

**CLUSTER**

Software Solutions



# **"CORE JAVA"**

---

**The Complete OCPJP (SCJP) Guide**

**-By Ravikumar Hadagali**

## Table of Contents

|   |    |
|---|----|
| Environment Setting.....                                  | 5  |
| Path:.....  | 5  |
| Classpath.....  | 5  |
| Setting Path and Class Path.....                          | 5  |
| Java Language .....                                       | 6  |
| 1. Character Set: .....                                   | 6  |
| 2. Keywords:.....   | 6  |
| 3. Identifiers: .....                                     | 7  |
| 4. Variables:.....  | 7  |
| 5. Data Types: .....                                      | 7  |
| 6. Literals:.....   | 7  |
| 7. Operators .....  | 9  |
| 8. Control Statement: .....                               | 11 |
| 9. Separators: .....                                      | 15 |
| OOPS in Java.....   | 16 |
| Varargs.....  | 18 |
| Constructor .....   | 19 |
| finalize() Method .....                                   | 20 |
| Constructor and finalize() difference.....                | 20 |
| static in Java .....                                      | 22 |
| Difference between instance block and static block: ..... | 23 |
| this keyword.....   | 26 |
| Inheritance.....  | 27 |
| super keyword .....                                       | 29 |
| Method Overriding.....                                    | 30 |
| Dynamic Polymorphism .....                                | 31 |
| instanceof Operator .....                                 | 32 |
| final keyword .....                                       | 33 |
| Abstract class .....                                      | 34 |

|  |     |
|--|-----|
| interface.....   | 35  |
| User Defined Data Types in Java .....                      | 39  |
| Packages .....   | 41  |
| import .....   | 42  |
| Access Specifiers .....                                    | 43  |
| Inner classes.....   | 45  |
| main() method in Java: .....                               | 47  |
| Exception Handling.....                                    | 49  |
| Exception and Exception Handling in Java.....              | 51  |
| Classification of exceptions.....                          | 54  |
| java.lang package .....                                    | 56  |
| class Object .....   | 56  |
| class String .....   | 58  |
| class StringBuffer .....                                   | 59  |
| class StringBuilder .....                                  | 59  |
| class System .....   | 61  |
| class Runtime .....  | 61  |
| Memory management and garbage collector in Java .....      | 62  |
| Wrapper classes .....                                      | 64  |
| Exploring Multithreading.....                              | 69  |
| Multitasking .....   | 69  |
| Multiprocessing:.....                                      | 70  |
| Multithreading:.....                                       | 71  |
| java.io package .....                                      | 88  |
| Exploring java.util package (library) .....                | 91  |
| Collection Framework.....                                  | 91  |
| History of Java collections:- .....                        | 93  |
| Differences and Similarities: .....                        | 102 |
| Accessing the elements of a collection or traversing ..... | 106 |
| Working with arrays using class Arrays .....               | 107 |
| Sorting in List.....                                       | 108 |

|  |     |
|--|-----|
| class Collections.....                                     | 109 |
| New features added in java 5 to support Collections.....   | 110 |
| New API addition in java 6 to support collections.....     | 112 |
| Generics (Java 1.5).....                                   | 114 |
| Bounded type parameters. ....                              | 115 |
| Wild card arguments .....                                  | 115 |
| Exploring more classes in java.util package.....           | 117 |
| StringTokenizer.....                                       | 117 |
| Internationalization (i18n) .....                          | 117 |
| class Locale.....  | 117 |
| class ResourceBundle .....                                 | 118 |
| Date .....   | 118 |
| TimeZone .....   | 118 |
| Calendar and GregorianCalendar .....                       | 119 |
| Scanner .....  | 119 |
| Formatter .....  | 120 |
| Exploring java.text package .....                          | 121 |
| class NumberFormat.....                                    | 121 |
| DateFormat .....   | 122 |
| SimpleDateFormat.....                                      | 123 |
| Exploring java.util.regex package.....                     | 124 |
| Regular Expressions:-.....                                 | 124 |
| class Class and reflection in java .....                   | 126 |
| interface Cloneable and cloning.....                       | 128 |
| Marker interface or Tag interface:- .....                  | 128 |
| interface Serializable and serialization .....             | 129 |
| interface Externalizable & externalization.....            | 131 |
| Difference between Serialization and Externalization ..... | 133 |
| J2SE 5 (java 5.0 or Tiger) .....                           | 134 |
| Java 5 new features:- .....                                | 134 |
| 1. static import:- .....                                   | 134 |

|                              |     |
|------------------------------|-----|
| 2. Generics:- .....          | 134 |
| 3. Autoboxing:- .....        | 135 |
| 4. Enhanced for loop:- ..... | 135 |
| 5. Varargs:- .....           | 135 |
| 6. Enumerations:- .....      | 135 |

## **Environment Setting**

### ***Path:***

- Is a variable name used by windows Operating System to locate the binary files or exe files.
- When we set the PATH for binary files or executable files then those files will be accessible in all the folders and drives of the Operating System.
- It is case insensitive.

### ***Classpath***

- Is a variable name used by JVM/JRE to locate the java libraries (.class files)
- When we set the CLASSPATH for libraries (.class files) then those libraries will be accessible in all the folders and drives of the Operating System.
- It is case insensitive.

### ***Setting Path and Class Path***

- We can set PATH and CLASSPATH in 2 ways:-
  - a) DOS prompt:  
Drawback:  
Applicable to a single window and you have to set every time.
  - b) Environment Variables:  
Advantage:  
It is applicable to all users and in every window.

## Java Language

- 1) Character Set.
- 2) Keywords.
- 3) Identifiers.
- 4) Variables.
- 5) Data Types
- 6) Literals
- 7) Operators
- 8) Control Statement.
- 9) Separators

### **1. Character Set:**

It is a list of characters which developer can use for writing java code. Java character set allows the following 3 types.

- a) Digits:- ( 0-9 )
- b) Alphabets (A-Z, a-z)
- c) Special symbols (+, -, ...etc)

### **2. Keywords:**

These are set of words with some predefined meaning.

| Primitive Data Types   | Userdefined Data Types   | Access Specifiers   | Exception Handling  |
|--|--|---|---|
| 1) byte<br>2) short<br>3) int<br>4) long<br>5) float<br>6) double<br>7) char<br>8) boolean   | 1) class<br>2) interface<br>3) enum  | 1) private<br>2) protected<br>3) public   | 1) try<br>2) catch<br>3) finally<br>4) throw<br>5) throws |
| Control Statements   | Modifiers  | Miscellaneous   | Data literal values (but not keywords)                    |
| 1) if<br>2) else<br>3) for<br>4) do<br>5) while<br>6) switch<br>7) case<br>8) default<br>9) break<br>10) continue<br>11) return<br>12) goto (reserved) | 1) static<br>2) final<br>3) abstract<br>4) volatile<br>5) synchronized<br>6) transient<br>7) native<br>8) strictfp | 1) new<br>2) void<br>3) extends<br>4) implements<br>5) this<br>6) super<br>7) instanceof<br>8) const (reserved)<br>9) assert<br>10) import<br>11) package | 1) null<br>2) true<br>3) false                            |

### 3. Identifiers:

- It is name given to a variable, method, class, interface package etc.
- It should contain alphabets, digits and only 2 special symbols \_ (underscore), \$(dollar).
- 1<sup>st</sup> character should be an alphabet, \_ or \$ and it cannot be a digit.
- It cannot have a space.
- It cannot be a keyword.

Examples:-

Valid identifiers:

- ☞ AVGTEMP
- ☞ abc123
- ☞ \$test
- ☞ if9
- ☞ \_123

Invalid identifiers

- ☞ 123abc
- ☞ if
- ☞ &abc
- ☞ abc 123
- ☞ student name

### 4. Variables:

- It is a named memory location that may be assigned a value.
- Value in a variable can be changed or modified. Syntax:

```
datatype variablename = value;
```

Example:

```
int a = 10;  
double d = 10.5;
```

### 5. Data Types:

Java has three types of data types

- a) Primitive(byte, short, int, long, float, double, char and boolean)
- b) Derived Types(arrays)
- c) User Defined Types(class, subclass, abstract class, interface, enumerations and annotations)

### 6. Literals:

- It is a value.
- Literal value should be stored in corresponding data type.
- It can be one of the following 5 types.
  - a) Integer literal.
  - b) Floating point literal



- c) Boolean literal.
- d) Character literal.
- e) String literal.

**a) Integer literal:**

It is a collection of digits without decimal point. There are 3 types of integer literals.

☞ Decimal Integer Literal: - is a collection of digits from 0 to 9. It uses base 10.

Eg: 125, 756.

☞ Octal Integer Literal: - is a collection of digits from 0 to 7 and must start with 0. It uses base 8.

Eg: 0765;

☞ Hexadecimal Integer Literal: is a collection of digits from 0 - 9 and A - F or a - f.

It must start with 0X. 0X can be small or capital. It uses base 16.

Eg: 0X523aed

**b) Floating point literal:**

It is a collection of digits with decimal point. There are two ways to represent floating point literals.

☞ Standard notation.

Eg; 750.987

☞ Scientific or exponential notation

7.50987e 2

**c) Boolean literal:**

There are only two types of boolean literals which are true and false.

**d) Character literal:**

It is a single character which is enclosed between single quotation marks. Eg: 'A', '9'.

**e) String literal:**

It consists of set of characters which is enclosed between double quotation marks. Eg: "hello", "cluster", "USN123esw"

When we are writing String literals we might need the following escape sequences.

| Escape sequence | Description     |
|-----------------|-----------------|
| \'              | Single quote    |
| \"              | Double quote    |
| \\              | Back slash      |
| \t              | Tab             |
| \n              | New line        |
| \b              | Backspace       |
| \r              | Carriage return |
| \f              | Form feed       |

## 7. Operators

Java has the following list of operators.

- a) Arithmetic operators(+, -, \*, /, %)
- b) Assignment operator(=, +=, -=, \*=, /=, %=)
- c) Relational operator (>, <, >=, <=, ==, !=)
- d) Logical operator (&&, ||, !)
- e) Increment / Decrement operator (++ , --)
- f) Ternary operator (?:)
- g) new
- h) instanceof
- i) Bitwise operator (>>, <<, &, |, ~, ^)

### a) Arithmetic operators:

- Arithmetic operators are used for doing mathematical operations.
- When we combine arithmetic operator with two or more operands then it is called arithmetic expression.
- The result of an arithmetic expression is integer value or floating value.
- All arithmetic operators are binary operators except “+” and “-”. Both can be used as a unary operator. When we use “-” with any operand it negates the values ( changes it to negative value)
- “+” is the only overloaded operator among all the operators in java. When it is used with numeric data types then it performs addition and when used with String it performs concatenation. The JVM detects the data types and performs the addition or concatenation operation implicitly.

**Note:** operator overloading is not possible in java. “+” is the only overloaded operator done by java language implicitly.

### b) Assignment operator:

- Assignment operators are used to assign value from right side operand to left side operand.
- All assignment operators are binary operators.
- When you are using assignment operator you should use the same type of operand or both the operands should be of same data type.
- If at all you have a requirement to assign one type of value to another, then you have to do type casting provided both the data types are compatible. Casting might be implicit or explicit.

### b) Relational operator:

- All relational operators are binary operators.
- Relational operators form relational expression whose result is a boolean value.

### c) Logical operator:

- In this “&&” and “||” are binary operators but “!” is unary operator.

- Logical operators form logical expression whose result is a boolean value.
- Operands of logical operator must be of boolean type

| A | B | A&&B | A  B | !A |
|---|---|------|------|----|
| T | T | T    | T    | F  |
| T | F | F    | T    | F  |
| F | T | F    | T    | T  |
| F | F | F    | F    | T  |

d) **Increment / Decrement operator:**

- Both are unary operators.
- “++” is an increment operator which increments the operand by 1 value.
- “--” is a decrement operator which decrements the operand by 1 value.
- A prefix operator (when used before the operand) first adds or subtracts 1 to the operand and then the result is assigned to the variable on left.
- A suffix operator (when used after the operand) first assigns the value to the variable on left and then adds or subtracts 1 from the operand.
- Both are used extensively in for and while loops.

e) **Ternary operator:**

- It is used to perform simple conditional check.

Syntax:

```
Var = (condition) ? exp1 : exp2;
Max = (a>b) ? a : b;
T F
```

- First condition will be evaluated and if the condition is true then the true block value will be assigned to variable. If the condition is false, then false block value will be assigned to the variable.

c) **new:**

- It is use for creating instances or objects of classes.

d) **instanceof:**

- It is used to check a given object is of a particular class type. Instanceof operator can obtain run time information about an object.

e) **Bitwise operator:**

- They are used for manipulating the bit values.
- They work upon the individual bits of their operands.
  - ☞ << (Left shift operator).
  - ☞ >> (Right shift operator).
  - ☞ & (Bitwise AND operator).
  - ☞ | (Bitwise OR operator).

☞ ~ (Bitwise NOT operator).

☞ ^ (Bitwise exclusive OR operator).

| A | B | A&B | A B | A^B | ~A |
|---|---|-----|-----|-----|----|
| 1 | 1 | 1   | 1   | 0   | 0  |
| 1 | 0 | 0   | 1   | 1   | 0  |
| 0 | 1 | 0   | 1   | 1   | 1  |
| 0 | 0 | 0   | 0   | 0   | 1  |

## 8. Control Statement:

- In a java program all statements are executed sequentially. But in some situations we may want to change the execution of statements based on certain conditions or repeat a group of statements until certain specified condition is met.
- Java gives rich support for writing such control statements.
- Java languages give rich support for writing such selection/conditional, iteration/looping or jump/branching statements.
- Selection statements allow choosing different path of execution based upon the outcome of an expression.
- Iteration statements enable to repeat set of statements until a specified condition is met.
- Jump statements allow transferring control to another part of the program.
- Following are the control statements.

1. if / else
2. switch / case
3. for
4. while and do / while
5. break
6. continue
7. return
8. goto (reserved)

### 1. if / else:

“if statement” is used to perform conditional checks.

a.

```
if(condition) {  
    statement 1  
}  
statement X;
```

In the 1st scenario first condition will be checked. If the condition is true then statement 1 will be executed and then control will be transferred to statement X. If the condition is false then control will be transferred to statement X without executing statement 1.

b.

```
if(condition) {  
    statement 1  
} else {  
    statement 2  
}  
  
statement x;
```

In the 2<sup>nd</sup> scenario if the condition is true then statement 1 will be executed and then control will be transferred to statement X. If the condition is false then statement 2 will be executed and control will be transferred to statement X. Here either if or else condition will execute and both of them will never execute together.

c.

```
if(condition) {  
    statement 1  
} else if (condition) {  
    statement 2  
} else {  
    statement 3;  
}  
  
statement x;
```

In the 3<sup>rd</sup> scenario if the condition is true then statement 1 will be executed and then control will be transferred to statement X. If the condition is false it will come to else if part and checks the condition. If true then statement 2 will be executed and control will be transferred to statement X. If false then statement 3 will be executed and control will be transferred to statement X.

## 2. switch / case:

- Switch statement is used to execute a particular set of statements among multiple conditions.
- It is an alternative to complicated if else if ladder conditions.
- The expression type in the switch must be any of the following data type.
  - 1) byte

- 2) Short
- 3) int
- 4) char
- 5) enum (java 1.5)
- 6) String (java 1.7)

- The expression type in the switch must not be of the following data type.
  - 1) long
  - 2) float
  - 3) double
  - 4) Boolean
- break is optional. Every case must end with break statement which will help to terminate and transfer the control outside of switch block. If no break is used then execution will continue to next case.
- default is optional. If present it will execute only if the value of the expression does not match with any of the case and then control goes to statement X. If not present then control will exit switch statement and goes to statement X.

### 3. for:

- for statement is used to execute a set of statements multiple times.

```
for (initialization ; condition ; increment / decrement) {  
    Statement 1;  
}
```

When for statement is encountered 1<sup>st</sup> initialization statement will be executed and then condition will be verified. If condition is true then statements inside for block will be executed and then increment or decrement operation will be executed. After increment or decrement operation again condition will be verified. If the condition is true then again for block will be executed and same process will be repeated till condition becomes false. If the condition is false then control will come out of the loop and executes statement X.

### 4. while and do / while:

- while is an entry controlled loop. In while statement 1<sup>st</sup> condition will be verified. If the condition is true loop will be repeated. If the condition is false then control will come out of the loop. do / while is exit controlled loop. In do while first all the statements inside the block will be executed and then condition will be verified. If the condition is true then loop will be repeated. If condition is

false then control will come out of the loop and execute statement X.

- In while when the condition is false for the first time then the statement inside the block will be executed for 0 times, whereas in do while statement will be surely executed for one time.

**5. break :**

- It helps to transfer control to another part of the program.
- It can be used in switch statement or in do, while and for loops. It also can be used in labeled blocks.
- It has three uses.
  - 1) It helps in terminating a switch statement.
  - 2) It is used to exit a loop or force immediate termination of a loop bypassing the conditional expression and other remaining code in the loop.
  - 3) It can be used in labeled loop to exit the loop. (It can be used as an alternate to goto keyword especially when you want to exit nested loops or blocks).

**6. continue :**

- It is used to move the control to the beginning of the loop.
- It is used to skip the statements inside the loop and continue with the next iteration.
- It can be used in do, while and for loops. It also can be used in labeled blocks.  
**Note:** break is used to move the control to the end of the loop but continue is used to move the control to the beginning of the loop.
- When continue is used in while or do-while loop then control is directly transferred to the test condition that controls the loop.
- When continue is used in for loop control is transferred to iteration portion first and then to the test condition.

**7. return:**

- return keyword terminates the execution in a method and returns the control to the caller method.
- It has two uses.
  - a. It immediately terminates the execution of the callee method and returns the control from callee to the caller method.
  - b. It is used to return a value from callee to caller method.

**8. goto:**

- is a reserved keyword and is not supported in java.

**9. Separators:**

| Symbol | Name         | Purpose  |
|--------|--------------|--|
| ()     | Parentheses  | Used to contain lists of parameters in method definition and invocation. Also used for defining precedence in expressions, containing expressions in control statements, and surrounding cast types. |
| { }    | Braces       | Used to contain the values of automatically initialized arrays. Also used to define a block , classes, methods, and local scopes.  |
| [ ]    | Brackets     | Used to declare array types. Also used when dereferencing array values.  |
| ;      | Semicolon    | Terminates statements.   |
| ,      | Comma        | Separates consecutive identifiers in a variable declaration. Also used to chain statements together inside for statement.  |
| .      | Period (Dot) | Used to separate package names from sub packages and classes. Also used to access members of a class using a reference variable.   |



## OOPS in Java

**Q** What is structured programming language?

**A** Any language which is supporting the following three flow structures is called structured programming language.

- a) Sequence structure or step by step execution.
- b) Conditional structure.
- c) Iterative structure.

Structured programming languages follow waterfall model.

Eg: C, Pascal, Ada, and ALGOL.

**Q** What is the difference between procedure oriented programming (POP) and structured programming?

**A** POP and structured orientation are not different; a structure orientation is part of procedural orientation.

**Q** What are the different OOP concepts?

**A** Following are main pillars of Object Oriented Programming

- a) Encapsulation: It is a process of binding the data(state) and the methods (behavior) together into a single unit.
- b) Abstraction: Providing essential properties and operations of an object by hiding internal things is called as abstraction. Because of encapsulation, abstraction is possible and where there is no encapsulation, there is no abstraction.
- c) Inheritance: It is a concept of inheriting the properties of one class (super class) into another class (sub class) using IS a relationship. A class should be closed for modification but open for extension. We can add additional features to an existing class without modifying it.
- d) Polymorphism: One form behaving differently in different situations is called polymorphism. Polymorphism is extensively used in implementing inheritance. It helps in static and dynamic binding.

**Note:** OOP makes a language highly reusable and we can also avoid complexity.

**Q** What is the difference POP and OOP (Object Oriented Programming)?

**A** Main Difference between POP and OOP are

- a) POP is one type of project development, in the same way OOP is another style of project development.
- b) Both are different approaches taken to complete a task.
- c) POP follows waterfall model whereas OOP follows spiral model.
- d) In POP the data is global, can be accessed anywhere and there is no control over data. If it is modified then it will be affecting other parts of the project.
- e) In OOP data is secure because of access specifiers and encapsulation. High level of reusability and avoiding complexity of code is possible because of relationships between classes ("IS A" and "HAS A" relationship).

**Q** What is the difference between object based and object oriented language?

**A** Object based languages are not supporting all OOP features.

Object based language = encapsulation + abstraction + compile time polymorphism.

Eg: JavaScript, VB 5.0

Object oriented languages = Object based + inheritance + dynamic polymorphism

Eg: C++, Java etc

**Q** What is the difference between partial OOP and complete OOP?

**A** In partial OOP we can define the method outside the class Eg: C++.  
In complete OOP every function is defined inside the class Eg: Java.

**Q** Is java pure object oriented or not?

**A** No, Java is not pure or 100 % object oriented because it is supporting primitive data types like int, float, etc. Java is having excellent performance because of the existence of primitive data types.

**Q** What is a pure OOP language?

**A** A complete OOP language should support the following conditions.

- a) Encapsulation
- b) Abstraction
- c) Inheritance
- d) Polymorphism
- e) All predefined data types (primitive types ) should be objects.
- f) All user defined types are objects.  
Eg: Eiffel, Ruby Small talk.

**Note:** In case of pure OOP primary data types also treated as objects.

**“Java is a complete OOP language but not pure OOP language.”**

**Java is also structured plus OOP language.**

**Q** What is the support given by java to implement OOP?

**A** Java is supporting to implement OOP with the following 4 user defined data types.

- 1) class data type
- 2) sub class data type
- 3) abstract class data type
- 4) interface data type

## **Varargs**

- It is used to simplify method overloading.
- It is generally used when you want to send multiple values of the same type as argument.
- One method can have only one vararg and it should be the last in the parameter list.
- Constructors also can have varargs.

Syntax:-

Eg: -

```
returntype methodname (datatype. . . var){  
}  
  
void m1(int... a) {  
    for(int x:a) {  
        System.out.println(x);  
    }  
}
```

## **Constructor**

- It's a special method which has the same name as class name.
- It does not have a return type even void also.
- It is also a member of class.
- They are specially designed to initialize the instance variable or to initialize the state of the object.
- Constructor is used to initialize object level resources.
- Constructor is always executed only once during the creation of every new object.
- An object can never be created without invoking a constructor.
- Constructor can be invoked only with the help of "new" keyword.
- Constructor can never be accessed with the help of dot (".") operator or it can never be invoked by an object.
- Constructor can never be invoked after creation of an object.
- If you write a return type for a constructor, then it is treated as a normal method and it can be accessed with the help of dot (".") operator and not with "new" operator.
- If you have not written a constructor in a class, a default constructor with no arguments is added in ".class" file by the compiler during compilation time.
- If you write any constructor in a class then the compiler will not add any default constructor.
- Constructor can be overloaded. Constructor overloading is a technique in java in which a class can have any number of constructors that differ in parameter lists. It is guaranteed when you make a new object; at least one constructor is executed.
- Constructor can have varargs.
- Constructor can have access specifiers (private, protected or public).
- Constructor cannot have modifiers like abstract, static and final keywords.
- Constructors are not inherited into sub class and can never be overridden because they are not methods and only methods

## **finalize() Method**

- It's a predefined method in java, which is present in the class Object.
- It is generally overridden in subclasses for finalization process. It is used for releasing object level resources before an object gets destroyed.
- It is executed only once during destruction of only dereferenced objects.
- It will be invoked implicitly by JVM before the GC (Garbage Collector) destroys the dereferenced or unused object.
- finalize() cannot be overloaded.

### ***Constructor and finalize() difference***

| constructor   | finalize()  |
|---|---|
| It is a special method, which has the same name of the class.                                 | It's a predefined method in java, which is present in the class Object.   |
| Used for initializing object level resources or state of an object or initialization process. | It is generally overridden in subclasses for finalization process. It is used for releasing object level resources before an object gets destroyed. |
| It is executed only once during <i>creation of every new object.</i>                          | It is also executed only once during <i>destruction of only dereferenced objects.</i>   |
| Constructor is invoked explicitly by developer whenever he wants to create new objects.       | It will be invoked implicitly by JVM before the GC (Garbage Collector) destroys the dereferenced or unused object.                                  |
| Constructor can be overloaded   | finalize() cannot be overloaded.  |

**Note:** Calling finalize() explicitly will never destroy the object but will only release the resources used by the object and makes the object useless. The only way we can destroy an object in java is by dereferencing or abandoning the object.

**Q** What is finalization?

**A** It is the final task which will be performed by an object before an object gets destroyed.

**Q** What is the final task which has to perform by an object ?

**A** Final task means releasing object level resources like database, file or socket connections.

**Q** What is the support given by java for doing finalization?

**A** finalize() can be used for doing finalization. It is a predefined method which is present in class Object. It is generally overridden in subclasses to release object level resources. (All classes in java are subclass of class Object).

**Q** What is the difference between object and method level resource?

**A** Any resource (Eg: Database, file or socket connection) which is used only in a single

method of a class is called as method level resource. Method level resources are created with the help of local variables. Any resource (Eg: Database, file or socket connection) which is used only in multiple methods of a class is called as object level resource. Object level resources are created with the help of instance variables.

- Q** Can a developer explicitly destroy an object in Java ?  
**A** Developer can never destroy the object in java. It is the JVM (garbage collector) which actually destroys the object implicitly. Developer can only deference or abandon an object in java.
- Q** When does the garbage collector destroy the dereferenced object?  
**A** When JVM finds there are too many dereferenced objects in the heap then JVM will send the garbage collector to destroy all the dereferenced or unused objects.
- Q** What is garbage collector?  
**A** It is a program which is implicitly executed by the JVM when it finds there are too many dereferenced or unused objects in the heap.
- Q** Why does the GC destroy unused objects?  
**A** GC destroys the unused objects because it reclaims the memory so that the same memory can be used for some other object creation.

