**JAVA**

**What is java:**

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

**Platform**:

Any hardware or software environment in which a program runs, is known as a platform. Since Java has a runtime environment (JRE) and API, it is called a platform.

**Application:**

According to Sun, 3 billion devices run Java. There are many devices where Java is currently used. Some of them are as follows:

1. Desktop Applications such as acrobat reader, media player, antivirus, etc.
2. Web Applications such as irctc.co.in, javatpoint.com, etc.
3. Enterprise Applications such as banking applications.
4. Mobile
5. Embedded System
6. Smart Card
7. Robotics
8. Games, etc.

**Types of Java Applications:**

There are mainly 4 types of applications that can be created using Java programming:

**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.

1. Servlet
2. JSP
3. Struts
4. Spring
5. Hibernate
6. JSF

etc. technologies are used for creating web applications in Java.

**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 devices 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

Many java versions have been released till now. The current stable release of Java is Java SE 10.

1. JDK Alpha and Beta (1995)
2. JDK 1.0 (23rd Jan 1996)
3. JDK 1.1 (19th Feb 1997)
4. J2SE 1.2 (8th Dec 1998)
5. J2SE 1.3 (8th May 2000)
6. J2SE 1.4 (6th Feb 2002)
7. J2SE 5.0 (30th Sep 2004)
8. Java SE 6 (11th Dec 2006)
9. Java SE 7 (28th July 2011)
10. Java SE 8 (18th Mar 2014)
11. Java SE 9 (21st Sep 2017)
12. Java SE 10 (20th Mar 2018)
13. Java SE 11 (September 2018)
14. Java SE 12 (March 2019)
15. Java SE 13 (September 2019)
16. Java SE 14 (Mar 2020)
17. Java SE 15 (September 2020)
18. Java SE 16 (Mar 2021)
19. Java SE 17 (September 2021)
20. Java SE 18 (by March 2022)

Since Java SE 8 release, the Oracle corporation follows a pattern in which every even version is release in March month and an odd version released in September month.

Java Version History

**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.

**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.

**JDK 1.1 (February 19, 1997):**

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

1. The concept of Inner Class
2. JavaBeans
3. JDBC
4. RMI
5. AWT event model was totally reshaped.
6. Reflection (which supported Introspection only, modification was not possible at runtime).
7. JIT(Just In Time) compiler on Microsoft Windows platforms, produced for Java Soft by Symantec Internationalization
8. Unicode support originating from Telligent.

**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

**J2SE 1.3 (May 8, 2000):**

Its code name was **Kestrel**. Some additions were included to this version. i.e.

1. HotSpot JVM included.
2. RMI was modified to support optional compatibility with CORBA.
3. JNDI (Java Naming and Directory Interface).
4. Java Platform Debugger Architecture (JPDA) included.
5. JavaSound.
6. Synthetic proxy classes.

**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.

1. Improved libraries.
2. Perl regular expressions included.
3. Provided exception chaining (It allows an exception to encapsulate original lower-level exception).
4. IPv6 support (Internet Protocol version 6).
5. Logging API (Specified in JSR 47.)
6. Image I/O API for reading and writing images in formats like JPEG and PNG.
7. XML parser and XSLT processor integrated.
8. Security and cryptography extensions (JCE, JSSE, JAAS) integrated.
9. Support and security updates for Java 1.4 ended in October 2008.

**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

**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.

1. Dropped the support for older Win9x versions.
2. Scripting Language Support.
3. Generic API for tight integration with scripting languages.
4. Improved Web Service support.
5. JDBC 4.0 support.
6. Use a Java Compiler API to invoke a Java Compiler programmatically.
7. After the release of Java 6, Sun released many updates to fix bugs.

**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.

1. JVM support for dynamic languages.
2. Compressed 64-bits pointer.
3. Strings added in switch.
4. Automatic resource management in try-statement.
5. Underscores allowed in numeric literals.
6. Binary integer literals.
7. Improved type interface for creating generic instance. (also called diamond operator <>)
8. Improved catching and throwing. (catch multiple exceptions and rethrow with improved type checking)
9. Provided Java Deployment rulesets.
10. It was the default version to download on java.com from April 2012 up to the release of Java 8.

**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.

1. Language-level support for Lambda expressions.
2. Allowed developers to embed JavaScript code within applications.
3. Annotation of Java Types.
4. Provided Date and Time API.
5. Repeating Annotations.
6. Launching of JavaFX applications.
7. Removal of permanent generation.
8. Java 8 is set as a default version to download from java.com from October 2014.

**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.

1. Modularization of the JDK under Project Jigsaw.
2. Provided Money and Currency API.
3. Tight integration with JavaFX.
4. Java implementation of reactive streams.
5. More Concurrency Updates.
6. Provided Java Linker.
7. Automatic scaling and sizing.

**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:

1. Local-Variable Type Inference
2. Experimental Java-Based JIT Compiler This is the integration of the Graal dynamic compiler for the Linux x64 platform
3. Application Class-Data Sharing This allows application classes to be placed in the shared archive to reduce startup and footprint for Java applications
4. Time-Based Release Versioning
5. Parallel Full GC for G1
6. Garbage-Collector Interface
7. Additional Unicode Language-Tag Extensions
8. Root Certificates
9. Thread-Local Handshakes
10. Heap Allocation on Alternative Memory Devices
11. Remove the Native-Header Generation Tool - javah
12. 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 each and 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 then executed by the machine. The following figure represents the **Java Architecture** in which each step is elaborate graphically.



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 each and 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)

**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 of all, 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. **ClassLoader:** 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.

**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.



Java Runtime Environment

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 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.**



**Variables in Java**

There are 4 types of variables in Java programming language:

1. Local Variables
2. Instance Variables (Non-Static Fields)
3. Class Variables (Static Fields)
4. Parameters

**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.

**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 test = new Test();**

**test.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

**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:**

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.

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 in Java**

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:** The primitive data types include Boolean, char, byte, short, int, long, float and double.
2. **Non-primitive data types:** The non-primitive data types include Classes, Interfaces, and Arrays.

**Java Primitive Data Types:**

In Java language, primitive data types are the building blocks of data manipulation. These are the most basic data types available in Java language. Java is a statically typed programming language. It means, all variables must be declared before its use. That is why we need to declare variable's type and name.

There are 8 types of primitive data types:

**Data Type Default Value Default size**

Boolean false 1 bit

char '\u0000' 2 byte

byte 0 1 byte

short 0 2 byte

int 0 4 byte

long 0L 8 byte

float 0.0f 4 byte

double 0.0d 8 byte

**java non-primitive data types:**

1. Arrays.
2. Classes
3. Interfaces

**Operators in Java**

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 | = += -= \*= /= %= &= ^= |= <<= >>= >>>= |

**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 and Blocks**

**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 Blocks:**

A block is a group of statements (zero or more) that is enclosed in curly braces {}. For example,

**class Main {**

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

**String band = "Beatles";**

**if (band == "Beatles") { // start of block**

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

**System.out.print("Jude!");**

**} // end of block**

**}**

**}**

**Java Control Statements**

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**

**If Statement:**

In Java, the "if" statement is used to evaluate a condition. The control of the program is diverted depending upon the specific condition. The condition of the If statement gives a Boolean value, either true or false. In Java, there are four types of if-statements given below.

