# PROGRAMING LANGUAGE

**Program:** A program is a series of steps to complete a task.

**Programing language:** programming language is some syntax which help the program run.

**Type of program:**

1. Low Level: low level programing language directly interact with machine hardware. Machine language, assembly language (add, store, load).
2. High Level: high level language writes in human language but convert into machine language. Java, C++, COBAL.

**Complier:** A compiler is a special program that translates a programming language's source code into machine code, bytecode, or another programming language.  The source code is typically written in a high-level, human-readable language such as Java or C++. A programmer writes the source code in a code editor or an integrated development environment (IDE) that includes an editor, saving the source code to one or more text files. Compilers that translate source code to machine code target specific operating systems and computer architectures.

**Interpreter:** Compilers are sometimes confused with programs called interpreters. Although the two are similar, they differ in important ways. Interpreters do not generate IR code or save generated machine code. They process the code one statement at a time at runtime, without pre-converting the code or preparing it in advance for a particular platform. Interpreters are used for code written in scripting languages such as Perl, PHP, Ruby, or Python.

**Programing Language can be divided with Approach to develop a software ---**

1. **Procedural oriented:** Procedural Programming is a programming language that follows a step-by-step approach to break down a task into a collection of variables and routines (or subroutines) through a sequence of instructions. In procedural oriented programming, each step is executed in a systematic manner so that the computer can understand what to do.

The programming model of the procedural oriented programming is derived from structural programming. The concept followed in the procedural oriented programming is called the "procedure". These procedures consist several computational steps that are carried out during the execution of a program. Examples of procedural oriented programming language include – C, Pascal, ALGOL, COBOL, BASIC, etc.

1. **Object oriented:** Object-oriented Programming is a programming language that uses classes and objects to create models based on the real world environment. These objects contain data in the form of attributes and program codes in the form of methods or functions. In OOP, the computer programs are designed by using the concept of objects that can interact with the real world entities.

We have several types of object oriented programming languages, but the most popular is one among all is class-based language. In the class-based OOP languages, the objects are the instances of the classes that determine their types. Examples of some object oriented programming languages are – Jave, C++, C#, Python, PHP, Swift, etc.

1. **Aspect oriented:** Aspect oriented programming (AOP) as the name suggests uses aspects in programming. It can be defined as the breaking of code into different modules, also known as modularisation, where the aspect is the key unit of modularity. Aspects enable the implementation of crosscutting concerns such as- transaction, logging not central to business logic without cluttering the code core to its functionality. It does so by adding additional behavior that is the advice to the existing code. For example- Security is a crosscutting concern, in many methods in an application security rules can be applied, therefore repeating the code at every method, define the functionality in a common class and control were to apply that functionality in the whole application. Java spring,

**Programming language can be divided with object based:**

* + - 1. Object oriented
      2. Object based
      3. Pure object

**Platform:**

1. Platform is the place or a runtime environment on which our program is going to be execute.
2. Any hardware or software environment in which a program runs, is known as a platform.
3. Since Java has a runtime environment (JRE) and API, it is called a platform.

**Platform Type:**

1. Software platform
2. Hardware platform
3. Combination of both

**Topics:**

1. **Source code:** The code writes in human language and save is called source code.
2. **Object code**: After compiler the code make that called object code.
3. **Explicit:** which in can be explained called explicit
4. **Implicit:** which cannot be explained but can be understand by logic is called implicit. If x>y and y>z then x>z we know by logic. Is called implicit
5. **32bit:** 32-bit, in computer systems, refers to the number of bits that can be transmitted or processed in parallel. In computer architecture, 32-bit integers, memory addresses, or other data units are those that are 32 bits (4 octets) wide.
6. **64bit:** In computer architecture, 64-bit integers, memory addresses, or other data units are those that are 64 bits wide.

**How C / C++ program run in computer:**

First, we write a C/C++ source code. Then compiler, compile it with GCC (GNU Compiler Collections)/TCC (turbo c compiler). Compiler called the O.S library and convert all code to object code.

But now in windows, compiler convert source code in exc file. In exc file add necessary O.S file at compiler time. So, there is no need for create object file. This exc file only run-in windows.

**How run java:**

When we compile the java code, first java toolkit run JVM software. Then JVM software run main Thread. Main thread loads all class of java. Main thread called main method. Then run software line by line.

# JAVA

**What is java:**

Java is a programming language and a **platform**. Java is a high level, robust, platform independent, object-oriented, and secure programming language. Java was developed by Sun Microsystems (which is now the subsidiary of Oracle) in the year 1995.

**Java feature:**

1. Simple
2. Secure
3. Platform independent
4. Robust
5. Architecture Neutral: Java run any type of hardware architecture.
6. Multithread:
7. Distributed: Java can create distribute software. Distribute software are RMI, EJB, Web service.
8. Dynamic:
9. No Explicit pointer: we can excess the address of dynamic memory but cannot excess by \* or & .
10. Automatic Garbage collector.

## Types of Java Applications:

1. **Standalone Application:** Standalone applications are also known as desktop applications or window-based applications. These are traditional software that we need to install on every machine. Examples of standalone application are Media player, antivirus, etc. AWT and Swing are used in Java for creating standalone applications.
2. **Web Application:** An application that runs on the server side and creates a dynamic page is called a web application. Servlet , JSP , Struts , Spring , Hibernate, JSF.
3. **Enterprise Application:** An application that is distributed in nature, such as banking applications, etc. is called an enterprise application. It has advantages like high-level security, load balancing, and clustering. In Java, EJB is used for creating enterprise applications.
4. **Mobile Application:** An application which is created for mobile device is called a mobile application. Currently, Android and Java ME are used for creating mobile applications.

## There are 4 platforms or editions of Java

1. **Java SE (Java Standard Edition):** It is a Java programming platform. It includes Java programming APIs such as java.lang , java.io, java.net, java.util , java.sql, java. Math etc. It includes core topics like OOPs, String, Regex, Exception, Inner classes, Multithreading, I/O Stream, Networking, AWT, Swing, Reflection, Collection, etc.
2. **Java EE (Java Enterprise Edition):** It is an enterprise platform that is mainly used to develop web and enterprise applications. It is built on top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.
3. **Java ME (Java Micro Edition):** It is a micro platform that is dedicated to mobile applications.
4. **JavaFX:** It is used to develop rich internet applications. It uses a lightweight user interface API.

## Java Version History

1. **JDK Alpha and Beta (1995):** The Java Alpha and Beta was the first releases, but they have highly unstable APIs and ABIs. The supplied Java web browser was named Web Runner.
2. **JDK 1.0 (January 23, 1996):** It was the first stable released version of Java. Its codename was **Oak**. The first stable version of JDK was JDK 1.0.2 and it was called Java 1. Up to JDK 1.0.1, private and protected keywords could be used together to create yet another form of protection which used to act as a restriction to methods or variables mainly to subclasses of a given class. In JDK 1.0.2, this capability has been removed.
3. **JDK 1.1 (February 19, 1997):** Some additions were included to this version. i.e.
4. The concept of Inner Class
5. JavaBeans
6. JDBC
7. RMI
8. AWT event model was totally reshaped.
9. Reflection (which supported Introspection only, modification was not possible at runtime).
10. JIT(Just In Time) compiler on Microsoft Windows platforms, produced for Java Soft by Symantec Internationalization
11. Unicode support originating from Telligent.
12. **J2SE 1.2 (December 8, 1998):**Its codename was **Playground**. First time, it was called **J2SE (Java 2 Platform, Standard Edition). It** replaced JDK to recognize the base platform from **J2EE (Java 2 Platform, Enterprise Edition) and J2ME (Java 2 Platform, Micro Edition)**.It was a very important java release as it tripled the size of the Java platform to 1520 classes in 59 packages.

Some additions were included to this version. i.e.

* + 1. Java plug-in
    2. Java IDL, an IDL implementation for CORBA interoperability
    3. Collection’s framework
    4. the Swing graphical API was integrated into the core classes

1. **J2SE 1.3 (May 8, 2000):** Its code name was **Kestrel**. Some additions were included to this version. i.e.
2. HotSpot JVM included.
3. RMI was modified to support optional compatibility with CORBA.
4. JNDI (Java Naming and Directory Interface).
5. Java Platform Debugger Architecture (JPDA) included.
6. JavaSound.
7. Synthetic proxy classes.
8. **J2SE 1.4 (February 6, 2002):** Its codename was **Merlin**. It was the first Java platform which was released under the Java Community Process. Some additions were included to this version. i.e.
9. Improved libraries.
10. Perl regular expressions included.
11. Provided exception chaining (It allows an exception to encapsulate original lower-level exception).
12. IPv6 support (Internet Protocol version 6).
13. Logging API (Specified in JSR 47.)
14. Image I/O API for reading and writing images in formats like JPEG and PNG.
15. XML parser and XSLT processor integrated.
16. Security and cryptography extensions (JCE, JSSE, JAAS) integrated.
17. Support and security updates for Java 1.4 ended in October 2008.
18. **J2SE 5.0 (September 30, 2004):** Its codename was Tiger. It was originally numbered 1.5, which is still used as the internal version number. So, it was changed to 5.0 to "better reflect the level of maturity, stability, scalability and security of the J2SE". This process also was released under the Java Community Process. Support and security updates for Java 5.0 ended on November 3, 2009, but updates were available to paid Oracle customers until May 2015. J2SE 5.0. added some significant new language features:

It provided compile-time (static) type safety for collections and eliminates the need for most typecasts.

1. Used Metadata or annotations.
2. Autoboxing/unboxing.
3. Enumerations.
4. Enhanced for each loop.
5. Improved semantics of execution for multi-threaded Java programs.
6. Static imports.

There were also some improvements in standard libraries:

1. Automatic stub generation for RMI objects.
2. Swing: It provided a skinny look and feel.
3. The concurrency utilities in package java.util.concurrent.
4. Scanner class for parsing data from various input streams and buffers.
5. Java 5 was the last release of Java which officially supported the Microsoft Windows 9x line (Windows 95, Windows 98, Windows ME).
6. Windows Vista was the last version of Windows that J2SE 5 supported before going to end in October 2009.
7. Java 5.0 is the default version of Java installed on Apple Mac OS X 10.5 (Leopard). Java 6 can be installed
8. **Java SE 6 (December 11, 2006):** Its codename was **Mustang**. After the release of this version, Java replaced the name J2SE to Java SE and dropped the .0 from the version number. Some additions were included to this version. i.e.
9. Dropped the support for older Win9x versions.
10. Scripting Language Support.
11. Generic API for tight integration with scripting languages.
12. Improved Web Service support.
13. JDBC 4.0 support.
14. Use a Java Compiler API to invoke a Java Compiler programmatically.
15. After the release of Java 6, Sun released many updates to fix bugs.
16. **Java SE 7 (July 28, 2011):** Its codename was Dolphin. It was launched on 7, July 2011 but was made available for developers on July 28, 2011.
17. JVM support for dynamic languages.
18. Compressed 64-bits pointer.
19. Strings added in switch.
20. Automatic resource management in try-statement.
21. Underscores allowed in numeric literals.
22. Binary integer literals.
23. Improved type interface for creating generic instance. (also called diamond operator <>)
24. Improved catching and throwing. (catch multiple exceptions and rethrow with improved type checking)
25. Provided Java Deployment rulesets.
26. It was the default version to download on java.com from April 2012 up to the release of Java 8.
27. **Java SE 8 (March 18, 2014):** its codename was **Spider**. Although, codenames have been discontinued, but the codename **Spider** is common among java developers. It includes some features which were proposed for Java SE 7 but added in Java SE 8.
28. Language-level support for Lambda expressions.
29. Allowed developers to embed JavaScript code within applications.
30. Annotation of Java Types.
31. Provided Date and Time API.
32. Repeating Annotations.
33. Launching of JavaFX applications.
34. Removal of permanent generation.
35. Java 8 is set as a default version to download from java.com from October 2014.
36. **Java SE 9 (September 21, 2017):** In 2016, Oracle discussed some features to release in Java 9. It was hoped that Java 9 would include better support for multi-gigabyte heaps, better native code integration, a different default garbage collector and a self-tuning JVM. The release of Java 9 was postponed many times and finally it was released on September 21, 2017.
37. Modularization of the JDK under Project Jigsaw.
38. Provided Money and Currency API.
39. Tight integration with JavaFX.
40. Java implementation of reactive streams.
41. More Concurrency Updates.
42. Provided Java Linker.
43. Automatic scaling and sizing.
44. **Java SE 10 (March, 20, 2018):** Java SE 10 was released to remove primitive data types and move towards 64-bit addressable arrays to support large data sets. It was released on 20 March 2018, with twelve new features confirmed. These features are:
45. Local-Variable Type Inference
46. Experimental Java-Based JIT Compiler This is the integration of the Graal dynamic compiler for the Linux x64 platform
47. Application Class-Data Sharing This allows application classes to be placed in the shared archive to reduce startup and footprint for Java applications
48. Time-Based Release Versioning
49. Parallel Full GC for G1
50. Garbage-Collector Interface
51. Additional Unicode Language-Tag Extensions
52. Root Certificates
53. Thread-Local Handshakes
54. Heap Allocation on Alternative Memory Devices
55. Remove the Native-Header Generation Tool - javah
56. Consolidate the JDK Forest into a Single Repository

## Java Architecture

Java Architecture is a collection of components, i.e., JVM, JRE, and JDK. It integrates the process of interpretation and compilation. It defines all the processes involved in creating a Java program. Java Architecture explains every step of how a program is compiled and executed. Java Architecture can be explained by using the following steps: There is a process of compilation and interpretation in Java. Java compiler converts the Java code into byte code. After that, the JVM converts the byte code into machine code. The machine code is executed by the machine.



Now let's dive deep to get more knowledge about **Java Architecture**. As we know that the Java architecture is a collection of components, so we will discuss every component into detail.

**Components of Java Architecture:** The Java architecture includes the three main components:

1. **Java Virtual Machine (JVM)**
2. **Java Runtime Environment (JRE)**
3. **Java Development Kit (JDK)**
4. **Java Virtual Machine (JVM):** The main feature of Java is **Write Once Run Anywhere (WORA)**. The feature states that we can write our code once and use it anywhere or on any operating system. Our Java program can run any of the platforms only because of the Java Virtual Machine. It is a Java platform component that gives us an environment to execute java programs. JVM's main task is to convert byte code into machine code. JVM, first, loads the code into memory and verifies it. After that, it executes the code and provides a runtime environment. Java Virtual Machine (JVM) has its own architecture, which is given below:

**JVM Architecture:** JVM is an abstract machine that provides the environment in which Java bytecode is executed. The falling figure represents the architecture of the JVM.