## **static in Java**

- static is a keyword.
- It is used for declaring the members of the class(variables, methods or blocks).
- It cannot be used with local variables.
- static keyword cannot be used with constructor because constructor is related to instance creation.
- static members belong to the class.
- static members can be accessed without creating an object.
- static members (static block and method) cannot access non-static members i.e. instance members.
- Instance variables cannot be used in static context i.e. in static block or static method directly. Instance method cannot be invoked from static method or static block directly.
- this keyword cannot be used in static context (static block or instance block) because this keyword is used to work with instance variables and instance variables cannot be accessed in static context. this keyword cannot be used to access static variables, because they belong to class. "this" is a reference variable which refers to the current object.
- But static variables and static methods can be accessed in instance method or instance block directly.
- static keyword can be used with following members of the class
  - a. Variables.
  - b. Methods.
  - c. Blocks.
  - a. **Variables:**
    - Variables which are declared as static are called as static variables
    - static variables belong to class and they do not belong to instance or object.
    - Memory will be allocated for static variables or they come to life when class is loaded into the memory.
    - Static variables are destroyed when the JVM shuts down.
    - static variables are declared and initialized to default values (if we do not provide values) when classes are loaded into JVM.
    - There is one and only one copy of static variable and it belongs to class.
    - static variables behave globally because if the value is changed, then it will affect globally.
    - static variables are not global because they have to be accessed with class name or reference variable.
    - static variables can be accessed in static context (methods or blocks) directly .
    - static variables can be accessed in instance methods and blocks directly
    - static variables are stored in the heap
    - Scope of static variable i.e. static variables can be accessed inside the class directly

or outside the class with the help of "." (dot) operator. "." operator should be used with class name or reference variable.

- Static variables are used when same value is needed for all the instances.

**b. Methods:**

- Methods which are declared with static keyword are called static methods.
- static methods can be accessed without creation of object. They can be accessed either with class name, reference variable or by creating an object.
- static methods cannot access instance methods.
- static methods are used when we want to write some logic without using the instance variables.

**c. Blocks:**

- static block will be executed only once when the class is loaded into JVM.
- static block can be used for writing any logic which we want to execute only once during class loading time.
- static block is generally used for initializing static variables.

***Difference between instance block and static block:***

| instance block  | static block   |
|---|--|
| Block defined inside a class without static keyword is called instance block. | Block defined inside a class with static keyword is called static block. |
| It is executed only once in every new object when it gets created.            | It is executed only once when a class is loaded into the JVM             |
| It is used for initializing instance variables                                | It is used for initializing static variables.                            |

➤ There are two phases which happen before an object gets created.

1. Class loading phase.
2. Object creation phase.

**1) Class loading phase:**

Following things happen during class loading time.

- Class is loaded into the stack frame (JVM) by using classpath variable.
- An object of class Class gets created implicitly which has complete information about the loaded class.

(Note: The number of class Class objects depends on the number of classes getting loaded into the JVM. Class class is used by the developer for reflection process).

- static variables are declared and initialized or static variables come to life.
- static block gets executed.



**2) Object creation phase:**

Following things happen during object creation time.

- Reference variable is declared.
- Instance variables are declared and initialized or instance variables come to life in heap.
- Instance block gets executed.
- Constructor is invoked.
- The reference of the object is given to the reference variable.

**Note:** Classes are loaded only once into the JVM but any number of objects can be created after that. So class loading phase happens only once but object creation phase can happen multiple times for a particular class.

➤ There are 3 types of variables.

- a. Static variables.
- b. Instance variables.
- c. Local variables.

**a) static variables:**

- Variables which are declared as static are called as static variables
- static variables belong to class and they do not belong to instance or object.
- Memory will be allocated for static variables or they come to life when class is loaded into the memory.
- Static variables are destroyed when the JVM shuts down.
- static variables are declared and initialized to default values (if we do not provide values) when classes are loaded into JVM.
- There is one and only one copy of static variable and it belongs to class.
- static variables behave globally because if the value is changed, then it will affect globally.
- static variables are not global because they have to be accessed with class name or reference variable.
- static variables can be accessed in static context (methods or blocks) directly .
- static variables can be accessed in instance methods and blocks directly
- static variables are stored in the heap
- Scope of static variable i.e. static variables can be accessed inside the class directly or outside the class with the help of "." (dot) operator. "." operator should be used with class name or reference variable.
- Static variables are used when same value is needed for all the instances.

**b) instance variables:**

- Instance variables are declared and initialized or they come to life when the object gets created.
- They are destroyed when the object is dereferenced.

- Instance variables are declared and initialized to default values (if we do not provide values) in every object when it gets created.
- Instance variables belong to objects.
- Number of instance variables will depend on the number of instances created.
- If any instance variable value is modified in one instance it will not affect in other instance or object.
- Instance variables cannot be accessed in static context directly .
- Instance variables can be accessed in instance methods and blocks directly.
- Instance variables reside in heap.
- Scope of instance variable i.e. instance variables can be accessed inside the class directly or outside the class with the help of “.” (dot) operator. “.” operator should be used with reference variable after it is initialized by creating an object.
- instance variables are used when different values are needed for all the instances.

**c) local variables:**

- They are declared and initialized when the method is invoked.
- They are destroyed when the method completes execution.
- We should initialize local variables before using otherwise we get the “Variable might not have been initialized” compilation error.
- Local variables reside in stack.
- Scope of local variable i.e. local variables can be accessed within the method directly and not outside the class.
- local variables are used when new value is needed for every for every method processing.

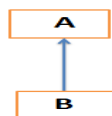
**this keyword**

- this is a keyword in java.
- this is a reference variable which refers to the current object.
- this keyword is used to access the members of the class within the same class.
- this can be used only to access instance members .
- this cannot be used to access static members of the class. this also cannot be used in static context.
- this keyword can be used to access the following members of the class
  - a. instance variables
  - b. constructors
  - c. methods
- a. **Instance variables:**
  - this keyword is used to resolve the conflicts between instance variables and local variables. (The conflict between the instance variables and local variables might be either in constructor or methods).
- b. **Constructors :**
  - this() can be used to invoke another constructor from one constructor within the same class.
  - this() can be used for invoking current class constructor (constructor chaining).
  - this() will be used for doing constructor chaining within the same class.  
**Note:** One constructor will invoke another constructor to reuse initialization logic.
  - When you are using this() to invoke a constructor then it should be the first statement inside the constructor.
- c. **Methods:**
  - this can be used to invoke another method from one method.

## Inheritance

- It is a concept of inheriting the properties of one class (super class) into another class (sub class) when there “IS” a relationship between classes.
- A class should be closed for modification but open for extension. We can add additional features to an existing class without modifying it.
- It is a process of reusing existing class functionality in new class.
- The existing class is called super class and the new class is called subclass.
- All the members of super class will be inherited into subclass (except constructor and private members) and can be accessed directly in subclass.
- Super classes are generalized classes but subclasses are specialized classes.
- Different types of Inheritance.
  - a. Simple Inheritance
  - b. Multilevel Inheritance.
  - c. Hierarchical Inheritance.
  - d. Multiple Inheritance.
  - e. Hybrid Inheritance.
- a. **Simple Inheritance:**
  - In simple inheritance there will be exactly one super class and one subclass i.e. subclass will get the functionality from exactly only one super class.
  - It is supported by class data type.

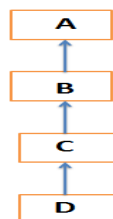
### Simple Inheritance



### b. Multilevel Inheritance:

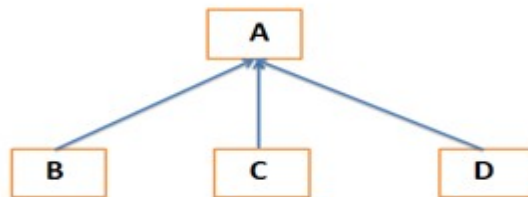
- In multilevel inheritance one super class can have many sub classes.
- One subclass can have many indirect super classes i.e. one is direct super class all remaining are indirect super classes.
- It is supported by class data type.

### Multilevel Inheritance

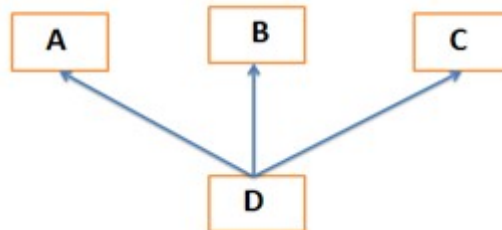


**c. Hierarchical Inheritance:**

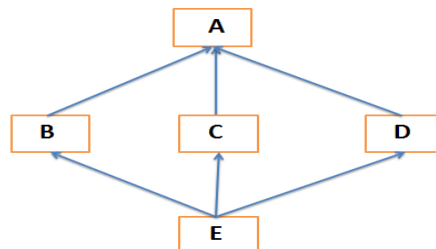
- In this one super class can have many direct sub classes.
- It is supported by class data type.

**Hierarchical****d. Multiple Inheritances:**

- In this one sub class can have many direct super classes.
- Multiple inheritance is not supported by java with class data type. But it is supported with interface data type

**Multiple Inheritance****e. Hybrid Inheritance:**

- It is the combination of multiple, multilevel and hierarchical inheritance.
- Hybrid inheritance is not supported by java with class data type. But it is supported with interface data type.

**Hybrid inheritance**

## **super keyword**

- super is a keyword in Java which should always be used in subclasses.
- super is reference variable which refers to the immediate super class of the current object.
- super is used to access the instance members of the super class from sub class.
- super keyword cannot be used to access the static members of the class or super keyword cannot be used in static context.
- super is used to access the following members of the super class from subclass.
  - a. Variables
  - b. Constructors.
  - c. Methods
- a. **Access variables:**
  - super keyword is used to resolve the conflict between super class instance variables and subclass instance variables or local variables.
  - super keyword is generally used to access the super class instance variables in subclass whenever there is a conflict between the subclass and super class instance variables.
- b. **super() (constructor) :**
  - Used to access the constructor of super class in subclass constructor.
  - super() keyword is used for constructor chaining.
  - super() keyword should be used in the first line of the sub class constructor.
  - Constructors are executed in the order of derivation, from super class to subclass. It is designed in this way because while creating a sub class object the super class is initialized first, then subclass and so on.
  - super() can be called with arguments if you want to invoke super class parameterized constructor.
  - If we don't write super() in the subclass then a default super() is added in the first line of the sub class constructor during run time by JVM .
- c. **super to access methods :**
  - super is used to access the method of super class in overridden method of subclass. The super class method is generally invoked in the subclass method for adding extra logic to the method of super class without actually modifying it.

## **Method Overriding**

- Implementing the super class method in the subclass with same name, same signature (number, order and type of parameters) and same return type is called method overriding.
- Criterion for method overriding is the type of sub class object which is created. The super class method will be hidden and it should be invoked explicitly (if required) from the overridden method of subclass.
- Method overriding is done to add or change functionality to the method present in super class.
- Method overriding helps for achieving dynamic polymorphism (also known as dynamic method dispatch).
- Only instance methods can be overridden.
- Only inherited methods can be overridden.
- private, static and final methods cannot be overridden.
- static method cannot be overridden. static method is bound with class but instance method is bound with object. static methods do not support dynamic polymorphism.
- private methods cannot be overridden because they are not inherited.
- final methods cannot be overridden because if a method is marked as final it cannot be overridden. final stops overriding or modification of a method.
- Constructors are not inherited into sub class and can never be overridden. Java supports only instance method overriding but not constructor overriding. Java does not support variable overriding also.
- If the super class method is having an access specifier then the overridden method of the sub class should have an equal or higher access specifier.

| Access specifier in super class | Access specifier which can be used for overridden method of subclass in same package |
|---------------------------------|--|
| private                         | Cannot override.   |
| default                         | default, protected, public   |
| protected                       | protected, public  |
| public                          | Public   |

**Note:-** A subclass within the same package of superclass can override any superclass method that is not declared private or final in the same package.

A subclass in a different package can only override the non-final methods and which are declared protected or public. The default method cannot be overridden because they are not inherited.

- If the super class method is throwing an exception then the overridden method in the subclass can do any one of the following
  - a. The overridden method can omit the exception.

- b. The overridden method can throw the same method level exception as used with the super class method.
- c. The overridden method can throw any unchecked exceptions, regardless of whether the super class method throws an exception or not.
- d. The overriding method cannot throw checked exceptions that are new or broader than the ones declared by the super class method. The overriding method can throw narrower or lesser checked exceptions than the overridden method.

For example if a method throws **IOException** then the subclass overridden method cannot throw **SQLException** (because new type of exception) or **Exception** (super class type of **IOException**). But it can throw its subclass type of exception like **FileNotFoundException**.

## Dynamic Polymorphism

*"Binding subclass object with super class reference variable is called as Dynamic Polymorphism"*

- A super class reference variable can refer to subclass object but what members can be accessed from the sub class object depends upon the type of super class reference variable. But if the methods are overridden it always depends on the type of subclass object, the super class reference variable is referring to. This concept is only called as dynamic polymorphism.
- To achieve dynamic polymorphism following rules must be followed:-
  - a. There must be method overriding.
  - b. Subclass object must be assigned to a super class reference variable.

The rule followed is "when a subclass object is assigned to a super class reference variable, the super class reference will call subclass overridden method".

## Type Casting Reference Variables (User Defined Data type)

- A super class reference variable can refer to subclass object but what members can be accessed from the sub class object depends upon the type of super class reference variable.
- But if you want to access the sub class members then the super class reference variable can be casted to sub class type.
- A super class reference variable can be casted to sub class type only and if the super class reference variable is referring to sub class object.
- This is done only for one main reason i.e. to access the sub class members and nothing else.
- This process is also known as down casting (means casting down the inheritance tree). Down casting has to be done explicitly.
- Up casting means casting up the inheritance tree. This happens implicitly.



## **instanceof Operator**

- instanceof is keyword in java.
- The instanceof operator is used to test whether the object is an instance of the specified data type(class, subclass or interface).
- You can use it to test if an object is an instance of a class, an instance of a subclass, or an instance of a class that implements a particular interface.
- The instanceof operator provides information about an object during runtime.
- The instanceof operator is also known as type comparison operator because it compares an object with the data type.
- It returns either true or false.
- instanceof operator can be used only when the classes are in same hierarchy or related with IS-A relation .
- instanceof operator cannot be used only when the classes are in different hierarchy or are not related.
- instanceof operator can be used only if the class used on the right side of the operator passes the IS – A test for the class used on the left side of the operator. Ex:

```
class Dog extends Anima { }  
class Demo {  
    Animal animal = new Dog(); if (animal instanceof Dog){  
        Dog dog = (Dog)animal;  
    }  
}
```

- Here class Dog used on the right side of the instanceof operator passes the IS-A test with the class Animal used on the left side of the operator.
- If we apply the instanceof operator with any variable that has null value, it returns false.
- It is always a better practice to use instanceof operator before you down cast a super class reference variable to a subclass type.

### ***Steps to identify class and members of the class:***

- Identify the number of classes, name of classes and relation between the classes.
- Identify the members of the super class as follows:-
  - a. Identify the number of variables, name of variables and data type of variables. Also identify static and instance variables.
  - b. Identify the blocks (instance and static blocks).
  - c. Identify the number of constructors and their parameters (number, data type and order)
  - d. Identify the number of methods and their name; return type and signature (number, order and data type of parameters). Identify static and instance methods.
- Follow the same above rules for the subclass and other classes also.

## **final keyword**

- final is a keyword in java.
- final stops modification of class, method or variable.
- final keyword can be used with the following things:-
  - a. class.
    - If a class is declared as final, it cannot be inherited or it stops inheritance.
    - final prevents modification of a class.
  - b. method
    - If a method is declared as final, it cannot be overridden or it stops overriding.
    - final prevents modification of a method.
  - c. variable
    - A final variable may only be assigned a value only once and cannot be modified or reinitialized later.
    - Any variable primitive or reference (static, instance, local or parameter) can be declared as final.
    - A final variable will not have a default value and has to be initialized explicitly. If the final variable is not initialized then you get a compilation error.
    - A final variable can only be initialized only once, either with an assignment statement or with an initializer (constructor/instance block for instance variable or static block for static variable).
    - A variable that is declared as final and not initialized is called a “blank final variable”. A instance “blank final” variable can be initialized in the constructor or instance block. A static “blank final” variable can be initialized in the static block. A local “blank final” variable can be initialized in the method.
    - If final is used with a static variable then it becomes a constant.
    - If the final variable is a reference variable then it cannot refer to other object but values in the instance variables of the object can be modified.

## **Abstract class**

- It is a user defined data type which can be used for implementing OOP.
- It is used to do partial implementation and setting the standards for the subclasses to complete the implementation.
- A class that is declared abstract is called abstract class.
- Abstract class may contain both concrete and abstract methods. But if a class has an abstract method, then the class should be declared as abstract.
- An abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon).
- Eg:-

```
abstract void add(int x, int y);
```

- If a class includes abstract methods, the class itself must be declared abstract, as in:
- Eg:-

```
public abstract class Hello {  
    abstract void add(int x, int y);  
}
```

- An abstract method should not be declared as static and final.
  - An abstract class also should not be declared final.
- 
- Abstract classes cannot be instantiated, but you can declare a reference variable of it.
  - When an abstract class is subclassed, the subclass must override the abstract method and provide implementation for all of the abstract methods in its parent class. If it does not, the subclass must also be declared abstract.
  - A single abstract class is subclassed by many classes that have a lot in common (the implemented parts of the abstract class), but also have some differences (the abstract methods which are overridden by subclasses).
  - A class containing all concrete methods can be declared as abstract. This is done generally if you do not want to create an instance of a particular class.
  - An abstract class can have the following members:-
    - a. static variables, blocks and methods.
    - b. instance variables, blocks and methods .(Abstract class can have state).
    - c. constructor.

## interface

- It is a user defined data type which can be used for implementing OOP.
- It does not contain any implementation.
- It is used for setting the standards for the subclasses to do the implementation.
- It contains only 2 members:-
  - a. public static final variables.
  - b. public abstract methods.
- It cannot contain any other following members:-
  - a. Instance variables
  - b. static and instance blocks.
  - c. Concrete methods (instance and static methods)
  - d. Constructor

**Note:** - All these members create the diamond problem in multiple inheritance.

- Syntax:-

```
interface name {  
    datatype variableName = value ;  
    returntype methodName (parameter list);  
}
```

- Eg:-

```
interface Hello  
{  
    int x = 10;  
    int add(int x, int y);  
    int multiply(int x, int y);  
}
```

- Interfaces cannot be instantiated but we can declare the reference variable of the interface.
- Each method in an interface is also implicitly public and abstract.
- All variables declared in an interface are by default public static final variables and have to be initialized during declaration.
- When a class is implementing the interface, then that class must override all the abstract method in the interface otherwise class should be declared as abstract.
- When a class is implementing interface, then all final static variables of the interface will be inherited into subclass.
- Eg: to implement the Hello interface:-

```

class HelloImpl implements Hello {
    public int add(int x, int y) {
        int z = 0;
        z = x + y;
        return z;
    }
    public int multiply(int x, int y){
        int z = 0;
        z = x * y;
        return z; }
}

```

- Interface supports multiple inheritance. One class can implement many interfaces. One interface can extend any number of interfaces. Multiple inheritance is not supported with the other data types (class, sub class and abstract class).
- It is used for decoupling declaration and implementation of methods.
- It always specifies what a class must do, but not how it has to be done.
- It supports dynamic polymorphism. It provides very high level of decoupling and abstraction.
- It is used for loose coupling or decoupling the dependencies between classes (Avoid inter-dependency) when there is HAS a relationship between two classes.
- It is a data type which can be used as an interface to work with a object. It is because of this feature this data type gets the name interface.
- It is used for establishing IS a relationship. It is used for doing REALIZATION.

**Q** What is the difference between an abstract class and interface?

**A** Difference between an abstract class and interface

| Abstract Class   | Interface   |
|--|---|
| Abstract class can contain the following members<br>Instance and static variables.<br>Instance and static block.<br>Concrete methods (Instance and static).<br>Abstract methods.<br>Constructor. | An interface can contain the following members:<br>public static final variables.<br>public abstract methods.<br>An interface cannot contain the following members<br>Instance variables.<br>Instance and static block.<br>Concrete methods (Instance and static methods)<br>Constructor. |
| Sub class has to extend abstract class using "extends" keyword.  | Class has to implement an interface using "implements" keyword.   |
| Abstract class doesn't support multiple inheritance.   | Interface support multiple inheritance.   |
| It is used for doing partial implementation  | It is completely open for implementation and does not contain any implementation.   |
| It supports abstraction to a smaller extent.   | It supports abstraction to a greater extent.  |
| It is used for doing GENERALIZATION  | It is used for doing REALIZATION  |

**Q** When should I use an abstract class or an interface?

**A** It depends on the scenario.

- If you do not want to do any implementation and set the standards or specification for other classes to implement then choose an interface. If you want to achieve multiple inheritance then you can choose interface.
- If you want to do partial implementation and set the standards for other classes to complete the implementation then choose abstract class.

**Q** What is the similarity between abstract class and interface?

**A**

- Both cannot be instantiated.
- We can declare reference variable for both.
- We have to override abstract methods for both.
- Both support dynamic polymorphism.
- Both support abstraction. Abstract class supports abstraction to a lesser extent, but interface supports abstraction to a greater extent.
- Both are data types provided by java for implementing OOP.
- Both are used for establishing IS a relationship.

**Q** What is the difference between an abstract class and class?

**A** Difference between an abstract class and class

| Abstract Class  | Class   |
|---|---|
| Abstract class can contain the following members <ul style="list-style-type: none"> <li>• Instance and static variables.</li> <li>• Instance and static block.</li> <li>• Concrete methods (Instance and static).</li> <li>• Abstract methods.</li> <li>• Constructor.</li> </ul> | A class can contain the following members: <ul style="list-style-type: none"> <li>• Instance and static variables.</li> <li>• Instance and static block.</li> <li>• Concrete methods (Instance and static )</li> <li>• Constructor.</li> </ul> A class cannot contain an abstract method. |
| It is used for doing partial implementation   | It is used for doing new concrete implementation.   |
| It cannot be instantiated   | It can be instantiated.   |
| It supports dynamic polymorphism.   | It does not support dynamic polymorphism.   |

**Q** When should I use a class or abstract class?

**A** It depends on the scenario.

- If you want to do new concrete implementations then choose class.
- If you want to do partial implementation and set the standards for other classes to complete the implementation then choose abstract class. If you want dynamic polymorphism support use abstract class.

**Q** What is the similarity between abstract class and class?

**A**

- We can declare reference variable for both.
- Both are data types provided by java for implementing OOP.

**Q** What is the difference between class and interface?

**A** Difference between a class and interface

| Class   | Interface   |
|---|---|
| A class contains the attributes and behaviors of an object.   | An interface contains behaviors that a class implements.  |
| You can instantiate a class.  | You cannot instantiate an interface.  |
| A class can contain the following members<br>Instance and static variables.<br>Instance and static block.<br>Instance and static methods.<br>Constructor.<br>A class cannot contain an abstract method. | An interface can contain the following members:<br>public static final variables.<br>public abstract methods.<br>An interface cannot contain the following members<br>Instance variables.<br>Instance and static block.<br>Concrete methods (Instance and static)<br>Constructor. |
| In class the declaration and implementation of the method is tightly coupled.   | Interface helps to decouple declaration and implementation of a method. This is really helpful in setting standards for other classes to complete implementation.   |
| Class is used for doing concrete implementation   | Interface does not contain any implementation and is used for setting standards for other classes to complete implementation.   |
| Class does not support multiple inheritance   | Interface supports multiple inheritance.  |
| It does not support dynamic polymorphism.   | It supports dynamic polymorphism.   |

**Q** When should I use a class or an interface?

**A** It depends on the scenario.

- If you want to do new concrete implementations then choose class.
- If you do not want to do any implementation and set the standards or specification for other classes to implement then choose an interface. If you want to achieve multiple inheritance then you can choose interface. Interface also supports dynamic polymorphism.

**Q** What is the similarity between class and interface?

**A**

- We can declare reference variable for both.
- Both are data types provided by java for implementing OOP.

## User Defined Data Types in Java

- Java has 4 user defined data types for implementing OOPs. The advantages of the four data types (when and why they are used) are as follows:-

**a. Class data type:**

- It is a user defined data type which can be used for implementing OOP.
- It is used when we want to write new concrete implementations.

**b. Sub class data type:**

- It is a user defined data type which can be used for implementing OOP.
- It is used for doing inheritance or inheriting the functionality of one class into another class.
- It is used for adding or changing implementations for existing class.
- It is used for modification of existing class.
- It is used for specialization whereas super class will be used for generalization.
- It is used for doing inheritance.
- It is used for establishing IS a relationship between classes.

**c. Abstract class:**

- It is a user defined data type which can be used for implementing OOP.
- It is used to provide partial implementation, leaving it to subclasses to complete the implementation.
- It is used for setting standards for the subclass to complete the implementation.
- It is used for doing generalization and it provides the standard or framework for the subclasses to complete the implementation.
- It is used for establishing IS a relationship.

**d. Interface:**

- It is a user defined data type which can be used for implementing OOP.
- Interface supports multiple inheritance.
- It is completely open for implementation and does not contain any implementation.
- It is used for decoupling declaration and implementation of methods.
- It is used to set the standard for other implementing classes.
- Interfaces specify only what to do but not how to do.
- It supports dynamic polymorphism.
- It provides high level of abstraction (by changing the sub class implementation).
- It is used for loose coupling or decoupling the dependencies between classes (Avoid inter-dependency) when there is HAS a relationship between two classes.
- Before Java 1.5 interfaces are used to write constants.
- It is used for establishing IS a relationship. It is used for doing REALIZATION.
- In projects developers use interfaces for decoupling layers.  
Eg: - DAO layer from service layer and Service layer from Controller layer..
- If multiple programmers are working in different module of project they still use each other's API without knowing the implementations. This brings more flexibility and speed in terms of coding and development.
- Interfaces make the application more flexible and maintenance easier.
- It is used for setting specification. (SUN uses interfaces to set specification for all



vendors).

- SUN uses interface for setting standards (API- Application Programming Interfaces) for many vendors like
- server vendor :- Servlet API, JSP API(Java Server Page), JSF API(Java Server Faces) , EJB
- API(Enterprise Java Bean)
- database vendor:-JDBC API (Java Database Connection)
- persistence provider vendor :- JPA API(Java Persistence API)
- It is a data type which can be used to interact with 3rd party classes and objects.

## **Packages**

- Packages are created with the keyword called package.
- The package statement should be the first line in the source file. Two package declaration statements are not allowed in the same source file.
- Packages are containers or are used for storing all type of java user defined data types and sub packages. Packages are used for storing the following things.
  - a. Class
  - b. Subclass
  - c. abstract class
  - d. interface
  - e. Annotation
  - f. Enumeration
  - g. Sub packages.
- A package is used for organizing set of related classes and interfaces.
- Packages help in organizing the files of your projects in a better way.
- Syntax :-

```
package packagename;
```

Eg:

- a. package a;
  - b. package a.b;
  - c. package com.cluster;
- The standard coding practice for giving a package name is lower case.
  - If you do not store a class in a package then it will be stored in a default package.
  - Packages are used to avoid naming conflicts between two classes (when both classes have same name).
  - Packages help in reusability of class.
  - Packages provide security to class and its members.
  - Packages tell the history of class or packages help us to find origin of class.
  - java & javax are two main packages which are provided by Java.