1. Simple if statement
2. if-else statement
3. if-else-if ladder
4. Nested if-statement

**Simple 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. Syntax of if statement is given below.

if(condition) {

statement 1; //executes when condition is true

}

**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.

**Syntax:**

**if**(condition)

 {

statement 1; //executes when condition is true

}

**else**{

statement 2; //executes when condition is false

}

**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.

Syntax of if-else-if statement is given below.

**if(condition 1) {**

**statement 1; //executes when condition 1 is true**

**}**

**else if(condition 2) {**

**statement 2; //executes when condition 2 is true**

**}**

**else {**

**statement 2; //executes when all the conditions are false**

**}**

**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.

Syntax of Nested if-statement is given below.

**if** (condition 1) {

statement 1; //executes when condition 1 is true

**if** (condition 2) {

statement 2; //executes when condition 2 is true

}

**else** {

statement 2; //executes when condition 2 is false

}

}

**Switch Statement**

Switch statements are similar to if-else-if statements. The switch statement contains multiple blocks of code called cases and a single case is executed based on the variable which is being switched. The switch statement is easier to use instead of if-else-if statements. It also enhances the readability of the program.

Points to be noted about switch statement:

1. The case variables can be int, short, byte, char, or enumeration. String type is also supported since version 7 of Java
2. Cases cannot be duplicate
3. Default statement is executed when any of the case doesn't match the value of expression. It is optional.
4. Break statement terminates the switch block when the condition is satisfied.  
   It is optional, if not used, next case is executed.
5. 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.

The syntax to use the switch statement is given below.

**switch** (expression){

**case** value1:

     statement1;

**break**;

**case** valueN:

     statementN;

**break**;

**default**:

**default** statement;

}

**Student.java**

**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).

}

}

}

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. The switch permits only int, string, and Enum type variables to be used.

**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. The execution of the set of instructions depends upon a particular condition.

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

**For loop:**

In Java, for loop is similar to C and C++. 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.



**Calculation.java:**

**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 first 10 natural numbers is " + sum);

}

}

s is 55

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 var : array\_name/collection\_name ){

//statements

}

**Calculation.java:**

**public** **class** Calculation {

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

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

System.out.println("Printing the content of the array names:\n");

**for**(String name:names) {

System.out.println(name);

}

}

}

**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

}

**Calculation .java:**

**public** **class** Calculation {

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

**int** i = 0;

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

**while**(i<=10) {

System.out.println (i);

i = i + 2;

}

}

}

**Java 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);



**Calculation.java:**

**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, i.e.,

1. break
2. continue.

**Java break statement:**

As the name suggests, the break statement is used to break the current flow of the program and transfer the control to the next statement outside a loop or switch statement. However, it breaks only the inner loop in the case of the nested loop. The break statement cannot be used independently in the Java program, i.e., it can only be written inside the loop or switch statement.

**The break statement example with for loop** . Consider the following example in which we have used the break statement with the for loop.

**BreakExample.java**

**public** **class** BreakExample

{

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

{

**for**(**int** i = 0; i<= 10; i++)

 {

System.out.println(i);

**If** ( i==6)

{

**break**;

}

}

}

}

**Java continue statement:**

Unlike break 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. Consider the following example to understand the functioning of the continue statement in Java.

**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.
2. The statements break and continue in Java alter the normal control flow of control flow statements. They can use labels which are valid java identifiers with a colon.
3. Labeled blocks can only be used with break and continue statements.
4. Labaled break and continue statements must be called within its scope. We can not refer them outside the scope of labeled block.
5. The break statement immediately jumps to the end (and out) of the appropriate compound statement.
6. The continue statement immediately jumps to the next iteration (if any) of the appropriate loop.
7. 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

Normally, an array is a collection of similar type of elements which has contiguous memory location. **Java array** is an object which contains elements of a similar data type. Additionally, the elements of an array are stored in a contiguous memory location. It is a data structure where we store similar elements. We can store only a fixed set of elements in a Java array. Array in Java is index-based, the first element of the array is stored at the 0th index, 2nd element is stored on 1st index and so on. Unlike C/C++, we can get the length of the array using the length member. In C/C++, we need to use the sizeof operator. In Java, array is an object of a dynamically generated class. Java array inherits the Object class, and implements the Serializable as well as Cloneable interfaces. We can store primitive values or objects in an array in Java. Like C/C++, we can also create single dimensional or multidimensional arrays in Java.

Moreover, Java provides the feature of anonymous arrays which is not available in C/C++.



Advantages

1. **Code Optimization:** It makes the code optimized, we can retrieve or sort the data efficiently.
2. **Random access:** We can get any data located at an index position.

Disadvantages

1. **Size Limit:** We can store only the fixed size of elements in the array. It doesn't grow its size at runtime. To solve this problem, collection framework is used in Java which grows automatically.

**Types of Arrays in java:**

1. Single Dimensional Array
2. Multidimensional Array

**1) Single Dimensional Array in Java:**

**Syntax to Declare an Array in Java:**

1. datatype[] arr
2. dataType []arr
3. dataType arr[]

**Instantiation of an Array in Java:**

arrayRefVar = **new** datatype[size];

Example :

**class Testarray**

**{**

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

**{**

**int a[]=new int[5];//declaration and instantiation**

**a[0]=10;//initialization**

**a[1]=20;**

**for(int i=0;i<a.length;i++)//length is the property of array**

**System.out.println(a[i]);**

**}**

**}**

Declaration, Instantiation and Initialization of Java Array:

**int** a[]={33,3,4,5}; //declaration, instantiation and initialization

**Example**:

**class** Testarray1

{

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

{

**int** a[]={33,3,4,5};//declaration, instantiation and initialization

**for**(**int** i=0;i<a.length;i++)//length is the property of array

System.out.println(a[i]);

}

}

**For-each Loop for Java Array:**

We can also print the Java array using **for-each loop**. The Java for-each loop prints the array elements one by one. It holds an array element in a variable, then executes the body of the loop.

The syntax of the for-each loop is given below:

**for**(datatype variable: array)

{

//body of the loop

}

**Example:**

**class** Testarray1

{

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

{

**int** arr[]={33,3,4,5};

**for**(**int** i:arr)

 {

System.out.println(i);

}

}

}

33

**Passing Array to a Method in Java:**

We can pass the java array to method so that we can reuse the same logic on any array. Let's see the simple example to get the minimum number of an array using a method.

**class** Testarray2

{

**static** **void** min(**int** arr[])

{

**int** min=arr[0];

**for**(**int** i=1;i<arr.length;i++)

**if**(min>arr[i])

{

   min=arr[i];

}

System.out.println(min);

}

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

{

**int** a[]={33,3,4,5}; //declaring and initializing an array

min(a); //passing array to method

}

}

3

**Anonymous Array in Java:**

Java supports the feature of an anonymous array, so you don't need to declare the array while passing an array to the method.

**public** **class** TestAnonymousArray

{

**static** **void** printArray(**int** arr[])

{

**for**(**int** i=0;i<arr.length;i++)

System.out.println(arr[i]);

}

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

{

printArray(**new** **int**[]{10,22,44,66});//passing anonymous array method

}

}

66

**Returning Array from the Method:**

We can also return an array from the method in Java.

