# Java™

By Ashirbad Swain

**Index Pages**

7. **Interface**
    - Interface Declarations
    - Marker Interface
    - Adapter Classes
    - Interface vs Inheritance

8. **Garbage Collector**

9. **String Manipulations**
    - String
    - StringBuffer
    - StringBuilder
    - StringTokenizer

10. **Wrapper Class**
    - Datatypes vs Wrapper classes
    - All 8 wrapper classes explanations
    - Autoboxing vs Autounboxing
    - All possible conversions
    - toString (), parseXXX (), valueOf (), XXXValue ()

11. **java.io package**
    - Introduction
    - Character Oriented Streams
    - Byte Oriented Stream
    - Writing and reading operations on file
    - Normal streams vs Buffered streams
    - File class
    - Serialization
    - Deserialization

12. **Exception Handling**
    - Types of Exceptions
    - Exception vs Error
    - Try-catch blocks usage
    - Try with resources
    - Exception propagation
    - Finally block usage
    - Throws keyword usage
    - Exception handling vs method overriding
    - Throw keyword usages
    - Customization of Exception Handling
    - Different types of Exception and error

**13. Multithreading**

- Thread info
- Single Threaded model vs Multithreaded model
- Main Thread vs User Thread
- Creation of user defined Thread
- Life cycle stages of Thread
- Thread Naming
- Thread Priority
- Thread Synchronization
- Inter Thread Communication
- Hook Thread
- Daemon Thread
- Difference between wait (), notify (), notifyAll ()

**14. Nested Classes**

- Introduction
- Advantages of nested classes
- Nested classes vs inner classes
- Normal Inner classes
- Method Local inner classes
- Anonymous Inner classes
- Static nested classes

**15. Functional Interfaces & Lambda Expressions**

**16. Annotations**

- Introduction
- Advantages of annotations
- Different annotations working

**17. Enumeration**

- Introduction
- Advantages of Enumeration
- Enum vs enum
- Difference between enum vs class

**18. Arrays**

- Introduction
- Declarations of Arrays
- Arrays storing Object data & Primitive data

**19. Collection Framework**

- Introduction about arrays
- Advantages of collection over arrays
- Collection vs Collections
- Key Interfaces of Collections
- Characteristics of Collection framework classes
- Information about cursors
- Introduction about Map interface
- List interface implementation classes
- Set interface implementation classes
- Map interface implementation classes
- Comparable vs Comparator
- Sorting mechanisms of Collection Objects

**20. Generics**

- Type safety

**21. Networking**

- Introduction
- Socket and ServerSocket
- URL info
- Client – Server Programming

**22. AWT (Abstract Window Toolkit)**

- Introduction
- Frame class
- Different layouts
- Components of AWT (TextField, RadioButton, Checkbox…. etc)
- Event Handling or Event delegation Model
- Different types of Listeners

**23. Swings**

- Difference between Awt and swings
- Advantages of swings
- Different components of swings (TextField, RadioButton, Checkbox…etc)
- Event Handling in Swings

**24. Applet in Java**

25. **Internationalization (I18N)**
    - Design application to support dif country languages
    - Local class
    - ResourceBundle
    - Date in different formats
    - Info about properties files

26. **JVM Architecture**
    - What is JVM
    - Structure of the JVM
    - Components of JVM

27. **Interview Questions**

28. **Regular Expressions**

29. **Assertions**

## Java Introduction

Author – James Gosling

Vendor – Sun Micro System (which has since merged into Oracle Corporation)

Project Name – Green Project

Type – open source & free software

Initial Name – OAK language

Present Name – Java

Extensions - .java & .class & .jar

Initial version – Jdk 1.0 (java development kit)

Present version – java 13 2019

Operating System – Multi Operating System

Implementation Language – C, Cpp...

Symbol – coffee cup with saucer

Objective – To develop web applications

SUN – Stanford Universally Network

Slogan/Motto – WORA (Write once run anywhere)

## Importance of Core Java

According to the SUN 3 billion devices run on the java language only.

- Java is used to develop Desktop Applications such as MediaPlayer, Antivirus etc.
- Java is used to develop Web Applications such as irctc.co.in etc.
- Java is used to develop Enterprise Applications such as Banking Applications.
- Java is used to develop Mobile Applications.
- Java is used to develop Embedded System.
- Java is used to develop Smart Cards.
- Java is used to develop Robotics.
- Java is used to develop Games.....etc.

## Technologies depends on Core Java

The following are depending on java language,

- Advanced Java
- Hadoop
- TIBCO
- Selenium
- Android
- Cloud Computing
- Salesforce
- ADF
- SAP
- Spring
- Webservices
- Hibernate
- Structs. Etc

## Parts of Java

As per the sun micro system standard the java language is divided into three parts,

1. J2SE/JSE (Java 2 Standard Edition)
2. J2EE/JEE (Java 2 Enterprise Edition)
3. J2ME/JME (Java to Micro Edition)

## Keywords of Java (50)

- Data types (8) – byte, short, int, long, float, double, char, boolean
- Method level (2) – void, run
- Flow control (10) – if, else, switch, case, default, break, for, while, do, continue
- Object level (4) – new, this, super, instanceof
- Source file (6) – class, extends, interface, implements, package, import
- Exception handling (5) – try, catch, finally, throw, throws
- 1.5 version (2) – enum, assert
- Unused (2) – goto, const
- Modifiers (11) – public, private, protected, abstract, final, static, strictfp, native, transient, volatile, synchronized
- Predefined constants (3) – true, false, null

Difference between C & CPP & Java

| C – Language | Cpp – Language | Java – Language |
|---|---|---|
| #include<stdio.h><br>Void main ()<br>{<br>Printf("ashirbad");<br>} | #include<iostream.h><br>Void main ()<br>{<br>Cout<<" ashirbad";<br>} | Import java.lang.System;<br>Import java.lang.String;<br>Class test<br>{<br>Public static void main (String args [])<br><br>{System.out.println("ashirbad");}<br>} |
| Author – Dennis Ritchie | Author – Bjarne Stroustrup | Author – James Gosling |
| Implementation language – COBOL, FORTRAN, BCPL… | C, ada, ALGOL68… | C, CPP, ObjectiveC… |
| In c – language the predefined support is available in the form of header files.<br>Ex – stdio.h, conio.h | In cpp language the predefined is maintained in the form of header files.<br><br>Ex – iostream.h | In java predefined support is available in the form of packages.<br><br>Ex – java.lang, java.io, java.awt |
| The header files contain predefined functions.<br><br>Ex – printf, scanf… | The header files contain predefined functions.<br><br>Ex – cout, cin… | The packages contain predefined classes and interfaces and these class & interface contains predefined methods.<br>Ex – String, System… |
| In the above example we are using printf predefined function that is present in stdio.h header file hence must include that header file by using #include statement.<br>Ex - #include<stdio.h> | In the above example we are using cout predefined function that is present in iostream header file hence must include that header file by using #include statement.<br>Ex - #include<iostream.h> | In the above example we are using two classes (String, System) these classes are present in java.lang package must import it by using import keyword.<br>Ex –<br>import java.lang.*; - for all cases<br>import java.lang.String;<br>import java.lang.System; |
| In c language program execution starts from main method called by operating system. | In cpp language execution starts from main method called by operating system. | In java execution starts from main called by JVM. |
| To print data use printf () | To print data use cout () | To print data use System.out.println() |

| Version Name | Code Name | Release Date |
|---|---|---|
| JDK 1.0 | Oak | January 1996 |
| JDK 1.1 | (none) | February 1997 |
| J2SE 1.2 | Playground | December 1998 |
| J2SE 1.3 | Kestrel | May 2000 |
| J2SE 1.4 | Merlin | February 2002 |
| J2SE 5.0 | Tiger | September 2004 |
| JDK 6 | Mustang | December 2006 |
| JDK 7 | Dolphin | July 2011 |
| JDK 8 | | March 2014 |
| JDK 9 | | September, 21st 2017 |
| JDK 10 | | March, 20th 2018 |
| JDK 11 | | September, 25th 2018 |
| JDK 12 | | March, 19th 2019 |
| JDK 13 | | September, 10th 2019 |

## What is Java?

- ✓ Java is a high-level, robust, object-oriented and secure programming language.
- ✓ Application of Java:
  - Mainly used for Application programming
  - Desktop application such as acrobat reader, media player, antivirus, etc.
  - Web application such as irctc.co.in etc.
  - Enterprise application such as banking application
  - Mobile
  - Embedded system
  - Smart card
  - Robotics
  - Games
- ✓ Java is developed by James Gosling.

## Features of Java

- ✓ The primary objective of the Java programming language was to make it simple, portable and secure programming language.
- ✓ The most important features are,

### Simple

- ✓ Java is easy to learn, and its syntax is simple, clean and easy to understand.
- ✓ Java syntax is based on C++.
- ✓ Java has removed many complicated and rarely used features, for example, explicit pointer, operator overloading, etc.
- ✓ There is no need to remove unreferenced objects because there is an Automatic Garbage Collector in Java.

### Object-oriented

- ✓ Java is an object-oriented programming language because everything in java is an object.
- ✓ Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behaviour.
- ✓ Basic concepts of OOPs are,
  - Object
  - Class
  - Inheritance
  - Polymorphism
  - Abstraction
  - Encapsulation

### Platform Independent

- ✓ The meaning of platform independent is that, the java source code can run on all operating systems.
- ✓ But the main point is that the JVM is platform dependent because JVM depends on the operating system. So, if you're running MAC OS you will have a different JVM than if you are running Windows or some other operating system.
- ✓ This fact can be verified by trying to download the JVM for your particular machine – when trying to download it, you will be given a list of JVM's corresponding to different operating systems, and you will pick whichever JVM is targeted for the operating system that you are running.

- ✓ So, we can conclude that JVM is platform dependent and it is the reason why Java is able to become "Platform Independent".

Secured

- ✓ Java is secured because,
  - No explicit pointer
  - Java program runs inside a virtual machine sandbox
  - Classloader: classloader in java is a part of the Java Runtime Environment (JRE) which is used to load Java classes into the java virtual machine dynamically.
  - Bytecode Verifier: it checks the code fragments for illegal code that can violate access right to objects.
  - Security Manager: it determines what resources a class can access such as reading and writing to the local disk.

Robust

- ✓ Robust means strong. Java is robust because,
  - It uses strong memory management.
  - There is a lack of pointers that avoids security problems.
  - There is automatic garbage collector in java which runs on the Java virtual machine.
  - There are exception handling and the type checking mechanism in Java.

Architecture – Neutral

- ✓ Java tech applications compiled in one Architecture/hardware (RAM, hard disk) and that compiled program runs on any architecture (hardware) is called architectural neutral.

Portable

- ✓ Java is portable because it facilitates you to carry the Java bytecode to any platform. It doesn't require any implementations.

High-Performance

- ✓ Java is faster than other traditional interpreted programming languages because Java bytecode is "close" to native code.
- ✓ It is still a little bit slower than a compiled language because java is an interpreted language.

Distributed

- ✓ Java is distributed because it facilitates users to create distributed application in Java. (Distributed applications are applications or software that runs on multiple computers within a network at the same time and can be stored on server or with cloud computing.)

Multi-threaded

- ✓ A thread is like a separate program, executing concurrently.
- ✓ We can write Java programs that deal with many tasks at once by defining multiple threads.
- ✓ The main advantages of multi-threading are that it doesn't occupy memory for each thread. It shares a common memory area.
- ✓ Threads are important for multi-media, web application etc.

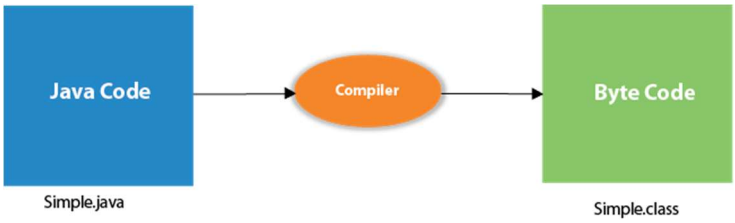
Dynamic

- ✓ Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand.

## How Java program works?

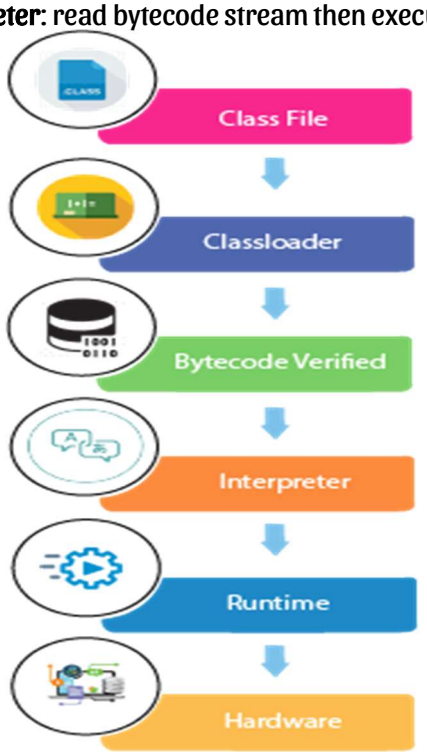
### What happens at compile time?

- ✓ At compile time, Java file is compiled by Java Compiler and convert the java source code into the byte code. And then the code is executed by interpreter to produce output.




### What happens at runtime?

- ✓ At runtime, the following steps are performed,
- ✓ **Classloader**: is the subsystem of JVM that is used to load class files.
- ✓ **Bytecode verifier**: checks the code fragments for illegal code that can violate access right to objects.
- ✓ **Interpreter**: read bytecode stream then execute the instructions.




### Can you save a java source file by other name than the class name?

- ✓ Yes, if the class is not public.

### Can you have multiple classes in a java source file?

- ✓ Yes.

## Types of Java Applications

1. Standalone Applications
   - ✓ It is known as window-based applications or desktop applications.
   - ✓ This type of applications must install in every machine-like media player, antivirus etc.
   - ✓ By using AWT & Swings we are developing these types of applications.
   - ✓ This type of application does not required client – server architecture.

2. Web Applications
   - ✓ The applications which are executed as server side those applications are called web applications like Gmail, Facebook, yahoo etc.
   - ✓ All applications present in internet those are called web – applications.
   - ✓ The web applications required client – server architecture.
     - **Client** – who sends the request
     - **Server** – it contains application & it process the app & it will generate response.
     - **Database** – used to store the data.
   - ✓ To develop web applications we are using servlets, structs, spring etc.

3. Enterprise applications
   - ✓ It is a business application 7 most of the people use the term it I big business application.
   - ✓ Enterprise applications are used to satisfy the needs of an organization rather than individual users. Such organizations are business, schools, government etc.
   - ✓ An application designed for corporate use is called enterprise application.
   - ✓ An application is distributed in nature such as banking applications.
   - ✓ All J2EE and EJB is used to create enterprise applications.

4. Mobile applications
   - ✓ The applications which are design for mobile platform are called mobile applications.
   - ✓ We are developing mobile applications by using android, IOS, J2ME etc.
   - ✓ There are 3 types of mobile applications,
     - **Web application** – Gmail, online shopping, oracle etc
     - **Native** (run on device without internet or browser) – phone call, calculator, alarm, games – these are installed from application store & to run these apps internet not required.
     - **Hybrid** (required internet data to launch) – WhatsApp, Facebook, LinkedIn etc. – these are installed from app store but to run this application internet is required.

5. Distributed applications
   - Software that executes on two or more computers in a network. In a client – server environment.
   - Application logic is divided into components according to function.
   - Ex: aircraft control system, industrial control system, network applications etc.

## Types of Software

The set of instructions that makes the computer system do something,

1. Application Software
   - The program that allows the user to perform particular task.
   - Ex: business apps, entertainment apps, MS – office etc.

2. System Software
   - The program that allows the hardware to run properly.
   - Ex: operating system, device drivers etc.

## Java Coding Conventions

### Classes: -

- Class name start with upper case letter and every inner word starts with upper case letter.
- This convention is also known as **camel case** convention.
- The class name should be nouns.

  Ex: - **S**tring **S**tring**B**uffer I**n**put**S**tream**R**eader .....etc

### Interfaces: -

- Interface name starts with upper case and every inner word starts with upper case letter.
- This convention is also known as **camel case** convention.
- The class name should be nouns.

  Ex: **S**erializable Cloneable
  **R**andom**A**ccess

### Methods: -

- Method name starts with lower case letter and every inner word starts with upper case letter.
- This convention is also known as mixed case convention
- Method name should be verbs.

  Ex: - post () char**A**t () to**U**pper**C**ase () compare**T**o**I**gnore**C**ase ()

### Variables: -

- Variable name starts with lower case letter and every inner word starts with upper case letter.
- This convention is also known as mixed case convention.

  Ex: - out in page**C**ontext

### Package: -

- Package name is always must written in lower case letters.

  Ex: - java.lang java.util java.io etc

### Constants: -

- While declaring constants all the words are uppercase letters.

  Ex: - MAX_PRIORITY MIN_PRIORITY NORM_PRIORITY

Note: The coding standards are mandatory for predefined library & optional for user defined library but as a java developer it is recommended to follow the coding standards for user defined also.

Note:

- ✓ Java contains 14 predefined packages but the default package in java is java.lang.
- ✓ The class contains main method is called Main class and java allows to declare multiple main class in a source file.
- ✓ The source file allows to declare only one public class, if you are declaring more than one public class compiler generate error messages.
- ✓ Some example compiled & executed but it is not recommended because the class starting with lower case letters.

## Separators in Java

| *Symbol* | *name* | *usage* |
|---|---|---|
| *()* | *parentheses* | *used to contains list of parameters & contains expression.* |
| *{}* | *braces* | *block of code for class, method, constructors & local scopes.* |
| *[]* | *brackets* | *used for array declaration.* |
| *;* | *semicolon* | *terminates statements.* |
| *,* | *comma* | *separate the variables declaration & chain statements in for.* |
| *.* | *period* | *used to separate package names from sub packages. And also used for separate a variable, method from a reference type.* |

## Print () vs println ()

- **Print ()**: - used to print the statement in console and the control is present in the same line.

  Ex: - System.out.print ("Cerner Healthcare");
  System.out.print ("core java");
  Output: - Cerner HealthcareCorejava

- **Println ()**: - used to print the statement in the console but the control is there in next line.

  Ex: - System.out.println("Cerner Healthcare");
  System.out.println ("core java");
  Output: - Cerner Healthcare
  Core java

## Java tokens

- ✓ Smallest individual part of a java program is called Token.
- ✓ It is possible to provide any number of spaces in between two tokens.

```
Class test
{
        public static void main (String args [])
        {
                Int a = 10;
                System.out.println ("java tokens");
        }
}
// Tokens are: - class, test, {, ", [.... etc
```

## Escape Sequence

A character preceded by a backslash (\) is an escape sequence and has special meaning to the compiler. The following table shows the Java escape sequences,

**Escape Sequences**

| Escape Sequence | Description |
|---|---|
| \t | Insert a tab in the text at this point. |
| \b | Insert a backspace in the text at this point. |
| \n | Insert a newline in the text at this point. |
| \r | Insert a carriage return in the text at this point. |
| \f | Insert a formfeed in the text at this point. |
| \' | Insert a single quote character in the text at this point. |
| \" | Insert a double quote character in the text at this point. |
| \\ | Insert a backslash character in the text at this point. |

## Java Identifiers

Every name in java is called identifier such as,

- Class – name
- Method – name
- Variable – name

### Rules to declare identifier,

- An identifier contains group of Uppercase & lower-case characters, numbers, underscore & dollar sign characters but not start with number.
- Java identifiers are case sensitive of course java is case sensitive programming language. The below three declarations are different & valid,

```
Class Test
{
        int NUMBER = 10;
        int number = 10;
        int Number = 10;
}
```

- The identifier should not duplicated & below example is invalid because it contains duplicate variable name,

```
class Test
{
        int a = 10;
        int a = 20;
}
```

- In the java applications it is possible to declare all the predefined class names & interfaces names as an identifier but it is not recommended to use.
- It is not possible to use keywords as an identifier.

```
Class Test
{
        int if = 10;
        int try = 20;
}
```

- There is no length limit for identifiers but is never recommended to take lengthy names because it reduces readability of the code.

## Java Variables

- ✓ A variable is a container which holds the value while the java program is executed.
- ✓ Variable is reserved area allocated in memory.

- ✓ Types of variable:
  - Local variable:
    - A variable declared inside the body of the method is called a local variable. You can use that variable within that method but not by other.
    - Local variable cannot be defined with "static" keyword.
  - Instance variable:
    - A variable declared inside the class but outside the body of the method, is called instance variable.
    - Instance variable doesn't get any memory at compile time. It gets memory at runtime when an object or instance is created.
    - It is not declared as static.
  - Static variable:
    - A variable which is declared as static is called static variable.
    - It cannot be local. You can create a single copy of static variable and share among all the instances of the class.

## Java Data Types

- ✓ Data types specify the different sizes and values that can be stored in the variable.
- ✓ Data types are used to represent type of the variable & expressions.
- ✓ Specifies the range value of the variable.

There are two types of data type,

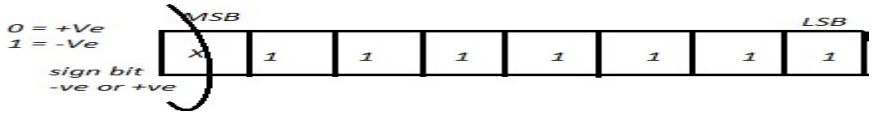
Primitive datatype

| *Data Type* | *size (in bytes)* | *Range* | *default values* |
|---|---|---|---|
| *byte* | *1* | *-128 to 127* | *0* |
| *short* | *2* | *-32768 to 32767* | *0* |
| *int* | *4* | *-2147483648 to 2147483647* | *0* |
| *long* | *8* – 9,223,372,0 | 36,854,775,808 to 9 ,223,372,036,854,775,807 | *0* |
| *float* | *4* | *-3.4e38 to 3.4e* | *0.0* |
| *double* | *8* | *-1.7e308 to 1.7e308* | *0.0* |
| *char* | *2* | *0 to 6553* | *single space* |
| *Boolean* | *no-size* | *no-range* | *false* |

***Byte :-***

*Size : 1-byte*
*MAX_VALUE : 127*
*MIN_VALUE : -128*
*Range : -128 to 127*

Formula : $-2^n$ to $2^{n-1}$ $-2^8$ to $2^{8-1}$




✓ Non – primitive datatype

- It includes classes, interfaces and arrays.

Note:

- To represent numeric values (10, 20, 30…etc) use **byte, short, int, long.**
- To represent decimal values (floating point values 10.5, 30.6…etc) use float, double.
- To represent character use char and take the character within single quotes.
- To represent true, false use Boolean.
- Except Boolean and char remaining all data types consider as signed data types because we can represent both +ve & -ve values.

Float vs double

- Float will give 5 to 6 decimal places of accuracy but double gives 14 to 15 places of accuracy.
- Float will fallow single precision but double will fallow double precision.

```
Syntax:
Data-type name-of-variable = value/literal;

Ex: - int a = 10;

int --- Data type
a ----- variable name
= ----- assignment
10 ---- constant value
; ----- statement terminator
```

- String is not a data type & it is a class present in java.lang package to represent group of characters or character array enclosed with in double quotes.

## Keywords in Java

- Java keywords are also known as Reserved words. Keywords are particular words which acts as a key to a code.
- Java keywords are,
- abstract: Java abstract keyword is used to declare abstract class. Abstract class can provide the implementation of interface. It can have abstract and non-abstract methods.
- boolean: Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
- break: Java break keyword is used to break loop or switch statement. It breaks the current flow of the program at specified condition. When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes next statement following the loop.

```
package OopsConcept;

public class BreakStatement {

	public static void main(String[] args)
	{
		//using for loop
		for(int i = 1; i <= 10; i++)
		{
			if(i == 5)
			{
				//breaking the loop
				break;
			}
			System.out.println(i);
		}
	}

}

1 2 3 4
```

- byte: Java byte keyword is used to declare a variable that can hold an 8-bit data values.
- case: Java case keyword is used to with the switch statements to mark blocks of text.
- catch: Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
- char: Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
- class: Java class keyword is used to declare a class.

- continue: Java continue keyword is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. The continue statement is used in loop control structure when you need to jump to the next iteration immediately.

```
package OopsConcept;

public class ContinueStatement {

	public static void main(String[] args)
	{
		//for loop
		for(int i = 1; i <= 10; i++)
		{
			if(i == 5)
			{
				//using continue statement
				continue;// it will skip the rest statement
			}
			System.out.println(i);
		}
	}

}

// as you can see the output, 5 is not printed on the console. it is because the loop is continued when it reaches to 5
1 2 3 4 6 7 8 9 10
```

- default: Java default keyword is used to specify the default block of code in a switch statement.
- do: Java do keyword is used in control statement to declare a loop. It can iterate a part of the program several times.
- double: Java double keyword is used to declare a variable that can hold a 64-bit floating-point numbers.
- else: Java else keyword is used to indicate the alternative branches in an if statement.
- enum: Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
- extends: Java extends keyword is used to indicate that a class is derived from another class or interface.
- final: Java final keyword is used to indicate that a variable holds a constant value. It is applied with a variable. It is used to restrict the user.
- finally: Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether exception is handled or not.
- float: Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.

- for: Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some conditions become true. If the number of iterations is fixed, it is recommended to use for loop.
- if: Java if keyword tests the condition. It executes the if block if condition is true.
- implements: Java implements keyword is used to implement an interface.
- import: Java import keyword makes classes and interfaces available and accessible to the current source code.
- instanceof: Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
- int: Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
- interface: Java interface keyword is used to declare an interface. It can have only abstract methods.
- long: Java long keyword is used to declare a variable that can hold a 64-bit integer.
- native: Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
- new: Java new keyword is used to create new objects.
- null: Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
- package: Java package keyword is used to declare a Java package that includes the classes.
- private: Java private keyword is an access modifier. It is used to indicate that a method or variable may be accessed only in the class in which it is declared.
- protected: Java protected keyword is an access modifier. It can be accessible within package and outside the package but through inheritance only. It can't be applied on the class.
- public: Java public keyword is an access modifier. It is used to indicate that an item is accessible anywhere. It has the widest scope among all other modifiers.
- return: Java return keyword is used to return from a method when its execution is complete.
- short: Java short keyword is used to declare a variable that can hold a 16-bit integer.
- static: Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is used for memory management mainly.
- strictfp: Java strictfp is used to restrict the floating-point calculations to ensure portability.
- super: Java super keyword is a reference variable that is used to refer parent class object. It can be used to invoke immediate parent class method.
- switch: The Java switch keyword contains a switch statement that executes code based on test value. The switch statement tests the equality of a variable against multiple values.
- synchronized: Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
- this: Java this keyword can be used to refer the current object in a method or constructor.

- throw: The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exception. It is followed by an instance.
- throws: The Java throws keyword is used to declare an exception. Checked exception can be propagated with throws.
- transient: Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
- try: Java try keyword is used to start a block of code that will be tested for exceptions. The try block must be followed by either catch or finally block.
- void: Java void keyword is used to specify that a method does not have a return value.
- volatile: Java volatile keyword is used to indicate that a variable may change asynchronously.
- while: Java while keyword is used to start a while loop. This loop iterates a part of the program several times. If the number of iterations is not fixed, it is recommended to use while loop.

## Java Flow Control Statements

### Control statements

### if-else

- ✓ Java if statement tests the condition. It executes the if block if condition is true.
- ✓ Java if-else statement also check the condition. It executes the if block if condition is true otherwise else block is executed.

```
if(condition)
{
//code if condition is true
}
else
{
//code if condition is false
}
```

### Switch statement

- ✓ The Java switch statement executes one statement from multiple conditions. It is like if-else-ladder statement.
- ✓ Switch statement is used to declare multiple selections.

```
//Syntax
switch(expression)
{
case value1:
 //code to be executed;
 break;  //optional
case value2:
```

```
//code to be executed;
break;  //optional
.....

default:
code to be executed if all cases are not matched;
}
```

## Loops in Java

- ✓ When to use,
  - If the number of iterations is fixed, it is recommended to use for loop.
  - If the number of iterations is not fixed, it is recommended to use while loop.
  - If the number of iterations is not fixed and you must have to execute the loop at least once, it is recommended to use the do-while loop.

For Loop,

```
for (initialization; condition; incr/decr) {
//code to be executed
}
Example:
for (int i=1; i<=10; i++) {
System.out.println(i);
}
```

While Loop,

```
while(condition) {
//code to be executed
}
Example:
int i=1;
while(i<=10){
System.out.println(i);
i++;
}
```

Do – while Loop,

```
do {
//code to be executed
} while(condition);
Example:
int i=1;
do {
```

```
System.out.println(i);
i++;
} while(i<=10);
```

- ✓ In programming languages, loops are used to execute a set of instructions/functions repeatedly when some conditions become true.

Note:

- ✓ In Java semicolon is empty statement.
- ✓ Float, double and long is not allowed for a switch argument because these are having too message.
- ✓ Inside the switch the case labels must be unique; if we are declaring duplicate case labels the compiler will raise compilation error "**duplicate case label**".
- ✓ Inside the switch for the case labels & switch argument it is possible to provide expressions (10+10+20, 10*4, 10/2).
- ✓ Inside the switch, the case label must be constant values. If we are declaring variables as a case labels the compiler will show compilation error "constant expression required".
- ✓ It is possible to declare final variables as a case label. Because the variables are replaced with constants during compilation.
- ✓ Inside switch, both cases and default are optional.
- ✓ Break is used to stop the execution. And is possible to use the break statement only two areas.
  - Inside the switch statement
  - Inside the loops
- ✓ Continue is used to skip the current iteration and it is continuing the rest of the iterations normally.

## Java Variables

- ✓ Variables are used to store the constant values by using these values we are achieving project requirements.
- ✓ Variables are also known as **fields** of a class or **properties** of a class.
- ✓ All variables must have a type. You can use primitive types such as int, float, boolean etc. or array type or class type or enum type or interface type.
- ✓ Variables declaration is composed of three components in order,
  - Zero or more modifiers
  - The variable type
  - The variable name
- ✓ **Ex**: - public final int x = 100;

**public int a=10;**

***public*** ----> *modifier (specify permission)*

***int*** ----> *data type (represent type of the variable)*

***a*** ----> *variable name*

*10* ----> *constant value or literal;*

*;* ----> *statement terminator*

There are 3 types of variables in Java,

- Local variables
- Instance variables
- Static variables

## Local Variables

- The variables which are declare inside a **method** or **constructor** or **blocks** those are called local variables.

```
Class Test
{
        public static void main (String args [])//Execution starts from the
                main method
        {
                int a = 10; //local variables
                int b = 20;
                System.out.println (a);
                System.out.println (b);
        }
}
```

- It is possible to access local variables only inside the method or constructor or blocks only, it is not possible to access outside of method or constructor or blocks.

```
Void add ()
{
        int a = 10;//local variable
        System.out.println (a);//possible
}
Void mul ()
{
        System.out.println (a);//not possible
}
```

- For the local variables, memory allocated when method starts and memory released when method completed.
- The local variables are stored in stack memory.

## Instance variable (non – static variables)

- The variables which are declare inside a class but outside of methods those variables are called instance variables.
- The scope (permission) of instance variable is inside the class having global visibility.
- For the instance variables memory allocated during object creation & memory released when object is destroyed.
- Instance variables are stored in **heap** memory.

**Example**:

```
Class Test
{
        //Instance variables
        int a = 10;
        int b = 20;
        //Static method
        Public static void main (String args [])
        {
                //Static area
                Test t = new Test ();
                System.out.println (t.a);
                System.out.println (t.b);
                t.m1 ();
        }
        //Instance method
        Void m1 ()//user defined method must call by user inside main method
        {
                //Instance area
                System.out.println (a);
                System.out.println (b);
        }//main ends
};//class ends
```

## Static variables (class variables)

- The variables which are declare inside a class but outside of methods with static modifier those variables are called static variables.
- Scope of the static variables within the class global visibility.
- Static variables memory allocated during class file loading and memory released at class file unloading time.
- Static variables are stored in non-heap memory.
- Static variables can be access by using class name without object creation.

Example

```
Class Test
{
        //static variables
        static int a = 1000;
        static int b = 2000;
        public static void main (String args []) //static method
        {
                System.out.println (Test.a);
                System.out.println (Test.b);
                Test t = new Test ();
                t.m1 (); //instance method calling
        }
        //instance method
        void m1 ()
        {
                System.out.println (Test.a);
                System.out.println (Test.b);
        }
};
```

Static Variable calling

We are able to access the static members inside the static area in three ways.

- Direct accessing
- By using class name
- By using variables

In the above three approaches second approach is best approach.

```
Class Test
{
        static int a = 100; //static variables
        public static void main (String args [])
        {
                //1-way (directly possible)
                System.out.println (a);

                //2-way (by using class name)
                System.out.println (Test.a);

                //3-way (by using reference variable)
                Test t = new Test ();
                System.out.println (t.a);
        }
};
```

Note:

- ✓ When we create object inside method that object is destroyed when method is completed, if any other method required object then create the object inside that method.
- ✓ For the instance and static variables JVM will assign default values but for the local variables the JVM won't provide default values.
- ✓ In java before using local variables must initialize some values to the variables otherwise compiler will raise compilation error "variable a might not have been initialized".

| Characteristics | Local variable | Instance variable | Static variable |
|---|---|---|---|
| where declared | Inside method or constructor or block | Insid1e the class outside of methods | Inside the class outside the method with static modifier |
| usage | Within the method | Inside the class | Inside the class all |
| when memory allocated | When method starts | When object created | When .class file loading |
| when memory destroyed | When method ends | When object destroyed | When .class file unloading |
| Initial values | None, must initialize the value before first use | Default values are assigned by JVM | Default values are assigned by JVM |
| Relation with object | No way related to object | For every object one copy of instance variable created, it means memory | For all objects one copy is created.<br>It means single memory |
| accessing | Directly possible | By using object name | By using class name |
| memory | Stored in stack memory | Stored in heap memory | Method area |

## JAVA METHODS (behaviours)

- ➢ Inside the classes it is not possible to write the business logics directly, hence inside the class declare the method, inside that method writes the logics of the application.
- ➢ Methods are used to write the business logics of the project.
- ➢ In java, a method is like a function which is used to expose the behaviour of an object.
- ➢ A set of codes which perform a particular task is known as method.

### Method Syntax

```
Access-modifier return-type method-name (list of parameters) throws Exception
{
        //body
}
```

| | | |
|---|---|---|
| ***Modifiers-list*** | ------☒ | *represent access permissions* |
| ***Return-type*** | ------☒ | *functionality return value* |
| ***Method name*** | ------☒ | *functionality name* |
| ***Parameter-list*** | ------☒ | *input to functionality* |
| ***Throws Exception*** | ------☒ | *representing exception handling* |

Coding convention

- Method name starts wit lower case letter and every inner word starts with uppercase letter (mixed case).
- Ex: - post (), charAt (), toUpperCase (), compareToIgnoreCase () ...etc

Note

- Inside the class it is possible to declare any number of instance & static methods based on the developer requirement.
- It will improve the **reusability** of the code and we can **optimize** the code.
- Whether it is an instance method or static method, the methods are used to provide business logics of the project.
- There are 2 types of methods,
  - Instance method
  - Static method

Instance method

- For the instance members memory is allocated during object creation hence access the instance members by using object-name (reference variable).

```
void m1 () //instance method
{
        //body //instance area
}
```

- Method calling syntax

```
void m1 () {logics here} //instance method
Objectname.instancemethod (); // calling instance method
Test t = new Test ();
t.m1 ();
```

## Static method

- For the static member's memory allocated during **.class** file loading hence access the static members by using class name.

```
Static void m1 () //static method
{
        //body //static area
}
```

- Method calling syntax

```
Static void m2 () {logics here} //static method
Classname.staticMethod (); //call static method by using class name
Test.m2 ();
```

## Method Signature

- Method – name & parameters list is called method signature.
- Syntax: - **method-name (parameter-list)**
- Ex: - m1 (int a)
  m1 (int a, int b)

**Example**

```
class Test
{
        void m1 ()
        {
                System.out.println ("m1 instance method");
        }
        static void m2 ()
        {
                System.out.println ("m2 static method");
        }
        public static void main (String args [])
        {
                Test t = new Test ();
                //calling of instance method by using object-name
                t.m1 ();
                //calling of static method by using class-name
                Test.m2 ();
        }
}
```

Note:

- Inside the class it is not possible to declare two methods with same signature, if we are trying to declare two methods with same signature compiler will raise compilation error message **"m1 () is already defined in Test"** (java class not allowed duplicate methods).
- For java methods return type is mandatory otherwise the compilation will generate error message **"invalid method declaration; return type required"**.
- Declaring the class inside another class is called **inner classes**, java supports inner classes.
- Declaring the methods inside other methods is called inner methods, but java not supporting inner methods concept if we are trying to declare inner methods compiler will generate error message "**illegal start of expression**".
- One operator with more than one behaviour is called **operator overloading**. Java not supporting operator overloading concept but only one implicit overloaded operator in java is **+ operator**.
    - If two operands are integers then plus (+) perform **addition**.
    - If at least one operand is string then plus (+) perform **concatenation**.
- In java one method is calling another method by using method name. one java method is able to call more than one method, but once the method is completed the control return to the caller method.
- In java **this** keyword is instance variable hence it is not possible to use inside static area. If we are using **this** variable inside static context then compiler will generate error message **"non-static variable** this **cannot be referenced from a static context"**. In the static context it is not possible to use this & super keywords.
- **Stack mechanism**
    - In java, program execution starts from main method called by JVM & just before calling main method JVM will creates one empty stack memory for that application.
    - When JVM calls particular method then that method entry and local variables of that method stored in stack memory 7 when the method completed, that particular method entry and local variables are destroyed from stack memory 7 that memory becomes available to other methods.
    - The JVM will create stack memory just before calling main method & JVM will destroyed stack memory after completion of main method.
- When we call methods recursively then we will get **StackOverFlowError**.
- **Java.util. Scanner**
    - Scanner class present in **java. util** package and it is introduced in 1.5 versions & it is used to take dynamic input from the keyboard.
    - To communicate with system resources, use **system** class & to take input from keyboard use **in** variable (in=input).

**Scanner s = new Scanner (System.in); //Scanner object creation**

| | | |
|---|---|---|
| *to get int value* | ----> | *s.nextInt()* |
| *to get float value* | ----> | *s.nextFloat()* |
| *to get byte value* | ----> | *s.nextbyte()* |
| *to get String value* | ----> | *s.next()* |
| *to get single line* | ----> | *s.nextLine()* |
| *to close the input stream* | ----> | *s.close()* |

- When we print object reference variable it always prints **hash code** of the object.

## CONSTRUCTORS in JAVA

### Class Vs Object

- Class is a **logical entity** it contains logics whereas object is **physical** entity it is representing memory.
- Class is blue print it decides object creation without class we are unable to create object.
- Based on single class (blue print) it is possible to create multiple objects but every object occupies memory.
- We are declaring the class by using class keyword but we are creating object by using **new** keyword.
- We are able to create object in **different** ways but we are able to declare the class by using **class** keyword.

### Constructor

- ✓ A constructor is a block of codes similar to the method.
- ✓ It is called when an instance of the class is created.
- ✓ Does not have any return type, not even void.
- ✓ The only modifiers applicable for constructor are public, protected, default and private.
- ✓ It executes automatically when we create an object.

### What is the need of java constructor?

- ✓ Java constructor is used to initialize an object.

### What is a copy constructor and its needs?

- ✓ Copy constructor is used to create an exact copy of an object with the same values of an existing object.
- ✓ Copy constructor is used only for initialization.
- ✓ Copy constructor cannot be used to clone objects in case of polymorphism, because copy constructor does not support polymorphism.

### Why you don't use void in constructor?

- ✓ Because, the main work of constructor is to initialize the object. So, there is always a value as return type. That's why constructor has no return type.
- ✓ Also, if I don't declare any constructor, it is created by compiler by default. So, the compiler does not judge the return type.

### Rules to declare constructor

- ✓ Constructor name and class name must be same.
- ✓ It is possible to provide parameters to constructors (just like methods).
- ✓ Constructor not allowed explicit return type. (return type declaration not possible even **void**).

## Types of Constructors

There are two types of constructors,

1. Default Constructor (provided by compiler)
2. User defined Constructor (provided by user) or Parameterized Constructor

### Default Constructor

- ✓ Whenever we create an object, the compiler automatically creates a zero-argument constructor if we do not create a constructor that is called the default constructor.
- ✓ A constructor is called "Default Constructor" when it does not have any parameter.
- ✓ The compiler generated constructor is called **Default constructor**.
- ✓ Inside the class default constructor is in invisible mode.
- ✓ To check default constructor provided by compiler open .class file code by using java decompiler software.

```
//Syntax
<class_name> ()
{
}
```

### What is the purpose of default constructor?

- ✓ The default constructor is used to provide the default value to the object like 0, null etc. depending on the type.

### Application before compilation

- ✓ In the below application when we create object by using new keyword **"Test t = new Test ();"** then compiler is searching for **"Test ()"** constructor inside the class, since not available hence compiler generate default constructor at the time of compilation.
- ✓ The default constructor is always **0-argument** constructor with **empty** implementation.

```
class Test
{
        void m1 ()
        {
                System.out.println("m1 method");
        }
        public static void main (String args [])
        {
                Test t = new Test ();
                t.m1 ();
        }
}
```

Application after compilation

- ✓ In the below example at run time JVM will execute compiler provide default constructor during object creation.

```
class Test
{
        void m1 ()
        {
                System.out.println ("m1 method");
        }
        //default constructor generated by compiler
        Test
        {
        }
        public static void main (String args [])
        {
                Test t = new Test ();
                t.m1 ();
        }
}
```

Example

```
// Java Program to create and call a default constructor
class Bike1
{
        //Creating a default constructor
        Bike1()
        {
                System.out.println("Bike is created");
        }

        //main method
        public static void main (String args[])
        {
                //Calling a default constructor
                Bike1 b = new Bike1();
        }
}

Output:
Bike is created
```

### Does constructor return any value?

- ✓ Yes, it is the current class instance (you cannot use return type yet it returns a value).

### Can constructor perform other tasks instead of initialization?

- ✓ Yes, like object creation, starting a thread, calling a method etc. you can perform any operation in the constructor as you perform in the method.

### Is there constructor class in java?

- ✓ Yes

### What is the purpose of constructor class?

- ✓ Java provides a constructor class which can be used to get the internal information of a constructor in the class. It is found in the java. lang. reflect package.

## Parameterized Constructor

- ✓ A constructor which has specific number of parameters is called parameterized constructor.
- ✓ The constructor which are declared by user are called user defined constructor.

### Why we use the parameterized constructor?

The parameterized constructor is used to provide different values to distinct objects. However, you can provide the same values also.

