Unit: 1

Introduction

Java is an object-oriented, platform independent, class-based, concurrent, secured and general-purpose computer-programming language. It is a widely used robust technology.

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. James Gosling is known as the father of Java. Before Java, its name was Oak. Since Oak was already a registered company, so James Gosling and his team changed the Oak name to Java.

→ **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. Currently, Servlet, JSP, Struts, Spring, Hibernate, 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 enterprise application. It has advantages of the 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.

Java Platforms / Editions

There are 4 platforms or editions of Java:

1) Java SE (Java Standard Edition): J2SE

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):J2EE

It is an enterprise platform which is mainly used to develop web and enterprise applications. It is built on the top of the Java SE platform. It includes topics like Servlet, JSP, Web Services, EJB, JPA, etc.

3) Java ME (Java Micro Edition):J2ME

It is a micro platform which is mainly used to develop mobile applications.

4) JavaFX

It is used to develop rich internet applications. It uses a light-weight user interface API.

History of Java

The history of Java is very interesting. Java was originally designed for interactive television, but it was too advanced technology for the digital cable television industry at the time. The history of Java starts with the Green Team. Java team members (also known as **Green Team**), initiated this project to develop a language for digital devices such as set-top boxes, televisions, etc. However, it was suited for internet programming. Later, Java technology was incorporated by Netscape.

The principles for creating Java programming were "Simple, Robust, Portable, Platform-independent, Secured, High Performance, Multithreaded, Architecture Neutral, Object-Oriented, Interpreted, and Dynamic". Java was developed by James Gosling, who is known as the father of Java, in 1995. James Gosling and his team members started the project in the early '90s.

Currently, Java is used in internet programming, mobile devices, games, e-business solutions, etc. There are given significant points that describe the history of Java.

→ **James Gosling**, **Mike Sheridan**, and **Patrick Naughton** initiated the Java language project in June 1991. The small team of sun engineers called **Green Team**.
→ Initially designed for small, embedded systems in electronic appliances like set-top boxes.
→ Firstly, it was called **"Greentalk"** by James Gosling, and the file extension was .gt.
→ After that, it was called **Oak** and was developed as a part of the Green project.
→ In 1995, Oak was renamed as **"Java"** because it was already a trademark by Oak Technologies. Java is an island of Indonesia where the first coffee was produced (called java coffee). It is a kind of espresso bean. Java name was chosen by James Gosling while having coffee near his office. Notice that Java is just a name, not an acronym.
→ Initially developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995.
→ In 1995, Time magazine called **Java one of the Ten Best Products of 1995**.
→ JDK 1.0 released in(January 23, 1996). After the first release of Java, there have been many additional features added to the language. Now Java is being used in Windows applications, Web applications, enterprise applications, mobile applications, cards, etc. Each new version adds the new features in Java.

Version	Release Date	Major changes
OAK	1991	OAK Language Developed
JDK Beta	1995	OAK rename was "JAVA"