**class** TestReturnArray

{

**static** **int**[] get()

{

**return** **new** **int**[]{10,30,50,90,60};

}

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

{

**int** arr[]=get();

**for**(**int** i=0;i<arr.length;i++)

System.out.println(arr[i]);

}

}

60

**2)Multidimensional Array in Java:**

**Syntax:**

1. dataType[][]  arrayRefVar;
2. datatype   [][]arrayRefVar;
3. datatype  arrayRefVar[][];
4. dataType [] arrayRefVar[];

**Example :**

**int**[][] arr=**new** **int**[3][3];

Example of Multidimensional Java Array

Let's see the simple example to declare, instantiate, initialize and print the 2Dimensional array.

**class** Testarray3

{

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

{

**int** arr[][]={{1,2,3},{2,4,5},{4,4,5}};

**for**(**int** i=0;i<3;i++)

{

**for**(**int** j=0;j<3;j++)

{

    System.out.print(arr[i][j]+" ");

 }

 System.out.println();

}

}

}

**Jagged Array in Java:**

If we are creating odd number of columns in a 2D array, it is known as a jagged array. In other words, it is an array of arrays with different number of columns.

**class** TestJaggedArray

{

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

{

**int** arr[][] = **new** **int**[3][];

        arr[0] = **new** **int**[3];

        arr[1] = **new** **int**[4];

        arr[2] = **new** **int**[2];

**int** count = 0;

**for** (**int** i=0; i<arr.length; i++)

{

**for**(**int** j=0; j<arr[i].length; j++)

                arr[i][j] = count++;

**for** (**int** i=0; i<arr.length; i++)

{

**for** (**int** j=0; j<arr[i].length; j++){

                System.out.print(arr[i][j]+" ");

            }

            System.out.println(); //new line

        }

     }

}

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

1. getClass().
2. 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);

}

}

**Copying a Java Array**

We can copy an array to another by the arraycopy() method of System class.

Example of Copying an Array in Java

**class** TestArrayCopyDemo

 {

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

 {

**char**[] copyFrom = { 'd', 'e', 'c', 'a', 'f', 'f', 'e',   'i', 'n', 'a', 't', 'e', 'd' };

**char**[] copyTo = **new** **char**[7];

        System.arraycopy(copyFrom, 2, copyTo, 0, 7);

        System.out.println(String.valueOf(copyTo));

    }

}  caffein

Cloning an Array in Java

Since, Java array implements the Cloneable interface, we can create the clone of the Java array. If we create the clone of a single-dimensional array, it creates the deep copy of the Java array. It means, it will copy the actual value. But, if we create the clone of a multidimensional array, it creates the shallow copy of the Java array which means it copies the references.

**class** Testarray1

{

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

{

**int** arr[]={33,3,4,5};

System.out.println("Printing original array:");

**for**(**int** i:arr)

{

System.out.println(i);

}

System.out.println("Printing clone of the array:");

**int** carr[]=arr.clone();

**for**(**int** i:carr)

 {

System.out.println(i);

  }

System.out.println("Are both equal?");

System.out.println(arr==carr);

}

}  A

re both equal?

Addition of 2 Matrices in Java:

Let's see a simple example that adds two matrices.

**class** Testarray5

{

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

{

**int** a[][]={

{1,3,4},

{3,4,5}

};

**int** b[][]={ {1,3,4}, {3,4,5} };

**int** c[][]=**new** **int**[2][3];

**for**(**int** i=0;i<2;i++)

{

**for**(**int** j=0;j<3;j++)

{

c[i][j]=a[i][j]+b[i][j];

System.out.print(c[i][j]+" ");

}

System.out.println();//new line

}

}

}

Multiplication of 2 Matrices in Java:

In the case of matrix multiplication, a one-row element of the first matrix is multiplied by all the columns of the second matrix which can be understood by the image given below.



Let's see a simple example to multiply two matrices of 3 rows and 3 columns.

//Java Program to multiply two matrices

**public** **class** MatrixMultiplicationExample

{

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

{

**int** a[][]= { {1,1,1}, {2,2,2}, {3,3,3} };

**int** b[][]={ {1,1,1}, {2,2,2}, {3,3,3} };

**int** c[][]=**new** **int**[3][3];   //3 rows and 3 columns

**for**(**int** i=0;i<3;i++)

{

**for**(**int** j=0;j<3;j++)

{

c[i][j]=0;

**for**(**int** k=0;k<3;k++)

{

c[i][j]+=a[i][k]\*b[k][j];

} //end of k loop

System.out.print(c[i][j]+" ");   //printing matrix element

} //end of j loop

System.out.println(); //new line

}

}

}

**Method of JAVA**

A method is a block of code that performs a specific task.  We write a method once and use it many times.

In Java, there are two types of methods:

1. **User-defined Methods**: We can create our own method based on our requirements.
2. **Standard Library Methods**: These are built-in methods in Java that are available to use

**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.

Create a user defined method that checks the number is even or odd. First, we will define the method.

public static void findEvenOdd ( int num )

{

If (num%2 == 0)

System.out.println(num+" is even");

else

System.out.println(num+" is odd");

}

**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

1. **length()**
2. **equals()**
3. **compareTo()**
4. **sqrt()**

we call any of the predefined methods in our program, which is already stored in the library. Each predefined method is defined inside a class.

Such as **print()** method is defined in the **java.io.PrintStream** class.

**Declaring a Java Method:**

The syntax to declare a method is:

1. **Access Specifier:**
2. **ReturnType**
3. **Method Name**
4. **Parameter List**
5. **Method Body**



**Access Specifier:**

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:

1. Public: The method is accessible by all classes when we use public specifier in our application.
2. Private: When we use a private access specifier, the method is accessible only in the classes in which it is defined.
3. Protected: When we use protected access specifier, the method is accessible within the same package or subclasses in a different package.
4. 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.

**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.

**Method Name**:

It is an identifier that is used to refer to the method in a program.

**Method body**:

It includes the programming statements that are used to perform some tasks. The method body is enclosed inside the **curly braces {}.**

**Naming a Method**

1. **Single-word method name:** sum(), area()
2. **Multi-word method name:** areaOfCircle(), stringComparision()

**Other Method**

**Static Method:**

1. A method that has static keyword is known as static method. In other words, a method that belongs to a class rather than an instance of a class is known as a static method.
2. We can also create a static method by using the keyword **static** before the method name.
3. The main advantage of a static method is that we can call it without creating an object. It can access static data members and change the value of it. It is used to create an instance method.
4. It is invoked by using the class name. The best example of a static method is the **main()** method.
5. Static method can’t use non static member
6. “this” and “super” keyword can’t be used here

**public** **class** Display

{

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

{

show();

}

**static** **void** show()

{

System.out.println("It is an example of static method.");

}

}  of

a static method.

**Instance Method:**

The method of the class is known as an **instance method**. It is a **non-static** method defined in the class. Before calling or invoking the instance method, it is necessary to create an object of its class. Let's see an example of an instance method.