## import

- import is a keyword in java.
- import statement should be used immediately after package statement and before any class definitions.
- Generally a class stored in one package has to be used in another package with a fully qualified class name. Fully qualified class name means using a class with its package name.

➤ Eg:

```
java.util.LinkedList l = new java.util.LinkedList();
```

- import keyword helps in importing a class stored in one package to be used in another package.
- If you do import then you can use the class with simple name.

➤ Eg:

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

This statement will import all the classes inside the util package.

```
import java.util.LinkedList;
```

This statement will import only LinkedList class.

**Q** What is the difference between the above two imports? Which one is better way of importing and why?

**A Case No: 1**

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

**Advantages:**

- a. This statement will import all the classes inside the package and will be helpful when you want to use many classes from the same package.

**Disadvantages:**

- a. The star form increases compilation time because it has to import all the classes from that package. But it has no effect on the run time performance.
- b. It would be very difficult to identify which class is coming from which package.

**Note:** Using star form is not a recommended approach and is not followed by good programmers.

**Case No: 2**

```
import java.util.LinkedList;
```

**Advantages:**

- a. This statement will import only LinkedList class. It would be very clear to identify which class is coming from which package

**Note:** This is a recommended approach and is followed by all good programmers.

- Sometimes when you want to use two classes with same name from two different packages, then you have to use fully qualified class names.

## Access Specifiers

- Access specifiers are used to specify the scope for class and members of a class.
- They help in providing security to class and members of the class.
- Access specifiers can be used to restrict access.
- A class can have only two types of access.

| Type of Access for class | Keyword provided by java to support the access |
|--------------------------|--|
| Default access           | No keyword                                     |
| Public access            | public   |

- a. When a class has default access then it can be used only within the same package and cannot be visible or used outside the package.
  - b. A public class can be used or is visible in same package and in other packages.
- Members of the class (variables, constructor and methods) can have the following types of access.

| Type of Access for members of class | Keyword provided by java to support the access |
|-------------------------------------|--|
| Private access                      | private  |
| Default access                      | No keyword                                     |
| Protected access                    | protected                                      |
| Public access                       | public   |

| Access specifiers             | Access in same class | Access in sub class in same package | Access in different class in same package | Access in sub class in other package | Access in different class in other package |
|-------------------------------|----------------------|-------------------------------------|---|--------------------------------------|--|
| private                       | Yes                  | No                                  | No  | No                                   | No   |
| no access specifier (default) | Yes                  | Yes                                 | Yes                                       | No                                   | No   |
| protected                     | Yes                  | Yes                                 | Yes                                       | Yes                                  | No   |
| public                        | Yes                  | Yes                                 | Yes                                       | Yes                                  | Yes  |

### a. private:

- ✓ private members can be accessed only in the same class.
- ✗ private members cannot be accessed by another class(subclass or different class) in the same package or different package.

### b. default

- ✓ default members can be accessed by any class(same class, subclass or different class) in

the same package.

- X** default members cannot be accessed by any class(subclass or different class) in another package.

**c. protected**

- ✓ protected members can be accessed by any class(same class, subclass, different class) in same package.
- ✓ protected members can be accessed by a subclass in different package.
- X** protected members cannot be accessed by a different class in different package.

**d. public**

- ✓ public members can be accessed by any class, in any package.

## Inner classes

- Inner class is a class and is a member of another class.
- Inner classes are also known as nested classes.
- Generally inner class functionality will be used by outer classes only.
- There are four types of inner classes.

- a. Instance inner classes.
- b. Static inner classes.
- c. Method local inner classes.
- d. Anonymous inner classes.

### **a. Instance inner classes:**

- All the outer class members can be accessed inside the inner class directly.
- All the inner class members cannot be accessed inside the outer class directly but can be accessed with object.
- We cannot access inner class and its members outside the outer class directly.
- If you want to access inner class functionality outside the outer class, you have to create the object for inner class with the following special syntax.

// to access Inner class functionality outside the outer class

```
Outer o = new Outer();  
Outer.Inner oi = o.new Inner();  
OR  
Outer.Inner oi = new Outer().new Inner();
```

- We can't access outer class members outside the outer class with inner class reference variables.
- When we compile then two .class files will be created for both inner class and outer class
  - a) Outer\$Inner.class (for inner class)
  - b) Outer.class (for outer class)

### **b. Static inner classes:**

- When you write inner class as static, then that inner class area becomes static context.
- The variables and methods of the inner class will not become automatically static. If you want them as static then you need to write that member of the inner class as static.
- If you want to access the static members of the static inner class in the outer class, then you can access them directly with class name.
- If you want to access instance members of static inner class in the outer class then you have to create the object of the inner class and access.
- If you want to access outer class members inside the static inner class then those members must be static. Instance members of outer class are not allowed

inside static inner class.

- If you want to access inner class functionality outside the outer class you need to create the object of inner class with the following special syntax.

```
// To access Inner class functionality outside the outer class
```

```
Outer.Inner oi = new Outer.Inner();  
oi.showB();
```

**c. Method local inner classes:**

- A class which is declared inside a method is called method local inner class.
- When you define a class inside a method that is accessible within that method only.
- Method local inner classes cannot be used outside the method where they are defined.
- They can be used only in the same method after the class definition.

**d. Anonymous inner classes**

- Inner classes declared without any class name are called as anonymous inner class.
- Anonymous inner class implementation is nothing but writing a subclass (anonymous class) for existing class, abstract class or interface without a name and overriding the method.
- Anonymous classes are written for overriding the methods.
- When we are defining anonymous inner class, in the same line only object will be created and will assign the reference to super class reference variable.

**Q** How many inner classes can I write inside the outer class?

**A** Any number of inner classes can be written inside the outer class.

**Q** Assume that there are two inner classes A and B inside an outer class Hello. Can I write class B extends A?

**A** Yes, we can do inheritance of inner class.

**Q** Can I write interface inside a class?

**A** Yes, we can write an interface inside a class.

**Q** Can I write inner class inside an interface?

**A** Yes, we can write inner class inside an interface.

**Q** Can I write interface inside an interface?

**A** Yes, we can write interface inside an interface.

## main() method in Java:

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

Here:

**public** :- Access specifier.

**static** :- Access modifier.

**void** :- return type of the method.

**main** :- name of the method.

**String[] args** :- parameter of the method (args is a variable name of type String array) .

- Q** Why is the main() method declared as public ?
- A** Because it will be accessed by the JVM from outside the package, so we have to provide public type of access.
- Q** Why is the main() method declared as static ?
- A** Because the method is accessed by the JVM without creation of the object of the class. Static members can be accessed without creation of an object.
- Q** Why does the main() method have return type as void ?
- A** Because it is not returning any value to the caller of the method i.e. JVM.
- Q** What is String[] args?
- A** It is the parameter of the main method and it can accept command line arguments during a program execution. args is the name of the parameter of type String array.
- Q** What is main?
- A** main is the name of the method. It is the starting point of execution. After we execute java Demo in the DOS prompt, Demo.class is loaded into the JVM and it looks for main() method only. If main() is not found by the JVM in the first loaded class you get NoSuchMethodError. main() method then starts execution, loads other classes, creates objects of other classes and completes the execution.
- Q** Explain how main method executes?
- A** When we type java Demo in DOS prompt following thing will happen.
- java is a command. As soon we type it in DOS prompt, the OS will use the PATH variable, locates the exe file and executes it. Java command starts the JVM. If it is not found in the PATH, then we get internal or external command error.
  - Once it executes the JVM starts.
  - Now the class ClassLoader will use CLASSPATH variable and tries to load Demo.class file. If it is found it will be loaded into the JVM, else we get NoClassDefFoundError.
  - After loading into the JVM, it invokes only the main() method which is having String args[] as the parameter. A class containing main() method can have other methods but JVM invokes only the main() with String array as parameter. If that method is not found we get the error NoSuchMethodError.
  - main() will be accessed by the JVM from outside the package, so we have to provide public type of access.



- f. `main()` method will be accessed without creation of an object, that is why it should be declared as static.
- g. All the command line arguments are given to the parameter String array. It is optional to provide command line arguments.
- h. `main()` method starts execution and it starts loading other classes, creates objects of other classes and completes the execution.

**Q** Can I overload `main` method in java? What happens?

**A** YES. You can overload `main()` method. Compiler never reports any error because compiler never checks for `main()` in java. It only looks into the syntax and compiles. But the JRE or JVM will look for particularly `main()` method with String array and executes it. If it is not found we get the error `NoSuchMethodError`.

**Q** Can I override `main` method in java?

**A** NO. You cannot override `main()` method because it is static. Static methods cannot be overridden. If you override static methods it becomes method hiding but not method overriding. Static methods do not support dynamic polymorphism.

**Q** Can I use `varargs` for `main`?

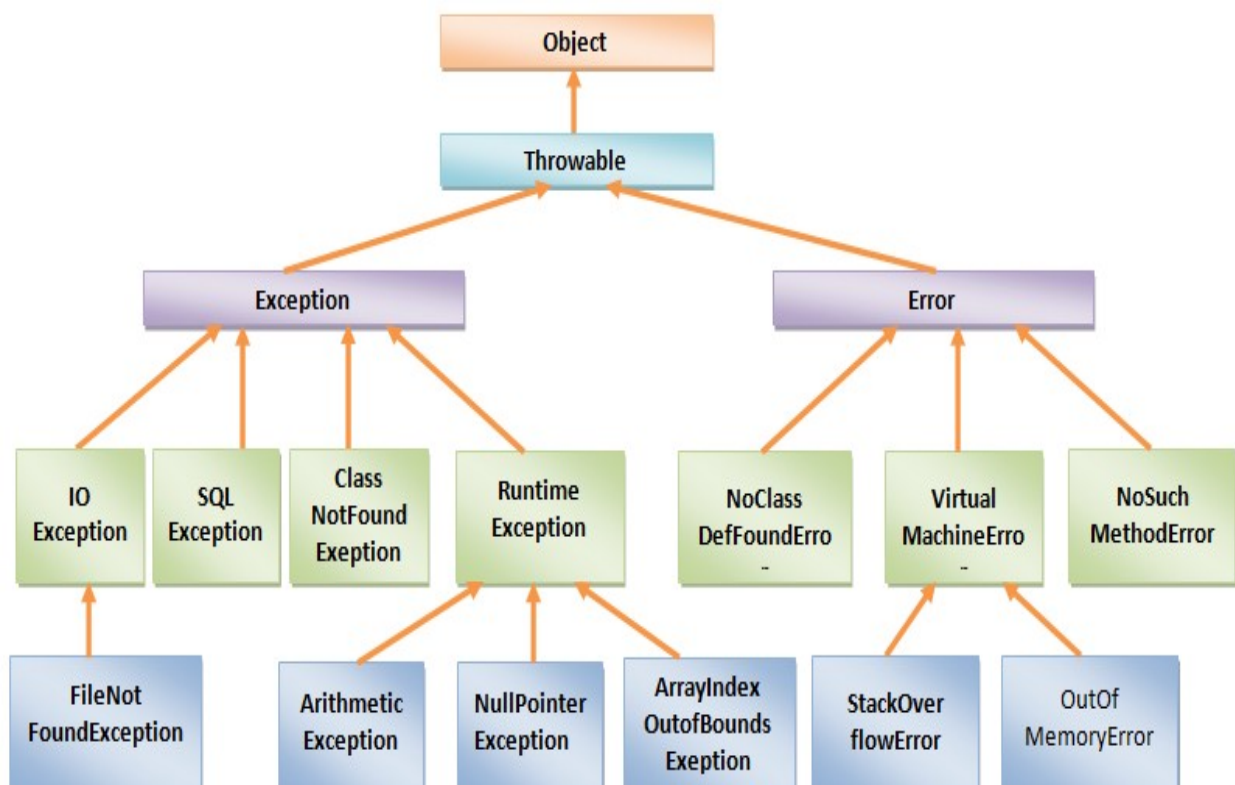
**A** Yes we can use `varargs` for `main`.

**Q** Can I print Hello World without `main`?

**A** Yes, by using static block.

## Exception Handling

- In java programming there are two types of error.
  - a. **Compilation Error:** - is an error which occurs at compilation time. They are syntax errors which have to be solved with proper coding.
  - b. **Runtime Error:-** is an error which occurs at runtime and can cause the program to terminate. It can be classified into two types.
    1. **Exception:** is a runtime error which can cause the program to terminate. You can handle the exception and continue the program execution. Here program does not terminate and execution continues normally.  
Eg: - NumberFormatException, IOException, etc.
    2. **Error:** - is a runtime error which will cause the program to terminate. You cannot handle it. Error will terminate the program.  
Eg: - NoClassDefFoundError, VirtualMachineError.
- Java clearly classifies the reason for the program termination and reports the message, so that the developer can decide how to handle the runtime error.
- All exceptions are subclasses of class java.lang.Exception.
- All errors are subclasses of class java.lang.Error.
- Both Exception and Error are subclasses of class java.lang.Throwable.



**Q** What is Exception?

**A** It is the runtime error which causes the program to terminate but it can be handled.  
Eg: NumberFormatException, ClassCastException, ArithmeticException, etc.

**Q** What is exception handling?

**A** It is the mechanism of handling the runtime error and maintaining the normal execution of the program.

**Q** How do you do exception handling?

**A** Java provides five keywords for exception handling.

- a. try.
- b. catch.
- c. finally.
- d. throws
- e. throw.

**Q** What is the advantage of exception handling?

**A** There are two main advantages of exception handling.

- a. Program will not terminate and it will retain the control within the program.
- b. Fix the runtime error.

**Q** What is Error?

**A** It is the runtime error which cannot be handled and causes the program to terminate.

**Q** How do you handle Error?

**A** Java does not support any error handling mechanism. It has to be solved with proper coding.

- In most of the other languages, if you do not handle the exception, following things will happen.
  - a. Program or application terminates.
  - b. System also terminates.
- In java whenever an exception occurs and if the developer does not handle it following things will happen.
  - a. Program terminates or application crashes.
  - b. Exception will be caught or handled by the default handler of the JVM.
- Here, in java even though the application is getting terminated, system does not crash because the exception is getting handled by the default handler of the JVM.

**Q** What happens when you do not handle an exception?

**A** If you do not handle an exception the following things might occur

- a. When you run a java program, JVM invokes main() method and then program execution will be started.
- b. All the statements inside the main will be executed one by one.
- c. If any problem happens then JVM will stop the program execution and following things will happen
  - 1. The problem is identified.

2. Identifies the corresponding java class for the problem occurred.
3. An exception object (which contains information about the error) is created and thrown inside the method.
4. JVM checks if there is any catch block or exception handler inside the method.
5. If there is no catch block and the exception is not handled then the object is thrown outside the method and main() method (program) terminates.
6. The object is caught by default handler of the JVM and prints message describing the reason for program termination. That is why system does not crash.

### ***Exception and Exception Handling in Java***

➤ Java has provided five keywords to do exception handling.

- a. try.
- b. catch.
- c. finally.
- d. throws.
- e. throw.

**a. try:**

- In a method, write the code which is subject to create an exception or error inside a try block.
- try should be used with catch or finally or both.

**b. catch:**

- catch block is used to catch the exception created in the try block and fix the error.
- catch must be used with try.
- One try block can have multiple catch statements. In some cases, more than one exception can occur in a try block. To handle such situations one try block can have multiple catch statements. But always only one type of an exception object will be created in the try block and only one catch block gets executed. When an exception is thrown, each catch statement is inspected in order and the first one which matches that exception type will get executed.
- Even though class Exception can handle all the exceptions, it is always advisable to write the specific type of Exception in the catch block. Because after catching the Exception, next we have to check the type of Exception which has occurred, and then write the logic to fix the error for that exception.
- When we are writing many catch blocks, always sub-classes should be on top and super classes should be in the bottom. (Because unreachable code in java is error).
- We cannot write any statements between try and catch block.
- We also cannot write any statements between catch and catch block.

**Q** What happens if you use try/catch in your program?

**A** Following things will happen if you use try/ catch in your program.

- a. The problem will be identified.
- b. Identifies the corresponding java class for the problem occurred.

- c. That exception object is created and thrown inside the try block.
- d. Once the exception object is created all the statements in the try block will be skipped.
- e. Control is immediately transferred to catch block.
- f. The Exception object will be caught by the catch block and statements inside the catch block will be executed. Program continues normal execution after catch block.
- g. If no error occurs then no Exception object will be created and catch block also does not get executed. Program will execute successfully without termination.

**c. *finally:***

- finally is a keyword related to exception handling in java.
- If you want to execute some statements without fail, write the code in the finally block.
- finally block is used to clear method level resources.
- Whether the exception is raised in try block or not always finally will be executed.
- finally is guaranteed to execute, even if no correct catch block is found with try.
- return statement in try or catch block doesn't stop finally block execution.
- Only System.exit() method can stop the execution of finally block.

***Different types of association with try, catch and finally.***

**A.**

```
try{  
    }  
catch(){  
    }
```

**B.**

```
try{  
    }  
finally{  
    }
```

**C.**

```
try{  
    }  
catch(){  
    }  
finally{  
    }
```

***Difference between finally & finalize***

| <b>finally</b>   | <b>finalize()</b>  |
|--|--|
| finally is a keyword in java related to exception handling             | finalize() is a predefined method present in class Object.                                       |
| finally is used for releasing method level resources.                  | finalize() is used for releasing object level resources.   |
| finally is used for writing blocks.                                    | finalize() is a method which is overridden in subclasses.  |
| finally block gets executed many times whenever the method is invoked. | finalize() is invoked only once by the garbage collector before reclaiming dereferenced objects. |

**Note:** Java insists release all resources in finally block only. You should not depend on finalize() for releasing resources because there are many times this method will not get executed.

**Q** What is the difference between final, finally & finalize?

**A** They are not similar to each other in any way. Following are the differences.

- a. final is a keyword in java:
  1. If used with a class, the class cannot be inherited. It stops modification of a class.
  2. If used with a method, the method cannot be overridden.
  3. If used with a variable, the variable has to be initialized only once.
- b. finally is a keyword related to exception handling. It is used for cleaning up method level resources.
- c. finalize() is a predefined method in class Object. It is invoked implicitly by the garbage collector while destroying dereferenced objects. It is overridden in subclasses to release object level resources.

**Q** What are the different conditions that a method gets terminated?

**A** There are three conditions or reasons for which method gets terminated.

- a. return keyword. (control returns to calling line of the caller method)
- b. An exception occurs (control returns to the catch block of the caller method)
- c. System.exit() method (program terminates)

**d. throws:**

- throws keyword is used at method level to propagate the exception object to the caller method.
- When an exception object gets created inside a method we have two options:-
  - a) You can handle it inside the same method by writing try/catch.
  - b) You can propagate to the caller method by writing throws.
- A throws clause lists the types of exceptions that a method might throw to the caller method.
- It is mandatory to use throws if the method is throwing checked exceptions (sub

classes of Exception) to the caller method.

- It is optional to use throws if the method is throwing unchecked exceptions (sub classes of RuntimeException) to the caller method.

**Q** What is exception propagation and how do you do it?

**A** Exception propagation is a mechanism of transferring or passing the exception from callee method to caller method. It is done with the help of “throws” keyword.

**Q** How did you do Exception handling in your project?

**A** I use to throw all the exceptions from lowest layer to highest layer and handle it the highest layer or in the main caller method.

**e. throw:**

- throw is a keyword used to deliberately create an exception object inside a method.
- Execution stops immediately after the throw statement and any subsequent statements are not executed.
- If catch block is present in the same method then it will be caught.
- If catch block is not present, then that exception object is thrown to the caller method.
- throw keyword can be used with only subclasses of class Throwable and not with non- subclasses of Throwable like String, System etc.

## ***Classification of exceptions***

We can classify exceptions into two types based on the provider of the exception class.

1. Built-in exceptions:- are all exception classes provided by java(SUN).
2. User defined exceptions:- are all exception classes written by developer.

Again exceptions can be classified into two types based on whether it is checked by compiler or not.

1. Checked Exception.
2. Unchecked Exception.

**Note:-** All built in or user defined exceptions are either checked or unchecked exceptions.

**Q** How to write user defined exception?

**A** Use following steps

- The class must extend either class Exception or RuntimeException.
- If you want to create a checked exception extend class Exception.
- If you want to create an unchecked exception extend class RuntimeException.

***Difference between Checked and Unchecked exceptions.***

| Checked exceptions   | Unchecked exceptions   |
|--|--|
| They are all subclasses of Exception   | They are all subclasses of RuntimeException.   |
| They are checked by compiler whether you are handling or not.  | They are not checked by the compiler whether you are handling or not.  |
| They force to the developer to handle by either writing try/catch or throws.                               | They do not force the developer to handle.   |
| They are also called as caught exceptions.   | They are also called as uncaught exceptions.   |
| Eg:- ClassNotFoundException, IOException, InterruptedException, InstantiationException, SQLException , etc | Eg:- NullPointerException, ClassCastException, NumberFormatException, ArithmeticException, ArrayIndexOutOfBoundsException. |

**Q** When will you write a checked or unchecked exception?

**A** If you want to force the caller (client) of your method to be aware and guard about the exception which might occur in your callee method then write a checked exception. If you do not want the caller (client) method not to be aware or guard about the exception in your callee method then write an unchecked exception.



## java.lang package

java.lang package is a built in package provided by SUN. All the classes of this package can be used directly because they are imported into all programs. This package contains classes which are required for basic programming.

### class Object

- class Object is the super class of all classes in java. A reference variable of type Object can refer to an object of any class. A variable of type Object can refer to any array. The class Object provides some common features to all classes like comparing, cloning, notifying and converting to String, etc.

a) **getClass():** is a final method in class Object. It returns the class of the object.

b) **finalize():** It is a predefined method in class Object.

- finalize() can be used for doing finalization. (Finalization is the final task which will be performed by an object before an object gets destroyed).
- It is generally overridden in subclasses to clear the object level resources initialized in the object.
- It is called by the garbage collector on an object when garbage collection determines that there are no more references to the object.
- It is executed only once on dereferenced objects during their destruction.

c) **equals():** It is overridden to compare equality of two objects.

- If we want to compare two primitive values we can use “=” operator. It compares the values in those variables.

Eg:

```
{  
    int a = 10;  
    int b = 20;  
    if (a==b)  
}
```

- If we want to compare two reference variables, we have two options

- a) “=” operator.
- b) equals(Object obj) method.

a) “=” operator : It compares the reference value of the variables and checks whether both the reference variables are referring to the same object. It does not compare the contents of the object but compares the address of the object.

Eg:

```
{  
    Test t1 = new Test(10,20);  
    Test t2 = new Test(10,20);  
}
```

```
        if(t1==t2) //false
    }
```

b) **equals(Object obj)** : is overridden to compare the contents of the object or the values stored in the object .

Eg:

```
{
    Test t1 = new Test(10,20);
    Test t2 = new Test(10,20);
    t1.equals(t2);//true
}
```

- If we want to compare contents of two objects then we have to override equals() method.
- If we do not override the equals() method then default implementation of equals() method in class Object is invoked whose functionality is same as "==" operator.
- This method is already overridden for String and Wrapper classes.
- This method is used when an Object is stored in java collections which use hashing like HashSet, HashMap, and Hashtable etc.

d) **hashCode()**: It returns the hashcode value of an object in terms of integer.

- Every object in the Java system has a hashcode. The hashcode is a number that is usually different for different Objects. It is used when we are storing objects in java collection classes that are using hashing like HashSet, HashMap, LinkedHashSet, etc .
- Hash code is nothing but an identification number for an object which will be used by the JVM to search a particular object among the multiple objects available in the memory. Internally JVM uses hashing technique as a searching algorithm for searching objects.
- Whenever we add an element in java collection classes that use hashing like HashSet, HashMap, LinkedHashSet, etc hashCode() is invoked implicitly which returns the hashcode of the object . This hashcode value of the object will be used while inserting the objects into the collections.
- If we do not want to use existing hash code generation algorithm which is already available in JVM, u can use your own algorithm by overriding hashCode() method in your class.
- hashCode() is already overridden for String class and wrapper classes. It is generally overridden for our user defined classes when we add them into collections (which use hashing) like HashSet, LinkedHashSet, and HashMap etc .

e) **toString()**: method is generally overridden to give a String representation to an object.

- We override this method to display the state of the object.
- toString() method available in class Object returns the String which is a combination of classname+@ +hexadecimal equivalent of hash code of an object.
- For example: Box@214ac13
- If we want to return your own String representation instead of default format, u can do by overriding toString() method in your class.

- This method is implicitly executed on an object when we use the reference variable in print statement.

Eg:-

```
Test t1 = new Test(10,20);  
System.out.println(t1);
```

- f) **clone()**: Creates and returns a copy of the object.
  - It creates a clone (duplicate object) on the invoking object.
- g) **wait()**: When we call wait() on the thread then thread will be moved to wait state and will be back to ready to run from “wait state”, only when you call notify()/notifyAll().
- h) **wait(long)**: You can specify elapsed waiting time for the thread using this method. When you call this with some specified amount of time then thread will be moved to “wait state” and will stay in the “wait state” for specified amount of time.
- i) And h) **notify()/notifyAll()**: Before the wait specified time is over if notify()/notifyAll is called on thread then thread will be moved to “Ready to run state”.
  - If notify/notifyAll() is not called on the thread until the specified time, then after the specified time is over thread will be moved to “Ready to run state”.

## class String

- String is a final class available in java.lang package which represents a group of characters.
- String is fixed in length and immutable. (Because we can never modify the contents of a String. Whenever we try to change the value of a String a new String object gets created.)
- Strings are generally used when we want to have constants in our program.
- We can create the String class object in two ways.
  - a) With “new” operator.
  - b) Without “new” operator.
- Only String class has the facility to create the object without “new” operator.

### *Difference between String object creation with new operator and without new operator:*

- When you create the String object with the “new” operator following things will happen.
  - a) String constant will be taken and it will be verified in the String constant pool.
  - b) If String constant is available in the pool JVM wouldn't place the object in the pool.
  - c) If the String constant is not available in the pool then one String object will be created and will be placed in the pool.
  - d) One more String object will be created and will be placed outside the pool.
  - e) Address of String object which is outside the pool will be assigned to reference variable

- When you create the String object without “new” operator following things will happen.
    - a) String constant will be taken and will be verified in the pool.
    - b) If String constant is available in the pool, then same address will be assigned to reference variable.
    - c) If String constant is not available in the pool, then new object will be placed in the pool and its address will be assigned to the reference variable.
  - There is concept of pooling only for String objects in java. This has been done for optimum usage of the memory.
  - String constants in the pool are never eligible for garbage collection. The same value will be used for a new String reference variable.
  - String concatenation: It is nothing but adding 2 Strings.
  - We can do this in two ways.
    - d) Overloaded “+” operator.
    - e) concat() method of String class.
- When we concatenate 2 Strings always a new String object will be created with the result. Existing String object will not be modified at any cost and because of this we can say Strings are fixed in length and immutable.