Example - 1:

```
class Test
{
        //constructor declared by user
        Test ()
        {
                System.out.println ("0-arg constructor");
        }
        Test (int i)
        {
                System.out.println ("1-arg constructor");
        }
        Test (int a, int b)
        {
                System.out.println ("2-arg constructor");
        }
        public static void main (String args [])
        {
                Test t1 = new Test ();
                Test t2 = new Test (10);
                Test t3 = new Test (100, 200);
        }
}
```

Example – 2:

- ✓ Inside the class if we are declaring at least one constructor (either 0-arg or parameterized) then compiler won't generate default constructor.
- ✓ Inside the class if we are not declaring at least one constructor (either 0-arg or parameterized) then compiler will generate default constructor.
- ✓ If we are trying to compile below application the compiler will generate error message "Cannot find symbol" because compiler is unable to generate default constructor.

```
class Test
{
        Test (int i)
        {
                System.out.println ("1-arg constructor");
        }
        Test (int a, int b)
        {
                System.out.println ("2-arg constructor");
        }
        public static void main (String args [])
        {
                Test t1 = new Test ();
                Test t2 = new Test (10);
                Test t3 = new Test (100, 200);
        }
}

E:\javac Test.java
Test.java:9:cannot find symbol
symbol: constructor Test ()
location: class Test
```

Example – 3:

```
// Java Program to create and call a parameterized constructor
class Student
{
        int id;
        String name;

        //Creating a parameterized constructor
        Student (int i, String n)
        {
                id = i;
                name = n;
        }

        //Method to display the values
        void display()
        {
```

```
                System.out.println(id+" "+name);
        }

        public static void main(String args[])
        {
                //Creating objects and passing values
                Student s1 = new Student(111, "Karan");
                Student s2 = new Student(222, "Aryan");
                //Calling method to display the values of object
                s1.display();
                s2.display();
        }
}
111 Karan
222 Aryan
```

## Advantages of Constructor

- ✓ Constructors are used to write block of java code that will be executed during object creation.
- ✓ Constructors are used to initialize instance variable during object creation.

## Note

- ✓ Default constructor is zero argument constructor but all zero argument constructors are not default constructors.
- ✓ To **call** current class constructor use **this** keyword

| | | |
|---|---|---|
| ***this ();*** | ----> | *current class 0-arg constructor calling* |
| ***this (10);*** | ----> | *current class 1-arg constructor calling* |
| ***this (10, true);*** | ----> | *current class 2-arg constructor calling* |
| ***this (10,"ashu", 'a')*** | ----> | *current class 3-arg constructor calling* |

- ✓ Inside the constructor this keyword must be first statement otherwise compiler generate error message **"call to this must be first statement in constructor"**.

No compilation error:- (this keyword first statement)

```
Test()
{       this(10); //current class 1-argument constructor calling

        System.out.println("0 arg");
}
```

Compilation error: - (this keyword not a first statement)

```
Test()
{       System.out.println("0 arg");
        this(10); //current class 1-argument constructor calling

}
```

In java, constructor is able to call only one constructor at a time but not possible to call more than one constructor.

Constructor chaining

- ✓ One constructor is calling same class constructor is called constructor calling.
- ✓ We are achieving constructor calling by using **this** keyword.
- ✓ Inside constructor we are able to declare only one this keyword that must be first statement of the constructor.

Constructor Overloading

- ✓ In Java, a constructor is just like a method but without return type. It can also be overloaded like methods.
- ✓ Constructor overloading in Java is a technique of having more than one constructor with different parameter list. They are arranged in a way that each constructor performs a different task.

Example

```
//Java  Program to overload constructors
class Student
{
        int id;
        String name;
        int age;
        //creating two arg constructor
        Student (int i, String n)
        {
                id = i;
                name = n;
        }
        //creating three arg constructor
        Student (int i, String n, int a)
        {
                id = i;
                name = n;
                age = a;
        }
        void display()
        {
```

```
                System.out.println(id+" "+name+" "+age);
        }
        public static void main(String args[])
        {
                Student s1 = new Student(111, "Kiran");
                Student s2 = new Student(222, "Aryan", 25");
                s1.display();
                s2.display();
        }}
111 Karan 0
222 Aryan 25
```

Difference between Constructor and Method in Java

| Java Constructor | Java Method |
|---|---|
| A constructor is used to initialize the state of an object. | A method is used to expose the behavior of an object. |
| A constructor must not have a return type. | A method must have a return type. |
| The constructor is invoked implicitly. | The method is invoked explicitly. |
| The Java compiler provides a default constructor if you don't have any constructor in a class. | The method is not provided by the compiler in any case. |
| The constructor name must be same as the class name. | The method name may or may not be same as the class name. |

## OOPs (Object Oriented Programming)

- ✓ The first object-oriented programming is: **Simula**, **Smalltalk**.
- ✓ In object-oriented programming language everything represented in the form of object.
- ✓ Object is real world entity that has state & behaviour.
- ✓ Example: such as pen, chair, table, house...etc.
- ✓ Every object contains three characteristics,
  - State : well defined condition of an item (instance variable/fields/properties)
  - Behaviour : effects on an item (methods/behaviour)
  - Identity : identification of an item (hash code)

Example:

Object : Car
State : gear, speed, color...etc
Behaviour : current speed, current gear, Accelerate...etc
Identity : car number

Object : house
State : location
Behaviour : doors open/close
Identity : house no

## What is oops?

It based on

- Real world objects: before java, all the programming language are based on structural or procedural or functional oriented concept, but java brought oops concept. Means that if we need to work on a car or bicycle concept, first we have to create a car class then we have to create a car objects which works around the car.so object-oriented programming based on real world objects.

- State and behavior: suppose in a car the states are may be no of types, mirrors or may be seats etc. and the acceleration and speed are the behavior of the car.

- Objects and data: we have the objects that revolve around the data. so, lets you have a class and data members and a function on that data members. so now when you create an object of that class and the objects can operates on the data members through the functions to manipulate the class.

OOPs is a methodology or paradigm to design a program using classes and objects.

The main concepts in OOPs are,

- **Object**
- **Class**
- **Inheritance**
- **Polymorphism**
- **Abstraction**
- **Encapsulation**

## Advantages:

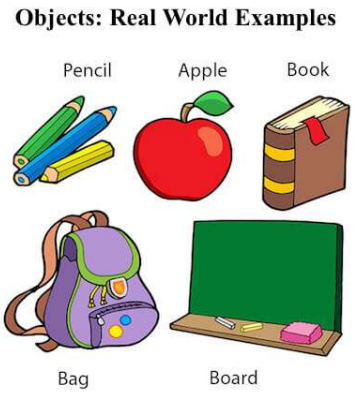
- ✓ OOPs makes development and maintenance easier.
- ✓ OOPs provides data hiding, whereas, in a procedure-oriented language, global data can be accessed from anywhere.
- ✓ OOPs provides the ability to simulate real-world event much more effectively.

## What is the difference between an object-oriented programming language and object-based programming language?

- Object – based programming language follows all the features of OOPs except Inheritance.
- JavaScript and VBScript are examples of object-based programming languages.

## Object in Java

- ✓ An entity that has state and behaviour is known as an object.
- ✓ An object is a real-world entity.
- ✓ The object is an instance of a class.
- ✓ It can be physical or logical.
- ✓ Example: chair, bike, pen, table, car etc.




- ✓ An object has three characteristics;
  - State – color, age
  - Behaviour – eat, run, barking
  - Identity - name
- ✓ Example:
  - pen is an object. Its name is Elkos or Reynolds; color is white, known as its state. It is used to write, so writing is its behaviour.
  - A dog is an object because it has states like color, name, breed etc. as well as behaviour like wagging the tail, barking, eating etc.
  - Banking system

## How to create an object?

- ✓ Object creation syntax
  Class-name reference variable = new class-name ();
  **Test t = new Test ();**
  Test -------→ class name
  t -------→ reference variables
  = -------→ assignment operator
  new -------→ keyword used to create object
  Test () ----→ constructor
  ; -------→ statement terminator
- When we create new instance (object) of a class using new keyword, a constructor for that class is called.
- The **new** keyword is used to create object in java.

There are 5 ways to create an object in Java,

- ✓ New keyword: the object created by 3 steps,

  Declaration: it is declaring a variable name with an object type

  Example: Animal dog; (where "Animal" is ClassName & "dog" is reference_variable_name)

  Instantiation: this is when memory is allocated for an object. The "new" keyword is used to create the object.

  Example: dog = new

  Initialization: the new keyword is followed by a call to a constructor. This call initializes the new objects.

  Example: dog = new Animal (); and finally, the syntax is,

  Animal dog = new Animal ();

  For calling the method through object the syntax is: dog.run ();
- ✓ newInstance () method
- ✓ clone () method
- ✓ deserialization
- ✓ factory method
- ✓ If I declare some method inside the class then when I want to call this method, for that I have to create an object inside the main method.

```
class Animal
{
	public void eat()
	{
		System.out.println("I am eating");
	}

	public static void main(String args[])
	{
		System.out.println(1);
		Animal dog = new Animal();
		dog.eat();
	}
}
```

Initialization of an object (3 ways)

```
By reference variable
// Object Initialization through reference variable by dot operator
class Animal
{
	String color;
	int age;
	public static void main(String args[])
	{
		Animal dog = new Animal();
		dog.color() = "black";
		dog.age() = 10;

		System.out.println(dog.color+" "+dog.age);
	}
}
```

```
By method
// Object Initialization through method
class Animal
{
	String color;
	int age;

	void initObj(String c, int a)
	{
		color = c;
		age = a;
	}
```

```
        void display()
        {
                System.out.println(color+" "+age);
        }
        public static void main(String args[])
        {
                Animal dog = new Animal();
                dog.initObj("black",10);
                dog.display();
        }
}
```

```
By constructor: Done
```

## Class in Java

- ✓ A class is a group of objects which have common properties.
- ✓ It is a template or blueprint from which objects are created.
- ✓ It a logical entity and it can't be physical.
- ✓ Class does not consume any space or memory.
- ✓ A class in Java contains,
  - Fields
  - Methods
  - Constructors
  - Blocks
  - Nested class and interfaces

```
class <class_name>{
  field;
  method;
}
```

```
Object and Class Example
//Java Program to illustrate object and class
package OopsConcept;

//Creating a class named ObjectClassEx
public class ObjectClassEx
{
        // Defining field or data member or instance variable
                int id;
                String name;

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

```
        {
                // Creating an object or instance of class ObjectClassEx
                ObjectClassEx s1 = new ObjectClassEx();

                //Printing values of the object
                System.out.println(s1.id);//accessing member through reference variable
                System.out.println(s1.name);
        }

}
 0
null
```

## Static keyword in Java

- ✓ Static keyword can be used with,
    - Variable (with class level variable) but can't be used with local variable.
    - Methods
    - Block
    - Inner class/nested class but not with outer class.

## Static variable

- ✓ Static keyword is a non – accessible modifier.
- ✓ Static variable belongs to a class not to an object.
- ✓ If a variable is declared as static then we can access it without creating an object through class.

```
//Java Program for static variable
class Test
{
        //Declare a static variable
        static int a = 10;
}
class Demo
{
        public static void main(String[] args)
        {
                System.out.println(Test.a);
        }}
10
```

- ✓ Static variable used for memory management.
- ✓ When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level.
- ✓ The static variable gets memory only once in the class area at the time of class loading.

Example: suppose I'm writing a program for a company that contains details of every employee such as employee id, employee name and company name. as we know the company name is same for all it is not required to define again and again so, we make a static variable for that. Hence a single copy of this variable is created and shared among all objects at class level.

```
//Java Program for common static variable
class Student
{
	int studentID;
	String student_name;
	static String student_clg = "NIT";

	Student (int studentID, String student_name)
	{
		this.studentID = studentID;
		this.student_name  = student_name;
	}

	void display()
	{
		System.out.println(studentID+" "+student_name+" "+student_clg);
	}

	public static void main(String args[])
	{
		Student s1 = new Student(101, "Ashirbad");
		s1.display();
		Student s2 = new Student(102, "Abinash");
		s2.display();
	}
}
101 Ashirbad NIT
102 Abinash NIT
```

## Static Method

- ✓ Static method belongs to class, not to the object.
- ✓ Static method can be called directly by class name as: ClassName.methodName();
- ✓ A static method can access only static data/method. It can't access non-static data/method (instance data/method).
- ✓ A static method cannot refer to "this" or "super" keyword in anyway.
- ✓ Whenever a method is declared as static there is no need to create an object, we can directly call the method. So, that's why use for memory management.

```
//Java Program for static method
class StaticMethod
{
        static void display()
        {
                System.out.println("1");
        }

        public static void main(String args[])
        {
                display();
                // or StaticMethod.display();
        }
}
1
```

## Static block

- ✓ Static block executes automatically when the class is loaded in the memory.
- ✓ Use of static block
    - Static block is executed at class loading, hence at the time of class loading if we want to perform any activity, we have to define that inside static block.
    - Static block is used to initialize the static members.
- ✓ In the below example, the static block always executes first then the main method. If there is more than one static block is present it executes in top to button approach, then the main method.

```
//Java Program for static block
class StaticBlock
{
        static
        {
                System.out.println("Hello");
        }
```

```
        public static void main (String args[])
        {
                System.out.println("I am in main method");
        }

        static
        {
                System.out.println("I am second static block");
        }
}
Hello
I am second static block
I am in main method
```

### Why is the Java main method static?

It is because the object is not required to call a static method. If it were a non-static method, JVM creates an object first then call main () method that will lead the problem of extra memory allocation.

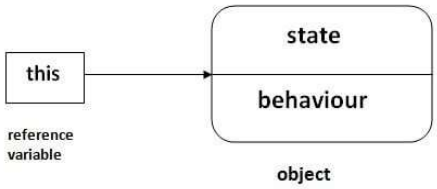
### Can we execute a program without main () method?

Yes, one of the ways was the static block, but it was possible till JDK 1.6, since 1.7, it is not possible to execute a java class without the main method.

```
class A3
{
 static
        {
 System.out.println("static block is invoked");
 System.exit(0);
        }
}
Output:
static block is invoked
Since JDK 1.7 and above, output would be:
Error: Main method not found in class A3, please define the main method as:
public static void main (String [] args)
or a JavaFX application class must extend javafx.application. Application
```

## Java this keyword

This keyword is a reference variable that refers to the current object (that is to the current instance variable).




### Why to use this keyword?

When you declare the instance variable and local variable with the same name, the instance variable always refers as local variable, that's why the problem arises. So, you have to use the "**this**" keyword.

## Java Inheritance

- ✓ The technique of creating a new class by using an existing class functionality is called inheritance.
- ✓ In other words, Inheritance in Java is a process where a child class acquires all the properties and behaviours of the parent class.
- ✓ The existing class is called a parent/base/main/super class and the new class is called a child/derived/sub class.
- ✓ For example, In the real world, a child inherits the features of its parent such as the beauty of mother and intelligence of father.
- ✓ Inheritance represents the **IS-A relationship**, also known as parent-child relationship.
- ✓ The process of acquiring properties (variables) & methods (behaviour) from one class to another class is called inheritance.
- ✓ We are achieving inheritance concept by using **extends** keyword.
- ✓ The main objective of inheritance is code extensibility whenever we are extending the class automatically code is reused.
- ✓ To reduce length of the code and redundancy of the code & to improve reusability of code, inheritance is introduced.

Syntax:

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

- ✓ In Java, if we are extending the class then it will be parent class, if we are not extending the class then **object** class will become parent class.
- ✓ The root class of all java classes is "**object**" class.
- ✓ Every java class contains parent class except **object** class.
- ✓ Object class present in **java.lang** package and it contains 11 methods & all java classes able to use these 11 methods because object class is root class of all java class.

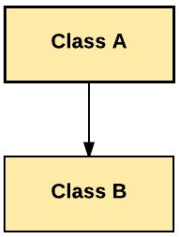
## Types of Inheritance

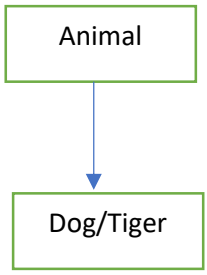
There are five types of inheritance in java,

- ✓ Single Inheritance
- ✓ Multilevel Inheritance
- ✓ Hierarchical Inheritance
- ✓ Multiple Inheritance
- ✓ Hybrid Inheritance

## Single inheritance

- ✓ In single inheritance one class extends another class (one class only).
- ✓ In the diagram, class B extends only class A. class A is super class and class B is a sub-class.



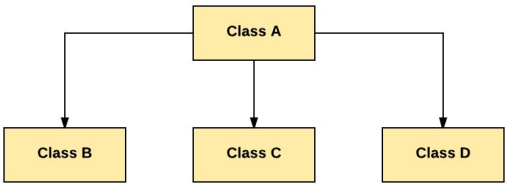



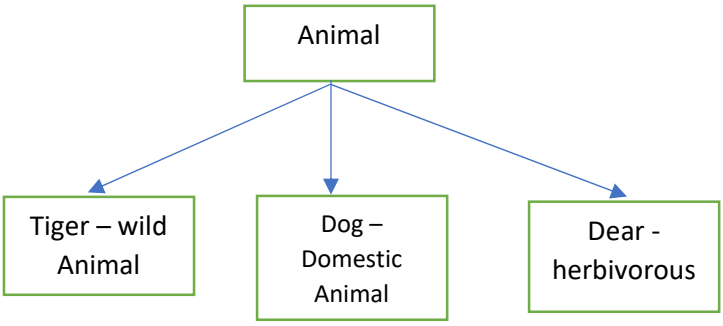

**Example**

```
Class parent
{
}
Class child extends parent
{
}

Example:
Class A
{
}
Class B extends A
```

## Hierarchical Inheritance

- ✓ More than one sub class is extending single parent is called HI.
- ✓ In the diagram, class B, C, and D inherit the same class A.



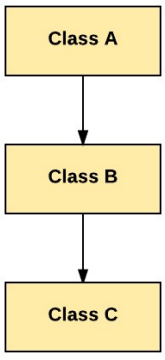



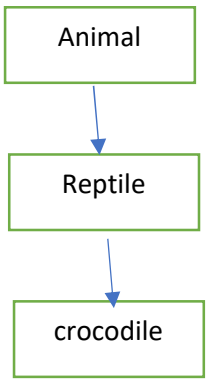

**Example**

```
Class A
{------}
Class B extends A
{------}
Class C extends A
{------}
```

## Multilevel Inheritance

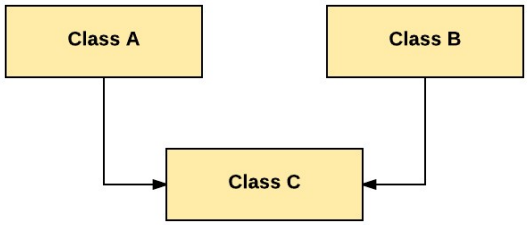
- ✓ One sub class is extending parent class then that sub class will become parent class of next extended class this flow is called multilevel inheritance.
- ✓ In the diagram, class C is subclass of B and B is a of subclass A.

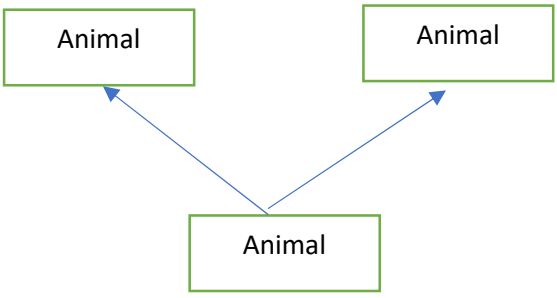


## Multiple Inheritance

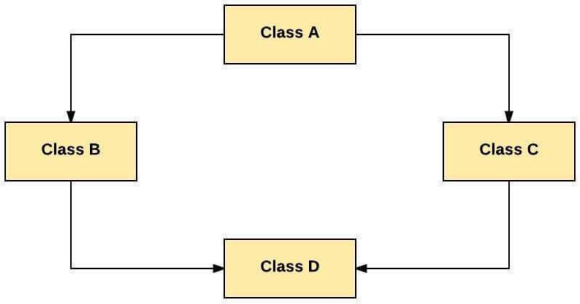
- ✓ In multiple Inheritance, one sub class extending more than one super class. Java does not support multiple inheritance because it is creating ambiguity problems (confusion state).
- ✓ Inheritance is a compile-time mechanism. A parent class can have any number of derived class but a derived class can have only one parent class. Therefore, Java does not support multiple inheritance.
- ✓ But multiple inheritance can be possible by using **interface** because a class can extend only one class but it can implement multiple interfaces.
- ✓ In the diagram, class c extends class A and class B both.

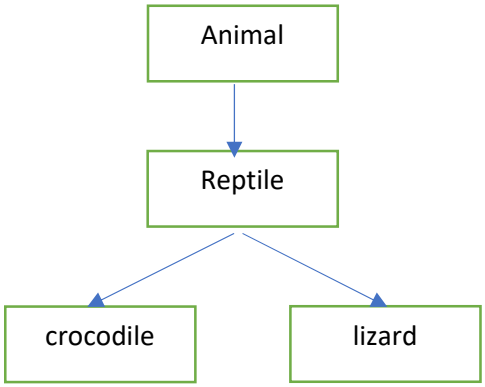



## Hybrid Inheritance

- ✓ Hybrid inheritance is a combination of **single** and **multiple** inheritance.
- ✓ In the diagram, all the public and protected members of class A are inherited into class D, first via class B and secondly via class C.
- ✓ In this type of inheritance, the diamond problem may arise.






Java is **not** supporting hybrid inheritance because multiple inheritance (not supported by java) is included in hybrid inheritance.

## Advantages

- ✓ We can **reuse** the code from the base class or for code reusability.
- ✓ Using inheritance, we can increase the features of class or method by overriding.
- ✓ Inheritance is used to use the existing features of the class.
- ✓ It is used to achieve **runtime polymorphism** i.e. method overriding.

## How is Inheritance implemented/achieved in Java?

- ✓ Inheritance in Java can be implemented or achieved by using two keywords:
  - **Extends**: extends is a keyword that is used for developing the inheritance between two classes and two interfaces. Note that a class always extends another class. An interface always extends another interface and can extends more than one interface.
  - **Implements**: implements keyword is used for developing the inheritance between a class and interface. A class always implements the interface.
- ✓ All the properties and behaviour of a class **does not** inherit to another class. There is some part that does not inherit like;
  - Constructor: because the constructor is not a part of class.
  - Private method/variable: because it will work within that class.
- ✓ Object is the parent class of all the classes of java in inheritance.
- ✓ It is used to achieve runtime polymorphism.
- ✓ If a parent class declared as **final**, we can't create sub class for that class.

**Example:**

```
// Single Inheritance
class Animal
{
        void eat()
        {System.out.println("eating...");}
}
class Dog extends Animal
{
        void bark()
```

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

```
// Multilevel Inheritance
class Animal
{
        void eat()
        {System.out.println("eating...");}
}
class Dog extends Animal
{
        void bark()
        {System.out.println("barking...");}
}
class BabyDog extends Dog
{
        void weep()
        {System.out.println("Weeping...");}
}

class TestInheritance
{
        public static void main(String args[])
        {
                BabyDog d = new BabyDog();
                d.weep();
                d.bark();
                d.eat();
        }
}

weeping...
barking...
eating...
```

```
//Hierarchical Inheritance
class Animal
{
	void eat()
	{
		System.out.println("eating...");
	}
}
class Dog extends Animal
{
	void bark()
	{
		System.out.println("barking...");
	}
}
class Cat extends Animal
{
	void meow()
	{
		System.out.println("meowing...");
	}
}

class TestInheritance
{
	public static void main(String args[])
	{
		Cat c = new Cat();
		c.meow();
		c.eat();
		c.bark();//Error
	}
}
meowing...
eating...
```

### Why multiple inheritance not supported by Java?

- ✓ To reduce the complexity and simplify the language, multiple inheritance is not supported in java.
- ✓ Consider a scenario where A, B, and C are three classes. The C class inherits A and B classes. If A and B classes have the same method and you call it from child class object, there will be ambiguity to call the method of A or B class.
- ✓ Since compile-time errors are better than runtime errors, Java renders compile-time error if you inherit 2 classes. So, whether you have same method or different, there will be compile time error.
- ✓ But it can be possible by using **interface** because a class can extend only one class but can implement multiple interfaces.
- ✓ Also, this is called the **diamond problem**.

Example:

```
// Diamond problem
class A
{
        void msg ()
        {
                System.out.println("Hello");
        }
}
class B
{
        void msg ()
        {
                System.out.println("Welcome");
        }
}
class c extends A, B
{
        public static void main (String args[])
        {
                C obj = new C();
                //Now which msg() method would be invoked?
                obj.msg();
        }
}
Compile Time Error
```

## Instanceof Operator

- ✓ It is used to check the type of the object and it returns boolean values as a return value.
- ✓ **Syntax: reference-variable instanceof class-name;**
- ✓ To use the **instanceof** operator the class name & reference variable must have some relationship either parent to child or child to parent otherwise compiler will generate error message "**inconvertible types**".
- ✓ If the relationship is child to parent it returns **true** & the relationship is parent to child it returns **false**.

## Association

- ✓ Class A uses class B
- ✓ When one object wants another object to perform services for it.
- ✓ Relationship between teacher and student, number of students associated with one teacher or one student can associate with number of teachers. But there is no ownership and both objects have their own life cycles.
- ✓ Association represents the relationship between objects.

Example:

```
//In the below example student uses Teacher class object services
class Teacher
{
        void course ()
        {
                System.out.println("core java");
        }
}
class Student
{
        public static void main (String args [])
        {
                Teacher t = new Teacher ();
                t.course ();
        }
}
```

## Aggregation

Aggregation in Java is a relationship between two classes that is best described as a "**HAS-A**" relationship.

The aggregate class contains a reference to another class and is said to be have ownership of that class.

- ✓ Class A has instance of class B is called aggregation.
- ✓ Class A can exist without presence of class B. **Ex**: - A university can exist without chancellor.

### Examples

Take the relationship between teacher and department. A teacher may belong to multiple departments hence teacher is a part of multiple departments but if we delete department object teacher object will not destroy.

Consider two classes i.e. student class and address class. Every student has an address so the relationship between student and address is a HAS-A relationship. But if you consider its vice versa then it would not make any sense as an address doesn't need to have a student necessarily.

### Why to use aggregation in Java?

To maintain code reusability.

### Example:

Suppose there are two classes College and Staff along with two other classes Student and Address. In order to maintain Student's address, College Address and Staff's address we don't need to use the same code again and again. We just have to use the reference of Address class while defining each of these classes like:

Student Has-A Address (Has-a relationship between student and address)
College Has-A Address (Has-a relationship between college and address)
Staff Has-A Address (Has-a relationship between staff and address)
Hence, we can improve code re-usability by using Aggregation relationship.

```
class Address
{
  int streetNum;
  String city;
  String state;
  String country;
  Address(int street, String c, String st, String coun)
  {
    this.streetNum=street;
    this.city =c;
    this.state = st;
    this.country = coun;
  }
}
class StudentClass
{
  int rollNum;
  String studentName;
  //Creating HAS-A relationship with Address class
  Address studentAddr;
  StudentClass(int roll, String name, Address addr){
    this.rollNum=roll;
    this.studentName=name;
    this.studentAddr = addr;
  }
  ...
}
class College
{
  String collegeName;
  //Creating HAS-A relationship with Address class
  Address collegeAddr;
  College(String name, Address addr){
    this.collegeName = name;
    this.collegeAddr = addr;
  }
  ...
}
class Staff
{
  String employeeName;
  //Creating HAS-A relationship with Address class
  Address employeeAddr;
  Staff(String name, Address addr){
    this.employeeName = name;
    this.employeeAddr = addr;
  }
  ...}
```

## Composition

- ✓ Class A owns class B, it is a strong type of aggregation. There is no meaning of child without parent.
- ✓ The composition represents the relationship where one object contains other objects as a part of its state.
- ✓ Order consists of list of items, without order no meaning of items. **Or** bank consists of transaction history, without bank account no meaning of transaction history **or** without student class no meaning of mark class.
- ✓ Let's take example, house contains multiple rooms, if we delete house object no meaning of room object hence the room object cannot exist without house object.
- ✓ Relationship between question and answer, if there is no meaning of answer without question object hence the answer object cannot exist without question objects.

## Object Delegation

The process of sending request from one object to another object is called object delegation.

Example

```
class RealPerson      //delegate class
{
        void book()
        {
                System.out.println("real java book");
        }
}
class DummyPerson //delegator class
{
        RealPerson r = new RealPerson();
        void book()
        {
                r.book();
        } //delegation
}
class Student
{
        public static void main(String[] args)
        {
                //outside world thinking dummy Person doing work but not.
                DummyPerson d = new DummyPerson (); d.book();
        }
}
```

**Note**

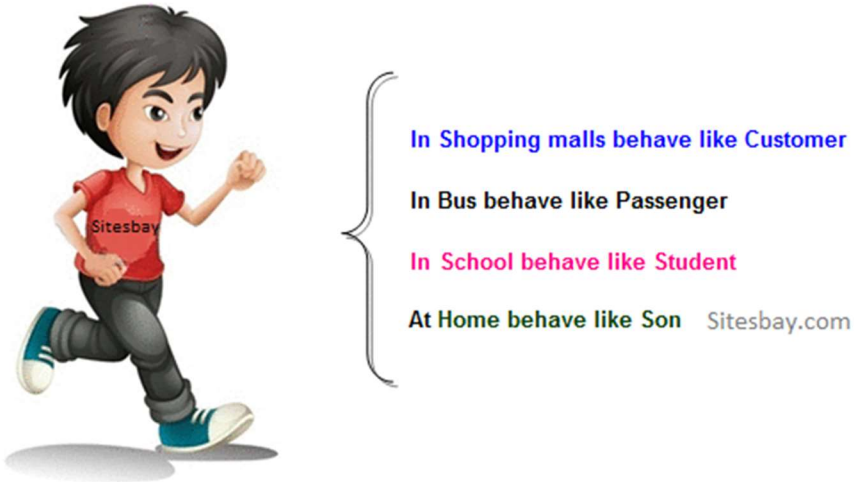
- ✓ Inside the constructor whether it is a 0-argument or parameterized if we are not declaring **super** or **this** keyword then compiler generate super keyword at first line of the constructor.
- ✓ The compiler generated **super** keyword is always 0-arg constructor calling.
- ✓ In parent and child relationship **first** parent class instance blocks are executed then child class instance blocks are executed.
- ✓ In parent and child relationship **first** parent class static blocks are executed only one time then child class static blocks are executed only one time because static blocks are executed with respect to .class loading.
- ✓ Instance block execution **depends** on object creation & static block execution depends on class loading.

## Polymorphism in Java

- ✓ If one task is performed in different ways, it is known as polymorphism.
- ✓ The ability to appear in more forms is called polymorphism.
- ✓ We use method overloading and method overriding to achieve polymorphism.

**Example:**

- ✓ to convince the customer differently
- ✓ to draw something, for example, shape, triangle, rectangle, etc.
- ✓ speak something, for example, a cat speaks meow, dog barks etc.
- ✓ state of water – solid, liquid, gas
- ✓ suppose, consider you in real life then,



## Types of polymorphism

There are two types of polymorphism in java,

Compile time polymorphism/static binding/early binding

- ✓ **Example**: Method Overloading

Run time polymorphism/dynamic binding/late binding

- ✓ **Example**: method Overriding

## Method Overloading

Two methods are said to be in method overloading if,

- ✓ Same name
- ✓ Same class
- ✓ Different arguments by no. of arguments/sequence of arguments/types of arguments

If java class allows more than one method with same name but different number of arguments or same number of arguments but different data types those methods are called overloaded methods.

- Same method name but different number of arguments

  **void m1 (int a) {}**

  **void m1 (int a, int b) {}**

- Same method name & same number of arguments but different data types.

  **void m1 (int a) {}**

  **void m2 (char ch) {}**

To achieve overloading concept one java class is sufficient.

It is possible to overload any number of methods in single java class.

Real time example:

- ✓ Let's consider the basic human function of speaking.
- ✓ Assume, you are supposed just perform the function of talking. Say, you have to tell the story of your day, to a total stranger. Your function will get over pretty quickly. Say, now you are telling the same to your beloved. You will go through more details as compared to the previous one. What has happened here, is, you have performed the same function, but based on the parameter, stranger/beloved your way of implementing the function changed!
- ✓ Suppose I have set my FB profile pic then I can set in many ways like by clicking photo by myself, or from the gallery.

Example: compile time polymorphism achieved through method overloading

```
class Test
{
	//below three methods are overloaded methods.
	void m1(int i)
	{
		System.out.println("int-argument method");
	}
	void m1(int i, int j)
	{
		System.out.println("int, int argument method");
	}
	void m1(char ch)
	{
		System.out.println("char-argument method");
	}
	public static void main(String[] args)
	{
		Test t = new Test();
		t.m1(10);
		t.m1(10,20);
		t.m1('a');
	}
}
```

### Why to use polymorphism:

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in oops occurs when a parent class reference is used to refer to a child class object, it is important to know that the only possible way to access an object is through a reference variable.

### Can we achieve method overloading by changing the return type of method only?

In Java, method overloading is not possible by changing the return type of the method because of ambiguity.

### Can we overload java main () method?

Yes, we can have any number of main methods in a class by method overloading. This is because JVM always calls main () method which receives string as arguments only.

Example:

```
class Test
{
	public static void main(String args[])
	{
		System.out.println("1");
		Test t = new Test();
```

```
                t.main(20);
        }
        public static void main(int a)
        {
                System.out.println("2");
        }
}
```

**Note**

- ✓ In method overloading it is possible to have different data types for overloaded method.
- ✓ While overloading the methods check the signature (method name + parameters) of the method but not return type.

## Types of Overloading

There are three types of overloading in Java,

1. Method overloading (explicitly by the programmer)
2. Constructor overloading
3. Operator overloading (implicitly by the JVM) ('+' addition & concatenation)

## Constructor Overloading

If the class contains more than one constructor with same name but different arguments or same number of arguments with different data types those constructors are called overloaded constructors.

- ✓ Same constructors name but different number of arguments

  **Test (int a) {} //assume Test is java class**

  **Test (int a, int b) {}**

- ✓ Same constructor name & same no. of arguments but different data types.

  **Test (int a) {}**

  **Test (char ch) {}**

**Example**

```
class Test
{       //overloaded constructors
        Test ()
        {
                System.out.println("0-arg constructor");
        }
        Test (int i)
```

```
        {
                System.out.println("int argument constructor");
        }
        Test (char ch, int i)
        {
                System.out.println("char, int argument constructor");
        }
        Test (char ch)
        {
        System.out.println("char argument constructor");
        }
        public static void main (String [] args)
        {
                new Test ();
                new Test (10);
                new Test('a',100);
                new Test('r');
        }
}
```

## Method overriding (Runtime Polymorphism)

- ✓ Two methods are said to be method if,
  - Same name
  - Different class
  - Same argument by no. of arguments/ types of arguments/ sequence of arguments
  - Existence of Inheritance
- ✓ To achieve method overloading one java class is sufficient but to achieve method overriding we required two java classes with parent and child relationship.
- ✓ The method implementation already presents in parent class,
  - If the child class required that implementation then access those implementations.
  - If child class not required those implementations then override parent class method in child class to provide child specific implementations.
- ✓ The sub class overrides super class method to provide sub class method implementation.
- ✓ To overriding,
  - Parent class method is called --- **overridden method**
  - Child class method is called ----- **overriding method**

Real time example:

- ✓ A son inherits his father's public properties e.g. home and car and using it. At later point of time, he decides to buy and use his own car, but still he wants to use his father's home. So, he can use method overring feature and use his own car.
- ✓ Suppose we have camera of 5 MP in mobile, but we want more MP in camera, then we have to implement some new things with the existing camera to make it 10 MP or 25 MP which is the real time example of method overriding.

Example

In the below example marry method present in parent class with some implementation but child class overriding marry method to provide child specific implementation is called overriding.

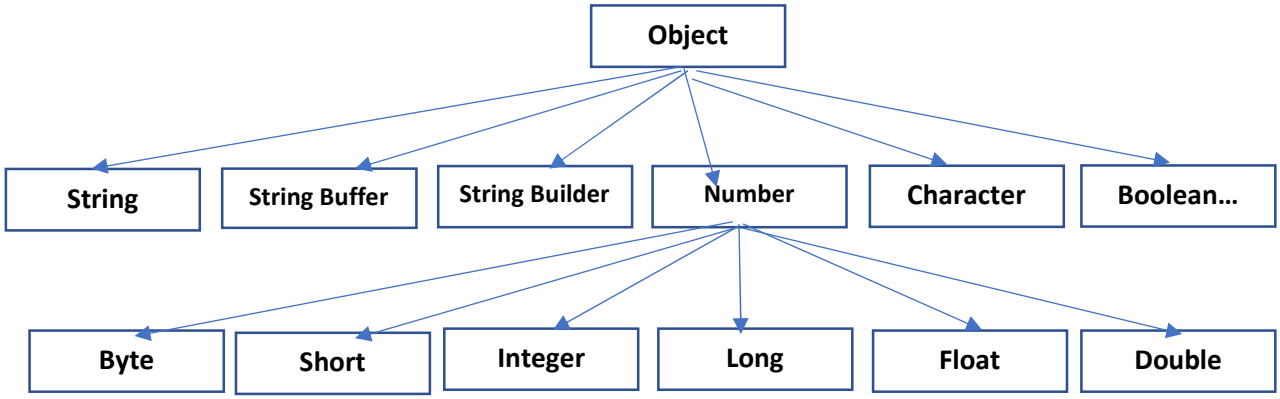
```
class Parent
{
        void property ()
        {
                System.out.println("money +land +house");
        }
        void marry ()          //overridden method
        {
                System.out.println("black girl");
        }
};
class Child extends Parent
{
        void marry ()          //overriding method
        {
                System.out.println("white girl/red girl");
        }
        public static void main (String [] args)
        {
                Child c=new Child();
                c.property();
                c.marry();
        }
};

E:\>java Child
Money +land +house
white girl/red girl
```

While overriding methods must follow these rules

- ✓ Overridden method signature & overriding method signatures must be same.
- ✓ The return types overridden method & overriding method must be same.
- ✓ While overriding it is possible to change return type by using co-variant return types concept.
- ✓ Final methods can't override.
- ✓ Static method can't override but method hiding is possible.
- ✓ Private methods can't override.
- ✓ While overriding it is not possible to maintain same level permission or increasing order but not decreasing.
- ✓ Overriding with exception handling rules.

### Do overriding method must have same return type (or subtype)?

- ✓ From Java 5.0 onwards it is possible to have different return type for overriding method in child class, but child's return type should be sub-type of parent's return type. This phenomenon is known as **covariant return** type.
- ✓ But before JDK 5.0, it was not possible to override a method by changing the return type.
- ✓ **Example**: parent has object return type, then child has string return type.



## Type casting

The process of converting data one type to another type is called type casting.

There are two types type casting

- ✓ Implicit type casting/widening/up casting
- ✓ Explicit type casting/narrowing/down casting

When we assign higher value to lower data type range then will rise compiler error "possible loss of precision" but whenever we are typecasting **higher data type – lower data type** compiler won't generate error message but we will loss the data.

### Implicit type- casting

- ✓ When we assign a lower data type value to higher data type that type-casting is called up casting.
- ✓ When we perform up casting, no data will be loss.
- ✓ It is also known as up-casting or widening.
- ✓ Compiler is responsible to perform implicit typecasting.

### Explicit type-casting

- ✓ When we assign a higher data type value to lower data type that type-casting is called down casting.
- ✓ When we perform down casting data will be loss.
- ✓ It is also known as narrowing or down casting.
- ✓ User is responsible to perform explicit typecasting.

### Note

- ✓ Parent class reference variable is able to hold child class object but child class reference variable is unable to hold parent class object.
- ✓ In Java, it is possible to override methods in child classes but it is not possible to override variables in child classes.
- ✓ If a subclass defines a static method with the same signature as a static method in the superclass, then the method in the subclass hides the one in the superclass.
- ✓ **Static** method cannot be overridden because static method bounded with class whereas instance methods are bounded with object.

## toString ()

- ✓ toString () method present in object and it is printing string representation of object.
- ✓ toString () method return type is String object it means toString method is returning string object.
- ✓ Whenever you are printing reference variable internally **toString ()** method is called.

Example:

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

```
class Test
{
	public static void main (String [] args)
		{
			Test t = new Test ();
			System.out.println(t);
			System.out.println(t.toString()); // [Object class toString() executed]
		}
};
```

### What is the relationship between Override and access-modifier?

- ✓ The access modifier for an overriding method can allow more, but not less, access than the overridden method.
- ✓ For example, a protected instance method in the super-class can be made public, but not private, in the subclass. Doing so, will generate compile time error.

Example:

```
class Test1
{
	void show()
	{
		System.out.println("1");
	}
}
class Test2 extends Test1
{
	public void show()
	{
```

```
                System.out.println("2");
        }
        public static void main(String args[])
        {
                Test1 t1 = new Test1();
                t1.show();

                Test2 t2 = new Test2();
                t2.show();
        }
}
```

## Why can we not override,

### Final methods cannot be overridden:

- ✓ If we don't want a method to be overridden, we declare it as final.

### Static methods cannot be overridden:

- ✓ (Method overriding vs method hiding) when you define a static method with same signature as a static method in base class. It is known as method hiding.
- ✓ Static method cannot be overridden because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

### Private methods cannot be overridden:

- ✓ Private methods cannot be overridden as they are bonded during compile time.

### Can we override java main method?

- ✓ No, because the main is a static method.

## Garbage Collector

- ✓ In Java, garbage collector is a special program which runs on JVM and removes objects which are not in used.
- ✓ JVM calls garbage collector by default or you can also invoke explicitly with **System.gc ()** or **Runtime.gc** commands.
- ✓ Garbage collector is destroying the useless object and it is a part of the JVM.
- ✓ There are three ways to make eligible objects to garbage collector.
- ✓ If any exceptions raised in finalize () method those exceptions are ignored & object is destroyed.
- ✓ To call the garbage collector explicitly use gc () method it is a static method system class.
- ✓ Just before destroying object garbage collector will call finalize () method.

Example-1

Whenever we are assigning null constants to our objects then objects are eligible for garbage collector.

```
class Test
{
        public static void main(String[] args)
        {
                Test t1=new Test();
                Test t2=new Test();
                System.out.println(t1);
                System.out.println(t2);
      ----------------------
                t1=null;//t1 object is eligible for Garbage collector
                t2=null;//t2 object is eligible for Garbage Collector
                System.out.println(t1);
                System.out.println(t2);
        }
};
```

Example-2

Whenever we reassign the reference variable the objects are automatically eligible for garbage collector.