**Instance Method Example,**

**public** **class** ABC

{

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

{

ABC obj = **new** ABC();

System.out.println ("The sum is: "+obj.add(12, 13));

}

**int** s ;

**public** **int** add(**int** a, **int** b)

{

s = a+b ;

**return** s ;

}

}  25

There are two types of instance method:

1. **Accessor Method**
2. **Mutator Method**

**Accessor 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. **Example:**

**public** **int** getId()

{

**return** Id;

}

**Mutator Method:**

The method(s) read the instance variable(s) and also 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. **Example,**

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

{

**this**.roll = roll;

}

Example of accessor and mutator method,

**public** **class** Student

{

**private** **int** roll;

**private** String name;

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

{

**return** roll;

}

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

{

**this**.roll = roll;

}

**public** String getName()

{

**return** name;

}

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

{

**this**.name = name;

}

**public** **void** display()

{

System.out.println("Roll no.: "+roll);

System.out.println("Student name: "+name);

}

}

**Abstract Method:**

The method that does not has method body is known as abstract method. In other words, without an implementation is known as abstract method. It always declares in the **abstract class**. It means the class itself must be abstract if it has abstract method. To create an abstract method, we use the keyword **abstract**.

**Syntax**

**abstract** **void** method\_name();

Example of abstract method

**Demo.java**

**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();

}

}

method...

**Factory method**

It is a method that returns an object to the class to which it belongs. All static methods are factory methods. For example,

**NumberFormat obj = NumberFormat.getNumberInstance();**

OOP’s

Java is an Object-Oriented Language. 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 in Java:**

A class is a blueprint from which individual objects are created.

**Example**

public class Dog {

String breed;

int age;

String color;

void barking() {

}

void hungry() {

}

void sleeping() {

}

}

A class can contain any of the following variable types.

1. **Local variables** − Variables defined inside methods, constructors or blocks are called local variables. The variable will be declared and initialized within the method and the variable will be destroyed when the method has completed.
2. **Instance variables** − Instance variables are variables within a class but outside any method. These variables are initialized when the class is instantiated. Instance variables can be accessed from inside any method, constructor or blocks of that particular class.
3. **Class variables** − Class variables are variables declared within a class, outside any method, with the static keyword. A class can have any number of methods to access the value of various kinds of methods. In the above example, barking(), hungry() and sleeping() are methods.

**Objects in Java**

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:**

As mentioned previously, a class provides the blueprints for objects. So basically, an object is created from a class. In Java, the new keyword is used to create new objects.

There are three steps when creating an object from a class −

1. **Declaration** − A variable declaration with a variable name with an object type.
2. **Instantiation** − The 'new' keyword is used to create the object.
3. **Initialization** − The 'new' keyword is followed by a call to a constructor. This call initializes the new object.

**Example:**

public class Puppy {

public Puppy(String name) {

System.out.println("Passed Name is :" + name );

}

public static void main(String []args) {

Puppy myPuppy = new Puppy( "tommy" );

}

}

**Example:**

This example explains how to access instance variables and methods of a class.

public class Puppy {

int puppyAge;

public Puppy(String name)

{

System.out.println("Name chosen is :" + name );

}

public void setAge( int age )

{

puppyAge = age;

}

public int getAge( )

{

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

return puppyAge;

}

public static void main(String []args)

{

Puppy myPuppy = new Puppy( "tommy" );

myPuppy.setAge( 2 );

myPuppy.getAge( );

System.out.println("Variable Value :" + myPuppy.puppyAge );

}

}

**Constructors:**

When discussing about classes, one of the most important sub topic would be constructors. Every class has a constructor. If we do not explicitly write a constructor for a class, the Java compiler builds a default constructor for that class. Each time a new object is created, at least one constructor will be invoked. The main rule of constructors is that they should have the same name as the class. A class can have more than one constructor.

**Example:**

public class Puppy {

public Puppy() {

}

public Puppy(String name) {

// This constructor has one parameter, *name*.

}

}

Java also supports [Singleton Classes](https://www.tutorialspoint.com/java/java_using_singleton.htm) where you would be able to create only one instance of a class.

**Source File Declaration Rules:**

As the last part of this section, let's now look into the source file declaration rules. These rules are essential when declaring classes, *import* statements and *package* statements in a source file.

1. There can be only one public class per source file.
2. A source file can have multiple non-public classes.
3. The public class name should be the name of the source file as well which should be appended by **.java** at the end. For example: the class name is *public class Employee{}* then the source file should be as Employee.java.
4. If the class is defined inside a package, then the package statement should be the first statement in the source file.
5. If import statements are present, then they must be written between the package statement and the class declaration. If there are no package statements, then the import statement should be the first line in the source file.
6. Import and package statements will imply to all the classes present in the source file. It is not possible to declare different import and/or package statements to different classes in the source file.

Classes have several access levels and there are different types of classes; abstract classes, final classes, etc. We will be explaining about all these in the access modifiers chapter.

Apart from the above-mentioned types of classes, Java also has some special classes called Inner classes and Anonymous classes.

**Java Package**

In simple words, it is a way of categorizing the classes and interfaces. When developing applications in Java, hundreds of classes and interfaces will be written, therefore categorizing these classes is a must as well as makes life much easier.

**Import Statements**

In Java if a fully qualified name, which includes the package and the class name is given, then the compiler can easily locate the source code or classes. Import statement is a way of giving the proper location for the compiler to find that particular class.

For example, the following line would ask the compiler to load all the classes available in directory java\_installation/java/io −

import java.io.\*;

**A Simple Case Study**

For our case study, we will be creating two classes. They are Employee and EmployeeTest.

First open notepad and add the following code. Remember this is the Employee class and the class is a public class. Now, save this source file with the name Employee.java.

The Employee class has four instance variables - name, age, designation and salary. The class has one explicitly defined constructor, which takes a parameter.

**Example**

import java.io.\*;

public class Employee {

String name;

int age;

String designation;

double salary;

public Employee(String name) {

this.name = name;

}

public void empAge(int empAge) {

age = empAge;

}

public void empDesignation(String empDesig) {

designation = empDesig;

}

public void empSalary(double empSalary) {

salary = empSalary;

}

public void printEmployee() {

System.out.println("Name:"+ name );

System.out.println("Age:" + age );

System.out.println("Designation:" + designation );

System.out.println("Salary:" + salary);

}

}

As mentioned previously in this tutorial, processing starts from the main method. Therefore, in order for us to run this Employee class there should be a main method and objects should be created. We will be creating a separate class for these tasks.

Following is the *EmployeeTest* class, which creates two instances of the class Employee and invokes the methods for each object to assign values for each variable.