### **class StringBuffer**

- StringBuffer is a final class in java.lang package. It is used to represent group of characters.
- StringBuffer is not fixed in length and mutable.(We can modify the contents of a StringBuffer).
- StringBuffer grows and shrinks dynamically.
- StringBuffer objects should be created only with “new” operator. We cannot create an object of StringBuffer without “new” operator.
- There is no pooling concept for StringBuffer. Hence there is no reusability for StringBuffer objects like String.
- It is generally used when too many manipulations will be done.
- StringBuffer is threading safe. All the methods of StringBuffer are synchronized which makes it thread safe but slow in performance.
- It has slow performance.
- StringBuffer is recommended when thread safety is required.

### **class StringBuilder**

- StringBuilder is a final class in java.lang package. It is used to represent group of characters.
- It was added in java 1.5 version.
- StringBuilder is not fixed in length and mutable.(We can modify the contents of a StringBuilder).

- StringBuilder objects should be created only with “new” operator. We cannot create an object of StringBuilder without “new” operator.
- There is no pooling concept for StringBuilder.
- StringBuilder has the same methods which StringBuffer is having. But StringBuilder methods are not synchronized.
- All the methods of StringBuilder are not synchronized which does not make it thread safe. Hence performance is faster.
- StringBuilder is recommended when thread safety is not required.

### ***Difference between String and StringBuffer***

| <b>String</b>  | <b>StringBuffer</b>   |
|--|---|
| String is fixed in length and immutable i.e. it cannot be modified.  | StringBuffer is not fixed in length and is mutable i.e. it can be modified. |
| If you modify a String a new String object gets created, thereby creating garbage values. String has got memory leaks because dereferenced String objects in pool are never eligible for garbage collection. | If you modify a StringBuffer it is modified in the same memory.             |
| String does not grow and shrink dynamically.   | It can grow and shrink dynamically.   |
| Strings are reusable because it has a String pool.   | There is no pooling concept for StringBuffer. Hence it is not reusable.     |
| String objects can be created with new operator or without it.   | StringBuffer object has to be created with new operator only.               |
| String has got plenty of utility methods.  | StringBuffer does not have too many methods.                                |

**Q** What is the similarity between String and StringBuffer?

**A** Both are used to represent or work with group of characters.

### ***Difference between StringBuffer and StringBuilder***

| <b>StringBuffer</b>                               | <b>StringBuilder</b>                                   |
|---|--|
| All the methods of StringBuffer are synchronized. | All the methods of StringBuilder are not synchronized. |
| It is slow in processing.                         | It is faster in processing.                            |
| It is an old class.                               | It is a new class added in 1.5 version.                |
| It is thread safe.                                | It is not thread safe.                                 |
| It is used when thread safety is required.        | It is used when thread safety is not required.         |

**Q** What is the similarity between StringBuffer and StringBuilder?

**A**

- Both are used to represent or work with group of characters.
- Both are having common methods.
- Both do not have pooling mechanism.
- Both objects have to be created using “new” operator.

**Q** When to use String, StringBuilder or StringBuffer?

**A**

- String: - If the Object value will not change in a scenario use String Class because a String object is immutable.
- StringBuilder: - If the Object value can change and will only be modified from a single thread, use a StringBuilder because StringBuilder is unsynchronized (means faster).
- StringBuffer: - If the Object value may change, and can be modified by multiple threads, use a StringBuffer because StringBuffer is thread safe (synchronized).

**Note:** It is highly recommended to use StringBuilder nowadays if any manipulation has to be done. Else use String when no manipulation is done or when you require a constant. StringBuffer is very rarely used nowadays.

**Q** How to convert a StringBuffer into String?

**A**

- By using toString() method of StringBuffer.
- By using the constructor of String(StringBuffer sb)

**Q** How to convert a String into StringBuffer?

**A**

- By using constructor of StringBuffer (String s).

**Q** How do you check the equality of two StringBuffer objects?

**A**

Because equals() method is not overridden in StringBuffer it will invoke the super class Object equals() method whose default implementation checks the references. So convert StringBuffer to String by calling toString() method and then call equals() on the String objects to check the equality of the objects.

## **class System**

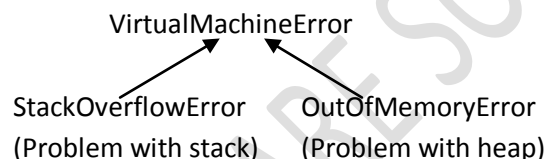
- It encapsulates all the details about the System. It contains all static members.

## **class Runtime**

- Encapsulates the details of the runtime environment i.e. JVM.
- You cannot instantiate a Runtime class because it is implemented based on singleton design pattern.

## **Memory management and garbage collector in Java**

- In C, we have two functions malloc() and calloc() to allocate the memory and we can clear that memory using free() function.
- In C++, we can allocate the memory or create an object using new operator and we can destroy the object using delete operator.
- But in java we can create objects or allocate memory using new operator but there is no way to clear the memory explicitly by the programmer.
- In java when you create object then memory will be allocated in the heap and because of not having any process to deallocate memory of objects, at some point heap will be full. Due to this JVM is unable to create new objects in the heap and JVM throws VirtualMachineError (OutOfMemoryError).
- When you create more objects of different classes, then more classes will be loaded in stack memory, because of this JVM is unable to load other classes into the memory and then JVM throws VirtualMachineError (StackOverflowError).



### **Memory management in JVM.**

- In java JVM is only responsible to clean the unused objects. Whenever memory shortage is coming in the heap, then JVM invokes garbage collector and handovers list of unused object. Then garbage collector will deallocate the memory of all unused objects in that list. JVM identifies one particular object as unused object by verifying in any of the following ways.
  1. When you assign the null to the reference variable then object referred by that reference variable is unused and is eligible for garbage collection.
  2. When u assign the value of one reference variable to another reference variable.
  3. When the reference variable is out of scope (related with methods) then the object that is referred by that reference variable is eligible for garbage collection.
- So JVM task is to identify the unused objects and invoking gc to clean all the unused objects which have been marked.
- gc is a thread which is running behind the scene whose task is to collect the list of unused objects and clean the memory.
- JVM uses one of the following ways to invoke the gc.
  1. System.gc();
  2. Runtime rt = Runtime.getRuntime();  
rt.gc();

- If we want to invoke gc, we can invoke using the above mentioned ways but there is no guarantee it will be invoked. So we can't force the garbage collector
- Sometimes some objects may refuse the gc to deallocate the memory even though they are unused and have been marked for garbage collection. The reason being the objects might be holding some resources like JDBC connections, file, printer etc.
- If we want to clean the memory of objects which are holding the resources, first we have to release the resources and then only gc will clean the memory of those objects without any trouble.
- You need to override Object class finalize() method to write the resource release code or cleanup code.

### ***Garbage collection management in JVM***

1. Identifying the unused objects and preparing a list.
2. Whenever memory shortage is expected then invoke the gc method in either of 2 ways
  - ☞ `System.gc()`
  - ☞ `Runtime rt = Runtime.getRuntime();`  
`rt.gc();`
3. The gc() method will call the runFinalization() method in either class System or class Runtime.
4. The runFinalization() method will invoke the finalize() method on all the deferred objects.
5. Lastly the deferred objects or unused objects will be cleared from the memory or deallocated from the memory.

**Q** When does the garbage collector run?

**A** The garbage collector is under the control of JVM. The JVM decides when to run the garbage collector. The developer can try to run the garbage collector from within his program by calling `System.gc()`, but there is no guarantee. As of java 6 version the garbage collector has evolved to such an advanced state that it is recommended that you never use `System.gc()` in your code.



## **Wrapper classes**

- Wrapper class encapsulates or wraps the primitive type.
- There are two main advantages of converting a primitive to wrapper or objects:-
  1. We get support for lot of utility methods like converting from one data type to another data type, comparing two values, checking equality of two values, etc.
  2. Since java collection classes like LinkedList, TreeSet etc. store all value as objects and do not allow primitives to be stored; we have to convert all primitives to objects.
- Java is giving support for both primitive data types and objects to store primitive values.
- There are 8 wrapper classes which are equivalent to 8 primitive data types.

| Primitive type | Wrapper class |
|----------------|---------------|
| byte           | Byte          |
| short          | Short         |
| int            | Integer       |
| long           | Long          |
| float          | Float         |
| double         | Double        |
| char           | Character     |
| boolean        | Boolean       |

### ***Advantages of wrapper classes:-***

- ☞ We can convert primitive to objects and objects to primitive.
- ☞ We can convert one data type to another data type. For example:- converting int to double, double to float, float to String, String to float etc.
- ☞ It also provides support for utility methods like checking equality of two values, comparing two values etc.

**Q** Is Java 100% or pure object oriented language?

**A** No, it is not pure object oriented language because Java is supporting primitive data types like int, float, char etc.

**Q** When should you use primitive or wrapper class?

**A** If you want to store values in collection or you want to use the utility methods then use wrapper class. But if you want write expressions or do mathematical operations like addition, subtraction etc. use primitive data type.

**Q** Why is Java giving support for both primitive and wrapper?

**A** Reason is performance.

- Java is having excellent performance because of the existence of primitive data types. Primitive is having better performance while doing mathematical operations or when used in expressions.
- If you use wrapper classes in expressions or mathematical operations too much of boxing and unboxing will occur. This will become an overhead and brings down the performance.

- Wrapper classes should be used only when you want to store the values in collections or when you want to use the support of utility methods like converting from one data type to another data type, comparing two values, checking equality of two values, etc.
- Following are the general conversions we generally do.
  1. Primitive to wrapper. (int to Integer).
  2. One primitive to another primitive type. (int to float).
  3. Primitive to String. (int to String).
  4. wrapper to primitive.(Integer to int)
  5. wrapper to String.(Integer to String)
  6. String to primitive.(String to int)
  7. String to wrapper.(String to Integer)

**Note:** All primitive types can be converted to String and corresponding Wrapper objects.

**Q** How to convert primitives to wrapper objects?

**A**

- a) By using the corresponding constructor.
- b) By using `valueOf(primitive type)` method

```
//Program to convert primitive to wrapper object
// 1) By using constructor
//2) By using valueOf() method
// to convert to int to wrapper object
int i = 10;
Integer x = new Integer(i);
Integer p = Integer.valueOf(i);
//to convert double to wrapper object
double d = 99.99;
Double y = new Double(d);
Double q = Double.valueOf(d);
//to convert boolean to wrapper object
boolean b = true;
Boolean z = new Boolean(b);
Boolean r = Boolean.valueOf(b);
```

**Q** How to convert primitive int to other primitive type like float or double etc?

**A** By using the `xxxValue()` method.

```
//Program to convert one primitive to another primitive type

int i = 999;
Integer z = new Integer(i);
float f = z.floatValue();
double d = z.doubleValue();
byte b = z.byteValue();
short s = z.shortValue();
long l = z.longValue();
```

```
//cannot convert int to boolean and character  
//char c = z.charValue();  
//boolean b = z.booleanValue();
```

**Q** How to convert primitive to String?

**A**

- 1) By using the toString() method.
  - a) toString(primitive type) method of corresponding wrapper class
  - b) toString() method
- 2) By using the valueOf() method of String class.
  - a) valueOf(primitive type) method
  - b) valueOf(wrapper object) method

```
// Program to convert primitive to String  
  
//1)Using toString() method  
//a) Using toString(primitive value) method of  
corresponding wrapper class  
int x= 99;  
String s1 = Integer.toString(x);  
  
//b)Using toString() method  
byte b = 10;  
Byte bb = new Byte(b);  
String s2 = bb.toString();  
  
//2)Using valueOf() method class String  
//a) Using valueOf(primitive value) method of class String  
short s = 12;  
String s4 = String.valueOf(s);  
  
//b)Using valueOf(wrapper object) method of class String  
double d = 99.99;  
Double dd = new Double(d);  
String s3 = String.valueOf(dd);
```

**Q** How to convert wrapper object to primitive?

**A** By using the xxxValue() method.

```
// Program to convert wrapper object to another primitive  
type  
  
int i = 999;  
Integer z =new Integer(i);  
  
byte b = z.byteValue();  
short s = z.shortValue();  
long l = z.longValue();  
float f = z.floatValue();  
double d = z.doubleValue();  
  
// cannot convert int to boolean and character  
// char c = z.charValue();
```

```
// boolean b = z.booleanValue();
```

**Q** How to convert wrapper object to String?

**A**

- a) By using toString() method.
- b) By using the toString(wrapper object) method of the corresponding wrapper class.
- c) By using the valueOf(wrapper object) method of class String class.

```
// Program to convert wrapper to String
    int x = 99;
    Integer i = new Integer(x);
    //1)using toString() method
    String s1 = i.toString();

    //2) Using toString(wrapper object) method in the
        corresponding wrapper class

    String s2 = Integer.toString(i);

    //3)Using valueOf(wrapper object) method of class String
    double d = 99.99;
    Double dd = new Double(d);
    String s3 = String.valueOf(dd);
```

**Q** How to convert String to primitive?

**A**

- a) By using the parse XXX(String) method.
- b) By using valueOf(String) method.

```
//Program to convert String to primitive

    String s1 = "99";

    //1) By using parseInt(String) method
    int i = Integer.parseInt(s1);

    //2)By using valueOf(String) method, convert into
        //wrapper object and then call intValue() method

    Integer z = Integer.valueOf(s1);
    int j = z.intValue();
```

**Q** How to convert String to Wrapper object?

**A**

- a) By using constructor.
- b) By using the valueOf() method.

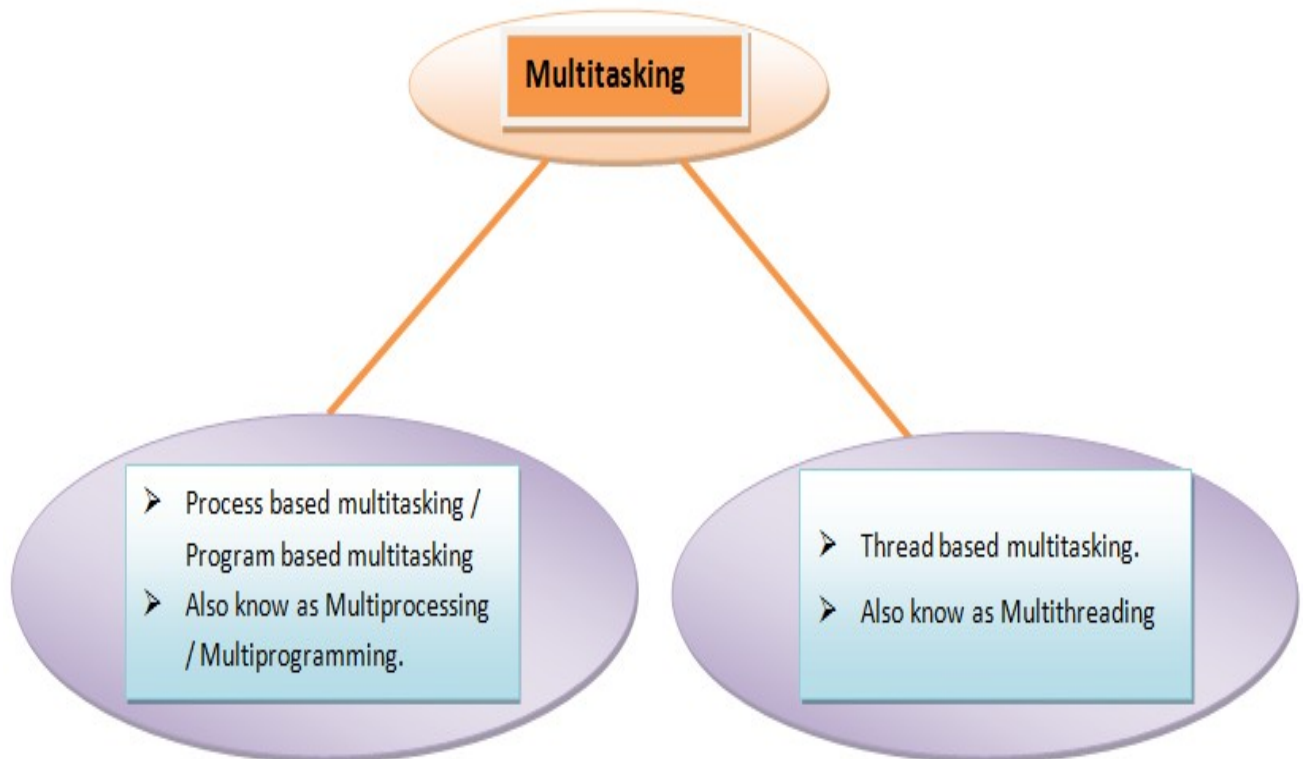
```
// Program to convert String to Wrapper  
  
String s1 = "999";  
  
// 1) using constructor  
  
Integer y = new Integer(s1);  
  
// 2) Using valueOf(String) method of corresponding  
wrapper class  
  
Integer x = Integer.valueOf(s1);
```

- Q** How to convert primitive int to primitive boolean?  
**A** We cannot.
- Q** How to convert primitive int to primitive character?  
**A** We cannot.
- Q** How to convert primitive boolean to any primitive number type?  
**A** We cannot.
- Q** How to convert character to other data type?  
**A** We cannot.

## Exploring Multithreading

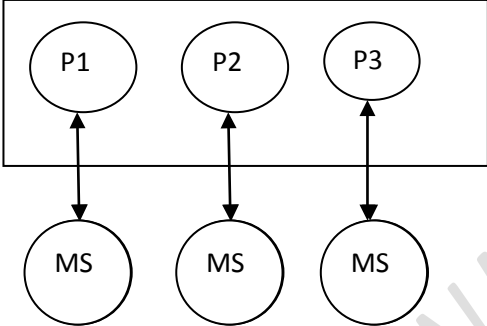
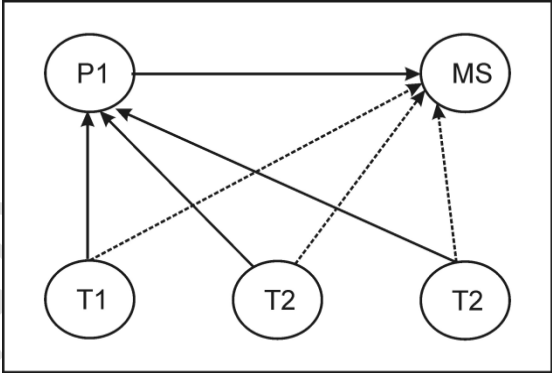
### ***Multitasking***

- Running more than one task concurrently at the same time is called multitasking.
- It is done to use CPU idle time.
- There are two types of multitasking:-



- Multiprocessing (Process based multitasking):-** Running more than one process or a program concurrently is called multiprocessing or multiprogramming. For example using Microsoft word, playing media player and at the same time using a browser.
- Multithreading (Thread based multitasking):-** Running more than one execution in a process or a program concurrently is called multithreading. For example when using Microsoft word, there will be many executions like spell checker, printing and user typing all happening at the same time.

***Difference between multitasking and multithreading:-***

| Multiprocessing  | Multithreading  |
|--|---|
| Definition: Process is nothing but a program. Multiprocessing is nothing but running multiple process or programs concurrently | Definition: Thread is nothing but a part of a program. Multithreading is nothing but running different parts of same program concurrently or running multiple threads concurrently. |
| It is heavy weight and all process share different memory space.   | It is light weight and all threads share the same memory space.   |
| Context switching one process to another is expensive.   | Context switching from one execution (thread) to another is inexpensive.  |
| Data sharing and communication is also expensive.  | Data sharing and communication is inexpensive.  |
| Diagram:<br> <p>MS=Memory Space</p>          | Diagram:<br> <p>MS=MEMORY SPACE</p>  |

**Multiprocessing:**

- All the tasks will be executed by CPU. Even though it looks CPU can execute more than one task at a time, but practically it is not possible. CPU can execute only one task at a time.
- As a user of computer we can observe multiple programs are running concurrently which is happening because of Operating System CPU scheduling algorithm.
- Some of the CPU scheduling algorithms used by the OS scheduler are:-
  - a. Round robin scheduling / Time Slicing.
  - b. Priority based / Preemptive.
  - c. Shortest job first.
  - d. First come first serve.
- The task of above scheduling algorithms is switching the CPU from one task to another task or we can say allocating the CPU time for all the tasks which are started.
- Among these time slicing or priority based are very popular.
- Windows follows priority based, UNIX follows round robin.

**Multithreading:**

**Q** What is a thread?

**A** A thread is a part of a program which has a separate path of execution.

**Q** What is multithreading?

**A** Multithreading is a process of running multiple threads concurrently or running different parts of same program concurrently. A multithreaded program contains two or more parts that can run concurrently.

**Q** What is the advantage of doing multithreading?

**A** Multithreading is always done to use CPU idle time. It thus helps in increasing the performance of an application.

**Q** Why thread is lightweight?

**A** Whenever a thread is created within a program it will not occupy any separate space. It will share the same memory space of the program. It will also not consume more resources; hence we say thread is lightweight.

☞ In a single threaded environment, your program has to wait to finish a task completely and then move to the next task which decreases the performance of the application.

☞ Java gives full support to multithreading with the help of its library.

☞ The JVM has a thread scheduler which decides which thread to execute in your java program.

☞ The JVM scheduler implementation varies for different JVM of different OS.

☞ The JVM thread scheduler negotiates with the OS scheduler and finally decides which thread to execute.

☞ That is the reason a multi threaded java program execution will behave differently in different OS.

☞ JVM creates one thread called "main" and main thread calls the main() method.

☞ Following thing happen when a java program starts execution or when you say "java Demo".

1. JVM will create a thread group called "main".
2. JVM will create a thread with the name "main".
3. JVM adds the "main thread" to "main group".
4. JVM will take the command line arguments and based on the arguments a String array[] will be constructed.
5. The class will be loaded into the JVM.
6. main() method will be invoked by passing the String array[] as an argument.

☞ If we want, we can also write our own threads. Threads written by us are called child threads or user threads.

☞ JVM starts the main thread and we can start the user thread or child thread from main() method.

☞ Every thread in java has a separate call stack.

☞ SUN has provided 2 different ways to create the child threads:

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

**Note:** In both the cases you override the run() method and write your logic inside that method you want to execute in a separate thread. The run() method can call other methods, create objects, etc.



**1. By extending class Thread.**

Following are the steps to create and execute a thread when you extend class Thread.

**Step 1:** Write a class by extending class Thread and define the thread by overriding run() method.

```
class MyThread extends Thread // define a thread
{
    public void run {
        System.out.println("Child thread executing");
    }
}
```

**Step 2:** Create a thread by instantiating that class.

```
MyThread mt = new MyThread(); // create a thread
```

**Step 3:** Start the thread by calling the start() method.

```
mt.start(); // start thread
```

**2. By implementing interface Runnable.**

**Step 1:** Write a class by implementing interface Runnable and define the thread by overriding run() method.

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Child thread executing");
    }
}
```

**Step 2:** Create an instance of the Runnable object.

```
MyRunnable mr = new MyRunnable();
```

**Step 3:** Create a thread by passing the runnable object to the constructor of Thread class.

```
Thread t = new Thread(mr);
```

**Step 4:** Start the thread by calling the start() method.

```
t.start();
```

**Q** Which is the better way of creating a thread? By extending class Thread or by implementing interface Runnable.

**A** It is not good to write a thread by extending class Thread because:-

- If you extend class Thread then you will inherit all the super class functionality into your subclass which might become a overhead.
- You also cannot extend another class.
- According to OOP you are tightly coupling two things, the thread and the business logic which is a bad design.

It is good to write a thread by implementing interface Runnable because:-

- You can extend another class.
- You are decoupling both the thread and the business logic which is a very good design.

**Q** Can I overload run() method. If I do what will happen to that overloaded method?

**A** Yes, we can overload run() but there is no use. JVM invokes run() method with no arguments implicitly and it will always execute in a separate call stack. In the case of the overloaded run() method you have to invoke the method explicitly and this will never create a new call stack or different path of execution. It will execute in the same stack where you are invoking.

**Q** Why should you call the start() method after creating the class Thread instance?

**A** When you create an instance of class Thread only a normal java object gets created. You have to call start() method to begin the new thread .

**Q** What happens when you call the start() method?

**A** When you call the start() method following things will happen:-

- a. A new call stack will be created or a new thread of execution starts.
- b. The new thread will be moved from "new state" to "runnable state".
- c. The thread in the "runnable state" will move to "running state" when the run() method is called by the JVM implicitly.

Note that every thread in java has a separate call stack.

**Q** After a thread completes execution, can it be restarted by again calling call the start() method ?

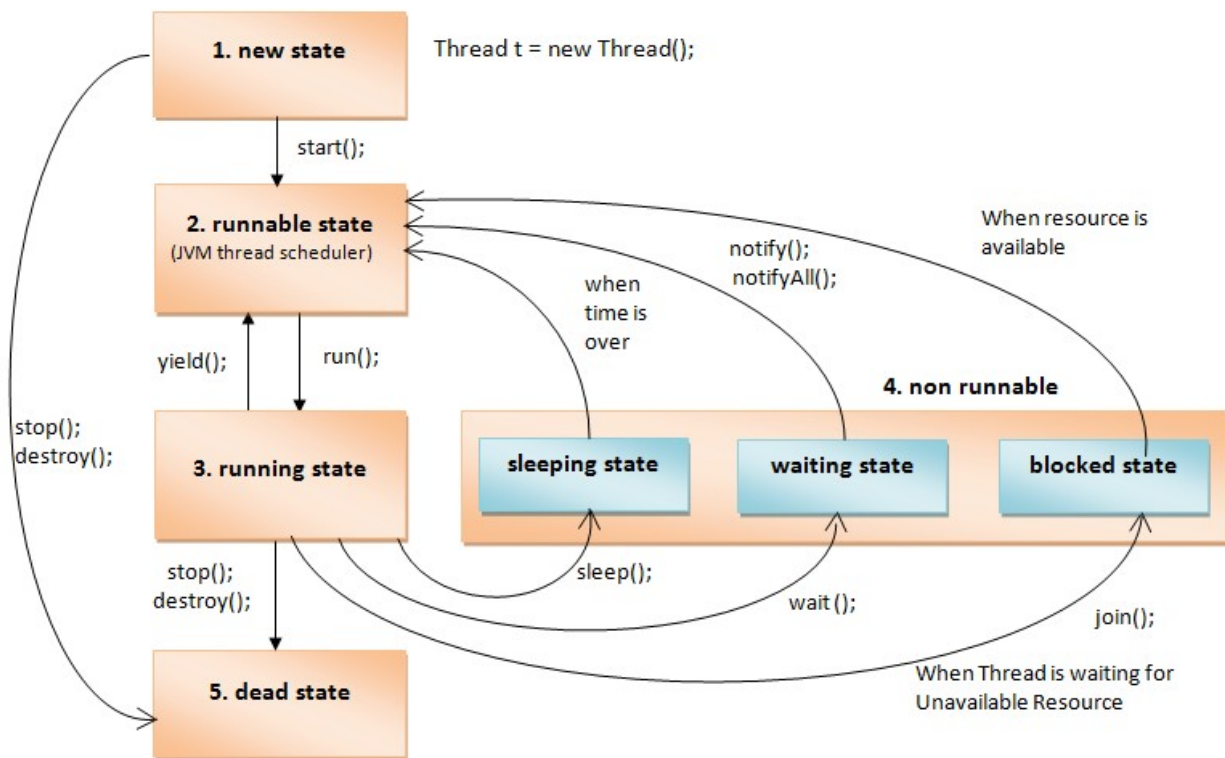
**A** No, once a thread has been started, it cannot be restarted again. If you call the start() method again it will cause `IllegalThreadStateException`. Once the run() method of a thread completes execution, then the thread moves into the dead state and the call stack is removed. You can always start a new thread only once. A runnable thread or a dead thread cannot be started again.

**Q** Who invokes the run() method?

**A** The JVM thread scheduler implicitly invokes the run() method when it decides to execute a thread and moves the new thread from "runnable state" to "running state" . This method will execute in a separate call stack after the thread is started.

**Q** Can the run() method be called explicitly by you?

**A** Yes, but it will execute as a normal method and it will never create a separate call stack or it will not start a new thread of execution. It will execute like a normal method in the same stack you are invoking.

**Life cycle of a thread**

➤ A thread can be in any of the 5 states.

- New state.
- Runnable state.
- Running state.
- Non runnable state (Sleeping state, Waiting state and Blocked state).
- Dead state.

- As a java developer your task is creating the thread and starting the thread.
- After creating the thread it will enter into "new state". A thread in "new state" is not considered to be alive.

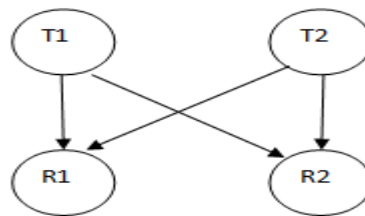
```
Thread t1 = new Thread(new MyRunnable()); // thread will enter new state
```

- When you call the `start()` method then new thread will create a separate call stack and move from "new state" to "runnable state". A thread in "runnable state" is considered to be alive.