```
class Test
{
        public static void main(String[] args)
        {
        Test t1=new Test();
        Test t2=new Test();
        System.out.println(t1);
        System.out.println(t2);
        t1=t2;//reassign reference variable then one object is destroyed.
        System.out.println(t1);
        System.out.println(t2);
        }
};
```

Example-3

Whenever we are creating objects inside the methods one methods one method is completed the objects are eligible for garbage collector.

```
class Test
{
        void m1()
        {
                Test t1=new Test ();
        }
        public static void main (String [] args)
        {
        Test t=new Test();
        t.m1();
        System.gc ();
        }
};
```

Super keyword

Super keyword is a reference variable which is used to refer immediate parent class object.

Used to access the parent class constructor from child class.


```
class A
{
        int i = 10;
}
class B extends A
{
        int i = 20;
        void show(int i)
```

```
        {
                //Super Keyword
                System.out.println(super.i);
                //this keyword
                System.out.println(this.i);
                //Normal
                System.out.println(i);
        }
        public static void main (String args[])
        {
                B b = new B();
                b.show(30);}}
Output: 10 20 30
```

## Final keyword

The final keyword is used to restrict the user.

Final is the modifier applicable for classes, methods and variables.

Every method present inside a final class is always final but every variable present inside the final class not be final variable.

Final variable always needs initialization, if you don't initialize compiler will generate compilation error "**variable a might not have been initialized**".

The final keyword can be used in many contexts like final can be;

- ✓ **Variable**: if a variable declared as a final, we cannot reassign that variable, if we are trying to reassign compiler generate error message.
- ✓ **Method**: if a method declared as final, we cannot override that method in child class.
- ✓ **Class**: if the class is declared as final, then we cannot inherit that class it means we cannot create child class for that final class.

```
//final keyword used for variable
class Test
{
        public static void main(String args[])
        {
                final int i = 10;
                i = i + 20;// it shows error because of final keyword
                System.out.println(i);
        }
}
```

```
//final keyword used for method
class Test
{
```

```
        final void m1()
        {
                System.out.println("I am in class Demo");
        }
        class Test extends Demo
        {
                void m1()
                {
                        System.out.println("I am in class Test");
                }
                public static void main(String args[])
                {
                        //code
                }
        }
}
//final keyword used for class
final class Demo
{
        //Code snippet
}
class Test extends Demo//here inheritance don't work due to final keyword
{
        public static void main(String args[])
        {
                //code
        }
}
```

### Is final method inherited?

Yes, final method is inherited but you cannot override it.

### What is blank or uninitialized final variable?

A final variable that is not initialized at the time of declaration is known as blank variable.

If you want to create a variable that is initialized at the time of creating object and once initialized may not be changed. It is useful.

Example: PAN card number of an employee

### Can we initialize blank final variable?

Yes, but only in constructor

### Can we declare a constructor final?

No, because constructor is never inherited.

## Runtime polymorphism in Java

Runtime polymorphism or dynamic method dispatch is a process in which a call to an overridden method is resolved at runtime rather than compile time.

In this process, an overridden method is called through the reference variable of a superclass.

### Upcasting

If the reference variable of parent class refers to the object of child class, it is known as upcasting.

```
Class A {}
Class B extends A {}
A a = new B (); //upcasting
```

### Down casting

If the reference variable of child class refers to the object of parent class, it is known as down casting.

```
Dog d = new Animal();//Compilation error
Dog d = (Dog)new Animal();
//Compiles successfully but ClassCastException is thrown at runtime

But if we use the Instanceof operator it runs successfully.
```

### Example

In this example, we are creating two classes Bike and Splendor. Splendor class extends Bike class and overrides its run () method. We are calling the run method by the reference variable of Parent class. Since it refers to the subclass object and subclass method overrides the Parent class method, the subclass method is invoked at runtime.

Since method invocation is determined by the JVM not compiler, it is known as runtime polymorphism.

```
class Bike
{
        void run()
        {
                System.out.println("running");
        }
}
class Splendor extends Bike
{
        void run()
        {
                System.out.println("running fast");
        }
        public static void main(String args[])
```

```
	{
		Bike b = new Splendor();//upcasting
		b.run();
	}
}
```

## Static Binding & Dynamic Binding

Connecting a method call to the method body is known as binding.

There are two types of binding

- Static Binding (Early binding)
- Dynamic Binding (Late binding)

### Static Binding

When type of the object is determined at compiled time (by the compiler), it is known as static binding.

If there is any private, final or static method in a class, there is static binding.

Example

```
//Static Binding
class Dog
{
	private void eat()
	{
		System.out.println("dog is eating...");
	}
	public static void main(String args[])
	{
		Dog d1 = new Dog();
		d1.eat();
	}

}
```

## Dynamic Binding

When the type of the object is determined at runtime, it is known as dynamic binding.

In the example object type cannot be determined by the compiler, because the instance of Dog is also an instance of Animal. So, compiler doesn't know its type, only its base type.

```
//Dynamic Binding
class Animal
{
        void eat()
        {
                System.out.println("animal is eating...");
        }
}
class Dog extends Animal
{
        void eat()
        {
                System.out.println("dog is eating...");
        }
        public static void main(String args[])
        {
                Animal a = new Dog();
                a.eat();
        }
}

Output: dog is eating...
```

## Java Abstraction

Hiding internal details and showing functionality is called abstraction.

Abstraction is hiding internal implementation and just highlighting the necessary things or the setup services that we are offering.

Hiding the data is known as abstraction.

We use abstract class because it allows you to provide default functionality for the subclasses compared to interfaces.

Real world example:

- ✓ When we ride a bike, we only know about how to ride bikes but cannot know about how it works? Ans also do not know the internal functionality of a bike.
- ✓ All are performing operations on the ATM machine like cash withdrawal, money transfer, retrieve mini-statement...etc. but we don't know the internal details about ATM.
- ✓ Phone call, we don't know the internal processing.

**Abstraction in Java achieved by two ways,**

Abstract class (0 – 100%)

- ✓ A method without body (no implementation) is known as abstract method.
- ✓ If a class has an abstract method, then it should be declared as abstract class but not vice versa.
- ✓ If a regular class extends an abstract class, then the class must have to implement all the abstract methods of abstract parent class or it has to be declared abstract as well.
- ✓ Abstract methods in an abstract class are meant to be overridden in derived concrete classes otherwise compile-time error will be thrown.
- ✓ Abstract classes cannot be instantiated, means we can't create an object of abstract class because the abstract method have empty body, but we can create the reference.

Example

```
abstract class Vehicle
{
        abstract void start();
}
class Car extends Vehicle
{
        void start()
        {
                System.out.println("car starts with keys");
        }
}
class Scooter extends Vehicle
{
        void start()
```

```
        {
                System.out.println("Scooter starts with kick");
        }
        public static void main(String args[])
        {
                Car c = new Car();
                c.start();

                Scooter s = new Scooter();
                s.start();
        }
}
```

## Interface (100%)

- ✓ Interface are blueprint of class.
- ✓ Interface is also one of the types of the class, it contains only abstract methods.
- ✓ Interfaces are not alternative for abstract class it is **extension** for abstract classes.
- ✓ The interface allows to declare **only abstract methods** and these methods are by default public & abstract if we are declaring or not.
- ✓ The interface is highlighting set of **functionalities** but **implementations** are hiding.
- ✓ For the interfaces also compiler will generate **.class** files after compilation.
- ✓ It specifies what a class **must** do but not how.
- ✓ Inside the interface all the methods are abstract and public.
- ✓ Inside the source file it is possible to declare any number of interfaces. And we are declaring the interfaces by using **interface** keyword.

### Why to use Java Interface?

- ✓ It is used to achieve abstraction.
- ✓ It is used achieve multiple inheritance.
- ✓ It can be used to achieve loose coupling.

### How to declare an interface?

An interface is declared by using the interface keyword. It provides total abstraction; means all the methods in an interface are declared with the empty body, and all the fields are public, static and final by default. A class that implements an interface must implement all the methods declared in the interface.

```
//syntax
interface <interface_name>{

  // declare constant fields
  // declare methods that abstract
  // by default.
}
```

Realtime example:

A remote control is a type of interface, you deal with the buttons to operate it, you have no idea what kind of circuitry (implementation) it is hiding. The company can send you an upgraded version with exactly same interface and you won't know the difference (except may be improvements in response time, for instance).

Example

In this example, the printable interface has only one method, and its implementation is provided in class A.

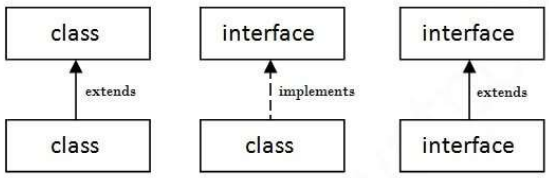
```
interface printable
{
	void print();
}
class A implements printable
{
	public void print()
	{
		System.out.println("Hello");
	}
	public static void main(String args[])
	{
		A obj = new A();
		obj.print();
	}
}
Hello
```

This example shows the implementation of bank interface.

```
interface Bank
{
	float rateOfInterest();
}
class SBI implements Bank
{
	public float rateOfInterest()
	{return 9.15f;}
}
class PNB implements Bank
{
	public float rateOfInterest()
	{return 9.7f;}
}
class TestInterface
{
	public static void main(String args[])
	{
		Bank b = new SBI();
```

```
                System.out.println("ROI:"+b.rateOfInterest());
        }
}
ROI: 9.15
```

Relationship between class and interface:




**Note:**

- ✓ If you don't know anything about implementation just, we have the requirement specification then declare that requirement by using **interface**.
- ✓ If you know the implementation but not completely then we should go for **abstract** classes.
- ✓ If you know the implementation completely then we should go for **concrete** classes.
- ✓ If we are declaring or not, each and every interface method by default **public abstract**. And the interface is by default abstract hence for the interfaces object creation is not possible.
- ✓ Inside the interfaces it is possible to declare variables and methods.
- ✓ By default, interface methods are **public abstract** and by default interface variables are **public static final**.
- ✓ The final variables are replaced with their values at **compilation** time only.
- ✓ Marker Interface: an interface that has no members (methods and variables) is known as marker interface or tagged interface or ability interface. (Ex: - serializable, cloneable, RandomAccess...etc)
- ✓ User defined empty interfaces are not a marker interfaces only, predefined empty interfaces are marker interfaces.
- ✓ The process of creating exactly duplicate object is called **cloning**. We are creating cloned object by using **clone ()**.
- ✓ Adaptor class: it is an intermediate class between the interface and user defined class. And it contains empty implementation of interface methods.

## Multiple inheritance is not supported through class in Java, but it is possible by an interface, why?

Multiple inheritance not supported in the case of class because of ambiguity. However, it is supported in case of an interface because there is no ambiguity. It is because its implementation is provided by the implementation class.

Example

```
Interface I1
{
        void show();
}
Interface I2
{
        void display();
}
class Test implements I1,I2
{
        public void show()
        {System.out.println("1");}
        public void display()
        {System.out.println("2");}
        public static void main(String args[])
        {
                Test t = new Test();
                t.show();
                t.display();
        }
}
1 2
```

**Note:**

- ✓ **Normal method** is a method which contains method declaration as well as method implementation.
- ✓ The method which is having only method declaration but not method implementation such type methods are called **abstract methods**.
- ✓ Normal class is an ordinary java class it contains only normal methods if we are trying to declare at least one abstract method that class will become abstract class.
- ✓ Abstract modifier is applicable for methods and classes but not for variables.
- ✓ To represent particular class is abstract class and particular method is abstract method to the compiler use abstract modifier.
- ✓ Abstract classes are partially implemented classes hence object creation is not possible.
- ✓ The process of highlighting the set of services and hiding the internal implementation is called **abstraction**.
- ✓ **Bank** ATM screens hiding the internal implementation and highlighting set of services like money transfer, mobile registration...etc.
- ✓ **Syllabus** copy of institute just highlighting the contents of java but implementation there in class rooms.
- ✓ We are achieving abstraction concept by using **abstract class** & **interfaces**.

Difference between abstract class and interface

| Interface | Abstract class |
|---|---|
| Interface support multiple inheritance | Abstract class does not support multiple inheritance |
| Interface does'n Contains Data Member | Abstract class contains Data Member |
| Interface does'n contains Cunstructors | Abstract class contains Cunstructors |
| An interface Contains only incomplete member (signature of member) | An abstract class Contains both incomplete (abstract) and complete member |
| An interface cannot have access modifiers by default everything is assumed as public | An abstract class can contain access modifiers for the subs, functions, properties |
| Member of interface can not be Static | Only Complete Member of abstract class can be Static |

## Encapsulation in Java

- The process of binding data and corresponding methods together into a single unit is called as encapsulation in Java.
- Encapsulation is a technique for protecting data from misuse by the outside world, which is referred as "Information Hiding" or "Data Hiding".
- To achieve encapsulation, we have to make all the data member as **private** and make all the methods as **public** or use getter and setter.
- Every Java class is an example of encapsulation because we write everything within the class only that binds variables and methods together and hides their complexity from other classes.

### Example

Example of encapsulation is a capsule. Basically, capsule encapsulate several combinations of medicine. If combination of medicine are variables and methods then the capsule will act as a class and the whole process is called Encapsulation.

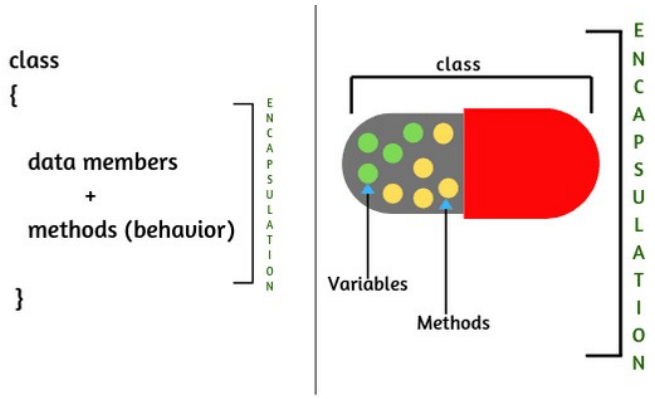



Fig: Encapsulation

### Real-time Example of Encapsulation

1. School bag is one of the most real examples of Encapsulation. School bag can keep our books, pens, etc.
2. When you log into your email account such as Gmail, Yahoo mail, or Rediff mail, there is a lot of internal processes taking place in the backend and you have no control over it. When you enter the password for logging, they are retrieved in an encrypted form and verified and then you are given the access to your account. You do not have control over it that how the password has been verified. Thus, it keeps our account safe from being misused.

### How to achieve or implement Encapsulation in Java?

- ✓ Declaring the instance variable of the class as private. So, that it cannot be access directly by anyone from outside the class.
- ✓ Provide the public setter (we can modify the data) and getter (we can view the data) methods in the class to set/modify the values of the variable.

## Advantages

- ✓ The encapsulated code is more flexible and easier to change with new requirements.
- ✓ It prevents the other classes to access private fields.
- ✓ It keeps the data and codes safe from external world.

## Disadvantages

The main disadvantage of the encapsulation in Java is it increases the length of the code and slow shutdown execution.

## Coding Example

```
class Employee
{
	private int empid;
	public void setEmpid(int eid)
	{
		empid = eid;
	}
	public int getEmpid()
	{
		return empid;
	}
}
class Company
{
	public static void main(String args[])
	{
		Employee e = new Employee();
		e.setEmpid(101);
		System.out.println(e.getEmpid());
	}
}

101
```

## Command Line Arguments

- ✓ The arguments which are passed from command prompt to application at runtime are called command line arguments.
- ✓ The command line arguments separator is space.
- ✓ In single command line argument, it is possible to provide space by declaring that command line argument in double quotes.

**Example**

```
class Test
{
        public static void main(String[] ratan)
        {
                System.out.println(ratan[0]+" "+ratan[1]);   //printing command line arguments
                System.out.println(ratan[0]+ratan[1]);
                //conversion of String-int String-double
                int a = Integer.parseInt(ratan[0]);
                double d = Double.parseDouble(ratan[1]);
                System.out.println(a+d);
        }
};
```

```
D:\>java Test 100 200
100 200
100200
300.0
```

**Note:**

- ✓ To provide the command line arguments with spaces then take that command line argument within double quotes.
- ✓ It allows the methods to take any number of parameters to particular method.
  **Syntax:**
  **Void m1 (int...a) {-----} (only 3 dots)**
  The above m1 () is allows any number of parameters (0 or 1 or 2 or 3...n).
- ✓ For java methods, it is possible to provide normal arguments along with variable arguments.
- ✓ Inside the method it is possible to declare only one variable-arguments and that must be last argument otherwise the compiler will generate compilation error.
- ✓ If the call contains both var – arg method & normal argument method then it prints normal argument value.

## Wrapper classes in Java

- ✓ Wrapper class is used to convert primitive into object and object into primitive.
- ✓ Java is an object-oriented programming language so represent everything in the form of the object, but java supports 8 primitive data types these all are not the part of object.
- ✓ To represent 8 primitive data types in the form of object, we required 8 java classes, these are called wrapper classes.
- ✓ All wrapper classes are **immutable** classes.
- ✓ To create wrapper objects all most all wrapper classes contain two constructors (Ex – byte, string or short, string…) but **float** contains three constructors (float, double, string) & **char** contains one constructor (char).
- ✓ The eight classes of the **java. lang** package are known as **wrapper** classes in Java. They are,

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

## toString

- ✓ toString () method present in object class it returns **class-name@hashcode**.
- ✓ String, StringBuffer classes are overriding toString () method, it returns content of the object.
- ✓ All wrapper classes overriding toString () method to return content of the wrapper class objects.
- ✓ In Java, we are able to call toString () method only on reference type but not primitive type.
- ✓ If we are calling toString () method on primitive type then compiler generate error message.

### Example

- ✓ In the below example for the integer constructor, we are passing "**1000**" value in the form of string it is automatically converted into integer format.
- ✓ In the below example for the integer constructor we are passing "**ten**" in the form of string but this string is unable to convert into integer format it generate exception "**java.lang.NumberFormatException**".

```
class Test
{
	public static void main(String[] args)
	{
		Integer i1 = new Integer(100);
		System.out.println(i1);
		System.out.println(i1.toString());

		Integer i2 = new Integer("1000");
		System.out.println(i2);
		System.out.println(i2.toString());

		Integer i3 = new Integer("ten");
		//java.lang.NumberFormatException
		System.out.println(i3);
	}
}
```

## valueOf ()

- ✓ In Java, we are able to create wrapper object in two ways,
  - By using constructor
  - By using valueOf () method
- ✓ valueOf () method is used to create wrapper object just it is alternate to constructor and it is a static method present in wrapper classes.

Example

```
class Test
{
	public static void main(String[] args)
	{
		//constructor approach to create wrapper object
		Integer i1 = new Integer(100);
		System.out.println(i1);

		Integer i2 = new Integer("100");
		System.out.println(i2);

		//valueOf() method to create Wrapper object
		Integer a1 = Integer.valueOf(10);
		System.out.println(a1);

		Integer a2 = Integer.valueOf("1000");
		System.out.println(a2);
	}
}
```

## xxxValue ()

- ✓ It is used to convert wrapper object into corresponding primitive value.

## parseXXX ()

- ✓ It is used to convert string into corresponding primitive value & it is a static method present in wrapper classes.

## Autoboxing

The conversion of primitive data type into its corresponding wrapper class is known as autoboxing.

For **example**, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Since Java 5, we do not need to use ValueOf () method of wrapper classes to convert the primitive into object.

Example:

```
//Java program to convert primitive into objects
//Autoboxing example of int to Integer
public class Autoboxing
{
	public static void main(String args[])
	{
		int a = 20;
		//converting int into Integer explicitly
		Integer i = Integer.valueOf(a);
		//autoboxing, now compiler will write Integer.valueOf(a) internally
		Integer j = a;

		System.out.println(a+" "+i+" "+j);
	}
}
 20 20 20
```

## Unboxing

The conversion of wrapper type into corresponding primitive type is known as unboxing.

It is the reverse process of autoboxing.

Since java 5, we do not need to use the intValue () method of wrapper classes to convert the wrapper type into primitives.

**Example:**

```
//Java program to convert object into primitives
//Unboxing example of Integer to int
public class Unboxing
{
	public static void main(String args[])
	{
		Integer a = new Integer(3);
		//converting Integer to int explicitly
		int i = a.intValue();
		//Unboxing, now compiler will write a.intValue() internally
		int j = a;

		System.out.println(a+" "+i+" "+j);
	}
}
 3 3 3
```

## Conversion of datatypes:

1. **Primitive -----→ wrapper object**
   Integer i = Integer. ValueOf (100);
2. **Wrapper object --------→ primitive**
   byte b = i.byteValue ();
3. **String value ---------→ primitive**
   String str = "100";
   int a = Integer.parseInt (str);
4. **Primitive value ---------→ String object**
   int a = 100;
   int b = 200;
   String s1 = String. ValueOf (a);
   String s2 = String. ValueOf (b);
   System.out.println (s1+s2); //100200
5. **String value ------→ wrapper object**
   Integer i = Integer. ValueOf ("1000");
6. **Wrapper object ---------→ String object**
   Integer i = new Integer (1000);
   String s = i.toString ();

## Factory method

- ✓ One java class method returns same class object or different class object is called factory method.
- ✓ There are three types of factory methods in java.
  - Instance factory method
  - Static factory method
  - Pattern factory method
- ✓ The factory is called by using class name is called **static** factory method.
- ✓ The factory is called by using reference variable is called **instance** factory method.
- ✓ One java class method is returning different class object is called **pattern** factory method.

## Java Strictfp Modifier

- ✓ Java strictfp keyword ensures that you will get the same result on every platform if you perform operations in the floating-point variable.
- ✓ The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that you get same result on every platform. So, now you have better control over the floating-point arithmetic.
- ✓ Strictfp is a modifier applicable for classes and methods.
- ✓ If a class is declared as strictfp then every method in that class will follow IEEE754 standard so we will get platform independent results.

Legal code for strictfp keyword

```
strictfp class A {} //strictfp applied on class
strictfp interface M {} //strictfp applied on interface

class A {
strictfp void m () {} //strictfp applied on method
}
```

But cannot be applied on abstract methods, variables or constructors.

**Native modifier**

- ✓ Native is the modifier applicable only for methods.
- ✓ Native method is used to represent particular method implementations there in non-java code (other languages like C, CPP).
- ✓ Native methods are also known as "foreign methods".

Explain public static void main (String args [])?

**Public** ---→ To provide access permission to the JVM declare main method with public.

**Static** ---→ To provide direct access permission to the JVM declare main is static (without object creation able to access main method).

**Void** ----→ Don't return any values to the JVM.

**String [] args** ---→ used to take command line arguments (the arguments passed from command prompt).

**String** ----→ it is possible to take any type of arguments.

**[]** --------→ represent possible to take any number of arguments.

Note:

- ✓ Modifiers order is not important it means it is possible to take **public static** or **static public**.
- ✓ The following declarations are valid
  **String[] args**
  **String []args**
  **String args []**
- ✓ Instead of args it is possible to take any variable name (a, b, c ... etc).
- ✓ For 1.5 version instead of String[] args it is possible to take string...args (only three dots represent variable argument).
  **Public static void main (String...args)**
- ✓ The applicable modifiers on main method.
  Public
  Static
  Final
  Strictfp
  synchronized

## Exception Handling in Java

### Exception

Exceptions are caused by our program and these are recoverable.

**Example**: for example, if our program requirement is to read data from a remote file locating at London, at runtime if the London file is not available then we will get **FileNotFoundException**.

### Error

Most of the time errors are not caused by our program, these are cause due to lack of system resources.

Errors are non-recoverable.

**Example**: for example, if out of memory error occurs, being a programmer, we can't do anything and the program will be terminated abnormally.

### Exception

- Abnormal termination of code
- Next of the code will not executed

Two ways to handle exception

- Try-catch block: used to write the try-catch block
- Throws keyword: used to delegate the exception

We mainly use exception handling for the normal termination of the program code.

**There are 3 types exception such as,**

  - Checked exception
  - Unchecked exception
  - Error

Whether it is a checked exception or unchecked exception, exception obviously raised at runtime but not the compile time.

The safest exception is checked exception because it shows some information at compile time.

Exceptions are caused due to several reasons but error are occurred due to lack of system resources, such as:

- Out of memory error (heap memory problem)
- Stack overflow error

Error is always an unchecked type of exception.

It is possible to handle exception but not possible to handle the errors.

Checked exception:

- The exception which are checked by the compiler
- It is the child class of Exception class

Unchecked exception:

- The exception which are not checked by compiler
- It is the child class of runtime exception

To check which exception belongs to which class in command prompt is,

**Javap java.lang. Exception name**
Example: javap java. lang. ArrayOutOfBoundException

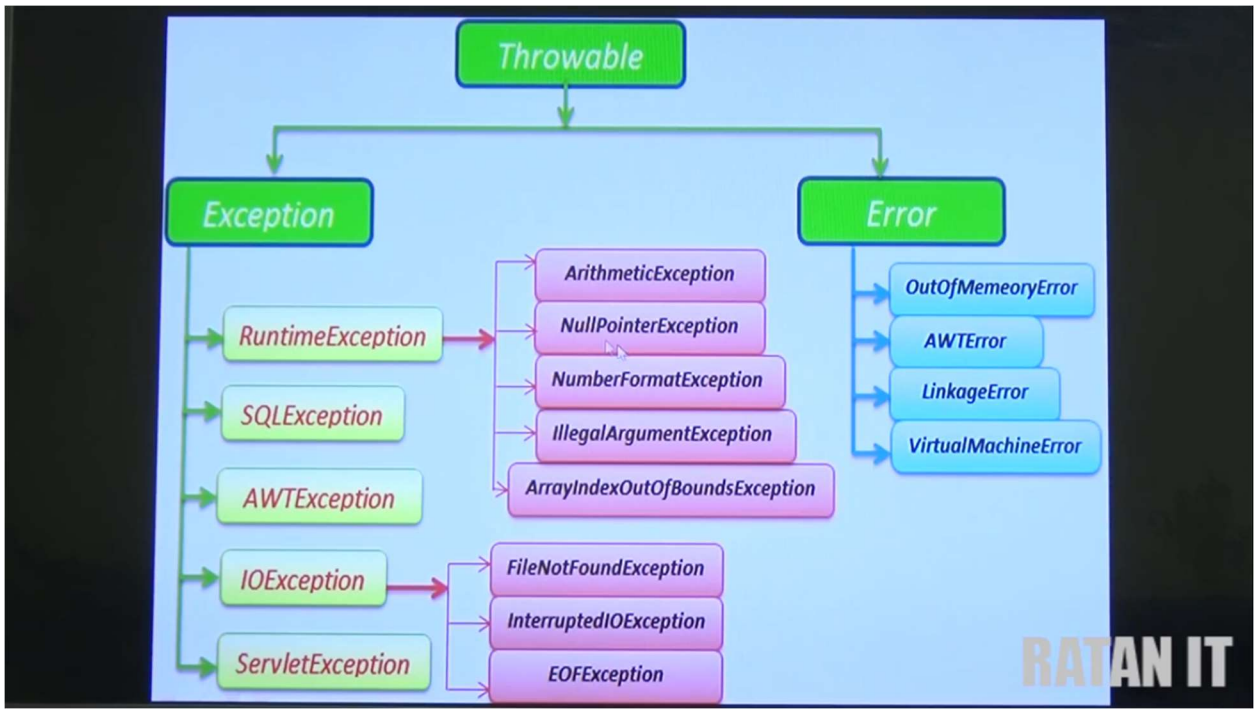
```
// Checked Exception : child class of exception class
import java.io.*;
class CheckedException
{
	public static void main (String args[])
	{
		System.out.println("ashirbad world...");
		Thread.sleep(1000); // InterruptedException

		FileInputStream fis = new FileInputStream("abc.txt"); //FileNotFoundException
		System.out.println("rest of the app");
	}
}
```

```
//Unchecked Exception : child class of runtime exception

class Test
{
	public static void main(String args[])
	{
		System.out.println("ashirbad world");
		System.out.println("10/0"); // ArithmeticException

		int a[] = {10,20,30};
		System.out.println(a[6]); //ArrayIndexOutOfBoundsException

		System.out.println("ashirbad".CharAt("12");

//java.lang.StringOutofBoundExceptionption

		Integer i=Integer.valueOf("ten"); //java.lang.NumberFormatException
		System.out.println("Rest of the app");
	}
}
```

```
//Error : occurs due to lack of system resources

class Error
{
	public static void main(String args[])
	{
		int[] a = new int[10000000]; //java.lang.OutOfMemoryError: java heap space
	}
}
```

## Exception Hierarchy

- The root class of exception is Throwable class.
- Throwable class is of two types
  - Exception: checked type
  - Error: unchecked type




The keywords most used in exception handling are,

- Try
- Catch
- Finally
- Throws
- Throw

Handle the exception by using try-catch block

```
try-catch
{
        exceptional code: may or may not
}
catch (Exception-name ref-var)
{
        code: executed when the exception raised in try
}
```

Example 1:

```
//Application without try-catch

class Test
{
        public static void main (String args[])
        {
                System.out.println("ashirbad world");
                System.out.println(10/0);
                System.out.println("rest of the app...");
        }
}

E:\>java Test
ashirbad world
Exception in thread "main" java.lang.ArithmeticException: /by zero

1. abnormal termination
2. rest of the app not executed

//Application with try-catch

class Test
{
        public static void main (String args[])
        {
                System.out.println("ashirbad world");
                try
                {
                        System.out.println(10/0);
                }
                catch (ArithmeticException ae)
                {
                        System.out.println(10/2);
                }
                System.out.println("rest of the app...");
```

```
		}
}

E:\>java Test
ashirbad world
5
rest of the app...

1. normal termination
2. rest of the app executed
```

Example 2:

```
//catch block not matched: program terminated abnormally

class Test
{
		public static void main (String args[])
		{
				try
				{
						System.out.println(10/0);
				}
				catch (NullPointerException ae)
				{
						System.out.println(10/2);
				}
				System.out.println("rest of the app...");
		}
}

E:\>java Test
Exception in thread "main" java.lang.ArithmeticException: /by zero
```

Example 3:

```
//if no exception in try block: catch blocks are not checked

class Test
{
		public static void main (String args[])
		{
				try
				{
						System.out.println("ashirbad");
				}
				catch (NullPointerException ae)
```

```
                {
                        System.out.println(10/2);
                }
                System.out.println("rest of the app...");
        }
}

E:\>java Test
ashirbad
rest of the app...
```

Example 4:

```
// invalid: only try blocks are not allowed
class Test
{
        public static void main (String args[])
        {
                try
                {
                        System.out.println("ashirbad");
                }
                System.out.println("rest of the app...");
        }
}
E:\>javac Test.java
error: 'try' without 'catch', 'finally' or resource declarations
```

Example 5:

```
// invalid: in between two blocks statement declarations are invalid
class Test
{
        public static void main (String args[])
        {
                try
                {
                        System.out.println("ashirbad");
                }
                System.out.println("ashu");
                catch (NullPointerException ae)
                {
                        System.out.println(10/2);
                }
                System.out.println("rest of the app...");
        }
}
```

```
E:\>javac Test.java
error: 'try' without 'catch', 'finally' or resource declarations
```

Example 6:

```
// exception raised other than try: abnormal termination
class Test
{
	public static void main (String args[])
	{
		try
		{
			System.out.println(10/0);
		}
		catch (ArithmeticException ae)
		{
			System.out.println(10/0);
		}
		System.out.println("rest of the app...");
	}
}
E:\>javac Test.java
Exception in thread "main" java.lang.ArithmeticException: /by zero
```

Example 7:

```
// case 1:

class Test
{
	public static void main (String args[])
	{
		try
		{
			System.out.println("ashirbad");
			System.out.println("ashu");
			System.out.println(10/0);
		}
		catch (ArithmeticException ae)
		{
			System.out.println(10/2);
		}
		System.out.println("rest of the app...");
	}
}

E:\>java Test
```

```
ashirbad
ashu
5
rest of the app...

// case 2: if exception raised in the try block once, then the rest of the statement will not be executed
class Test
{
	public static void main (String args[])
	{
		try
		{
			System.out.println(10/0);
			System.out.println("ashirbad");
			System.out.println("ashu");
		}
		catch (ArithmeticException ae)
		{
			System.out.println(10/2);
		}
		System.out.println("rest of the app...");
	}
}

E:\>java Test
5
rest of the app...
```

Category 1: try with multiple catch blocks

```
// case - 1:
import java.util.*;
class Test
{
	public static void main(String args[])
	{
		try
		{
			Scanner s = new Scanner(System.in);
			System.out.println("Enter a number:");
			int num = s.nextInt();
			System.out.println(10/num); //AE
			System.out.println("ratan".charAt(12)); //SIOBE
		}
		catch (ArithmeticException ae)
		{
			System.out.println("ratanit.com");
		}
```

```
                        catch (StringOutOfBoundException e)
                        {
                                System.out.println("durgasoft");
                        }
                        System.out.println("rest of the app...");
                }
}


E:\>java Test
Enter a number:
0
ratanit.com
rest of the app...

E:java Test
Enter a number:
2
5
durgasoft
rest of the app...
```

```
// case - 2: try with only one catch block
import java.util.*;
class Test
{
        public static void main(String args[])
        {
                try
                {
                        Scanner s = new Scanner(System.in);
                        System.out.println("Enter a number:");
                        int num = s.nextInt();
                        System.out.println(10/num); //AE
                        System.out.println("ratan".charAt(12)); //SIOBE
                }
                catch (Exception e)
                {
                        System.out.println("ratanit.com"+e);
                }
                System.out.println("rest of the app...");
        }
}
```

```
// case - 3: child to parent
import java.util.*;
```

```
class Test
{
	public static void main (String args[])
	{
		try
		{
			Scanner s = new Scanner(System.in);
			System.out.println("Enter a number:");
			int num = s.nextInt();
			System.out.println(10/num); //AE
			System.out.println("ratan".charAt(12)); //SIOBE
		}
		catch (ArithmeticException ae)
		{
			System.out.println("ratanit.com"+e);
		}
		catch (Exception e)
		{
			System.out.println("durgasoft");
		}
		System.out.println("rest of the app...");
	}
}

//case - 4: parent to child
		catch (Exception e)
		{
			System.out.println("ratanit.com");
		}
		catch (ArithmeticException ae)
		{
			System.out.println("durgasoft");
		}
Error: exception ArithmeticException has already been caught
```

Category 2: how to print exception message in different ways

Following are the different ways to handle exception messages in Java.

**Using printStackTrace () method** – It print the name of the exception, description and complete stack trace including the line where exception occurred.

```
catch (Exception e) {
  ae. printStackTrace ();
}
```

**Using toString () method** – It prints the name and description of the exception.

```
catch (Exception e) {
  System.out.println(e. toString ());
}
```

**Using getMessage () method** – Mostly used. It prints the description of the exception.

```
catch (Exception e) {
System.out.println(e. getMessage ());
}
```

```
// case - 1:
class Test
{
        void m3()
        {
                try
                {
                        System.out.println(10/0);
                }
                catch (ArithmeticException ae)
                {
                        System.out.println(ae.toString());
                        System.out.println(ae.getMessage());
                        ae.printStackTrace();
                }
        }
        void m2()
        {
                m3();
        }
        void m1()
        {
                m2();
        }
        public static void main (String args[])
        {
                Test t = new Test();
                t.m1();
        }
}

E:> java Test
java.lang.ArithmeticException: //by zero   //toString()
/by zero                    //getMessage()
java.lang.ArithmeticException: //by zero   //printStackTrace()
                        at Test.m3(Test.java:4)
```

```
                    at Test.m2(Test.java:13)
                    at Test.m1(Test.java:16)
                    at Test.main(Test.java:20)
```

```
// case - 2: JVM internally uses printStackTrace() method to print exception message
class Test
{
        void m3()
        {
                System.out.println(10/0);
        }
        void m2()
        {
                m3();
        }
        void m1()
        {
                m2();
        }
        public static void main(String args[])
        {
                Test t = new Test();
                t.m1();
        }
}
E:> java Test
Exception in thread "main" java.lang.ArithmeticException: /by zero
                    at Test.m3(Test.java:3)
                    at Test.m2(Test.java:6)
                    at Test.m1(Test.java:9)
                    at Test.main(Test.java:13)
```

```
// case - 3: exception propagation : only unchecked exceptions are automatically propagated but not
checked
class Test
{
        void m3()
        {
                System.out.println(10/0);
        }
        void m2()
        {
                m3();
        }
        void m1()
        {
```

```
                    try{m2();}
                    catch(ArithmeticException ae){System.out.println(10/2);}
        }
        public static void main(String args[])
        {
                    Test t = new Test();
                    t.m1();
        }
}
E:> java Test
5
```

Category – 3: Pipe symbol (|)

- When we declare the unchecked exception by using pipe symbol, those exceptions no need to present in the try block.
- When we declare the checked exception by using pipe symbol, those exception must be in try block.
- When we declare two checked exception goes to their try block there is no issue.
- We can use both checked and unchecked exception using pipe but checked exception must be present in try block.
- By using pipe symbol, we cannot mix both exceptions i.e., the we can't mix the child class and parent class exception.

```
// case - 1: unchecked exception

import java.util.*;
class Test
{
        public static void main(String args[])
        {
                    try
                    {
                                Scanner s = new Scanner(System.in);
                                System.out.println("Enter a num:");
                                int num = s.nextInt();
                                System.out.println(10/num);
                                System.out.println("ratan".charAt(12));
                    }
                    Catch (ArithmeticException| NumberFormatException e)
                    {
                                System.out.println("ratanit..."+e);
                    }
                    Catch    (StringIndexOutOfBoundsException|ClassCastException|NullPointerException
a)
                    {
                                System.out.println("durgasoft..."+a);
```

```
                }
                System.out.println("rest of the app");
        }
}
E:\>java Test
Enter a num:
0
ratanit...java.lang.ArithmeticException/by zero
rest of the app

E:\>java Test
Enter a num:
2
5
durgasoft...java.lang.StringIndexOutofBoundsException
rest of the app
```

```
// case - 2: checked exception : interrupted & FileNotFound //not ok
import java.io.*;
class Test
{
        public static void main(String args[])
        {
                try
                {
                        Thread.sleep(1000);
                }
                catch (InterruptedException| FileNotFoundException e)
                {
                        System.out.println("ratanit..."+e);
                }
                System.out.println("rest of the app");
        }
}

Error: exception FileNotFoundException is never thrown in body of corresponding try statement
```

```
// case - 3: checked exception : Interrupted & FileNotFound //ok
import java.io.*;
class Test
{
        public static void main(String args[])
        {
                try
                {
                        Thread.sleep(1000);
```

```
                                FileInputStream fis = new FileInputStream("abc.txt");
                        }
                        catch(InterruptedException| FileNotFoundException e)
                        {
                                System.out.println("ratanit..."+e);
                        }
                        System.out.println("rest of the app");
                }
}

Exception: FileNotFoundException
```

```
// case - 4: we can mix both checked & unchecked exception using pipe but checked exception must present in try block
import java.io.*;
class Test
{
        public static void main(String args[])
        {
                try
                {
                        Thread.sleep(1000);
                }
                catch(InterruptedException| ArithmeticException e)
                {
                        System.out.println("ratanit..."+e);
                }
                System.out.println("rest of the app");
        }
}

E:\> java Test
rest of the app
```

```
// case - 5:
import java.io.*;
class Test
{
        public static void main(String args[])
        {
                try
                {
                        FileInputStream fis = new FileInputStream("abc.txt");
                }
                catch(IOException| FileNotFoundException e)
                {
```

```
                                System.out.println("ratanit..."+e);
                        }
                        System.out.println("rest of the app");
                }
}

E:\> javac Test.java
error: Alternatives in a multi-catch statement cannot be related by sub classing
Alternative FileNotFoundException is a subclass of alternative IOException
```

Category – 4: try with resources

- Some of the benefits of using try with resources in java are,
  - More readable code and easy to write
  - No need of finally block just to close the resources.
  - The try-with-resources statement ensures that each resource is closed at the end of the statement.
- When we declare the resources with try block, once the try block is completed then the resource is automatically released.