Save the following code in EmployeeTest.java file.

import java.io.\*;

public class EmployeeTest {

public static void main(String args[]) {

/\* Create two objects using constructor \*/

Employee empOne = new Employee("James Smith");

Employee empTwo = new Employee("Mary Anne");

// Invoking methods for each object created

empOne.empAge(26);

empOne.empDesignation("Senior Software Engineer");

empOne.empSalary(1000);

empOne.printEmployee();

empTwo.empAge(21);

empTwo.empDesignation("Software Engineer");

empTwo.empSalary(500);

empTwo.printEmployee();

}

}

Relation of class

1. Inheritance (IS-A),
2. Association (HAS-A)

**IS-a relationship:**

In Java, inheritance is an **IS-A** relationship. That is, we use inheritance only if there exists an IS-A relationship between two classes.

class Animal {

}

class Mammal extends Animal {

}

class Reptile extends Animal {

}

public class Dog extends Mammal {

public static void main(String args[]) {

Animal a = new Animal();

Mammal m = new Mammal();

Dog d = new Dog();

System.out.println(m instanceof Animal);

System.out.println(d instanceof Mammal);

System.out.println(d instanceof Animal);

}

}

ow, based on the above example, in Object-Oriented terms, the following are true −

1. Animal is the superclass of Mammal class.
2. Animal is the superclass of Reptile class.
3. Mammal and Reptile are subclasses of Animal class.
4. Dog is the subclass of both Mammal and Animal classes.
5. Now, if we consider the IS-A relationship, we can say −
6. Mammal IS-A Animal
7. Reptile IS-A Animal
8. Dog IS-A Mammal

Hence: Dog IS-A Animal as well

With the use of the extends keyword, the subclasses will be able to inherit all the properties of the superclass except for the private properties of the superclass.

We can assure that Mammal is an Animal with the use of the instance operator.

**HAS-A relationship(**Aggregation**):**

If a class have an entity reference, it is known as Aggregation. Aggregation represents HAS-A relationship. These relationships are mainly based on the usage. This determines whether a certain class **HAS-A** certain thing. This relationship helps to reduce duplication of code as well as bugs. Employee object contains many information’s such as id, name, emailed etc. It contains one more object named address, which contains its own information’s such as city, state, country, zip code etc. as given below.

**class** Employee{

**int** id;

String name;

Address address;

}

**class** Operation{

**int** square(**int** n){

**return** n\*n;

 }

}

**class** Circle{

  Operation op;//aggregation

**double** pi=3.14;

**double** area(**int** radius){

    op=**new** Operation();

**int** rsquare=op.square(radius);

**return** pi\*rsquare;

  }

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

  Circle c=**new** Circle();

**double** result=c.area(5);

   System.out.println(result);

 }

}

When uses Aggregation?

Code reuse is also best achieved by aggregation when there is no is-a relationship.

Inheritance should be used only if the relationship is-a is maintained throughout the lifetime of the objects involved; otherwise, aggregation is the best choice.

In this example, Employee has an object of Address, address object contains its own information’s such as city, state, country etc. In such case relationship is Employee HAS-A address.

**Address.java**

**public** **class** Address {

String city,state,country;

**public** Address(String city, String state, String country) {

**this**.city = city;

**this**.state = state;

**this**.country = country;

}

}

**Emp.java**

**public** **class** Emp {

**int** id;

String name;

Address address;

**public** Emp(**int** id, String name,Address address) {

**this**.id = id;

**this**.name = name;

**this**.address=address;

}

**void** display(){

System.out.println(id+" "+name);

System.out.println(address.city+" "+address.state+" "+address.country);

}

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

Address address1=**new** Address("gzb","UP","india");

Address address2=**new** Address("gno","UP","india");

Emp e=**new** Emp(111,"varun",address1);

Emp e2=**new** Emp(112,"arun",address2);

e.display();

e2.display();

}

}

**Java – Inheritance**

Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order. The class which inherits the properties of other is known as subclass (derived class, child class) and the class whose properties are inherited is known as superclass (base class, parent class).

**Extends Keyword:**

**extends** is the keyword used to inherit the properties of a class. Following is the syntax of extends keyword.

**Syntax:**

class Super

{

………….

}

class Sub extends Super

{

………….

}

Example:

class Animal

{

String name;

public void eat() {

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

}

}

class Dog extends Animal {

public void display() {

System.out.println("My name is " + name);

}

}

class Main {

public static void main(String[] args) {

Dog labrador = new Dog();

labrador.name = "Rohu";

labrador.display();

labrador.eat();

}

}

Types of Inheritance

There are various types of inheritance as demonstrated below.



Java Encapsulation

Encapsulation is one of the key features of object-oriented programming. Encapsulation refers to the bundling of fields and methods inside a single class. It prevents outer classes from accessing and changing fields and methods of a class. In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class. Therefore, it is also known as data hiding.

**class** Account {

**private** **long** acc\_no;

**private** String name,email;

**private** **float** amount;

**public** **long** getAcc\_no() {

**return** acc\_no;

}

**public** **void** setAcc\_no(**long** acc\_no) {

**this**.acc\_no = acc\_no;

}

**public** String getName() {

**return** name;

}

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

**this**.name = name;

}

**public** String getEmail() {

**return** email;

}

**public** **void** setEmail(String email) {

**this**.email = email;

}

**public** **float** getAmount() {

**return** amount;

}

**public** **void** setAmount(**float** amount) {

**this**.amount = amount;

}

}

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

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

    Account acc=**new** Account();

    acc.setAcc\_no(7560504000L);

    acc.setName("Sonoo Jaiswal");

    acc.setEmail("sonoojaiswal@javatpoint.com");

    acc.setAmount(500000f);

    System.out.println(acc.getAcc\_no()+" "+acc.getName());

}

}

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.

Java Polymorphism

Polymorphism means more than one form. the same entity (method or operator or object) can perform different operations in different scenarios.

We can achieve polymorphism in Java using the following ways:

1. [Method Overriding](https://www.programiz.com/java-programming/method-overriding)
2. [Method Overloading](https://www.programiz.com/java-programming/method-overloading)
3. Operator Overloading

Java 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

}

}

**Rules for Method Overriding**

1. The argument list should be exactly 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. An overriding method can throw any uncheck exceptions, regardless of whether the overridden method throws exceptions or not. However, the overriding method should not throw checked exceptions that are new or broader than the ones declared by the overridden method. The overriding method can throw narrower or fewer exceptions than the overridden method.
11. Constructors cannot be overridden.

Super Keyword in Java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object.

Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable.

Usage of Java super Keyword

1. super can be used to refer immediate parent class instance variable.
2. super can be used to invoke immediate parent class method.
3. super() can be used to invoke immediate parent class constructor.

Example:

**class** Animal{

Animal(){

System.out.println("animal is created");

}

}

**class** Dog **extends** Animal{

Dog(){

**super**();

System.out.println("dog is created");

}

}

**class** TestSuper3{

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

Dog d=**new** Dog();

}

}

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 have to perform only one operation, having same name of the methods increases the readability of the program.

There are two ways to overload the method in java

1. By changing number of arguments
2. By changing the data type

1.Overloading 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);

}

}

2. Method Overloading by changing the data type of parameters

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.");

}

public static void main(String[] args) {

display(1);

display("Hello");

}

}

Abstract class in Java

Before learning the Java abstract class, let's understand the abstraction in Java first.

Abstraction in Java:

**Abstraction** is a process of hiding the implementation details and showing only functionality to the user. Another way, it shows only essential things to the user and hides the internal details.

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%)

Two types of abstraction :

1. Java Abstract Class
2. Java Abstract Method

Abstract class in Java

A class which is declared as abstract is known as an **abstract class**. It can have abstract and non-abstract methods. It needs to be extended and its method implemented. It cannot be instantiated.

1. An abstract class must be declared with an abstract keyword.
2. Abstract classes may or may not contain abstract methods.
3. if a class has at least one abstract method, then the class must be declared abstract.
4. If a class is declared abstract, it cannot be instantiated. (We cannot create objects of abstract classes)
5. To use an abstract class, you must inherit it from another class, provide implementations to the abstract methods in it.
6. If you inherit an abstract class, you must provide implementations to all the abstract methods in

it.

