.getRuntime().gc() method- This class allows the program to interface with the Java Virtual machine. The “gc()” method allows us to call the garbage collector method.

-> We can also use the System.gc() method which is common.

Note: However, you cannot guarantee that these statements would run the garbage collector. This is because the Java Virtual Machine performs clean-up.

How Garbage Collection works internally

-> In java, Garbage Collection works in 3 phases

Phase-1 : Mark objects as alive

-> In this step, the GC identifies all the live objects in memory by traversing the object graph.

-> When GC visits an object, it marks it as accessible and thus alive. Every object the garbage collector visits is marked as alive. All the objects which are not reachable from GC Roots are garbage and considered as candidates for garbage collection.

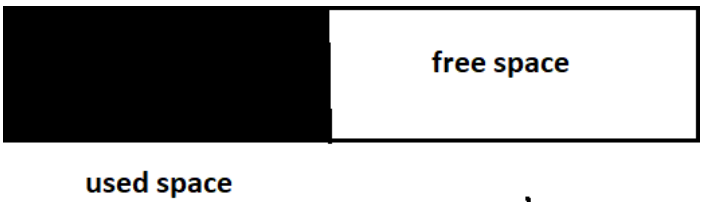
Phase-2 : Sweep dead objects

-> After marking phase, we have the memory space which is occupied by live (visited) and dead (unvisited) objects. The sweep phase releases the memory fragments which contain these dead objects.

Phase-3 : Compact remaining objects in memory

-> The dead objects that were removed during the sweep phase may not necessarily be next to each other. Thus, you can end up having fragmented memory space.

-> Memory can be compacted after the garbage collector deletes the dead objects, so that the remaining objects are in a contiguous block at the start of the heap.



The compaction process makes it easier to allocate memory to new objects sequentially.

Types of Garbage Collectors Java

Serial – This collector primarily relies on a single thread and continues to serially collect unreferenced objects.

Parallel – This collector uses multiple threads for collecting unreferenced objects in Java. There are two variants to it. The collector can use multiple threads for Minor garbage collection and one thread for Major garbage collection or vice versa.

CMS – This collector stands for Concurrent Mark Sweep. This essentially means that this collector uses multiple threads for major and Minor garbage collection, however, there is one important advantage of this collector. This collector does not allow the garbage collection to stop the application itself. This means that the collector is concurrent with the application, i.e, runs alongside the application itself. However, if the collector needs to update some objects from the young to the old generation but the collector itself is busy, a race condition occurs. This collection also uses up a lot of processing power.

Garbage First(G1) – This is similar to the CMS in terms of concurrency and usage of multithreading. This divides the memory into several parts based on the size. The collector specifically clears the part of the memory which has the greatest number of unreferenced objects at first.

Java Inner classes

-> If a class is declared with in another class, then this concept is called as Inner classes or nested classes

-> If a class contains other class, then it is called as outer class or top-level class

-> Class declared inside the outer class is called as inner class

```
syntax:
class A{
 class B{
 }
}
```

Here

A - outer class or top-level class

B - inner class

-> Here outer class can be default or public only but not private or protected or static

-> But inner class can be default or public or private or protected or static

-> The main advantage of Inner classes is that we can access the members of (both instance and static) Outer class inside the inner class directly without taking the help of any object.

-> inner classes can also be called as helper classes

-> inner classes are hidden from other class of same package or other package

Types of inner classes

We have following 2 types of inner classes

1. non-static Inner classes
2. static Inner classes

1. non-static inner classes

-> If any inner class is created without using any static keyword then it is called as non-static Nested classes or Inner classes

-> Non-static Nested or Inner classes can contain only non-static members(instance members)

-> based on the place where we declare these inner classes are again classified into following 3 types

1. Regular Inner class (member level)

2. Method local Inner class (statement level)

3. Anonymous Inner class(expression level

1. Regular Inner class

-> If an inner class is declared outside the methods of Outer class then it is called as Regular Inner class.

-> Regular Inner class are always created at member level

-> Regular Inner class can be default or public or private or protected

syntax:

Outer.java

```
1  package in.ashokit;
2
3  class Outer {
4      class Inner {
5      }
6
7      void outerMethod() {
8      }
9  }
10
```


if we compile the above program compiler will generate .class for both outer class and inner class

available in the program like follows,

1 Outer.class 2. Outer$Inner.class

// Creating an object for Regular Inner class inside the instance methods of Outer class

Outer.java

```
1  package in.ashokit;
2
3  class Outer {
4      class Inner {
5          void innerMethod() {
6              System.out.println("Inner class innerMethod()");
7          }
8      }
9
10     void outerMethod() {
11         Inner i = new Inner();
12         i.innerMethod();
13         System.out.println("Outer class outerMethod()");
14     }
15
16     public static void main(String args[]) {
17         Outer o = new Outer();
18         o.outerMethod();
19     }
20 }
```

2. Method local Inner class

-> If we declare an inner class inside the methods of Outer class then it is called as Method Local Inner class.

-> Method Local Inner classes are always created in side the method at statements level.

-> A method local Inner class can not be created as private, public, protected or static

syntax:

Outer.java

```
1  package in.ashokit;
2
3  class Outer {
4      void outerMethod() {
5          class Inner {
6
7          }
8      }
9  }
10
```

-> if we compile the above program then compiler will create following 2 .class files

1. Outer.class 2.Outer$1Inner.class

3. Anonymous Inner class

-> If any inner class is declared with out any name then it is called as Anonymous Inner class

-> Anonymous inner classes are always created at expression level

-> If we want to create Anonymous Inner classes we have to take the help of any existing class or interface

-> At the time of creating the object of Anonymous Inner classes only Anonymous Inner classes will be created.

-> For anonymous classes we can create only 1 object because it does't contain any name so that memory can be utilized properly.