```
// case - 1:
import java.util.*;
class Test
{
        public static void main(String args[])
        {
                try(Scanner s = new Scanner(System.in))
                {
                        System.out.println("Enter a num:");
                        int num = s.nextInt();
                        System.out.println("Entered value="+num);
                }
        }
}
Scanner---> closable--->ava.lang.AutoClosable

try-catch : valid
try with resources : valid
try-finally : valid
```

```
// case - 2:try with checked exception then catch is mandatory but with unchecked exception is optional.
import java.io.*;
class Test
{
        public static void main(String args[])
        {
                try(FileInputStream fis = new FileInputStream("abc.txt"))
```

```
                {
                        System.out.println("hii sir");
                }
                catch(IOException e)
                {
                        e.printStackTrace();
                }
        }
}
```

```
//case - 3: by using try block multiple resources can be declared with a semicolon(;)
import java.io.*;
import java.util.*;
class Test
{
        public static void main(String args[])
        {
                try(FileInputStream fis = new FileInputStream("abc.txt");
                        Scanner s = new Scanner(System.in);)
                {
                        System.out.println("hy sir");
                }
                catch(IOException e)
                {
                        e.printStackTrace();
                }
        }
}
```

Finally block

- The try is executed or not executed, the catch block is matched or not matched and if it has the normal termination or abnormal termination, the finally block is always executed.
- To release the resources or closing the operations, we use the finally block.
- Finally, block is executed irrespective of try and catch block.

Example:

```
//case - 1: Normal Termination
class Test
{
        public static void main(String args[])
        {
                try
                {
                        System.out.println("try block");
                }
```

```
                    catch(ArithmeticException ae)
                    {
                            System.out.println("catch");
                    }
                    finally
                    {
                            System.out.println("finally block");
                    }
            }
}
output:
try block
finally block
```

```
//case - 2: Normal Termination
class Test
{
            public static void main(String args[])
            {
                    try
                    {
                            System.out.println(10/0);
                    }
                    catch(ArithmeticException ae)
                    {
                            System.out.println("catch");
                    }
                    finally
                    {
                            System.out.println("finally block");
                    }
            }
}
output:
catch
finally block
```

```
//case - 3: Abnormal Termination
class Test
{
            public static void main(String args[])
            {
                    try
                    {
                            System.out.println(10/0);
                    }
```

```
                    catch(NullPointerException ae)
                    {
                            System.out.println("catch");
                    }
                    finally
                    {
                            System.out.println("finally block");
                    }
            }
}
E:\>java Test
finally block
Exception in thread "main" java.lang.ArithmeticException: /by zero
```

```
//case - 4: Abnormal Termination
class Test
{
            public static void main(String args[])
            {
                    try
                    {
                            System.out.println(10/0);
                    }
                    catch(NullPointerException ae)
                    {
                            System.out.println(10/0);
                    }
                    finally
                    {
                            System.out.println("finally block");
                    }
            }
}
E:\>java Test
finally block
Exception in thread "main" java.lang.ArithmeticException: /by zero
```

```
//case - 5: Abnormal Termination
class Test
{
            public static void main(String args[])
            {
                    try
                    {
                            System.out.println(10/0);
                    }
```

```
                    catch(NullPointerException ae)
                    {
                            System.out.println("catch");
                    }
                    finally
                    {
                            System.out.println(10/0);
                    }
            }
}
E:\>java Test
catch
Exception in thread "main" java.lang.ArithmeticException: /by zero
```

```
//case - 6: Normal Termination
class Test
{
            public static void main(String args[])
            {
                    try
                    {
                            System.out.println("try");
                    }
                    finally
                    {
                            System.out.println("finally");
                    }
            }
}
try
finally
```

```
//case - 7: once the control entered in try block, then only finally block is executed otherwise not
class Test
{
            public static void main(String args[])
            {
                    System.out.println(10/0);
                    try
                    {
                            System.out.println("try");
                    }
                    finally
                    {
                            System.out.println("finally");
                    }
```

```
		}
}
// here the try block will not executed
E:\>java Test
Exception in thread "main" java.lang.ArithmeticException: /by zero
```

```
//case - 8: only try block will be executed if we use System.exit(0);
class Test
{
		public static void main(String args[])
		{
				try
				{
						System.out.println("try");
						System.exit(0);
				}
				finally
				{
						System.out.println("finally");
				}
		}
}
E:\>java Test
try
```

```
//case-9: JVM is able to print only one exception, that to which exception is raised most recently
class Test
{
		public static void main(String args[])
		{
				try
				{
						System.out.println(10/0); //AE
				}
				catch(ArithmeticException ae)
				{
						System.out.println("ratan".charAt(10)); //SIOBE
				}
				finally
				{
						int[] a = {10,20,30};
						System.out.println(a[7]);
				}
		}
}
```

```
E:\>java Test
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
```

Nested try-catch blocks

```
//try-catch possibilities
1.
try
{
}
catch ()
{
}

2.
try
{
}
catch ()
{
}
//code snippet
try
{
}
catch ()
{
}

3.
try
{
}
catch ()
{
}
catch ()
{
}

4.
try
{
        try
        {
        }
        catch ()
        {
```

```
        }
}
catch ()
{
}

5.
try
{
}
catch ()
{
        try
        {
        }
        catch ()
        {
        }
}

6.
try
{
        try
        {
        }
        catch ()
        {
        }
}
catch ()
{
        try
        {
        }
        catch
        {
        }
}
```

## Throws keyword

- The throws keyword used to delegate the exception whereas tr-catch is used to write the exception code.
- Example: suppose exception raised at Kid or child but he is unable to handle that exception then he delegates the exception to his father through throws keyword.
- Exception handling using throws keyword:

```
//Example - 1: here principal used to handle the exception using try-catch
class Test
{
	void sdetails() throws InterruptedException
	{
		System.out.println("Anu is sleeping");
		Thread.sleep(1000); //IE
		System.out.println("Anu is sleeping");
	}
	void hod() throws InterruptedException
	{
		sdetails();
	}
	void principal()
	{
		try
		{
		hod();
		}
		catch(InterruptedException ie)
		{
			System.out.println("Exception...");
		}
	}
	void officeBoy()
	{
		principal();
	}
	public static void main(String args[])
	{
		Test t = new Test();
		t.officeBoy();
	}
}
output:
Anu is sleeping
Anu is sleeping
```

```
//Example - 2: if no one is able to handle the exception, it is handled by JVM
class Test
{
	void sdetails() throws InterruptedException
	{
		System.out.println("Anu is sleeping");
		Thread.sleep(1000); //IE
		System.out.println("Anu is sleeping");
	}
	void hod() throws InterruptedException
	{
		sdetails();
	}
	void principal() throws InterruptedException
	{
		hod();
	}
	void officeBoy() throws InterruptedException
	{
		principal();
	}
	public static void main(String args[]) throws InterruptedException
	{
		Test t = new Test();
		t.officeBoy();
	}
}
output:
Anu is sleeping
Anu is sleeping
```

```
//Example - 3: throws keyword can handle more exception at a time
import java.io.*;
class Test
{
	void m2() throws InterruptedException, FileNotFoundException
	{
		Thread.sleep(1000);
		FileNotFoundException fis = new FileInputStream("abc.txt");
	}
	void m1()
	{
		try
		{m2();}
		catch(FileNotFoundException| InterruptedException ie)
		{ie.printStackTrace();}
	}
```

```
	public static void main(String args[])
	{
		Test t = new Test();
		t.m1();
	}
}
//Example - 4: instead of writing all exception, we can write only Exception
import java.io.*;
class Test
{
	void m2() throws Exception
	{
		Thread.sleep(1000); //IE
		FileNotFoundException fis = new FileInputStream("abc.txt"); //FNE
	}
	void m1()
	{
		try
		{
			m2();
		}
		catch(Exception ie)
		{
			ie.printStackTrace();
		}
	}
	public static void main(String args[])
	{
		Test t = new Test();
		t.m1();
	}
}
//Example - 5:
import java.io.*;
class Test
{
	void m2() throws InterruptedException, FileNotFoundException
	{
		Thread.sleep(1000); //IE
		FileInputStream fis = new FileInputStream("abc.txt"); //FNE
	}
	void m1() throws InterruptedException
	{
		try
		{m2();}
		catch (FileNotFoundException ie)
		{
			ie.printStackTrace();
		}
```

```
	}
	public static void main(String args[])
	{
		Test t = new Test();
		try
		{t.m1();}
		catch (InterruptedException e)
		{e.printStackTrace();}
	}
}
```

**Note**:

- Unchecked exceptions are automatically propagated.
- Checked exceptions are propagated by using throws keyword.

```
//Example - 6:
//here case-1 & case-2 both codes are same
//case 1:
class Test
{
	void m1()
	{
		System.out.println(10/0);
	}
	public static void main(String args[])
	{
		Test t = new Test();
		try
		{t.m1();}
		catch(ArithmeticException a)
		{System.out.println(10/5);}
	}
}
```

```
//case 2: here throws keyword not required because unchecked exceptions are automatically propagated
class Test
{
	void m1() throws ArithmeticException
	{
		System.out.println(10/0);
	}
	public static void main(String args[])
	{
		Test t = new Test();
		try
		{t.m1();}
```

```
                    catch(ArithmeticException a)
                    {System.out.println(10/5);}
        }
}
```

```
output: 2
```

Throw keyword

JVM is responsible for handling the pre-defined exception by creating its object but the user-defined exceptions are not handled by the JVM directly. By the help of throw keyword, it first creates an object then pass the exception to JVM.

Two types of user exception:

- User checked exceptions (class MyException extends Exception {})
  - Default constructor approach
  - Parametrized constructor approach
- User unchecked exceptions (class MyException extends RuntimeException {})
  - Default constructor approach
  - Parameterized constructor approach

Example:

```
//step 1: create the exception (checked)
//InvalidAgeException.java
class InvalidAgeException
{
        //Default constructor: 0-arg constructor
}
//step 2: use the exception in your project
//Test.java
import java.util.*;
class Test
{
        static void status(int age) throws InvalidAgeException
        {
                if(age>20)
                {
                        System.out.println("eligible for mrg");
                }
                else
                {
                        throw new InvalidAgeException();
                }
        }
        public static void main(String args[])
```

```
	{
		Scanner s = new Scanner(System.in);
		System.out.println("enter your age:");
		int age = s.nextInt();
		Test.status(age);
	}
}
E:/>java Test
enter your age:
23
eligible for mrg

E:/>java Test
enter your age:
12
Exception in thread "main" InvalidAgeException
 //InvalidAgrException.java(checked exception)
class InvalidAgeExecption extends Exception
{
	InvalidAgeExecption(String str)
	{
		//parameterized constructor
		//calling super class cons by passing our info
		super(str);
	}
}
//Test.java
import java.util.*;
class Test
{
	static void status(int age) throws InvalidAgeExecption
	{
		if(age>20)
		{
			System.out.println("eligible for mrg");
		}
		else
		{
			throw new InvalidAgeExecption("u r not eligible for mrg");
		}
	}
	public static void main(String args[]) throws InvalidAgeExecption
	{
		Scanner s = new Scanner(System.in);
		System.out.println("enter your age:");
		int age = s.nextInt();
		Test.status(age);
	}
}
```

```
E:\> java Test
enter your age:
12
Exception in thread "main" InvalidException: u r not eligible for mrg
```

**Note:**

**User checked exception**: must handle by try-catch or throw keyword

**User unchecked exception**: try-catch or throws keyword optional but recommended.

In default constructor, only exception is printed on the console as output but in case of Parameterized constructor, exception is printed along with some description.

| Checked Exception | Description |
|---|---|
| ClassNotFoundException | If the loaded class is not available |
| CloneNotSupportedException | Attempt to clone an object that does not implement the Cloneable interface. |
| IllegalAccessException | Access to a class is denied. |
| InstantiationException | Attempt to create an object of an abstract class or interface. |
| InterruptedException | One thread has been interrupted by another thread. |
| NoSuchFieldException | A requested field does not exist. |
| NoSuchMethodException | If the requested method is not available. |
| ArithmeticException | Arithmetic error, such as divide-by-zero. |
| ArrayIndexOutOfBoundsException | Array index is out-of-bounds. (Out of range) |
| InputMismatchException | If we are giving input is not matched for storing input. |
| ClassCastException | If the conversion is Invalid. |
| IllegalArgumentException | Illegal argument used to invoke a method. |
| IllegalThreadStateException | Requested operation not compatible with current thread state. |
| IndexOutOfBoundsException | Some type of index is out-of-bounds. |

| | |
|---|---|
| NegativeArraySizeException | Array created with a negative size. |
| NullPointerException | Invalid use of a null reference. |
| NumberFormatException | Invalid conversion of a string to a numeric format. |
| StringIndexOutOfBoundsException | Attempt to index outside the bounds of a string. |

## String Manipulation

1. Java.lang.String
2. Java.lang.StringBuffer
3. Java.lang.StringBuilder
4. Java.util.StringTokenizer

### Java.lang.String

- ✓ String is used to represent group of characters or character array enclosed within the double quotes.
- ✓ **String vs StringBuffer:** String & StringBuffer, both classes are final classes present in java.lang package.
- ✓ String vs StringBuffer: we are able to create string object in two ways,
  - Without using new operator
    **String str = "ashirbad";**
  - By using new operator
    **String str = new String ("ashirbad");**
- ✓ We are able to create **StringBuffer** object only one approach by using new operator,
  **StringBuffer sb = new StringBuffer ("ashirbad");**

### == operator

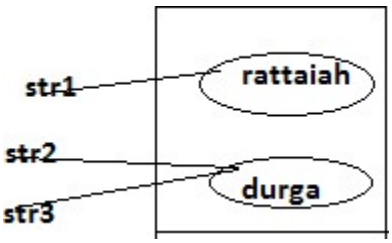
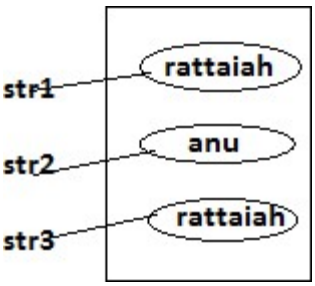
- ✓ It is comparing **reference** type and it returns Boolean value as a return value.
- ✓ If two reference variables are pointing to same object then it returns true otherwise false.

### Immutability vs mutability

- ✓ String is **immutability** class; it means once we are creating string objects it is not possible to perform modifications on existing object. (because the reference of string can't be change)
- ✓ StringBuffer is a **mutability** class, it means once we are creating StringBuffer objects on that existing object it is possible to perform modifications.
- ✓ We can **achieve** immutability nature of the class by using following properties,
  - Declare the fields as final & private
  - Declare the class with final
  - Set the data by using parameterized constructor

### Concat ()

- ✓ Concat () method is combining two string objects and it is returning new string object.
  **Public java.lang.String concat (java.lang.String);**

| Creating a string object without using new operator | Creating a string object by using new operator |
|---|---|
| When we create a string object without using new operator the objects are created in **SCP** (String constant pool) area. | Whenever we are creating object by using new operator the object created in **heap** area. |
| String str1 = "rattaiah";<br>String str2 = "Sravya";<br>String str3 = "Sravya";<br>  <br><br>**SCP area** | String str1 = new String ("rattaiah");<br>String str2 = new String ("anu");<br>String str3 = new String ("rattaiah");<br>  <br><br>**Heap area** |
| When we are creating object without using new operator then just before object creation it is always checking previous objects.<br>✓ If the previous object is available with the same content then it won't create new object that reference variable pointing to existing object.<br>✓ If the previous objects are not available then JVM will create new object. | Whenever we create object in Heap area instead of checking previous objects it directly creates objects. |
| SCP area does not allow duplicate objects. | Heap memory allows duplicate objects. |

## equals () method

equals () method present in object used for **content** comparison & returns Boolean value.

String is child class of object and it is overriding equals () methods used for content comparison. If two objects content is same then returns true otherwise false.

## compareTo () & compareToIgnoreCase ()

By using compareTo (), we are comparing two strings **character by character**, such type of checking is called **lexicographically** checking or **dictionary** checking.

compareTo (), return type is integer and it returns three values,

- if the two strings are equal then it returns **zero**.
- If the first string first character Unicode value is bigger than second string first character Unicode value then it returns **+ve** value.
- If the first string first character Unicode value is smaller than second string first character Unicode value then it returns **-ve** value.

**compareTo ()** method comparing two strings with case sensitive.

**compareToIgnoreCase ()** method comparing two strings character by character by ignoring case.

## Length () method & length variable

- ✓ **length** variable used to find length of the array.
  **int [] a = {10, 20, 30};**
  **System.out.println (a.length); //3**
- ✓ length () is method used to final length of the string.
  **String str = "ashirbad";**
  **System.out.println (str.length ());**

## charAt (int) & split () & trim ()

- ✓ **charAt (int):** by using above method, we are able to extract the character from particular index position.

  **Public char chatAt (int);**

- ✓ **Split (String):** by using split () method we are dividing string into number of tokens.

  **Public java.lang.String [] split (java.lang.String);**

- ✓ **trim ():** trim () is used to remove the trail and leading spaces, this method always used for memory saver.

  **Public java.lang.String trim ();**

## Replace () & toUpperCase () & toLowerCase ()

- ✓ replace () method used to replace the String or character.
  **Public java.lang.String replace (String str, String str);**
  **Public java.lang.String replace (char, char);**
- ✓ The below methods are used to convert lower case to upper case & upper case to lower case.
  **Public java.lang.String toLowerCase ();**
  **Public java.lang.String toUpperCase ();**

## endsWith () & startsWith () & substring ()

- ✓ **endsWith ()** is used to find out if the string is ending with particular character/string or not.
- ✓ **startsWith ()** is used to find out the particular string starting with particular character/string or not.
  **Public boolean startsWith (java.lang.String);**
  **Public boolean endsWith (java.lang.String);**
- ✓ **substring ()** used to find substring in main string.
  Public java.lang.String substring (int); **int = starting index**
  Pubic java.lang.String substring (int, int); **int = starting index to int = ending index**
  While printing substring () it includes starting index & excludes ending index.

## StringBuffer class methods

## Reverse () & delete () & deleteChatAt ()

```
class Test
{
	public static void main(String[] args)
	{
		StringBuffer sb=new StringBuffer("rattaiah");
		System.out.println(sb);
		System.out.println(sb.delete(1,3));
		System.out.println(sb);
		System.out.println(sb.deleteCharAt(1));
		System.out.println(sb.reverse());
	}
}
```

## Append ()

```
class Test
{
	public static void main(String[] args)
	{
		StringBuffer sb=new StringBuffer("rattaiah");
		String str=" salary ";
		int a=60000;
		sb.append(str);
		sb.append(a);
		System.out.println(sb);
	}
};
```

## Insert ()

By using above method, we are able to insert the string in any location of the existing string.

```
class Test
{
	public static void main(String[] args)
	{
		StringBuffer sb=new StringBuffer("ratan");
		sb.insert(0,"hi ");
		System.out.println(sb);
	}
}
```

## replace ()

```
class Test
{
	public static void main(String[] args)
	{
		StringBuffer sb=new StringBuffer("hi ratan hi");
		sb.replace(0,2,"oy");
		System.out.println("after replaceing the string:-"+sb);
	}
}
```

## Java.lang. StringBuilder

- ✓ introduced in JDK 1.5 version.
- ✓ StringBuilder is identical to StringBuffer except for one important difference.
- ✓ Every method present in the StringBuilder is not synchronized means that is not thread safe.
- ✓ Multiple threads are allowed to operate on StringBuilder methods; hence the performance of the application is increased.

## StringTokenizer

The java.util.StringTokenizer class allows an application to break a string into tokens.

```
import java.util.*; class Test
{
	public static void main(String[] args)
	{
		String str="hi ratan w r u wt bout anushka";
		StringTokenizer st = new StringTokenizer(str);//split the string with by default (space
symbol)
		while (st.hasMoreElements())
		{
			System.out.println(st.nextElement());
		}

		//used our string to split giver String
		String str1 = "hi,rata,mf,sdfsdf,ara";
		StringTokenizer st1 = new StringTokenizer("hi,rata,mf,sdfsdf,ara",",");
		while (st1.hasMoreElements())
		{
			System.out.println(st1.nextElement());
		}
	}
}
```

## Java Multithreading

### Thread

- ✓ A thread is a lightweight subprocess, the smallest unit of processing.
- ✓ It is a separate path of execution.
- ✓ The independent execution technical term is called thread.
- ✓ Whenever different parts of the program executed simultaneously that each and every part is called thread.
- ✓ The thread is light weight process because whenever we are creating thread it is not occupying the separate memory, it uses the same memory. Whenever the memory is shared means it is not consuming more memory
- ✓ If exception occurs in one thread, it doesn't affect other thread.
- ✓ It uses a shared memory area.

### Why do we use threads in java?

- ✓ The threads work independently and provides the maximum **utilization** of the CPU, thus enhancing the CPU performance. Threads to make java application faster by doing multiple things at same time. In technical terms, thread helps you to achieve parallelism in Java program.
- ✓ Threads use a shared memory area. They don't allocate separate memory area so saves memory, and context-switching between the threads takes less time than process.

NOTE: If the application contains more than one thread, then thread execution decided by thread scheduler otherwise decided by JVM. Thread scheduler is a component of JVM.

### Information about main Thread

When java program started, one thread is running immediately that thread is called main thread of your program.

- ✓ It is used to create a new Thread (child class).
- ✓ It must be the last thread to finish the execution because it performs various actions.

It is possible to get the current thread reference by using **currentThread ()** method, it is a static public method present in Thread class.

```
class CurrentThreadDemo
{
        public static void main (String [] args)
        {
                Thread t=Thread.currentThread();
                System.out.println("current Thread--->"+t);
                //change the name of the thread
                t.setName("ratan");
                System.out.println("after name changed---> "+t);
        }
};
```

## Application areas of Multithreading

- ✓ Developing video games
- ✓ Implementing multimedia graphics
- ✓ Developing animations

## Multithreading:

- ✓ Multithreading is a process of executing multiple threads simultaneously.
- ✓ Java multithreading is mostly used in video game, animation etc.
- ✓ Realtime example:
- ✓ **Online ticket booking**: here many users trying to book available ticket at same time (ex: tatkal booking), here application needs to handle different threads (different users request to server), if tickets sold out/not available then rest users will get correct response as not available to book.

## Creating a Thread

Two ways to approach/create thread,

- By extending thread class
- By implementing runnable interface (this is the best approach for creating thread because java doesn't support multiple inheritance).

### 1. By extending Thread class

**Step – 1**: our normal java class will become Thread class whenever we are extending predefined Thread class.

```
Class MyThread extends Thread
{
};
```

**Step – 2**: override the run () method to write the business logic of the Thread (run () method present in Thread class).

```
class MyThread extends Thread
{
        public void run ()
        {
        System.out.println("business logic of the thread");
        System.out.println("body of the thread");
        }
}
```

**Step – 3**: create user defined Thread class object

```
MyThread t = new MyThread ();
```

**Step – 4**: start the Thread by using start () method

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

**Example**

```
//Java thread example by extending thread class
class Multi extends Thread //defining a Thread
{
        //business logic of user defined Thread
        public void run ()
        {
                System.out.println("thread is running...");
        }
        public static void main(String args[])
        {
                Multi t1 = new Multi();
                t1.start();
        }
}
output: thread is running...
```

Flow of Execution

- ✓ Whenever we are calling **t.start ()** method, then JVM will search for start () method in the MyThread class since not available so JVM will execute parent class (Thread) start () method.
- ✓ Thread class start () method responsibilities,
  - User defined thread is registered into Thread Scheduler then only decide new Thread is created.
  - The Thread class start () automatically calls run () to execute logics of user defined Thread.

Thread Scheduler

- ✓ Thread scheduler is a part of the JVM. It decides thread execution.
- ✓ Thread scheduler is a mental patient, we are unable to predict the exact behaviour of Thread scheduler it is JVM vendor dependent.
- ✓ Thread scheduler mainly uses two algorithms to decide thread execution.
  - **Preemptive algorithm**: in this, highest priority task is executed first after this task enters into waiting state or dead state, then only another higher priority task come to existence.
  - **Time slicing algorithm**: a task is executed predefined slice of time and then return pool of ready task. The scheduler determines which task is executed based on the priority and other factors.
- ✓ We can't except exact behaviour of the thread scheduler it is JVM vendor dependent. So, we can't say except output of the multithreaded examples we can say the possible outputs.
- ✓ It is not possible to start a thread **twice**.

## Life cycle of Thread

Life cycle stages of thread are,

1. New
2. Ready
3. Running state
4. Blocked/waiting/non-running mode
5. Dead state

**New**

Creating a new thread i.e.

MyThread t = new MyThread ();

**Ready**

Starting the thread i.e.

t. start ()

**running state**

- ✓ if the thread scheduler allocates CPU for particular thread. Thread goes to running state.
- ✓ The Thread is running state means the run () is executed.

**Blocked state**

If the running thread got interrupted of goes to the sleeping state at that moment it goes to the blocked state.

**Dead state**

If the business logic of the project is completed means run () over thread goes to dead state.

## 2. By implementing Runnable interface

**Step – 1**: our normal java class will become thread class whenever we are implementing runnable interface.

```
Class MyClass extends Runnable
{};
```

**Step – 2**: override the main method to write logic of Thread

```
class MyClass extends Runnable
{
        public void run()
        {
        System.out.println("Rattaiah from SravyaInfotech");
        System.out.println("body of the thread");
        }
}
```

**Step – 3**: creating an object

```
MyClass obj = new MyClass ();
```

**Step – 4**: creates a Thread class object

After new thread is created, it is not started running until we are calling start () method. So, whenever we are calling start method that start () method call run () method then the new Thread execution started.

```
Thread t = new Thread (obj);
t.start ();
```

**Example**

```
//Java thread example by implementing Runnable interface
class Multi implements Runnable
{
        public void run()
        {
                System.out.println("threading is running...");
        }
        public static void main(String args[])
        {
                Multi m1 = new Multi();
                Thread t1 = new Thread(m1);
                t1.start();
        }
}
output: thread is running...
```

NOTE:

- ✓ Important point is that when extending the **Thread** class, the sub class cannot extend any other base classes because Java allows only single inheritance.
- ✓ Implementing the **Runnable** interface does not give developers any control over the thread itself, as it simply defines the unit of work that will be executed in a thread.
- ✓ By implementing the **runnable** interface, the class can still extend other base class if required.
- ✓ If we are not overriding the run () method, thread class is executing which is having empty implementation so we are not getting any output.
- ✓ If we are overriding start () method then JVM executes override start () method at this situation we are not giving chance to the thread class start (), hence a new thread will be created, only one thread is available – the name of that thread is main method.

## Difference between t.start () & t.run ()

- ✓ In the case of t.start (), Thread class start () is executed a new thread will be created that is responsible for the execution of run () method.
- ✓ But in the case of t.run () method, no new thread will be created and the run () is executed like a normal method call by main thread.

## Java.lang.Thread.yield ()

- ✓ Yield () method causes to pause current executing Thread for giving the chance for waiting threads of same priority.
- ✓ If there are no waiting threads or all threads are having low priority then the same thread will continue its execution once again.
- ✓ Syntax:
  **Public static native void yield ();**

```
class MyThread extends Thread
{
        public void run()
        {
                for(int i=0;i<10;i++)
                {       Thread.yield();
                        System.out.println("child thread");}
        }
}
class ThreadYieldDemo
{
        public static void main(String[] args)
        {
                MyThread t1=new MyThread();
                t1.start();
                for(int i=0;i<10;i++)
                {System.out.println("main thread");}
        }}
```

## Java.lang.Thread.join () method

- ✓ Join () method allows one thread to wait for the completion of another thread.
  **t.join ():** here "t" is a Thread object whose thread is currently running.
- ✓ Join () is used to stop the execution of the thread until completion of some other Thread.
- ✓ If a t1 thread is executed t2.join () at that situation t1 must wait until completion of the t2 thread.

## Java.lang.Thread. Interrupted ()

- ✓ A thread can interrupt another sleeping or waiting thread. But one thread is able to interrupted only another sleeping or waiting thread.
- ✓ To interrupt, a thread uses thread class **interrupt** () method.
- ✓ The interrupt () is affected whenever our thread enters into waiting state or sleeping state and if our thread doesn't enters into the waiting/sleeping state, interrupted call will be wasted.

## Shutdown Hook

- ✓ Shutdown hook is used to perform cleanup activities when JVM shutdown normally or abnormally.
- ✓ Clean-up activities like,
  - Resource release
  - Database closing
  - Sending alert message
- ✓ So, if you want to execute some code before JVM shutdown use shutdown hook.
- ✓ The JVM will be shut down in following cases.
  - When you type **ctrl + C**
  - When we used **System.exit (int)**
  - When the system is shutdown…etc
- ✓ To add the shutdown hook to JVM use **addShutdownHook (obj)** method of runtime class.
  **Public void addShutdownHook (java.lang.Thread);**
- ✓ To remove the shutdown hook from JVM use **removeShutdownHook (obj)** method of runtime class.
  **Public boolean removeShutdownHook (java.lang.Thread);**
- ✓ To get the runtime class object use static factory method **getRuntime** () & this method present in runtime class.
  **Runtime r = Runtime.getRuntime ();**

## Factory method

One java class method is able to return same class object or different class object is called factory method.

## Synchronized

- ✓ Synchronized modifier is the modifier applicable for methods but not for classes and variables.
- ✓ If a method or a block declared as synchronized then at a time only one Thread is allowed to operate on the given object.
- ✓ The main advantage of synchronized modifier is we can resolve data **inconsistency** problems.
- ✓ But the main disadvantage of synchronized modifier is it increase the waiting time of the Thread and effects performance of the system. Hence if there is no specific requirement it is never recommended to use.
- ✓ The main purpose of this modifier is to reduce the data inconsistence problem.

## Non-synchronized method

In non-synchronized case, multiple methods are accessing the same methods hence we are getting data inconsistency problems. These methods are not **thread safe**.

But in this case multiple threads are executing so the performance of the application will be increased.

Hence it is not recommended to use the synchronized modifier in the multithreading programming.

## Synchronized blocks

If the application method contains 100 lines but if we want to synchronized only 10 lines of code use synchronized blocks.

The synchronized block contains less scope compare to method.

If we are writing all the method code inside the synchronized block it will work same as the synchronized method.

## Daemon thread

The threads which are executed at background is called daemon thread.

**Ex**: - garbage collector, Thread scheduler, default exception handler

## Non – Daemon thread

- ✓ The threads which are executed in fore ground is called non-daemon threads.
- ✓ **Ex**: - normal java application
- ✓ When we create a thread in java that is user defined thread if it is running JVM will not terminate that process.
- ✓ If a thread is marked as a daemon thread, JVM does not wait to finish and as soon as all the user defined threads are finished then it terminates the program and all associated daemon threads.
- ✓ Set the daemon nature to thread by using **setDaemon ()** method
  **MyThread t = new MyThread ();**
  **t.setDaemon (true);**
- ✓ To know whether a thread is daemon or not use **isDaemon ()** method
  **Thread.currentThread ().isDaemon();**

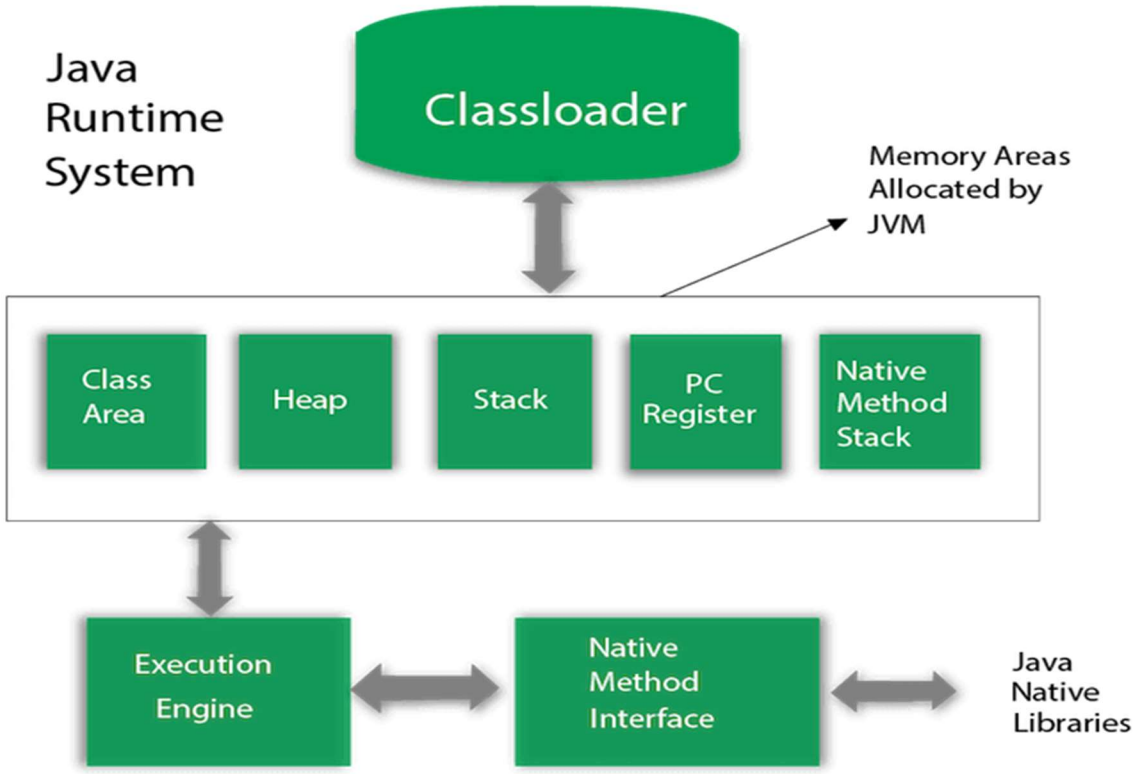
## Volatile modifier

Volatile modifier is also applicable only for variables but not for methods and classes.

If the values of a variable keep on changing such type of variables we have to declare with volatile modifier.

If a variable declared as volatile then for every thread a separate local copy will be created.

## JVM Architecture




### Class loader

- ✓ Class loader is a subsystem of JVM which used to load class files. Whenever we run the java program, it is loaded first by class loader.
- ✓ It is used to load classes & interfaces.
- ✓ It verifies the byte code instructions.
- ✓ It allots the memory required for the program.

### Class (Method) area

- ✓ Class area stores pre - class structure such as the runtime constant pool, field and method data, the code for methods.
- ✓ It is used to store the class data and the method data.

### Heap

- ✓ It is the runtime data area in which objects are allocated.
- ✓ It is used to store the objects.

## Stack

- ✓ Java stack stores frames.
- ✓ It holds local variables and partial variable results, and plays a part in method invocation and return.
- ✓ Whenever new thread is created for each and every new thread the JVM will creates PC (program counter) register and stack.
- ✓ If a thread executing java method the value of PC register indicates the next instruction to execute.
- ✓ Stack will store method invocations of every thread. The java method invocation includes local variables and return values and intermediate calculations.
- ✓ The each and every entry will be stored in stack. And the stack contains group of entries and each and every entry stored in one stack frame hence stack is group of stack frames.
- ✓ Whenever the method completed the entry is automatically deleted from the stack so whatever the functionalities declared in method it is applicable only for respective methods.

## Program counter register

Program register counter contains the address of the java virtual machine instruction currently being executed.

## Native method stack

It contains all the native methods used in the application.

## Execution Engine

It contains:

- ✓ A virtual processor
- ✓ Interpreter: read bytecode stream then execute the instruction
- ✓ Just-In-Time (JIT) compiler: it is used to improve the performance.

## Java Native Interface:

Java native interface (JNI) is a framework which provides an interface to communicate with another application written in another language like C, C++ etc.

## Difference between JDK, JRE and JVM

JDK (Java Development Kit) is for development (for application development) purpose whereas JRE (Java Runtime Environment) is for running the java programs.

JDK and JRE both contains JVM so that we can run our java program.

JVM is the heart of Java programming language and provides platform independence.

### JRE

- ✓ The Java Runtime Environment is a set of software tools which are used for developing Java applications.
- ✓ It is used to provide the runtime environment.
- ✓ It physically exists.
- ✓ It contains a set of libraries and other files that JVM uses at runtime.

### JDK

- ✓ The Java Development Kit is a software development environment which is used to develop Java application and applets.
- ✓ It physically exists.
- ✓ It contains JRE plus development tools.

### JVM

- ✓ It is virtual machine because it has no physical existence.
- ✓ It is a specification that provides a runtime environment in which Java bytecode can be executed.
- ✓ The JVM performs following operations:
    - Loads code
    - Verifies code
    - Executes code
    - Provides runtime environment

## Java API

### What is an API?

An application programming Interface (API) is a set of subroutine definitions, protocols, and tools for building application software.

An API make it easier for developers to use certain technologies in building applications by using certain predefined operations.

### API in Java

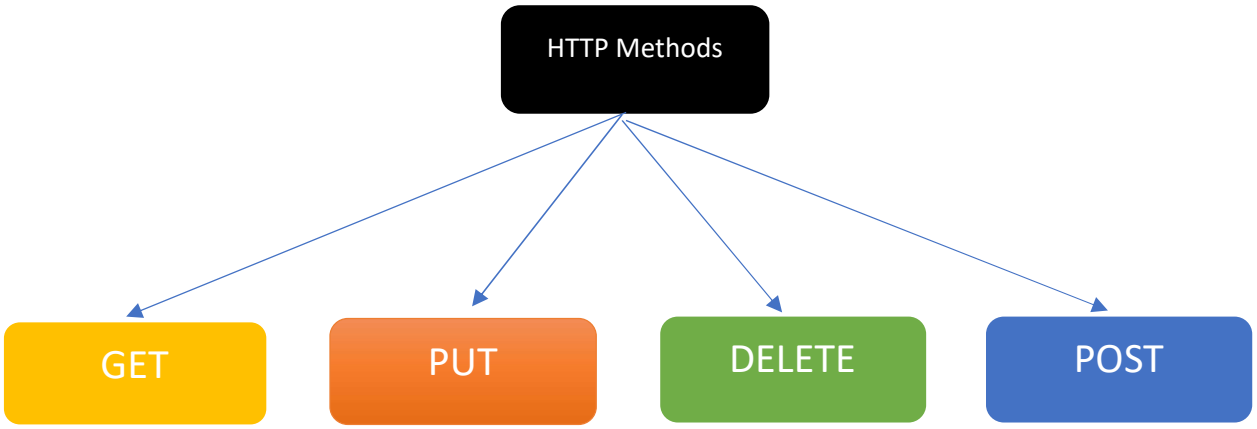
An API in Java, is a collection of prewritten packages, classes, and interfaces with their respective methods, fields and constructors.

In Java there are over 4500+ API available for developers.

### Rest API

Representational State Transfer or REST is a web standards-based architecture and uses HTTP Protocol for data communication.

- ✓ REST means create an object on the server side and return the values of an object.
- ✓ It is an architectural style as well as an approach for communications purpose that is often used in various web services development.
- ✓ It is often regarded as the "language of the internet".
- ✓ It is a stateless client and server model.
- ✓ The following HTTP methods are most commonly used in a REST based architecture:



Example

- ✓ Amazon.com released its API so that Website developers could more easily access Amazon's product information……An API is a software-to-software interface, not a user interface. With APIs, applications talk to each other without any user knowledge or invention.
- ✓ Book My Show application

Features of REST API

It is **stateless**

- ✓ One of the main features of a REST API is that its server is stateless, which means that every time we refer to it, it will be necessary to remind it our data, whether it is our user credentials or any other information.
- ✓ This factor is particularly relevant for any bank API.

It **supports** JSON and XML

- ✓ There are developers for all tastes and an API should aim to adapt to them all. Thus, another advantage of REST API is that it satisfies the expectations of those who use the JSON languages as much as it satisfies those that rely on XML.

It is **simple** than SOAP

- ✓ Beyond the REST architecture, developers are using the standard SOAP (Simple Object Access Protocol), another possibility when writing an API.
- ✓ The main advantage of the former over the latter is that its implementation is much simpler.
- ✓ A clear example can be seen in the API catalogue that Salesforce provides: it has tools with both architectures, but it notices that REST allows access to services that are "powerful, convenient and simpler to interact with Salesforce."

**Documentation**

- ✓ Each change in the architecture of the REST API should be reflected in its documentation so that any developer using it knows what to expect.
- ✓ The documentation does require the creators of the API to keep that information fully updated, which sometimes can be difficult to handle.

**Error messages**

- ✓ When making a mistake while working with an API, any developer will appreciate knowing that what the error has been. Hence, the possibility offered by REST architecture of including error message providing some clue in this regard is also relevant.
- ✓ Returning to Microsoft, the services offered by the company founded by Bill Gates through Azure – its tool for the cloud – have a clear list of possible error messages which, surely, must have been useful in more than one occasion.

## Principles of REST API or REST Architectural Constraints

REST defines **6** architectural constraints which make any web service – a RESTful API.

- Uniform Interface
- Client – Server
- Stateless
- Cacheable
- Layered System
- Code on Demand

### Uniform Interface

The uniform interface constraint is fundamental to the design of any RESTful system.

It simplifies and decouples the architecture, which enables each part to evolve independently.

The four constraints for this uniform interface are,

- Resource Identification in requests
- Resource Manipulation through representations
- Self – Descriptive messages
- Hypermedia as the engine of application state

### Client – Server

The principle behind the client-server constraints is the separation of concerns. Separating the user interface concerns from the data storage concerns improves the portability of the user interfaces across multiple platforms. It also improves scalability by simplifying the server components. Perhaps most significant to the Web is that the separation allows the components to evolve independently, thus supporting the Internet-scale requirement of multiple organizational domains.

### Stateless

The client-server communication is constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and the session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that can be used the next time the client chooses to initiate a new state-transition.

### Cacheable

As on the World Wide Web, clients and intermediaries can cache responses. Responses must, implicitly or explicitly, define themselves as either cacheable or non-cacheable to prevent clients from becoming stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client-server interactions, further improving scalability and performance.

**Layered System**

- ✓ A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way.
- ✓ If a proxy or load balancer is placed between the client and server, it won't affect their communications and there won't be a need to update the client or server code.
- ✓ Intermediary servers can improve system scalability by enabling load balancing and by providing shared caches. Also, security can be added as a layer on top of the web services, and then clearly separate business logic from security logic.
- ✓ Adding security as a separate layer enforces security policies. Finally, it also means that a server can call multiple servers to generate a response to the client.

**Code on Demand**

- ✓ Servers can temporarily extend or customize the functionality of a client by transferring executable code.
- ✓ For example: compiled components such as Java applets, or client-side scripts such as JavaScript.

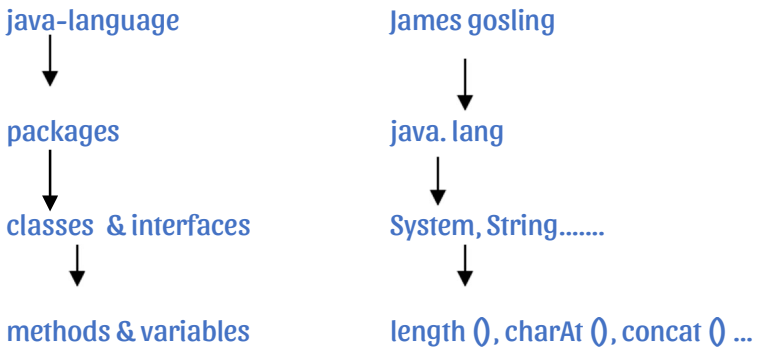
**Methods of REST API**

- ✓ **GET**: for fetching or read information from records
- ✓ **DELETE**: for removing the data from records
- ✓ **PUT**: for update the data in the records
- ✓ **POST**: to create a new entry
- ✓ **CRUD** methods (Create – Read – Update - Delete)

## Packages

### Java – Language:

In Java, James Gosling is maintained predefined support in the form of packages and these packages contains classes 7 interfaces, and these classes and interfaces contains predefined methods & variables.




### Java Source Code

Java contains 14 predefined packages but the default package in java is **java.lang**. and these are

- Java.lang
- Java.io
- Java.util
- Java.awt
- Java.beans
- Java.net
- Java.applet
- Java.times
- Java.text
- Java.nio
- Java.rmi
- Java.security
- Java.sql
- Java.math

Package is nothing but **physical** folder structure.

The default package in java is java.lang package.

### Types of Packages

There are two types of packages in java

- Predefined packages
- User defined packages

## Predefined packages

The predefined package is introduced by James Gosling and these packages contains predefined classes & interfaces and these class & interfaces contains predefined variables and methods.

Example – java. lang, java.io, java. util…etc

## User defined packages

- ✓ The packages which are defined by user, and these packages contains user defined classes and interfaces.
- ✓ Declare the package by using **package** keyword.
  Syntax: **package package – name;**
  Example: **package com.ashu;**
- ✓ Inside the source file it is possible to declare only one package statement and that statement must be first statement of the source file.
- ✓ Example:
  Package com.ashu;
  import java.io.*;
  import java.lang.*;

## Some predefined package and it's classes & interfaces

### Java.lang package

- ✓ The most commonly required classes and interfaces to write a sample program is encapsulated into a separate package is called java.lang. Package.
- ✓ The default package in the java programming is **java.lang** package.

| Lang package |
|---|
| String class |
| StringBuffer class |
| Object class |
| Runnable Interface |
| Cloneable Interface |

**Note**: In real time the project is divided into number of modules that each and every module is nothing but package statement.

## Java.io package

The classes which are used to perform the input output operations that are present in the java.io packages.

| io package |
|---|
| FileInputStream class |
| FileOutputStream class |
| FileReader class |
| FileWriter class |
| Serializable Interface |

## Java.net package

The classes which are required for connection establishment in the network that classes are present in the java.net package.

| net package |
|---|
| Socket class |
| ServerSocket class |
| URL class |
| SocketOption Interface |

### Advantages of Packages

- It improves **parallel development** of the project.
- Project **maintenance** will become easy.
- It improves **shareability** of the project.
- It improves **readability**.
- It improves **understandability**.

### Access Modifiers in Java

✓ The access modifiers in Java specifies the accessibility or scope of a field, method, constructor or class.
✓ There are four types of Java access modifier,
  - Private
  - Default
  - Protected
  - Public

✓ **Private**
  - The access level of a private modifier is only within the class.
  - It cannot be accessed from outside of the class.

✓ **Default**
  - The access level of default modifiers is only within the package.
  - It cannot be accessed from outside of the package.
  - If you do not specify any access level, it will be the default.

- ✓ **Protected**
  - The access level of a protected modifier is within the package and outside of the package through child class.
  - If you do not make the child class, it cannot be accessed from outside the package.
- ✓ **Public**
  - The access level of a public modifier is everywhere.
  - It can be accessed from within the class, outside the class, within the package and outside the package.

## Java.io Package

- ✓ Java.io package contains classes to perform input and output operations.
- ✓ By using java.io package we are performing **file handling** in java.

## Stream

- ✓ Stream is nothing but sequence of data, and it is called as stream means stream of water continuous flow.
- ✓ There are three streams are automatically created
  - **System.in** ----- standard input stream
  - **System.out** ----- standard output stream
  - **System.error** ----- standard error stream

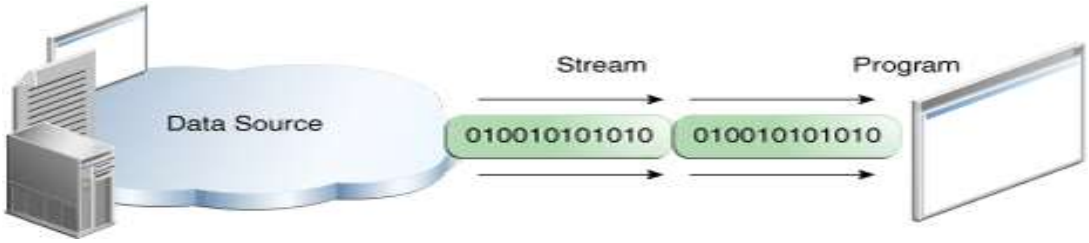
## I/O streams

- ✓ Byte streams handle I/O of raw binary data.
- ✓ Character streams handle I/O of character data, automatically handling translation to and from the local character set.
- ✓ Buffered streams optimize input and output by reducing the number of calls to the native API.
- ✓ Scanning and formatting allows a program to read and write formatted text.
- ✓ I/O from the command line describes the standard streams and the console object.
- ✓ Data streams handle binary I/O of primitive data type and string values.
- ✓ Object streams handle binary I/O of objects.

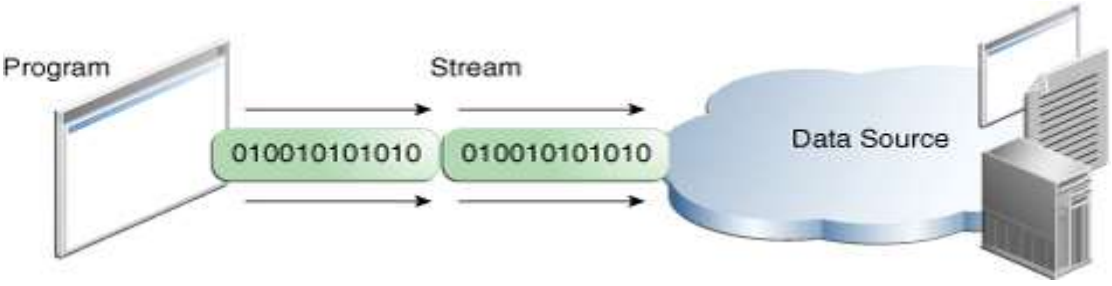
## I/O Streams

- ✓ An I/O stream represents input source or an output destination.
- ✓ A stream represents many kinds of source and destination like disk files & memory arrays.
- ✓ Stream support different kind of data,
  - Simple bytes
  - Primitive data types
  - Localized characters
  - Objects...etc
- ✓ Stream is a communication channel between source and destination & A stream is a sequence of data.

Input Stream – program uses input stream to read the data from a source one item at a time.




Output Stream – program uses output stream to write the data to a destination one item at a time.




Byte Streams

- ✓ Program uses byte stream to perform input & output of byte data. All byte stream classes developed based on InputStream & OutputStream.
- ✓ Program uses byte stream to perform input and output of 8-bit bytes.

Limitation of byte stream

- ✓ It reads the data only one byte at a time hence it takes more time to copy.
- ✓ Must close the streams always.
- ✓ These streams always hitting hard disk to retrieve the data it reduces the performance.

Character streams

- ✓ Program uses character stream to perform input & output of character data.
- ✓ All character stream classes developed based on Reader & Writer classes.
- ✓ **FileReader**
  - It is used to read the data from source one item at a time.
  - To read the data from source use **read ()** method of FileInputStream class.
- ✓ **FileWriter**
  - It is used to write the data to destination one item at a time.
  - To write the data to destination use **write ()** method of FileOutputStream class.

## CharArrayWriter

It is used to write the data to multiple files & this call implements appendable interface.

## Line Oriented I/O

- ✓ In the above two streams (byte & character) it is possible to read only one item at time it increases number of read & write operations hence the performance is decreased.
- ✓ To overcome above limitation to improve performance of the application instead of reading the data item by item, read the data line-by-line format to improve the performance.
- ✓ To perform line-oriented operations, use two classes,
- ✓ **BufferedReader**
  - It is used to read the data from source file in line by line format.
  - To read the data use **readLine ()** method of BufferedReader class.
- ✓ **PrintWriter**
  - It is used to write the data to destination file in line by line format.
  - To write the data to file use **println ()** method.

## Buffered Streams

- ✓ In previous examples we are using un-buffered I/O. This means each read and write request is handled directly by the underlying OS.
- ✓ In normal streams each request directly triggers disk access it is relatively expensive & performance is degraded.
- ✓ Buffered input stream read the data from buffered memory and it interacting with hard disk only when buffered memory is empty.
- ✓ Buffered output stream writes the data to buffer memory.
- ✓ There are four buffered stream classes,
  - BufferedInputStream
  - BufferedOutputStream
  - BufferedReader
  - BufferedWriter

## Serialization

- ✓ The process of saving an object to a file or the process of sending an object across the network is called serialization.
- ✓ To do the serialization we required following classes,
  - FileOutputStream
  - ObjectOutputStream

## Deserialization

- ✓ The process of reading the object from file supported form or network supported form to the java supported form is called deserialization.
- ✓ We can achieve the deserialization by using fallowing classes,
  - FileInputStream
  - ObjectInputStream

## Transient Modifiers

- ✓ Transient modifier is the modifier applicable for only variables and we can't apply for methods and classes.
- ✓ At the time of serialization, if we don't want to save the values of a particular variable to meet security constraints then we should go for transient modifier.
- ✓ At the time of serialization JVM ignores the original value of transient variable and default value will be serialized.

## Nested Classes

- ✓ Declaring the class inside another class is called nested classes.
- ✓ This concept is introduced in the 1.1 version.
- ✓ Without existing one type of object there is no chance of existing another type of object we should use inner classes.
- ✓ Declaring the methods inside another method is called inner methods, but java not supported inner methods concept.
- ✓ There are two types of nested classes in Java,
    - Static nested classes
    - Non static nested classes (these are called inner classes)
        - Normal inner classes
        - Method inner classes
        - Anonymous inner classes

## Uses of nested classes

- ✓ It is the way logically grouping classes that are only used in one place.
- ✓ It increases the encapsulation.
- ✓ It led the more readability and maintainability of the world.
- ✓ Code optimization.

## Enumeration

- ✓ Enumeration is used to declare group of named constants.
- ✓ You can define an **enum** type by using enum keyword.
- ✓ For example, you would specify a **day-of-the-week** enum type as,