Example:

public abstract class Employee {

private int number;

public Employee(String name) {

this.name = name;

}

public double computePay() {

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

}

}

public class Main {

public static void main(String [] args) {

Employee e = new Employee("George W.", "Houston, TX", 43);

System.out.println("………………………");

e.mailCheck();

}

}

Abstract Method in Java

A method which is declared as abstract and does not have implementation is known as an abstract method.

1. An abstract method contains a method signature, but no method body.
2. Instead of curly braces, an abstract method will have a semi colon (;)
3. if any method declared in abstract class, then extends class must be override the method.
4. If abstract method declared in class and class is not declared abstract class then, class will be abstrack class . class object cannot create .

Example:

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 in Java

An interface is a reference type in Java. An interface is a fully abstract class. It includes a group of abstract methods. Along with abstract methods, an interface may also contain constants, default methods, static methods, and nested types. Method bodies exist only for default methods and static methods. Java Interface also **represents the IS-A relationship**.

1. An interface does not contain any constructors.
2. An interface is not extended by a class; it is implemented by a class.
3. An interface can extend multiple interfaces.
4. We cannot create objects of an interface
5. An interface is implicitly abstract. You do not need to use the abstract keyword while declaring an interface.
6. Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
7. Methods in an interface are implicitly public.
8. Since Java 8, we can have default and static methods in an interface.
9. Since Java 9, we can have private methods in an interface.

Declaring Interfaces

The interface keyword is used to declare an interface. Here is a simple example to declare an interface

interface Animal {

public void eat();

public void travel();

}

Implementing Interfaces

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

Example:

interface Animal {

public void eat();

public void travel();

}

public class MammalInt implements Animal {

public void eat() {

System.out.println("Mammal eats");

}

public void travel() {

System.out.println("Mammal travels");

}

}

public static void main(String args[]) {

MammalInt m = new MammalInt();

m.eat();

m.travel();

}

}



Multiple inheritance in Java by interface

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



**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();

 }

}

**Java Package**

A **java package** is a group of similar types of classes, interfaces and sub-packages. Package in java can be categorized in two forms,

1. built-in package
2. user-defined package.

**How to access package from another package?**

There are three ways to access the package from outside the package.

1. import package.\*;
2. import package.classname;
3. fully qualified name.

Various class of java:

1. [Inner Class](https://www.programiz.com/java-programming/nested-inner-class)
2. [Java Singleton](https://www.programiz.com/java-programming/singleton)
3. [Java enum Class](https://www.programiz.com/java-programming/enums)
4. [Java enum Constructor](https://www.programiz.com/java-programming/enum-constructor)
5. [Java enum String](https://www.programiz.com/java-programming/enum-string)
6. [Java Reflection](https://www.programiz.com/java-programming/reflection)

Java Inner Classes (Nested Classes):

you can define a class within another class. Such class is known as nested class. it can access all the members of the outer class, including private data members and methods.

class OuterClass {

……………………..

class NestedClass {

…………………….

}

}

There are two types of nested classes you can create in Java.

1. Non-static nested class (inner class): These are the non-static members of a class.
2. Static nested class: These are the static members of a class

**Non-static Nested Classes**

Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

Inner classes are of three types depending on how and where you define them. They are −

1. Inner Class/**member inner class/regular inner class**.
2. Method-local Inner Class
3. Anonymous Inner Class

Inner Class/**member inner class/regular inner class :**

Creating an inner class is quite simple. You just need to write a class within a class. Unlike a class, an inner class can be private and once you declare an inner class private, it cannot be accessed from an object outside the class.

Following is the program to create an inner class and access it. In the given example, we make the inner class private and access the class through a method.

**Example**

**class** TestMemberOuter1{

**private** **int** data=30;

**class** Inner{

**void** msg(){System.out.println("data is "+data);}

 }

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

  TestMemberOuter1 obj=**new** TestMemberOuter1();

  TestMemberOuter1.Inner in=obj.**new** Inner();

  in.msg();

 }

}

**Method-local Inner Class:**

In Java, we can write a class within a method and this will be a local type. Like local variables, the scope of the inner class is restricted within the method.

A method-local inner class can be instantiated only within the method where the inner class is defined. The following program shows how to use a method-local inner class.

**Example**

public class Outerclass

{

void my\_Method()

{

int num = 23;

class MethodInner\_Demo

{

public void print() {

System.out.println("This is method inner class "+num);

}

}

MethodInner\_Demo inner = new MethodInner\_Demo();

inner.print();

}

public static void main(String args[])

{

Outerclass outer = new Outerclass();

outer.my\_Method();

}

}

**Anonymous Inner Class:**

An inner class declared without a class name is known as an **anonymous inner class**. In case of anonymous inner classes, we declare and instantiate them at the same time. Generally, they are used whenever you need to override the method of a class or an interface. The syntax of an anonymous inner class is as follows −

**Syntax**

AnonymousInner an\_inner = new AnonymousInner() {

public void my\_method() {

........

........

}

};

**Example**

abstract class AnonymousInner {

public abstract void mymethod();

}

public class Outer\_class {

public static void main(String args[]) {

AnonymousInner inner = new AnonymousInner() {

public void mymethod() {

System.out.println("This is an example of anonymous inner class");

}

};

inner.mymethod();

}

}

**Anonymous Inner Class as Argument**

Generally, if a method accepts an object of an interface, an abstract class, or a concrete class, then we can implement the interface, extend the abstract class, and pass the object to the method. If it is a class, then we can directly pass it to the method.

But in all the three cases, you can pass an anonymous inner class to the method. Here is the syntax of passing an anonymous inner class as a method argument −

obj.my\_Method(new My\_Class() {

public void Do() {

.....

.....

}

});

The following program shows how to pass an anonymous inner class as a method argument.

**Example**

interface Message {

String greet();

}

public class My\_class {

public void displayMessage(Message m) {

System.out.println(m.greet() );

}

public static void main(String args[]) {

My\_class obj = new My\_class();

obj.displayMessage(new Message() {

public String greet() {

return "Hello";

}

});

}

}

**Static Nested Class**

A static inner class is a nested class which is a static member of the outer class. It can be accessed without instantiating the outer class, using other static members. Just like static members, a static nested class does not have access to the instance variables and methods of the outer class. The syntax of static nested class is as follows −

**Syntax**

class MyOuter {

static class Nested\_Demo {

}

}

Instantiating a static nested class is a bit different from instantiating an inner class. The following program shows how to use a static nested class.

**Example**

public class Outer {

static class Nested\_Demo {

public void my\_method() {

System.out.println("This is my nested class");

}

}

public static void main(String args[]) {

Outer.Nested\_Demo nested = new Outer.Nested\_Demo();

nested.my\_method();

}

}

**Java Singleton Class**

In Java, 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:

* Create a private constructor of the class to restrict object creation outside of the class.
* Create a private attribute of the class type that refers to the single object.
* 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 SingletonExample {

private static SingletonExample singleObject;

private SingletonExample() {

// constructor of the SingletonExample class

}

public static SingletonExample getInstance() {

// write code that allows us to create only one object

// access the object as per our need

}

}

**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**

Java enums

In Java, an enum (short for enumeration) is a type that has a fixed set of constant values. We use the enum keyword to declare enums.