// anonymous inner class using Interface

Outer.java

```
1  package in.ashokit;
2
3  class Outer {
4      public static void main(String args[]) {
5          Runnable r = new Runnable() { // anonymous Inner class
6              public void run() {
7                  Thread t = Thread.currentThread();
8                  for (int i = 1; i <= 10; i++) {
9                      System.out.println(t.getName() + " value:" + i);
10                 }
11             }
12         };
13         Thread t = new Thread(r);
14         t.start();
15     }
16 }
17
```

// anonymous inner class using Class

Outer.java

```
1  package in.ashokit;
2
3  class Outer {
4      public static void main(String args[]) {
5          Thread mt = new Thread() { // anonymous Inner class
6              public void run() {
7                  Thread t = Thread.currentThread();
8                  for (int i = 1; i <= 10; i++) {
9                      System.out.println(t.getName() + " value:" + i);
10                 }
11             }
12         };
13         Thread t = new Thread(mt);
14         t.start();
15     }
16 }
```

2. static Nested classes

-> If we declare any nested class using static key word then it is called as static Nested classes

->We can create static regular classes but we cannot create any method local static or anonymous static nested classes

->static nested classes can have both static and non-static members

Outer.java

```
1  package in.ashokit;
2
3  class Outer {
4      int x = 10;
5      static int y = 20;
6
7      static class Inner {
8          void innerMethod() {
9              System.out.println("Inner class innerMethod()");
10             // System.out.println("x="+x); -invalid
11             System.out.println("y=" + y);
12         }
13     }
14
15     void outerMethod() {
16         System.out.println("Outer class outerMethod()");
17     }
18
19     public static void main(String args[]) {
20         Outer o = new Outer();
21         o.outerMethod();
22         Outer.Inner i = new Outer.Inner();
23         i.innerMethod();
24     }
25 }
26
```

Knowledge – Check

1) What is Generics and why we need Generics?
2) What is Reflection API ?
3) What is the use of Class.forName ( )
4) How many ways available to create Object for a class ?
5) Write a java program to print methods available in the class
6) Write a java program to print variables available in the class
7) How to set values to private variables outside of class
8) What is Garbage Collection?
9) How Garbage Collection works in Java?
10) Which algorithm used by Garbage Collector?
11) How many types of Garbage Collectors available in Java?
12) What is Inner class?
13) Why we need inner classes?
14) How many types of inner classes available in Java?
15) What is Anonymous implementation ?

Chapter - 12

JAVA 8 – New Features

Java 8 Features

-> Java 8 introduced lot of new features

-> Java 8 version changed coding style with those new features

Main aim of java - 8

-> To simplify programming

-> To enable functional programming

-> To write more readable & concise code

New Features in Java 8

- Interface changes (default & static methods)
- Lambda Expressions
- Functional Interfaces
    - Consumer
    - Supplier
    - Predicate
    - Function
- Stream API
- Date & Time API changes
- Optional class
- Spllterator
- Stringjoiner
- Method References
- Constructor References
- Collections Framework changes

Interface Changes in Java 8

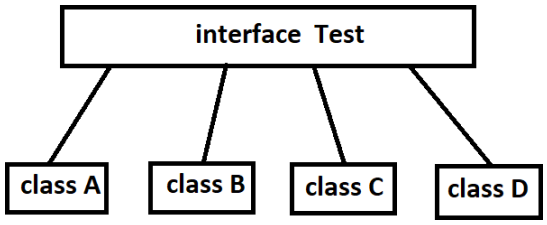
-> Prior to java 8, interface should have only abstract methods (methods without body)

-> Java 8 allows the interfaces to have default and static methods

Q) What is the advantage of having Default & Static methods in java 8?

-> Default & Static methods provides backward compatibility.

-> For example, if several classes such as A, B, C and D implements an interface Test then if we add a new method to the Test, we have to change the code in all the classes (A, B, C and D) that implements this interface.



-> In this example we have only four classes that implements the interface, but imagine if there are hundreds of classes implementing an interface it will become very difficult to change all the classes which are implementing that interface. This is why in java 8, we have a new concept "default methods".

-> Default methods can be added to any existing interface and we do not need to implement these methods in the implementation classes (if required we can override them in implementation classes)

// java program on Interface – Default Method

Car.java

```
1  package in.ashokit;
2
3  interface Vehicle {
4
5      void cleanVehicle();
6
7      default void startVehicle() {
8          System.out.println("Vehicle is starting");
9      }
10 }
11
12 public class Car implements Vehicle {
13
14     @Override
15     public void cleanVehicle() {
16         System.out.println("Cleaning the vehicle");
17     }
18
19     public static void main(String args[]) {
20         Car car = new Car();
21         car.cleanVehicle();
22         car.startVehicle();
23     }
24 }
25
```

Some key points about default methods are :

- A default method must have a body.
- The access modifier of default methods are implicitly public.
- The class implementing such interface are not required to implement default methods. If needed, implementing class can override default methods.

Static Methods In Java

-> The static methods in interfaces are similar to default methods but the only difference is that you can't override them. Now, why do we need static methods in interfaces if we already have default methods?

-> Suppose you want to provide some implementation in your interface and you don't want this implementation to be overridden in the implementing class, then you can declare the method as static.

-> In the below example, we will defined a Vehicle interface with a static method called cleanVehicle().

Car.java

```
1  package in.ashokit;
2
3  interface Vehicle {
4
5      static void cleanVehicle() {
6          System.out.println("I am cleaning vehicle");
7      }
8
9      default void startVehicle() {
10         System.out.println("Vehicle starting...");
11     }
12 }
13
14 public class Car implements Vehicle {
15
16     public static void main(String args[]) {
17
18         // calling static method
19         Vehicle.cleanVehicle();
20
21         Car c = new Car();
22
23         // calling default method
24         c.startVehicle();
25
26     }
27 }
28
```

Some key points about static methods are:

- A static method must have a body.
- The access modifier of static methods is implicitly public.
- This method must be called using interface name.
- Since these methods are static, we cannot override them in implementing class.

Functional Interfaces

-> If an interface contains only one abstract method, then it is called as Functional Interface.

-> Functional Interface is used to invoke lambda expressions

-> Runnable, Callable, Comparable and ActionListener are predefined functional interfaces.

Runnable---- > run ( )

Callable---- > call ( )

ActionListener ----> actionPerformed ( )

Comparable ----> compareTo ( )

Note: We can take default and static methods also in functional interface. Only one method rule is applicable for only abstract methods.

MyInterface.java

```
1  package in.ashokit;
2
3  public interface MyInterface {
4
5      public void m1();
6
7      default void m2() {
8          // logic
9      }
10
11     public static void m3() {
12         // logic
13     }
14 }
15
```

-> To represent our interface as functional interface we will use @FunctionalInterface

```
@FunctionalInteface
public interface MyInterface {

   public void m1();
   public void m2();

}
```

// invalid bcz of 2 abstract methods

```
@FunctionalInterface
public interface MyInterface{

}
```

// invalid bcz of no abstract methods

```
@FunctionalInterface
public interface Parent {
   public void m1();
}

@FunctionalInterface
public interface Child extends Parent {

}
```

// valid bcz child inheriting from Parent

What is lambda expression

-> Java is an object-oriented language. By introducing lambdas in Java 8, the authors of Java tried to add elements of functional programming in Java.

Q- Now you might be wondering what the difference between object-oriented programming and functional programming is?

-> In object-oriented programming, objects and classes are the main entities. If we create a function then it should exist within a class. A function has no meaning outside the scope of the class object.

-> In functional programming, functions can exist outside the scope of an object. We can assign them to a reference variable and we can also pass them to other methods as a parameter.