```
Public enum Day
{
        SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

- ✓ Every enum constant represents object of type enum.
- ✓ Compiler will generate **.class** file for enum.
- ✓ The enum constants are by default public static final.
- ✓ The main purpose of the enum is to declare the own data types.

```
Enum representation:
enum Directions
{
        NORTH, SOUTH, EAST, WEST;
}
```

```
Internal implementation:
Class Directions
{
        Public static final Directions NORTH = new Directions ();
        Public static final Directions SOUTH = new Directions ();
        Public static final Directions EAST = new Directions ();
        Public static final Directions WEST = new Directions ();
}
```

Java enum is powerful compare to other language enums

- ✓ Enum improves the type safety.
- ✓ In Java it is possible to declare fields, methods, constructors where as it is not possible in other languages.
- ✓ Enum may implements interfaces.
- ✓ Inside the switch it is possible to use enum.
- ✓ Inside the enum it is possible to take main method also.

Note:

- ✓ Enum constructor is by default **private** hence it is not possible to create the object in outside of the enum.
- ✓ Every enum constant contains index value & it starts from 0.
- ✓ Every enum is **final** by default hence other classes are unable to extends.
- ✓ Values () method is used to retrieve the all the constants at a time.
- ✓ Inside the enum it is possible to declare the main method but to execute this main method just execute the .class file.
- ✓ If we declare the enum outside of the class the applicable modifiers are public, <default>, strictfp.
- ✓ If we are declaring enum inside the class the applicable modifiers are public, default, strictfp, private, protected and static.
- ✓ It is not possible to declare the enum inside the methods it means locally.

## Comparator Vs Comparator

| Property | Comparable | Comparator |
|---|---|---|
| Sorting logics | Sorting logics must be in the class whose class objects are sorting. | Sorting logics in separate class hence we are ale to sort the data by using different attributes. |
| Sorting method | **Int compareTo (Object o1)**<br>This method compares this object with o1 objects and returns an integer. Its value has following meaning.<br>**Positive** – this object is greater than o1<br>**Zero** – this object is equals to o1<br>**Negative** – this object is less than o1 | **Int compare (Object o1, Object o2)**<br>This method compares o1 and o2 objects. And returns an integer. Its value has following meaning,<br>**Positive** – o1 is greater than o2<br>**Zero** – o1 is equals to o2<br>**Negative** – o1 is less than o1 |
| Method calling to perform sorting | **Collections.sort (List)**<br>Here objects will be sorted on the basis of compareTo method. | **Collections.sort (List, Comparator)**<br>Here objects will be sorted on the basis of Compare method in Comparator. |
| Package | Java.lang | Java.util |
| Which type of sorting | Default natural sorting order | Customized sorting order |

## Networking (java.net package)

- The process of connecting the resources (computers) together to share the data is called networking.
- Java.net is package it contains number of classes by using that classes we are able to connection between the devices (computers) to share the data between different computing devices.
- Java socket programming provides the facility to share the data between different computing devices.
- In the network we are having two components
    - **Sender** (source): the person who is sending the data is called sender.
    - **Receiver** (destination): the person who is receiving the data is called receiver.
- In the network one system can acts as a sender as well as receiver.
- In the networking terminology we have client and server,
    - Client: who takes the request and who takes the response is called client.
    - Server:
        - The server contains the projects.
        - It takes the request from the client.
        - It identifies the requested resource.
        - It processes the request.
        - It will generate the response to client.

### Categories of Network

We are having two types of networks,

- ✓ Peer-to-peer network
- ✓ Client-server network

**Client-server**

In the client server architecture always client system acts as a client and server system acts as a server.

**Peer-to-peer**

In the peer-to-peer client system sometimes behaves as a server, server system sometimes behaves like a client the roles are not fixed.

**Types of networks**

- ➢ **Intranet**: it is also known as a private network. To share the information in limited area range (within the organization) then we should go for intranet.
- ➢ **Extranet**: this is extension to the private network means other than the organization, authorized persons able to access.
- ➢ **Internet**: It is also known as public network where the data maintained in a centralized server hence, we are having more shareability. And we can access the data from anywhere else.

The frequently used terms in the networking

1. IP Address
2. URL (Uniform Resource Locator)
3. Protocol
4. Port Number
5. MAC address
6. Connection oriented and connectionless protocol
7. Socket

## IP Address

- ✓ IP address is a unique identification number given to the computer to identify the computer uniquely in the network.
- ✓ The IP address is uniquely assigned to the computer, it's should not duplicate.
- ✓ The IP address range is 0 – 255.
  **EX:** 125.0.4.255 --- valid
  124.654.5.6 --- invalid
- ✓ Each and every website contains its own IP address, we can access the sites through the names otherwise IP address.
  **Ex:**
  Site name – www.google.com
  IP address – 74.125.224.72
- ✓ **Example**;

```
Import java.net.*;
Import java.util.*;
Class Test
{
        Public static void main (String args [])
        {
                Scanner s = new Scanner (System.in);
                System.out.println ("please enter site name");
                String sitename = s.nextLine ();
                InetAddress in = InetAddress.getByName (sitename);
                System.out.println ("the ip address is:"+in);
        }
}
```

```
OUTPUT:
C:\ratan>java Test
Please enter site name
www.google.com
the IP address is: www.google.com/216.58.199.196
```

## Protocol

- ✓ The protocol is a set of rules followed in communication.
- ✓ TCP (Transmission Control Protocol) (Connection oriented Protocol)
- ✓ UDP (User Datagram protocol) (connectionless protocol)
- ✓ Telnet
- ✓ SMTP (Simple Mail Transfer Protocol)
- ✓ IP (Internet Protocol)

## MAC

MAC address is a unique identifier for NIC (Network interface protocol). A network protocol can have multiple NIC but one unique MAC.

## Port Number

The port number is used to identify the different applications uniquely. And it is associated with the IP address for communication between two applications.

## URL (Uniform Resource Locator)

- URL is a class present in the java.net package.
- By using the URL, we are accessing some information present in the world wide web.

  Ex:

  http://www.palmertechnology.com:10/corejava_ratan.asp

  protocol path port path
- The URL contains information like,
  - Protocol: **http://**
  - Server name IP address: **www.palmertechnology.com**
  - Port number of the particular application and it is optional: **10**
  - File name or directory name: **Corejava_ratan.asp_**

## Communication using networking

In the networking, it is possible to do two types of communications,

- Connection Oriented (TCP/IP Communication)
- Connection Less (UDP Communication)

## Connection Oriented

- ✓ In this type of communication, we are using combination of two protocols TCP, IP.
- ✓ In this communication acknowledgement sent by receiver so it is reliable but slow.
- ✓ To achieve the following communication the java peoples are provided the following classes.
  - **Socket**
  - **ServerSocket**

## Connection Less (UDP)

- ✓ UDP is a protocol by using this protocol we are able to send data without using physical connection.
- ✓ In this communication acknowledgement not sent by receiver so, it is not reliable but fast.
- ✓ This is a light weight protocol because no need of the connection between the client and server. To achieve the UDP communication the java peoples are provided the following classes.
    - **Datagram Packet**
    - **Datagram Socket**

## Socket

1. Socket is used to create the connection between the client and server.
2. Socket is nothing but a combination of IP address and port number.
3. The socket is created at client side.

Ex:

Socket s = new Socket (int IPAddress, int PortNumber)

Socket s = new Socket ("125.125.0.5", 123) ---- Server IPAddress, Server Port Number

Socket s = new Socket (String HostName, int PortNumber); ------ Socket s = new Socket (Palmer Technology, 123);

## Java.awt package

- ✓ AWT (Abstract Window Toolkit) is an **API**. It supports graphical user interface programming.
- ✓ AWT components are platform dependent it displays the application according to the view of operating system.
- ✓ By using java.awt package, we are able to prepare static components to provide the dynamic nature to the component use **java.awt. event** package (it is a sub package of java.awt).
- ✓ This application not providing very good look and feel hence the normal users facing problem with these types of applications.
- ✓ By using AWT we are preparing application these applications are called console based or CUI application.

## NOTE

- ✓ Java.awt package is used to prepare static components.
- ✓ Java.awt.event package is used to provide the life to the static components.

## GUI (Graphical User Interface)

- ✓ It is a mediator between end user and the program.
- ✓ AWT is a package, it will provide very good predefined support to design GUI applications.

## Component

- ✓ The root class of java.awt package is Component class.
- ✓ Component is an object which is displayed pictorially on the screen.
- ✓ Ex: Button, Label, TextField...etc

## Container

- ✓ It is a component in awt that contains another component like Button, TextField ... etc.
- ✓ Container is a sub class of component class.
- ✓ The classes that extend container classes those classes are containers such as **Frame**, **Dialog** and **Panel**.

## Event

- ✓ The event nothing but an action generated on the component or the change is made on the state of the object.
- ✓ **Ex:** Button clicked, Checkbox checked, Item selected in the list, Scrollbar scrolled horizontal/vertically.

## AWT Component Hierarchy

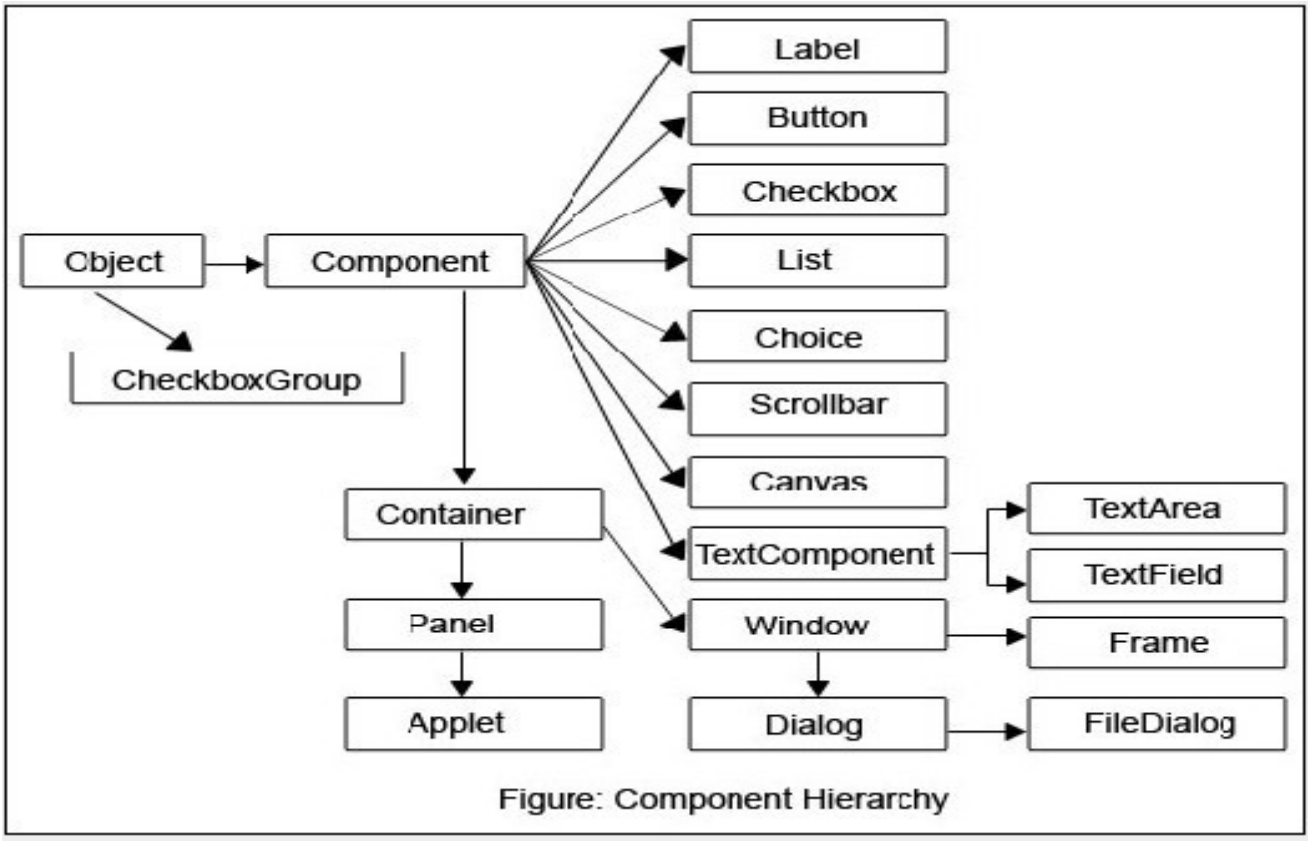



Figure: Component Hierarchy

## Java.awt.Frame

Frame is a container, it contains other components like title bar, Button, Text Field … etc.

- When we create a frame class object. Frame will be created automatically with invisible mode, so to provide visible nature to the frame use **setVisible ()** method of Frame class.
  **Public void setVisible (boolean b)**
  Where,
  b == true visible mode
  b == false means invisible mode
- When we created a frame, the frame is created with initial size 0-pixel heights & 0-pixel width hence it is not visible. To provide particular size to the frame use **setSize ()** method.
  **Public void setSize (int width, int height)**
- To provide title to the frame use,
  **Public void setTitle (String Title)**
- When we create a frame, the default background color of the frame is white. If you want to provide particular color to the frame, we have to use the following method.
  **Public void setBackground (color c)**

## Creating a Frame

There are two approaches to create a frame in Java

1. By creating object of the Frame class
2. By extending the Frame class

### Approach – 1

Creation of Frame by creating Object of Frame class

```
import java.awt. *;
class frame
{
        public static void main (String args [])
        {
                Frame f = new Frame ();
                f.setVisible (true);
                f.setSize (400, 400);
                f.setBackground (Color.blue);
                f.setTitle ("myFrame");
        }
}
```

Approach – 2

Taking user defined class by extending Frame class

```
import java.awt. *;
class MyFrame extends Frame
{
        MyFrame ()
        {
                setVisible (true);
                setSize (500, 500);
                setTitle ("myframe");
                setBackground (Color.green);
        }
        public static void main (String args [])
        {
                MyFrame f = new MyFrame ();
        }
}
```

Displaying Text on the Screen

1. To display some textual message on the frame override **paint** () method.
   **Public void paint (Graphics g)**
2. To set a particular font to the text use Font class present in java.awt package.
   Font f = new Font (String type, int style, int size);
   Font f = new Font ("arial", Font.Bold, 30);

Example

```
import java.awt.*;
class MyFrame extends Frame
{
        public static void main (String args [])
        {
                MyFrame t = new MyFrame ();
                t.setVisible (true);
                t.setSize (500, 500);
                t.setTitle ("myframe");
                t.setBackground (Color.red);
        }
        public static paint (Graphics g)
        {
                Font f = new Font ("arial", Font.ITALIC, 25);
                g.setFont (f);
                g.drawString ("hi ratan how r u", 100, 100);
        }
}
```

## Preparation of Components

- **Label**: Label is a constant text which is displayed along with a TextField or TextArea.
  Constructor: **Label l = new Label ();**
  **Label l = new Label ("user name");**
- **TextField**: TextField is an editable area & it is possible to provide single line of text.
  - Enter Button doesn't work on TextField.
  - To set Text to the TextArea : **t.setText ("ashirbad");**
  - To get the text from TextArea : **String s = t.getText ();**

  Constructor: TextField tx = new TextField ();
  TextField tx = new TextField ("ratan");
- **TextArea**: TextArea is a Editable Area & enter button will work on TextArea.
  - To set text to TextArea: ta.setText ("ashirbad");
  - To get the text form TextArea: String s = ta.getText ();

  Constructor: **TextArea t = new TextArea ();**
  **TextArea t = new TextArea (int rows, int columns);**
  **TextArea t = new TextArea (String text, int rows, int columns);**

## Layout Manager

- ✓ The layout managers are used to arrange the components in a Frame in particular manner **or** A layout manager is an object that controls the size and the position of components in a container.
- ✓ Different layouts in Java,
  - **Java.awt.FlowLayout**
  - **Java.awt.BorderLayout**
  - **Java.awt.GridLayout**
  - **Java.awt.CardLayout**
  - **Java.awt.GridBagLayout**

## Java.awt.FlowLayout

The FlowLayout is used to arrange the components into row by row format. Once the first row is filled with components then it is inserted into second row. And it is the default layout of the applet.

## Java.awt.BorderLayout

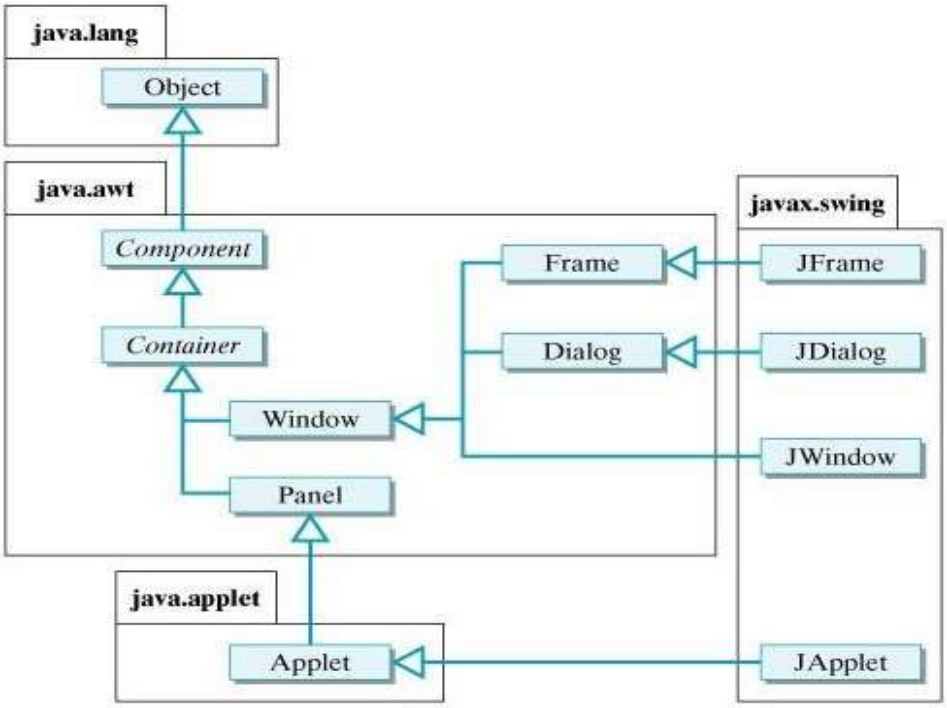
- ✓ The BorderLayout is dividing the frame into five areas north, south, east, west, center so, we can arrange the components in these five areas.
- ✓ To represent these five areas BorderLayout is providing the following 5-constants,
  - Public static final java.lang.String NORTH;
  - Public static final java.lang.String SOUTH;
  - Public static final java.lang.String EAST;
  - Public static final java.lang.String WEST;
  - Public static final java.lang.String CENTER;

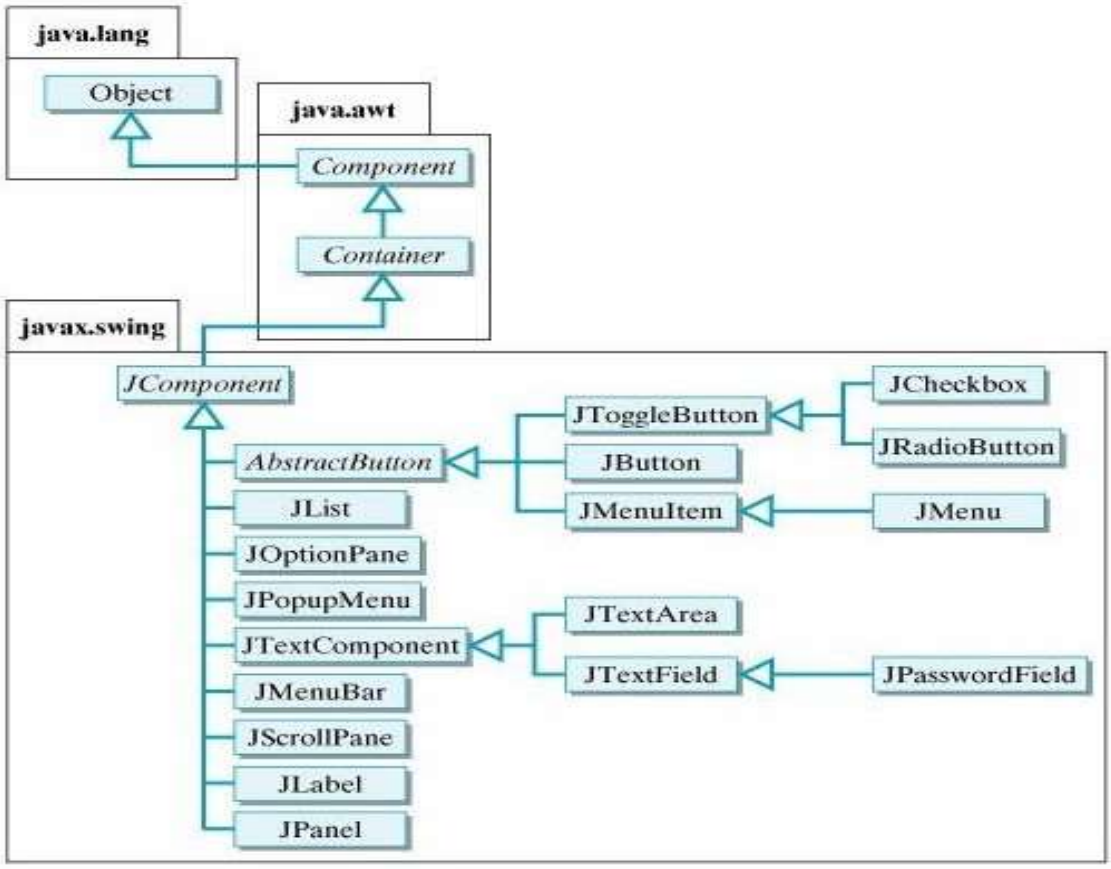
## Swings

- ✓ Sun Micro Systems introduced AWT to prepare GUI applications but awt not satisfy the client requirement.
- ✓ An alternative to AWT Netscape communication Corporation has provided set of GUI components in the form of IFC (Internet Foundation Class) but IFC also provides less performance and it is not satisfying the client requirement.
- ✓ In the above context (Sun & Netscape) combine and introduced common product to design GUI applications is called JFS (Java Foundation classes). Then it is renamed as Swings.

## Difference Between AWT and Swings

- ✓ AWT components are platform dependent but swings components are platform independent because these components are completely written in Java.
- ✓ AWT components are heavy weight component but swing component are light weight component.
- ✓ AWT components consume a greater number of system resources swings consume a smaller number of system resources.
- ✓ AWT provides a smaller number of components whereas a swing provides a greater number of components.
- ✓ AWT doesn't provide Tooltip Test support but swing components have provided Tooltip Test support.
- ✓ In case of AWT we will add the GUI components in the Frame directly but swing we will add all GUI components to panes to accommodate GUI components.
- ✓ The awt classes & interfaces are present in java.awt packages & swing classes are present in javax.swing package.
- ✓ In AWT to close the window:
  **WindowListener windowAdaptor**
  In case of swing use small piece of code:
  F.setDefaultCloseOption (JFrame.EXIT-ON-CLOSE);


java.lang
Object
java.awt
Component
Container
Window
Panel
Frame
Dialog
javax.swing
JFrame
JDialog
JWindow
java.applet
Applet
JApplet


java.lang
Object
java.awt
Component
Container
javax.swing
JComponent
AbstractButton
JList
JOptionPane
JPopupMenu
JTextComponent
JMenuBar
JScrollPane
JLabel
JPanel
JToggleButton
JButton
JMenuItem
JCheckbox
JRadioButton
JMenu
JTextArea
JTextField
JPasswordField

## Java Applet (java.awt.Applet)

- ✓ The applet is runs on browser window to display the dynamic content on browser window.
- ✓ The applet does not contain main methods to start the execution but it contains life cycle methods these methods are automatically called by web browser.
- ✓ To run the applet in browser window we need to install plugin in.

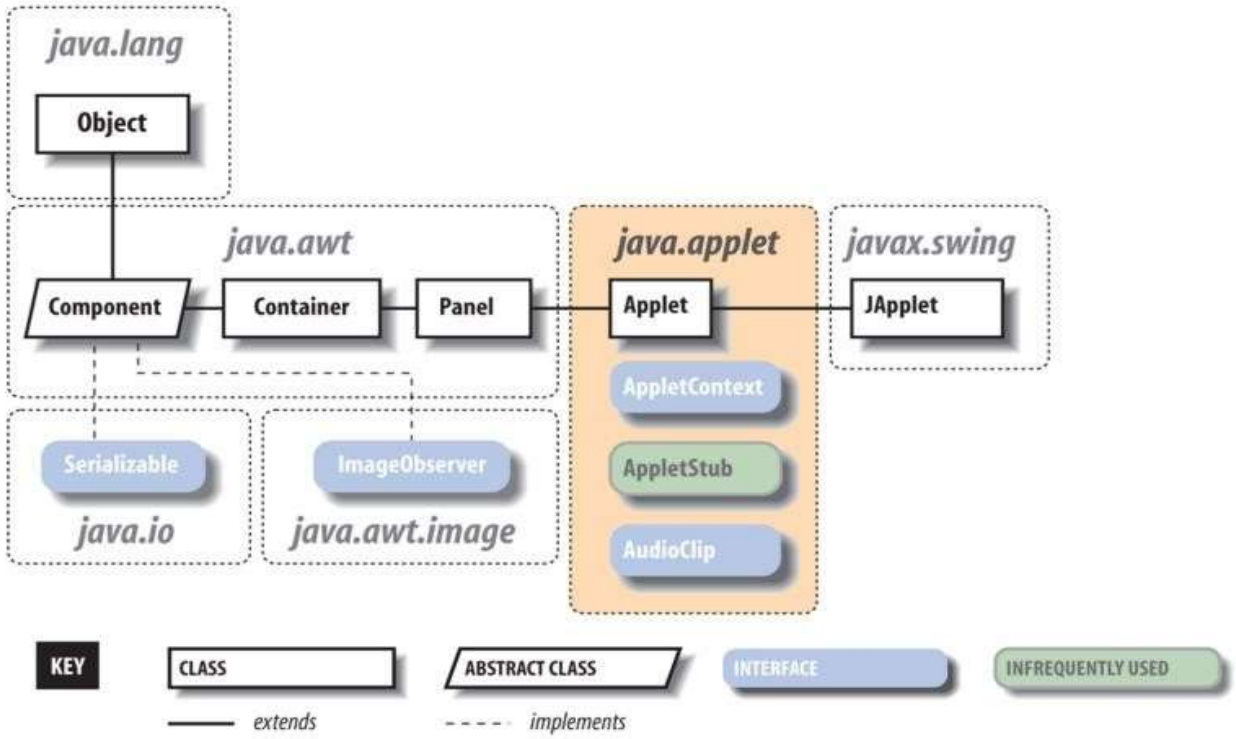



*Figure 23-1. The java.applet package*

- ✓ The applet contains four life cycle methods
  - Init ()
  - Start ()
  - Stop ()
  - Destroy ()
- ✓ These methods are called by applet viewer or web browser,
  - **Init ()** - used to initialize the applet and it called only once.
  - **Start ()** – it invoked after init () method to start the applet then the applet become visible.
  - **Stop ()** – it is used to stop the applet then the applet become invisible.
  - **Destroy ()** – it called after stop () method. It gives to applet last chance to cleanup.
- ✓ Java.awt. Component class provide one life cycle method of applet,

  **Public void paint (Graphics g)**

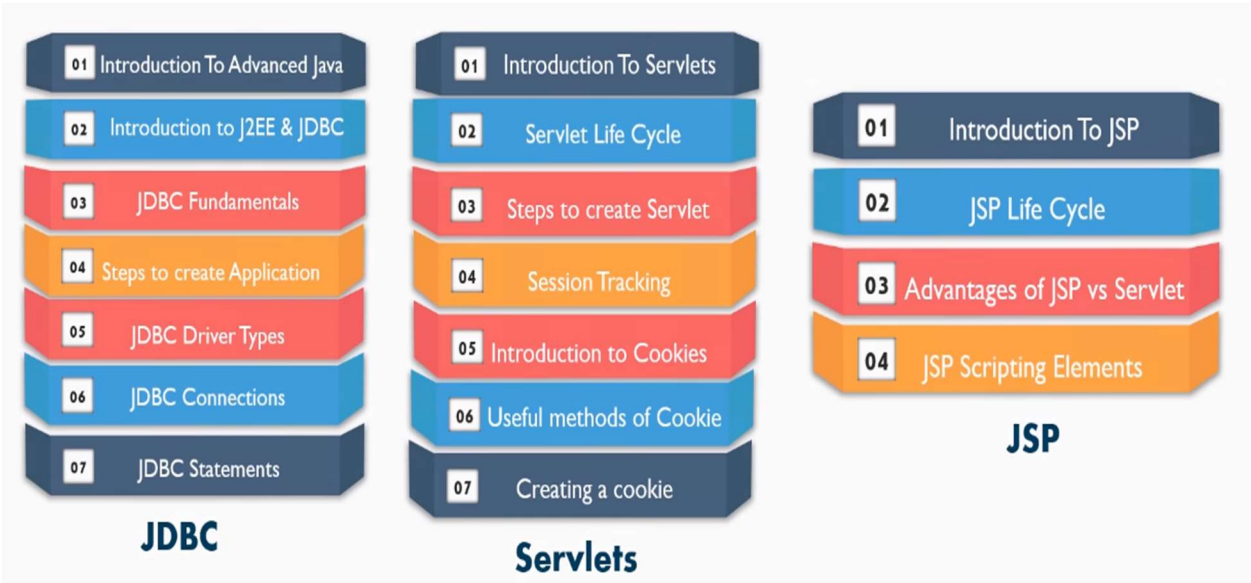
  The above method used to paint the applet to print the data in applet.

- ✓ **Steps to design the applications,**
  - Create the applet by extending Applet class.
  - Then configure the applet in html code.

- ✓ **Running applet**

  It is possible to run the applet in two ways,
  - By using html file
    - Configure the applet in html file then open the html file in browser window.
    - Click on **Firstapplet.html (Program name)** then the apple is displayed on browser window.
  - By using applet viewer tool
    - Run the application by using below command

      **Appletviewer Firstapplet.html**




Ashirbad Swain – 2020

**Advanced Java Contents**

- Introduction to Advanced Java

- JDBC
  - What is JDBC?
  - JDBC Architecture
  - Steps to create JDBC Application
  - JDBC Driver Types & Connections

- Java Servlets
  - Introduction to Java Servlets
  - Servlet Life Cycle
  - Steps to create Servlet
  - Session Tracking in Servlets

- JSP
  - Introduction to JSP
  - Lifecycle of JSP
  - JSP Scripting Elements


## Introduction

**Advanced Java** is everything that goes beyond **Core Java** – most importantly the APIs defined in Java Enterprise Edition, includes Servlet Programming, Web Services, the Persistence API, etc. It is a Web & Enterprise application development platform which basically follows client & server architecture.

## Advantages of Advanced Java

- Advanced Java i.e, J2EE (Java Enterprise Edition) gives you the library to understand the **Client-Server architecture** for Web Application Development, which Core Java doesn't support.
- J2EE is platform independent, **Java Centric** environment for developing, building and deploying Web-based applications online. It also consists of a set of services, APIs and protocols, which provides the functionality that is necessary for developing multi-tiered web-based applications.
- You will be able to work with **Web and Application Servers** like Apache Tomcat, Glassfish etc and understand the communication over HTTP protocol, in Core Java, it is not possible.
- There are a lot of Advanced **Java Frameworks** like Spring, JSF, structs etc. which enable you to develop a secure transactions-based web apps for the domains like E-commerce, Banking, Legal, Financial, Healthcare, Inventory etc.
- To work and understand the **hot technologies** like Hadoop and Cloud services, you should be prepared with core and advanced Java concepts.

## JDBC (Java Database Connectivity)

### Introduction

JDBC is a standard Java API for a database-independent connectivity between the Java Programming language and a wide range of databases.

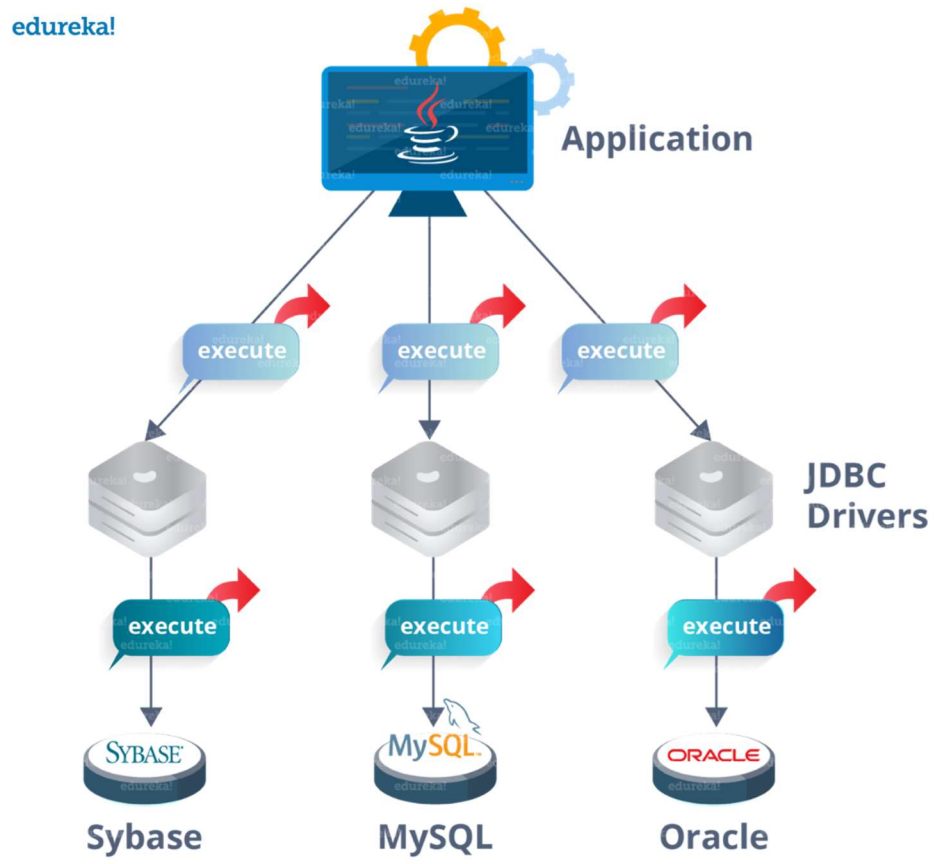
This application program interface lets you encode the access request statements, in Structured Query Language (SQL). They are then passed to the program that manages the database. It mainly involves opening a connection, creating a SQL Database, executing SQL queries and then arriving at the output.