Example:

enum Size {

SMALL, MEDIUM, LARGE, EXTRALARGE

}

class Main {

public static void main(String[] args) {

System.out.println(Size.SMALL);

System.out.println(Size.MEDIUM);

}

}

Java Enum with the switch statement

enum Size {

SMALL, MEDIUM, LARGE, EXTRALARGE

}

class Test {

Size pizzaSize;

public Test(Size pizzaSize) {

this.pizzaSize = pizzaSize;

}

public void orderPizza() {

switch(pizzaSize) {

case SMALL:

System.out.println("I ordered a small size pizza.");

break;

case MEDIUM:

System.out.println("I ordered a medium size pizza.");

break;

default:

System.out.println("I don't know which one to order.");

break;

}

}

}

class Main {

public static void main(String[] args) {

Test t1 = new Test(Size.MEDIUM);

t1.orderPizza();

}

}

**Methods of Java Enum Class:**

There are some predefined methods in enum classes that are readily available for use.

1. Java Enum ordinal()

The ordinal() method returns the position of an enum constant. For example,

ordinal(SMALL)

// returns 0

2. Enum compareTo()

The compareTo() method compares the enum constants based on their ordinal value. For example,

Size.SMALL.compareTo(Size.MEDIUM)

// returns ordinal(SMALL) - ordinal(MEDIUM)

3. Enum toString()

The toString() method returns the string representation of the enum constants. For example,

SMALL.toString()

// returns "SMALL"

4. Enum name()

The name() method returns the defined name of an enum constant in string form. The returned value from the name() method is final. For example,

name(SMALL)

// returns "SMALL"

5. Java Enum valueOf()

The valueOf() method takes a string and returns an enum constant having the same string name. For example,

Size.valueOf("SMALL")

// returns constant SMALL.

6. Enum values()

The values() method returns an array of enum type containing all the enum constants. For example,

Size[] enumArray = Size.value();

**Why Java Enums?**

In Java, enum was introduced to replace the use of int constants.

Suppose we have used a collection of int constants.

class Size {

public final static int SMALL = 1;

public final static int MEDIUM = 2;

public final static int LARGE = 3;

public final static int EXTRALARGE = 4;

}

Here, the problem arises if we print the constants. It is because only the number is printed which might not be helpful.

So, instead of using int constants, we can simply use enums. For example,

enum Size {

SMALL, MEDIUM, LARGE, EXTRALARGE

}

This makes our code more intuitive.enum provides compile-time type safety. If we declare a variable of the Size type. For example,

Size size;

Here, it is guaranteed that the variable will hold one of the four values. Now, If we try to pass values other than those four values, the compiler will generate an error.

**Java enum Constructor**

n enum class may include a constructor like a regular class. These enum constructors are either

private - accessible within the class  
or

package-private - accessible within the package

enum Size {

// enum constants calling the enum constructors

SMALL("The size is small."),

MEDIUM("The size is medium."),

LARGE("The size is large."),

EXTRALARGE("The size is extra large.");

private final String pizzaSize;

// private enum constructor

private Size(String pizzaSize) {

this.pizzaSize = pizzaSize;

}

public String getSize() {

return pizzaSize;

}

}

class Main {

public static void main(String[] args) {

Size size = Size.SMALL;

System.out.println(size.getSize());

}

}

Java enum Strings

e can get the string representation of enum constants using the toString() method or the name() method. For example,

enum Size {

SMALL, MEDIUM, LARGE, EXTRALARGE

}

class Main {

public static void main(String[] args) {

System.out.println("string value of SMALL is " + Size.SMALL.toString());

System.out.println("string value of MEDIUM is " + Size.MEDIUM.name());

}

}

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.

**Java Exceptions**

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:

The exception hierarchy also has two branches:

1. Runtime Exception
2. IO Exception.

1. 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:

1. Improper use of an API - IllegalArgumentException
2. Null pointer access (missing the initialization of a variable) - NullPointerException
3. Out-of-bounds array access - ArrayIndexOutOfBoundsException
4. Dividing a number by 0 - ArithmeticException

The NullPointerException  would not have occurred if you had checked whether the variable was initialized or not before using it. An ArrayIndexOutOfBoundsException would not have occurred if you tested the array index against the array bounds.

2. 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:

1. Trying to open a file that doesn’t exist results in FileNotFoundException
2. 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.

class Main {

public static void main(String[] args) {

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 keyword

**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

Here is a hierarchy of classes to deal with Input and Output streams.

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

**Collections in Java**

The **Collection in Java** is a framework that provides an architecture to store and manipulate the group of objects. 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.





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. remove() - removes the specified element from the collection
4. iterator() - returns an iterator to access elements of the collection
5. addAll() - adds all the elements of a specified collection to the collection
6. removeAll() - removes all the elements of the specified collection from the collection
7. clear() - removes all the elements of the collection

**Java List:**

In Java, the List interface is an ordered collection that allows us to store and access elements sequentially. It extends the Collection interface. Since List is an interface, we cannot create objects from it. In order to use functionalities of the List interface, we can use these classes:

1. [ArrayList](https://www.programiz.com/java-programming/arraylist)
2. [LinkedList](https://www.programiz.com/java-programming/linkedlist)
3. [Vector](https://www.programiz.com/java-programming/vector)
4. [Stack](https://www.programiz.com/java-programming/stack)

Java ArrayList Class:

The ArrayList class of the Java collections framework provides the functionality of resizable-arrays. Creating an ArrayList Before using ArrayList, we need to import the java.util.ArrayList package first. Here is how we can create arraylists in Java:

ArrayList<Type> arrayList= new ArrayList<>();

**Basic Operations on ArrayList:**

1. Add elements
2. Access elements
3. Change elements
4. Remove elements

**Methods of ArrayList Class:**

1. add()
2. get(),
3. set(),
4. remove()
5. ArrayList
6. [size()](https://www.programiz.com/java-programming/library/arraylist/size)
7. [sort()](https://www.programiz.com/java-programming/library/arraylist/sort)
8. [clone()](https://www.programiz.com/java-programming/library/arraylist/clone)
9. [contains()](https://www.programiz.com/java-programming/library/arraylist/contains)
10. [ensureCapacity()](https://www.programiz.com/java-programming/library/arraylist/ensurecapacity)
11. [isEmpty()](https://www.programiz.com/java-programming/library/arraylist/isempty)
12. [indexOf()](https://www.programiz.com/java-programming/library/arraylist/indexof)

**There are various ways to traverse the collection elements:**

1. By Iterator interface.
2. By for-each loop.
3. By ListIterator interface.
4. By for loop.
5. By forEach() method.
6. By forEachRemaining() method.

For example:

import java.util.ArrayList;

**class Main {**

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

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

**animals. Add("Cow");**

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

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

**for (String language : animals) {**

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

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

**}**

**}**

**}**

Java LinkedList class:

The LinkedList class of the Java collections framework provides the functionality of the linked list data structure (doubly linkedlist). Each element in a linked list is known as a node.