1. **Class Loader:** ClassLoader is a subsystem used to load class files. ClassLoader first loads the Java code whenever we run it
2. **Class Method Area:** In the memory, there is an area where the class data is stored during the code's execution. Class method area holds the information of static variables, static methods, static blocks, and instance methods.
3. **Heap:**  The heap area is a part of the JVM memory and is created when the JVM starts up. Its size cannot be static because it increases or decrease during the application runs.
4. **Stack:** It is also referred to as thread stack. It is created for a single execution thread. The thread uses this area to store the elements like the partial result, local variable, data used for calling method and
5. Returns etc.
6. **Native Stack:** It contains the information of all the native methods used in our application.
7. **Execution Engine:** It is the central part of the JVM. Its main task is to execute the byte code and execute the Java classes. The execution engine has three main components used for executing Java classes.
8. **Interpreter:** It converts the byte code into native code and executes. It sequentially executes the code. The interpreter interprets continuously and even the same method multiple times. This reduces the performance of the system, and to solve this, the JIT compiler is introduced.
9. **JIT Compiler:** JIT compiler is introduced to remove the drawback of the interpreter. It increases the speed of execution and improves performance.
10. **Garbage Collector:** The garbage collector is used to manage the memory, and it is a program written in Java. It works in two phases, i.e., **Mark** and **Sweep**. Mark is an area where the garbage collector identifies the used and unused chunks of memory. The Sweep removes the identified object from the **Mark**
11. **Java Native Interface:** Java Native Interface works as a mediator between Java method calls and native libraries.
12. **Java Runtime Environment (JRE):** It provides an environment in which Java programs are executed. JRE takes our Java code, integrates it with the required libraries, and then starts the JVM to execute it. JRE (Java Runtime Environment) is a software package that provides Java class libraries, Java Virtual Machine (JVM), and other components that are required to run Java applications is the superset of JVM. If you need to run Java programs, but not develop them, JRE is what you need. You can download JRE from Java SE Runtime Environment 8 Down.



Java Runtime Environment

1. **Java Development Kit:** It is a software development environment used in the development of Java applications and applets. Java Development Kit holds JRE, a compiler, an interpreter or loader, and several development tools in it. JDK (Java Development Kit) is a software development kit required to develop applications in Java. When you download JDK, JRE is also downloaded with it.In addition to JRE, JDK also contains a number of development tools (compilers, JavaDoc, Java Debugger, etc.)



**Relationship between JVM, JRE, and JDK.**



**Java run:**

1. With IDE: With IDE all programing necessary environment setup perfectly. So we do not need anything. Eclipse, NetBeans.
2. Without IDE: Without IDE we need compiler, library, run environment, editor.

## Class loading:

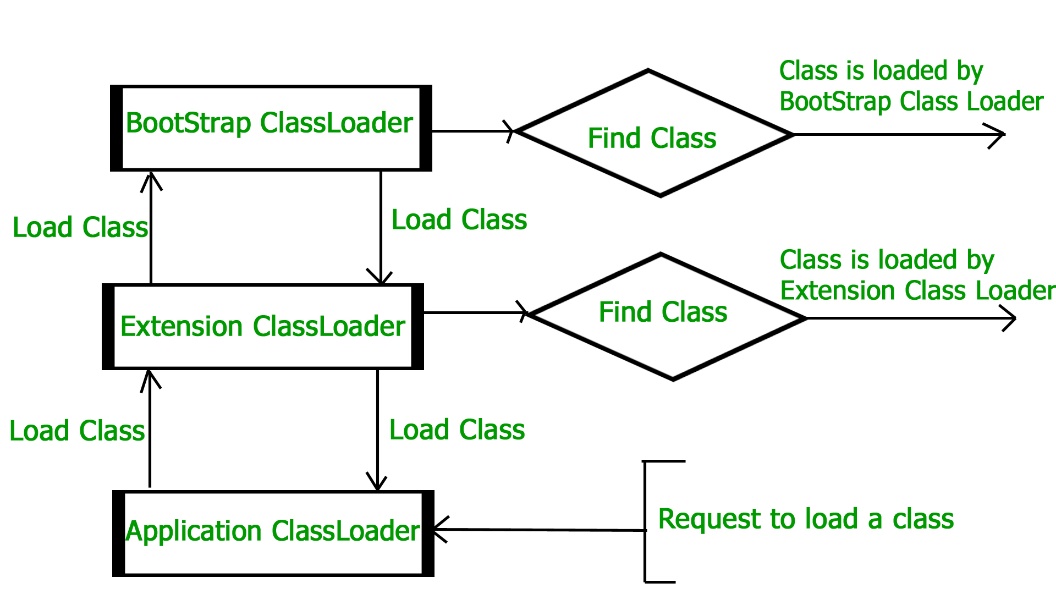
The Java ClassLoader is a part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine. The Java run time system does not need to know about files and file systems because of classloaders. Java classes aren’t loaded into memory all at once, but when required by an application. At this point, the Java ClassLoader is called by the JRE and these ClassLoaders load classes into memory dynamically.

**Types of Class Loaders in Java**

Not all classes are loaded by a single ClassLoader. Depending on the type of class and the path of class, the ClassLoader that loads that particular class is decided. To know the ClassLoader that loads a class the getClassLoader() method is used. All classes are loaded based on their names and if any of these classes are not found then it returns a NoClassDefFoundError or ClassNotFoundException. A Java Classloader is of three types:

1. **BootStrap ClassLoader**: A Bootstrap Classloader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location rt.jar. Bootstrap ClassLoader doesn’t have any parent ClassLoaders. It is also called as the Primordial ClassLoader.
2. **Extension ClassLoader:** The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from jre/lib/ext directory or any other directory pointed by the system property java.ext.dirs.
3. **System ClassLoader:** An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable CLASSPATH, -classpath or -cp command line option. The Application ClassLoader is a child class of Extension ClassLoader.

Note: The ClassLoader Delegation Hierarchy Model always functions in the order Application ClassLoader->Extension ClassLoader->Bootstrap ClassLoader. The Bootstrap ClassLoader is always given the higher priority, next is Extension ClassLoader and then Application ClassLoader. Types of ClassLoaders in Java

[](https://media.geeksforgeeks.org/wp-content/uploads/jvmclassloader.jpg)

Not all classes are loaded by a single ClassLoader. Depending on the type of class and the path of class, the ClassLoader that loads that particular class is decided. To know the ClassLoader that loads a class the getClassLoader() method is used. All classes are loaded based on their names and if any of these classes are not found then it returns a NoClassDefFoundError or ClassNotFoundException. A Java Classloader is of three types:

1. BootStrap ClassLoader: A Bootstrap Classloader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location rt.jar. Bootstrap ClassLoader doesn’t have any parent ClassLoaders. It is also called as the Primordial ClassLoader.
2. Extension ClassLoader: The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from jre/lib/ext directory or any other directory pointed by the system property java.ext.dirs.
3. System ClassLoader: An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable CLASSPATH, -classpath or -cp command line option. The Application ClassLoader is a child class of Extension ClassLoader.

Note: The ClassLoader Delegation Hierarchy Model always functions in the order Application ClassLoader->Extension ClassLoader->Bootstrap ClassLoader. The Bootstrap ClassLoader is always given the higher priority, next is Extension ClassLoader and then Application ClassLoader.

**Principles of functionality of a Java ClassLoader**

Principles of functionality are the set of rules or features on which a Java ClassLoader works. There are three principles of functionality, they are:

1. Delegation Model: The Java Virtual Machine and the Java ClassLoader use an algorithm called the Delegation Hierarchy Algorithm to Load the classes into the Java file. The ClassLoader works based on a set of operations given by the delegation model. They are:

ClassLoader always follows the Delegation Hierarchy Principle.

* Whenever JVM comes across a class, it checks whether that class is already loaded or not.
* If the Class is already loaded in the method area then the JVM proceeds with execution.
* If the class is not present in the method area then the JVM asks the Java ClassLoader Sub-System to load that particular class, then ClassLoader sub-system hands over the control to Application ClassLoader.
* Application ClassLoader then delegates the request to Extension ClassLoader and the Extension ClassLoader in turn delegates the request to Bootstrap ClassLoader.
* Bootstrap ClassLoader will search in the Bootstrap classpath(JDK/JRE/LIB). If the class is available then it is loaded, if not the request is delegated to Extension ClassLoader.
* Extension ClassLoader searches for the class in the Extension Classpath(JDK/JRE/LIB/EXT). If the class is available then it is loaded, if not the request is delegated to the Application ClassLoader.
* Application ClassLoader searches for the class in the Application Classpath. If the class is available then it is loaded, if not then a ClassNotFoundException exception is generated.

1. Visibility Principle: The Visibility Principle states that a class loaded by a parent ClassLoader is visible to the child ClassLoaders but a class loaded by a child ClassLoader is not visible to the parent ClassLoaders. Suppose a class GEEKS.class has been loaded by the Extension ClassLoader, then that class is only visible to the Extension ClassLoader and Application ClassLoader but not to the Bootstrap ClassLoader. If that class is again tried to load using Bootstrap ClassLoader it gives an exception java.lang.ClassNotFoundException.
2. Uniqueness Property: The Uniqueness Property ensures that the classes are unique and there is no repetition of classes. This also ensures that the classes loaded by parent classloaders are not loaded by the child classloaders. If the parent class loader isn’t able to find the class, only then the current instance would attempt to do so itself.

# VARIABLES IN JAVA

1. A variable is a container which holds the value while the Java program is executed.
2. A variable is assigned with a data type.
3. Variable is a name of memory location

There are 3 types of variables in Java programming language:

1. Local Variables
2. Instance Variables (Non-Static Fields)
3. Static Variables (class Fields)

**1)Local Variable:**

1. A variable declared inside the declared method, constructor, or block is called local variable.
2. Local variables are visible only within the declared method, constructor, or block.
3. A local variable cannot be defined with "static" keyword.
4. There is no default value for local variables, so local variables should be declared and an initial value should be assigned before the first use.

public class Test

{

public void pupAge() {

int age = 0 ; //// local variable

age = age + 7;

System.out.println("Puppy age is : " + age);

}

public static void main(String args[]) {

Test pet = new Test();

pet.pupAge();

}

}

**2) Instance Variable:**

1. A variable declared inside the class but outside the body of the method, is called an instance variable.
2. It is not declared as static.
3. It is called an instance variable because its value is instance-specific and is not shared among instances.
4. The instance variables are visible for all methods, constructors, and block in the class.
5. Instance Variable can be used only by creating objects
6. Instance variables have default values. For numbers, the default value is 0, for Booleans it is false, and for object references it is null. Values can be assigned during the declaration or within the constructor.
7. Intense value must be initializer.

public class Employee

{

public String name; /instance variable

private double salary;

public Employee (String empName) {

name = empName;

}

public void setSalary(double empSal) {

salary = empSal;

}

}

**3) Static variable/class variable:**

1. A variable that is declared as static is called a static variable.
2. It cannot be local.
3. You can create a single copy of the static variable and share it among all the instances of the class.
4. Memory allocation for static variables happens only once when the class is loaded in the memory.
5. Default values are same as instance variables. For numbers, the default value is 0; for Booleans, it is false; and for object references, it is null. Values can be assigned during the declaration or within the constructor. Additionally, values can be assigned in special static initializer blocks.

**public class Employee{**

**private static double salary;**

**public static final String DEPARTMENT = "Development "; /static variable**

**public static void main(String args[]) {**

**salary = 1000;**

**System.out.println(DEPARTMENT + "average salary:" + salary);**

**}**

**}**

# DATA TYPES

Data types specify the different sizes and values that can be stored in the variable. There are two types of data types in Java:

1. **Primitive data types:**
2. **Non-primitive data types**

**JAVA PRIMITIVE DATA TYPES:**

1. In Java language, primitive data types are the building blocks of data manipulation.
2. All variables must be declared before its use.
3. We need to declare variable's type and name.
4. Boolean: false 1 bit
5. Char : 2 bytes
6. Byte : 1 byte
7. Short: 2 bytes
8. Int : 4 bytes
9. Long: 8 bytes
10. Float : 4 bytes
11. Double : 8 bytes

**JAVA NON-PRIMITIVE DATA TYPES:**

1. Arrays.
2. Classes
3. Interfaces

**Literals in Java:** Any constant value which can be assigned to the variable is called literal/constant.

int x = 100;

1. **Integral literals:**
2. **Floating-Point literal**: this can be expressed using only decimal fractions, exponential notation.
3. **Char literals:** this literal have the type char and are an unsigned integer primitive type.
4. **String literals:** A sequence of characters within double quotes is referred to as string literals.
5. **Boolean literals**: Boolean literals allow only two values and thus are divided into two literals:
   * + 1. True: it represents a real Boolean value
       2. False: it represents a false Boolean value
6. **Null Literals:** Null literal is a particular literal in Java representing a null value

**TYPE CONVERTION:** It is the process of converting one type of data into another type.

1. **Widening / Type Promotion (implicit):** Converting a lower datatype to a higher datatype is known as widening. In this case the casting/conversion is done automatically therefore, it is known as implicit type casting. In this case both datatypes should be compatible with each other.



public class Widening{

   public static void main(String args[]){

      char X = 'C';

      int value = X;

   }

}

1. **Narrowing / Type Casting (Explicit):** Converting a higher datatype to a lower datatype is known as narrowing. In this case the casting/conversion is not done automatically, you need to convert explicitly using the cast **operator “( )”** explicitly. Therefore, it is known as explicit type casting. In this case both datatypes need not be compatible with each other.



**public class** **Narrowing** {

**public static void** **main**(String args[]){

**int** value = 10;

**char** ch = (char) value ;

    }

}

# FUNDAMENTAL

## Operators in Java ----- 106

Operator in Java is a symbol that is used to perform operations. For example: +, -, \*, / etc. There are many types of operators in Java which are given below:

1. Unary Operator,
2. Arithmetic Operator,
3. Shift Operator,
4. Relational Operator,
5. Bitwise Operator,
6. Logical Operator,
7. Ternary Operator and
8. Assignment Operator.