-> A lambda expression is just an anonymous function, i.e., a function with no name and that is not bound to an identifier. We can pass it to other methods as parameters, therefore, using the power of functional programming in Java.

-> Lambda is an Anonymous function

- No Name
- No Modifier
- No Return Type

Why to use Lambda expressions?

-> To write functional programming in java

-> To write more readable, maintainable and concise code

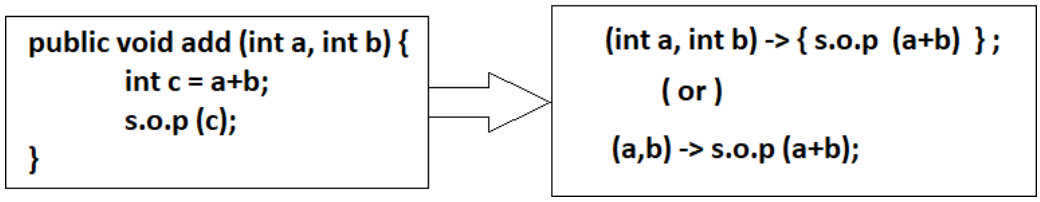
-> To enable parallel processing

Example - 1

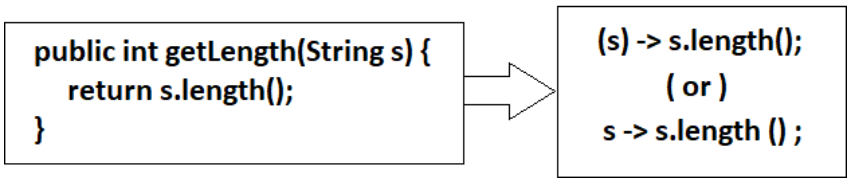
```
// java method

public void m1 ( ) {
   s.o.p("Hi");
}
```

```
( ) ->    { S.o.p ("Hi") };
        ( or )
( ) -> S.o.p ("Hi") ;
```

Example - 2



Example - 3



// Java Program with Functional Interface and Lambda Expression

Greeting.java

```
1  package in.ashokit;
2
3  @FunctionalInterface
4  interface Wish {
5      void wishMsg();
6  }
7
8  public class Greeting {
9
10     public static void main(String args[]) {
11
12         Wish wish = () -> System.out.println("Hello");
13         wish.wishMsg();
14
15     }
16
17 }
18
```

// Java Program with Functional Interface and Lambda Expression

Test.java

```
1  package in.ashokit;
2
3  @FunctionalInterface
4  interface Calculator {
5      public void add(int i, int j);
6  }
7
8  class Test {
9      public static void main(String[] args) {
10
11         Calculator c = (i, j) -> System.out.println("Sum :" + (i + j));
12
13         c.add(10, 20);
14         c.add(30, 50);
15     }
16 }
```

// Java Program with Lambda Expression to Sort ArrayList elements

Demo.java

```
1  package in.ashokit;
2
3  import java.util.ArrayList;
4  import java.util.Collections;
5
6  public class Demo {
7
8      public static void main(String[] args) {
9          ArrayList<Integer> al = new ArrayList<Integer>();
10         al.add(205);
11         al.add(102);
12         al.add(98);
13         al.add(275);
14         al.add(203);
15         System.out.println("Elements of the ArrayList before sorting : " + al);
16
17         // using lambda expression in place of comparator object
18         Collections.sort(al, (o1, o2) -> (o1 > o2) ? -1 : (o1 < o2) ? 1 : 0);
19
20         System.out.println("Elements of the ArrayList after sorting : " + al);
21     }
22
23 }
24
```

Demo.java

```
1  package in.ashokit;
2
3  public class Demo {
4
5      public static void main(String[] args) {
6
7          Runnable myThread = () -> {
8
9              // Used to set custom name to the current thread
10             Thread.currentThread().setName("myThread");
11             System.out.println(Thread.currentThread().getName() + " is running");
12         };
13
14         // Instantiating Thread class by passing Runnable
15         // reference to Thread constructor
16         Thread run = new Thread(myThread);
17
18         // Starting the thread
19         run.start();
20     }
21 }
22
```

u

Predefined Functional Interfaces

There are several predefined functional interfaces provided by java 8

1) Predicate

2) Function

3) Consumer

4) Supplier

-> These functional interfaces are provided in java.util.function package

Predicate

-> It is used to perform some conditional check and returns true or false value

Ex: Check weather no is greater than 10 or not

Note: Predicate is boolean valued function in java 8

Syntax:

```
interface Predicate<T>{
    boolean test(T k);
}
```

Demo.java

```java
package in.ashokit;

import java.util.function.Predicate;

public class Demo {

    public static void main(String[] args) {

        Predicate<Integer> p = i -> 1 > 10;

        System.out.println(p.test(100)); // true
        System.out.println(p.test(5)); // false

        String[] names = { "Kajal", "Katrina", "Karrena", "Anushka", "Mallika", "Alia" };

        Predicate<String> p1 = s -> s.charAt(0) == 'K';

        for (String name : names) {
            if (p1.test(name)) {
                System.out.println(name);
            }
        }
    }
}
```

Predicate Joining

-> To combine multiple predicates, we will use predicate joining

-> In Predicate we have below methods

test () - > abstract method

p1.negate();

p1.and(p2);

p1.or(p2);

Note: negate(), and() , or() methods are default methods in Predicate interface

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

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

Demo.java

```java
package in.ashokit;

import java.util.function.Predicate;

public class Demo {

    static boolean isPersonEligibleForMembership(Person person, Predicate<Person> predicate) {
        return predicate.test(person);
    }

    public static void main(String args[]) {
        Person person = new Person("Alex", 23);

        // Created a predicate. It returns true if age is greater than 18.
        Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
        // Created a predicate. It returns true if age is less than 60.
        Predicate<Person> lessThanSixty = (p) -> p.age < 60;

        // joining predicates
        Predicate<Person> predicate = greaterThanEighteen.and(lessThanSixty);

        boolean eligible = isPersonEligibleForMembership(person, predicate);
        System.out.println("Person is eligible for membership: " + eligible);
    }
}
```

```java
Person person = new Person("Alex", 23);

// Created a predicate. It returns true if age is greater than 18.
Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
// Created a predicate. It returns true if age is less than 60.
Predicate<Person> lessThanSixty = (p) -> p.age < 60;

// joining predicates
Predicate<Person> predicate = greaterThanEighteen.or(lessThanSixty);

boolean eligible = isPersonEligibleForMembership(person, predicate);
System.out.println("Person is eligible for membership: " + eligible);
```

Demo.java

```
1  package in.ashokit;
2
3  import java.util.function.Predicate;
4
5  public class Demo {
6
7      static boolean isPersonEligibleForMembership(Person person, Predicate<Person> predicat
8          return predicate.test(person);
9      }
10
11     public static void main(String args[]) {
12         Person person = new Person("Alex", 23);
13
14         // Created a predicate. It returns true if age is greater than 18.
15         Predicate<Person> greaterThanEighteen = (p) -> p.age > 18;
16         // Created a predicate. It returns true if age is less than 60.
17         Predicate<Person> lessThanSixty = (p) -> p.age < 60;
18
19         // joining predicates
20         Predicate<Person> predicate = greaterThanEighteen.or(lessThanSixty);
21
22         boolean eligible = isPersonEligibleForMembership(person, predicate);
23         System.out.println("Person is eligible for membership: " + eligible);
24     }
25 }
26
```

BiPredicate interface

-> The Predicate<T> takes only one parameter and returns the result. Now suppose we have a requirement where we need to send two parameters (i.e person object and min age to vote) and then return the result. Here, we can use BiPredicate<T, T>.