```
t1.start(); // new thread will move from new state to runnable state
```

- Only the threads in "runnable state" are eligible for moving into "running state". All threads in "runnable state" simply wait for the JVM thread scheduler. The JVM thread scheduler negotiates with the OS scheduler and based on the scheduling algorithm decides which thread to move into "running state".

- Note:** This is the only state from which a thread can enter into “*running state*”. A thread cannot enter into “*running state*” from any other state.
5. Once the thread is decided, JVM calls `run()` on the thread and the thread will move to “*running state*”.
- Note:** At any point of time, we can see only one thread in “*running state*”.
6. When you call `yield()` method on the running thread then it will move back to “*runnable state*” to allow other threads of the equal priority to get their turn. The `yield()` method is used for explicitly moving the running thread from “*running state*” to “*runnable state*”. In some JVM which use round robin scheduling algorithm, the thread itself will implicitly or voluntarily give up control and move from “*running state*” to “*runnable state*” to give a chance to other threads for execution.
7. When you call `sleep()` on the running thread then it will be moved to “*sleeping state*” and when specified elapsed time is over then the sleeping thread will move to “*runnable state*”. When thread is sleeping, we can’t force the thread to move to “*runnable state*”. Any other disturbances happen for sleeping thread then thread will show `InterruptedException`.
8. When you call `wait()` on the running thread then thread will move to “*waiting state*”.
- Note:** We have 3 overloaded `wait()`, `notify` and `notifyAll()` method in class `Object`.
- If you want to move the waiting thread to “*runnable state*” you can call `notify()`/`notifyAll()` method which are present in class `Object`.
  - Using `notify()` will move only one waiting thread to “*runnable state*”. If there are multiple threads present in “*waiting state*”, then only one thread enters the “*runnable state*” and there is no guarantee which thread will move from the “*waiting state*” to “*runnable state*”. It will again be decided by the JVM thread scheduler.
  - Using `notifyAll()` will move all waiting threads from “*waiting state*” to “*runnable state*”.
9. When a running thread is waiting for a resource which is not available then thread will move to “*blocked state*”. Later when it is available, thread will move to “*runnable state*”. “*Blocked state*” is very dangerous, sometimes it leads to deadlocks.



- In the above scenario T1 which is holding R1 is waiting for R2 and T2 which is holding R2 is waiting for R1.
  - T1 releases the resource R1 after getting R2 only and T2 releases the resource R2 after getting R1 only. This kind of a situation is called Deadlock.
  - As a java developer, we should write the code to avoid deadlocks.
10. When `run()` method execution is completed then thread task is completed and thread will go to “*dead state*”. The stack for that thread is also removed. A dead thread is always not alive. A dead thread cannot be bought back into “*runnable state*” or any other state. In the previous versions there are 2 methods `stop()` and `destroy()` to destroy the thread explicitly.

But in current version both the methods are deprecated.

**Note:** Any number of threads can be present in new, runnable, sleeping, waiting or blocked state but there will be only one thread in “running state”.

### ***Different states of a thread:***

There are five states of a thread.

#### **1. New state:**

- A thread moves into the “new state” when a Thread instance has been created, but the start() method has not been invoked on the thread. A thread is not alive in the “new state”.

```
Thread t1 = new Thread(new MyRunnable()); // thread will enter new state
```

#### **2. Runnable state.**

- When you call the start() method then new thread will create a separate call stack and move from “new state” to “runnable state”. A thread in “runnable state” is considered to be alive.

```
t1.start(); // new thread will move from new state to runnable state
```

- A thread can also return to the “runnable state” after coming back from a running, sleeping, waiting or blocked state.
- Only the threads in “runnable state” are eligible for moving into “running state” .
- All threads in “runnable state” simply wait for the JVM thread scheduler.
- Once the thread is decided, the JVM thread scheduler calls run() on the thread and the thread will move from “runnable state” to “running state”

#### **3. Running state.**

- It is the “running state” state in which the thread is actually executing.
- A new thread moves into the “running state” for the first time when the run() method is invoked by the JVM thread scheduler.
- A thread from “non runnable state” can also enter into “running state” but only after first moving to “runnable state”.
- A thread can move out of the “running state” to runnable, non runnable or dead state for various reasons like for example when we call yield(), sleep(), wait(), join or stop() method etc.

#### **4. Non runnable state (Sleeping state, Waiting state and Blocked state).**

- A thread in a “non runnable state” is not eligible for running but it is alive.
- All threads in a “non runnable state” can only move to “runnable state” but not to “running state”.
- We can again classify non runnable state into three types.

##### **1) Sleeping state:**

- A thread moves into the “sleeping state” when sleep() is called on a running thread.

##### **2) Waiting state:**

- A thread moves into the “waiting state” when wait() is called on a running thread

**3) Blocked state:**

- A thread moves into the “blocked state” when join() is called or when a resource is not available.

**5. Dead state.**

- Once the run() method completes execution, the thread moves into a “dead state”.
- A thread once dead cannot be restarted again.
- A dead thread is not alive.
- We can also call stop() or destroy() method explicitly to move a running thread into “dead state” but the methods have been deprecated.

**Note:** A thread is alive only in runnable, running and non runnable state. A thread is not alive in new and dead state.

- Following are the different methods present in different classes related to multithreading which play a key role in thread life cycle and moving the threads from one state to another state.

| interface Runnable | class Thread | class Object |
|--------------------|--------------|--------------|
| run()              | run()        | wait()       |
|                    | start()      | notify()     |
|                    | sleep()      | notifyAll()  |
|                    | join()       |              |
|                    | yield()      |              |

***Different methods of class Thread.***

- 1) sleep():- When you call sleep() on the running thread then the thread will be moved to “sleeping state” and when specified elapsed time is over then the sleeping thread will move to “runnable state”.
- 2) join(): The join() method joins the current thread or calling thread to the end of the joining thread. The calling thread will enter into the blocked state and waits for the joined thread to complete execution. After the joined thread completes execution then the calling thread will complete execution. The overloaded join (int millis) method waits for the particular amount of time to complete the execution. If the joined thread does not complete the execution within that time, then the calling thread will move into “runnable state” and become eligible for running.

**Q** What is join() method ?

**A** If you use join() method, it joins current thread or calling thread to the end of the joining thread. It also stops the execution of the current running thread and it waits in the “blocked state” until the joined thread completes execution.

**Q** When do you use join()?

**A** If there are two threads “A” and “B” and if you do not want “B” thread to complete execution before “A” thread then call join() on the “B” thread. “A” thread now waits for “B” thread to complete execution and then it will complete its own execution.

- 3) yield(): When you call yield() method on the running thread then it will move back to “runnable state” to allow other threads of the equal priority to get their turn.

**Thread Priorities:-**

- Priority is a number which is given to a thread.
- This number is ranging from 1 to 10. 1 is the minimum priority and 10 is the maximum priority.
- In Thread class, 3 constants are defined related to this priority.
  - a. `int MIN_PRIORITY.` (1)
  - b. `int NORM_PRIORITY.`(5)
  - c. `int MAX_PRIORITY.` (10)
- In Thread class two methods are related to priority.
  - a. `int getPriority()`. We can get the priority of the thread using this method.
  - b. `void setPriority(int p)`. We can set or change the priority of the thread using this method.
- When there is no priority set for a thread then JVM sets the default priority of the thread that creates it. For example, if "A" thread with a priority 7 creates another child thread "B", then the newly created child thread "B" will also have the same priority of 7.
- Thread priority concepts will be useful only when your operating system is supporting "preemptive" or "priority based" scheduling algorithm, otherwise no use.
- In case of priority based scheduling algorithm the thread with the highest priority will get the CPU time first.
- If a thread enters the "runnable state" and it has a higher priority than all the threads and a higher priority than the currently running thread, the lower priority running thread will be moved back to "runnable state" and the highest priority thread will be chosen to run.
- For example: Thread "A", "B" are in "runnable state" with priority 4 and 5. Another thread "C" is in "running state" with priority 6. At the same time another thread "D" with priority 9 has entered into "runnable state". Now the scheduler will move the lower priority running thread "C" to "runnable state" and gives the chance to "D" because it has the highest priority.
- In case of non-priority based scheduling algorithms there is no use of this value.

**Q** What values are allowed for the priority of the thread?

**A** Values from 1 to 10.

**Q** What is the default priority of the thread?

**A** The default priority of the new thread depends on the priority of the thread which is creating it. For example, if "A" thread with a priority 7 creates another child thread "B", then the newly created child thread "B" will also have the same priority of 7.

**Q** What will happen when you give a value greater than 10 as priority of the thread?

**A** Program will compile successfully but will give the following exception during runtime: `java.lang.IllegalArgumentException`.

**Q** What is `isAlive()` method?

**A** This method is used to check whether the thread is alive or dead, it returns the boolean value `true/false`. If `true` thread is alive, if `false` thread is dead.



**Synchronization:**

- Synchronization is a process of locking the object so that only one thread can access that object.
- When object is locked by the thread then other threads have to wait until the lock is released.
- In java every object has just one lock.
- The lock has to be enabled by using “synchronized” keyword.
- In multithreaded environment multiple threads will run concurrently and will try to access single object. Sometimes this can lead to inconsistent results. Because of this we can stop multiple threads accessing the object and allow only one thread to access the object at a time, and this can be done by locking the particular object.
- A running thread will enter either a blocked/waiting state when it does not get the lock on a particular object.
- We can implement synchronization in two ways.
  - a. Method level synchronization.
  - b. Block level synchronization.

**a. Method level synchronization:-**

Syntax:

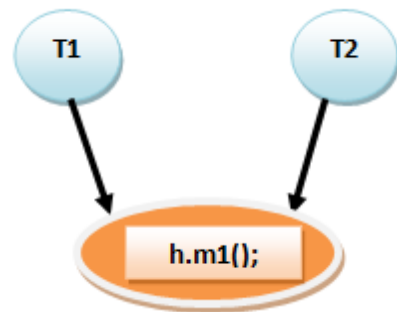
```
synchronized void m1() {  
}
```

Eg:

```
class Hello {  
    synchronized void m1() {  
        System.out.println ("Inside m1");  
    }  
    void m2() {  
        System.out.println("Inside m2");  
    }  
}
```

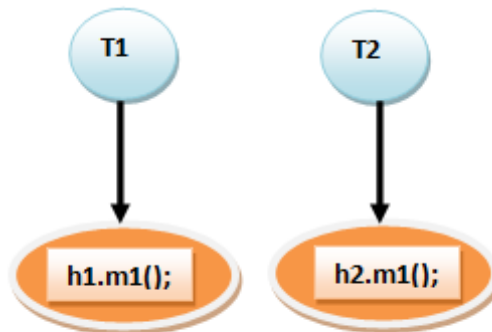
- Method level synchronization is done when you do not want many threads to access a method of an object at the same time.
- The lock on the object will be released immediately after the method completes execution.
- Assume that there are two threads T1 & T2 which are trying to use m1() method with the same Hello object. When T1 is using object “h” then object “h” will be locked by T1 and T1 executes the m1() completely. After the method execution completes the lock is released by T1. As long as T1 is executing m1(), T2 will be in the “blocked state” and has to wait to use that object. After T1 completes execution T2 will use “h” object and then executes m1() method.





**Different threads accessing same method with same object**

- Assume that there are two threads T1 and T2 where T1 is trying to invoke m1() with “h1” object and T2 is trying to invoke m1() with “h2” object. In this case both the threads can execute concurrently because both the threads are using different objects.



**Different threads accessing same methods with different object.**

- In the case of method level synchronization, the object which you are using to invoke the synchronized method will be locked.
- An object can have more than one synchronized method but there is only one lock for that object.
- The locks are not per method, they are per object.
- A class can have both synchronized and non-synchronized methods.

**b. Block level synchronization.**

Syntax:

```
synchronized (anyobject)
{
}
```

Eg :

```
class Hello
{
    void m1(){
        System.out.println ("Begin m1");
        synchronized(someobject){
        }
    }

    void m2(){
        // some statements
        System.out.println("Inside m2");
    }
}
```

- When you are using synchronized block, then the object which you are passing as a parameter to the synchronized block will be locked.
- There are two advantages with block level synchronization:-
  - 1) We can lock only few required statements inside a method.
  - 2) We can lock third party objects, whose methods are not synchronized.

#### ***Difference between method level synchronization and block level synchronization***

| Method level synchronization   | Block level synchronization   |
|--|---|
| With this the object which you are using to invoke the synchronized method will be locked. | The object which you are passing as parameter to synchronized block will be locked. |
| In this all the statements inside the method will be synchronized.                         | With this we can synchronize only some statements of the method.                    |
| With this we can lock only the objects of the class whose methods are synchronized.        | With this we can lock objects of 3 <sup>rd</sup> party class(other classes)         |

#### ***Some key points about synchronization:***

- Every thread in java will have its own copy of local variables but will use the same copy of instance variable when multiple threads are using one object.
- Only methods or blocks can be synchronized, not variables or classes.
- An object can have more than one synchronized method but there is only one lock. If one thread has entered a synchronized method on an object, then no thread can enter any other synchronized method on the same object.
- If a thread has invoked a synchronized method on an object and has a lock, the same thread can invoke another synchronized method on the same object because it is this thread which is having the lock on that object.
- If a class has both synchronized and non-synchronized methods, multiple threads can access the non-synchronized methods concurrently.
- static methods also can be synchronized.

Syntax:

```
class Hello {
    static synchronized void m1() {
        System.out.println ("Inside static m1");
    }
}
```

- Instance methods are synchronized to protect instance variables and static methods are used to protect static variables.
- For every java class loaded, the JVM implicitly creates a default object of type java.lang.Class. This object will also have a lock. If you have a static synchronized method in a class and when the static method is invoked on the class, the lock will be applied on the object of java.lang.Class.
- If you have a class Hello and you have created two objects of Hello class. Now there will be totally three objects which will be created, two of type Hello and one of type java.lang.Class.
- If a static method on class Hello is synchronized and if a thread invokes it, the lock will be applied on the object of java.lang.Class.

**Q** If I declare m1() as synchronized what will happen?

**A** The object which you are using to invoke the method m1() will be locked.

**Q** Inside a method fun(), I have written one synchronized block and passing Hello object as parameter to synchronized block. What will happen?

```
class Test {
    void fun() {
        System.out.println ("Begin fun");
        synchronized(hello) {
            // some statements
        }
        System.out.println ("End fun");
    }
}
```

**A** The "hello" object which you are passing to the synchronized block as parameter will be locked.

**Q** I have a class Hello with two synchronized methods m1() and m2() and one object called "h" is created for this Hello class. Now two threads T1 and T2 are trying to use m1() and m2() respectively with same object "h". What will happen?

```
class Hello {
    synchronized void m1() {
        System.out.println ("Inside m1");
    }
    synchronized void m2() {
        System.out.println("Inside m2");
    }
}
```

**A** In this scenario only one thread can access object "h". When T1 holds the lock on "h" and is calling m1() then T2 is unable to access object "h". After T1 completes execution of the method the lock is released on the "h" object. Now T2 will hold the lock on "h" object and will invoke m2() method.

**Q** I have a class Hello with two methods m1() and m2() where m1() is synchronized and m2() is non-synchronized. There is one object "h" created for this class Hello. There are two threads called T1 and T2 trying to access m1() and m2() respectively with the same object "h". What will happen?

```
class Hello {  
    synchronized void m1() {  
        System.out.println ("Inside m1");  
    }  
    void m2() {  
        System.out.println("Inside m2");  
    }  
}
```

**A** Concurrently T1 can access m1() and T2 can access m2() on the same object "h".

**Q** I have a class Hello with method m1() which is synchronized and static. What will happen when I call the method m1() with class name?

```
class Hello {  
    synchronized static void m1() {  
        System.out.println ("Inside m1");  
    }  
}
```

**A** In this case the default object of java.lang.Class will be locked.

**Q** I have a class Hello with two static methods m1() and m2() which are synchronized. What will happen when one thread T1 is calling m1 with class name and another thread T2 is trying to access m2() with class name?

```
class Hello {  
    synchronized static void m1() {  
        System.out.println ("Inside m1");  
    }  
    synchronized static void m2() {  
        System.out.println("Inside m2");  
    }  
}
```

**A** When T1 is accessing m1() the default object of java.lang.Class will be locked. Then T2 is unable to access m2() with the same default object, because the default object is already locked by T1.

**Q** When should you do synchronization?

**A** Synchronization is done to protect data when two or more threads are concurrently accessing method that is making manipulation on the data.

**Q** When is the lock on the object released?

**A** It is released when the method completes execution.

**Q** What is race condition?

**A** It is condition where multiple threads are racing against each other on the same resource to complete execution in a method.

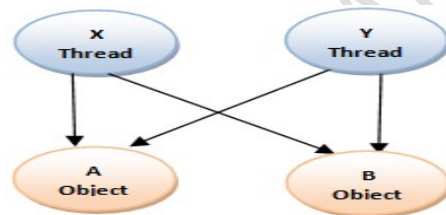
**Q** What are the drawbacks of synchronization?

**A** Following are the drawbacks of synchronization:

- It restricts concurrency because multiple threads cannot execute the same method at the same time.
- It brings down performance of an application because all thread who wants to access the object have to wait to get the lock and then continue execution.
- It can also causes deadlocks.

**Q** What is a deadlock?

**A** Deadlock can occur in a situation when a thread is waiting for an object lock, that is acquired by another thread and second thread is waiting for an object lock that is acquired by first thread. Since, both threads are waiting for each other to release the lock, the condition is called deadlock.



**Q** What is starvation?

**A** It is condition where one thread is waiting for other thread to complete execution for a long time.

**Q** What is the difference between starvation and deadlock?

**A** Starvation means long time waiting, dead lock means forever waiting. If starvation is serious, it is converted to deadlock.

### ***Interthread Communication:-***

- It is a process of making synchronized threads communicates with each other.
- Inter-thread communication is a mechanism in which a running thread is temporarily suspended and moved out of the synchronized code (method or block). Then another thread is allowed to enter into the synchronized code and execute. (Remember only one thread which has acquired the lock can execute the synchronized code).
- It is supported by following methods of Object class:
- Please note that there are also two more overloaded wait() methods in class object.
  - wait():-** It temporarily suspends the execution of a running thread and moves the thread into "waiting state". This waiting thread waits until another thread enters the synchronized

context and invokes `notify()/notifyAll()` method or until the specified waiting time is over. When a thread waits it releases the lock for other threads to use. The `wait()` gives up the lock immediately.

- b. `notify()`:** moves only one waiting thread that is waiting for the object's lock to "runnable state".  
If there are multiple threads present in "waiting state", then only one thread will move from the "waiting state" to "runnable state". There is no guarantee which thread will move and it will be decided by the JVM thread scheduler.
- c. `notifyAll()`:** will move all waiting threads that are waiting for the object's lock from "waiting state" to "runnable state".

**Note:** Both `notify()/notifyAll` will give up the lock after the thread completes the execution of synchronized code(method or block) but not immediately like `wait()`. But the main goal of using these methods is to communicate with the waiting thread and move it to "runnable state".

**Some key points about `wait()`, `notify()`, `notifyAll()` methods:-**

- All the methods should be called in a synchronized context.
- If any of these methods are invoked in a non-synchronized method or block then `IllegalMonitorStateException` will be caused.
- Only a thread owning a lock can call any of these methods.
- The `wait()` method moves a running thread to "waiting state" and makes it to wait endlessly till the `notify()/notifyAll()` method is invoked.
- The overloaded `wait(long time)` moves a running thread into "waiting state" and makes the thread wait until `notify()/notifyAll()` is invoked. If the `notify()/notifyAll()` method is not invoked then the waiting thread moves to "runnable state" after the specified waiting time is over.
- `notify()/notifyAll()` tells a waiting thread to move into "runnable state".
- `notify()/notifyAll()` gives up the lock only after the execution of the synchronized code completes.

**Q** What is the difference between `wait()` and `sleep()` ?

**A** Following are the differences:-

| <b><code>sleep()</code></b>   | <b><code>wait()</code></b>   |
|---|--|
| It doesn't release any lock or monitor.   | It releases the lock or monitor.   |
| It is used for pausing an execution.  | It is used for inter-thread communication.   |
| It can be used in any method.   | It should be used only in synchronized context.  |
| A sleeping thread moves into a "runnable state" when the specified sleeping time is over. | A waiting thread moves to "runnable state" when <code>notify()/notifyAll()</code> is invoked or when specified waiting time is over. |

- Q** What is inter-thread communication?
- A** It is a process of making synchronized threads communicates with each other. It is also known as thread to thread communication.
- Q** What is wait() ?
- A** It temporarily suspends the execution of a running thread and moves the thread into "waiting state". This waiting thread waits until another thread enters the synchronized context and invokes notify()/notifyAll() method or until the specified waiting time is over. When a thread waits it releases the lock for other threads to use.
- Q** What is notify() ?
- A** It moves only one waiting thread from "waiting state" to "runnable state".
- Q** What is notifyAll() ?
- A** A It moves all waiting threads from "waiting state" to "runnable state".
- Q** What happens when you call wait() or notify()/notifyAll() in non-synchronized context ?
- A** A It will cause IllegalMonitorStateException.
- Q** Which are the methods which give up or release the lock?
- A** wait(), notify()/notifyAll() methods give up the lock. The wait() releases the lock immediately but notify()/notifyAll() will release after the execution of the synchronized code(method or code) completes.
- Q** Which are the methods that do not give up the lock?
- A** sleep(), yield() and join() methods do not give up the lock.
- Q** Which are the deprecated methods of class Thread ?
- A** Following methods have been deprecated:
- a. suspend()
  - b. resume()
  - c. stop()
  - d. destroy()
- Q** Why are these methods deprecated?
- A** All these methods were causing serious system failures and other performance problems like deadlock etc.

### **Daemon Threads:-**

- The daemon threads are also known as service threads and simply run in the background to provide service to user threads in your java program.
- Sometimes we want to execute some invisible independent process. Such threads are called as daemon threads. Like example spell checker, grammar observer in MS word.
- There are two types of threads "user thread/child thread" and "daemon thread".
- Garbage collector in java is a good example for a daemon thread which is implicitly created by the JVM.
- If required we can also create daemon threads.
- By default all threads created by main thread are user threads or foreground thread.

- You can create a daemon thread or background thread by calling `setDaemon(true)` method.
- Once a thread execution is started as user thread, it is not possible to change it to daemon thread.
- So always `setDaemon(true)` should be invoked before starting a thread or before `start()`.
- We can check whether a thread is a daemon thread or not by calling `isDaemon()` whose return type is boolean. If true it is a daemon thread.
- JVM treats both user thread and daemon thread in the same way. The only difference is JVM will shut down as soon as all the user threads complete executions no matter how many daemon threads are executing. JVM will not wait for the daemon threads to complete the execution.
- JVM does not wait for the daemon threads to complete execution because when there is no user thread running then there is no use of daemon thread to provide service.
- Generally the `run()` of a daemon thread will be an infinite loop.
- For a JVM to keep running it must have at least one live user thread.

**Q** What is the difference between user thread and daemon thread?

**A** JVM treats both user thread and daemon thread in the same way. The only difference is JVM will shut down as soon as all the user threads complete executions no matter how many daemon threads are executing. JVM does not wait for the daemon threads to complete execution.

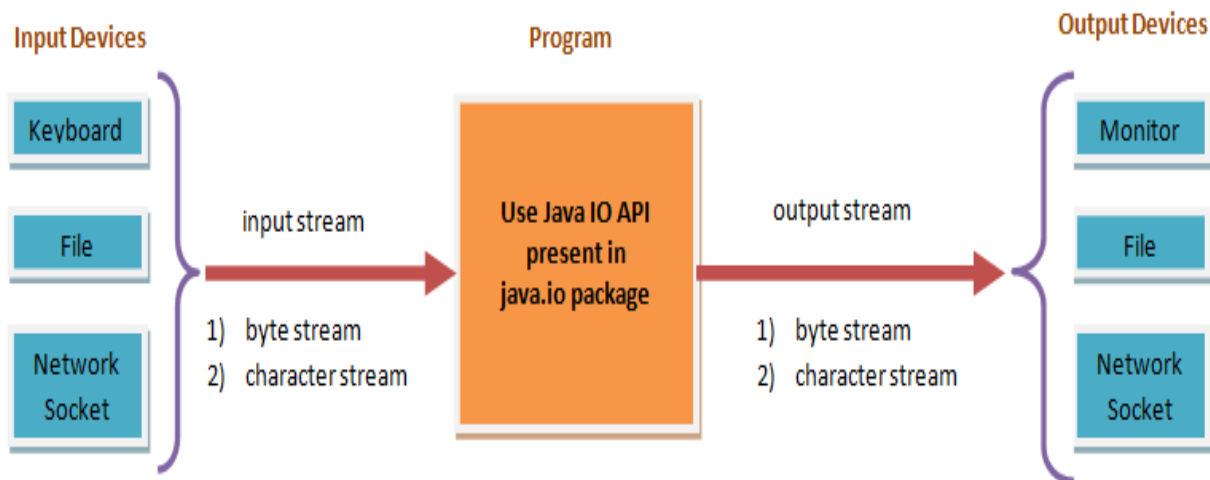


## java.io package

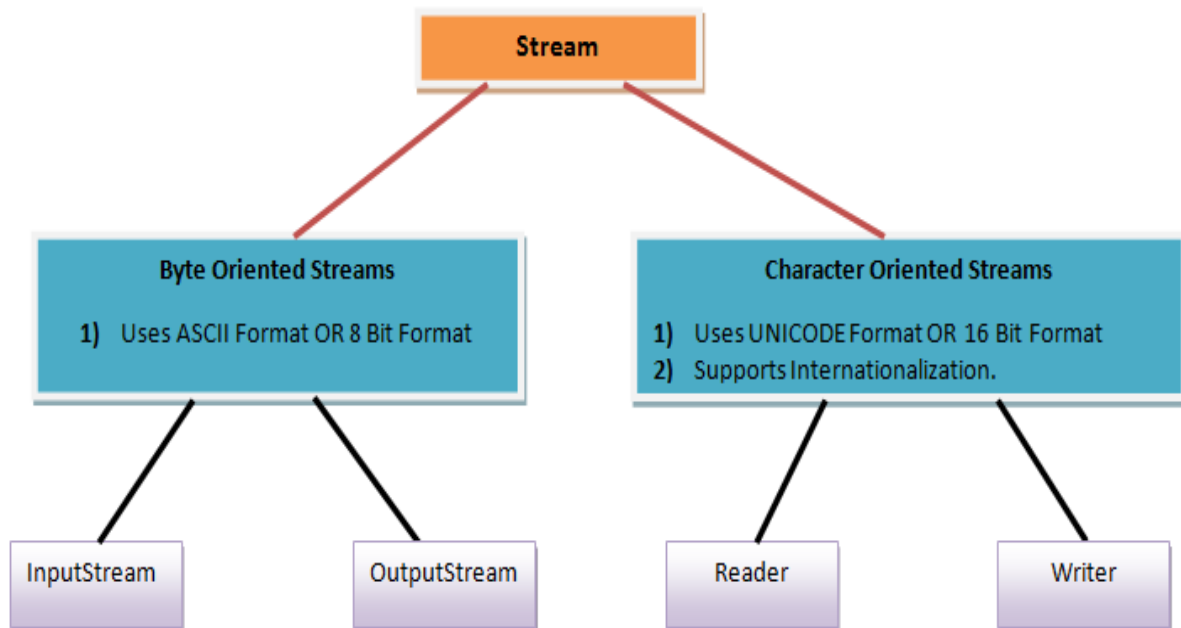
- java.io package is a built in package provided by SUN.
- This package contains classes which are used for doing I/O operations.
- All the classes of this package have to be imported into all programs.

### **Streams:-**

- Java programs perform I/O through streams.
- A stream is an abstract layer which is used for transferring data.
- A stream is a logical connection between a java program and a device.
- A stream is used to read the data from input device or it is used to write the data to output device.



- We can divide the streams into two types based on the operations they perform.
  - 1) Input streams: - It is a logical connection between a java program and input device.
  - 2) Output streams: - It is a logical connection between a java program and output device.
- Again we can divide the streams into two types based on the data they carry.
  - 1) Byte streams.
  - 2) Character streams.



**1) Byte streams:-**

- ☞ It reads and writes data in the form of bytes.
- ☞ It uses ASCII format (8 bits).
- ☞ java.io.InputStream is super class for all input byte streams.
- ☞ java.io.OutputStream is super class for all output byte streams.

**2) Character streams:-**

- ☞ It reads and writes data in the form of characters.
- ☞ It uses Unicode format (16 bits).
- ☞ It supports I18N (Internationalization).
- ☞ java.io.Reader is super class for all input character streams.
- ☞ java.io.Writer is super class for all output character streams.

**Q** Explain System.out.println() in java.

**A** Following is the answer

- a) System is a predefined class which contains information about the system.
- b) System.out is an object of type PrintStream. It is an output stream object which makes a connection between monitor and the program.
- c) println() is a method present in class PrintStream. It prints the value.

**Q** Explain