**Java Operator Precedence:**

|  |  |  |
| --- | --- | --- |
| Operator Type | Category | Precedence |
| Unary | postfix | expr++ expr-- |
| prefix | ++expr --expr +expr -expr ~ ! |
| Arithmetic | multiplicative | \* / % |
| additive | + - |
| Shift | shift | << >> >>> |
| Relational | comparison | < > <= >= instance of |
| equality | == != |
| Bitwise | bitwise AND | & |
| bitwise exclusive OR | ^ |
| bitwise inclusive OR | | |
| Logical | logical AND | && |
| logical OR | || |
| Ternary | ternary | ? : |
| Assignment | assignment | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

Operator overload:

Some operator uses in various purpose. As like + operator use in integer as add , but in string it uses as concatenation.

## List of Java Keywords

A list of Java keywords or reserved words are given below:

1. [**abstract**](https://www.javatpoint.com/abstract-keyword-in-java)**:** Java abstract keyword is used to declare an abstract class. An abstract class can provide the implementation of the interface. It can have abstract and non-abstract methods.
2. [**boolean:**](https://www.javatpoint.com/boolean-keyword-in-java) Java boolean keyword is used to declare a variable as a boolean type. It can hold True and False values only.
3. [**break**](https://www.javatpoint.com/java-break)**:** Java break keyword is used to break the loop or switch statement. It breaks the current flow of the program at specified conditions.
4. [**byte**](https://www.javatpoint.com/byte-keyword-in-java)**:** Java byte keyword is used to declare a variable that can hold 8-bit data values.
5. [**case**](https://www.javatpoint.com/case-keyword-in-java)**:** Java case keyword is used with the switch statements to mark blocks of text.
6. [**catch**](https://www.javatpoint.com/try-catch-block)**:** Java catch keyword is used to catch the exceptions generated by try statements. It must be used after the try block only.
7. [**char**](https://www.javatpoint.com/char-keyword-in-java)**:** Java char keyword is used to declare a variable that can hold unsigned 16-bit Unicode characters
8. [**class**](https://www.javatpoint.com/class-keyword-in-java)**:** Java class keyword is used to declare a class.
9. [**continue**](https://www.javatpoint.com/java-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.
10. [**default**](https://www.javatpoint.com/default-keyword-in-java)**:** Java default keyword is used to specify the default block of code in a switch statement.
11. [**do**](https://www.javatpoint.com/java-do-while-loop)**:** Java do keyword is used in the control statement to declare a loop. It can iterate a part of the program several times.
12. [**double**](https://www.javatpoint.com/double-keyword-in-java)**:** Java double keyword is used to declare a variable that can hold 64-bit floating-point number.
13. [**else**](https://www.javatpoint.com/java-if-else)**:** Java else keyword is used to indicate the alternative branches in an if statement.
14. [**enum**](https://www.javatpoint.com/enum-in-java)**:** Java enum keyword is used to define a fixed set of constants. Enum constructors are always private or default.
15. [**extends**](https://www.javatpoint.com/inheritance-in-java)**:** Java extends keyword is used to indicate that a class is derived from another class or interface.
16. [**final**](https://www.javatpoint.com/final-keyword)**:** Java final keyword is used to indicate that a variable holds a constant value. It is used with a variable. It is used to restrict the user from updating the value of the variable.
17. [**finally**](https://www.javatpoint.com/finally-block-in-exception-handling)**:** Java finally keyword indicates a block of code in a try-catch structure. This block is always executed whether an exception is handled or not.
18. [**float**](https://www.javatpoint.com/float-keyword-in-java)**:** Java float keyword is used to declare a variable that can hold a 32-bit floating-point number.
19. [**for**](https://www.javatpoint.com/java-for-loop)**:** Java for keyword is used to start a for loop. It is used to execute a set of instructions/functions repeatedly when some condition becomes true. If the number of iteration is fixed, it is recommended to use for loop.
20. [**if**](https://www.javatpoint.com/java-if-else)**:** Java if keyword tests the condition. It executes the if block if the condition is true.
21. [**implements**](https://www.javatpoint.com/interface-in-java)**:** Java implements keyword is used to implement an interface.
22. [**import**](https://www.javatpoint.com/package)**:** Java import keyword makes classes and interfaces available and accessible to the current source code.
23. [**instanceof**](https://www.javatpoint.com/downcasting-with-instanceof-operator)**:** Java instanceof keyword is used to test whether the object is an instance of the specified class or implements an interface.
24. [**int**](https://www.javatpoint.com/int-keyword-in-java)**:** Java int keyword is used to declare a variable that can hold a 32-bit signed integer.
25. [**interface**](https://www.javatpoint.com/interface-in-java)**:** Java interface keyword is used to declare an interface. It can have only abstract methods.
26. [**long**](https://www.javatpoint.com/long-keyword-in-java)**:** Java long keyword is used to declare a variable that can hold a 64-bit integer.
27. **native:** Java native keyword is used to specify that a method is implemented in native code using JNI (Java Native Interface).
28. [**new**](https://www.javatpoint.com/new-keyword-in-java)**:** Java new keyword is used to create new objects.
29. [**null**](https://www.javatpoint.com/null-keyword-in-java)**:** Java null keyword is used to indicate that a reference does not refer to anything. It removes the garbage value.
30. [**package**](https://www.javatpoint.com/package)**:** Java package keyword is used to declare a Java package that includes the classes.
31. [**private**](https://www.javatpoint.com/private-keyword-in-java)**:** 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.
32. [**protected**](https://www.javatpoint.com/protected-keyword-in-java)**:** Java protected keyword is an access modifier. It can be accessible within the package and outside the package but through inheritance only. It can't be applied with the class.
33. [**public**](https://www.javatpoint.com/public-keyword-in-java)**:** 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.
34. [**return**](https://www.javatpoint.com/return-keyword-in-java)**:** Java return keyword is used to return from a method when its execution is complete.
35. [**short**](https://www.javatpoint.com/short-keyword-in-java)**:** Java short keyword is used to declare a variable that can hold a 16-bit integer.
36. [**static**](https://www.javatpoint.com/static-keyword-in-java)**:** Java static keyword is used to indicate that a variable or method is a class method. The static keyword in Java is mainly used for memory management.
37. [**strictfp**](https://www.javatpoint.com/strictfp-keyword)**:** Java strictfp is used to restrict the floating-point calculations to ensure portability.
38. [**super**](https://www.javatpoint.com/super-keyword)**:** Java super keyword is a reference variable that is used to refer to parent class objects. It can be used to invoke the immediate parent class method.
39. [**switch**](https://www.javatpoint.com/java-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.
40. [**synchronized**](https://www.javatpoint.com/synchronization-in-java)**:** Java synchronized keyword is used to specify the critical sections or methods in multithreaded code.
41. [**this**](https://www.javatpoint.com/this-keyword)**:** Java this keyword can be used to refer the current object in a method or constructor.
42. [**throw**](https://www.javatpoint.com/throw-keyword)**:** The Java throw keyword is used to explicitly throw an exception. The throw keyword is mainly used to throw custom exceptions. It is followed by an instance.
43. [**throws**](https://www.javatpoint.com/throws-keyword-and-difference-between-throw-and-throws)**:** The Java throws keyword is used to declare an exception. Checked exceptions can be propagated with throws.
44. [**transient**](https://www.javatpoint.com/transient-keyword)**:** Java transient keyword is used in serialization. If you define any data member as transient, it will not be serialized.
45. [**try**](https://www.javatpoint.com/try-catch-block)**:** 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.
46. **void:** Java void keyword is used to specify that a method does not have a return value.
47. [**volatile**](https://www.javatpoint.com/volatile-keyword-in-java)**:** Java volatile keyword is used to indicate that a variable may change asynchronously.
48. [**while**](https://www.javatpoint.com/java-while-loop)**:** 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 the while loop.

## Java Expressions, Statements

**Java Statements:**

In Java, each statement is a complete unit of execution. For example,

int score = 9\*5;

**Expression statements:**

We can convert an expression into a statement by terminating the expression with a **“;”.**

# JAVA CONTROL STATEMENTS ----- 107

Java compiler executes the code from top to bottom. The statements in the code are executed according to the order in which they appear. However, Java provides statements that can be used to control the flow of Java code. Such statements are called control flow statements.

Java provides three types of control flow statements.

1. Decision Making statements
   * if statements
   * switch statement
2. Loop statements
   * do while loop
   * while loop
   * for loop
   * for-each loop
3. Jump statements
   * break statement
   * continue statement

#### Decision Making statements:

1. **If Statement:** It is the most basic statement among all control flow statements in Java. It evaluates a Boolean expression and enables the program to enter a block of code if the expression evaluates to true.

if(condition) {

statement 1;

}

1. **if-else statement:** The if-else statement is an extension to the if-statement, which uses another block of code, i.e., else block. The else block is executed if the condition of the if-block is evaluated as false.

**if**(condition)

 {

statement 1;

}

**else**{

statement 2;

}

1. **if-else if ladder:** The if-else-if statement contains the if-statement followed by multiple else-if statements. it is the chain of if-else statements that create a decision tree where the program may enter in the block of code where the condition is true. An else statement at the end of the chain.

if(condition 1) {

statement 1;

}

else if(condition 2) {

statement 2;

}

else {

statement 2;

}

1. **Nested if-statement:** In nested if-statements, the if statement can contain a **if** or **if-else** statement inside another if or else-if statement.

if (condition 1) {

statement 1;

if (condition 2) {

statement 2;

}

else {

statement 2;

}

}

#### Switch Statement:

1. Switch statements are similar to if-else-if statements.
2. The case variables can be int, short, byte, char, or enumeration, String type.
3. Cases cannot be duplicate
4. Default statement is executed when any of the case doesn't match the value of expression. It is optional.
5. Break statement terminates the switch block when the condition is satisfied.  
   It is optional, if not used, next case is executed.
6. While using switch statements, we must notice that the case expression will be of the same type as the variable. However, it will also be a constant value.

**Syntax:**

switch (expression){

    case value1:

     statement1;

     break;

    case valueN:

     statementN;

     break;

    default:

      default statement;

}

**code:**

public class Student implements Cloneable {

public static void main(String[] args) {

int num = 2;

switch (num){

case 0:

System.out.println("number is 0");

break;

case 1:

System.out.println("number is 1");

break;

default:

System.out.println(num).

}

}

}

#### Loop Statements:

In programming, sometimes we need to execute the block of code repeatedly while some condition evaluates to true. However, loop statements are used to execute the set of instructions in a repeated order.

1. for loop
2. while loop
3. do-while loop
4. **For loop:** It enables us to initialize the loop variable, check the condition, and increment/decrement in a single line of code. We use the for loop only when we exactly know the number of times, we want to execute the block of code.

**for** (initialization, condition, increment/decrement) {

//block of statements

}

The flow chart for the for-loop is given below.



**code:**

**public** **class** Calculation {

**public** **static** **void** main (String[] args) {

**int** sum = 0;

**for** (**int** j = 1; j<=10; j++) {

sum = sum + j;

}

System.out.println("The sum of  " + sum);

}

}

1. **For-Each loop**: Java provides an enhanced for loop to traverse the data structures like array or collection. In the for-each loop, we don't need to update the loop variable. The syntax to use the for-each loop in java is given below.

**For** (datatype : array\_name){

//statements

}

**code:**

**public** **class** Calculation {

**public** **static** **void** main(String[] args) {

String[] names = {"Java","C",” Python","JavaScript"};

System.out.println("Printing the :\n");

**for**(String name: names) {

System.out.println(name);

}

}

}

1. **while loop:**The while loop is also used to iterate over the number of statements multiple times. However, if we don't know the number of iterations in advance, it is recommended to use a while loop. Unlike for loop, the initialization and increment/decrement doesn't take place inside the loop statement in while loop. It is also known as the entry-controlled loop since the condition is checked at the start of the loop. If the condition is true, then the loop body will be executed; otherwise, the statements after the loop will be executed.

The syntax of the while loop is given below.

**while**(condition) {

//looping statements

}



**code:**

**public** **class** Calculation {

**public** **static** **void** main(String[] args) {

**int** i = 0;

System.out.println("Printing the \n");

**while**(i<=10) {

System.out.println (i);

i = i + 2;

}

}

}

1. **do-while loop:** The do-while loop checks the condition at the end of the loop after executing the loop statements. When the number of iterations is not known and we have to execute the loop at least once, we can use do-while loop. It is also known as the exit-controlled loop since the condition is not checked in advance. The syntax of the do-while loop is given below.

**do**

{

//statements

} **while** (condition);



code:

**public** **class** Calculation {

**public** **static** **void** main(String[] args) {

**int** i = 0;

System.out.println("Printing the list of first 10 even numbers \n");

**do** {

System.out.println(i);

i = i + 2;

} **while**(i<=10);

}

}

#### Jump Statements:

Jump statements are used to transfer the control of the program to the specific statements. In other words, jump statements transfer the execution control to the other part of the program. There are two types of jump statements in Java,

1. break
2. continue.
3. **Java break statement:**
4. the break statement is used to break the current flow
5. it breaks only the inner loop in the case of the nested loop.
6. The break statement can only be written inside the loop or switch statement.

code:

public class BreakExample {

public static void main(String[] args) {

for(int i = 0; i<= 10; i++) {

System.out.println(i);

If ( i==6) {

break;

}

}

}

}

1. **Java continue statement:**

the continue statement doesn't break the loop, whereas it skips the specific part of the loop and jumps to the next iteration of the loop immediately.

**public** **class** ContinueExample {

**public** **static** **void** main(String[] args) {

**for**(**int** i = 0; i<= 2; i++) {

**for** (**int** j = i; j<=5; j++) {

**if**(j == 4)

**continue**;

System.out.println(j);

}

}

}

}

#### Java Label Statement:

1. Java does not have a general goto statement.They can use labels which are valid java identifiers with a colon.
2. Labeled blocks can only be used with break and continue statements.
3. Labaled break and continue statements must be called within its scope. We cannot refer them outside the scope of labeled block.
4. The break statement immediately jumps to the end (and out) of the appropriate compound statement.
5. The continue statement immediately jumps to the next iteration (if any) of the appropriate loop.
6. A continue statement does not apply to a switch statement or a block statement, only to compound statements (for-loop, while-loop, and do-while loop.)

outer:

for ( int i = 0; i < 10; i++){

inner:

for (int j = 10; j > 0; j--) {

if ( i == j) {

System.out.println(i);

break outer;

}else{

System.out.println("-->>" + i ) ;

continue inner;

}

}

}

# JAVA ARRAYS --- 111

1. An array is a collection of similar type of elements store in the contiguous memory location.
2. **Java array** is an object which contains elements of a similar data type.
3. We can store only a fixed set of elements in a Java array.
4. Array in Java is index-based, the first element of the array is stored at the 0th index
5. Java array inherits the Object class, and implements the Serializable as well as Clone able interfaces.
6. We can store primitive values or objects in an array in Java.