-> The BiPredicate<T, T> has a functional method test(Object, Object) . It takes in two parameters and returns a boolean value.

Demo.java

```
1  package in.ashokit;
2
3  import java.util.function.BiPredicate;
4
5  public class Demo {
6
7      public static void main(String args[]) {
8
9          BiPredicate<String, Integer> filter = (x, y) -> {
10             return x.length() == y;
11         };
12
13         boolean result = filter.test("ashok", 6);
14         System.out.println(result); // true
15
16         boolean result2 = filter.test("ashok", 10);
17         System.out.println(result2); // false
18     }
19 }
20
```

Supplier

-> Supplier is an interface that does not take in any argument but produces a value when the get() function is invoked. Suppliers are useful when we don't need to supply any value and obtain a result at the same time.

-> The Supplier<T> interface supplies a result of type T.

-> It is a predefined functional interface

-> It contains only one abstract method i.e get ( )

-> Supplier will only returns the value R

```
interface Supplier {
    R get();
}
```

// Java program to generate OTP using Supplier interface

RandomOTP.java

```java
package in.ashokit;

import java.util.function.Supplier;

public class RandomOTP {

    public static void main(String args[]) {
        Supplier<String> supplier = () -> {
            StringBuilder otp = new StringBuilder("");
            for (int i = 1; i <= 5; i++) {
                otp.append((int) (Math.random() * 10));
            }
            return otp.toString();
        };

        System.out.println(supplier.get());
        System.out.println(supplier.get());
        System.out.println(supplier.get());

    }
}
```

Consumer

-> It is a predefined functional interface

-> It contains one abstract method i.e accept (T t)

-> Consumer will accept values and will perform operation but it won't return any value

-> A consumer can be used in all contexts where an object needs to be consumed. taken as input, and some operation is performed on the object without returning any result.

```
interface Consumer {
  void accet(T t);
}
```

ConsumerDemo.java

```java
package in.ashokit;

import java.util.function.Consumer;

public class ConsumerDemo {

    public static void main(String[] args) {

        Consumer<String> consumer1 = (arg) -> System.out.println(arg + "My name is Ashok.");

        Consumer<String> consumer2 = (arg) -> System.out.println(arg + "I am from Ashok IT.");

        consumer1.andThen(consumer2).accept("Hello. ");
    }
}
```

BiConsumer<T,U>

This interface takes two parameters and returns nothing.

T - the type of the first argument to the operation

U - the type of the second argument to the operation.

This interface has the same methods as present in the Consumer<T> interface.

ConsumerDemo.java

```java
package in.ashokit;

import java.util.function.BiConsumer;

public class ConsumerDemo {

    public static void main(String[] args) {

        BiConsumer<String, String> greet = (s1, s2) -> System.out.println(s1 + s2);
        greet.accept("Ashok", "IT");
    }
}
```

Function

-> Function is a category of functional interfaces that takes an object of type T and returns an object of type R.

-> Until now, the functional interfaces that we've discussed have either not taken any argument (Supplier), not returned any value (Consumer), or returned only a boolean (Predicate).

-> Function interfaces are very useful as we can specify the type of input and output.

-> It is a predefined functional interface

-> It is having one abstract method that is "apply ()"

-> Function can return any type of value whereas Predicate can return only boolean value

Syntax:

```
interface Function {
   R apply(T t);
}
```

FunctionDemo.java

```
1  package in.ashokit;
2
3  import java.util.function.Function;
4
5  public class FunctionDemo {
6
7      public static void main(String[] args) {
8          // Created a function which takes string returns the length of string.
9          Function<String, Integer> lengthFunction = str -> str.length();
10         System.out.println("String length: " + lengthFunction.apply("welcome to ashokit"));
11
12         // Program to remove spaces in String Using Function
13         Function<String, String> f1 = s -> s.replaceAll(" ", "");
14         System.out.println(f1.apply("ashok it - learn here lead anywhere"));
15
16         // Program to count no.of spaces in given String
17         Function<String, Integer> f2 = s -> s.length() - s.replaceAll(" ", "").length();
18         System.out.println(f2.apply("ashok it - learn here lead anywhere"));
19     }
20 }
21
```

Function Chaining

-> To join one function with another function we will use Function Chaining

Assume that f1 & f2 are 2 functions

f1.andThen(f2) ---- > First f1 will be applied and then followed by f2

f1.compose(f2)---- > First f2 will be applied and then followed by f1

compose(Function<? super V, ? extends T> before)

Returns a composed function that first applies the function provided as a parameter on the input, and then applies the function on which it is called, to the result.

andThen(Function<? super R,? extends V> after)

This method returns a composed function that first applies the function on which it is called on the input, and then applies the function provided as parameter, to the result.

FunctionDemo.java

```
1  package in.ashokit;
2
3  import java.util.function.Function;
4
5  public class FunctionDemo {
6
7      public static void main(String[] args) {
8
9          Function<String, String> f1 = s -> s.toUpperCase();
10
11         Function<String, String> f2 = s -> s.substring(0, 5);
12
13         System.out.println(f1.apply("ashokit")); // ASHOKIT
14         System.out.println(f2.apply("ashokit")); // ashok
15
16         System.out.println(f1.andThen(f2).apply("ashokit")); // ASHOK
17         System.out.println(f1.compose(f2).apply("ashokit")); // ASHOK
18     }
19 }
20
```

BiFunction<T,U,R>

The BiFunction<T, U, R> is similar to Function<T, R> interface; the only difference is that the BiFunction interface takes in two parameters and returns an output.

In the below example, we will create a BiFunction that takes two numbers as input and returns their sum.