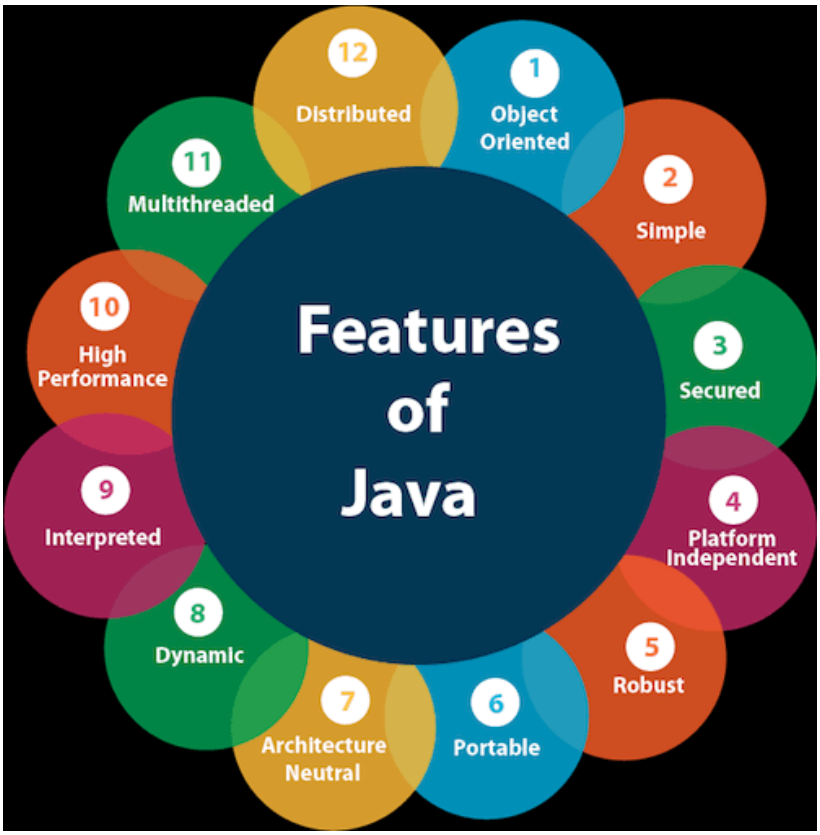
JDK 1.0	January 1996	The Very first version was released on January 23, 1996. The principal stable variant, JDK(Java Development Kit) 1.0.2, is called Java 1.
JDK 1.1	February 1997	Was released on February 19, 1997. There were many additions in JDK 1.1 as compared to version 1.0 such as A broad retooling of the AWT occasion show Inner classes added to the language JavaBeans JDBC RMI
J2SE 1.2	December 1998	"Play area" was the codename which was given to this form and was released on 8th December 1998. Its real expansion included: strictfp keyword the Swing graphical API was coordinated into the centre classes Sun's JVM was outfitted with a JIT compiler out of the blue Java module Java IDL, an IDL usage for CORBA interoperability Collections system
J2SE 1.3	May 2000	Codename- "KESTREL" Release Date- 8th May 2000 Additions: HotSpot JVM included Java Naming and Directory Interface JPDA JavaSound Synthetic proxy classes
J2SE 1.4	February 2002	Codename- "Merlin" Release Date- 6th February 2002 Additions: Library improvements Regular expressions modelled after Perl regular expressions The image I/O API for reading and writing images in formats like JPEG and PNG Integrated XML parser and XSLT processor (JAXP) (specified in JSR 5 and JSR 63) Preferences API (java.util.prefs) Public Support and security updates for this version ended in October 2008.
J2SE 5.0	September 2004	Codename- "Tiger" Release Date- "30th September 2004" Originally numbered as 1.5 which is still used as its internal version. Added several new language features such as: for-each loop Generics Autoboxing Var-args
JAVA SE 6	December 2006	Codename- "Mustang" Released Date- 11th December 2006 Packaged with a database supervisor and encourages the utilization of scripting languages with the JVM. Replaced the name J2SE with ava SE and dropped the .0 from the version number. Additions: Upgrade of JAXB to version 2.0: Including integration of a StAX parser. Support for pluggable annotations (JSR 269). JDBC 4.0 support (JSR 221)
JAVA SE 7	July 2011	Codename- "Dolphin" Release Date- 7th July 2011 Added small language changes including strings in the switch. The JVM was extended with support for dynamic languages. Additions:

		Compressed 64-bit pointers. Binary Integer Literals. Upstream updates to XML and Unicode.
JAVA SE 8	March 2014	Released Date- 18th March 2014 Language level support for lambda expressions and default methods and a new date and time API inspired by Joda Time.
JAVA SE 9	September 2017	Release Date: 21st September 2017 Project Jigsaw: designing and implementing a standard, a module system for the Java SE platform, and to apply that system to the platform itself and the JDK.
JAVA SE 10	March 2018	Released Date- 20th March Addition: Additional Unicode language-tag extensions Root certificates Thread-local handshakes Heap allocation on alternative memory devices Remove the native-header generation tool – javah. Consolidate the JDK forest into a single repository.
JAVA SE 11	September 2018	Released Date- 25th September, 2018 Additions- Dynamic class-file constants Epsilon: a no-op garbage collector The local-variable syntax for lambda parameters Low-overhead heap profiling HTTP client (standard) Transport Layer Security (TLS) 1.3 Flight recorder
JAVA SE 12	March 2019	Released Date- 19th March 2019 Additions- Shenandoah: A Low-Pause-Time Garbage Collector (Experimental) Microbenchmark Suite Switch Expressions (Preview) JVM Constants API One AArch64 Port, Not Two Default CDS Archives
JAVA SE 13	September 2019	Released Date- 17th September 2019 Additions- Text Blocks – JEP 355. New Methods in String Class for Text Blocks. Switch Expressions Enhancements – JEP 354. Reimplement the Legacy Socket API – JEP 353. Dynamic CDS Archive – JEP 350. ZGC: Uncommit Unused Memory – JEP 351. Support for Unicode 12.1.
JAVA SE 14	March 2020	Released Date- 17th March 2020 Additions- Pattern Matching for instanceof (Preview) Packaging Tool (Incubator) NUMA-Aware Memory Allocation for G1. JFR Event Streaming. Non-Volatile Mapped Byte Buffers. Helpful NullPointerExceptions. Records (Preview) Switch Expressions (Standard)
JAVA SE 15	September	Released Date- 15th March 2020