**Array is initialized:**

1. Passing no value within the initializer: int arr[ 5 ] = {  };
2. By passing specific values within the initializer: int arr[ 5 ] = { 1, 2, 3, 4, 5 };
3. Passing specific values within the initializer but not declaring the size: int arr[  ] = { 1, 2, 3, 4, 5 };

**Types of Arrays in java:**

1. **Single Dimensional Array:**

**Syntax:** datatype [] arrayName = new datatype[size];

datatype [] arrayName = {data1, data2, data3};

code:

class Testarray{

public static void main(String args[]){

int a [] =new int[5];

a[0]=10;

}

}  60

1. **Multidimensional Array:** Multidimensional arrays are arrays of arrays with each element of the array holding the reference of other arrays. These are also known as Jagged Arrays. A multidimensional array is created by appending one set of square brackets ([]) per dimension.

**Syntax:** dataType[][] arrayName = new datatype[size][];

**code:** int[][] arr=new int[3][3];

**What is the class name of Java array?**

In Java, an array is an object. For array object, a proxy class is created whose name can be obtained by

getClass(). getName() method on the object.

class Testarray4{

public static void main(String args[]){

int arr[] = {4,4,5};

Class c = arr.getClass();

String name = c.getName();

System.out.println(name);

}

}

#### Jagged Array:

#### Anonymous Array

# String handling ---- 113

1. String is sequence of character. It is combination of character.
2. String is object in java.
3. String is immutable.
4. The process of performing operation over the string is called string handling.
5. String handle package
6. Java.lang.string
7. Java.lang.StrignBuffer
8. Java.lang.StrignBuilder
9. Java.lang.StringTokenizer
10. **string class:**

**Create object of:**

1. **With new:** Here memory allocation in heap memory area. Every time new create memory allocated in heap memory.   
    string s = new String (“shuvo”)
2. **Without new or string literal:** In string literal, we just hold the string literal in variable. In java, string literal treated as string object.

Whenever we creating the string object via literal it allocated memory in String constant poll.

Here same string value memory allocated for only one. Other share same memory.

String s = “shuvo”

String cannot modify:

String s = “shuvo”’;

S= s.concat(“main”);

**Memory allocation of string:**

In java, we can store string value in two ways. By string literal and new keyword. For this two method, memory allocated process and location is different.

1. **For String literal**: memory allocated in “string constrain pool area”. Which is middle of head memory area. Here every string is allocated only time. if same value declared multiple time reference value only share with them new memory not allocation for same value. If string value not find in pool area then new memory allocated and new ref value pass.

Heap Memory

String str= “hello”;

String PP = “hello”;

1. For new keyword: “new keyword” object is always creating new memory allocation. Every time string create by new keyword JVM create new memory in heap area.

String str= “hello”;

String PP = “hello”;

String ob = new String(“shuvo”);

String m = new String(“shuvo”)

Heap Memory

“shuvo”

“shuvo”

**Method of String class:**

1. contains() : checks whether the string contains a substring
2. substring() : returns the substring of the string
3. join() : join the given strings using the delimiter
4. replace() : replaces the specified old character with the specified new character
5. charAt() : returns the character present in the specified location
6. getBytes() : converts the string to an array of bytes
7. indexOf() : returns the position of the specified character in the string
8. trim() : removes any leading and trailing whitespaces
9. format() : returns a formatted string
10. split() : breaks the string into an array of strings
11. toLowerCase() : converts the string to lowercase
12. toUpperCase() : converts the string to uppercase
13. valueOf() : returns the string representation of the specified argument
14. toCharArray() : converts the string to a char array
15. matches() : checks whether the string matches the given regex
16. isEmpty() : checks whether a string is empty of not
17. hashCode() : returns a hash code for the string
18. subSequence() : returns a subsequence from the string
19. **String Buffer class:**
20. Java StringBuffer class is used to create mutable (modifiable) String objects.
21. The StringBuffer class in Java is the same as String class except it is mutable
22. String buffer onlye creagte by new keyword:

stringBuffer obj = new StringBuffer (“shuvo”);

**constructor:**

1. StringBuffer()
2. StringBuffer(String str)
3. StringBuffer(int capacity)

**method:**

1. append(String s)
2. insert(int offset, String s)
3. replace(int startIndex, int endIndex, String str)
4. delete(int startIndex, int endIndex)
5. reverse()
6. capacity()
7. **Java StringBuilder Class:** Java StringBuilder class is used to create mutable (modifiable) String. The Java StringBuilder class is same as StringBuffer class except that it is non-synchronized.
8. **String Tokenizer class:**

The java.util.StringTokenizer class allows you to break a String into tokens. It is simple way to break a String. It is a legacy class of Java.

 StringTokenizer st = new StringTokenizer("my name is khan"," ");

# OPP

Java is an Object-Oriented Language. Everything in Java is associated with classes and objects, along with its attributes and methods when we make software for real world, we can identify real world entity by object.

As a language that has the Object-Oriented feature, Java supports the following fundamental concepts –

1. Classes
2. Objects
3. Inheritance
4. Polymorphism
5. Encapsulation
6. Abstraction
7. Instance
8. Method
9. Message Passing

## CLASSES

1. A class is a blueprint of object.
2. Class is user defined data type.
3. A class is a group of objects which have common properties
4. Class declared by class key word.
5. The class body is enclosed between curly braces { }.
6. The data or variables, defined within a class are called instance variables.
7. The code is contained within methods.
8. Collectively, the methods and variables defined within a class are called members of the class



**code:**

public class Dog {

String breed;

int age;

void barking() {

}

}

**Types of Classes:**

* + - 1. Bean class
      2. Pojo Class
      3. Final Class
      4. Static Class
      5. Abstract Class
      6. Concrete Class
      7. Inner Class
      8. Java Enum Class
      9. Java Reflection

**Other Special Classes in Java:**

1. Immutable Class
2. Singleton Class
3. Object Class
4. Wrapper Class

#### JAVA BEAN CLASS:

If normal class having the following features called java bean class---

1. Bean Class should be public.
2. It should have public default constructor.
3. Data member should be private.
4. It should have public setter & getter method.
5. It must implement Serialization.

**Code:**

import java.io.Serializable;

public class Login implements Serializable {

private String name;

public void setName(String x) {

this.name = x;

}

public String getName() {

return name;

}

}

Public class Test {

Login ob = new Login ();

ob.setName(“shuvo”);

System.out.println(ob.getName());

}

#### POJO Class (plain old java object):

1. The POJO class in java should always be public.
2. It should always have a public default constructor.
3. It may contain the arguments constructor.
4. It should contain public Getters and Setters.
5. The only difference between them that Bean Files have some restrictions but, the POJO files do not have any special restrictions.

code :

public class Sample{

public String field;

Sample();

String s {

this.field = s;

}

public String getField1() {

return this.field;

}

}

#### Concreate class:

1. A concrete class is any normal class in a Java program. This class will not have any abstract methods. All the methods in the concrete class are completely implemented.
2. A concrete class can inherit from another class, even an abstract class or implement an interface. We can instantiate the concrete class and create its objects.

public class Main { // Concrete Class example

    static int total (int val1, int val2) {

      return val1 + val2;

    }

    public static void main(String args[]) {

       int sum = total (100, 108);

      System.out.println("Total of two integers: " + sum);

    }

}

#### Object Class:

1. Object class is present in java.lang package.
2. Every class in Java is directly or indirectly derived from the Object class.
3. If a class does not extend any other class, then it is a direct child class of Object and if extends another class then it is indirectly derived.
4. Therefore, the Object class methods are available to all Java classes.
5. Hence object class acts as a root of the inheritance hierarchy in any Java Program.

**Object Class Methods:**

1. **tostring() method:** it provides a String representation or convert of an object to string form. Whenever we try to print any Object reference, then internally toString() method is called.
2. **hashCode() method**: it generates unique hash code for each object.
3. **equals(Object obj) method:** it is used to compare the two objects dynamically.
4. **finalize() method:** this method call required to perform garbage collector.
5. **getClass() method:**
6. **clone() method:** it used to create the copy or clone of object.
7. **wait()**
8. **notify()**
9. **notifyAll() methods**

#### Java Singleton Class:

Singleton is a design pattern that ensures that a class can only have one object. To create a singleton class, a class must implement the following properties:

1. Create a private constructor of the class to restrict object creation outside of the class.
2. Create a private attribute of the class type that refers to the single object.
3. Create a public static method that allows us to create and access the object we created. Inside the method, we will create a condition that restricts us from creating more than one object.

class Database {

private static Database dbObject;

private Database() {

}

public static Database getInstance() {

if(dbObject == null) { // create object if it's not already created

dbObject = new Database();

}

return dbObject; // returns the singleton object

}

public void getConnection() {

System.out.println("You are now connected to the database.");

}

}

class Main {

public static void main(String[] args) {

Database db1;

db1= Database.getInstance(); // refers to the only object of Database

db1.getConnection();

}

}

**## Singletons can be used while working with databases. They can be used to create a connection pool to access the database while reusing the same connection for all the client**

#### Wrapper Class:

1. The wrapper class convert primitive data type into object and object into primitive data type.
2. A Wrapper class is a class whose object wraps or contains primitive data types.
3. When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types. In other words, we can wrap a primitive value into a wrapper class object.



Each of the 8 primitive types has corresponding wrapper classes.

Primitive Type Wrapper Class

byte ----- Byte

boolean ------ Boolean

char ---- Character

double --- Double

float --- Float

int --- Integer

long --- Long

short ---- Short

**Boxing:** when we convert primitive data type to object is called boxing.

code:

int x = 10;

Integer y = new Integer(x);

**Unboxing:** when object is converting in primitive data type is called unboxing.

**code:**

Interger y = new Interger(10);

int x = y.intValue();

**Auto-boxing:** The automatic conversion of primitive data type into its corresponding wrapper class is known as auto-boxing, 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 J2SE 5.0, auto-boxing and unboxing feature convert primitives into objects and objects into primitives automatically

**Method of wrapper class:**

1. typeValue() -Converts Number object to the stated primitive data type.
2. compareTo() – Compares the Number object with the argument.
3. equals()   – Checks whether the Number object equals the argument.
4. valueOf() – Returns an Integer object containing the specified primitive data type’s value
5. toString() – Returns a String object containing the specified Integer type argument’s value
6. parseInt() – Returns an Integer type value of a stated String representation
7. decode() – Decodes a String into an integer
8. min() – Compares the two arguments and returns the smaller value
9. max() – Compares the two arguments and returns the bigger value
10. round() – Returns the closest round off of int or long value according to the method return type.

#### Java Enum:

Enum (enumeration) is a type that has a fixed set of constant values. We use the enum keyword to declare enums.

code:

enum Size {

SMALL, MEDIUM, LARGE, EXTRALARGE

}

class Main {

public static void main(String [] args) {

System.out.println(Size.SMALL);

System.out.println(Size.MEDIUM);

}

}

Enum with the switch statement:

enum Size {

SMALL, MEDIUM

}

class Test {

Size pizzaSize;

public Test(Size pizzaSize) {

this.pizzaSize = pizzaSize;

}

public void orderPizza() {

switch(pizzaSize) {

case SMALL:

System.out.println("small size pizza.");

break;

case MEDIUM:

System.out.println("medium size pizza.");

break;

default:

System.out.println("which one to order.");

break;

}

}

}

class Main {

public static void main(String[] args) {

Test obj = new Test (Size.MEDIUM);

obj.orderPizza();

}

}

**Methods of Java Enum Class:**

1. ordinal(): The ordinal() method returns the position of an enum constant. Exp- ordinal(SMALL)
2. compareTo(): This method compares the enum constants based on their ordinal value.
3. toString() : convert enum to string. Exp- SMALL.toString()
4. name():
5. valueOf() :
6. values():

#### Java Reflection:

Reflection allows us to inspect and manipulate classes, interfaces, constructors, methods, and fields at run time.

There is a class in Java named Class that keeps all the information about objects and classes at runtime. The object of Class can be used to perform reflection.

## METHOD

1. A method is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation.
2. Method creates in class.
3. It is used to achieve the reusability of code
4. We write a method once and use it many times.
5. We can also easily modify code using methods.

**The syntax to declare a method is:**

1. Access Specifier:
2. Return Type
3. Method Name
4. Parameter List
5. Method Body



1. **Access Specifier/Modifier:** Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:
2. **Return Type**: It specifies what type of value a method returns for example if a method has an **int** return type then it returns an integer value. **If the method does not return a value, its return type** is void.
3. **Method Name**: It is an identifier that is used to refer to the method in a program.
4. **Method body**: It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the **curly braces {}.**

#### There are two types of methods:

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

**public static void findEvenOdd ( int num ){**

**If (num%2 == 0)**

**System.out.println(num+" is even");**

**else**

**System.out.println(num+" is odd");**

**}**

1. **Standard library Method**: In Java, Method that is already defined in the Java class libraries is known as **standard library method** or **built-in method**. We can directly use these methods just by calling them in the program at any point. Some pre-defined methods are-
2. **length()**
3. **equals()**
4. **compareTo()**

#### Other Method:

1. Static Method
2. Instance Method
3. Abstract Method
4. Factory Method
5. **Static Method:**
6. We can also create a static method by using the keyword **static** before the method name.
7. Static method is called without creating an object.
8. It is invoked by using the class name.
9. Static method can’t use non static member.
10. “this” and “super” keyword can’t be used here.
11. They are accessed by the class name and a dot (.) followed by the name of a method.

class boy{

int x =10;

static void show() {

System.out.println(10);

}

}

public class main {

public static void main(String[] args) {

boy.show();

}

}

1. **Instance Method/non-static :**
2. The simple method of the class is known as an **instance method**.
3. It is a **non-static** method defined in the class.
4. Before calling or invoking the instance method, it is necessary to create an object of its class.

**Code:**

public class ABC {

public static void main(String [] args){

ABC ob = new ABC();

System.out.println( ob.add(12, 13) );

}

public int add(int a, int b) {

return a + b ;

}

}

There are two types of instance method:

1. **Accessor/Getter Method:**  The method(s) that reads the instance variable(s) is known as the accessor method. We can easily identify it because the method is prefixed with the word **get**. **It is also known as getters**. It returns the value of the private field. It is used to get the value of the private field.