```
BufferedReader reader = new BufferedReader(new  
InputStreamReader(System.in));
```

**A** Following is the answer

- a) System is a predefined class which contains information about the system.
  - b) System.in is an object of type InputStream. It is an input stream object which makes a connection between keyboard and the program.
  - c) InputStreamReader is a mediator which converts 8 bit format to 16 bit format.
  - d) BufferedReader will read the data in the 16 bit format by using readLine() method.
- In the above diagram, we have seen some input devices like keyboard, file, network, etc. These are the source for data.
  - There are also some output devices like monitor, file, network socket, etc. These are the destination for the data.
  - We developers need to write a program which has to collect the data from input device, process the data and send the output to some output device.
  - Generally in other languages like "C" we have to write low level implementation directly. When device changes you need to modify the program again. This increases maintenance and reduces system flexibility.

*This problem is eliminated in java by providing the portable API in java.io package.*

## **Exploring java.util package (library)**

- java.util package is a built in package and is a very rich library provided by SUN.
- The classes present in this package are used for development of all types of java application (standalone, web based and mobile).
- This package is having many utility classes that provide support to data structures, string tokenizing, internationalization, data formatting, time and date formatting, generating random numbers, etc.
- All the classes from this package have to be imported explicitly.
- The collections framework is part of the java.util package.

## ***Collection Framework***

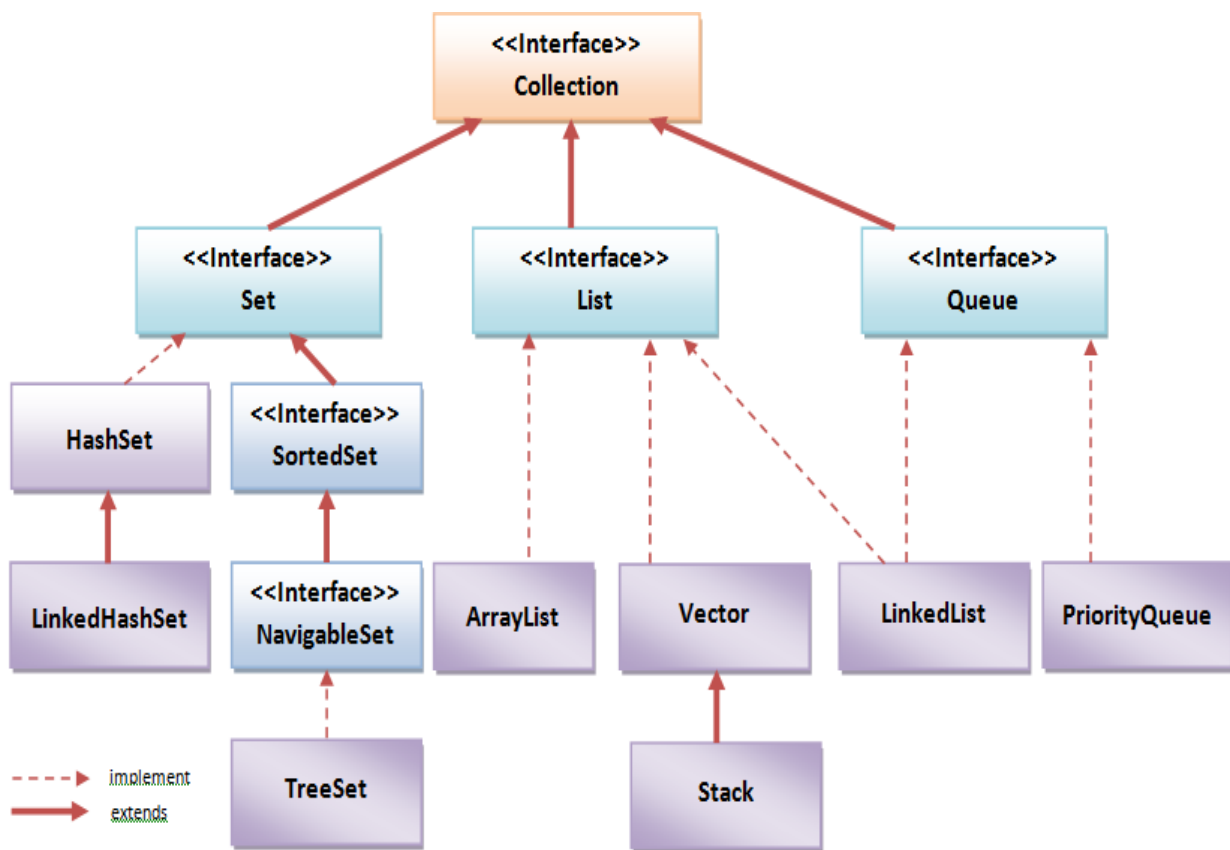
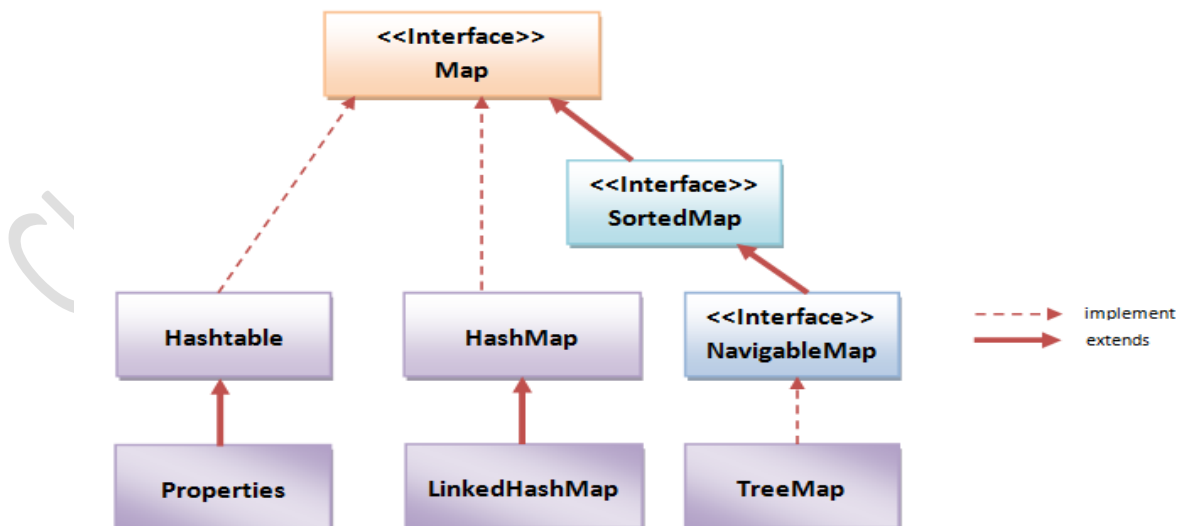
- Java provides us an API to work with different data structures such as trees, lists, sets, arrays, maps, etc.

**Q** What is a data structure?

**A** A data structure is a logical way of storing and organizing data. The different examples for data structures are array, linked list, tree, stack, queue etc.

**Q** What is a data engine?

**A** A data engine is a practical implementation of a particular data structure and allows us to do add, update, delete or select operations in a data structure.

**Diagram 1:- Collection Framework to support collections:-****Diagram 2:- collection framework to support map (key-value pair) :-**

***Collections in Java:-***

- Collection is group of similar data.
- Collections allow us to add, retrieve update or delete data (CRUD operations) in primary memory using different data structures.
- Java provides collection framework to work with collections.
- The collection framework was added in java 1.2 version.
- A framework is a standard way of providing services very easily.
- The collections framework has a set of standard interfaces to work with different data structures in a standard way.
- All the classes in the collection framework implement these standard interfaces and hence provide a standard way to work with different data structures.
- All the classes in the collection framework are data engines encapsulating different data structures and have very high efficiency.
- In Java, developer need not worry about preparing these data engines, instead he can directly use the ready implementations (classes) provided in the library (java.util package).
- For example class LinkedList encapsulates linked list data structure, class TreeSet encapsulates tree data structure, class ArrayList encapsulates array data structure, etc.
- Other classes and interfaces which are supporting collections in Java are:-
  - 1) Collections. (Class)
  - 2) Enumeration.(Interface)
  - 3) Iterator.(Interface)
  - 4) ListIterator.(Interface)

***History of Java collections:-***

| HISTORY OF JAVA COLLECTIONS |   |
|-----------------------------|---|
| Java Version                | Features Added  |
| 1.0                         | Vector, Stack, Dictionary, Hashtable, Properties and Enumeration are introduced   |
| 1.1                         |   |
| 1.2                         | Collections Framework was introduced (List, Set & Map interfaces and subsequent implementing classes like LinkedList, ArrayList, TreeSet, HashSet, TreeMap etc were released) |
| 1.3                         |   |
| 1.4                         |   |
| 1.5                         | New java syntax was introduced (Enhanced for loop, Autoboxing & Generics). PriorityQueue class was added newly  |
| 1.6                         | NavigableSet, NavigableMap, Deque interfaces were introduced.   |
| 1.7                         | Empty diamond syntax for generics was introduced  |

➤ Before to Java 1.2, there are five legacy classes and one legacy interface which are as follows.

- 1) Vector.
  - 2) Stack.
  - 3) Dictionary. (obsolete)
  - 4) Hashtable.
  - 5) Properties.
- 1) Enumeration.(interface)

- ☞ These five legacy classes are used to store and manipulate collection of objects.
- ☞ Vectors, Stack are used to store multiple objects.
- ☞ Dictionary, Hashtable and Properties are used to store key-value pair.

Problems with these legacy classes: -

- ☞ All legacy classes are synchronized by default i.e. when an object is synchronized only one thread can access that object at a time.
- ☞ All these classes support only few data structures.

**Some key points to remember while working with collections:-**

- Collections cannot store primitive data.
- All collection classes (classes which are implementing Collection interface) can be converted to an array by calling toArray( ) method which is present in interface Collections.

Eg: -

```
LinkedList lnk = new LinkedList ( );  
Object [ ] = lnk.toArray( );
```

There are some advantages of converting collections to an array

- 1) To pass as an argument to a legacy method which do not accept Collections.
  - 2) To obtain faster processing times for certain operations
- All collections are eligible for cloning and serialization because all the collection classes are implementing two marker interfaces – interface Cloneable & interface Serializable.

**Collection framework has mainly 3 categories.**

1. List: - collection of objects with duplicates.
2. Set: - collection of objects without duplicates.
3. Map: - collection of key-value pairs.

**1. List: - is used to store multiple objects with duplicates:**

- ☞ List is an interface which extends Collection interface.
- ☞ List allows duplicates.
- ☞ All classes implementing List interface store values using index position.
- ☞ List has many concrete sub class implementations.
  - A). Vector.
  - B). ArrayList.
  - C). LinkedList.
  - D). Stack.

**A). Vector.**

- Vector is a legacy class (Java 1.0) and represents array data structure.
- Vector is re-engineered to fit into collection framework and is implementing List interface and hence supports both 1.0 method and 1.2 methods.
- Vector is same as ArrayList but Vector is synchronized and ArrayList is not synchronized.
- Vector is synchronized; multiple threads cannot access the Vector object concurrently. Only one thread can access the Vector object at a specific time.
- Since Vector is synchronized, it has low performance than ArrayList.
- Vector stores values sequentially, but the values can be accessed randomly.
- Vector can grow and shrink dynamically.
- Duplicates, null and dissimilar values are allowed.
- Before to java 1.2 there was an interface called Enumeration to visit the elements of Vector. But now we have two interfaces to visit the elements of Vector. They are Iterator and ListIterator.
- Values will be stored in the same order as inserted.
- Vector elements can be accessed randomly because it is implementing RandomAccess marker interface.

**Q** How will you get a synchronized list or thread safe list?

**A** You can use a Vector object if you need a synchronized list. But it is more advisable to store the values in ArrayList and make it synchronized with the help of the methods present in class Collections.

```
ArrayList employeeList = new ArrayList();  
employeeList.add(employee1);  
employeeList.add(employee2);  
List list = Collections.synchronizedList(employeeList);
```

**B). ArrayList.**

- ArrayList is a class used to store multiple objects.
- ArrayList uses array data structure internally to store the elements.
- ArrayList stores values sequentially, but the values can be accessed randomly.
- ArrayList can grow and shrink dynamically.
- Duplicate values are allowed.
- ArrayList is not synchronized.
- Values will be stored in the same order as inserted.
- The elements in ArrayList can be accessed by Iterator and ListIterator.
- ArrayList elements can be accessed randomly because it is implementing RandomAccess marker interface.



**Q** What are the advantages of ArrayList?

**A** It provides very fast iteration. It is better to choose ArrayList over LinkedList when fast iteration is required.

**Q** What are the drawbacks of ArrayList?

**A** Insertion & deletion operation at a specified index in ArrayList is time consuming and more complicated. It is better to choose LinkedList when too many insert and delete operations are being performed.

**C). LinkedList.**

- LinkedList uses linked list data structure internally i.e. it uses nodes.
- LinkedList is double linked.
- LinkedList stores values non-contiguously or randomly.
- LinkedList elements can be accessed sequentially only.
- It consumes lot of memory.
- Duplicate values are allowed.
- Values will be stored in the same order as inserted.
- LinkedList is not synchronized.
- The elements in LinkedList can be accessed by Iterator and ListIterator.
- LinkedList can be used for implementing basic stack and queue data structure.

**Q** What are the advantages of LinkedList?

**A** The main advantage of LinkedList is insertion and deletion operation is very fast and can be performed without affecting other elements.

**Q** What are the drawbacks of LinkedList?

- A** Following are the drawbacks of LinkedList:
- a) Iteration in LinkedList is bit slowly as compared to ArrayList.
  - b) Values are accessed sequentially in LinkedList, hence it is time consuming.
  - c) It consumes more memory.

**D). Stack**

- Stack is a legacy class.
- It is also re-engineered like Vector to implement List interface. Stack is a sub class of Vector.
- It is synchronized.
- It is generally used when we want to retrieve the elements of the collection in FILO order (First In Last Out).

**2. Set: - is used to store multiple objects without duplicates.**

☞ Set is an interface which is extending Collection.

- ☞ Set is used to store multiple objects without duplicates.
- ☞ Set has 3 different implementations and all of them can use only Iterator but not Enumeration and ListIterator.

A). TreeSet.

B). HashSet.

C). LinkedHashSet.

**A). TreeSet:-**

- TreeSet implements SortedSet which is sub interface of Set.
- TreeSet uses tree data structure to store values.
- Duplicates are not allowed.
- Values are stored in sorted, ascending order.
- Different types of objects are not allowed.
- Null value is not allowed.
- TreeSet can use only Iterator but not Enumeration and ListIterator

**B). HashSet.**

- HashSet stores the elements in a hash table.
- It uses the hash code of the object being inserted.
- Duplicates are not allowed.
- Values are stored in unsorted random order and will not be in the same order as inserted.
- Different types of objects are allowed.
- Null value is allowed.
- HashSet can use only Iterator but not Enumeration and ListIterator.
- HashSet provides very fast search rate because it uses hash code.

**Explain hash table, hash code and hashing technique.**

- Hash table is also a data structure which can be used for storing values.
- The values in the hash table are always stored after generating the hash code for that value.
- The mechanism of generating the hash code for a value is called hashing technique. In java hashCode() method provides the support for generating the hash code.

**hashCode() method in java**

- In java hashCode() method acts as a hashing function and generates the hash code for every object created.
- Every object in the Java system has a hash code. The hash code is a number that is usually different for different objects. JVM assigns unique hash code value to each object when they are created in the memory based on the object's memory address.

So, no two objects will have the same hash code when you use the default implementation of hashCode() method of class Object.

- The hashCode() method returns the hash code value of the object in terms of integer.
- It is this hash code value which will be used when we are storing objects in java collection classes that are using hashing like HashSet, HashMap, LinkedHashSet, and Hashtable etc.
- Whenever we add an element in the collection classes that use hashing like HashSet, HashMap etc. hashCode() is invoked implicitly which returns the hash code of the object.
- All these collection classes use the hash code value of an object to determine how the object should be stored or searched in the collection.
- When you put an object in a collection that uses hash codes, the collection uses the hash code of the object to decide in which bucket/slot the object should land.
- If we do not want to use existing hash code generation algorithm which is already available in JVM, u can use your own algorithm by overriding hashCode() method in your class.
- hashCode() is already overridden for String class and wrapper classes. It is generally overridden for our user defined classes when we add them into collections (which use hashing) like HashSet, LinkedHashSet, and HashMap etc .

**Q** What is the advantage of using hashing technique?

**A** Hashing technique provides very fast search rate.

**Rules for two objects to be equal in java:**

- If two objects are equal, they must have the same hash code. But if two objects have the same hash code they need not be equal.
- If two objects are equal, that is obj1.equals(obj2) is true then, obj1.hashCode() and obj2.hashCode() must return same hash code value.
- If you want to treat two objects equal then you must override both the hashCode() and equals() method inherited from class Object.

**Q** What is the best way to override hashCode() method?

**A** The best way to override hashCode() method is to use all the values of the instance variables of the object and then generate the hash code. The hash code value should be same for two objects whenever an identical object is created. (An identical object is an object which has the same instance variable values).

**Q** Explain how hashing works?

**A** Hashing retrieval is a two step process.

- a) First find the right bucket/slot by using hashCode() method.
  - b) Second find the right element by using equals() method.
- So to put an element into the collection classes, it should have these two methods.

**Q** Why should you override hashCode() and equals() method ?

**A** We should override both these methods so that they can be used efficiently for searching and storing in collection classes which use hashing.

**Q** What happens if you don't override hashCode() method ?

**A** The collection classes which use hashing technique to store elements will allow duplicates to be inserted. And also we can't search a particular object or element in the collection.

The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of the same type can ever be considered equal.

**Q** What happens if you don't override equals()?

**A** The default behavior of equals() method always checks whether both the reference variables are referring to the same object. If we want to compare contents of two objects then we have to override equals() method.

**Q** How does searching work in HashSet ?

**A** The search object and the object in the collection must have both identical hashCode values and return true for the equals() method.

**C). LinkedHashSet.**

- LinkedHashSet has a combined implementation of hash table and linked list.
- LinkedHashSet stores elements in a hash table and links them with double linked list.
- Values are stored in the same order as inserted and are unsorted.
- Different types of objects are allowed.
- Null value is allowed.
- Duplicates are not allowed.
- LinkedHashSet can use only Iterator but not Enumeration and ListIterator.

**Q** When do you use a LinkedHashSet?

**A** Use this class when you want fast searches and also when you want the elements to be accessed in the same order as inserted.

**Q** Explain the different scenarios you will use the different collection classes.

| Collection class | Storing Element  | Performance  | Data Structure                           |
|------------------|--|--|--|
| ArrayList        | It allows duplicates, null and dissimilar data.<br>It stores values in the same order as inserted.                                 | Provides very fast iteration.  | array.                                   |
| LinkedList       | It allows duplicates, null and dissimilar data.<br>It stores values in the same order as inserted.                                 | Should be used when too many insertions and deletions are performed    | linked list.                             |
| TreeSet          | It allows only similar type of data.<br>It doesn't allow duplicates, null and dissimilar data.<br>It stores values in sorted order | Should be used when sorting is required.                               | tree.                                    |
| HashSet          | It allows dissimilar and null values.<br>It doesn't allow duplicates.<br>It stores values in random order                          | Search rate is very fast.  | hash table.                              |
| LinkedHashSet    | It allows dissimilar and null values.<br>It doesn't allow duplicates.<br>It stores values in same order as inserted.               | Search rate is fast and can access elements in same order as inserted. | Combination of hash table & linked list. |

### 3. Map: - is used to store key-value pairs.

- ☞ Map is also part of collections framework.
- ☞ A map is an object which stores *key/value* pairs.
- ☞ You need a key and a value to store in a map.
- ☞ Both key and value are objects.
- ☞ Given a key, you can find its value.
- ☞ The keys must be unique, but you can have duplicate values.
- ☞ Some maps allow null keys and null values, others cannot.
- ☞ There are many different implementations of the Map interface.

A). TreeMap.

B). HashMap.

C). LinkedHashMap.

D). Hashtable.

E). Properties

A). TreeMap.

- TreeMap is implementing SortedMap interface.

- TreeMap can store *key/value* pairs.
- TreeMap uses tree data structure to store the keys.
- The keys are stored in sorted order.
- Only similar types of keys are allowed.
- Different types of values are allowed.
- Duplicate keys are not allowed but duplicate values are allowed.
- Null keys are not allowed but null values are allowed.
- TreeMap is not synchronized.

**B). HashMap.**

- HashMap is implementing Map interface.
- HashMap can store key/value pairs.
- HashMap uses hash table data structure to store the keys.
- The keys will be unordered or undefined order.
- Null keys and null values are allowed.
- Different types of keys and values are allowed.
- Duplicate values are allowed.
- HashMap is not synchronized.

**C). LinkedHashMap.**

- LinkedHashMap is implementing Map interface.
- LinkedHashMap can store key/value pairs.
- LinkedHashMap store the keys in hash table data structure and links them with double linked list.
- The keys will be stored in the insertion order and are unsorted.
- Null keys and null values are allowed.
- Different types of keys and values are allowed.
- Duplicate values are allowed.
- LinkedHashMap is not synchronized.

**D). Hashtable.**

- Hashtable can store key/value pairs.
- It uses hash table data structure to store the keys.
- Hashtable is a legacy class (Java 1.0) and it was extending Dictionary class.
- Hashtable is re-engineered to fit into collection framework and is implementing Map interface and hence supports both 1.0 methods and 1.2 methods.
- Hashtable is same as HashMap, but Hashtable is synchronized and HashMap is not synchronized.
- Hashtable is synchronized; multiple threads cannot access the Hashtable object concurrently. Only one thread can access the Hashtable object at a specific time.
- Since Hashtable is synchronized, it has low performance than HashMap.
- Before to java 1.2 there was an interface called Enumeration to visit the elements of Hashtable. But now we have Iterator interface to visit the elements of Hashtable.
- The keys will be unordered or undefined order.
- Duplicate values are allowed.

- Different types of keys and values are allowed.
- Null keys and null values are not allowed

#### E). Properties

- Properties can store key/value pairs.
- It uses hash table data structure to store the keys.
- Properties is a legacy class (Java 1.0) and it was extending Dictionary class.
- Properties is re-engineered to fit into collection framework and is implementing Map interface and hence supports both 1.0 methods and 1.2 methods.
- Properties is synchronized; multiple threads cannot access the Properties object concurrently. Only one thread can access the Properties object at a specific time.
- Since Properties is synchronized, it has low performance.
- Before to java 1.2 there was an interface called Enumeration to visit the elements of Properties. But now we have Iterator interface to visit the elements of Properties.
- Properties is the only collection class which can interact with file concept directly. It can load the key/value pair from file and store also.
- This class is extensively useful and is used in technologies like Struts, JSF etc to read key/value pairs from .properties file. The key/value read from a file will be always of type String.
- The keys will be unordered or undefined order.
- Duplicate values are allowed.
- Different types of keys and values are allowed.
- Null keys and null values are not allowed.

## ***Differences and Similarities:***

### ***Difference between Array and ArrayList***

| Array   | ArrayList   |
|---|---|
| Arrays are fixed in length. Once array is created with some size, you can't alter the size. | ArrayList is expandable. It can increase and decrease dynamically.      |
| Arrays can store both primitive data and objects.   | ArrayList can store only objects.                                       |
| Array cannot store dissimilar data.(Except the case of Object array).                       | ArrayList can store dissimilar data.                                    |
| We can only add elements into array.  | We can add and remove elements in ArrayList                             |
| We have to use for loop to visit every element in an array.                                 | We can use Iterator or ListIterator to visit every element in ArrayList |
| We cannot insert an element into the specified index of an array.                           | We can add elements into the specified index of ArrayList.              |

**Similarity between Array and ArrayList**

- Both store values sequentially using indexing notation.
- Both can access elements randomly.
- Both allow duplicate values.
- Both can store multiple values.
- Insertion & deletion operation are bit slow in array and ArrayList.

***Difference between ArrayList and Vector***

| ArrayList   | Vector   |
|---|--|
| ArrayList is a new class (Java 1.2) and is part of collections framework.   | Vector is a legacy class (Java 1.0) and is reengineered to fit into collections framework.   |
| Synchronization: - ArrayList is not synchronized by default i.e. multiple threads can access ArrayList object concurrently and there are no performance issues. | Synchronization: - Vector is synchronized by default i.e. only one thread can access Vector object at a time and it has low performance. |
| To visit the elements we have to use Iterator or ListIterator   | To visit the elements we can use Enumeration, Iterator or ListIterator.  |

**Similarity between ArrayList and Vector**

- Both store values using indexing notation.
- Both allow duplicate, null and dissimilar values.
- Both are eligible for cloning and serialization.
- Both can use Iterator and ListIterator.

***Difference between ArrayList and LinkedList***

| ArrayList   | LinkedList  |
|---|---|
| ArrayList uses array data structure.              | LinkedList uses linked list data structure.                                 |
| All elements are stored contiguously.             | All elements will be stored non-contiguously.                               |
| Values can be accessed randomly and is very fast. | Values are accessed sequentially in LinkedList, hence it is time consuming. |
| Iteration is fast.                                | Iteration is slow.  |
| Insertion and deletion operation is very slow.    | Insertion and deletion operation is very fast.                              |
| ArrayList consumes less memory.                   | LinkedList consumes more memory.  |

**Similarity between ArrayList and LinkedList**

- Both are not synchronized.
- Both allow duplicate values.
- Both are eligible for cloning and serialization.
- Both can use Iterator and ListIterator and cannot use Enumeration



***Difference between HashSet and TreeSet***

| HashSet                                | TreeSet   |
|--|---|
| HashSet stores elements in hash table. | TreeSet stores elements in tree data structure. |
| Elements are unordered or undefined.   | Elements are sorted.                            |
| We can add different types of objects. | We can add only similar objects.                |
| Null values are allowed (only one).    | Null values are not allowed.                    |
| It provides very fast search rate.     | It provides very slow search rate.              |

**Similarity between HashSet and TreeSet**

- Both are not synchronized.
- Both do not allow duplicate values.
- Both are eligible for cloning and serialization.
- Both can use only Iterator but not ListIterator and Enumeration

***Difference between HashSet and LinkedHashSet***

| HashSet                                | LinkedHashSet   |
|--|---|
| HashSet stores elements in hash table. | LinkedHashSet stores elements in a hash table and links them with double linked list. |
| Elements are unordered or undefined.   | Elements are stored in the same order as inserted.                                    |

**Similarity between HashSet and LinkedHashSet**

- Both do not allow duplicate values.
- Both allow different types of objects.
- Null values are allowed.
- Both provide very fast search rate.
- The elements must have overridden hashCode() & equals() methods for both the collections.
- Both are not synchronized.
- Both are eligible for cloning and serialization.
- Both can use only Iterator but not ListIterator and Enumeration.

***Difference between HashMap and TreeMap***

| HashMap   | TreeMap   |
|---|---|
| HashMap uses hash table data structure to store its keys. | TreeMap uses tree data structure to store its keys. |
| Keys are unordered.                                       | Keys are sorted.                                    |
| Different types of key elements are allowed.              | Only similar types of key elements are allowed.     |
| Null keys are allowed.                                    | Null keys are not allowed.                          |
| It provides very fast search rate.                        | It is relatively slow.                              |

**Similarity between HashMap and TreeMap**

- Both are used to store key/value pair.
- Both do not allow duplicate keys.
- Both allow null values.
- Both allow duplicate values.
- Both are not synchronized.
- Both are eligible for cloning and serialization.
- Both can use only Iterator but not ListIterator and Enumeration.

***Difference between HashMap and LinkedHashMap***

| HashMap                                | LinkedHashMap   |
|--|---|
| HashMap stores elements in hash table. | LinkedHashMap stores elements in a hash table and links them with double linked list. |
| The keys are unordered or undefined.   | The keys will be stored in the insertion order and are unsorted.                      |

**Similarity between HashMap and LinkedHashMap**

- Both are used to store key/value pair.
- Both do not allow duplicate keys.
- Both allow dissimilar keys.
- Both allow null keys.
- Both allow null values.
- Both allow duplicate values.
- The elements which are used as key must have overridden hashCode() & equals() methods.
- Both are not synchronized.
- Both are eligible for cloning and serialization.
- Both can use only Iterator but not ListIterator and Enumeration.

***Difference between HashMap And Hashtable***

| HashMap   | Hashtable  |
|---|--|
| HashMap is a new class.   | Hashtable is a legacy class.   |
| HashMap is not synchronized, i.e. many threads can access the same object concurrently. | Hashtable is synchronized i.e. only one thread can access the object at the same time. |
| Null key and null values are allowed.   | Null keys and null values are not allowed.   |
| HashMap can use only Iterator.  | Hashtable can use both Iterator and Enumeration.                                       |

**Similarity between HashMap and Hashtable**

- Both are used to store key/value pair.
- Both store keys in unordered way.

- Both do not allow duplicate keys.
- Both allow dissimilar keys.
- Both allow duplicate values.
- The elements which are used as key must have overridden hashCode() & equals() methods.
- Both are eligible for cloning and serialization.

### ***Accessing the elements of a collection or traversing***

- Collection framework provides 3 interfaces to visit the elements of a collection.
- 1) Enumeration: - Can be used with only legacy classes. It was added in Java 1.0
  - 2) Iterator: - Can be used with all classes. It was added in Java 1.2.
  - 3) ListIterator: - Can only be used with classes which implement List interface. (Vector, ArrayList and LinkedList). It was also added in Java 1.2.

### ***Difference between Iterator and ListIterator***

| Iterator   | ListIterator   |
|--|--|
| Using Iterator you can visit the elements of a collection only in the forward direction. | Using Iterator you can visit the elements of a collection both in the forward and reverse direction. |
| We can remove the elements in collection using Iterator.                                 | We can add, remove and replace the elements using ListIterator.                                      |
| It is available to all the classes in collection frame work.                             | It is available only to classes which implement List interface. (ArrayList, LinkedList, Vector)      |

#### **Similarity between Iterator and ListIterator**

- Both can be used to iterate only in forward direction.
- Both allow removing an element from a collection.

### ***Difference between Iterator and Enumeration***

| Iterator  | Enumeration  |
|---|--|
| Iterator is a collection framework interface.   | Enumeration is a legacy interface.   |
| We can remove the elements in collection using Iterator.                                    | We cannot remove the elements using Enumeration.   |
| It can be used to iterate through any collection (Vector, ArrayList, HashMap, TreeMap etc). | It can be used to iterate through only legacy collection classes (Vector, Hashtable, etc). |

#### **Similarity between Iterator and Enumeration**

- Both are used to iterate only in forward direction.

***Difference between Enumeration and ListIterator***

| Enumeration  | ListIterator   |
|--|--|
| Enumeration is a legacy interface.   | ListIterator is a collection framework interface.  |
| We can iterate through only in forward direction through a collection.                     | We can iterate both in forward and reverse direction through a collection.   |
| It can be used to iterate through only legacy collection classes (Vector, Hashtable, etc). | It can be used to iterate through any collection classes which implements List interface (Vector, ArrayList, etc). |
| We cannot add, remove or replace the elements of collection using Enumeration.             | We can add, remove and replace the elements of a collection using ListIterator.                                    |

**Similarity between Enumeration and ListIterator**

- Both can be used to iterate only in forward direction.

***Working with arrays using class Arrays***

- Arrays class is present in java.util package.
- This class provides various methods that are useful when working with arrays.
- Arrays class provides various methods to search, sort, fill etc in an array.

**Q** How to convert an array to ArrayList?

**A** We can convert an array to ArrayList by using Arrays.asList(ia).