FunctionDemo.java

```
1  package in.ashokit;
2
3  import java.util.function.BiFunction;
4
5  public class FunctionDemo {
6
7      public static void main(String[] args) {
8
9          BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
10
11         System.out.println("Sum = " + add.apply(2, 3));
12     }
13 }
```

Method Reference :: Operator

-> Method references, as the name suggests are the references to a method. They are similar to object references. As we can have reference to an object, we can have reference to a method as well.

-> Similar to an object reference, we can now pass behaviour as parameters. But, you might be wondering what the difference between a method reference and lambda expressions is. There is no difference. Method references are shortened versions of lambda expressions that call a specific method.

Let's say you have a Consumer as defined below

```
Consumer<String> consumer = s -> System.out.println(s);
```

This can be written as below

```
Consumer<String> consumer = System.out::println;
```

Consider we have a Function<T,R> functional interface as defined below:

```
Function<Person, Integer>  function = p  -> p.getAge();
```

This can be written as:

```
Function<Person, Integer>  function = Person::getAge;
```


// Java program with static method reference

MethodRef.java

```
1  package in.ashokit;
2
3  interface MyInterface {
4      public void m1();
5  }
6
7  public class MethodRef {
8
9      public static void m2() {
10         System.out.println("m2 method");
11     }
12
13     public static void main(String[] args) {
14         MyInterface i1 = MethodRef::m2;
15         i1.m1();
16     }
17 }
```

// java program with instance method reference

Test.java

```
1  package in.ashokit;
2
3  public class Test {
4
5      public void m1() {
6          for (int i = 1; i < 5; i++) {
7              System.out.println("Child Thread");
8          }
9      }
10
11     public static void main(String... args) {
12         Test t = new Test();
13         Runnable r = t::m1;
14         Thread t1 = new Thread(r);
15         t1.start();
16
17         for (int i = 1; i < 5; i++) {
18             System.out.println("main thread");
19         }
20     }
21 }
```

// java program with Constructor Reference

Demo.java

```
1  package in.ashokit;
2
3  import java.util.function.Supplier;
4
5  public class Demo {
6
7      public static void main(String[] args) {
8
9          Supplier<Sample> i = Sample::new; // Constructor Reference
10
11         System.out.println(i.get().hashCode());
12     }
13 }
14
15 class Sample {
16     public Sample() {
17         System.out.println("Sample::Constructor");
18     }
19 }
20
```

u

Java forEach ( ) method

-> Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface.

-> It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

-> This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

```
6  class Employee {
7      int id;
8      String name;
9
10     public Employee(int id, String name) {
11         this.id = id;
12         this.name = name;
13     }
14 }
15
```

// Java program to print list data with for-each loop

```
16 public class Test {
17
18     public static void main(String... args) {
19         List<Employee> emps = new ArrayList<>();
20
21         emps.add(new Employee(101, "Raju"));
22         emps.add(new Employee(101, "Rani"));
23         emps.add(new Employee(101, "Ashok"));
24
25         for (Employee e : emps) {
26             System.out.println(e.id + "--" + e.name);
27         }
28     }
29 }
30
```

// Java program to print list data with foreach method


```
16 public class Test {
17
18     public static void main(String... args) {
19         List<Employee> emps = new ArrayList<>();
20
21         emps.add(new Employee(101, "Raju"));
22         emps.add(new Employee(101, "Rani"));
23         emps.add(new Employee(101, "Ashok"));
24
25         emps.forEach(e -> {
26             System.out.println(e.id + "--" + e.name);
27         });
28     }
29 }
```

Java 8 Optional Class

-> Every Java Programmer is familiar with NullPointerException. It can crash your code. And it is very hard to avoid it without using too many null checks. So, to overcome this, Java 8 has introduced a new class Optional in java.util package.

-> Optional class help in writing a neat code without using too many null checks.

-> By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable.

// in below java program we will get NullPointerException

OptionalDemo.java

```
1  package in.ashokit;
2
3  //Java program without Optional Class
4
5  public class OptionalDemo {
6      public static void main(String[] args) {
7          String[] words = new String[10];
8          String word = words[5].toLowerCase();
9          System.out.print(word);
10     }
11 }
12
```

Console Progress Problems

```
<terminated> OptionalDemo [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (15-Nov-2022, 11:59:33 AM)
Exception in thread "main" java.lang.NullPointerException
        at in.ashokit.OptionalDemo.main(OptionalDemo.java:8)
```

-> To avoid abnormal termination, we use the Optional class. In the following example, we are using Optional. So, our program can execute without crashing.

OptionalDemo.java

```
5  //Java program with Optional Class
6
7  public class OptionalDemo {
8
9      public static void main(String[] args) {
10
11         String[] words = new String[10];
12
13         Optional<String> checkNull = Optional.ofNullable(words[5]);
14
15         if (checkNull.isPresent()) {
16             String word = words[5].toLowerCase();
17             System.out.print(word);
18         } else
19             System.out.println("word is null");
20     }
21 }
22
```

Console Progress Problems

```
<terminated> OptionalDemo [Java Application] C:\Program Files\Java\jdk1.8.0_202\bin\javaw.exe (15-Nov-2022, 12:01:57 PM)
word is null
```

-> Optional is a container object which may or may not contain a non-null value. You must import java.util package to use this class. If a value is present, isPresent() will return true and get() will return the value.

Java 8 StringJoiner

-> In java 8, a new class StringJoiner is introduced in the java.util package.

-> Using this class we can join more than one strings with the specified delimiter, we can also provide prefix and suffix to the final string while joining multiple strings.

OptionalDemo.java

```java
package in.ashokit;

import java.util.StringJoiner;

public class OptionalDemo {

    public static void main(String[] args) {

        // Passing Hyphen(-) as delimiter
        StringJoiner mystring = new StringJoiner("-");

        // Joining multiple strings by using add() method
        mystring.add("ashok");
        mystring.add("it");
        mystring.add("java");

        // Displaying the output String
        System.out.println(mystring);
    }
}
```


OptionalDemo.java

```java
package in.ashokit;

import java.util.StringJoiner;

public class OptionalDemo {

    public static void main(String[] args) {

        // Passing Hyphen(-) as delimiter with prefix and suffix
        StringJoiner mystring = new StringJoiner("-", "(", ")");

        // Joining multiple strings by using add() method
        mystring.add("ashok");
        mystring.add("it");
        mystring.add("java");

        // Displaying the output String
        System.out.println(mystring); // (ashok-it-java)
    }
}
```

Stream API

-> Stream API introduced in java 8

-> Collections are used to Store the data

-> Stream is used to process the data

Note: Collections & Streams both are not same.

-> The addition of stream api was one of the major features added to java8.

-> A stream in java can be defined as a sequence of elements from a source that supports aggregate operations on them.

-> The source here refers to collection or array that provide data to stream

Few Important points about Streams

1) Stream is not a data structure. It is a bunch of operations applied to a source. The source can be a collection, array or i/o channels.