**public** **int** **getId**(){

**return** Id;

}

1. **Mutator/Setter Method:** The method(s) read the instance variable(s) and modify the values. We can easily identify it because the method is prefixed with the word **set**. **It is also known as setters or modifiers**. It does not return anything. It accepts a parameter of the same data type that depends on the field. It is used to set the value of the private field.

**public** **void** **setRoll**(**int** roll){

**this**.roll = roll;

}

Example of accessor and mutator method,

**public** **class** Student{

**private** **int** roll;

**public** **int** getRoll(){     //accessor method

**return** roll;

}

**public** **void** setRoll(**int** roll) { //mutator method

**this**.roll = roll;

}

**public** **void** display(){

System.out.println("Roll no: "+roll);

}

}

1. Abstract Method:
2. The method that does not has method body is known as abstract method.
3. There is no need to provide the implementation code while declaring it.
4. It always declares in the abstract class.
5. It declares with abstract key word.

**Syntax: abstract** **void** method\_name();

**Code:**

**abstract** **class** Demo{ //abstract class

**abstract** **void** display();

}

**public** **class** MyClass **extends** Demo{

**void** display(){

System.out.println("Abstract method?");

}

**public** **static** **void** main(String args[]){

Demo obj = **new** MyClass();

obj.display();

}

}

1. **Factory method:** it is a design pattern. When a class method returns an object is called factory method.

Class User{

Public void name;

Public User person(){

Return new User;

}

}

1. **Chaining method:**
2. Method chaining is the chain of methods being called one after another.
3. For method changing we return “this” variable in method in class.
4. The return type should be class type of the method.

**syntax :** obj.method1().method2().method3();

code:

class User{

     private float b;

     User input(float b) {

         this.b = b;

         return this;

     }

     void display() {

         System.out.println("Display=" + b);

     }

}

public class Example {

     public static void main(String[] args)  {

         new User().input(10).display();

     }

}

#### Java Method Parameters:

**Parameters:** Parameter is the value that is defined in a functional block.

**Arguments:** Argument is a variable whose value is passed into a function and is used whenever a function is called.

We can pass value in method as parameters.

public class Main {

static void boy(String fname) {

System.out.println( fname );

}

Static void add (int y, int x){

Int z = y + x;

System.out.println(z);

}

public static void main(String[] args) {

boy("Liam");

add(2,5);

}

}

#### Method Overloading in Java:

If a class has multiple methods having same name but different in parameters, it is known as Method Overloading. If we must perform only one operation, having same name of the methods increases the readability of the program.

**void func() { ... }**

**void func(int a) { ... }**

**float func(double a) { ... }**

**float func(int a, float b) { ... }**

There are two ways to overload the method in java

1. **Changing number of parameters**: we can overload the method by changing the number of parameters.

class MethodOverloading {

private static void display(int a){

System.out.println("Arguments: " + a);

}

private static void display(int a, int b){

System.out.println("Arguments: " + a + " and " + b);

}

public static void main(String[] args) {

display(1);

display(1, 4);

}

}

1. **Changing the data type of parameters:** if change function argument type then it also function overloading.

class MethodOverloading {

private static void display(int a){

System.out.println("Got Integer data.");

}

private static void display(String a){

System.out.println("Got String object.");

}

private static void display(int a, long b){

System.out.println("Arguments: " + a + " and " + b);

}

public static void main(String[] args) {

display(1);

display("Hello");

}

}

#### Method rules:

1. Method under method not possible.
2. Method should declare under a class.

Class Test {

Static void sum (){

Int x = 10;

Int y = 16;

}

Public static void main (String a[]){

sum (x+y);

}

}

1. Method should call with class. None static method cannot call directly.

Class Test {

Static void sum (){

Int z = 15+ 16;

}

Public static void main (String a[]){

Test t = new Test ();

t.sum()

}

}

#### Main method in java:

When we write a program, we write multiple code and class in program. But when we run the program, java do not know from where we run the code. So that we write an identifier class called main. If we write a main class in java program, java know that from here it run code. So, we write main method, and JVM run the main method. Main method is not pre-defined or user define. It is a contract between programmer and programing language.

class Main {

public static void main(String[] args) {

Main obj = new Main();

System.out.println("Value of i: " + obj);

}

}

1. We can run java program without main method.
2. We can called main method in every class.
3. We can change the main method name. (For change the main method name we should change the JVM main method name).
4. main method is static because if we do main method as a non-static then we want to make object of main method class. So memory consumption we make main method static.
5. main method is public because we can access main method anywhere of code.
6. main method is void because it is not return type. (in java main Method not Return any things. but in C main method return ending to compiler, so that it need return type . but java did not. )
7. In main method we can pass array type argument. Because in compiler time we pass string type argument in main method. It is command line argument.

**Called of Main method:** we can called method on class to another class.

class A{

public static void main(String[] arg){

}

}

class B {

public static void main(String[] p){

A.mian();

}

}

**argument pass in main method:**

class A{

public static void main(String[] arg){

}

}

class B {

String ar[] = {“heelo”};

public static void main(String[] p){

A.main();

A.main(“ar”);

A.main(null);

A.main(p)

}

}

#### Variable Arguments (Varargs):

The varrags allows the method to accept zero or muliple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach. the Varargs method, we can differentiate arguments by using Index. A variable-length argument is specified by three periods or dots(…).

Syntax:

public static void fun (int ... a)

{

// method body

}

Code:

public class Demo {

public static void Varargs(String... str) {

      System.out.println("Number “ + str.length);

      System.out.println("The argument values are: ");

      for (String s : str)

         System.out.println(s);

 }

public static void main (String args []) {

Varargs("Apple", "Mango", "Pear");

Varargs();

Varargs("Magic");

}

}

## CONSTRUCTORS:

1. A constructor in Java is a special method that is used to initialize objects.
2. The constructor is called when an object of a class is created.
3. It can be used to set initial values for object attributes.
4. Java compiler creates a default constructor when object create.
5. The name of the constructor should be the same as the class.
6. A Java constructor must not have a return type.

**code:**

**public class Puppy {**

**public Puppy() {**

**}**

**}**

**Types of Constructors:**

1. **No-argument constructor:**  A constructor that has no parameter is known as the No-argument or Zero argument constructor.

class Main {

int i ;

private Main() {

i = 5;

System.out.println("Constructor is called");

}

public static void main(String[] args) {

Main obj = new Main();

System.out.println("Value of i: " + obj.i);

}

}

1. **Parameterized Constructor:** A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.

class Main {

String languages;

Main(String lang) {

languages = lang;

System.out.println(“Programming Language");

}

public static void main(String[] args) {

Main obj1 = new Main("Java");

Main obj2 = new Main("Python");

Main obj3 = new Main("C");

}

}

1. **Default Constructor:** If we do not create any constructor, the Java compiler automatically create a no-argument constructor during the execution of the program. This constructor is called default constructor.

Access modifier of constructor depends on class. If class public then constructor public.

class User{

User(){

}

}

**Construction overloading:** We create multiple construction with different parameter.

class User{

User(){

System.out.println(“zero”);

}

User( int a ){

System.out.println(“one”);

}

User( int a, int b){

System.out.println(“two”);

}

public static void main(String[] args){

new User();

new User(12);

}

}

## Block /initializer

A block in Java is a set of code enclosed within curly braces { } within any class, method, or constructor. It begins with an opening brace “{“ and ends with an closing braces “}” .Between the opening and closing braces, we can write codes which may be a group of one or more statements.

**Syntax:**

**public class{**

**……. code**

**}**

****

There are three types of blocks in Java. They are as follows:

1. Local block
2. **Static block: -** object independent
3. **Non static block (init block): -**object dependent
4. **Static block:**
5. static keyword with a block of code that is known as a static block.
6. A static block can have several instructions that always run when a class is loaded into memory.
7. It is also known as java static initializer block because we can initialize the static variables in the static block at runtime.
8. A class can have any number of static blocks, The JVM executes them in the sequence in which they have been written.
9. A static block only executes for a single time for the life cycle of the program.
10. The static block in a program is always executed first before any static method, non-static method, main method, or even instance block.

syntax:

static {

// Body of static block

}

code:

public class MainClass{

static int b;

public static void show(){

System.out.println("Static method");

}

static{

b = 5;

show();

}

public static void main(String arg[]){

System.out.println("Main method");

}

}

1. **Non static block( Initialization Block):**
2. If you declare a block without any static keyword, then it is a Non-Static Block.
3. Non-static blocks are executed before the constructor is called by the user.
4. It is object dependent.
5. Non static block is known as initialization block.
6. The initializer block contains the code that is always executed whenever an instance is created and it runs each time when an object of the class is created.
7. We do not keep non-static block inside a method. non-static block inside the method thread as normal statement

syntax:

**{**

**…. code**

**}**

**code:**

**class User{**

**{ System.out.println(“hello “); }**

**public static void main (String [] args){**

**new User ();**

**}**

**}**

## Objects

Let us now look deep into what are objects. If we consider the real-world, we can find many objects around us, cars, dogs, humans, etc. All these objects have a state and a behavior. If we consider a dog, then its state is - name, breed, color, and the behavior is - barking, wagging the tail, running. If you compare the software object with a real-world object, they have very similar characteristics. Software objects also have a state and a behavior. A software object's state is stored in fields and behavior is shown via methods. So, in software development, methods operate on the internal state of an object and the object-to-object communication is done via methods.

#### Creating an Object

1. **By new Keyword**: The new keyword is used to create objects. new keyword is use for memory allocation in ram. so that new keyword uses for object keyword.

Syntax:

new classname();

classname variable = new classname();

code:

Public class User{

String name;

Int id;

}

new User();

User shuvo = new User();

1. **By Factory Pattern:**
2. **By Clone Method:**
3. **newInstance method**:

#### Object has three characteristics:

1. **State:** represents the data (value) of an object.
2. **Behavior:** represents the behavior (functionality) of an object such as deposit, withdraw, etc.
3. **Identity:** An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.

#### Memory allocation for object:

1. new keyword allocates memory to the new objects at runtime.
2. Memory is allocated to the new objects from the heap memory.
3. The new keyword is followed by a call to a constructor, which instantiates the new object.
4. Construction is allocated the size of memory.

**There are two type of memory allocation for object.**

**STATIC (COMPILE TIME):**

1. When memory allocation occurred in compile time is called static or compile time memory allocation.
2. In compile time memory allocation not occurred. because if compile time memory allocation occurred then we not use this memory other computer.
3. Static memory is always referring via their name.

Int x = 10;

Here x is 10. When we need 10, we called x.

**DYNAMIC (RUN TIME):**

1. when memory is allocated in run time is called dynamic allocation. Dynamic memory is referring by pointer.
2. java does not support pointer explicitly. But implicitly it is used.



1. To use object multiple time, we save reference in reference variable.

**Public class User {**

**String name;**

**Int id;**

**public static void main (String [] args) {**

**User obj = new User ();**

**Obj.name = “shuvo”;**

**}**

**}**

#### Default value of Object:

When we create an object with new keyword, memory allocation in ram. So this time all variable set default value of this object.

1. For string value default value = null
2. For int value default value = 0

#### Anonymous Object:

1. An anonymous object is basically **a value that has been created but has no name.**
2. when creating an object in Java, you need to assign a name to the object. But anonymous object only uses when we create.

**Public class User {**

**String name;**

**Int id;**

**public static void main (String [] args) {**

**new User ();**

**}**

**}**

1. Every time anonymous object called, every time new object create and new ref value create.

code:

**Public class User {**

**String name;**

**Int id;**

**Public function show(){**

**System.out.println(name);**

**}**

**public static void main (String [] args) {**

**new User ().name=” shuvo”;**

**new User (). show ();**

**new User ().id=12364;**

**}**

**}**

#### Object Reference:

1. When object create a unique id create for object which is reference id. The reference id stores those variables called reference variable.
2. **if we** use null as object reference then garbage collection deletes the memory for object which do not have ref vary.
3. In java we can give on object Reference to another object. If we change one object value, another object will change.

**Public class User {**

**public static void main (String [] args) {**

**User obj1 = new User ();**

**User obj2 = new User ()**

**obj2= obj1**

**}**

**}**

1. **This Keyword:**

In data shadowing we face the problem for use intense variable in class. to solve this problem, we use this key word for reference variable.

For solving the shadowing problem, we pass object reference in method for use instance variable .

class User{

int x = 10;

void show(User ob){

int x = 100;

int ob.x = 20;

}

public static void main(String ss[]){

User Shuvo = new User();

shuvo.show(shuvo):

}

}

But it this very complex in programing . So, remove this complexity java introduced “this” keyword which is reference variable.

1. “this ” is implicit reference variable hold the reference id of that object.
2. “this” keyword is public final
3. “this” cannot use in static method and static block.

class Use{

int x = 10;

void show(){

int x = 100;

int this.x = 20;

}

public static void main(String ss[]){

User Shuvo = new User();

shuvo.show():

}

}

**uses of this:**

1. we use this for object refarance
2. method chaining: we use this in method chaining

class User{

     private float b;

     User input(float b) {

         this.b = b;

         return this;

     }

     void display() {

         System.out.println("Display=" + b);

     }

}

public class Example {

     public static void main(String[] args)  {

         new User().input(10).display();

     }

}

1. this can call static and non-static method.

Code:

class User{

int a ;

     void input(int a) {

this.a = a;

     }

     void show() {

        System.out.println(a);

     }

}

public class Example {

     public static void main(String[] args)  {

         new User()

     }

}

**constructor chaining**:

We called multiple constructors with “this” keyword. for this we called this() method and pass argument which constructor want to called. “this” should called first line in method.

Code:

class User{

     void User() {

this(2);

        System.out.println();

     }

     void User(int a) {

this(2,3)

        System.out.println(a);

     }

     void User(int a, int b) {

        System.out.println(a+ b);

     }

}

public class Example {

     public static void main(String[] args)  {

         new User()

     }

}

## Association/Reusability of class -- 70

1. Association is a relation between two separate classes which establishes through their Objects.
2. Association can be one-to-one, one-to-many, many-to-one, many-to-many.
3. In Object-Oriented programming, an Object communicates to another object to use functionality and services provided by that object.