We can use JDBC API to access tabular data stored in any relational database. By the help of JDBC API, we can save, update, delete and fetch data from the databases. It is similar to the Open Database Connectivity (ODBC) provided by Microsoft.

### JDBC Architecture

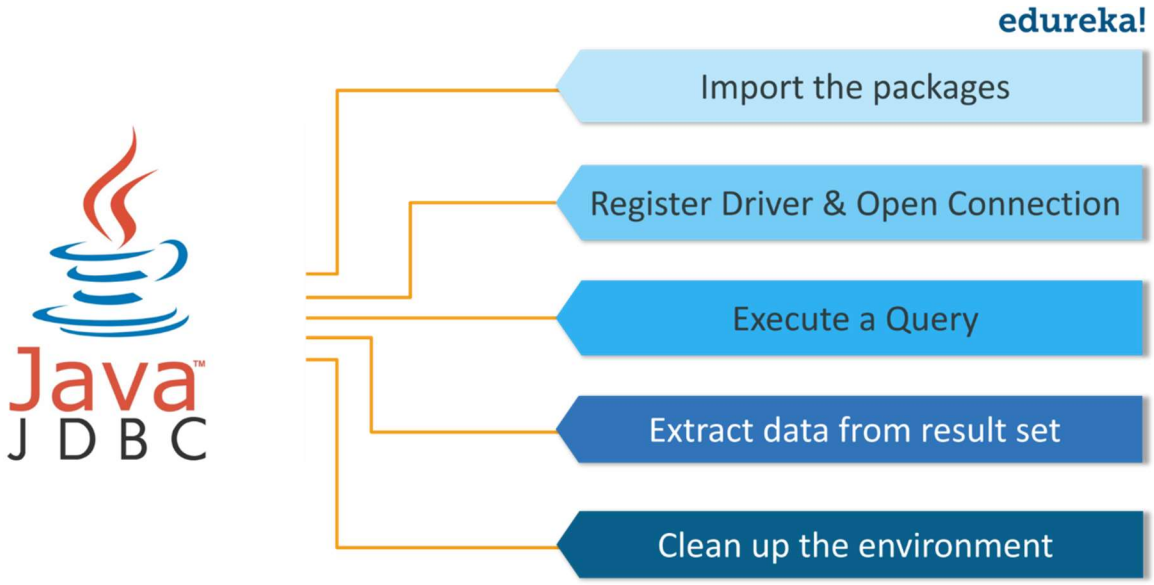
The JDBC API supports both two-tier and three-tier processing models for database access but in general, JDBC Architecture consists of two layers:

1. **JDBC API**: This provides the application-to-JDBC Manager Connection.
2. **JDBC Driver API**: This supports the JDBC Manager-to-Driver Connection.



Steps to create JDBC Application

In order to create JDBC Application, we need to follow few steps.




1. Import the Packages: You need to include the packages containing the JDBC classes needed for database programming. Most often, using import java.sql. * will surface.
2. Register the JDBC driver: Here you have to initialize a driver so that you can open a communication channel with the database.
3. Open a connection: Here, you can use the getConnection () method to create a connection object, which represents a physical connection with the database.
4. Execute a query: Requires using an object of type statement for building and submitting an SQL statement to the database.
5. Extract data from result set: Requires that you use the appropriate getXXX () method to retrieve the data from the result set.
6. Clean up the Environment: Requires explicitly closing all database resources versus relying on the JVM's garbage collection.

The JDBC API uses a driver manager and database-specific drivers to provide transparent connectivity to heterogeneous databases. The JDBC driver manager ensures that the correct driver is used to access each data source. The driver manager is capable of supporting multiple concurrent drivers connected to multiple heterogeneous databases.

## Common JDBC Components

The JDBC API provides the following interfaces and classes:

- Driver Manager is used to manage a list of database drivers. The first driver that recognizes a certain subprotocol under JDBC will be used to establish a database connection.
- Driver is an interface that handles the communications with the database server. It also abstracts the details associated with working with Driver objects.
- Connection is an interface that consists all the methods required to connect to a database. The connection object represents communication context, i.e. all communication with the database is through connection object only.

## JDBC Driver Types

JDBC drivers implement the defined interfaces in the JDBC API, for interacting with your database server. Essentially, a **JDBC Driver** makes it possible to **do** three things:

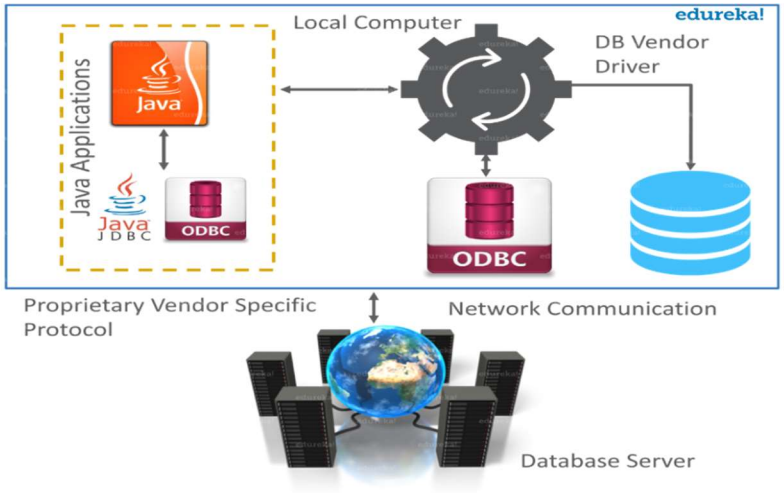
1. Establish a connection with a data source
2. Send queries and update statements to the data source
3. Process the results

For Example, the use of JDBC drivers enables you to open a database connection to interact with it by sending SQL or database commands.

There are **4** types of drivers, namely:

## Type 1: JDBC-ODBC Bridge Driver

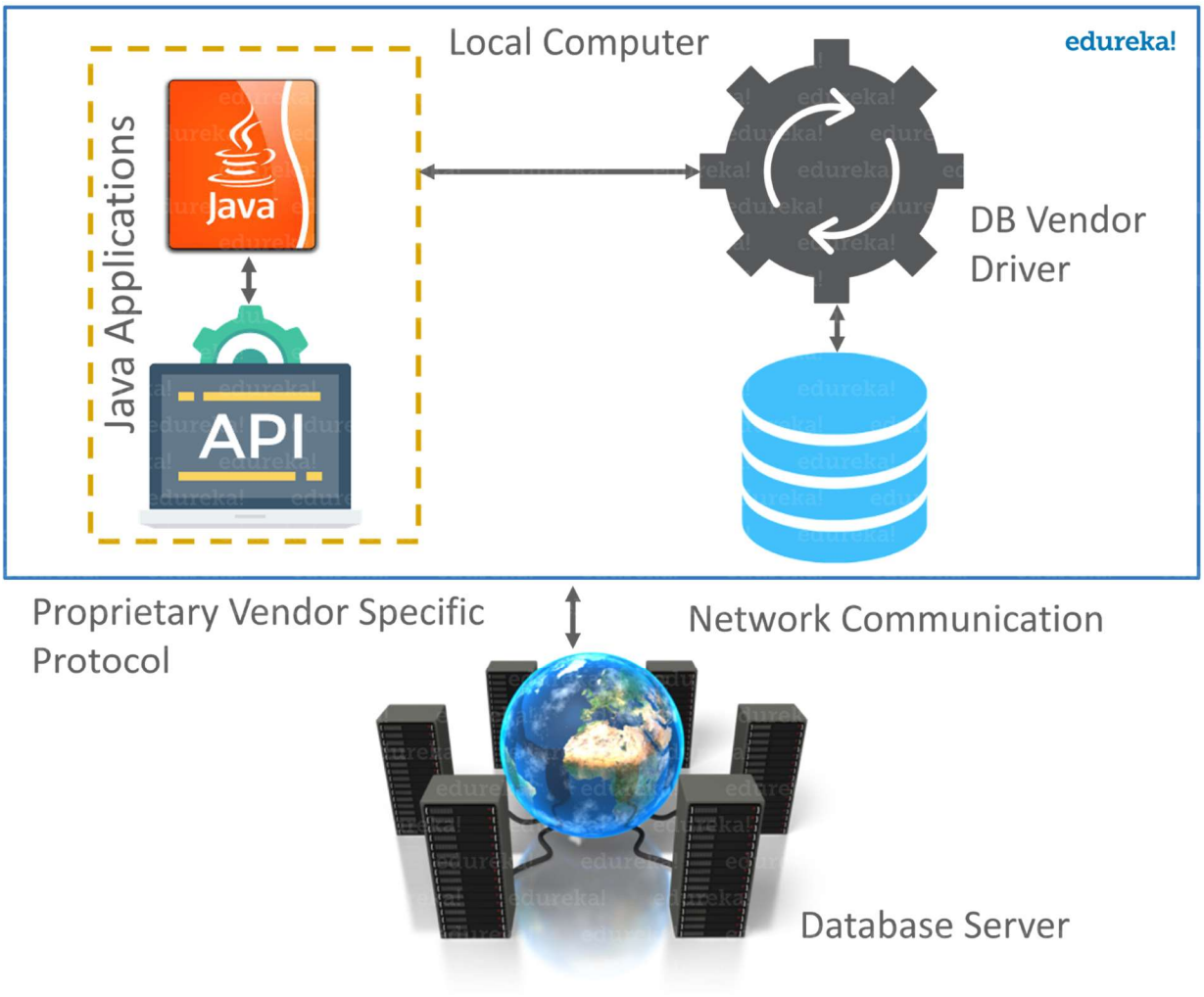
In Type 1 Driver, a JDBC bridge accesses ODBC drivers installed on each client machine. Further, ODBC configures Data Source Name (DSN) that represents the target database.




When Java first came out, this was a useful driver because most databases only supported ODBC (Open Database Connectivity) access but now this type of driver is recommended only for experimental use or when no other alternative is available.

## Type 2: JDBC-Native API

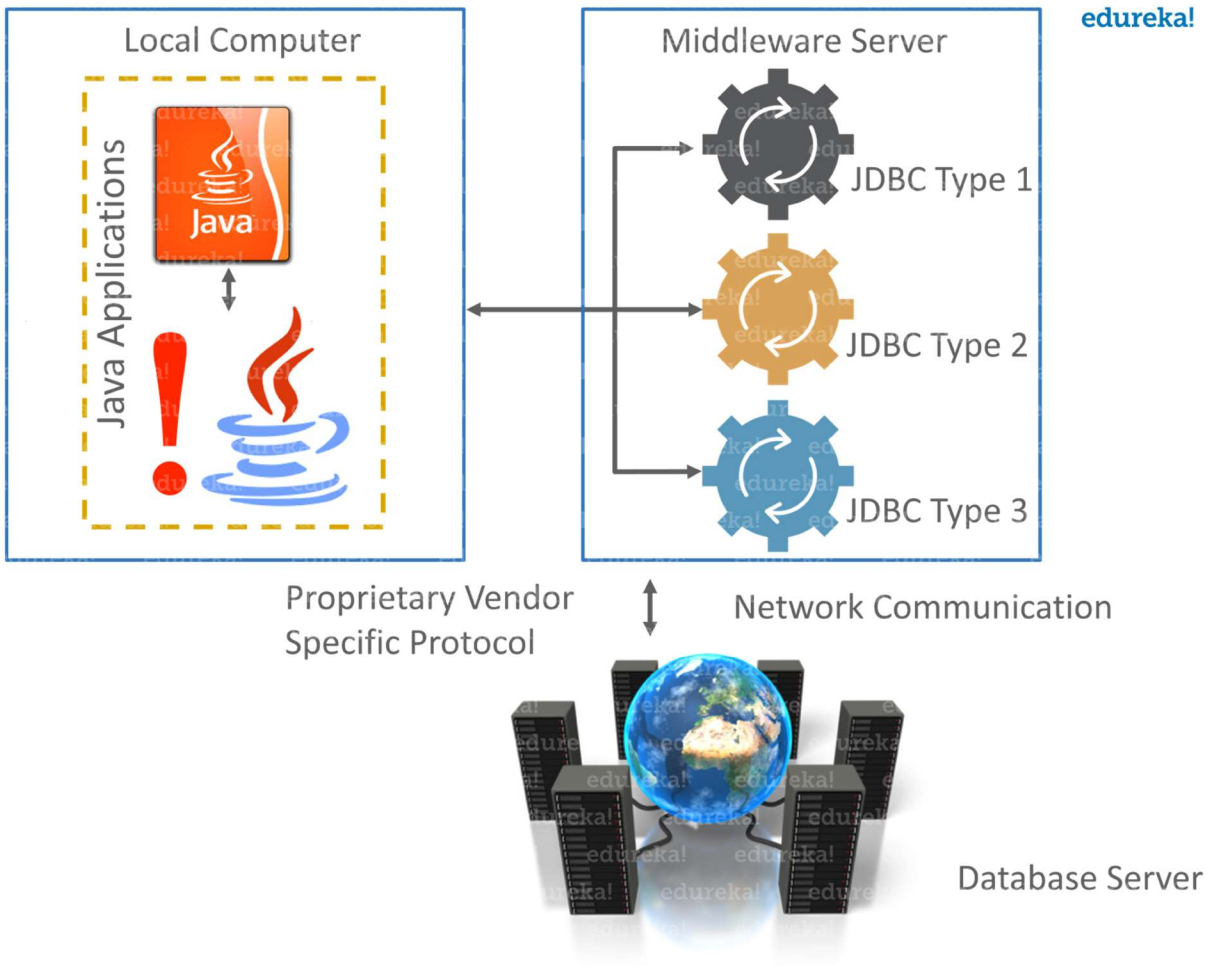
In a Type 2 driver, JDBC API calls are converted into native C/C++ API calls, which are unique to the database. These drivers are typically provided by the database vendors and used in the same manner as the JDBC-ODBC Bridge. The vendor-specific driver must be installed on each client machine.




The **Oracle Call Interface (OCI)** driver is an example of a Type 2 driver.

Type 3: JDBC-Net Pure Java

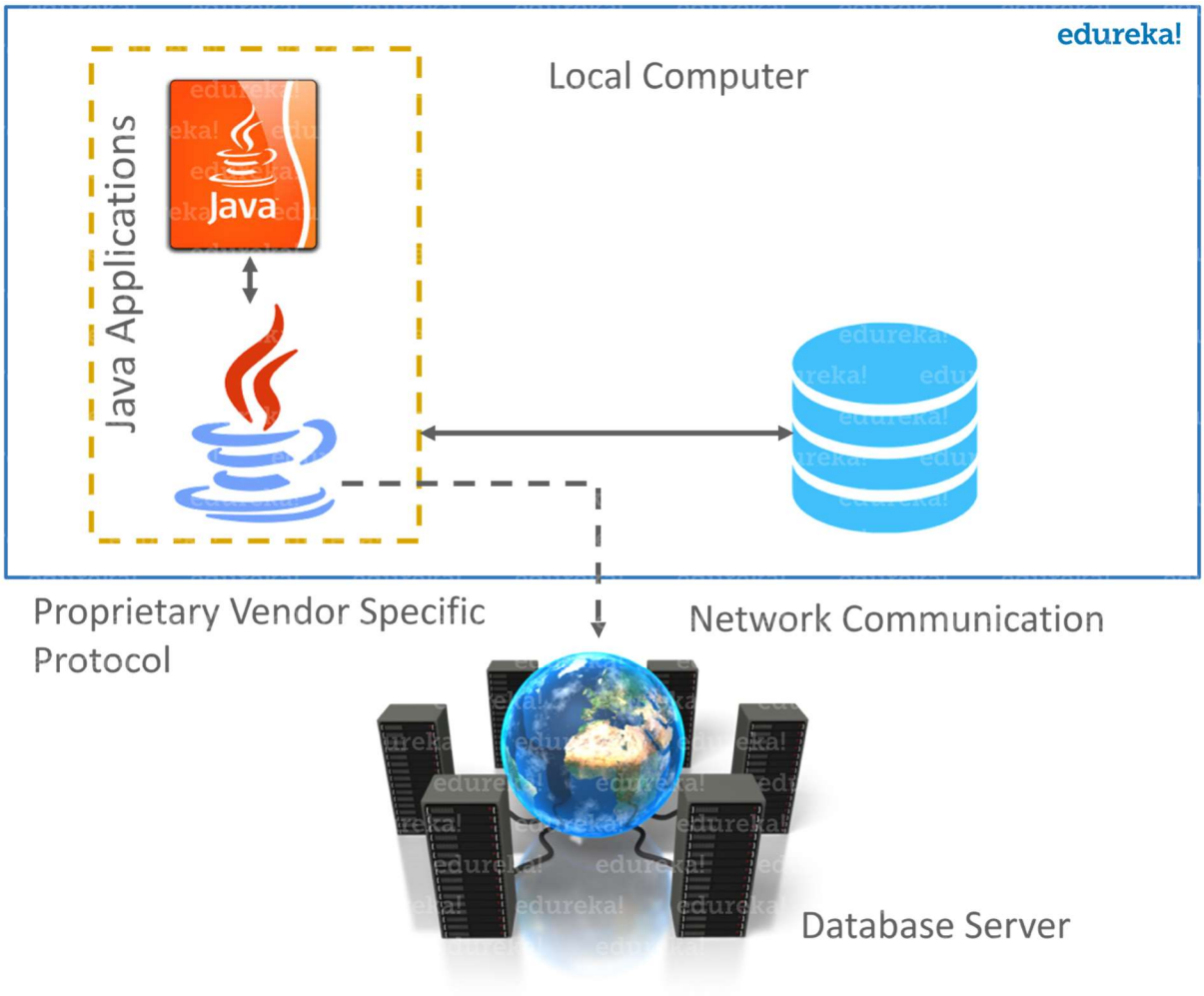
In a Type 3 driver, a three-tier approach is used to access databases. The JDBC clients use standard network sockets to communicate with a middleware application server. The socket information is then translated by the middleware application server into the call format required by the DBMS and forwarded to the database server.




This kind of driver is extremely flexible since it requires no code installed on the client and a single driver can actually provide access to multiple databases. You can think of the application server as a JDBC "proxy", meaning that it makes calls for the client application. As a result, you need some knowledge of the application server's configuration in order to effectively use this driver type. Your application server might use a Type 1, 2 or 4 drivers to communicate with the database.

Type 4: 100% Pure Java

In a Type 4 driver, a pure Java-based driver communicates directly with the vendor's database through a socket connection. This is the highest performance driver available for the database and is usually provided by the vendor itself.




This kind of driver is extremely flexible, you don't have to install special software on the client or server. Further, these drivers can be downloaded dynamically.

**MySQL's Connector/J** driver is a Type 4 driver, Because of the proprietary nature of their network protocols, database vendors usually supply type 4 drivers.

So here comes the Question, which Driver should be used?

- If you are accessing one type of database, such as Oracle, Sybase or IBM, the preferred driver type is 4.
- If your Java application is accessing multiple types of databases at the same time, type 3 is the preferred driver.
- Type 2 drivers are useful in situations, where a type 3 or type 4 driver is not available yet for your database.
- The type 1 driver is not considered a deployment-level driver and is typically used for development and testing purposes only.

JDBC Connections

- Import JDBC Packages: Add **import** statements to your Java program to import required classes in your Java code.
- Register JDBC Driver: This step causes the JVM to load the desired driver implementation into memory so that it can fulfill your JDBC requests. There are 2 approaches to register a driver.
    - The most common approach to register a driver is to use Java's **forName ()** method to dynamically load the driver's class file into memory, which automatically registers it. This method is preferable because it allows you to make the driver registration configurable and portable.

```
1. try {
2. Class.forName("oracle.jdbc.driver.OracleDriver");
3. }
4. catch (ClassNotFoundException ex) {
5. System.out.println("Error: unable to load driver class!");
6. System.exit(1);
7. }
```

    - The second approach you can use to register a driver is to use the static **registerDriver ()** method.

```
1. try {
2.   Driver myDriver = new oracle.jdbc.driver.OracleDriver();
3.   DriverManager.registerDriver(myDriver);
4.  }
5. catch(ClassNotFoundException ex)
6. {
7. System.out.println("Error: unable to load driver class!");
8. System.exit(1);
9. }
```

You should use the **registerDriver ()** method if you are using a non-JDK compliant JVM, such as the one provided by Microsoft.

- Database URL Formulation: This is to create a properly formatted address that points to the database to which you wish to connect. After you've loaded the driver, you can establish a connection using the **DriverManager.getConnection ()** method.

  DriverManager.getConnection () methods are:
  - getConnection (String url)
  - getConnection (String url, Properties prop)
  - getConnection (String url, String user, String password)

  here, each form requires a database **URL**. A database URL is an address that points to your database.

  A table lists down the popular JDBC driver names and database URL.

| RDBMS | JDBC Driver Name | URL |
|---|---|---|
| 1. MYSQL | com.mysql.jdbc.Driver | jdbc:mysql://hostname/ databaseName |
| 2. Oracle | oracle.jdbc.driver.OracleDriver | jdbc:oracle:thin:@hostname:port Number:databaseName |
| 3. Sybase | com.Sybase.jdbc.SybDriver | jdbc:sybase:Tds:hostname: port Number/databaseName |

- **Create a connection object:**
  You can simply create or open a connection using database url, username, and password and also using properties object. A properties object holds a set of keyword-value pairs. It is used to pass the driver properties to the driver during a call to the **getConnection ()** method.
- **Close**
  At the end of your JDBC program, we have to close all the database connections to end each database session. However, if you forget, Java's garbage collector will close the connection when it cleans up stale objects.

```
conn.close();// Used to close the connection
```

## Servlets

### Introduction to Servlets

A **Servlet** is a Java Programming language class that is used to extend the capabilities of servers that host applications accessed by means of a request-response programming model.

Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers.

**Servlets** can be described as follows:

- Servlet is a technology which is used to create a web application.


- It is an API that provides many interfaces and classes including documentation.
- Servlets is an interface that must be implemented for creating any Servlet.
- It is also a class that extends the capabilities of the servers and responds to the incoming requests. It can respond to any requests.

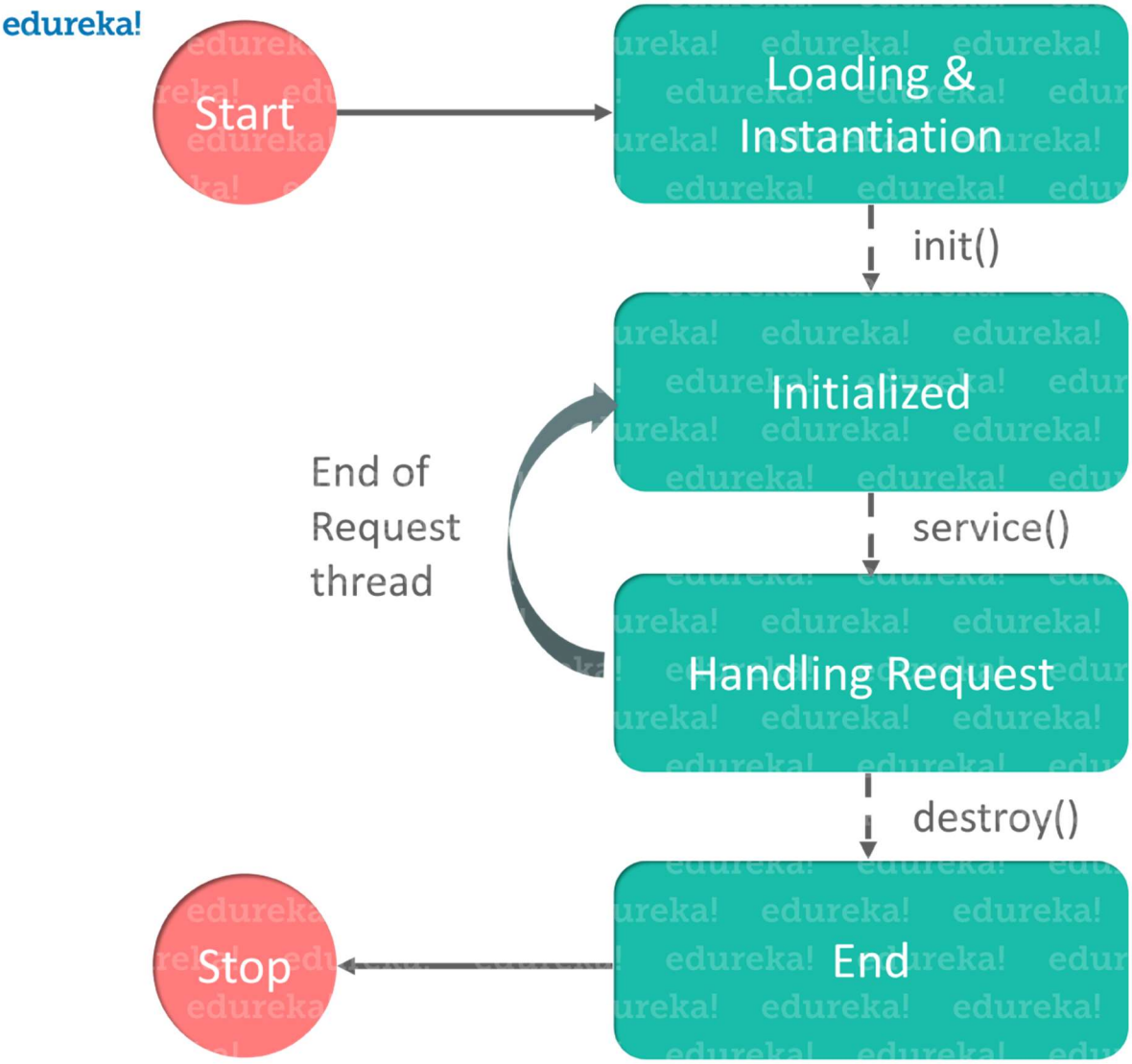
## Servlet Life Cycle

The entire life cycle of a Servlet is managed by the **Servlet container** which uses the **javax.servlet.Servlet** interface to understand the Servlet object and manage it.

### Stages of the Servlet Life Cycle

The Servlet life cycle mainly goes through **four** stages:

- Loading a Servlet
- Initializing the Servlet
- Request handling
- Destroying the Servlet



1. Loading a Servlet

   The first stage of the Servlet life cycle involves loading and initializing the Servlet by the Servlet container. The web container or Servlet Container can load the Servlet at either of the following two stages;

   - Initializing the context, on configuring the Servlet with a zero or positive integer value.
   - If the Servlet is not preceding the stage, it may delay the loading process until the Web container determines that this Servlet is needed to service a request.

2. Initializing a Servlet

   After the Servlet is instantiated successfully, the Servlet container initializes the instantiated Servlet object. The container initializes the Servlet object by invoking the **init (ServletConfig)** method which accepts ServletConfig object reference as a parameter.

3. Handling request

   After initialization, the Servlet instance is ready to serve the client requests. The Servlet container performs the following operations when the Servlet instance is located to service a request.

   - It creates the **ServletRequest** and **ServletResponse**. In this case, if this is an HTTP request then the Web container creates **HttpServletRequest** and **HttpServletResponse** objects which are subtypes of the ServletRequest and ServletResponse objects respectively.

4. Destroying a Servlet

   When a Servlet container decides to destroy the Servlet, it performs the following operations,

   - It allows all the threads currently running in the service method of the Servlet instance to complete their jobs and get released.
   - After currently running threads have completed their jobs, the Servlet container calls the **destroy ()** method on the Servlet instance.

   After the **destroy ()** method is executed, the Servlet container releases all the references of this Servlet instance so that it becomes eligible for garbage collection.

Steps to Create Servlet

1. Create a directory structure
2. Create a Servlet
3. Compile the Servlet
4. Add mapping to web.xml file
5. Start the Server and deploy the project
6. Access the Servlet

**Step 1:**

To run a servlet program, we should have Apache tomcat server installed and configured. Once the server is configured, you can start with your program.

**Step 2:**

For a servlet program, you need **3 files** – index.html file, Java class file and web.xml file. The very first step is to create Dynamic Web Project and then proceed further.

**Step 3:**

Now let's see how to add 2 numbers using servlets and display the output in the browser.

First. I will write **index.html** file

```
<!DOCTYPE html>
<html>
<body>


<form action ="add">
 Enter 1st number: <input type="text" name="num1">
 Enter 2nd number: <input type="text" name="num2">
<input type ="submit">
</form>


</body>
</html>
```

Above program creates a form to enter the number for the addition operation.

**Step 4:**

Now without the Java class file, you can't perform addition on 2 numbers. So, let's write a **class file**.

```
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class Add extends HttpServlet{
    public void service(HttpServletRequest req, HttpServletResponse res) throws IOException
    {
        int i = Integer.parseInt(req.getParameter("num1"));
        int j = Integer.parseInt(req.getParameter("num2");
        int k= i+j;
        PrintWriter out = res.getWriter();
        out.println("Result is"+k);
    }
}
```

**Step 5:**

After writing the Java class file, the last step is to add mappings to the **web.xml** file.

**Step 6:**

**Web.xml** file be present in the **WEB-INF** folder of your web content. If it is not present, then you can click on Deployment Descriptor and click on Generate Deployment Descriptor Stub.

**Step 7:**

After that, you can proceed further and add the mappings to it.

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <web-app xmlns:xsi="<a href="http://www.w3.org/2001/XMLSchema-
    instance">http://www.w3.org/2001/XMLSchema-instance</a>" xmlns=<"<a
    href="http://java.sun.com/xml/ns/javaee">http://java.sun.com/xml/ns/javaee</a>"
    xsi:schemaLocation="<a
    href="http://java.sun.com/xml/ns/javaee">http://java.sun.com/xml/ns/javaee</a> <a
    href="http://java.sun.com/xml/ns/javaee/web-
    app_3_0.xsd">http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd</a>"</em>
    version=<em>"3.0"</em>>
3.   <display-name>Basic</display-name>
4.   <servlet>
5.    <servlet-name>Addition</servlet-name>
6.    <servlet-class>edureka.Add</servlet-class>
7.   </servlet>
8.   <servlet-mapping>
9.    <servlet-name>Addition</servlet-name>
10.  <url-pattern>/add</url-pattern>
11. </servlet-mapping>
12. <welcome-file-list>
13.   <welcome-file>index.html</welcome-file>
14.  </welcome-file-list>
15. </web-app>
```

**Step 8:**

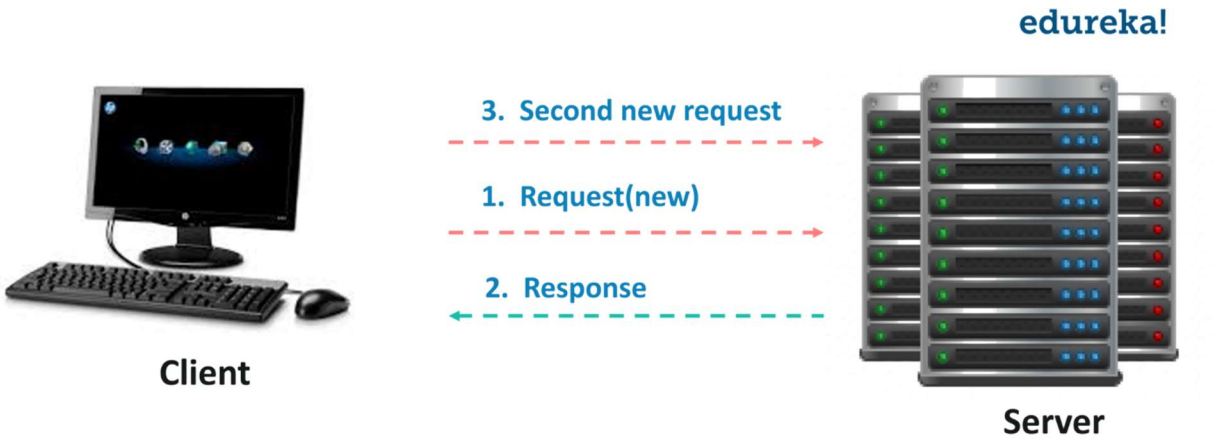
After all this, you can run the program by starting the server. You will get the desired output on the browser.

## Session Tracking

**Session** simply means a particular interval of time. And **Session Tracking** is a way to maintain state (data) of a user. It is also known as **Session Management** in servlet.

We know that the HTTP Protocol is stateless, so we need to maintain state using session tracking techniques. Each time user requests to the server, the server treats the request as the new request. So, we need to maintain the state of a user to recognize a particular user.

You can see in the figure when you send a request it is considered as a new request.




In order to recognize the particular user, we need session tracking. So, this was all about Servlets.

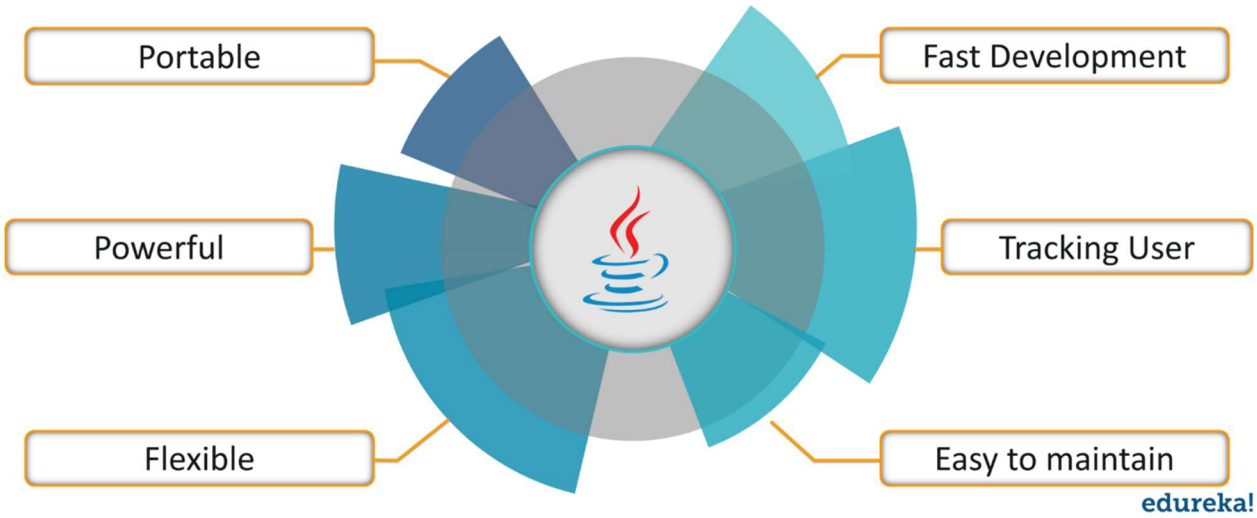
## JSP (Java Server Pages)

**JSP (Java Server Pages)** is a technology that is used to create web application just like Servlet technology. It is an extension to Servlet – as it provides more functionality than servlet such as expression language, JSTL etc.

A JSP page consists of HTML tags and JSP tags.

The JSP pages are easier to maintain than Servlet because we can separate designing and development. It provides some additional features such as Expression Language, Custom Tags etc.

### Features of JSP




1. **Portable**
   JSP tags will process and execute by the server-side web container, so that these are browser independent and J2EE server independent.
2. **Powerful**
   JSP consists of bytecode so that all Java features are applicable in case of JSP like robust, dynamic, secure, platform independent.
3. **Flexible**
   It allows to define custom tags so that the developer can fill conferrable to use any kind, framework-based markup tags in JSP.
4. **Fast Development**
   If JSP pages is modified, we don't need to recompile and redeploy the project. The Servlet code needs to be updated and recompiled if we have to change the look and feel of the application.
5. **Tracking the User**
   JSP allows us to track the selections made by the user during user interaction with the website by maintaining the information in the session or cookies.
6. **Easy**
   JSP is easy to learn, easy to understand and easy to develop. JSPs are more convenient to write than Servlets because they allow you to embed Java code directly into your HTML pages.

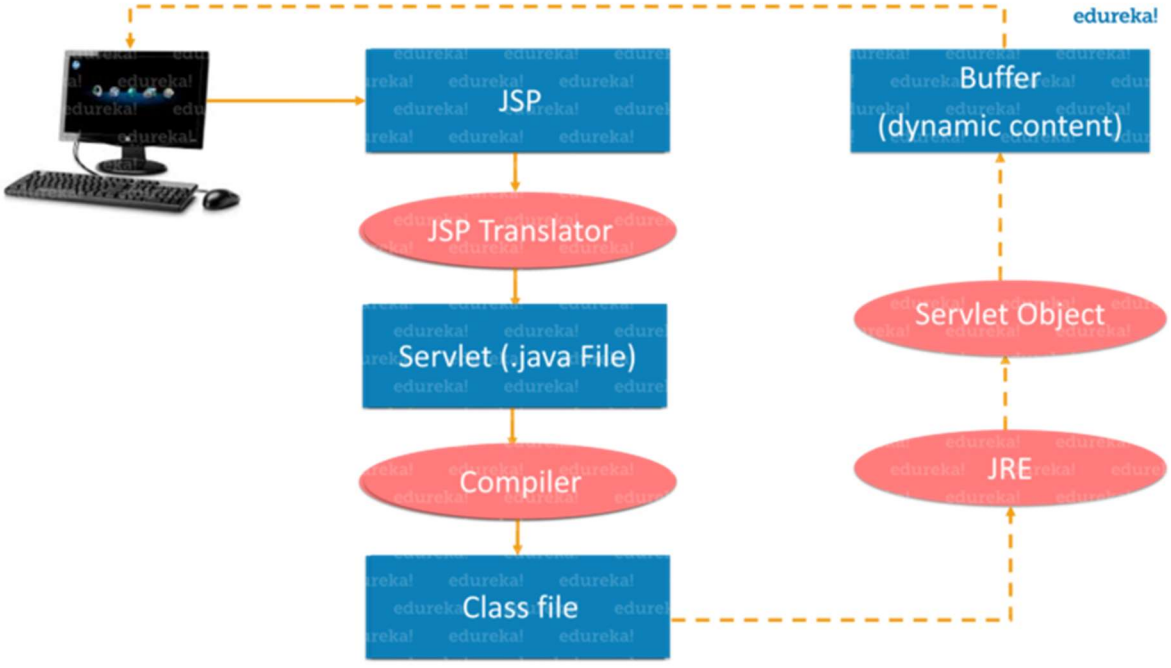
## Difference Between JSP and Servlets

| JSP | Servlets |
|---|---|
| Extension to Servlet | Not an extension to servlet |
| Easy to Maintain | Bit complicated |
| No need to recompile or redeploy | The code needs to be recompiled |
| Less code than a servlet | More code compared to JSP |

## Life Cycle of JSP

The JSP pages follow these phases:

1. Translation of JSP page
2. Compilation of JSP page
3. Classloading (the classloader loads class file)
4. Instantiation (object of the Generated Servlet is created)
5. Initialization (the container invokes jspInit ())
6. Request processing (the container invokes _jspService ())
7. Destroy (the container invokes jspDestroy ())




As depicted in the above diagram, a JSP page is translated into Servlet by the help of JSP translator. And then, JSP translator is a part of the web server which is responsible for translating the JSP page into Servlet. After that, Servlet page is compiled by the compiler and gets converted into the class file. Moreover, all the processes that happen in Servlet are performed on JSP later, like initialization, committing response to the browser and destroy.

## JSP Scripting Elements

The scripting elements provide the ability to insert Java code inside the JSP. There are 3 types of scripting elements:

- **Scriptlet tag** – A scriptlet tag is used to execute Java Source code in JSP.
  Syntax:
  <% java source code %>

- **Expression tag** – The code placed within JSP expression tag is written to the output stream of the response. So, you need not write out.print () to write data. It is mainly used to print the values of variable or method.
  Syntax:
  <%= statement %>

- **Declaring tag** – The JSP declaration tag is used to declare fields and methods. The code written inside the JSP declaration tag is placed outside the service () method of an auto-generated servlet. So, it doesn't get memory at each request.

  Syntax:

  <%! Field or method declaration %>

## Java Collection Framework

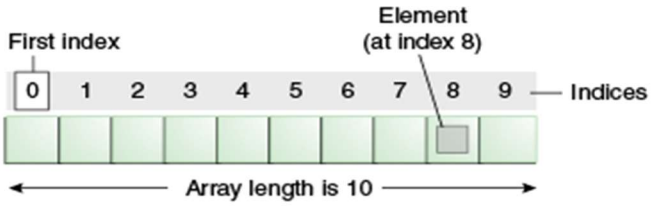
### Java Arrays

Array is the collection of homogeneous or similar data-types which have contiguous memory location.

An array is a data structure where we store similar elements.

The length of an array is assigned when the array is created and after creation, its length is fixed.

Example:




### Feature of Array:

- A java array can be declared like other variables with [] after the data type.
- The variables in the array are ordered and each have an index beginning from 0.
- In java, Arrays are objects, and thus they occupy memory in 'Heap area'.
- The direct superclass of an array type is object.
- They are always created at runtime.
- The length of an array can be finding by using member 'length'. This is the difference from C/C++ where we find length using sizeof.
- The elements of array are stored in consecutive or contiguous memory location.

### Advantages:

- Code optimization: it makes the code optimized, we can retrieve or sort the data efficiently.
- Array are used to store multiple data items of same type by using only single name.
- Random access: we can access any element randomly by using indexes provided by arrays.
- Arrays can be used to implement other data structures like linked list, stacks, queues, trees, graph etc.
- Primitive type to wrapper classes object conversion will not happen so it is fast.

### Disadvantages:

- **Fixed size**: we need to mention the size of the array, thus they have fixed size. When array is created, size cannot be changed.
- **Memory wastage**: there is a lot of chance of memory wastage. Suppose we create an array of length 100 but only 10 elements are inserted, then remaining 90 blocks are empty and thus memory wasted.
- **Strongly typed**: array stores only similar data type, thus strongly typed.
- **Reduced performance**: the elements of array are stored in consecutive memory locations, thus to delete an element in an array we need to traverse throughout the array so this will reduce performance.
- **No methods**: arrays do not have add or remove methods.

Realtime Examples:

- ✓ A quite trivial example would be your phone contacts. See the software will simply place your contacts in an array, use sorting to arrange them. Now each contact will have a unique index (acc. to its position), and there's it you can keep adding contacts, values will be updated and it will just be added to a single array. Do we need any variables? No just one array and we simply manipulate its positions.
- ✓ Another example would be arrangement of Leaderboards in games. Just a simple array that stores all the scores and sorts them in a descending manner.

Type of array:

There are two types of arrays,

- ✓ Single Dimensional Array
- ✓ Multi-Dimensional Array

Single Dimensional Array

Declaration of an array:

Different ways of declaration of arrays are,

```
int[] a; or int []a; or int []a or int a[];
```

- ✓ Most preferred array declaration is 'int[ ] a;', because here 'a' is one dimensional int array, thus name is clearly separated with type.
- ✓ We cannot provide size at the time of array declaration i.e. 'int[3] a' or 'int a[3];' statement is incorrect.
- ✓ Note that, there is difference between below two statements:

```
int [] a, b; // here 'a' and 'b' both are arrays.
int a[],b; // here 'a' is array and 'b' is simple int type variable, not an array.
```

Creation of an array:

- ✓ We can create an array after declaration as follows:

```
int[] a; //array declaration
a = new int[3]; //array creation
```

- ✓ It is compulsory to declare the size of an array at the time of creation.
- ✓ We can declare and create array within a single line as follows:

```
int[] a = new int[3];
```

- ✓ If we declare size of an array as '0' i.e. 'int[] a = new int[0];', then program will successfully compile and execute.
- ✓ If we declare size of an array as negative i.e. 'int[ ] a = new int[-3];', then program will compile successfully but when we run the program, it will through 'NegativeArraySizeException' exception.

Initialization of an array:

- ✓ We can create an array after declaration as follows:

```
int[] a = new int[3]; //array declaration and creation
//array initialization at 0 index position
a[0] = 10;
b[1] = 20;
c[2] = 30;
```

- ✓ If we initialize array at index position 3 or more i.e. 'a[3] = 40;', it will throw 'ArrayIndexOutOfBoundException' exception, so we cannot initialize an array more then its size.
- ✓ We can declare, create and initialize array within a single line as follows:

```
1. int[] a = {10, 20, 30};
2. int[] a = new int[]{10, 20, 30};
```

Retrieve Element from an array:

An array given i.e. 'int[] a = {10, 20, 30};', now we can print elements of an array by two ways, which are as follows:

```
Way 1: (using for loop)
for(int i=0; i < a.length; i++)
{
    System.out.println(a[i]);
}
```

```
Way 2: (using for-each loop)
for(int i:a)
{
   System.out.println(i);
}
```

Coding example:

```
//Java program to illustrate how to declare, instantiate, initialize and traversing the java array
class Testarray
{
            public static void main(String args[])
            {
```

```
                        //Initialization of array
                        int a[] = {10,20,30,40};
                        //printing array
                        for(int i = 0; i < a.length; i++)
                        {
                                System.out.println(a[i]);
                        }
        }
}
10 20 30 40
```

The Java Virtual Machine (JVM) throws an ArrayIndexOutOfBoundsException if length of the array is negative, equal to the array size or greater than the array size while traversing the array.