2) Streams don't change the original data structure

3) There can be zero or more intermediate operations that transforms a stream into another stream. Each intermediate operation is lazily executed

5) Terminal operations produce the result of the stream.

Stream Creation

-> In java 8 we can create stream in 2 ways

1) Stream.of(v1, v2, v3...)

2) stream() method

Demo.java

```
1  package in.ashokit;
2
3  import java.util.ArrayList;
4  import java.util.List;
5  import java.util.stream.Stream;
6
7  public class Demo {
8      public static void main(String[] args) {
9
10         // Approach - 1
11         Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5, 6);
12
13         List<String> list = new ArrayList<>();
14         list.add("Sachin");
15         list.add("Sehwag");
16         list.add("Dhoni");
17         list.add("Kohli");
18         list.add("Pandya");
19
20         // Approach - 2
21         Stream<String> stream2 = list.stream();
22     }
23 }
24
```

-> Stream API provided several methods to perform operations.

-> The methods which are provided by stream API can be categorized into 2 types

1) Intermediate Operational Methods

2) Terminal Operational Methods

-> Intermediate operational methods will not produce any results. They usually accept functional interfaces as parameter and always returns new Stream.

Ex: filter() and map() etc...

-> Terminal Operational methods will take input and produce results.

Ex: count(), toArray() and collect()

Streams with Filtering

-> Stream interface having filter( ) method.

-> filter( ) method will take Predicate as input

-> Predicate is a functional interface which will take input and returns boolean value

-> Predicate interface contains test( ) as abstract method. This method is used to execute lambda.

// Java program to filter the numbers which are > =20

Demo.java

```
1  package in.ashokit;
2
3  import java.util.stream.Stream;
4
5  public class Demo {
6      public static void main(String[] args) {
7
8          Stream<Integer> stream = Stream.of(4, 8, 12, 6, 7, 11, 24);
9
10         // filter the numbers which are >= 6
11
12         /*
13          Stream<Integer> filteredStream = stream.filter(i -> i >= 6);
14         filteredStream.forEach(i -> System.out.println(i));
15
16         */
17
18         // filter the numbers which are >=20
19         stream.filter(i -> i >= 20).forEach(System.out::println);
20     }
21 }
22
```

// Java program to filter the names which are starting with ”A”

Demo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

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

        List<String> list = new ArrayList<>();

        list.add("Anushka");
        list.add("TriSha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

        Stream<String> stream = list.stream();
        stream.filter(name -> name.startsWith("A")).forEach(System.out::println);
    }
}

```


// Java Program To Filter Person Objects

```java
class Person {

    private String name;
    private Integer age;
    private String job;

    public Person(String name, Integer age, String job) {
        this.name = name;
        this.age = age;
        this.job = job;
    }

    // setters, getters & toString()
}

```

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

        Person p1 = new Person("Raju", 28, "Software");
        Person p2 = new Person("Mahesh", 29, "Driver");
        Person p3 = new Person("Ashok", 30, "Teacher");
        Person p4 = new Person("Sunil", 27, "Mechanic");
        Person p5 = new Person("Bharat", 30, "Chef");

        List<Person> persons = Arrays.asList(p1, p2, p3, p4, p5);

                    persons.stream()
                           .filter(p -> p.getAge() > 21 && p.getAge() < 30 && p.getJob().equals("Software"))
                           .forEach(System.out::println);
    }
}
```

Mapping Operations In Streams

-> Mapping operations are belonging to Intermediate operations

-> Mapping operations are those operations that transform the elements of a stream and return a new stream with transformed elements.

MappingDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class MappingDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        list.add("Anushka");
        list.add("Trisha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

        Stream<String> stream = list.stream();

        stream.map(name -> name.toUpperCase()).forEach(System.out::println);
    }
}
```

MappingDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.IntStream;
import java.util.stream.Stream;

public class MappingDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        list.add("Anushka");
        list.add("Trisha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

        Stream<String> stream = list.stream();

        //Stream<String> tfStream = stream.map(name -> name.toUpperCase() + "::" + name.length());
        IntStream tfStream = stream.mapToInt(name -> name.length());
        tfStream.forEach(System.out::println);
    }
}
```

MappingDemo.java

```java
package in.ashokit;

import java.util.ArrayList;
import java.util.List;
import java.util.stream.Stream;

public class MappingDemo {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();

        list.add("Anushka");
        list.add("Trisha");
        list.add("Nayanatara");
        list.add("Deepika Padukone");
        list.add("Pooja Hegde");
        list.add("Anupama Parameswaran");
        list.add("Amisha Patel");

        Stream<String> stream = list.stream();

        //print heroine name with length whose name is starting with A
        stream.filter(name -> name.startsWith("A"))
              .map(name -> name +"::"+name.length())
              .forEach(System.out::println);
    }
}
```

```
Person p1 = new Person("Raju", "Software", 6301921083l);
Person p2 = new Person("Gopi", "Chef", 7897678991);
Person p3 = new Person("Mahesh", "PhotoGrapher", 668621083l);
Person p4 = new Person("Sunil", "Driver", 475757210831);
Person p5 = new Person("David", "Teacher", 56789210831);
Person p6 = new Person("Ashok", "Software", 6301921781);

//Display person name with phn number who is doing Software job

List<Person> persons = Arrays.asList(p1, p2, p3, p4, p5, p6);

persons.stream()
         .filter(person -> person.getJob().equals("Software"))
         .map(person -> person.getName()+"::"+person.getPhno())
         .forEach(System.out::println);
```

-> When we have Collection inside another collection then to flaten that stream we will use flatMap( ) method.

MappingDemo.java

```
1  package in.ashokit;
2
3  import java.util.Arrays;
4  import java.util.List;
5  import java.util.stream.Stream;
6
7  public class MappingDemo {
8      public static void main(String[] args) {
9          List<String> javacourses = Arrays.asList("Core Java", "Adv Java", "SBMS", "JRTP");
10
11         List<String> uicourses = Arrays.asList("HTML5", "CSS3", "Angular", "React JS");
12
13         List<String> cloudcourses = Arrays.asList("DevOps", "AWS", "Azure", "GCP");
14
15         List<List<String>> ashokitcourses = Arrays.asList(javacourses, uicourses, cloudcourses);
16
17         Stream<List<String>> stream1 = ashokitcourses.stream();
18
19         Stream<String> courses = stream1.flatMap(s -> s.stream());
20
21         courses.forEach(System.out::println);
22     }
23 }
24
```

Slicing Operations In Streams

-> distinct ( ) -> it is used to get unique elements from the Stream

-> limit (long maxSize) -> it is used to get number of elements from the stream based on given size

-> skip (long n) -> it is used to skip elements from starting to given length and returns remaining elements.