```
Integer ia[] = new Integer[5];
ia[0] = new Integer(102);
ia[1] = new Integer(104);
ia[2] = new Integer(103);
// asList() converts array to ArrayList
List list = Arrays.asList(ia);
System.out.println("Value in list" + list);
```

**Q** How to do sorting in arrays?

**A** We can do sorting in arrays by using the sort() method of class Arrays.

- 1) Arrays.sort(arr);
- 2) Arrays.sort(arr, mycomparator);

## ***Sorting in List***

- Comparable and Comparator both are used for sorting the collection.
- Comparable provides only one sorting sequence whereas Comparator provides multiple sorting sequences.

### **Rules for Sorting:-**

- 1) Objects of different or dissimilar type are not comparable.
- 2) We can sort only when the elements in the collection are of similar type.
- 3) We cannot sort when the elements in the collection are of dissimilar type.
- 4) The sorting algorithm uses the comparison logic present in the compareTo() method of Comparable interface or compare() method of Comparator interface.

## ***Sorting with interface Comparable***

- Comparable interface is present in java.lang package.
- A class implementing the Comparable interface can compare ITSELF with another class.
- It supports only one sorting sequence.
- Comparable interface contains 1 method

```
int compareTo(Object obj)
```

## ***Sorting with interface Comparator***

- Comparator interface is present in java.util package.
- Comparator defines how two objects can be compared in a completely new class.
- It is about moving the responsibility for doing the comparison from the class that is being compared, to another class.
- It supports multiple sorting sequences.
- It can be used for sorting 3rd party classes or classes which you can't modify(final classes).
- Comparator interface contains 2 methods

```
int compare(Object obj1, Object obj2)  
boolean equals(Object obj)
```

***Difference between Comparable and Comparator:-***

| Comparable  | Comparator   |
|---|--|
| A comparable object can compare itself with another class.  | A comparator contains how two objects can be compared in a different class.  |
| It is an interface in java.lang package.  | It is an interface present in java.util package  |
| We can create only one sort sequence.   | We can implement many sort sequences.  |
| We have to override this method.<br><br>int compareTo(Object o1)<br><br>This method compares this object with o1 object and returns an integer. Its value has following meaning<br><br>positive – this object is greater than o1<br>zero – this object equals to o1<br>negative – this object is less than o1 | We have to override this method.<br><br>int compare(Object o1, Object o2)<br><br>This method compares o1 and o2 objects. And returns an integer. Its value has following meaning.<br><br>positive – o1 is greater than o2<br>zero – o1 equals to o2<br>negative – o1 is less than o2 |
| Collections.sort(List)  | Collections.sort(List, Comparator)   |
| Here objects will be sorted on the basis of compareTo() method.   | Here objects will be sorted on the basis of compare() method in Comparator.  |
| Many built in classes like String, all wrapper classes, Date, Calendar implement it.  | It is generally implemented to compare 3rd party classes or our user defined classes.  |

**Q** What is the difference between the sort(List o) and sort(List o, Comparator c) methods present in class Collections ?

**A** Invoking the one argument sort(List o) method means the list element's compareTo() method decides the order. So the elements in the list must implement the Comparable interface.

Invoking the sort(List o, Comparator c) means the list element's compareTo() will not be called, and the Comparator's compare() method will be used instead. That means the elements in the list do not need to implement the Comparable interface.

***class Collections***

- The collections framework supports several algorithms that allow us to operate on collections.
- We can use these algorithms to sort, shuffle, manipulate and search a set of elements in a collection.
- Some of the algorithms available in collections framework are:
  - 1) Sorting: - enables to arrange the elements of a list in a certain order.
  - 2) Shuffling: - shuffles the order of elements in a collection randomly.
  - 3) Manipulating: - provides algorithms to perform operations such as reverse, copy, swap, etc.
  - 4) Searching: -allows searching an element in a collection.

- These algorithms are present in Collections class.
- The Collections class also provides methods for getting a synchronized or unmodifiable list/set/map.
- This is especially useful when you want a read only set/list/map.

```
// can get a unmodifiable list  
List list = Collections.unmodifiableList(employeeList);
```

```
// can get a synchronized list  
List list3 = Collections.synchronizedList(employeeList);
```

- It is more recommended to get a synchronized list/set/map in this way (if needed) rather than depending on legacy classes.

## ***New features added in java 5 to support Collections***

### **1. Autoboxing:**

- ☞ Autoboxing is a process of converting primitives to objects and objects to primitives automatically.
- ☞ Earlier boxing & unboxing was done explicitly by the developer while adding and removing a primitive from a collection because all collection classes cannot store primitives and can store only objects.
- ☞ Autoboxing is especially useful when we put values in collection and retrieve values from a collection. We will not have to do the conversions or casting.

### **2. Enhanced for loop / for each loop**

Syntax: - for(datatype variableName: Collection)

Eg: -

```
for(String x:ar1){  
    System.out.println(x);  
}
```

### **3. Generics**

- ☞ Generics provide type safety to collections( Discussed in detail in the next chapter)
- ☞ Earlier to java 5 the collection classes can work with any type of Object.
- ☞ After the introduction of generics in java 5 collection classes can now be used with type safety.
- ☞ Also earlier to java 5 all collection classes were non generic classes & from java 5 all collection classes were re-engineered to generic classes.

Problem without generics are: -

- 1) When you are adding an element into a collection compiler ignores the types and allows different type of elements to be added into the collection.
- 2) When you are getting an element out of a collection then many if conditional statements and many type casting is required. (When we take an element out of a collection we must

- do casting. Because of this inconvenience it is unsafe i.e. when a different type of an element is coming out from collection and we are casting to a different type we get a runtime exception called ClassCastException).
- 3) Generic provides a way to communicate the type of the collection to the compiler so that it can be checked by the compiler.

Advantage of using generics:-

- 1) Generic help you to restrict the type of element you add into collection during compilation time and thus provides type safe collections.
- 2) No conditional checks, no casting required, when you get an element out of a collection. What type of element goes in when using generic, come out as element of same type.

#### 4. New API addition (class PriorityQueue and interface Queue)

**From Java 5 Collection framework has mainly 4 categories.**

- 1) List: - collection of objects with duplicates.
- 2) Set: - collection of objects without duplicates.
- 3) Map: - collection of key-value pairs.
- 4) Queue:- Orders values based on how they are to be processed.

#### ***class PriorityQueue and interface Queue:-***

- PriorityQueue is class implementing Queue interface and is present in java.util package.
- Queue & PriorityQueue were added in java 5 version.
- PriorityQueue orders values based on priority.
- It stores the values in a heap (A heap is a new data structure and uses binary tree).
- PriorityQueue is a class used to store multiple objects.
- Duplicate values are allowed.
- It allows only comparable objects.
- Dissimilar values are not allowed.
- Null values are not allowed.
- The elements in PriorityQueue can be accessed by only Iterator.
- The elements should be always accessed by using poll() method while iterating otherwise the values will be accessed in an undefined order.
- PriorityQueue stores values in an undefined order, but when the elements are pulled from the queue it orders the values from the queue in a sorted order. Default is natural order or ascending order but if a custom comparator is used the values can be ordered in descending order.
- PriorityQueue is not synchronized.
- If a thread safe priority queue is required use class PriorityBlockingQueue.

**Q** What is the basic difference between basic queue and priority queue?

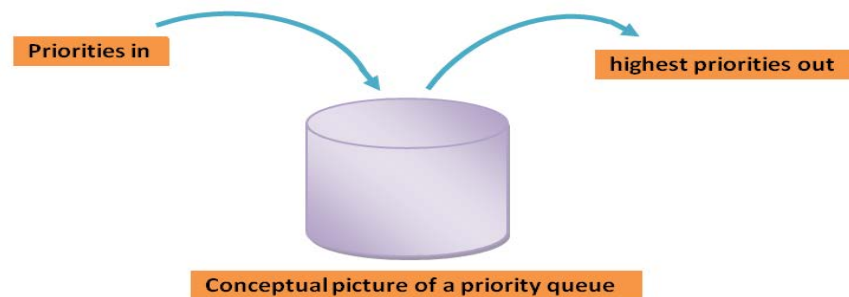
**A** A basic queue always orders the values in the First In First Out order (FIFO). A priority queue orders values based on priority.



**Q** What is the default order used by PriorityQueue for giving a value?

**A** The PriorityQueue by default gives highest priority to minimum value / ascending order / natural order in the queue for giving a value. The highest priority can be changed to maximum value / descending order by using a custom comparator.

### **Explanation of priority queue**



- 1) A priority queue is a kind of bag that holds priorities.
- 2) You can put one in, and you can take out the current highest priority. (Priorities can be any Comparable values like Integers, String, etc.)
- 3) Default highest priority is minimum value / ascending order / natural order.
- 4) In a priority queue, an element with high priority is served before an element with low priority. If two elements have the same priority, they are served according to their order in the queue.
- 5) A priority queue is different from a "normal" queue, because instead of being a "first-in-first-out" data structure, values come out in order by priority.
- 6) It does not allow insertion of non-comparable objects.
- 7) A priority queue can be implemented using many of the data structures that we've already studied (an array, a linked list). However, those data structures do not provide the most efficient operations. To make all the operations very efficient, priority queue is using a new data structure called a heap (A heap is a binary tree).

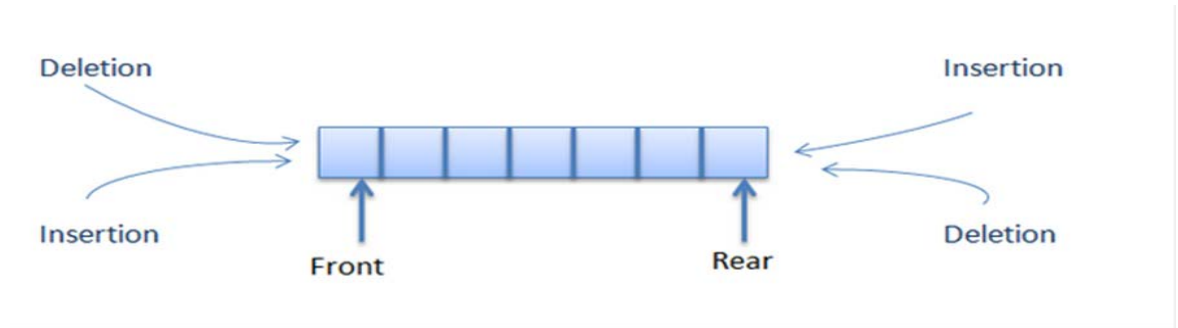
### ***New API addition in java 6 to support collections***

1. Deque interface & ArrayDeque class
2. NavigableSet interface
3. NavigableMap interface

#### **1. Deque interface & ArrayDeque class**

- Deque interface and ArrayDeque class were added in java 6.
- Both are present in java.util package.
- ArrayDeque is a class which implements Deque interface.
- Deque interface supports double ended queue.
- Deque also supports basic queue (FIFO) and stack (LIFO). Deque interface has push() & pop() to support stack.
- It is more recommended to use ArrayDeque as stack rather the legacy Stack class.

- ArrayDeque class is likely to be faster than Stack when used as a stack, and faster than LinkedList when used as a queue.
- ArrayDeque is using array structure.
- ArrayDeque allows dissimilar values.
- ArrayDeque does not allow null values.
- ArrayDeque is not synchronized.



## 2. NavigableSet interface:-

- NavigableSet interface was added in java 6.
- It is extending SortedSet interface.
- NavigableSet is implemented by class TreeSet.
- NavigableSet provides various methods to navigate or search through the TreeSet collection.

## 3. NavigableMap interface:-

- NavigableMap interface was added in java 6.
- It is extending SortedMap interface.
- NavigableMap is implemented by class TreeMap.
- NavigableMap provides various methods to navigate or search through the TreeMap.

## Generics (Java 1.5)

- Generics is a new feature which was added in java 1.5
- The main benefit of generics is it provides type safety.
- The main advantages of generics are.
  - 1) Helps in doing type checking at compilation time.
  - 2) No need of type casting because it will be done implicitly.
  - 3) Converts runtime errors to compile time errors and avoids ClassCastException during runtime.
  - 4) When we use generics with methods it simplifies method overloading. A single method with a common logic can be used by many data types in a type safe manner.
- A class which has a type parameter is called generic class.

Syntax for declaring a generic class:-

```
class classname<type parameter list> {  
  
}
```

Syntax for creating an instance of a generic class in java 1.5 & 1.6:-

```
classname<type argument list> ref-variable = new classname<type argument  
list> (constructor argument list)
```

- ☞ Syntax for creating an instance of a generic class in java 1.7:-  
(Just use diamond operator <> while creating an object )

```
classname<type argument list> ref-variable = new classname<>(constructor  
argument list);
```

- When creating an instance of a generic class the type argument must be of class type and not primitive type.

```
// valid because type argument is class type  
Hello<String> h1 = new Hello<String>();  
Hello<Integer> h2 = new Hello<Integer>();
```

```
// invalid because type argument is primitive type  
Hello<int> h3 = new Hello<int>();
```

- Generics code is available only in source code and not in byte code.
- The process of removing generic type information is called erasure. It is done by the compiler during compilation time.
- Generic is non-reified (not available at run time).

### ***Bounded type parameters.***

- Bounded type parameters are used for restricting the type arguments.
- When we are writing a generic class we can restrict the type argument that can be passed to the type parameter.
- When specifying a bounded type parameter we use upper bound.
- When a bounded type parameter is specified for a generic class, then the type argument used should either be the specified super class or its sub class.

```
class A {
    void show1() {
        System.out.println("Inside A show1");
    }
}

class B extends A {
}

class C extends B {
}

// bounded type parameter. Sample can use only A or sub class of A.
class Sample<T extends A> {
    T ob;

    Sample(T t) {
        ob = t;
    }

    T getOb() {
        return ob;
    }
}
```

### ***Wild card arguments***

- Wild card arguments are used when you are specifying a parameter for a method.
- There are three types of wild card arguments upper bound, lower bound and unbounded.
- Upper bound is supported with “**extends**” keyword and lower bound is supported with “**super**” keyword.