```
// ArrayIndexOutOfBoundsException example
public class TestArrayException
{
        public static void main(String args[])
        {
                int arr[] = {50,60,70,80};
                for(int i = 0; i <= arr.length; i++)
                {
                        System.out.println(arr[i]);
                }
        }
}
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 4
        at TestArrayException.main(TestArrayException.java:5)
50
60
70
80
```

## Need of Collection

- ✓ An array is an indexed collection of fixed number of homogeneous data elements.
- ✓ The main advantage of Arrays is we can represent multiple values with a single variable. So, that reusability of the code will be improved.
- ✓ Limitation of object type arrays:
    - Arrays are fixed in size i.e. once we created an array with some size there is no chance of increasing or decreasing its size based on our requirement. Hence to use arrays compulsory we should know the size in advance which may not possible always.
- ✓ Arrays can hold homogeneous data elements.

Example:

```
Student [] s = new Student[10000];
S[0] = new Student; (correct)
S[1] = new Customer(); (wrong)
```

But we can resolve this problem by using object Arrays

```
Object [] O = new Object [10000];
O [0] = new Student ();
O [1] = new Customer ();
```

Arrays concept is not implemented based on some standard data structure hence readymade method support is not available for every requirement we have to write the code explicitly, which is complexity of programming.

To overcome the above limitation of arrays we should go for collections,

- Collections are growable in nature, i.e, based on our requirement we can increase or decrease the size.
- Collection can hold both homogeneous and heterogeneous elements.
- Every collection class is implemented based on some standard data structure. Hence ready method support is available for every requirement. Being a programmer, we have to use this method and this method and we are not responsible to provide implementation.

## Difference between Arrays and Collections

| Array | Collections |
|---|---|
| Arrays are fixed in size. | Collections are growable in nature i.e. based on our requirement we can increase or decrease the size |
| W.r.t memory arrays are not recommended to use. | w.r.t to memory collections are recommended to use. |
| Arrays can hold only homogeneous datatype elements. | Collection can hold both homogeneous and heterogeneous elements. |
| w.r.t performance arrays are recommended to use. | w.r.t performance collections are not recommended to use. |
| There is no underlying data structure for arrays and hence readymade method support is not available. | Every collection class is implemented based on some standard data structure. Hence readymade method support is available for every requirement. |
| Array can hold both primitives and object types. | Collections can hold only objects but not primitives. |

## What is collection?

If we want to represent a group of individual objects as a single entity then we should go for collection.

Java collection is termed as container in C++.

Collection framework is termed as STL (standard template library) in C++.

## What is collection framework?

It defines several classes and interfaces which can be used a group of objects as single entity.

## 9 – key Interfaces of Collection Framework

- ✓ Collection
- ✓ List
- ✓ Set
- ✓ SortedSet
- ✓ NavigableSet
- ✓ Queue
- ✓ Map
- ✓ SortedMap
- ✓ NavigableMap

## Collection

If we want to represent a group of individual objects as a single entity then we should go for collection.

Collection interface defines the most common methods which are applicable for any collection object.

In general collection interface is considered as root interface of collection framework.

There is no concrete class which implements collection interface directly.

### Difference between Collection & Collections

Collection is an interface which can be used to represent a group of individual objects as a single entity.

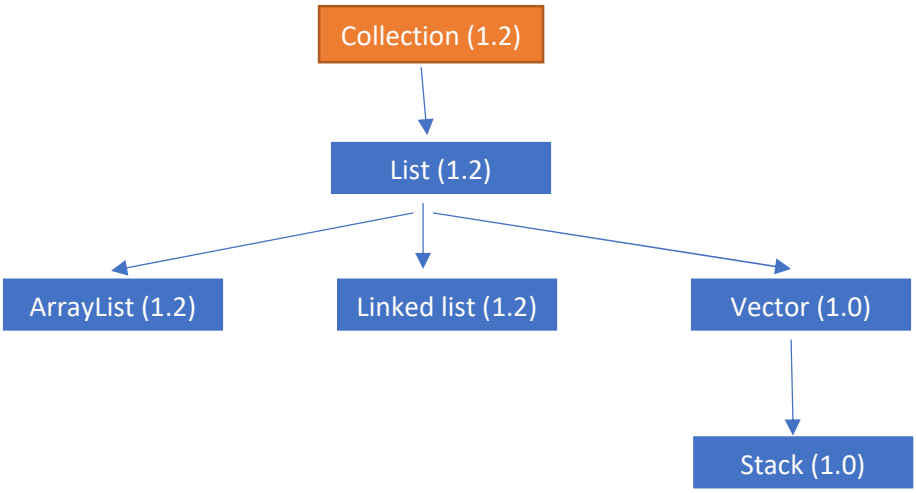
Collections is a utility class present in java.util. package to define several utility methods (like sorting, searching…) for collection objects.

## List

List is child interface of collection.

If we want to represent a group individual objects as a single entity where duplicates are allowed and insertion order preserved then we should go for list.

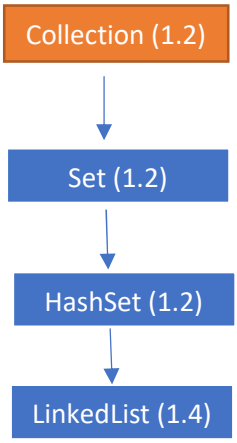
Vector and stack classes are re-engineer in 1.2 v to implement list interface.




## Set

It is the child interface of collection.

If we want to represent a group of individual objects as a single entity where duplicates are not allowed and insertion order not preserved then we should go for set.



## Difference between List & Set

| List | Set |
|---|---|
| Duplicates are allowed. | Duplicates are not allowed. |
| Insertion order preserved. | Insertion order not preserved. |

## SortedSet

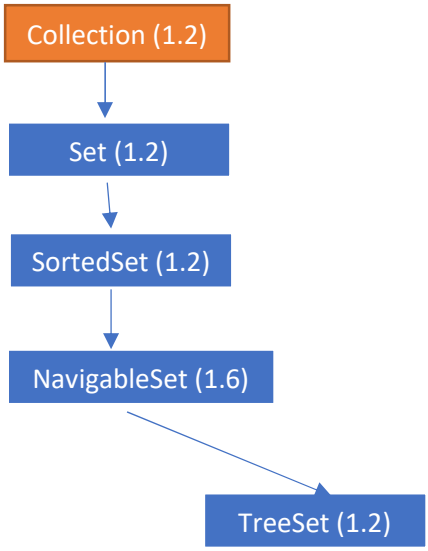
It is the child interface of set.

If we want to represent a group of individual objects as a single entity where duplicates are not allowed but all objects should be inserted according to some sorting order then we should go for SortedSet.

## NavigableSet

It is the child interface of SortedSet.

It defines several methods for navigation purposes.

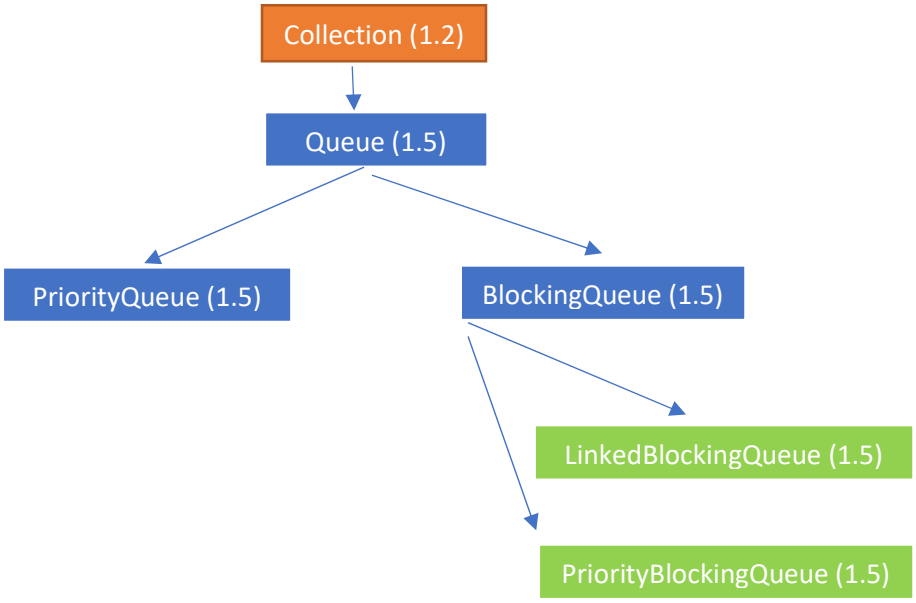



## Queue

It is the child class of collection.

If we want to represent a group of individual objects prior to processing then we should go for Queue.

Example: Before sending a mail all mail ids we have to store somewhere and in which order we saved in the same order mails should be delivered (First in First out) for this requirement Queue concept is the best choice.




Note

All the above interfaces (Collection, List, Set, SortedSet, NavigableSet and Queue) meant for representing a group of individual objects.

If we want to represent a group of objects as key value pairs then we should go for Map Interface.

Map

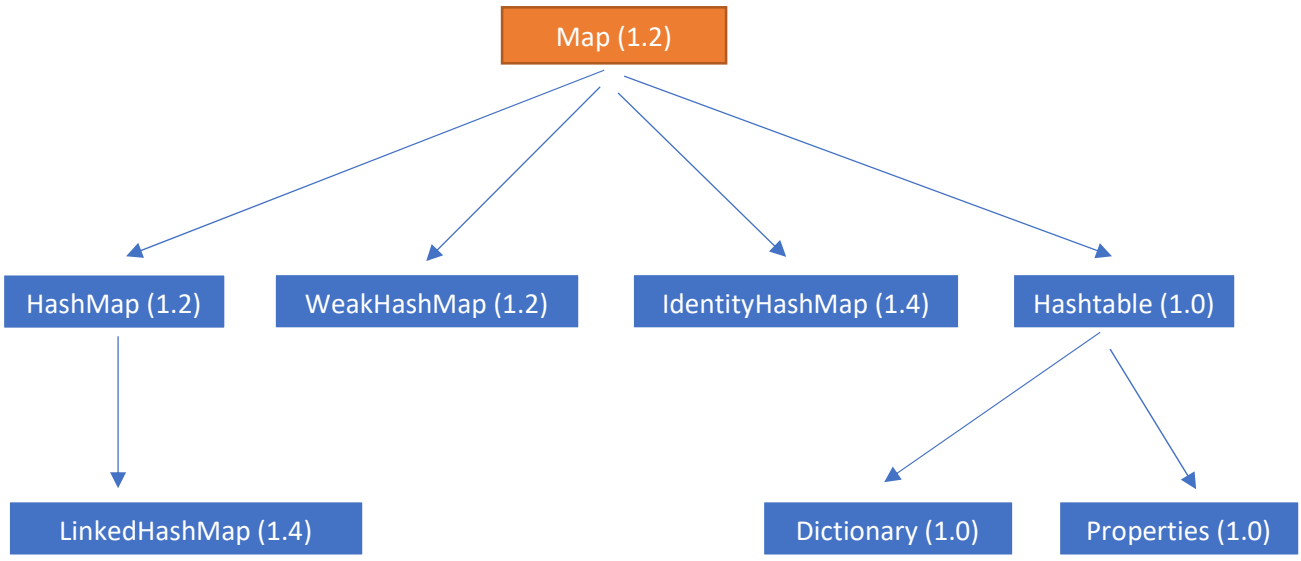
Map is not the child interface of collection.

If we want to represent a group of individual objects as key value pairs then we should go for map.

Example:

| Roll No | Name |
|---|---|
| 101 | Soumya |
| 102 | Ravi |
| 103 | Ashirbad |

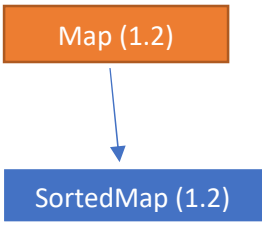
Both key and value are objects, duplicate keys are not allowed but values can be duplicated.




## SortedMap

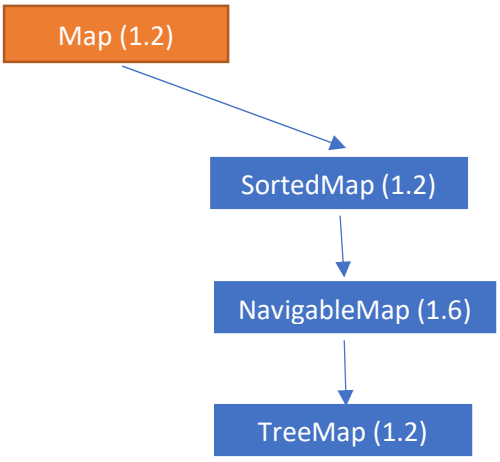
It is the child interface of map.

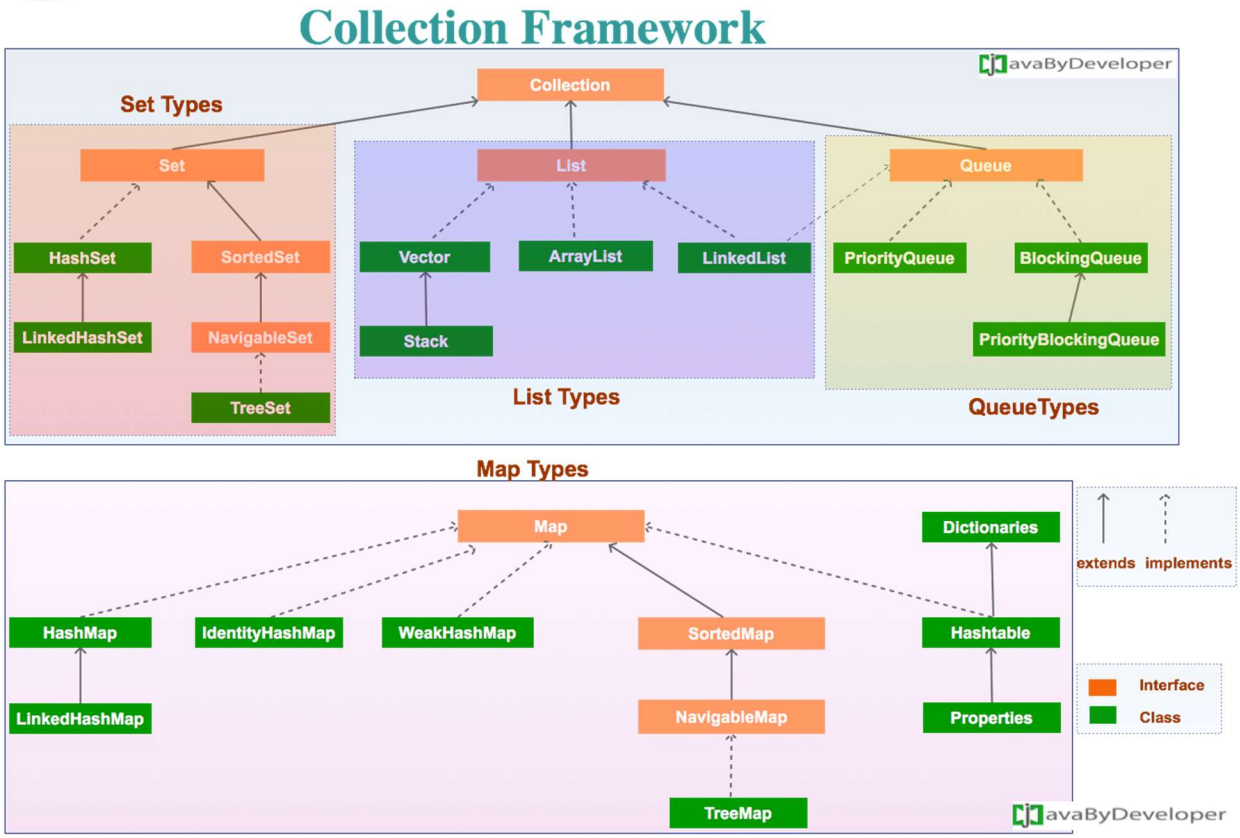
If we want to represent a group of key value pairs according to some sorting order of keys then we should go for SortedMap.




## NavigableMap

It is the child interface of sorted map; it defines several utility methods for navigation purpose.






## Collection Interface

If we want to represent a group of individual objects as a single entity then we should go for collection.

In general collection interface is considered as root interface of collection framework.

Collection interface defines the most common methods which are applicable for any collection object.

Important methods of collection interface

- Boolean add (Object o)
- Boolean addAll (Collection c)
- Boolean remove (Object o)
- Boolean removeAll (Collection c)
- Boolean retainAll (Collection c)
- Void clear ()
- Boolean contains (Object o)
- Boolean containsAll (Collection c)
- Boolean isEmpty ()
- Int size ()
- Object [] toArray ()
- Iterator iterator ()

```
boolean add (Object o)
boolean addAll (Collection c)
boolean remove (Object o)
boolean removeAll (Collection c)
boolean retainAll (Collection c)
// To remove all objects, expect those present in c
void clear ()
boolean contains (Object o)
boolean containsAll (Collection c)
boolean isEmpty ()
int size ();
Object [] toArray ();
Iterator iterator ()
```

Note:

Collection interface doesn't contain any method to retrieve objects there is no concrete class which implements collection class directly.

List Interface

- ✓ It is the child interface of collection.
- ✓ If we want to represent a group of individual objects as a single entity where duplicates are allowed and insertion order must be preserved then we should go for list.
- ✓ We can differentiate duplicates by using index.
- ✓ We can preserve insertion order by using index, hence index play very important role in list interface.
- ✓ List interface specific methods

```
// List Interface methods
void add (int index, Object o)

boolean addAll (int index, Collection c)

Object get (int index)

Object remove (int index)

//to replace the element, present at specified index with provided Object and returns old objects
object set (int index, Object new)

//returns index of first occurrence of 'o'
int indexOf (Object o)

int lastIndexOf (Object o)

ListIterator listIterator ();
```

## ArrayList

The underlined data structure is resizable array or growable array.

Duplicates are allowed.

Insertion order is preserved.

Heterogeneous objects are allowed (except TreeSet & TreeMap everywhere heterogenous objects are allowed).

Null insertion is possible.

ArrayList Constructors

```
1. ArrayList al = new ArrayList ()
// Creates an empty Array list object with default initial capacity 10. Once Array list reaches its map capacity a new Array list will be created
new capacity = (currentcapacity * 3/2) + 1

2. ArrayList al = new ArrayList (int initialCapacity);

3. ArrayList al = new ArrayList (Collection c);
```

```
//Example
import java.util.*;
class ArrayListDemo
{
	public static void main (String args[])
	{
		ArrayList l = new ArrayList();
		l.add("A");
		l.add(10);
		l.add("A");
		l.add(null);
		System.out.println(l); ///[A,10,A,null]
		l.remove(2);
		System.out.println(l); ///[A,10,null]
		l.add("2","m");
		l.add("N");
		System.out.println(l); ///[A,10,M,null,N]
	}
}
```

Usually, we can use collections to hold and transfer objects from one place to another place, to provide support for this requirement every collection already implements serializable and cloneable interface.

ArrayList and Vector classes implements RandomAccess interface so that we can access any Random element with the same speed.

Hence, if our frequent operation is retrieval operation then ArrayList is the best choice.

**RandomAccess**

- Present in java. util package.
- It doesn't contain any methods and it is a Marker interface.

```
//ArrayList
ArrayList l1 = new ArrayList ();
LinkedList l2 = new LinkedList ();
System.out.println (l1 instanceOf Serializable); //true
System.out.println (l2 instanceOf Cloneable); //true
System.out.println (l1 instanceOf RandomAccess); //true
System.out.println (l2 instanceOf RandomAccess); //false
```

ArrayList is best choice if our frequent operation is retrieval operation (because ArrayList implements RandomAccess interfaces).

ArrayList is the worst choice if our frequent operation is insertion or deletion in the middle (Because several shift operations are required).

Difference between ArrayList & Vector

| ArrayList | Vector |
|---|---|
| Every method present in ArrayList is non-synchronize. | Every method present in LinkedList is synchronize. |
| At a time, multiple threads are allowed to operate on ArrayList Object and hence ArrayList is not thread safe. | At a time only one thread is allowed to operate on vector object is thread safe. |
| Threads are not required to wait to operate on ArrayList, hence relatively performance is high. | Threads are required to wait to operator on vector object and hence relatively performance is low. |
| Introduced in 1.2 version and it is non legacy class. | Introduced in 1.0 version and it is a legacy class. |

## How to get synchronized version of ArrayList Object?

By default, ArrayList is object is non-synchronized but we can get synchronized version of ArrayList by using collection class synchronizedList () method.

```
public static List synchronizedList (List l)

//Non-synchronized
ArrayList l1 = new ArrayList ();
//Synchronized
List l = Collection. SynchronizedList (l1);

Similarly, we can get Synchronized version of Set, Map objects by using the following methods of collections class.

Public static Set synchronizedSet (Set s);
Public static Set synchronizedMap (Map m);
```

## LinkedList

- ✓ The underlying data structure is Double Linked List.
- ✓ Insertion order is preserved.
- ✓ Duplicates are allowed.
- ✓ Heterogeneous objects are allowed.
- ✓ Null insertion is possible.
- ✓ LinkedList implements Serializable and Cloneable interfaces but not RandomAccess interface.
- ✓ LinkedList is the best choice if our frequent operation is insertion or deletion in the middle.
- ✓ LinkedList is the worst choice if our frequent operation is retrieval operation.
- ✓ Usually, we can use LinkedList to implement stacks and queues to provide support for this requirement LinkedList class defines following specific methods,

```
Void addFirst ();
Void addLast ();
Object getFirst ();
Object getLast ();
Object removeFirst ();
Object removeLast ();
```

```
//LinkedList Constructor
1. LinkedList l1 = new LinkedList ();
//creates an empty LinkedList Objects
2. LinkedList l1 = new LinkedList (Collection c);
//creates an equivalent LinkedList objects for given collection
```

```
//LinkedList Demo program
import java.util.*;
public class LinkedListDemo
{
        public static void main ()
        {
                LinkedList l1 = new LinkedList ();
                l1.add ("durga");
                l1.add (30);
                l1.add (null);
                l1.add ("durga");
                l1.set (0, "software");
                l1.add (0, "venky");
                l1.addFirst ("ccc");
                System.out.println (l);
        }
}

output:
ccc venky software 30 null
```

## Difference between ArrayList & LinkedList

| ArrayList | LinkedList |
|---|---|
| It is the best choice if our frequent operation is retrieval. | It is the best choice if our frequent operation is insertion and deletion at middle. |
| ArrayList is the worst choice if our frequent operation is insertion or deletion. | LinkedList is the worst choice if our frequent operation is retrieval operation. |
| Underlying data structure for ArrayList is resizable or growable array. | Underlying data structure is double linked list. |
| ArrayList implements RandomAccess interface. | LinkedList doesn't implement RandomAccess interface. |

## Vector

- ✓ The underlying data structure for the vector is resizable array or growable array.
- ✓ Duplicate objects are allowed.
- ✓ Insertion order is preserved.
- ✓ Null insertion is possible.
- ✓ Heterogeneous objects are allowed.
- ✓ Vector class implemented serializable, cloneable and RandomAccess interface.
- ✓ Most of the methods present in vector are synchronized. Hence vector object is thread safe.
- ✓ Best choice if the frequent operation is retrieval.

Vector specific methods

```
for adding objects:
add (Object o)        [from Collection - list (I)]
add (int index, Object o) [from list]
addElement (Object o)    [from vector]

for removing objects:
remove (Object o)        [from Collection]
removeElement (Object o)   [form Vector]
remove (int index)        [from List]
removeElementAt (int index) [from collection]
clear ()              [from Collection]
removeAllElements ()      [from Vector]

for Accessing Elements:
Object get (int index)     [from Collection]
Object elementAt (int index) [from vector]
Object firstElement ()     [from vector]
Object lastElement ()      [from vector]

Other Methods
int size ();
int capacity ();
Enumeration elements ();
```

Constructors of Vector class

```
1. Vector v = new Vector ();
 - creates an empty vector object with default initial capacity 10, Once vector reaches its max capacity a
new vector Object will be created with
new capacity = 2 * current capacity

2. Vector v = new Vector (int initialCapacity);
 - creates an empty Vector object with specified initial capacity

3. Vector v = new Vector (int initialCapacity, int incrementalCapacity);

4. Vector v = new Vector (Collection c);
 - creates an equivalent Vector object for the given collection
```

```
//Demo program for vector
import java.util.*;
class VectorDemo
{
	public static void main (String args[])
	{
		Vector v = new Vector ();
		System.out.println (v.capacity ()); ///[10]
		for (int i = 0; i < 10; i++)
		{
			v.addElement (i);
		}
		System.out.println (v.capacity ()); ///[10]
		v.addElement ("A");
		System.out.println (v.capacity ()); ///[20]
		System.out.println (v);
	}
}
```

## Stack

It is a child class of Vector.

It is specially designed class for Last in First out order (LIFO).

Constructor of stack

```
Stack s = new Stack ();
```

## Methods in Stack

```
//Methods in Stack
1. Object push (Object obj);
 - for inserting an object to the stack
2. Object pop ();
 - to removes and returns top of the stack
3. Object peak ();
 - to returns the top of the stack without removal of object
4. int search (Object obj);
 - if the specified object is available it returns its offset from top of the stack.
 - if the object is not available then it returns -1.
5. Object pop ();
 - for inserting an object to the stack
```

Example

```
//Demo program for stack
import java.util.*;
class StackDemo
{
	public static void main (String args[])
	{
		Stack s = new Stack ();
		s.push ("A");
		s.push ("B");
		s.push ("c");
		System.out.println (s); ///[A,B,C]
		System.out.println (s.search ("A")); ///[3]
		System.out.println (s.search ("Z")); ///[-1]
	}
}
```

## Three cursors of Java

If we want to retrieve Objects one by one from the collection, then we should go for cursors.

There are three types of cursors are available in java.

- Enumeration
- Iterator
- ListIterator

## Enumeration

- ✓ Introduced in 1.0 version (for legacy).
- ✓ We can use Enumeration to get objects one by one from old collection objects (legacy collections).
- ✓ We can create Enumeration object by using elements () method of vector class.

```
Public Enumeration elements ();
Example:
Enumeration e = v. elements ();
```

## Method of Enumeration

```
Enumeration defines the following two methods,
1. public boolean hasMoreElements ();
2. public Object nextElements ();
```

Example

```
//Demo program for Enumeration
import java.util.*;
class EnumerationDemo
{
        public static void main (String args[])
        {
                Vector v = new Vector ();
                for (int i = 0; i <= 10; i++)
                {
                        v.addElements(i);
                }
                System.out.println (v); ///[0,1,2,3,.....,10]
                Enumeration e = v.elements ();
                while (e.hasMoreElements ())
                {
                        Integer i = (Integer)e.nextElement ();
                        if ((i%2) == 0)
                        System.out.println (i); ///[0 2 4 6 8 10]
                }
                System.out.println (v); ///[0,1,2,3,4,...,10]
        }
}
```

Limitations of Enumeration

- ✓ Enumeration concept is applicable only for legacy classes and hence it is not a universal cursor.
- ✓ By using Enumeration, we can get only read access and we can't perform remove operation.
- ✓ To overcome above limitation of Enumeration we should go for Iterator.

Iterator

- ✓ We can apply Iterator concept for any collection object hence it is universal cursor.
- ✓ By using Iterator, we can perform both read and remove operations.
- ✓ We can create Iterator object by using iterator () method of collection interface.

```
Public Iterator iterator ();
Example:
Iterator itr = C.iterator ();
//where C is any collection object
```

Methods in Iterator

```
1. public boolean hasNext ()
2. public Object next ()
3. public void remove ()
```

Example

```
//Demo program for Iterator
import java.util.*;
class iteratorDemo
{
	public static void main (String args[])
	{
		ArrayList l = new ArrayList ();
		for (int i = 0; i < 10; i++)
		{
			l.add (i);
		}
		System.out.println (l); ///[0,1,2,3,...,10]
		Iterator itr = l.iterator ();
		while (itr.hasNext ())
		{
			Integer n = (Integer) itr.next ();
			if (n%2 == 0)
			System.out.println (n); ///[0 2 4 6 8]
		}
		System.out.println (l); ///[0,1,2,3,...,10]
	}
}
```

Limitations of Iterator

- ✓ By using Enumeration and Iterator we can move only towards forward direction and we can't move to the backward direction, and hence these are single direction cursors.
- ✓ By using Iterator, we can perform only read and remove operations and we can't perform replacement of new Objects.
- ✓ To overcome above limitation of Iterator we should go for ListIterator.

ListIterator

- ✓ By using ListIterator we can move either to the forward direction or to the backward direction, and hence ListIterator is bidirectional cursor.
- ✓ By using ListIterator we can perform replacement and addition of new objects in addition of new objects in addition to read and remove operations.
- ✓ We can create ListIterator object by using listIterator () method of List Interface.

```
Public ListIterator listIterator ()
Example:
ListIterator itr = l.listIterator ();
//where l is any list operator
```

Methods in ListIterator

ListIterator is the child interface of Iterator and hence all methods of Iterator by default available to ListIterator.

ListIterator Interface defines the following 9 methods,

```
//Forward direction
1. public boolean hasNext ()
2. public void next ()
3. public int nextIndex ()

//Backward direction
1. public boolean hasPrevious ()
2. public void previous ()
3. public int previousIndex ()

//Other capability methods
1. public void remove ()
2. public void set (Object new)
3. public void add (Object new)
//Demo program for ListIterator
import java.util.*;
class ListIteratorDemo
{
	public static void main (String args[])
	{
		LinkedList l = new LinkedList ();
		l.add ("balakrishna");
		l.add ("chiru");
		l.add ("venky");
		l.add ("nag");
		System.out.println (l); ///[balakrishna, venky, chiru, nag]
		ListIterator ltr = l.listIterator ();
		while (ltr.hasNext ())
		{
			String s = (String)ltr.next ();
			if (s.equals ("venky"))
			{
				ltr.remove ();
			}
			else if (s.equals ("nag"))
			{
				ltr.add ("chaitu");
			}
			else if (s.equals ("chiru"))
			{
				ltr.set ("charan");
			}
```

```
                }
                System.out.println (l); ///[balakrishna, charan, nag, chaitu]
        }
}
```

Note: ListIterator is the most powerful cursor but its limitation is, it is applicable only for List implemented class objects and it is not a universal cursor.

| Property | Enumeration | Iterator | ListIterator |
|---|---|---|---|
| Applicable for | Only legacy classes | Any collection classes | Only list classes |
| Movement | Only forward direction (single direction) | Only forward direction (single direction) | Both forward and backward direction (bidirectional) |
| Accessibility | Only read access | Both read and remove access | Read, remove, replace and addition of new objects |
| How to get it? | By using elements () method of vector class | By using iterator () method of collection interface | By using listIterator () method of list interface |
| Methods | 2 methods<br>hasMoreElements ()<br>nextElement () | 3 methods<br>hasNext ()<br>next ()<br>remove () | 9 methods |
| Is it legacy | "yes" (1.0v) | "no" (1.2v) | "no" (1.2v) |

```
//Implementation classes of cursors
import java.util.*;
class cursorDemo
{
        public static void main (String args[])
        {
                Vector v = new Vector ();
                Enumeration e = v.element ();
                Iterator itr = v.iterator ();
                ListIterator ltr = v.listIterator ();

                System.out.println (e.getClass ().getName ());          //java.util.Vector$1
                System.out.println (itr.getClass ().getName ()); //java.util.Vector$Itr
                System.out.println (ltr.getClass ().getName ()); //java.util.Vector$ListItr
        }
}
```

## Set

Set is the child interface of collection.

If we want to represent a group of individual objects as single entity, where duplicates are not allowed and insertion order not preserved then we should go for set.

Set interface doesn't contain any new methods. So, we have to use only collection interface methods.

### HashSet

- ✓ The underlying data structure is Hashtable.
- ✓ Duplicates are not allowed. If we are trying to insert duplicates, we won't get any compile time or runtime errors. Add () method simply returns false.
- ✓ Insertion order is not preserved and all objects will be inserted based on hash-code of objects.
- ✓ Heterogeneous objects are allowed.
- ✓ 'null' insertion is possible.
- ✓ Implements serializable and cloneable interfaces but not RandomAccess.
- ✓ HashSet is the best choice, if our frequent operation is search operation.
- ✓ Constructors of HashSet

```
1. HashSet h = new HashSet ();
 - creates an empty HashSet object with default initial capacity 16 & default fill ratio 0.75

2. HashSet h = new HashSet (int initialCapacity);
 - creates an empty HashSet object with specified initial capacity & default fill ratio 0.75

3. HashSet h = new HashSet (int initialCapacity, float loadFactor);
 - creates an empty HashSet object with specified initial capacity & specified load factor (or fill ratio)

4. HashSet h = new HashSet (Collection c);
 - for inter conversion between collection objects.
```

### Load factor/Fill ratio

After loading, the how much factor, a new HashSet object will be created, that factor is called as Load Factor or Fill Ratio.

Example of HashSet

```
//Demo program for HashSet
import java.util.*;
class HashSetDemo
{
	public static void main (String args[])
	{
		HashSet h = new HashSet ();
		h.add ("B");
		h.add ("C");
		h.add ("D");
		h.add ("Z");
		h.add (null);
		h.add (10);
		System.out.println (h.add ("Z")); //false
		System.out.println (h); ///[null,D,B,C,10,Z]
	}
}
```

LinkedHashSet

- ✓ It is the child class of HashSet.
- ✓ Introduced in 1.4 version.
- ✓ It is exactly same as HashSet except the following differences.

| HashSet | LinkedHashSet |
|---|---|
| The underlying data structure is Hash table. | The underlying data structure is Hash table + Linked List. (That is hybrid data structure) |
| Insertion order is not preserved. | Insertion order is preserved. |
| Introduced in 1.2 version | Introduced in 1.4 version. |

```
//Demo program for LinkedHashSet
import java.util.*;
class HashSetDemo
{
	public static void main (String args[])
	{
		LinkedHashSet h = new LinkedHashSet ();
		h.add ("B");
		h.add ("C");
		h.add ("D");
		h.add ("Z");
		h.add (null);
		h.add (10);
```

```
                System.out.println (h.add ("Z")); //false
                System.out.println (h); ///[B,C,D,Z,null,10]
        }
}
```

Note

LinkedHashSet is the best choice to develop cache-based applications, where duplicates are not allowed and insertion order must be preserved.

SortedSet

It is the child interface of set.

If we want to represent a group of individual objects according to some sorting order and duplicates are not allowed then we should go for SortedSet.

SortedSet specific methods

1. Object first () - returns first element of the SortedSet

2. Object last () - returns last element of the SortedSet

3. SortedSet headSet (Object obj) - returns the SortedSet whose elements are < obj

4. SortedSet tailSet (Object obj) - returns the SortedSet whose elements are >= obj

5. SortedSet subSet (Object obj1, Object obj2) - returns the SortedSet whose elements are >= obj1 and <obj2

6. Comparator comparator () - returns comparator object that describes underlying sorting technique. If we are using default natural sorting order then we will get null.

Example

Given {100,101,103,104,107,110,115}

1. first () ---> 100
2. last () ---> 115
3. headset (104) ---> [100,101,103]
4. tailSet (104) ---> [104,107,110,115]
5. subset (103,110) ---> [103,104,107]
6. comparator () ---> null

Note

Default natural sorting order for numbers Ascending order and String alphabetical order.

We can apply the above methods only on SortedSet implemented class objects. That is on the TreeSet object.

TreeSet

- ✓ The underlying data structure for TreeSet is Balanced Tree.
- ✓ Duplicate objects are not allowed.
- ✓ Insertion order not preserved, but all objects will be inserted according to some sorting order.
- ✓ Heterogeneous objects are not allowed. If we are trying to insert heterogeneous objects then we will get runtime exception saying ClassCastException.
- ✓ Null insertion is allowed, but only once.

TreeSet Constructors

```
1. TreeSet t = new TreeSet ();
 - creates an empty TreeSet object where elements will be inserted according to default natural sorting order.

2. TreeSet t = new TreeSet (Comparator c);
 - creates an empty TreeSet Object where elements will be inserted according to customized sorting order.

3. TreeSet t = new TreeSet (SortedSet s);

4. TreeSet t = new TreeSet (Collection c);
```

Example

```
import java.util.*;
class TreeSetDemo
{
	public static void main (String args[])
	{
		TreeSet t = new TreeSet ();
		t.add ("A");
		t.add ("a");
		t.add ("B");
		t.add ("Z");
		t.add ("L");
		t.add (new Integer (10)); //ClassCastException
		t.add (null); //NullPointerException
		System.out.println (t); //[A,B,L,Z,a]
	}
}
```

## Null Acceptance

For empty TreeSet as the first element null insertion is possible. But after inserting that null if we are trying to insert any another element, we will get NullPointerException.

For non-empty TreeSet if we are trying to insert null then we will get NullPointerException.

Example:

```
import java.util.*;
class TreeSetDemo1
{
	public static void main (String args[])
	{
		TreeSet t = new TreeSet ();
		t.add (new StringBuffer ("A"));
		t.add (new StringBuffer ("Z"));
		t.add (new StringBuffer ("L"));
		t.add (new StringBuffer ("B"));
		System.out.println (t); //ClassCastException
	}
}
```

## Note

- ✓ If we are depending on default natural sorting order then objects should be homogeneous and comparable. Otherwise, we will get runtime exception saying ClassCastException.
- ✓ An object is said to be comparable if and only if the corresponding class implements java.lang. Comparable interface.
- ✓ String class and all wrapper classes already implements comparable interface. But, StringBuffer doesn't implements comparable interface.

## Comparable Interface

This interface present in java. lang package it contains only one method CompareTo ().

```
Public int CompareTo (Object obj)
```

```
Obj1.CompareTo (obj2)
--- returns -ve if obj1 has to come before obj2
--- returns +ve if obj1 has to come after obj2
--- returns 0 if obj1 & obj2 are equal
```

Example

```
class Test
{
	public static void main (String args[])
	{
		System.out.println ("A".compareTo ("Z")); //-ve
		System.out.println ("Z".compareTo ("B")); //+ve
		System.out.println ("A".compareTo ("A")); //0
		System.out.println ("A".compareTo (null)); //NullPointerException
	}
}
```

If we depending on default natural sorting order internally JVM will call CompareTo () method will inserting objects to the TreeSet. Hence the objects should be Comparable.