It consists of 3 fields:

1. Prev - stores an address of the previous element in the list. It is null for the first element
2. Next - stores an address of the next element in the list. It is null for the last element
3. Data - stores the actual data

**Creating a Java LinkedList:**

LinkedList<Type> linkedList = new LinkedList<>();

**Methods of Java LinkedList:**

1. add()
2. get()
3. iterator()
4. istIterator()
5. set()
6. remove()
7. contains()
8. indexOf()
9. lastIndexOf()
10. clear()

Java Vector:

The Vector class is an implementation of the List interface that allows us to create resizable-arrays similar to the ArrayList class.

1. Vector is synchronized.
2. Java Vector contains many legacy methods that are not the part of a collection’s framework.

Java Vector class Declaration:

public class Vector<E>

extends Object<E>

implements List<E>, Cloneable, Serializable

Java Vector Methods:

1. [add()](https://www.javatpoint.com/java-vector-add-method)
2. [addAll()](https://www.javatpoint.com/java-vector-addall-method)
3. [addElement()](https://www.javatpoint.com/java-vector-addelement-method)
4. [capacity()](https://www.javatpoint.com/java-vector-capacity-method)
5. [clear()](https://www.javatpoint.com/java-vector-clear-method)
6. [clone()](https://www.javatpoint.com/java-vector-clone-method)
7. [contains()](https://www.javatpoint.com/java-vector-contains-method)
8. [containsAll()](https://www.javatpoint.com/java-vector-containsall-method)
9. [copyInto()](https://www.javatpoint.com/java-vector-copyinto-method)
10. [elementAt()](https://www.javatpoint.com/java-vector-elementat-method)
11. [elements()](https://www.javatpoint.com/java-vector-elements-method)
12. [ensureCapacity()](https://www.javatpoint.com/java-vector-ensurecapacity-method)
13. [equals()](https://www.javatpoint.com/java-vector-equals-method)
14. [firstElement()](https://www.javatpoint.com/java-vector-firstelement-method)
15. [forEach()](https://www.javatpoint.com/java-vector-foreach-method)
16. [get()](https://www.javatpoint.com/java-vector-get-method)
17. [hashCode()](https://www.javatpoint.com/java-vector-hashcode-method)
18. [insertElementAt()](https://www.javatpoint.com/java-vector-insertelementat-method)
19. [isEmpty()](https://www.javatpoint.com/java-vector-isempty-method)
20. [iterator()](https://www.javatpoint.com/java-vector-iterator-method)
21. [lastElement()](https://www.javatpoint.com/java-vector-lastelement-method)
22. [lastIndexOf()](https://www.javatpoint.com/java-vector-lastindexof-method)
23. listIterator()
24. [remove()](https://www.javatpoint.com/java-vector-remove-method)
25. [removeAll()](https://www.javatpoint.com/java-vector-removeall-method)
26. [removeAllElements()](https://www.javatpoint.com/java-vector-removeallelements-method)
27. [removeElement()](https://www.javatpoint.com/java-vector-removeelement-method)
28. [removeElementAt()](https://www.javatpoint.com/java-vector-removeelementat-method)
29. removeIf()
30. removeRange()
31. [replaceAll()](https://www.javatpoint.com/java-vector-replaceall-method)
32. [retainAll()](https://www.javatpoint.com/java-vector-retainall-method)
33. set()
34. setElementAt()
35. setSize()
36. size()
37. sort()
38. spliterator()
39. subList()
40. toArray()
41. toString()
42. trimToSize

Java Stack Class :

The Java collections framework has a class named Stack that provides the functionality of the stack data structure. The Stack class extends the Vector class.

Creating a Stack:

In order to create a stack, we must import the java.util.Stack package first. Once we import the package, here is how we can create a stack in Java.

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

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)

Iterate Elements:

We can fetch elements of the stack using three different methods are as follows:

1. Using iterator() Method
2. Using forEach() Method
3. Using listIterator() Method

Java Queue Interface

The Queue interface of the Java collections framework provides the functionality of the queue data structure. It extends the Collection interface.

Diagram

Description automatically generated

**Methods of Queue:**

1. add()
2. offer()
3. element()
4. peek()
5. remove()

Java PriorityQueue class:

The PriorityQueue class provides the functionality of the heap data structure.

**Creating PriorityQueue:**

In order to create a priority queue, we must import the java.util.PriorityQueue package. Once we import the package, here is how we can create a priority queue in Java.

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

Other PriorityQueue Methods:

1. contains(element)
2. size()
3. toArray()

**Java Deque Interface:**

The Deque interface of the Java collections framework provides the functionality of a double-ended queue. It extends the Queue interface.

Classes that implement Deque-----

1. [ArrayDeque](https://www.programiz.com/java-programming/arraydeque)
2. [LinkedList](https://www.programiz.com/java-programming/linkedlist)

Methods of Deque:

Since Deque extends the Queue interface, it inherits all the methods of the Queue interface.

1. addFirst()
2. addLast()
3. offerFirst()
4. offerLast()
5. getFirst()
6. getLast()
7. peekFirst()
8. peekLast()
9. removeFirst()
10. removeLast()
11. pollFirst()
12. pollLast()

**Java LinkedList:**

**---------- Discuss Before -----------**

**Java ArrayDeque:**

Interfaces implemented by ArrayDeque. The ArrayDeque class implements these two interfaces:

1. [Java Queue Interface](https://www.programiz.com/java-programming/queue)
2. [Java Deque Interface](https://www.programiz.com/java-programming/deque)

**Creating ArrayDeque:**

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

Java Map Interface:

The Map interface of the Java collections framework provides the functionality of the map data structure. A map contains values on the basis of key, i.e. key and value pair. Each key and value pair is known as an entry. A Map contains unique keys.

Java Map Hierarchy:

The hierarchy of Java Map is given below:

1. [HashMap](https://www.programiz.com/java-programming/hashmap)
2. [EnumMap](https://www.programiz.com/java-programming/enummap)
3. [LinkedHashMap](https://www.programiz.com/java-programming/linkedhashmap)
4. [WeakHashMap](https://www.programiz.com/java-programming/weakhashmap)
5. [TreeMap](https://www.programiz.com/java-programming/treemap)

These classes are defined in the collections framework and implement the Map interface.



Interfaces that extend Map

The Map interface is also extended by these subinterfaces:

1. [SortedMap](https://www.programiz.com/java-programming/sortedmap)
2. [NavigableMap](https://www.programiz.com/java-programming/navigablemap)
3. [ConcurrentMap](https://www.programiz.com/java-programming/concurrentmap)



How to use Map?

In Java, we must import the java.util.Map package in order to use Map. Once we import the package,

Map<Key, Value> numbers = new HashMap<>();

Methods of Map:

1. put(K, V)
2. putAll()
3. putIfAbsent(K, V)
4. get(K)
5. getOrDefault(K, defaultValue)
6. containsKey(K)
7. containsValue(V)
8. replace(K, V)
9. replace(K, oldValue, newValue)
10. remove(K)
11. remove(K, V)
12. keySet()
13. values()
14. entrySet()

# **Java JDBC Tutorial**

JDBC stands for Java Database Connectivity. JDBC is a Java API to connect and execute the query with the database. It is a part of JavaSE (Java Standard Edition). JDBC API uses JDBC drivers to connect with the database.