```
class Hello {
    // unbounded
    void m1(Sample<?> t) {
        A a = t.getOb();
        a.show1();
    }
}
```

```
    }

    // A and anything which extends A
    // extends is UPPER BOUND
    void m2(Sample<? extends A> t) {
        A a = t.getOb();
        a.show1();
    }

    // B and anything which is super to B
    // super is LOWER BOUND
    void m3(Sample<? super B> t) {
        A a = t.getOb();
        a.show1();
    }
}
```

## **Exploring more classes in java.util package**

- This package is having many utility classes that provide support to data structures, string tokenizing, internationalization, data formatting, time and date formatting, generating random numbers, etc.
  - 1) StringTokenizer.
  - 2) Locale.
  - 3) ResourceBundle.
  - 4) Date.
  - 5) Calendar.
  - 6) TimeZone.
  - 7) SimpleTimeZone
  - 8) Scanner (was added in java 1.5).
  - 9) Formatter (was added in java 1.5).

### ***StringTokenizer***

- StringTokenizer is used to break the big String into small tokens (sub-strings) with the given delimiter ( ; , : / ) etc.
- Usually split() method of class String is preferred over StringTokenizer class.

### ***Internationalization (i18n)***

- Internationalization is the ability to support multiple languages.
- Internationalization is one of the powerful concepts of java if you are developing an application and want to display messages, currencies, date, time etc. according to the specific country or language.
- Following are the list of things that differ from one country or region to another.
  - 1) Messages
  - 2) Numbers
  - 3) Currencies
  - 4) Dates
  - 5) Times
  - 6) Phone Numbers

### ***class Locale***

- An object of Locale class represents a geographical region. The Locale object identifies a particular language and country.
- This object can be used to get the locale specific information such as country name, language, etc.
- A Locale object is only an identifier and is not used individually.
- A Locale object is generally used with locale sensitive classes like ResouceBundle, NumberFormat, DateFormat, , etc to support internationalization of messages, currency, number, date, etc.

**Q** What is a locale sensitive class ?

**A** If a class varies its behavior according to Locale, it is said to be locale-sensitive. Example for locale sensitive object is ResourceBundle, NumberFormat, Calendar, DateFormat, etc. These classes are used along with the Locale class to support i18n of messages, currency, number, date, etc.

### ***class ResourceBundle***

- The ResourceBundle class is used to internationalize the messages. In other words, we can say that it provides a mechanism to globalize the messages.
- The hardcoded message is not considered good in terms of programming, because it differs from one country to another.
- So we use the ResourceBundle class to globalize the messages. The ResourceBundle class loads the information from the properties file that contains the messages.
- Conventionally, the name of the properties file should be filename\_languagecode\_countrycode . It contains only key value pair. The key is used in the java program to get the value.
  - 1) The ISO standard language code is lower case and 2 letter.
  - 2) The ISO standard country code or region code is upper case and 2 letter.
  - 3) The file name is also known as base name or family name.

for example:-

MyMessage\_en\_US.properties. (for English language in US)

MyMessage\_hi\_IN.properties. (for hindi language in India)

MyMessage\_ta\_IN.properties. (for tamil language in India)

MyMessage\_de\_DE.properties. (for german language in Germany)

### ***Date***

- When java 1.1 version was released many methods contained in earlier version of Date class (java 1.0 version) were moved to class Calendar and class DateFormat. Due to that, class Date contains many deprecated methods.
- java.util.Date provides information of date, time in seconds and time zone.
- java.util.Date is having two sub classes:-
  - 1) java.sql.Date :- provides only date.
  - 2) java.sql.Timestamp:- provides date and time in milliseconds.
- java.util.Date is used for displaying (example in JSP or jsf, etc).
- But both java.sql.Date and java.sql.Timestamp are used for storing in database.
- Always convert a String or java.util.Date to java.sql.Date and then store in database.

### ***TimeZone***

- It is present in java.util package.
- It allows you to work with various time zones of the world.

### ***Calendar and GregorianCalendar***

- Both are present in java.util package.
- Calendar is an abstract class and cannot be instantiated.
- Calendar is generally used for scheduling.
- The Calendar fields can be changed using three methods set() , add(), roll().
- GregorianCalendar is a sub class of Calendar.
- Calendar is used for traversing but finally Calendar will be converted to java.util.Date and used
- Following are the different ways to instantiate Calendar and GregorianCalendar.

1)

```
// creating Calendar and getting current date and time
Calendar cal = Calendar.getInstance();

// converting Calendar to util.Date
Date d = cal.getTime();
```

2)

```
// creating GregorianCalendar based on default locale.
GregorianCalendar gc = new GregorianCalendar();

// converting Calendar to util.Date
Date d = cal.getTime();
```

3)

```
// Creating calendar for our required date
GregorianCalendar gc = new GregorianCalendar(1990,1,28,13,24,56);

// converting Calendar to util.Date
Date d = gc.getTime();
```

### ***Scanner***

- It is present in java.util package and was introduced in java 1.5 version.
- There are various classes BufferedReader, FileReader, etc provided in Java to read input from various devices like keyboard, file, etc.
- But the problem with all these classes is we have to read the values first and then do the data conversion explicitly (like String to int, String to double explicitly).
- This process is increasing the performance and size of the program also increases.
- To overcome these problems Scanner class was introduced in java 1.5
- In this class various methods have been provided to read the values in the required format (no need of doing any explicit data conversions) from various devices like keyboard, file, etc.
- Some of the methods present in Scanner class to read values are:-
  - 1) nextInt() :- reads as integer values.
  - 2) nextDouble() :- Reads as double values
  - 3) next() :- Reads as String values
- When reading a file the Scanner class breaks the input into tokens and uses whitespace as the default delimiter.



## Formatter

- It is present in java.util package and was introduced in java 1.5 version.
- It provides various format conversions to display various data types like strings, numbers, date and time.

### Example for formatting different data types

```
fmt.format("%c - %f - %b %n %d - %o - %x", 'c', 99.99, true, 25, 25, 25);  
System.out.println(fmt);
```

### Table of format specifiers:-

| Format Specifier | Conversion      |
|------------------|-----------------|
| %s               | String          |
| %d               | Decimal Integer |
| %f               | Floating value  |
| %c               | Character       |
| %x               | Hexadecimal     |
| %o               | Octal Integer   |
| %t               | Date and time   |
| %n               | New line        |

- ☞ %t is used for formatting date & time
- ☞ The %t specifier requires the following suffixes for displaying date and time.

### Example for formatting date and time

```
Formatter fmt = new Formatter();  
Calendar cal = Calendar.getInstance();  
  
//for formatting date and time, use suffix with format specifier %t  
fmt.format("%tr %n %tc %n %tl:%tM", cal, cal, cal, cal);  
System.out.println(fmt);
```

### Table of format suffixes for date & time:-

| Suffix | Conversion                         |
|--------|------------------------------------|
| %r     | hh:mm:ss (12 hour format)          |
| %c     | Day month date hh:mm:ss tzone year |
| %l     | Hour (01 to 12)                    |
| %M     | Minutes (00 to 59)                 |

## Exploring java.text package

- This package is having many classes that provide support to data formatting, time & date formatting when we want to do internationalization.
  - 1) NumberFormat.
  - 2) DateFormat.
  - 3) SimpleDateFormat.

### ***class NumberFormat***

- NumberFormat class is present in java.text package.
- It is a locale sensitive class and supports internationalization.
- The NumberFormat class provides methods to format the currency and number according to the locale.
- The `getCurrencyInstance()` method of the NumberFormat class returns the instance of the NumberFormat class. This is done when you want to do i18n with currency.
- The `getNumberInstance()` method of the NumberFormat class returns the instance of the NumberFormat class. This is done when you want to do i18n with number.

#### Example for i18n with currency:-

```
public class CurrencyInternationalizationDemo {  
  
    static void printCurrency(Locale locale){  
        double dbl=10500.32;  
        NumberFormat formatter=NumberFormat.getCurrencyInstance(locale);  
        String currency=formatter.format(dbl);  
        System.out.println(currency+" for the locale "+locale);  
    }  
  
    public static void main(String[] args) {  
        printCurrency(new Locale("kn", "IN"));  
        printCurrency(Locale.UK);  
        printCurrency(Locale.US);  
        printCurrency(Locale.FRANCE);  
    }  
}
```

#### Example for i18n with number:-

```
public class NumberInternationalizationDemo {  
    static void printNumber(Locale locale){  
        double dbl=105000.3245;  
        NumberFormat formatter=  
            NumberFormat.getNumberInstance(locale);  
        String number=formatter.format(dbl);  
        System.out.println(number+" for the locale "+locale);  
    }  
}
```

```
    }

    public static void main(String[] args) {
        printNumber(new Locale("kn", "IN"));
        printNumber(Locale.UK);
        printNumber(Locale.US);
        printNumber(Locale.JAPAN);
        printNumber(Locale.FRANCE);
    }
}
```

## ***DateFormat***

- It is present in java.text package.
  - It is a locale sensitive class and supports internationalization.
  - You can get the instance of DateFormat either by calling getDateInstance() or getTimeInstance().
  - This will give a format based on a Locale.
  - DateFormat has two important methods:-
    - 1) format():- converts Date to String. (Date to String is for displaying).
    - 2) parse():- converts String to java.util.Date. (String to Date is for inserting into database)
- 1) **format():**- It formats the date and time according to a locale and it supports internationalization.

### **Example for i18n with date and converting Date to String:-**

```
// for formatting date use getDateInstance() method and then use
// format() method
// formatting date for US country

DateFormat df = DateFormat.getDateInstance(DateFormat.FULL, new
                                                Locale("en", "US"));
String str = df.format(new Date());
```

### **Example for i18n with time and converting Date to String:-**

```
// for formatting time use getTimeInstance() method and then use
// format() method
// formatting time for US country

DateFormat df = DateFormat.getTimeInstance(DateFormat.FULL, new
                                                Locale("en", "US"));
String str = df.format(new Date());
```

- 2) **parse():**- converts String to java.util.Date.
- ```
String str = "11-March-2009";
```

```
DateFormat df = new SimpleDateFormat("dd-MMM-yyyy");  
  
// Converts String to Date  
Date date = df.parse(str);
```

### ***SimpleDateFormat***

- It is present in java.text package and is a sub class of java.text.DateFormat.
- It is a locale sensitive class and supports internationalization.
- The DateFormat class provides methods to format the date and time according to a locale.
- If you want a format other than predefined format use SimpleDateFormat which extends DateFormat.
- It helps you to display date and time in your own customized way.

#### **Example for customized formatting**

```
// create your own custom pattern and then format()  
SimpleDateFormat sdf = new SimpleDateFormat("hh:mm:ss");  
String str = sdf.format(new Date());
```

## Exploring java.util.regex package

- The java.util.regex package supports regular expression processing.
- Regular expression is a string of characters that describes a character sequence. The abbreviation for regular expression is regex.
- A regular expression is generally used for validation, searching or manipulating text.
- There are two classes that support regular expression processing and they are used together:-
  - 1) class Pattern
  - 2) class Matcher
- Pattern class is used to build a regular expression (regex).
- Matcher class is used to match the pattern (regex) built by the Pattern class.
- Any language that supports regular expression provides regex engine. Java language provides the Matcher class to invoke the regex engine and do the matching operations.



### ***Regular Expressions:-***

- A regular expression contains normal characters, meta characters, quantifiers or character classes (set of characters).
- Meta characters, quantifiers or character classes (set of characters) are used for building regular expressions to match dynamic values.

- 1) **Normal character:-** A normal character matches exactly as it is.

```
class Demo {  
    public static void main(String[] args) {  
        // Will match only Ravi  
        Pattern p = Pattern.compile("Ravi");  
        Matcher m = p.matcher("Ravi");  
        boolean b = m.matches();  
        System.out.println(b);  
    }  
}
```

- 2) **Meta characters:-** are symbols used to build regular expression to match dynamic values.

| Metacharacter | Description                                                                                              |
|---------------|----------------------------------------------------------------------------------------------------------|
| .             | Matches any character other than new line (A to Z a to z 0 to 9 and special symbols like @ & \$ , % etc) |
| \d            | Matches any numeric digit (0 to 9)                                                                       |
| \w            | Matches any alpha numeric character and underscore ( _ A to Z a to z 0 to 9 )                            |
| \s            | Matches whitespace character. (In java whitespace means space, newline, tab & carriage return)           |
| \D            | Matches any non digit. (Negation of \d )                                                                 |
| \W            | Negation of \w                                                                                           |
| \S            | Negation of \s. (Any non white space character)                                                          |

- 3) **Quantifiers:-** are used to specify the quantity or number of occurrences. A quantifier is also known as occurrence indicator.
- 4) **Character class:-** A character class is a set of characters. It is created with the help of [ ] brackets. A character class is created to match any sequence that contains one or more characters.

| Quantifier  | Description                                                                           |
|-------------|---------------------------------------------------------------------------------------|
| ?           | Preceding sub pattern must appear zero or one time.                                   |
| *           | Preceding sub pattern must appear zero or more times                                  |
| +           | Preceding sub pattern must appear one or more times.                                  |
| {n}         | Preceding sub pattern must appear 'n' number of times.                                |
| { min,max } | Preceding sub pattern must appear at least 'min' times and not more than 'max' times. |

### ***Different brackets used in a regex***

| Parentheses | Description                              |
|-------------|------------------------------------------|
| ( )         | Used for grouping sub patterns           |
| { }         | Used in quantifier for specifying range. |
| [ ]         | Used for specifying character set        |

**Q** What is the difference between matches() and find() present in Matcher class ?

**A** The matches() method matches an exact sequence but the find() method matches a sub sequence.

**Q** Why do you use regular expression?

**A** A regular expression is generally used for validation, searching or manipulating text.

***class Class and reflection in java***

- class Class encapsulates the details of a class which has got loaded into the JVM.
- Reflection is a process of getting information about a class and its members during runtime.
- Reflection is supported by class Class present in java.lang package and other classes like Field, Method, Constructor etc. present in java.lang.reflect package
- Reflection helps in getting the following runtime information about a class like:-
  - 1) Class name
  - 2) Fields
  - 3) Methods
  - 4) Constructors
  - 5) Package name
  - 6) Super classes or implemented interfaces
  - 7) Annotations
- Objects of class Class are created implicitly when classes are loaded into JVM.
- The number of class Class objects depends on the number of classes loaded into the JVM.

**Following are the three ways to get an instance of class Class.**

- 1) **getClass()** method of Object class.

```
class Test {  
    }  
class ReflectionDemo {  
    public static void main(String[] args) {  
        Test t = new Test();  
        Class c = t.getClass();  
    }  
}
```

- 2) **forName()** method of Class class.

```
class ReflectionDemo {  
    public static void main(String[] args) {  
        // use fully qualified class name.  
        Class c = Class.forName("com.cluster.Test");  
    }  
}
```

- 3) the .class syntax

```
class ReflectionDemo {  
    public static void main(String[] args) {  
        Class c = Test.class;  
    }  
}
```

**Q** What is reflection?

**A** It is a process of getting information about of a class and its members during runtime. It is supported by class `Class` and other classes like `Field`, `Method`, `Constructor` etc. present in `java.lang.reflect` package

**Q** What are the different ways a class can be loaded into JVM?

**A** A class can be loaded into JVM either implicitly or explicitly.

1. Implicitly:-

- a) When you use `new` keyword to create an object.
- b) When you access a static member.
- c) When you use `java` command.

2. Explicitly:-

- a) by using `forName()` method of class `Class`.

**Q** How can a class be explicitly loaded into JVM ?

**A** You can load any class explicitly into the JVM by using `forName()` method of class `Class`.



***interface Cloneable and cloning***

- 1) Cloning is a process of creating a duplicate copy of an object.
  - 2) Only classes that implement Cloneable interface can be cloned.
  - 3) If you try to clone a class that does not implement the Cloneable interface, CloneNotSupportedException is thrown.
  - 4) When a clone object is getting created the constructor is not called for that object.
  - 5) Some of the built in classes which implement Cloneable interface are Date, Calendar, all collection classes like LinkedList, ArrayList, TreeSet etc.
- Following are the rules to do cloning.
    - 1) To clone object of any class, that class must implement Cloneable marker interface otherwise CloneNotSupportedException will be thrown.
    - 2) You have to use clone() method which is present in class Object.
  - There are two types of cloning.
    - 1) **Shallow Cloning:-** We have to do shallow cloning when object to be cloned is having primitive variables as instance variables. By default clone() method in class Object is doing shallow cloning.
    - 2) **Deep Cloning:-** We have to do deep cloning when object to be cloned is having reference variables as instance variables.

***interface Cloneable:-***

- It is a marker interface and does not contain any members.
- It should be implemented for classes where cloning is required.

***Marker interface or Tag interface:-***

- Marker interface is used to provide indication to the JVM so that it can add special behavior to the class implementing it at runtime.
- A marker interface does not have any members (variable or method declarations).
- After introduction of Annotation in Java 5 version, Annotation is better choice than marker interface.
- Some of the examples for marker interface are
  - 1) Cloneable
  - 2) Serializable
  - 3) Remote
  - 4) RandomAccess

***interface Serializable and serialization***

- Serialization is a process of converting state of an object to a byte stream.
- Deserialization is a process of reconstructing a new object (restoring the bytes into values) from the byte stream. Deserialization is the reverse process of serialization.
- Deserialization will always create deep copies of the object.
- Only classes that implement java.io.Serializable interface can be serialized.
- If you try to serialize a class that does not implement the Serializable interface, then java.io.NotSerializableException is thrown.
- Some of the built in classes which implement Serializable interface are String class, all wrapper classes, all collection classes like LinkedList, ArrayList, TreeSet etc.
- If you do not want to serialize any data in the object then declare the field or instance variable as "transient". The keyword "transient" is used to skip serialization of an instance variable. The transient variable will be restored as null (for object references) or default values (for primitives).
- We can't serialize static variables because static variables belong to class and not to object. Serialization is a process of storing state of an object.
- When a new object is getting created by deserialization, the constructor is not called for that object.
- When a super class is implementing Serializable then both super class objects & also subclass objects are eligible for serialization. But the reverse is not true i.e. when a subclass is implementing Serializable interface then only the subclass objects are eligible for serialization and not the super class objects.
- When a super class is not implementing Serializable but its subclass is implementing it, then during the deserialization of subclass object the constructor of sub class will not be executed but the constructor of super class will be executed.

***interface Serializable:-***

- It is a marker interface and does not contain any members.
- It should be implemented for classes where serialization is required.

**Points to remember during Serialization**

- 1) The class has to implement Serializable. (Serializable is a marker interface).
- 2) If you do not want to save state of any instance variable then specify it as transient.
- 3) Static variables cannot be serialized.
- 4) If the class implementing Serializable interface, is having user-defined data as instance variable, then that user-defined data should also implement Serializable.

**Serialization is done for 3 reasons.**

- 1) Persistence (i.e. for saving the state of object to a file in the form of bytes)
- 2) Communication (i.e for sending the object via network from one JVM to another JVM)
- 3) Copying (i.e for copying and storing the values into the memory so that we can create duplicate objects whenever needed)

**1) Persistence:-**

- ☞ Serialization helps in saving the state of object to a file.
- ☞ At a later time, you may read the file and restore the object by using the process of deserialization.

```
class A implements Serializable {
    int x;
    int y;
    int z;

    A(int x, int y, int z) {
        System.out.println("Inside parameterized constructor");
        this.x = x;
        this.y = y;
        this.z = z;
    }
}

public class SerializeDemo {
    public static void main(String[] args) {
        try {
            A a = new A(10,20,30);
            // doing serialization
            FileOutputStream fos = new FileOutputStream("hello.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(a); // will serialize the object and write to
                               // file
            oos.close();

            // doing deserialization
            FileInputStream fis = new FileInputStream("hello.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            A a1 = (A) ois.readObject(); // will read the file and deserialize
   // the object
            ois.close();
            System.out.println("Value of object a is: " + a.x + "\t" + a.y +
                               "\t" + a.z);
            System.out.println("Value of object a1 is: " + a1.x + "\t" + a1.y +
                               "\t" + a1.z);
        }
        catch (Exception e) {
            System.out.println("Exception caught " + e);
        }
    }
}
```

**2) Communication:-** (i.e for sending the object via network from one JVM to another JVM)

- ☞ RMI (Remote Method Invocation) allows a java object on one machine to invoke a method of a Java object on a different machine. The RMI provides remote communication between two different applications.

- ☞ In the case of RMI or in distributed computing you may get a chance to pass data from one machine to another machine i.e from one JVM to another JVM. This data can be primitive data or user defined data (object).
- ☞ When you are passing the primitive data from one machine to another machine then call by value mechanism will be used i.e. data will be sent as it is via network.
- ☞ When you are passing the object, then call by reference has to be used i.e. address has to be copied from one JVM to another JVM but there is no use of passing the address. So you need a different mechanism to pass that object from one JVM to another JVM. For this we can use serialization and deserialization mechanism.
- ☞ Serialization is needed to implement Remote Method Invocation (RMI). RMI allows a java object on one machine to invoke a method of a Java object on a different machine. An object may be supplied as an argument to that remote method. The sending machine serializes the objects and transmits it. The receiving machine deserializes it.
- ☞ Serialization is the mechanism used by RMI to pass objects between JVMs, either as arguments in a method invocation from a client to a server or as return values from server to client.

**3) Copying:-** (i.e for copying and storing the values into the memory so that we can create duplicate objects whenever needed).

- ☞ It is used for creating duplicate objects.

### ***interface Externalizable & externalization***

- Externalization is a kind of serialization process where complete serialization process will be under developer's control.
- In serialization JVM will completely take control of serializing the object with its own logic and will try to serialize all the instance variables present in the object.
- When you are externalizing an object, developer has to serialize all the values explicitly. He can also decide which instance variable value he wants to serialize or not.
- For this process there is one interface called java.io.Externalizable

```
interface Externalizable extends Serializable {  
    void readExternal(ObjectInput in)  
    void writeExternal(ObjectOutput out)  
}
```

- interface Externalizable is not a marker interface because it is having two methods and marker interfaces do not have any members.
- The writeExternal() method is used for serializing the required instance variables. Also during serialization you can apply any encryption logic, compression logic etc.
- The readExternal() method is used for deserializing the values. This method can also be used for writing the decryption logic, decompression logic etc, which gets applied during deserialization.
- Externalization needs a default public constructor.
- Externalization is really helpful when you want to serialize transient variables.

**Q** Explain how externalization works ?

**A** JVM first checks for the Externalizable interface and if object supports Externalizable interface, then serializes the object using writeExternal() method.

When an Externalizable object is reconstructed, an instance is created first using the public no argument constructor, then the readExternal() method is called.

```
class Test {  
    int p;  
    int q;  
  
    Test(int i, int j) {  
        p = i;  
        q = j;  
    }  
}
```

```
class A implements Externalizable {  
    transient int x ;  
    transient int y ;  
    transient Test t ;  
  
    public A() {  
        System.out.println("Inside A constructor");  
    }  
  
    public A(int x,int y, Test t) {  
        this.x = x;  
        this.y = y;  
        this.t = t;  
    }  
  
    public void writeExternal(ObjectOutput out) throws IOException {  
        System.out.println("Using writeExternal() method during  
                                serialization");  
  
        out.writeInt(x);  
        out.writeInt(y);  
        out.writeInt(t.p);  
        out.writeInt(t.q);  
    }  
  
    public void readExternal(ObjectInput in) throws IOException {  
        System.out.println("Using readExternal()method during  
                                deserialization");  
  
        // values should be read in the same order as written in writeExternal()  
  method  
  
        x = in.readInt();  
        y = in.readInt();  
        int m = in.readInt();  
        int n = in.readInt();  
        t = new Test(m,n);  
    }  
}
```

```
public class ExternalizedDemo {
    public static void main(String[] args) {
        try {
            Test t = new Test(5,5);
            A a = new A(10, 20, t);
            FileOutputStream fos = new FileOutputStream("hello.ser");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(a);
            oos.close();

            FileInputStream fis = new FileInputStream("hello.ser");
            ObjectInputStream ois = new ObjectInputStream(fis);
            System.out.println("Creating deserialized object");
            A a1 = (A) ois.readObject();
            ois.close();
            System.out.println("Value of object a is: " + a.x + "\t" + a.y + "\t" +
                               a.t.p + "\t" + a.t.q + "\t" + a);
            System.out.println("Value of object a1 is: " + a1.x + "\t" + a1.y + "\t" +
                               a1.t.p + "\t" + a1.t.q + "\t" + a1);
        } catch (Exception e) {
            System.out.println("Exception caught " + e);
        }
    }
}
```

### ***Difference between Serialization and Externalization***

| Serialization                                                                   | Externalization                                                                                                                                             |
|---------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1) JVM takes control of complete process of serializing the data                | 1) Developer has complete control over serializing the data.                                                                                                |
| 2) The class has to implement interface Serializable. It is a marker interface. | 2) The class has to implement interface Externalizable. It is not a marker interface and you have to override two methods readExternal() & writeExternal(). |

## **J2SE 5 (java 5.0 or Tiger)**

### ***Java 5 new features:-***

- 1) static import
- 2) Generics
- 3) Autoboxing
- 4) Enhanced for loop
- 5) varargs
- 6) Enumerations
- 7) Annotations
- 8) new API addition (new classes have been added like StringBuilder, Formatter, Scanner, PriorityQueue etc)

### ***1. static import:-***

- Static import feature provided in Java 5 allows you to import static members of a class or interface (static variables and static methods).
- The advantage of static import is you can access the static members of a class or interface without qualifying them with class name.

**Q** What is the difference between normal import and static import?

- A** Using normal import we can import only classes in a package but static import will import all static members of a class(i.e. static variables and static methods).

### ***2. Generics:-***

- Generics provide type safety to collections.
- Earlier to java 5 the collection classes can work with any type of Object.
- After the introduction of generics in java 5 collection classes can now be used with type safety.
- Also earlier to java 5 all collection classes were non generic classes & from java 5 all collection classes were re-engineered to generic classes.

Problem without generics are: -

- When you are adding an element into a collection compiler ignores the types and allows different type of elements to be added into the collection.
- When you are getting an element out of a collection then many if conditional statements and many type casting is required. (When we take an element out of a collection we must do casting. Because of this inconvenience it is unsafe i.e. when a different type of an element is coming out from collection and we are casting to a different type we get a runtime exception called ClassCastException).
- Generic provides a way to communicate the type of the collection to the compiler so that it can be checked by the compiler.

### ***Advantage of using generics:-***

- Generic help you to restrict the type of element you add into collection during compilation time and thus provides type safe collections.
- No conditional checks, no casting required, when you get an element out of a collection.
- What type of element goes in when using generic, come out as element of same type.

### 3. Autoboxing:-

- Autoboxing is a process of converting primitives to objects and objects to primitives automatically.
- Earlier boxing & unboxing was done explicitly by the developer while adding and removing a primitive from a collection because all collection classes cannot store primitives and can store only objects.
- Autoboxing is especially useful when we put values in collection and retrieve values from a collection. We will not have to do the conversions or casting.

a) Boxing : converting primitives to objects.

```
Integer z = 10; // Boxing
```

b) Unboxing : converting objects to primitives.

```
int i = z; // Unboxing
```

### 4. Enhanced for loop:-

Syntax: - for(datatype variableName: Collection)

Eg: -

```
for(String x:ar1){  
    System.out.println(x);  
}
```

### 5. Varargs:-

- It helps to simplify method overloading.
- Varargs can be used for only one argument and that should be the last argument.
- Varargs can be applied to constructor also.

Syntax :- returntype methodname (datatype... var)

Eg: -

```
void m1(String... str){  
    for(String s:str){  
        System.out.println(s);  
    }  
}
```

### 6. Enumerations:-

- Enumeration is a new data type and was introduced in java 5.
- It is defined with a keyword called enum.
- An enum is used to define constants.
- An enum contains a list of constants.
- All the enum constants are implicitly declared as public static final.
- In Java an enum is similar to a class.



- An enum can have constructors, static and instance blocks, variables and methods but cannot have abstract methods.
- An enum should always have private constructor.
- You cannot create an object of enum because it has a private constructor.
- Each enum constant is an object of its own type.
- We cannot create an object of enum using "new" keyword but we can declare a reference variable of that enum type.
- All enums are subclasses of class java.lang.Enum class.
- enums cannot extend any class because it is already inheriting java.lang.Enum and java does not support multiple inheritance.
- An enum cannot be inherited.

### ***Difference between class and enum***

| class                                                       | enum                                                             |
|-------------------------------------------------------------|------------------------------------------------------------------|
| 1) It is a user defined data type                           | 1) It is also user defined data type.                            |
| 2) We can declare a reference variable of a class.          | 2) We can also declare a reference variable of enum.             |
| 3) A class can be instantiated.                             | 3) An enum cannot be instantiated.                               |
| 4) A class can have a constructor.                          | 4) An enum also can have a constructor.                          |
| 5) A class can have static variables, blocks and methods.   | 5) An enum also can have static variables, blocks and methods.   |
| 6) A class can have instance variables, blocks and methods. | 6) An enum also can have instance variables, blocks and methods. |
| 7) A class can extend another class.                        | 7) An enum cannot extend another enum.                           |
| 8) A class can be inherited.                                | 8) An enum cannot be inherited.                                  |

**Q** How do you compare whether two enums are equal or not?

**A** We can check equality of two enums by using equals() method or == (comparison) operator.

```
enum Box {
    b, b1, b2, b3, b4;
}

// can check for equality of enum using equals() method or ==
//                                     (comparison) operator

Box z = Box.b3;
if (z.equals(Box.b3))
{
    System.out.println("z contains box3");
}

if (z == Box.b3) {
    System.out.println("z contains box3");
}
```