```
TreeSet t = new TreeSet ();
t.add ("B");
t.add ("z"); //"Z".compareTo("B"); +ve
t.add ("A"); //"A".compareTo("B"); -ve
System.out.println (t); //[A,B,Z]
```

Note

- ✓ If we are not satisfied with default natural sorting order or if the default natural sorting order is not already available then we can define our own customized sorting by using Comparator.
- ✓ Comparable meant for default sorting order whereas comparator meant for customized sorting order.

Comparable Interface

- ✓ We can use comparator to define our own sorting (Customized sorting).
- ✓ Comparator interface present in java. util package.
- ✓ It defines two methods.
  - Compare ()
  - equals ()

```
1. public int compare (Object obj1, Object obj2)
  ---> returns -ve if obj1 has to come before obj2
  ---> returns +ve if obj1 has to come after obj2
  ---> returns 0 if obj1 & obj2 are equal

2. public boolean equals ();
```

whenever we are implementing comparator interface, compulsory we should provide implementation for compare () method.

And implementing equals () method is optional, because it is already available in every java class from object class through inheritance.

Write a program to insert integer objects into the TreeSet where the sorting order is descending order:

```
import java.util.*;
class TreeSetDemo3
{
	public static void main (String args[])
	{
		TreeSet t = new TreeSet (new MyComparator ());
		t.add (10);
		t.add (0);
		t.add (15);
		t.add (20);
		t.add (20);
		System.out.println (t);
	}
}
class MyComparator implements Comparator
{
	public int compare (Object obj1, Object obj2)
	{
		Integer I1 = (Integer)obj1;
		Integer I2 = (Integer)obj2;
		if (I1 < I2)
			return +1;
		else if (I1 >I2)
			return -1;
		else
			return 0;
	}
}
output:
20 15 10 0
```

Integer Objects into TreeSet, Descending order:

```
TreeSet t = new TreeSet (new MyComparator ());
t.add (10);
t.add (0); --- +ve ---> compare (0, 10);
t.add (15); --- -ve ---> compare (15, 10);

t.add (20); --- +ve ---> compare (20, 10);
t.add (20); --- -ve ---> compare (20, 15);
```

```
t.add (20); --- +ve ---> compare (20, 10);
t.add (20); --- -ve ---> compare (20, 15);
t.add (20); --- 0 ---> compare (20, 20);

System.out.println (t); ///[20,15,10,0]
```

At line -1 if we are not passing comparator object then internally JVM will call CompareTo () method which meant for default natural sorting order (ascending).

In this case output is [0,10,15,20].

If we are passing comparator object at line 1 then internally JVM will call compare () method which is meant for customized sorting (Descending).

In this case output is [20,15,10,0]

Various possible implementations of compare () method

```
class MyComparator implements Comparator
{
        public int compare (Object obj1, Object obj2)
        {
                Integer I1 = (Integer)obj1;
                Integer I2 = (Interger)Obj2;
                //return I1 CompareTo (I2); [0,10,15,20] ascending order
                //return -I1 CompareTo (I2); [20,15,10,0] descending order
                //return I2 CompareTo (I1); [20,15,10,0] descending order
                //return -I2 CompareTo (I1); [0,10,15,20] ascending order
                //return +1 [10,0,15,20,20] Insertion order
                //return -1 [20,20,15,0,10] Reverse of Insertion order
                //return 0 [10]
                (only first element will be inserted and all the other elements are considered as
duplicates)
        }
}
```

Write a program to insert string objects into the TreeSet where sorting order is Reverse of alphabetical order:

```
import java.util.*;
class TreeSetDemo2
{
        public static void main (String args[])
        {
                TreeSet t = new TreeSet (new MyComparator ());
                t.add ("Roja");
                t.add ("Shobharani");
```

```
                t.add ("RajaKumari");
                t.add ("GangaBhavani");
                t.add ("Ramulamma");
                System.out.println (t);
        }
}
class MyComparator implements Comparator
{
        public int compare (Object obj1, Object obj2)
        {
                String s1 = obj1.toString ();
                String s2 = (String)obj2;
                //return s2.compareTo (s1);
                return -s1.compareTo (s2);
        }
}
```

Write a program to insert StringBuffer objects into the TreeSet where sorting order is alphabetical order:

```
import java.util.*;
class TreeSetDemo3
{
        public static void main (String args[])
        {
                TreeSet t = new TreeSet (new MyComparator ());
                t.add (new StringBuffer ("A"));
                t.add (new StringBuffer ("Z"));
                t.add (new StringBuffer ("K"));
                t.add (new StringBuffer ("L"));
                System.out.println (t);
        }
}
class MyComparator implements Comparator
{
        public int compare (Object obj1, Object obj2)
        {
                String s1 = obj1.toString ();
                String s2 = obj2.toString ();
                return s1.compareTo (s2);
        }
}
output:
A K L Z
```

Note

If we are defining our own sorting by comparator, the objects need not be comparable.

Write a program to insert string and StringBuffer objects into the TreeSet where sorting order is increasing length order if two objects having the same length then consider their alphabetical order:

```
import java.util.*;
class TreeSetDemo4
{
	public static void main (String args[])
	{
		TreeSet t = new TreeSet (new MyComparator ());
		t.add ("A");
		t.add (new StringBuffer ("ABC"));
		t.add (new StringBuffer ("AA"));
		t.add ("XX");
		t.add ("ABCD");
		t.add ("A");
		System.out.println (t);
	}
}
class MyComparator implements Comparator
{
	public int compare (Object obj1, Object obj2)
	{
		String s1 = obj1.toString ();
		String s2 = obj2.toString ();
		int l1 = s1.length ();
		int l2 = s2.length ();
		if (l1 < l2)
			return -1;
		else if (l1 > l2)
			return 1;
		else
			return s1.compareTo (s2);
	}
}
output:
A AA XX ABC ABCD
```

Note

- ✓ If we are depending on default natural sorting order then objects should be homogeneous and comparable otherwise, we will get runtime exception saying ClassCastException.
- ✓ But if we are defining out own sorting by comparator then objects need not be homogeneous and comparable. We can insert heterogeneous non comparable objects also.
- ✓ For predefined comparable classes like string default natural sorting order already available. If we are not satisfied with that, we can define our own sorting by comparator object.
- ✓ For predefined non comparable classes like StringBuffer, default natural sorting order is not already available. We can define required sorting by implementing comparator interface.

- ✓ For our own classes like Employee, Student, Customer, the person who is writing our own class, he is responsible to define default natural sorting order by implementing comparable interface.
- ✓ The person who is using our class, if he is not satisfied with default natural sorting order, then he can define his own sorting by using comparator.

Demo program for Customized sorting for Employee class:

```
import java.util.*;
class Employee implements Comparable
{
        String name;
        int eid;
        Employee (String name, int eid)
        {
                this.name = name;
                this.eid = eid;
        }
        public String toString ()
        {
                return name+" "+eid;
        }
        public int compareTo (Object obj)
        {
                int eid1 = this.eid;
                Employee e = (Employee)obj;
                int eid2 = e.eid;
                if (eid1 < eid2)
                {return -1;}
                else if (eid1 > eid2)
                {return 1;}
                else
                {return 0;}
        }
}
class CompCompDemo
{
        public static void main (String args[])
        {
                Employee e1 = new Employee ("nag",100);
                Employee e2 = new Employee ("ashu",200);
                Employee e3 = new Employee ("chiru",50);
                Employee e4 = new Employee ("venki",150);
                Employee e5 = new Employee ("nag",100);
                TreeSet t = new TreeSet ();
                t.add (e1);
                t.add (e2);
                t.add (e3);
                t.add (e4);
                t.add (e5);
```

```
                    System.out.println (t);
                    TreeSet t1 = new TreeSet (new MyComparator ());
                    t1.add (e1);
                    t1.add (e2);
                    t1.add (e3);
                    t1.add (e4);
                    t1.add (e5);
                    System.out.println (t1);
          }
}
class MyComparator implements Comparator
{
          public int compare (Object obj1, Object obj2)
          {
                    Employee e1 = (Employee)obj1;
                    Employee e2 = (Employee)obj2;
                    String s1 = e1.name;
                    String s2 = e2.name;
                    return s1.compareTo (s2);
          }
}
```

Comparison table of Comparable and Comparator

| Comparable | Comparator |
|---|---|
| It is meant for default natural sorting order. | It is meant for customized sorting order. |
| Present in java. lang package. | Present in java. util package. |
| This interface defines only once method compareTo (). | This interface defines two methods compare () and equals (). |
| All wrapper classes and string class implements comparable interface. | The only implemented classes of comparator are collator and RuleBasedCollator. |

Comparison table of Set (I) implemented classes

| Property | HashSet | LinkedHashSet | TreeSet |
|---|---|---|---|
| 1. Underlying Data Structure | Hashtable | Hashtable + LinkedList | Balanced Tree |
| 2. Insertion order | Not preserved | preserved | Not applicable |
| 3. Sorting order | Not applicable | Not applicable | Applicable |
| 4. Heterogeneous objects | allowed | allowed | Not allowed |

| | | | |
|---|---|---|---|
| 5. Duplicate objects | Not allowed | Not allowed | Not allowed |
| 6. Null Acceptance | Allowed (only once) | Allowed (only once) | For empty Tree Set as first element Null is allowed and in all other cases, we will get NullPointerException. |

Difference between Iterator and ListIterator

| Iterator | listIterator |
|---|---|
| Used to retrieve the objects from collection classes. | Used to retrieve the data from collection classes. |
| Introduced in 1.2 version it is not a legacy | Introduced in 1.2 version it is not a legacy |
| It is used to retrieve the data from all collection classes | It is used retrieve the data from only list type of classes like ArrayList, LinkedList, Vector, Stack. |
| It is a universal cursor because it is applicable for all collection classes. | Not a universal cursor because it is applicable for only List interface classes. |
| Get the iterator object by using iterator () method.<br>**Vector v = new vector ();**<br>**v.add (10);**<br>**v.add (20);**<br>**Enumeration e = v. iterator ();** | Get the ListIterator Object by using listIterator () method.<br>**Vector v = new Vector ();**<br>**v.add (10);**<br>**v.add (20);**<br>**Enumeration e = v. listIterator ();** |
| It contains two methods<br>hasNext (): to check the objects available or not<br>Next (): to retrieve the objects | It contains 9 methods. |
| Read & remove operations are possible. | Read, remove, add and replace operations. |
| Only forward direction | Bidirectional cursor direction |
| Interface | Interface |
| It supports both normal and generic version | It supports both normal and generic version |



## How do you handle garbage collecting?

In java, garbage collector is a special program which runs on JVM and removes objects which are not in used. JVM calls garbage collector by default or you can also invoke explicitly with

System.gc () or Runtime.gc commands

## Difference between comparable and comparator?

Comparable and Comparator both are interfaces and can be used to sort collection elements.

| Comparable | Comparator |
|---|---|
| 1) Comparable provides a single sorting sequence. In other words, we can sort the collection on the basis of a single element such as id, name, and price. | The Comparator provides multiple sorting sequences. In other words, we can sort the collection on the basis of multiple elements such as id, name, and price etc. |
| 2) Comparable affects the original class, i.e., the actual class is modified. | Comparator doesn't affect the original class, i.e., the actual class is not modified. |
| 3) Comparable provides compareTo () method to sort elements. | Comparator provides compare () method to sort elements. |
| 4) Comparable is present in java. lang package. | A Comparator is present in the java. util package. |
| 5) We can sort the list elements of Comparable type by Collections. Sort (List) method. | We can sort the list elements of Comparator type by Collections.sort (List, Comparator) method. |

## Is Java completely object oriented?

A fully object-oriented language needs to have all the 4 oops concepts.in addition to that, all predefined and user-defined types must be objects and, all the operations should be performed only by calling the methods of a class.
Though java follows all the four object-oriented concepts,

- Java has predefined primitive data types (which are not objects).
- You can access the members of a static class without creating an object of it.

## Where does newly created string gets stored?

When we create a string via the new operator, the Java compiler will create a new object and store it in the heap space reserved for the JVM.

It is stored in the heap memory.

## Concurrency in Java

Concurrency is the ability to run several programs or several parts of a program in parallel.

The backbone of Java concurrency are threads.

A thread is a lightweight process which has its own call stack, but can access shared data of other threads in the same process.

## Is it possible to run a Java program without main method?

A Java program can run without the main method. You need a static block for that, you just need to put your executable code in that static block and execute it. The sequence goes something like this:

- The Java Virtual Machine first loads your class
- Then it arranges and runs all static blocks
- Then finally stare across your code for main () and uses it

However, if you are planning to run your program directly from the command-line without using static block, and with the use of JVM, then your program must be required to have the main () method. This is because the Java Virtual Machine keeps looking for the main ().

## What is "Write once and run anywhere" feature of Java?

WORA which is abbreviated as Write Once Run Anywhere is the feature applicable to those programs which hold the capability to execute itself on any operating systems or on any machine. This terminology was given by Sun Microsystem for their programming language – Java.

According to this concept, the same code must run on any machine and hence the source code needs to be portable. So, java allows run Java bytecode on any machine irrespective of the machine or the hardware, using JVM.

The bytecode generated by the compiler is not platform-specific and hence takes help of the JVM to run a wide range of machines. So, we can call Java programs as a write once and run on any machine residing anywhere.

## What is Just-in-time (JIT) compiler?

In Java programming, a Just-in-time compiler is additionally given, as it can convert the bytecode of Java to the instruction which can be feed straight to your processor. It starts running just once your program starts and compiles your source code on the fly (hence termed as just-in-time). It is typically faster than normal compiling. It can access the runtime information dynamically which is not possible by the normal compiler.

With the use of this compiler, at any moment, the system can compile your Java bytecode to that specific system code which makes Java code portable.

## Difference between Object-Oriented programming language and Object-based programming language?

| Object-oriented Programming Language | Object-based Programming Language |
|---|---|
| All the characteristics and features of object-oriented programming are supported. | All characteristics and features of object-oriented programming, such as inheritance and polymorphism are not supported. |
| These type of programming languages don't have a built-in object. Example: C++. | These type of programming languages have built-in objects. Example: JavaScript has a window object. |
| Java is an example of object-oriented programing language which supports creating and inheriting (which is reusing of code) one class from another. | VB is another example of object-based language as you can create and use classes and objects but inheriting classes is not supported. |

## How many types of memory areas are allocated by JVM?

Java Virtual Machine is one of the most important components which does particular types of work.

1. Loading of code
2. Verification of code
3. Executing the code
4. Providing a run-time environment for the users

All these functions take different forms of the memory structure. These data structures are:

- Heap
- Stack
- Program Counter Register
- Native Method Stack

Let us discuss each of the above memory structures' characteristics:

Heap memory structure is usually implemented for allocating memory dynamically. Variables assigned with this type of memory structure can be allocated at runtime, but they have slow accessing of memory.

Stack memory structure is mostly implemented for providing static memory allocation. Programmers could make use of stack if they knew in advance how much memory needs to be allocated for the storage of data.

Program Counter Register: Programs are a set of instructions or orders feed to a computer for performing. These instructions are delivered to the processor by the program written by a human. Program counter register holds the address of the upcoming instructions to be executed.

Native methods form a stack that is primarily implemented to line up with your system calls as well as libraries scripted in different computer language.

## What is the difference between wait and sleep methods in Java?

| wait () | sleep () |
|---|---|
| wait method is one of the methods of java.lang.Object class. | sleep method is one of the methods of java.lang.Thread class. |
| Only a synchronized block or method can call the wait() method. | From any point or context in a program, you can call the sleep() method. |
| notify() and notifyAll() methods are used for waking the waiting thread. | Interrupts are used for waking the sleeping thread. |
| This method is usually implemented in a Java program for performing multi-thread-synchronization. | This method is generally applied for controlling the time of execution for a single thread. |

## What is the difference between classes and objects?

| Class | Object |
|---|---|
| A class is a blueprint from which you can create the instance, i.e., objects. | An object is the instance of the class, which helps programmers to use variables and methods from inside the class. |
| A class is used to bind data as well as methods together as a single unit. | object acts as a variable of the class. |
| Classes have logical existence. | Objects have a physical existence. |
| A class doesn't take any memory spaces when a programmer creates one. | An object takes memory when a programmer creates one. |

| | |
|---|---|
| The class has to be declared only once. | Objects can be declared several times depending |

What is the difference between method overloading and overriding?

| OVERLOADING | OVERRIDING |
|---|---|
| It is performed at compile time. | It is performed at runtime. |
| It is carried out within a class. | It is carried out with two classes having an IS-A relationship between them. |
| Parameters do not remain the same in case of overloading. | The parameter must remain the same in case of overriding. |
| You can perform overloading on static methods. | You cannot perform overriding on static methods. |
| Overloaded methods use static binding. | Overridden methods use dynamic binding. |

what is object cloning in java?

Object Cloning is one of the extraordinary features provided by Java programming which helps in creating the same copy or duplicating an object of Java class. This entire duplication of an object is done by the *clone () method*. This method is used in a situation where programmers want to create the same to the same copy of an existing object, which eventually reduces the extra task of processing.

The clone () method definition is within the Java Object class, and you have to override this method in order to use. For using this predefined method, you have to use the java.lang.CloneableInterface and implement it with that specific class whose object cloning you want to make. If you do not implement this interface, your clone method will pop up with a CloneNotSupportedException.

## What is the difference between Data abstraction and Data encapsulation in Java?

| Data Abstraction | Data Encapsulation |
|---|---|
| Data Abstraction can be described as the technique of hiding internal details of a program and exposing the functionality only. | Data Encapsulation can be described as the technique of binding up of data along with its correlate methods as a single unit. |
| Implementation hiding is done using this technique. | Information hiding is done using this technique. |
| Data abstraction can be performed using Interface, and abstract class. | Data encapsulation can be performed using the different access modifiers like protected, private, and packages. |
| Abstraction lets somebody see the What part of program purpose. | Encapsulation conceals or covers the How part of program's purpose. |

## What is the difference between Abstract class and Interface?

| Abstract Class | Interface |
|---|---|
| Only a single abstract class be get inherited by a class. | A class can inherit several interfaces. |
| Abstract class provide the complete code or simply the details which need to be overridden. | Interface provides the signature rather than the code. |
| It is comparatively faster than the interface. | It requires more execution time and hence slower than an abstract class. |
| There you can provide the access specifier to functions and properties of the abstract class. | There cannot be any access specifier to the interface, and hence everything is taken as a public. |

## What is Deadlock in java threads?

In Java, deadlock is a situation that arises in the multithreading concept. This situation may appear in cases where one of your thread is waiting for an object lock, which is acquired by another thread and the second thread waiting for object lock acquired by the first one. Since both become dependent on each other's lock release, so it forms a situation which is termed as deadlock. In such scenarios, the threads get blocked for an infinite timestamp and keep on waiting for each other. When the synchronized keyword halts the thread execution for some time, multiple threads suffer that are associated with that particular object.

### How to avoid deadlock in Java:

These are some of the guidelines using which we can avoid most of the deadlock situations.

- Avoid deadlock by breaking circular wait condition: In order to do that, you can make arrangement in the code to impose the ordering on acquisition and release of locks. If lock will be acquired in a consistent order and released in just opposite order, there would not be a situation where one thread is holding a lock which is acquired by other and vice-versa.
- Avoid Nested Locks: This is the most common reason for deadlocks, avoid locking another resource if you already hold one. Its almost impossible to get a deadlock situation if you are working only one object lock.
- Lock only what is required: You should acquire lock only on the resources you have to work on, if we are only interested in one of its fields, then we should lock only that specific field not complete object.
- Avoid waiting indefinitely: You can get a deadlock if two threads are waiting for each other to finish indefinitely using thread join. If your thread has to wait for another thread to finish, it's always best to use join with maximum time you want to wait for the thread to finish.

## What are the nested classes in Java?

Java allows programmers to define a class inside another class. This concept helps in grouping classes logically. Nested classes are those classes that are defined or created within another class. It is to be noted that the lifetime and scope of the nested class remains bounded with the scope of its enclosing class. Moreover, it has many advantages as well. Nesting of classes increases the usefulness of encapsulation. Also, the readability and maintainability of code increase with the use of nesting of classes.

Furthermore, it has full access to its members, both public as well as private members. Nested classes are also treated as members of its encircled class. It is of two types. These are:

- **static nested class**: they are declared using the keyword static
- **inner class**: classes those are not declared as static (non-static)

## What is the difference between throw and throws in Java?

| Throw | Throws |
|---|---|
| This keyword is used for explicitly throwing an exception. | This keyword is used for declaring any exception. |
| Programmers cannot disseminate checked exceptions using throw keyword. | Programmers can disseminate checked exceptions using throws keyword. |
| An instance trails the throw keyword. | A class trails the throws keyword. |
| You have to use the throw keyword inside any method. | You have to use the throws keyword with any sign of the method. |
| Multiple exceptions cannot be thrown. | Multiple exceptions can be declared using the throws. |
| Code Snippet: | Code Snippet: |
| Class Example<br>{<br>public static void main (String argu[])<br>{<br>throw new ArithmeticException("Divided by zero");<br>}<br>} | public void testExc() throws<br>SQLException , IOException |

## Why Java is a Secure language?

Java is an extremely safe language due to a variety of features like,

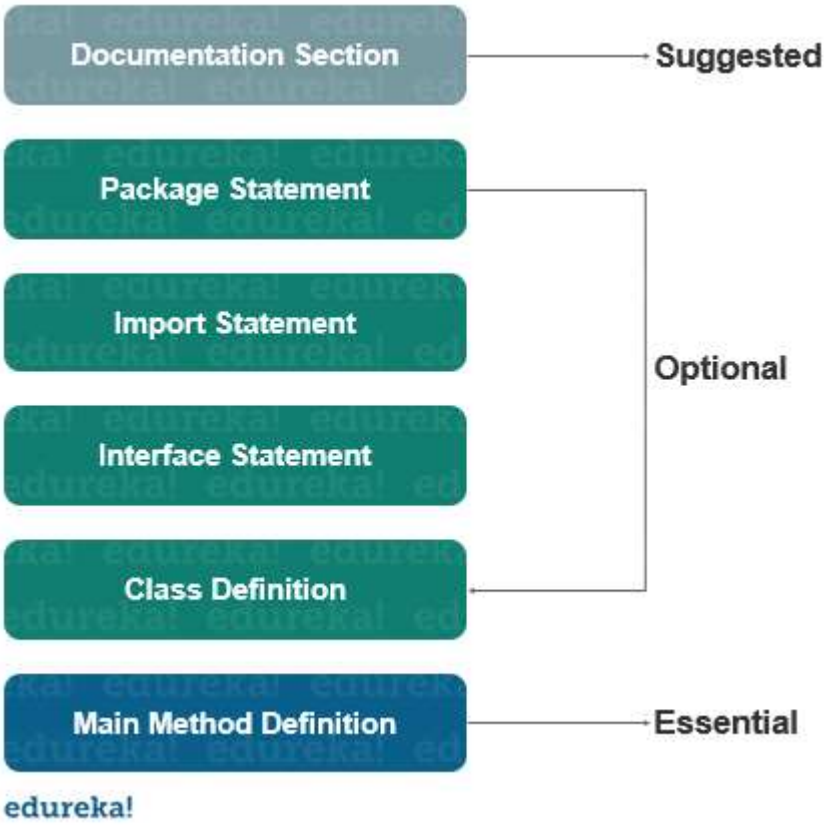
- The byte-code verification takes place before the execution; therefore, the program becomes unable to jump to a malicious or undefined instruction or to make a type error at the instruction level.
- Also, whenever a new code is being loaded, run-time security checks take place.
- Java provides library level support.

The top 10 features that make Java Secure:

1. **JVM** – JVM is an interpreter which is installed in each client machine that is updated with latest security updates by internet. When this byte codes are executed, the JVM can take care of the security.
2. **Security API's** – This API is involved in cryptographic algorithms secure communication and authentication protocols.
3. **Security Manager** – Security manager guaranties that the doubted code or some malicious code does not accomplish the goal of accessing some features of the platform and API's.
4. **Void of Pointers** – There is no concept of pointers in Java. The only disadvantage of pointer is that it can be used to refer another object for doing some unauthorized read and write operation. This puts the feature of security.
5. **Memory management**
6. **Compile time checking**
7. **Java Sandbox** – Java Sandbox is basically a restricted area in which the Java applets Run. These applets can't get system resources without a check.
8. **Exception Handling**

## What is the basic structure of a Java Program?

The structure of a Java Program is:




1. **Documentation Section** – It is used to improve the readability of the program.
2. **Package Statement** – There can be only one package statement in Java Program and it has to be at the beginning of the code before any class or interface declaration. This statement is optional.
3. **Import Statement** – we can import a specific class or classes in an import statement.
4. **Interface Section** – This section is used to specify an interface in Java. It is an optional section which is mainly used to implement multiple inheritance.
5. **Class Definition** – It defines the information about the user-defined classes in a program.
6. **Main Method Class** – The main method is from where the execution starts and follows the order specified.

What's new in versions?

**Java 8** includes the following:

1. Lambda Expressions
2. Method references
3. Default methods (Defender methods)
4. A new Stream API
5. Optional
6. A new Date/Time API
7. Nashorn, the new JavaScript engine
8. Removal of the Permanent Generation and more...

**Java 9** includes the following:

1. Factory methods for Immutable List, Set, Map
2. Private methods in interface
3. Java 9 module system
4. Process API improvements
5. Reactive Streams
6. HTTP 2 client
7. Stream API improvements
8. Diamond operator for anonymous Inner class and more...

**Java 10** features:

1. Time-Based releasing versioning
2. Local-Variable Type Interface
3. Experimental Java-Based JIT compiler
4. Root certificates
5. Heap allocation on Alternative memory devices
6. Collection API copyOf () method
7. Optional API or ElseThrow () method

What is JIT compiler?

The Just-In-Time (JIT) compiler is a part of the runtime environment. It helps in improving the performance of Java applications by compiling bytecodes to machine code at run time. The JIT compiler is enabled by default. When a method is compiled, the JVM calls the compiled code of that method directly. The JIT compiler compiles the bytecode of that method into machine code, compiling it "just in time" to run.

## What is the use of Destructor in Java?

A **destructor** is a special method that gets called automatically as soon as the life-cycle of an object is finished.

A **destructor** is called to de-allocate and free memory. The following tasks get executed when a destructor is called:

- Releasing the release locks
- Closing all the database connections or files
- Releasing all network resources
- Other Housekeeping tasks
- Recovering the heap space allocated during the life time of an object

Destructors in Java is also known as **finalizers**. The allocation and release of memory are implicitly handled by the garbage collector in Java.

**Garbage collector** is a program that runs on the Java virtual machine to recover the memory by deleting the objects which are no longer in use or have finished their life-cycle.

Garbage collection is mainly the process of making or identifying the unreachable objects and deleting them to free the memory. The different types of Garbage collectors are: **Serial Garbage Collector**, **Parallel/Throughput Garbage Collector**, **CMS Collector** and **G1 Collector**.

## What is a constant in Java and how to declare it?

**Constant** refers to '**a situation that does not change**'.

Constants in Java are used when a '**static**' value or a permanent value for a variable has to be implemented. Java doesn't directly support constants. To make any variable a constant, we must use '**static**' and '**final**' modifiers.

**Syntax:**

```
Static final datatype identifier_name = constant;

Example: static final int MIN_AGE = 18;
```

- The **static modifier** causes the variable to be available without an instance of its defining class being loaded.
- The **final modifier** makes the variable unchangeable.

The **static modifier** is finally used for **memory management**. It also allows the variable to be available without loading any instance of the class in which it is defined.

The **final modifier** means that the value of a variable cannot change. Once the value is assigned to a variable, a different value cannot be reassigned to the same variable.

Why constants?

**Constants** make your program easier to read and understand when read by others.

Using a constant also improves performance, as constants are cached by both the JVM and your application.

What is Enumeration in Java?

- An **enumeration** is nothing but a set of named constants which helps in defining its own datatypes.
- It is similar to **final** variables.
- The enumeration in Java is a data type that contains a fixed set of constants.
- An enumeration defines a **class type in Java**. By making enumerations into classes, it can have constructors, methods and instance variables.
- An enumeration is created using the **enum** keyword.

Defining Enumeration in Java

Enum declaration can be done either outside a class or inside a class. But we cannot declare Enum inside the method

1. Declaring an Enumeration outside a class

```
1. enum Directions{ // enum keyword is used instead of class keyword
2. NORTH, SOUTH, EAST, WEST;
3. }
4. public class enumDeclaration {
5. public static void main (String [] args) {
6. Directions d1 = Directions.EAST; // new keyword is not required to create a new object reference
7. System.out.println(d1);
8. }
9. }
```

2. Declaring an Enumeration inside a class

```
1. public class enumDeclaration {
2. enum Directions {
3. NORTH, SOUTH, EAST, WEST;
4. }
5. public static void main (String [] args) {
6. Directions d1 = Directions.EAST; // new keyword is not required to create a new object reference
7. System.out.println(d1);
8. }
9. }
```

The first line inside the enum type should be a list of constants. Then you can use methods, variables and constructors.

NOTE

- Enum basically improves type safety.
- It can be diversely used in switch case examples.
- Enum can be easily traversed.
- The enum has fields, constructors and methods.
- Enum basically implements many **interfaces** but, cannot extend any class because it **internally extends Enum class**.

## Values () and ValueOf () method

### Values ()

When you create an enum, the Java compiler internally adds the values () method. This method returns an array containing all the values of the enum.

```
public static enum-type [] values ()
```

### ValueOf ()

This method is used to return the enumeration constant whose value is equal to the string passed as an argument while calling this method.

```
public static enum-type valueOf (String str)
```

Basically, **Enumeration** can contain **constructor** and it is executed separately for each enum constant at the time of enum class loading.

Not just that, an enum can create **concrete** methods.

## What is the role of ClassLoader in Java?

While working in Java, we often use a large number of classes. These Java classes are not loaded all at once in the memory, instead, they are loaded when required by an application. This is where Java ClassLoader comes into picture.

**ClassLoader** in Java is called by the Java Runtime Environment to dynamically load the classes whenever required by the application in the JVM. Since ClassLoader are part of the JRE, the JVM will not have any idea about the underlying files and files systems.

The **different types** of built-in ClassLoaders in Java are:

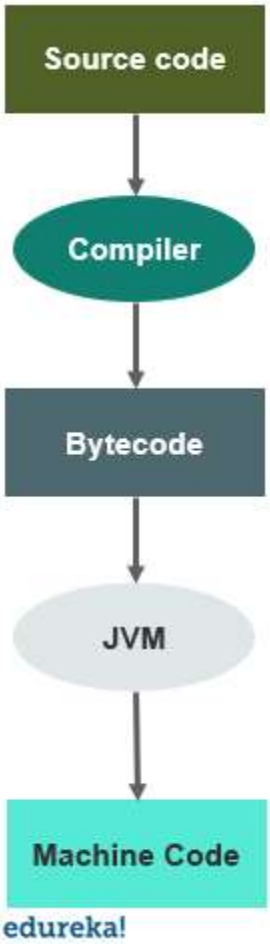
**NOTE**: The priority of Bootstrap is higher than Extension, and the priority given to the Extension ClassLoader is higher than Application ClassLoader.

1. **Extension ClassLoader**
   The Extension ClassLoader loads the extensions of the core Java classes from JDK Extension library. It is a child of the Bootstrap ClassLoader and loads the extensions from the JRE/lib/text directory.
2. **Application or System ClassLoader**
   This is a child of the Extension ClassLoader. This type of ClassLoader loads all the application level classes found in the **-cp** command line or **int** the classpath environment variable.
3. **Bootstrap ClassLoader**
   As we know that Java classes are loaded by an instance of **java.lang.ClassLoder**. But, since ClassLoaders are classes, the Bootstrap ClassLoader is responsible to load the JDK internal classes.
   Bootstrap ClassLoader is a machine code that starts the operation when JVM calls it and loads the classes from rt.jar. so, you can understand that the Bootstrap ClassLoader serves has no parent ClassLoader and is thus known as **Primordial ClassLoader**.

## What is Bytecode in Java and how it works?

A Bytecode in Java is the instruction set for JVM and acts similar to an assembler.

Bytecode in Java is the reason java is platform-independent. As soon as a Java Program is compiled bytecode is generated.




When a Java Program executed, the compiler compiles that piece of code and a Bytecode is generated for each method in that program in the form of a **.class file**.

**Bytecodes** are **non-runnable** codes that rely on the availability of an interpreter, this is where JVM comes into play. It is a **machine-level** language code that runs on the JVM.

It adds portability to Java which resonates with the saying, "**write once, read anywhere**".

## What is a Scanner class in Java?

The **Scanner** class is mainly used to **get the user input**, and it belongs to the **java.util.package**. in order to use the Scanner class, you can create an object of the class and use any of the Scanner class methods.

Example:

```
import java.util.Scanner;// Import the Scanner class

public class Example {

public static void main (String [] args) {

Scanner s = new Scanner (System.in); // Create a Scanner object

System.out.println("Enter username");

String name = s.nextLine();// Read user input

System.out.println("name is: " + name); // Output user input

}

}
```

Scanner Class Methods

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

## What is Protected in Java?

**Protected** in Java is an access modifier that helps a programmer in assigning the visibility of a class, its members, interfaces etc. When class members are declared as protected in Java, they are **accessible only within the same package** as well as to the direct subclasses of the base class.

**While using protected keyword** in Java, you must keep in mind that only class members can be declared as protected.

**Classes** and **Interfaces** can't be declared as protected in Java.

**Why?**

**For Class**: If a **class** is made protected, then it will be only visible to the classes present within the same package as well as to its subclasses. But here is the ambiguity. For other classes to extend a protected class, the parent class has to be visible. How will you extend something which is not visible in th first place? This causes ambiguity and creating a protected class is not allowed in Java.

**For Interface**: In Java the elements are generally made protected so that their implementations can be shared among others. But in the case of Interface, they have no implementation, so there is no point in sharing them. Thus, all the methods present within the interfaces must be public so that any class or structs can easily implement them.

## What is a Static keyword in Java?

**Static keyword** is mainly used for **memory management**. It can be used with variables, methods, blocks and nested classes. It is a keyword which is used to share the same variable or method of a given class.

Basically, static is used for a constant variable or a method that is same for every instance of a class. The **main method** of class is generally labeled static.

In Java programming language, static is a **non-access modifier** and can be used as **Static Block**, **Static Variable**, **Static Method** and **Static Class**.

## What is Iterator in Java and How to use it?

Java mainly support 4 different types of cursors.

1. Enumeration
2. Iterator
3. ListIterator
4. Spliterator

**Iterator** is an interface that belongs to a collection framework. It allows you to traverse the collection, accesses the data element and removes the data elements of the collection.

It is also considered as a Universal iterator as you can apply it to any collection object. By using an Iterator, you can perform both read and remove operations.

## What is comparator in Java?

Java **Comparator** interface is used **to order the objects inside a user-defined class**. This interface is available in **java.util.package** and incudes two crucial methods known as **compare (Object obj1, Object obj2)** and **equals (Objects element)**.

1. compare (Object obj1, Object obj2) - Compares the first object with another
2. equals (Object obj) – Compares current object with specified obj

| Comparator | Comparable |
|---|---|
| The Comparator is used to sort attributes of different objects. | Comparable interface is used to sort the objects with natural ordering. |
| A Comparator interface compares two different class objects provided. | Comparable interface compares "this" reference with the object specified. |
| A Comparator is present in the java.util package. | Comparable is present in java.lang package. |
| Comparator doesn't affect the original class | Comparable affects the original class, i.e., the actual class is modified. |
| Comparator provides compare () method, equals () method to sort elements. | Comparable provides compareTo () method to sort elements. |

## How to implement volatile keyword in Java?

A **volatile** keyword is used to modify the value of a variable by different threads. It is also used to make classes thread-safe. It means that multiple threads can use a method and instance of the classes at the same time without any problem. The volatile keyword can be used either with primitive type or objects.

## How do you exit a function in Java?

You can exit a function using **java.lang.System.exit ()** method. This method terminates the currently running JVM.

It takes an argument "status code" where a non-zero status code indicates abnormal termination.

**System.exit ()** method calls the exit method in class runtime. It exits the current program by terminating JVM.

**System.exit** function has status code, which tells about termination, such as:

- **exit (0):** indicates successful termination
- **exit (1) or exit (-1) or any non-zero value**: indicates unsuccessful termination.

## What is Thread Pool and why is it used?

The **thread pool** in Java is actually a pool of Threads. In simple sense, it contains a group of worker threads that are waiting for the job to be granted. They are reused in the whole process.

In a **Thread Pool**, a group of fixed size threads is created. Whenever a task has to be granted, one of the threads is pulled out and assigned that task by the service provider, as soon as the job is completed the thread is returned back to the thread pool.

**Thread pool** is preferably used because active threads consume system resources, when is JVM creates too many threads at the same time, the system could run out of memory. Hence the number of threads to be created has to be limited. Therefore, the concept of thread pool is preferred!

### Risk in the Java Thread Pool

- **Thread Leakage**: If a thread is removed from the pool to perform a task but not returned back to it when the task is completed, thread leakage occurs.
- **Deadlock**: In Thread pool is executing thread is waiting for the output from the block the thread waiting in the queue due to unavailability of thread for execution, there's a case of a deadlock.
- **Resource Thrashing**: More number of threads than the optimal number required can cause starvation problems leading to resource thrashing.

### Advantage:

- Better performance and easy to access
- Saves time
- No need to create thread again and again

## What is Java factory pattern?

**Factory pattern** is used to create instances for classes.

The core idea behind the static factory method is to create and return instances wherein the details of the class module are hidden from the user.

### Advantages:

- The object that you create can be used without duplication of code.
- If you use a factory method instead of a constructor, the factory method can have different and more descriptive names also.
- Also, it removes the instantiation of the implementation classes from the client code.
- This method makes the code more robust, less coupled and easy to expand.

## What is join method in Java?

Join method in Java allows one thread to wait until another thread completes its execution. In simpler words, it means it waits for the other thread to die. It has a **void** type and throws **InterruptedException**.

Joining threads in Java has **three functions** namely:

- **Join ()** – waits for this thread to die
- **Join (long millis)** – waits at most millis milliseconds for this thread to die
- **Join (long millis, int nanos)** – waits at most millis milliseconds plus nano nanoseconds for this thread to die.

**Syntax:**

```
Public final void join ()

Public final void join (long millis)

Public final void join (long millis, int nanos)
```

## What is EJB in Java and How to implement it?

**Enterprise Java Beans (EJB)** is server-side software that helps to summarizes the business logic of a certain application.

It is a specification provided by Sun Microsystems to develop secured, robust and scalable distributed applications.

To run EJB application, you need an application server (EJB) container such as Glass-fish, web-logic, web-sphere etc. The functions it performs are as follows:

- Life-cycle management
- Security
- Transaction management
- Object pooling

## What are the types of EJB?

There are several types of enterprise Java beans.

1. **Session beans**
   These are non-persistent enterprise beans. There are two kinds of session beans:
   **Stateful** – a stateful session Bean maintains client-specific session information across several transactions. It exists for the duration of a single client/server session.
   **Stateless** – a stateless session bean is an old bye their containers so that they can easily handle several requests from clients.
2. **Entity beans**
   These beans contain persistent data and it can be saved in the data source. There are two types:
   **Container managed persistence** – These entity beans assign their persistence to the EJB container.
   **Bean managed persistence** – These entity beans manage their own persistence.

3. **Message-driven beans**
   Message Driven beans are enterprise beans that receive and process Java message service messages. They can be accessed only through messaging. They do not have interfaces.

When to use EJB?

There are certain cases when you can use enterprise Java beans;

- When your application needs remote access.
- When your application is business logic.
- When your application needs to be scalable.

Disadvantages

- The specification of EJB is pretty complicated and large.
- It creates costly and complex solutions.
- It takes time for development.

What is the concept of String Pool in Java?

**String Pool** in Java is a pool of strings (storage area) which is stored in Java **Heap Memory**.

Each time a string literal is created, the JVM checks the string literal pool first. If the string already exists in the string pool, a reference to the pooled instance returns. If the string does not exist in the pool, a new string object initializes and is placed in the pool.

**How to create a string?**

To create a String object in Java, there are two ways:

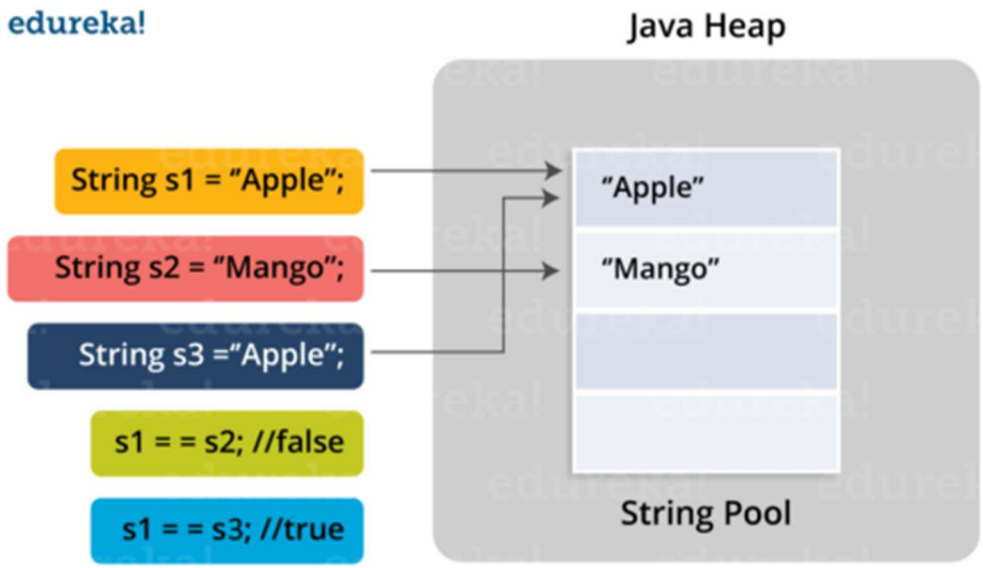
- Using the new operator.

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

- Using a string literal or constant expression.

```
String s1 = "Joey"; (string literal) or

String s1 = "Joe" + "y"; (string constant expression)
```

String Pool Flow in Java




How do I convert a String to an int in Java?

There are two different ways to convert a String to an int:

- By invoking the method Integer.parseInt:
  Public static int parseInt (String s) throws NumberFormatException

- By invoking the method Interger.valueOf:
  Public static Integer valueOf (String s) throws NumberFormatException

Both methods can process either negative or positive numbers.

They throw the unchecked NumberFormatException, depending on what software you are developing, it could be useful to catch it and process it.

Example:

```
1.  try{
2.     int number1 = Integer.parseInt("-16");
3.  }catch (NumberFormatException e){
4.     // Exception handling
5.  }
6.
7.  try{
8.     Integer number2 = Integer.valueOf("-16");
9.  }catch (NumberFormatException e){
10.    // Exception handling
11. }
12.
13. // java.lang.NumberFormatException: For input string: "--16"
14. try{
```

```
15.     Integer.valueOf("--16");
16.  }catch(NumberFormatException e){
17.     e.printStackTrace();
18.  }
19.
```

How do I declare and initialize an array in Java?

There are several approaches for creating arrays in Java.

1. Creation of an empty but defined size array. Also possible in multidimensional. The size of elements of each dimension is fix.
2. Creation of an array by defining its content. The size of the array will be computed by the given content. Also possible in multidimensional, but the size of the elements may differ.
3. Creation of an initialized array by invoking the toArry method of a java.util.Stream (only possible in one-dimensional).

Example

```
1.  // Creation:
2.  int[] intArray1 = new int[5];
3.  Integer[] intArray1_2 = new Integer[5];
4.  int[] intArray2 = new int[]{1, 2, 3};
5.  int[] intArray3 = {1, 2, 3};
6.
7.  int[][] intArray4 = new int[2][2];
8.  Integer[][] intArray4_2 = new Integer[2][2];
9.  int[][] intArray5 = new int[][]{{1, 2}, {3, 4}};
10. int[][] intArray5_2 = new int[][]{{1}, {1, 2}, {1, 2, 3}};
11. int[][] intArray6 = {{1, 2}, {3, 4}};
12. int[][][] intArray6_2 = {{{0}}, {{1}, {1, 2}, {1, 2, 3}}};
13.
14. // Different notations:
15. int[] notationAsUsual = {1};
16. int notationAlsoFine[] = {1};
17.
18. // Editing:
19. int number = 0;
20. for (int i = 0; i < intArray4.length; i++) {
21. for (int j = 0; j < intArray4[i].length; j++) {
22. intArray4[i][j] = number;
23. number += 10;
24. }
25. }
```