Types of Association

1. IS-A Association
2. HAS-A Association



**IS-a relationship:**

1. Whenever one class inherits another class, it is called an IS-A relationship.
2. it known as inheritance in java.

code:

public class Address {

String city;

int road;

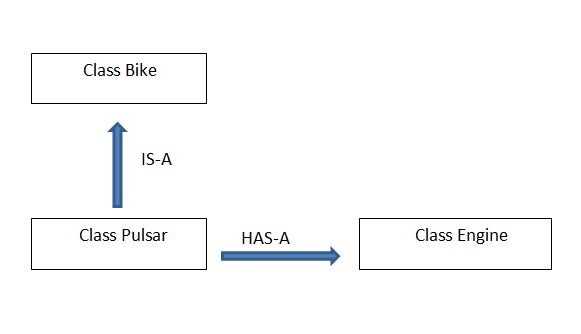
}

public class Person extends Address {

Address add = new Address ();

add.city = “dhaka”;

}

****

**HAS-A relationship:**

1. When an object of one class is created as a data member inside another class, it is called Has-A relationship.
2. Association is a relation between two separate classes which establishes through their Objects.
3. There are two types of Has-A relationship
4. Composition(Strong)
5. Aggregation(Weak)

code:

**public class Address {**

**String city;**

**int road;**

**}**

**public class Person { // Person has-a Address.**

**Address add = new Address ();**

**add.city = “dhaka”;**

**}**

1. **Aggregation(weak):**

* It represents Has-A’s relationship.
* It is a unidirectional association. For example, a department can have students but vice versa is not possible.
* In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity.

Example:- a library class extends s**tudent’s class**. If the **library** is destroyed, students will

exist without library.

1. **Composition(Strong):**

* It represents part-of relationship.
* In composition, both entities are dependent on each other.
* When there is a composition between two entities, the composed object cannot exist without the other entity.

Example:- a class Car cannot exist without class Engine.

## Inheritance ---71

1. Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another.
2. With the use of inheritance, the information is made manageable in a hierarchical order.
3. The class which inherits the properties of other is known as subclass/Child Class (derived class, child class) and the class whose properties are inherited is known as superclass /Parent Class (base class, parent class).
4. Inheritance represents the IS-A relationship which is also known as a parent-child relationship.
5. extends is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

Syntax:

class A {

………….

}

class Sub extends A {

………….

}

code:

class Parent {

String name;

public void eat() {

System.out.println("I can eat");

}

}

class Child extends Parent {

public void display () {

System.out.println( name);

}

public static void main(String[] args) {

Child ob = new Child();

ob.name = "shuvo";

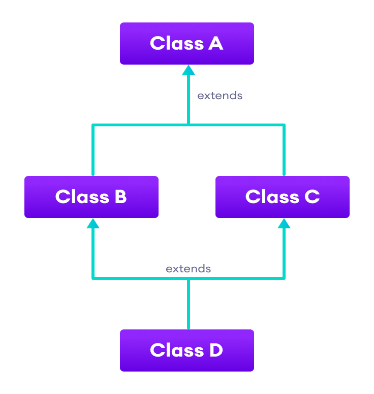
ob.eat();

}

}

**Types of Inheritance:**





Java Hybrid Inheritance

#### Data Hiding:

1. When in program, the name of parent class data variable and the name of child class data variable are same, then program take child class data as main variable, this concept is called data hiding.
2. Whenever data hiding is done always preference goes to child class data.

class Parant {

String name;

}

class Child extends Parant {

String name;

public void show () {

System.out.println( name );

}

public static void main(String[] args) {

Child obj = new Child ();

obj.show();

}

}

#### Memory Allocate for Inherence :

1. In java, parent class object not create when child class object creates. So how can allocate memory for parent class.
2. In java all parent class memory is allocated in child class memory location when child class object was created.



#### Upcasting:

1. Upcasting is typecasting in which a child object is typecasted to a parent class object.
2. Upcasting, we can easily access the variables and methods of the parent class to the child class.
3. Upcasting is also known as Generalization and Widening.
4. With upcasting we hold child object in parent reference variable.

**Parent to child:**

class Parent {

int x = 444;

}

class Child extends Parent {

int x = 10

void show(){

System.out.println(“hello ”);

}

}

class Upcasting{

   public static void main(String args[]) {

Parent obj = new Child ();

       obj.show(obj);

   }

}

**Grand-parent to child:**

class Grandparent{

int x = 9999;

}

class Parent extend Grandparent {

int x = 444;

}

class Child extends Parent {

int x = 10

public void show (**Parent objP, Grandparent objG** ) {

System.out.println(x) ;

System.out.println( objP.x );

System.out.println( objG.x );

}

public static void main(String args[]) {

Child obj =  new Child();

       obj.show(obj, obj );

   }

}

#### Downcasting:

downcasting is a type of object typecasting in which a Parent object is typecasted to a child class object.

class Parent {

int x = 444;

}

class Child extends Parent {

int x = 10

display(){

System.out.println(“dkfj”);

}

  public static void main(String args[]) {

Parent obj =  new Child();

**Child ob2 = (Child) obj;**

    }

}

#### Super:

1. If we want access the parent class data variable, we can do it by **super** keyword.
2. The super keyword in java is a reference variable that is used to refer parent class objects.
3. Super only call the immediate parent data not parent of parent.
4. Super cannot use inside static context

class Parant {

String name;

}

class Child extends Parant {

String name;

public void show () {

System.out.println(name);

System.out.println(**super.name);**

}

public static void main (String [] args) {

Child obj = new Child ();

obj.show();

}

}

#### Method overriding:

1. if the same method is defined in both the superclass and the subclass, then the method of the subclass class overrides the method of the superclass. This is known as method overriding.
2. The method must have the same name as in the parent class
3. The method must have the same parameter as in the parent class.
4. There must be an IS-A relationship (inheritance).
5. Method overrides access modifier always weaker to stronger. Public -> protected-> default->private

**code:**

class Vehicle{

   void run(){

System.out.println("Vehicle is running");

}

}

class Bike2 extends Vehicle{

   void run(){

System.out.println("Bike is running safely");

}

   public static void main(String args[]){

Bike2 obj = new Bike2();

 obj.run();

   }

}

Method overriding rule:

#### Covariant Return:

1. we can method override with change return type (it is come in after jdk 1.5). it is called covariant type.
2. Both class method must have return type is reference type whereas primitive data type not allowed.
3. The classes which used as return type they also must have relation of parent and child.

class Vehicle{

   void run(){

System.out.println("Vehicle is running");

}

}

class Bike2 extends Vehicle{

  Vehicle  run(){

System.out.println("Bike is running safely");

}

   public static void main(String args[]){

Bike2 obj = new Bike2();

 obj.run();

   }

}

**Shadowing:**

Shadowing is a concept of OOP (Object Oriented Programming) paradigm. Using Shadowing we can provide a new implementation to base class member without overriding it, which means that original implementation of base class member gets shadowed (hidden) with the new implementation of base class member provided in derived class.

when in a program the name of local variable and the name of instance variable is same, this concept is called data shadowing. In data shadowing preference goes to local variable.

code:

class User{

int x = 10;

void show(){

int x = 20;

}

public static void main(String ss[]){

User Shuvo = new User();

shuvo.show():

}

}

#### Method hiding:

When static method, of parent class override in static method of child class is called method hiding.

class Vehicle{

   void static run(){

System.out.println("Vehicle is running");

}

}

class Bike2 extends Vehicle{

   Vehicle static  run(){

System.out.println("Bike is running safely");

}

   public static void main(String args[]){

Bike2 obj = new Bike2();

 obj.run();

   }

}

#### Constructor chaining of inheritance:

1. In java before executing any child class constructor firstly execute its immediate parent class default constructor. (parameterize cannot execute). Because Compiler added super at the top of every class.
2. In class, super () method only called, if parent class have default constructor. So if parent class have not default construction then we should pass parameterize super (23) for parameterize constructor called.
3. this and super both cannot stay same constructor.
4. for data initialize the of parent, this rule creates.
5. If parameterize constructor called then super should called.

Class K{

K(int x){

System.out.println(“constructor K”);

}

}

Class L extends K{

Super(22);

L(){

System.out.println(“constructor L”);

}

}

Public class B extends L{

B(){

System.out.println(“constructor B”);

}

}

## Encapsulation ----- 87

**OPP Encapsulation:** in OOPs is the concept of binding fields (object state) and methods (behavior) together as a single unit.  Another way to think about encapsulation is, that it is a protective shield that prevents the data from being accessed by the code outside this shield.

**JAVA Encapsulation:** Programming languages such as Java use encapsulation with help of classes.  So that as aspect of java, Encapsulation is the process of wrapping the data member and member function within a class and create a protective barrier around them.

To achieve encapsulation in Java −

1. Declare the variables of a class as private.
2. Provide public setter and getter methods to modify and view the variables values.

code:

**class** Account {

**private** String name ,password;

**public** **void** **setName(String name) {**

**this**.name = name;

}

**public** **String getName()** {

**return** name;

}

**public** **void** **setPassword() {**

**this**.password = password;

}

**public** **String** **getPassword ()** {

**return password;**

}

}

**public** **class** TestEncapsulation {

**public** **static** **void** main(String[] args) {

Account user=**new** Account();

**user.setName(“shuvo”);**

**user.setPassword(“123”);**

System.out.println(**user.getName()+"** "+**acc.getPassword** ());

}

}

**Advantages of Encapsulation:**

1. **Data Hiding:** it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class. It will not be visible to the user how the class is storing values in the variables. The user will only know that we are passing the values to a setter method and variables are getting initialized with that value.
2. **Increased Flexibility:** We can make the variables of the class read-only or write-only depending on our requirement. If we wish to make the variables read-only then we have to omit the setter methods like setName(), setAge(), etc. from the above program or if we wish to make the variables write-only then we have to omit the get methods like getName(), getAge(), etc. from the above program
3. **Reusability:** Encapsulation also improves the re-usability and is easy to change with new requirements.
4. **Testing code is easy:** Encapsulated code is easy to test for unit testing

## Binding –91

Binding is a mechanism creating link between method call and method actual implementation.

Code:

Class User{

Void show(){ // method implementation

System.out.println(“ show ”);

}

Public void static main(String ss[]){

User shuvo = new User();

shuvo.show(): //method called.

}

}

**Static Binding (Compile Binding / Early binding):**

1. When a member called getting/linking their member definition in compile time, it called static binding.
2. compile time building do compiler.
3. Static binding done based on type of reference.
4. All things without non-static method ( virtual method/void show () ) is static binding.
5. This is done using static, private and, final methods, constructor, initializers, method call via super keyword.

**class** Parent {

**int x = 444;**

**static int y = 20;**

}

**class** Child **extends** Parent {

**int x = 10;**

**static int y = 20;**

**static show(){**

**System.out.println(x)**

**}**

**}**

**class** Dynamic {

**public** **static** **void** main(String args[]) {

      Parent obj = new Child();

      obj.show();  // invoke virtual

    }

}

**Dynamic Binding (Runtime Binding / late binding):**

1. When member call getting/linking their member definition at runtime it called runtime binding.
2. Run time building do JVM.
3. Runtime binding done on type of object.
4. Non-static/virtual method is bind in dynamic.

**class** Parent extend Grand\_Parent {

**int x = 444;**

}

**class** Child **extends** Parent {

**int x = 10**

**}**

**class** Dynamic {

**public** **static** **void** main(String args[]) {

      Parent obj = new Child ();   // invoke static

      obj.show();

    }

}

## Polymorphism ---96

1. Polymorphism means more than one form. the same entity (method or operator or object) can perform different operations in different scenarios.
2. A person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So, the same person possesses different behavior in different situations. This is called polymorphism.
3. Polymorphism always achieve on method not on data member.
4. We can perform polymorphism in java by method overloading and method overriding.

In Java polymorphism is mainly divided into two types:

1. Compile-time Polymorphism
2. Runtime Polymorphism

#### Compile-time Polymorphism:

1. Compile Time Polymorphism is also known as Static Polymorphism.
2. Furthermore, the call to the method is resolved at compile-time.
3. Compile-Time polymorphism is achieved through Method Overloading, constructor overloading

public class Addition {

void sum(int a, int b) {

int c = a+b;

System.out.println(“ Addition of two numbers :” +c);

}

void sum(int a, int b, int e) {

int c = a+b+e;

System.out.println(“ Addition of three numbers :” +c);

}

public static void main(String[] args) {

Addition obj = new Addition();

obj.sum ( 30,90);

obj.sum(45, 80, 22);

}

}

#### Runtime Polymorphism:

1. It is also known as Dynamic Method Dispatch.
2. It is a process in which a function call to the overridden method is resolved at Runtime.
3. This type of polymorphism is achieved by Method **Overriding** and **Upcasting**.

We can achieve polymorphism in Java using the following ways:

1. Method Overriding
2. Method Overloading
3. Operator Overloading
4. **Method Overriding:** If subclass (child class) has the same method as declared in the parent class, it is known as **method overriding in Java.**

class Animal {

public void move() {

System.out.println("Animals can move");

}

}

class Dog extends Animal {

public void move() {

System.out.println("Dogs can walk and run");

}

}

public class TestDog {

public static void main(String args[]) {

Animal a = new Animal(); // Animal reference and object

Animal b = new Dog(); // Animal reference but Dog object

a.move(); // runs the method in Animal class

b.move(); // runs the method in Dog class

}

}

1. Method OverLoading: it discusses in previous in method section.
2. Operator overloading: in java operator overloading not possible.

**Rules for Method Overriding:**

1. The argument list should be the same as that of the overridden method.
2. The return type should be the same or a subtype of the return type declared in the original overridden method in the superclass.
3. The access level cannot be more restrictive than the overridden method's access level. For example: If the superclass method is declared public then the overriding method in the sub class cannot be either private or protected.
4. Instance methods can be overridden only if they are inherited by the subclass.
5. A method declared final cannot be overridden.
6. A method declared static cannot be overridden but can be re-declared.
7. If a method cannot be inherited, then it cannot be overridden.
8. A subclass within the same package as the instance's superclass can override any superclass method that is not declared private or final.
9. A subclass in a different package can only override the non-final methods declared public or protected.
10. Constructors cannot be overridden.

## Abstract class ---102

**Abstraction:**

1. Abstraction is a process of hiding the implementation details and showing only functionality to the user.
2. it shows only essential things to the user and hides the internal details.

Example: IF sending SMS where you type the text and send the message. You don't know the internal processing about the message delivery.

There are two ways to achieve abstraction in java

1. Abstract class (0 to 100%)
2. Interface (100%)
3. **Abstract Class:**
4. An abstract class must be declared with an abstract keyword.
5. Abstract classes may or may not contain abstract methods.so that it is (0-100) % abstract method.
6. if a class has at least one abstract method, then the class must be declared abstract.
7. We cannot create objects of abstract classes
8. To use an abstract class, you must inherit it from another class.
9. If you inherit an abstract class, you must override to all the abstract methods in It.