Note: The above 3 methods are intermediate methods only. They perform operation and returns new stream.

```
List<String> countries = new ArrayList<>();
countries.add("India");
countries.add("USA");
countries.add("UK");
countries.add("China");
countries.add("India");
countries.add("USA");


//Getting unique values from collection using distinct()
/*countries.stream()
         .distinct()
         .forEach(System.out::println);*/


//Getting specific no.of records from collection using limit()
/*countries.stream()
         .limit(4)
         .forEach(System.out::println);*/

//removing first N no.of recording
countries.stream()
         .skip(3)
         .forEach(System.out::println);
```


Matching Operations in the streams

-> Matching operations are terminal operations that are used to check if elements with certain are present in the stream or not.

-> There are mainley three matching functions available in the Stream. these are

- anyMatch ( )
- allMatch( )
- noneMatch( )

```
6  public class Demo {
7
8      public static void main(String[] args) {
9
10         List<Person> list = new ArrayList<>();
11         list.add(new Person("David", 23, "India"));
12         list.add(new Person("Joy", 25, "USA"));
13         list.add(new Person("Ryan", 50, "Canada"));
14         list.add(new Person("Ram", 45, "India"));
15         list.add(new Person("Ching", 56, "China"));
16
17         /*boolean isIndiansAvailable = list.stream().anyMatch(p -> p.getCountry().equals("India"));
18         System.out.println("Is Indians Available in Collection :: "+ isIndiansAvailable);*/
19
20         /*boolean allMatch = list.stream().allMatch(p -> p.getCountry().equals("India"));
21         System.out.println("All persons are indians or not :: "+ allMatch);*/
22
23         boolean noneMatch = list.stream().noneMatch(p -> p.getCountry().equals("Germany"));
24         System.out.println("No Germans are available :: "+ noneMatch);
25
26     }
27 }
28
```

Finding Operations In Streams

-> In Matching operations we can check weather data present in the stream or not based on given criteria. After checking the condition it returns true or false value.

-> By using Finding Operations we can check the condition and we can get the data based on condition.

- findFirst()
- findAny( )

```
public class Demo {

    public static void main(String[] args) {

        List<Person> list = new ArrayList<>();
        list.add(new Person("David", 23, "India"));
        list.add(new Person("Joy", 25, "USA"));
        list.add(new Person("Ryan", 50, "Canada"));
        list.add(new Person("Ram", 45, "India"));
        list.add(new Person("Ching", 56, "China"));

        /*Optional<Person> findFirst = list.stream()
                                        .filter(p -> p.getCountry().equals("India"))
                                        .findFirst();*/

        Optional<Person> findAny = list.stream()
                                        .filter(p -> p.getCountry().equals("India"))
                                        .findAny();

        if(findAny.isPresent()) {
            System.out.println(findAny.get());
        }

    }
}
```

Collectors In Streams

-> Collectors operations are used to collect from Streams

-> We are having below methods to perform Collectors operations

Collectors.toList()

Collectors.toSet()

Collectors.toMap()

Collectors.toCollection() etc..

```
class Employee {

    String name;
    int age;
    int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }
    //setters & getters
}
```

// java program on Collectors with toList( )

```
public class Demo {

    public static void printEmpNames(List<String> empNames) {
        System.out.println(empNames);
    }

    public static void main(String[] args) {

        List<Employee> list = new ArrayList<>();

        list.add(new Employee("Ram", 23, 20000));
        list.add(new Employee("Ashok", 25, 30000));
        list.add(new Employee("Suresh", 33, 25000));
        list.add(new Employee("Charan", 26, 40000));

        List<String> empNames = list.stream()
                                    .map(e -> e.getName())
                                    .collect(Collectors.toList());

        printEmpNames(empNames);

    }
}
```

// java program on Collectors with toMap( )

```
public class Demo {

    public static void main(String[] args) {

        List<String> list = new ArrayList<>();
        list.add("Rahul");
        list.add("Sachin");
        list.add("Hardik");
        list.add("Dhoni");

        Map<String, Integer> namesMap = list.stream()
                                            .collect(Collectors.toMap(s -> s, s -> s.length()));

        System.out.println(namesMap);

    }
}
```

Stream API Interview Questions

1) Getting Min salary emp from collection

2) Getting Max salary emp from Collection

3) Getting Avg Salary of the employee

4) Group By

Employee.java

```java
package in.ashokit;

public class Employee {

    String name;
    int age;
    int salary;

    public Employee(String name, int age, int salary) {
        this.name = name;
        this.age = age;
        this.salary = salary;
    }

    // setters & getters
    // toString() method

}
```


Employee.java Demo.java

```java
import java.util.List;
import java.util.Optional;
import java.util.stream.Collectors;

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

        List<Employee> list = new ArrayList<>();

        list.add(new Employee("Ram", 23, 20000));
        list.add(new Employee("Ashok", 25, 60000));
        list.add(new Employee("Suresh", 33, 25000));
        list.add(new Employee("Charan", 26, 40000));

        Double avgSalary = list.stream()
                .collect(Collectors.averagingInt(emp -> emp.getSalary()));
        System.out.println("Emp Avg Salary :: " + avgSalary);

        Optional<Employee> minEmpSalary = list.stream()
                .collect(Collectors.minBy(Comparator.comparing(Employee::getSalary)));
        System.out.println("Minimum Salary Emp:: " + minEmpSalary.get());

        Optional<Employee> maxEmpSalary = list.stream()
                .collect(Collectors.maxBy(Comparator.comparing(Employee::getSalary)));
        System.out.println("Maximum Salary Emp:: " + maxEmpSalary.get());

    }
}
```

Group By Operation

User.java

```java
package in.ashokit;

public class User {

    String name;
    int salary;
    String country;

    public User(String name, int salary, String country) {
        this.name = name;
        this.salary = salary;
        this.country = country;
    }

    //setters & getters methods
    // toString ( ) method

}
```

Demo.java

```java
package in.ashokit;

import java.util.ArrayList;

public class Demo {
    public static void main(String[] args) {
        List<User> users = new ArrayList<>();

        users.add(new User("Ram", 10000, "India"));
        users.add(new User("Anil", 20000, "Canada"));
        users.add(new User("Sunil", 30000, "India"));
        users.add(new User("Orlen", 40000, "Japan"));
        users.add(new User("Cathie", 50000, "UK"));
        users.add(new User("Ching Chong", 10000, "China"));

        Map<String, List<User>> collect = users.stream()
                .collect(Collectors.groupingBy(User::getCountry));

        System.out.println(collect);

    }
}
```

Parallel Streams

-> Streams is one of the major change added in java 1.8 version

-> Generally Streams will execute in Sequential manner

-> We can use parallel streams also to execute program faster by utilizing system resources efficiently.