**Code:**

abstract class Employee {

private int number;

abstract public Employee(String name) {

this.name = name;

}

double computePay() {

System.out.println("Inside Employee computePay");

}

}

public class Main {

public static void main(String [] args) {

Employee e = new Employee("Shuvo”);

e.computerPay();

}

}

**Method**:

1. A method which is declared as abstract and does not have implementation is known as an abstract method.
2. An abstract method contains a method name, but no method body in abstract class.
3. Instead of curly braces, an abstract method will have a semi colon (;)
4. abstract method cannot private.
5. abstract method cannot static. Because abs
6. final.
7. if any method declared in abstract class, then extends class must be override the method.
8. If abstract method declared in class and class is not declared abstract class then, class will be abstract class.
9. Abstract method can be creating construction.

code:

abstract class Bike {

  abstract void run ();

}

class Honda4 extends Bike {

void run (){

System.out.println("running safely");

}

public static void main (String args[]){

 Bike obj = new Honda4();

 obj.run();

}

}

## Interface ----- 103

1. The interface in Java is a mechanism to achieve abstraction.
2. An interface in Java is a blueprint of a class
3. We use the implements keyword to implement an interface.
4. An interface does not contain any constructors.
5. An interface is not extended by a class; it is implemented by a class.
6. An interface can extend multiple interfaces.
7. When a class implements an interface then class must override all the method of interface with public access privilege otherwise class has been abstracted.
8. We cannot create objects of an interface.
9. An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
10. Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
11. Methods in an interface are implicitly public.
12. Since Java 8, we can have default and static methods in an interface.
13. Since Java 9, we can have private methods in an interface.

**Syntax:**

interface <interface\_name>{

     // declare constant fields

}

**code:**

interface Player {

final int id = 10;

int move ();

}



**Multiple inheritance in Java by interface:**

If a class implements multiple interfaces, or an interface extends multiple interfaces, it is known as multiple inheritance.



Code:

interface Printable{

void print();

}

interface Showable{

void show();

}

class A7 implements Printable,Showable{

public void print() { System.out.println("Hello"); }

public void show() { System.out.println("Welcome"); }

public static void main(String args[]){

A7 obj = new A7();

obj.print();

obj.show();

 }

}

# Modifier / keyword

Access specifier or modifier is the access type of the method. It specifies the visibility of the method. Java provides four types of access specifier:

#### Public:

The method is accessible by all classes when we use public specifier in our application.

#### Private:

When we use a private access specifier, the method is accessible only in the classes in which it is defined.

#### Protected:

When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.

#### Default:

When we do not use any access specifier in the method declaration, Java uses default access specifier by default. It is visible only from the same package only.

#### Static:

1. The static keyword in Java is mainly used for memory management.
2. The static keyword in Java is used to share the same variable or method of a given class.
3. The users can apply static keywords in –
4. variables,
5. methods,
6. blocks, and
7. nested classes.
8. we can use static before class, interface constructor.
9. When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
10. **Java static variable:** If you declare any variable as static, it is known as a static variable. The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.The static variable gets memory only once in the class area at the time of class loading.

class Student{

     int roll;

     String name;

     String college="ITS";

}

1. **Java static method:** When a method is declared with the static keyword, it is known as the static method. The most common example of a static method is the main( ) method. Any static member can be accessed before any objects of its class are created, and without reference to any object. static method represents the behavior of class. non-static method represent of object.
2. They can only directly call other static methods.
3. They can only directly access static data.
4. They cannot refer to this or super in any way.

class Test{

    static int a = 10;

    int b = 20;

    static void m1(){

        a = 20;

        System.out.println("from m1");

        b = 10; // compilation error

        m2();  // compilation error

        System.out.println(super.a); // compiler error

    }

    void m2() {

        System.out.println("from m2");

    }

    public static void main(String[] args)    {

        // main method

    }

}

1. **Static Classes:** A class can be made static only if it is a nested class. We cannot declare a top-level class with a static modifier but can declare nested classes as static. Such types of classes are called Nested static classes. Nested static class doesn’t need a reference of Outer class. In this case, a static class cannot access non-static members of the Outer class.

import java.io.\*;

public class GFG {

    private static String str = "GeeksforGeeks";

    static class MyNestedClass {

        public void disp(){

          System.out.println(str);

        }

    }

    public static void main(String args[])  {

        GFG.MyNestedClass obj = new GFG.MyNestedClass();

        obj.disp();

    }

}

#### Final keyword :

1. It is the keyword that used to restrict the user.
2. It is used in variable method, class.
3. Final method cannot override.
4. Final class cannot inheritance.
5. Final variable cannot change.
6. **Final class:**

final class A {

// methods and fields

}

class B extends A {

// COMPILE-ERROR! Can't subclass A

}

1. **Final method:**

class A {

final void m1() {

System.out.println("This is a final method.");

}

}

class B extends A {

void m1() {

// Compile-error! We cannot override

System.out.println("Illegal!");

}

}

1. **Final variable:**
2. If you make any variable as final, you cannot change the value of final variable (It will be constant). This also means that you must initialize a final variable.
3. A final variable that is not initialized at the time of declaration is known as blank final variable.
4. If the final variable is a reference, this means that the variable cannot be re-bound to reference another object, but the internal state of the object pointed by that reference variable can be changed i.e. you can add or remove elements from the final array or final collection.

class GFG {

    final int THRESHOLD = 5;

    final int CAPACITY;

    final int MINIMUM;

    static final double PI = 3.141592653589793;

    static final double EULERCONSTANT;

    {

         CAPACITY = 25;

    }

    static {

         EULERCONSTANT = 2.3;

    }

    public GFG () {

         MINIMUM = -1;

    }

}

# Package of java

PACKAGE in Java is a collection of classes, sub-packages, and interfaces. It helps organize your classes into a folder structure and make it easy to locate and use them. More importantly, it helps improve code reusability. Each package in Java has its unique name and organizes its classes and interfaces into a separate namespace, or name group.

Create a package:

package p1;

class c1(){

public void m1(){

System.out.println("m1 of c1");

}

public static void main(string args[]){

c1 obj = new c1();

obj.m1();

}

}

Use package :

import package.name.Class; // Import a single class

import package.name.\*; // Import the whole package

**Sub packages**: Packages that are inside another package are the sub packages. These are not imported by default; they must import explicitly. Also, members of a sub package have no access privileges, i.e., they are considered as different package for protected and default access specifiers.

Types of packages:

1. Built-in Packages
2. User-defined packages

# Java Exceptions --- 115

An exception is an unexpected event that occurs during program execution. It affects the flow of the program instructions which can cause the program to terminate abnormally.

An exception can occur for many reasons. Some of them are:

1. Invalid user input
2. Device failure
3. Loss of network connection
4. Physical limitations (out of disk memory)
5. Code errors
6. Opening an unavailable file

**Java Exception hierarchy:**



#### Errors:

Errors represent irrecoverable conditions such as

1. Java virtual machine (JVM) running out of memory,
2. memory leaks,
3. stack overflow errors,
4. library incompatibility,
5. infinite recursion, etc.

Errors are usually beyond the control of the programmer, and we should not try to handle errors.

#### Exceptions:

Exceptions can be caught and handled by the program. When an exception occurs within a method, it creates an object. This object is called the exception object. It contains information about the exception such as the name and description of the exception and state of the program when the exception occurred.

Java Exception Types:

1. Runtime Exception
2. IO Exception.
3. **Runtime Exception:** A runtime exception happens due to a programming error. They are also known as unchecked exceptions. These exceptions are not checked at compile-time but run-time. Some of the common runtime exceptions are:
4. Improper use of an API - IllegalArgumentException
5. Null pointer access (missing the initialization of a variable) - NullPointerException
6. Out-of-bounds array access - ArrayIndexOutOfBoundsException
7. Dividing a number by 0 – Arithmetic Exception
8. **IO Exception:** An IOException is also known as a checked exception. They are checked by the compiler at the compile-time and the programmer is prompted to handle these exceptions. Some of the examples of checked exceptions are:
9. Trying to open a file that doesn’t exist results in FileNotFoundException
10. Trying to read past the end of a file.

#### Java Exception Handling:

1. try...catch block
2. finally block
3. throw and throws keyword
4. **try...catch:** The try...catch block in Java is used to handle exceptions and prevents the abnormal termination of the program.

try {

int divideByZero = 5 / 0;

System.out.println("Rest of code in try block");

}

catch (ArithmeticException e) {

System.out.println("ArithmeticException => " + e.getMessage());

}

1. **finally block:** finally, block is always executed no matter whether there is an exception or not. The finally block is optional. And, for each try block, there can be only one finally block.

The basic syntax of finally block is:

try {

//code

}

catch (ExceptionType1 e1) {

// catch block

}

finally {

// finally block always executes

}

1. **throw and throws:** The throw statement allows you to create a custom error.

class ThrowExcep{

    static void fun() {

        try {

            throw new NullPointerException("demo");

        }

        catch(NullPointerException e) {

            System.out.println("Caught inside fun().");

            throw e; // rethrowing the exception

        }

    }

    public static void main(String args[]) {

        try{

            fun();

        }

        catch(NullPointerException e){

            System.out.println("Caught in main.");

        }

    }

}

# FILE/IO handling

Java I/O (Input and Output) is used to process the input and produce the output.

Stream: A stream can be defined as a sequence of data. There are two kinds of Streams −

1. Byte Stream
2. Character Stream

Byte Stream: Byte stream is used to read and write a single byte (8 bits) of data.

1. [Java InputStream Class](https://www.programiz.com/java-programming/inputstream)
2. [Java OutputStream Class](https://www.programiz.com/java-programming/outputstream)



Character Stream:

Character stream is used to read and write a single character of data. Java **Character** streams are used to perform input and output for 16-bit Unicode.

All the character stream classes are derived from base abstract classes Reader and Writer.

1. FileReader
2. FileWriter

File handling methods:

Method Type Description

canRead() Boolean Tests whether the file is readable or not

canWrite() Boolean Tests whether the file is writable or not

createNewFile() Boolean Creates an empty file

delete() Boolean Deletes a file

exists() Boolean Tests whether the file exists

getName() String Returns the name of the file

getAbsolutePath() String Returns the absolute pathname of the file

length() Long Returns the size of the file in bytes

list() String[] Returns an array of the files in the directory

mkdir() Boolean Creates a directory

File class :

1. file
2. fileReader
3. fileWrite
4. fileinputstream
5. fileoutputstream
6. bufferinputstream
7. bufferoutputstream

# Generics

Generics means parameterized types. The idea is to allow type (Integer, String, … etc., and user-defined types) to be a parameter to methods, classes, and interfaces. Using Generics, it is possible to create classes that work with different data types. An entity such as class, interface, or method that operates on a parameterized type is a generic entity.

1. We can hold only a single type of objects in generics. It does not allow to store other objects.
2. There is no need to typecast the object.
3. It is checked at compile time so problem will not occur at runtime.
4. We can generic in all interface of collection.

BaseType <Type> obj = new BaseType <Type>()

**Type Parameters in Java Generics**

The type parameters naming conventions are important to learn generics thoroughly. The common type parameters are as follows:

1. T – Type
2. E – Element
3. K – Key
4. N – Number
5. V – Value

**Types of Java Generics**

1. **Generic Method:** Generic Java method takes a parameter and returns some value after performing a task. It is exactly like a normal function; however, a generic method has type parameters that are cited by actual type. This allows the generic method to be used in a more general way. The compiler takes care of the type of safety which enables programmers to code easily since they do not have to perform long, individual type castings.

class Test {

    static <T> void genericDisplay(T element) {

         System.out.println(element.getClass().getName() + " = " + element);

    }

    public static void main(String[] args){

         genericDisplay(11);

         genericDisplay("GeeksForGeeks");

         genericDisplay(1.0);

    }

}

1. **Generic Classes:** A generic class is implemented exactly like a non-generic class. The only difference is that it contains a type parameter section. There can be more than one type of parameter, separated by a comma. The classes, which accept one or more parameters, ​are known as parameterized classes or parameterized types.

class Test<T> {

     T obj;

     Test( T obj) { this.obj = obj; }

     public T getObject() { return this.obj; }

}

class Main {

    public static void main(String[] args)    {

        Test<Integer> Obj1 = new Test<Integer>(15);

        System.out.println( Obj1.getObject() );

        Test<String> Obj2  = new Test<String>("GeeksForGeeks");

        System.out.println(Obj2.getObject());

        Obj1 = Obj2;

    }

}

# Lambda Expression

When we implement interface with anomious

Syntax:

( argument )->{ body }

# Collections Framework

1. The Collection in Java is a framework that provides an architecture to store and manipulate the group of objects.
2. Java Collections can achieve all the operations that you perform on a data such as searching, sorting, insertion, manipulation, and deletion.

**Hierarchy of Collection Framework:** Let us see the hierarchy of Collection framework. the **java.util** package contains all the classes and  Interfaces for the Collection framework.



**Java collection framework has multiple interface**

1. Collection interface
2. Iterator interface
3. Enumeration interface
4. Map interface

## Collection Interface

1. The Collection interface is the root interface of the collection’s framework hierarchy.
2. Java does not provide direct implementations of the Collection interface but provides implementations of its sub interfaces like List, Set, and Queue.



Sub interfaces of the Collection Interface

1. List Interface
2. Queue Interface
3. Set Interface

### List Interface:

1. The List interface is an ordered collection that allows us to store and access elements sequentially.
2. It extends the Collection interface.
3. Since List is an interface, we cannot create objects from it.
4. we must import **java.util.List** package in order to use List.
5. class of List interface ------------
6. ArrayList class
7. LinkedList class
8. Vector class
9. Stack class

#### ArrayList Class:

1. The ArrayList class of the Java collections framework provides the functionality of resizable-arrays.
2. ArrayList provides us with dynamic arrays in Java
3. Creating an ArrayList Before using ArrayList, we need to import **the java.util.ArrayList** package first.

**Syntax: ArrayList <Type> arrayList= new ArrayList<>();**

**code:**

import java.util.ArrayList;

**class Main {**

**public static void main(String[] args)** {

**ArrayList<String> user= new ArrayList<>();**

**user. add("shuvo");**

**user.add("korim");**

**for (String p : user) {**

**System.out.print(p);**

**System.out.print(", ");**

**}**

**}**

**}**

**Constructor of arrayList :**

1. **ArrayList():**This constructor is used to build an empty array list.

ArrayList arr = new ArrayList();

1. **ArrayList(Collection c):** This constructor is used to build an array list initialized with the elements from the collection c

ArrayList arr = new ArrayList(c);

1. **ArrayList(int capacity):** This constructor is used to build an array list with initial capacity being specified.

ArrayList arr = new ArrayList(N);

#### LinkedList class:

1. LinkedList class is an implementation of the LinkedList data structure which is a linear data structure where the elements are not stored in contiguous locations and every element is a separate object with a data part and address part.
2. The elements are linked using pointers and addresses. Each element is known as a node.
3. It consists of 3 fields:
4. Prev - stores an address of the previous element in the list. It is null for the first element
5. Next - stores an address of the next element in the list. It is null for the last element
6. Data - stores the actual data

**Syntax:** LinkedList<Type> linkedList = new LinkedList<>();

**code :**

import java.util.LinkedList;

class Main {

public static void main(String[] args){

LinkedList<String> animals = new LinkedList<>();

animals.add("Dog");

animals.add("Cat");

animals.add("Cow");

System.out.println("LinkedList: " + animals);

animals.add(1, "Horse");

System.out.println("Updated LinkedList: " + animals);

}

}

#### Vector class:

1. The Vector class is an implementation of the List interface that allows us to create resizable-arrays like the ArrayList class.
2. Vector is synchronized.
3. Java Vector contains many legacy methods that are not the part of a collection’s framework.
4. In vector capacity increase double.

**Syntax:** Vector<Type> vector = new Vector<>();

Example:

**class Main {**

**public static void main(String[] args) {**

**Vector<String> animals= new Vector<>();**

**animals.add("Dog");**

**animals.add("Horse");**

**animals.add("Cat");**

**Iterator<String> iterate = animals.iterator();**

**while(iterate.hasNext()) {**

**System.out.print( iterate.next() );**

**System.out.print(", ");**

}

}

}

**Constructor of vector :**

1. **vector() -** It constructs an empty vector with the default size as 10.
2. **vector(int initialCapacity) -** It constructs an empty vector with the specified initial capacity and with its capacity increment equal to zero.
3. **vector(int initialCapacity, int capacityIncrement) -** It constructs an empty vector with the specified initial capacity and capacity increment.
4. **Vector( Collection<? extends E> c) -** It constructs a vector that contains the elements of a collection c.

#### Java Stack Class :

1. Stack class models and implements the Stack data structure.
2. The class is based on the basic principle of last-in-first-out.
3. In addition to the basic push and pop operations, the class provides three more functions of empty, search and peek.
4. The class can also be referred to as the subclass of Vector.

Syntax: Stack<Type> stacks = new Stack<>();

code:

import java.util.\*;

public class GFG {

    public static void main(String args[])

    {

        Stack<String> stack = new Stack<String>();

        stack.push("Geeks");

        stack.push("For");

        stack.push("Geeks");

        Iterator<String> itr = stack.iterator();

        while (itr.hasNext()) {

            System.out.print(itr.next() + " ");

        }

        itr = stack.iterator();

        while (itr.hasNext()) {

            System.out.print(itr.next() + " ");

        }

    }

}

**Stack Methods:**

1. [empty()](https://www.javatpoint.com/java-stack#empty)
2. [push(E item)](https://www.javatpoint.com/java-stack#push)
3. [pop()](https://www.javatpoint.com/java-stack#pop)
4. [peek()](https://www.javatpoint.com/java-stack#peek)
5. [search(Object o)](https://www.javatpoint.com/java-stack#search)

#### Methods of Collection:

The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its sub interfaces.

1. add() - inserts the specified element to the collection.
2. size() - returns the size of the collection.
3. set() : To change elements of the arraylist.
4. remove() - removes the specified element from the collection.
5. iterator() - returns an iterator to access elements of the collection.
6. addAll() - adds all the elements of a specified collection to the collection.
7. removeAll() - removes all the elements of the specified collection from the collection.
8. clear() - removes all the elements of the collection.
9. isEmpty()- This method returns true if this collection contains no elements.
10. max()--This method is used to return the maximum value present in the collection.
11. toArray() --This method is used to return an array containing all the elements in this collection.

### Queue Interface

1. A queue interface maintains the FIFO(First in First Out) order similar to a real-world queue line.
2. This interface is dedicated to storing all the elements where the order of the elements matter.
3. class and interface list in queue interface
   * + 1. priority Queue class
       2. Deque interface
4. linkedList class
5. ArrayDeque class

Diagram

Description automatically generated

#### PriorityQueue class:

A PriorityQueue is used when the objects are supposed to be processed based on the priority. It is known that a queue follows the First-In-First-Out algorithm, but sometimes the elements of the queue are needed to be processed according to the priority and this class is used in these cases. The PriorityQueue is based on the priority heap. The elements of the priority queue are ordered according to the natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

**Syntax: PriorityQueue<Integer> numbers = new PriorityQueue<>();**

**code:**

import java.util.\*;

class GfG {

    public static void main(String args[])

    {

        PriorityQueue<Integer> pQueue = new PriorityQueue<Integer>();

        pQueue.add(10);

        pQueue.add(20);

        pQueue.add(15);

        System.out.println(pQueue.peek());

        System.out.println(pQueue.poll());

        System.out.println(pQueue.peek());

    }

}

Deque Interface:

Deque, also known as a double-ended queue, is a data structure where we can add and remove the elements from both ends of the queue. This interface extends the queue interface. The class which implements this interface is ArrayDeque. Since ArrayDeque class implements the Deque interface, we can instantiate a deque object with this class. we can insert and remove elements from both front and rear.



Classes that implement Deque:

1. ArrayDeque
2. LinkedList
3. **ArrayDeque class:** ArrayDeque class which is implemented in the collection framework provides us with a way to apply resizable-array. This is a special kind of array that grows and allows users to add or remove an element from both sides of the queue.

Syntax: ArrayDeque<Type> animal = new ArrayDeque<>();

code:

import java.util.ArrayDeque;

class Main {

public static void main(String[] args) {

ArrayDeque<String> animals= new ArrayDeque<>();

animals.add("Dog");

animals.addFirst("Cat");

animals.addLast("Horse");

System.out.println("ArrayDeque: " + animals);

}

}

Methods:

### set Interface

A set is an unordered collection of objects in which duplicate values cannot be stored. This collection is used when we wish to avoid the duplication of the objects and wish to store only the unique objects. class of set interface :

1. HashSet
2. LinkedHashSet
3. EnumSet
4. TreeSet
5. **HashSet**: The HashSet class is an inherent implementation of the hash table data structure. The objects that we insert into the HashSet do not guarantee to be inserted in the same order. The objects are inserted based on their hash code. This class also allows the insertion of NULL elements.

import java.util.\*;

public class HashSetDemo {

    public static void main(String args[])

    {

        HashSet<String> hs = new HashSet<String>();

        hs.add("Geeks");

        hs.add("For");

        hs.add("Geeks");

        hs.add("Is");

        hs.add("Very helpful");

        Iterator<String> itr = hs.iterator();

        while (itr.hasNext()) {

            System.out.println(itr.next());

        }

    }

}

**B. LinkedHashSet:** A LinkedHashSet is very similar to a HashSet. The difference is that this uses a doubly linked list to store the data and retains the ordering of the elements.

import java.util.\*;

public class LinkedHashSetDemo {

    public static void main(String args[])

    {

        LinkedHashSet<String> lhs = new LinkedHashSet<String>();

        lhs.add("Geeks");

        lhs.add("For");

        lhs.add("Geeks");

        Iterator<String> itr = lhs.iterator();

        while (itr.hasNext()) {

            System.out.println(itr.next());

        }

    }

}

## Map Interface:

A map is a data structure that supports the key-value pair for mapping the data. This interface doesn’t support duplicate keys because the same key cannot have multiple mappings, however it allows duplicate values in different keys. A map is useful if there is data and we wish to perform operations based on the key.

class of map interface :

1. HashMap
2. EnumMap
3. LinkedHashMap
4. WeakHashMap
5. TreeMap



1. **HashMap:**

HashMap provides the basic implementation of the Map interface of Java. It stores the data in (Key, Value) pairs. To access a value in a HashMap, we must know its key. HashMap uses a technique called Hashing. Hashing is a technique of converting a large String to a small String that represents the same String so that the indexing and search operations are faster. HashSet also uses HashMap internally.

import java.util.\*;

public class HashMapDemo {

    public static void main(String args[]){

        HashMap<Integer, String> hm = new HashMap<Integer, String>();

        hm.put(1, "Geeks");

        hm.put(2, "For");

        hm.put(3, "Geeks");

        System.out.println("Value for 1 is " + hm.get(1));

        for (Map.Entry<Integer, String> e : hm.entrySet())

            System.out.println(e.getKey() + " " + e.getValue());

    }

}

1. **Treemap:**

import java.util.TreeMap;

class Main {

public static void main(String[] args) {

TreeMap<String, Integer> evenNumbers = new TreeMap<>();

evenNumbers.put("Two", 2);

evenNumbers.put("Four", 4);

evenNumbers.putIfAbsent("Six", 6);

System.out.println("TreeMap of even numbers: " + evenNumbers);

TreeMap<String, Integer> numbers = new TreeMap<>();

numbers.put("One", 1);

numbers.putAll(evenNumbers);

System.out.println("TreeMap of numbers: " + numbers);

}

}

**Methods of Map:**

1. put(K, V) - Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.
2. putAll() - Inserts all the entries from the specified map to this map.
3. putIfAbsent(K, V) - Inserts the association if the key K is not already associated with the value V.
4. get(K) - Returns the value associated with the specified key K. If the key is not found, it returns null.
5. getOrDefault(K, defaultValue) - Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.
6. containsKey(K) - Checks if the specified key K is present in the map or not.
7. containsValue(V) - Checks if the specified value V is present in the map or not.
8. replace(K, V) - Replace the value of the key K with the new specified value V.
9. replace(K, oldValue, newValue) - Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.
10. remove(K) - Removes the entry from the map represented by the key K.
11. remove(K, V) - Removes the entry from the map that has key K associated with value V.
12. keySet() - Returns a set of all the keys present in a map.
13. values() - Returns a set of all the values present in a map.
14. entrySet() - Returns a set of all the key/value mapping present in a map.

## Iterator interface

1. The Iterator interface of the Java collections framework allows us to access elements of a collection.
2. Iterators are used to retrieve elements one by one.
3. Iterator is one of the Java cursors.
4. It is a universal iterator as we can apply it to any Collection object.
5. It supports only Forward direction iteration

**Syntax:** Iterator itr = array\_name.iterator();

**code:**

**public class JavaIteratorExample {**

**public static void main(String[] args)   {**

**ArrayList<String> cityNames = new ArrayList<String>();**

**cityNames.add("Delhi");**

**cityNames.add("Mumbai");**

**Iterator iterator = cityNames.iterator();**

**while ( iterator.hasNext() )**

**System.out.print( iterator.next() + " ");**

**}**

**}**

Methods of Iterator Interface---

1. hasNext() - returns true if there exists an element in the collection
2. next() - returns the next element of the collection
3. remove() - removes the last element returned by the next()

#### listIterator interface

1. It has a sub interface Iterator.
2. This means it allows us to iterate elements of a list in both the direction.
3. It is a java iterator that is used to traverse all types of lists including ArrayList, Vector, LinkedList, Stack.
4. There is **no current element** in ListIterator. Its cursor always lies between the previous and next elements.

**Syntax:** ListIterator itr = array\_name.listIterator();

**code:**

**public class GFG {**

**public static void main(String[] args)    {**

**List<String> names = new LinkedList<>();**

**names.add("learn");**

**names.add("from");**

**ListIterator<String> listIterator  = names.listIterator();**

**while (listIterator.hasNext()) {**

           System.out.println( listIterator.next() );

         }

**while (listIterator.hasPrevious()) {**

            System.out.println( listIterator.previous() );

         }

    }

}

**Methods of ListIterator:**

1. hasNext() - returns true if there exists an element in the list
2. next() - returns the next element of the list
3. nextIndex() returns the index of the element that the next() method will return
4. previous() - returns the previous element of the list
5. previousIndex() - returns the index of the element that the previous() method will return
6. remove() - removes the element returned by either next() or previous()
7. set() - replaces the element returned by either next() or previous() with the specified element

## Enumeration interface

# Thread --- 126

1. **Task:**
2. A task is a piece of work that needs to be done.
3. A task is a set of program instructions that are loaded in memory.
4. Task in Operating System may be synonymous with process.
5. If a printer prints a document, it is said to perform a printing task.

Type of task –

1. Single task: when a OS do a single work at time it called single task
2. Multi task: when a OS do multiple task at a time it called multi task.
3. **Process:**

When a task is executing, it become process. So we say that when A program/task is execution state, it become process.

Example: - When we execute a program, it remains on the hard drive of our system and when this program comes into the main memory it becomes a process. The process can be present on a hard drive, memory, or CPU.

1. **Thread:**

A thread is a lightweight sub-process, the smallest unit of processing. Multiprocessing and multithreading, both are used to achieve multitasking.

Thread creates by ---

1. Extending thread class
2. Association of thread class (Runnable interface )

# Synchronization ---141

class is not synchronize .

method should synchronize .

when method synchronize method can acess in mutlti thread .

synchronize method safe but not fast . unsynchronize method is fast but not safe.

# Java Rule various

1. we cannot use private modifier before class. Only public, abstract and final modifier we can use.

private class { } // not possible

1. Object memory allocation is occurred in heap memry area .
2. reference variable memory allocation in slack memory area.



1. **java memory area:**

heap area = object (state)

stack area = local variable

method area = method table, pool area.

native method( third party class) = method stack area

class area

1. bite code of every class store in separate dot(.)class file
2. main method can be called every class . ( We must not create main in every class).
3. In primitive data type, we can do arithmetic operation, but object type we cannot do arithmetic operation . we concat object data type value.
4. **command line interface:** The command-line arguments in Java allow us to pass arguments during the execution of the program. The String array stores all the arguments passed through the command line.

class Main {

public static void main(String[] args) {

System.out.println("Command-Line arguments are");

for(String str: args) {

System.out.println(str);

}

}

}

1. JVM run main method in java.
2. For call non-static method we must create object.
3. System.out.println() method not called void method.

Void show();

System.out.println(show())

1. Static method cannot override.
2. Local variable must be initializer before use;

Void main(){

Int x;

System.out.println(x); // error

}