-> Parallel Streams introduced to improve performance of the program.

ParallelStreamDemo.java

```
1  package in.ashokit;
2
3  import java.util.stream.Stream;
4
5  public class ParallelStreamDemo {
6
7      public static void main(String[] args) {
8          System.out.println("*** Serial Stream **");
9          Stream<Integer> stream1 = Stream.of(1, 2, 3, 4, 5);
10         stream1.forEach(num -> System.out.println(num + "::" + Thread.currentThread().getName()));
11
12         System.out.println("** Parallel Stream **");
13         Stream<Integer> stream2 = Stream.of(1, 2, 3, 4, 5);
14         stream2.parallel().forEach(num -> System.out.println(num + "::" + Thread.currentThread().getName()));
15     }
16 }
17
```

Java Spliterator

Like Iterator and ListIterator, Spliterator is a Java Iterator, which is used to iterate elements one-by-one from a List implemented object. Some important points about Java Spliterator are:

- Java Spliterator is an interface in Java Collection API.
- Spliterator is introduced in Java 8 release in java.util package.
- It supports Parallel Programming functionality.
- We can use it for both Collection API and Stream API classes.
- It provides characteristics about Collection or API objects.
- We can NOT use this Iterator for Map implemented classes.
- It uses tryAdvance() method to iterate elements individually in multiple Threads to support Parallel Processing.
- It uses forEachRemaining() method to iterate elements sequentially in a single Thread.
- It uses trySplit() method to divide itself into Sub-Spliterators to support Parallel Processing.
- Spliterator supports both Sequential and Parallel processing of data.

Demo.java

```java
package in.ashokit;

import java.util.ArrayList;

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

        List<String> names = new ArrayList<>();
        names.add("ashok");
        names.add("it");
        names.add("java");

        // Getting Spliterator
        Spliterator<String> namesSpliterator = names.spliterator();

        // Traversing elements
        namesSpliterator.forEachRemaining(System.out::println);

    }
}
```

Date API changes in java 1.8 version

-> In java we have java.util.Date class to work with date related functionality in our application.

Date date = new Date();

-> When we create object for Date class it will give both date and time (current date & time).

-> If we want to get only date without time or only time without date then we have to write our own logic using Date class object.

-> To overcome this problem, in java 1.8 v changes happened for Date API.

-> In Java 1.8v new classes provided to work with date related functionality

java.time.LocalDate (It will deal with only date)

java.time.LocalTime (It will deal with only time)

java.time.LocalDateTime (It will deal with both date & time)

-> The above classes got introduced in java.time package.

// java program LocalDate class

Demo.java

```java
package in.ashokit;

import java.time.LocalDate;
import java.time.Month;

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

        // Getting Current Date
        LocalDate now = LocalDate.now();
        System.out.println(now);

        // Getting specific date using of method
        LocalDate date = LocalDate.of(2022, 05, 20);
        System.out.println(date);

        date = LocalDate.of(2022, Month.MAY, 20);
        System.out.println(date);

        // Converting String to Date using Parse
        date = LocalDate.parse("2015-02-26");
        System.out.println(date);

        // Adding 4 days to given date
        date = date.plusDays(4);
        System.out.println(date);
    }
}
```

Demo.java

```java
package in.ashokit;

import java.time.LocalDate;

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

        // Getting specific date using of method
        LocalDate date = LocalDate.of(2022, 05, 20);
        System.out.println(date);

        // Adding 4 months to given date
        date = date.plusMonths(4);
        System.out.println(date);

        // Check date is before given date
        boolean isBefore = LocalDate.parse("2020-03-12").isBefore(LocalDate.parse("2018-06-14"));
        System.out.println(isBefore);

        // Check date is after given date
        boolean isAfter = LocalDate.parse("2020-03-12").isAfter(LocalDate.parse("2018-06-14"));
        System.out.println(isAfter);
    }
}
```

// java program on LocalTime class

Demo.java

```java
package in.ashokit;

import java.time.LocalTime;

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

        // Getting current Time
        LocalTime time = LocalTime.now();
        System.out.println(time);

        // Getting Specific Time using of method
        time = LocalTime.of(11, 20, 03);
        System.out.println(time);

        // Convert String value to Date using parse method
        time = LocalTime.parse("08:30:20");
        System.out.println(time);

        // Adding 4 seconds to given time
        time = time.plusSeconds(4);
        System.out.println(time);

        // Adding minutes to given time
        time = time.plusMinutes(10);
        System.out.println(time);

        // Adding hour to given time
        time = time.plusHours(2);
        System.out.println(time);
    }
}

```

-> Period class is used to check difference between 2 dates

// java program on Period class

Demo.java

```
1  package in.ashokit;
2
3  import java.time.LocalDate;
4  import java.time.Period;
5
6  public class Demo {
7      public static void main(String[] args) {
8
9          Period period = Period.ofDays(5);
10         System.out.println(period.getDays());
11
12         period = Period.ofMonths(3);
13         System.out.println(period.getMonths());
14
15         period = Period.ofYears(2);
16         System.out.println(period.getYears());
17
18         // Find Difference between 2 dates
19         Period p = Period.between(LocalDate.parse("1991-05-20"), LocalDate.now());
20         System.out.println(p);
21     }
22 }
23
```

-> Duration class is used to check difference between 2 times.

// java program on Duration class

Demo.java

```
1  package in.ashokit;
2
3  import java.time.Duration;
4  import java.time.LocalTime;
5
6  public class Demo {
7      public static void main(String[] args) {
8
9          Duration between = Duration.between(LocalTime.parse("12:14"), LocalTime.parse("13:15"));
10         System.out.println(between);
11     }
12 }
13
```

Java – 8 Knowledge – Check

1) What are the new features added in Java 8?
2) What is Default method and why we need it?
3) What is static method and why we need it?
4) Default methods vs Static Methods in interfaces?
5) What is Functional Interface?
6) What is Lambda expression?
7) How to write Lambda expression?
8) How to invoke Lambda expressions?
9) What are predefined Functional Interfaces available in java 8?
10) What is Predicate?
11) What is Supplier?
12) What is Consumer?
13) What is Function?
14) What is Optional class and why we need it?
15) What is Stream API?
16) How to create Stream object in java?
17) How to filter data using Stream API?
18) What is map ( ) operation in java 8?
19) What is flatMap ( ) in java 8 ?
20) What is the difference between intermediate and terminal operations in Stream?
21) Write java a program to get names of the employees whose salary is greater than 1 lakh.
22) What are date changes implemented in Java 8?
23) What is forEach ( ) method in java ?
24) What are the new changes introduced in java 8 w.r.t Collection Framework?
25) What is Spllterator?

====== Lean Here.. Lead Anywhere...!! =======