	2020	Additions- Edwards-Curve Digital Signature Algorithm (EdDSA) Sealed Classes (Preview) Hidden Classes Remove the Nashorn JavaScript Engine Reimplement the Legacy DatagramSocket API Disable and Deprecate Biased Locking Pattern Matching for instanceof (Second Preview) ZGC: A Scalable Low-Latency Garbage Collector Text Blocks Shenandoah: A Low-Pause-Time Garbage Collector Foreign-Memory Access API (Second Incubator) Records (Second Preview) Deprecate RMI Activation for Removal
JAVA SE 16	March 2021	JDK 16 was released on March 16, 2021. Java 16 removes Ahead-of-Time compilation (and Graal JIT) option. The Java implementation itself was and is still written in C++, while as of Java 16, more recent C++14 (but still not e.g. C++17 or C++20) is allowed. The code was also moved to GitHub (dropping the Mercurial source control system).
JAVA SE 17	September 2021	JDK 17 is the current long-term support (LTS) release since September 2021. Java 17 is the 2nd long-term support (LTS) release since switching to the new 6-month release cadence (the first being Java 11).

Features of Java (Buzzwords)

The primary objective of Java programming language creation was to make it portable, simple and secure programming language. Apart from this, there are also some excellent features which play an important role in the popularity of this language. The features of Java are also known as java *buzzwords*.



Simple

→ Java is very easy to learn, and its syntax is simple, clean and easy to understand. According to Sun, Java language is a simple programming language because:

- Java syntax is based on C++ (so easier for programmers to learn it after C++).
- Java has removed many complicated and rarely-used features, for example, explicit pointers, operator overloading, etc.
- There is no need to remove unreferenced objects because there is an Automatic Garbage Collection in Java.

Object-oriented

→ Java is an object-oriented programming language. Everything in Java is an object. Object-oriented means we organize our software as a combination of different types of objects that incorporates both data and behavior.

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

→ Basic concepts of OOPs are:

1. Object
2. Class
3. Inheritance
4. Polymorphism
5. Abstraction
6. Encapsulation

Platform Independent

→ Java is platform independent because it is different from other languages like C, C++, etc. which are compiled into platform specific machines while Java is a write once, run anywhere language. A platform is the hardware or software environment in which a program runs.

→ There are two types of platforms software-based and hardware-based. Java provides a software-based platform.

→ The Java platform differs from most other platforms in the sense that it is a software-based platform that runs on the top of other hardware-based platforms. It has two components:

1. Runtime Environment
2. API(Application Programming Interface)

→ Java code can be run on multiple platforms, for example, Windows, Linux, Sun Solaris, Mac/OS, etc. Java code is compiled by the compiler and converted into bytecode. This bytecode is a platform-independent code because it can be run on multiple platforms, i.e., Write Once and Run Anywhere(WORA).

Secure

→ Java is best known for its security. With Java, we can develop virus-free systems. Java is secured because:

- **No explicit pointer**
- **Java Programs run inside a virtual machine sandbox**
- **Classloader:** Classloader in Java is a part of the Java Runtime Environment(JRE) which is used to load Java classes into the Java Virtual Machine dynamically. It adds security by separating the package for the classes of the local file system from those that are imported from network sources.
- **Bytecode Verifier:** It checks the code fragments for illegal code that can violate access right to objects.
- **Security Manager:** It determines what resources a class can access such as reading and writing to the local disk.

→ Java language provides these securities by default. Some security can also be provided by an application developer explicitly through SSL, JAAS, Cryptography, etc.

Robust

→ Robust simply means strong. Java is robust because:

- It uses strong memory management.
- There is a lack of pointers that avoids security problems.
- There is automatic garbage collection in java which runs on the Java Virtual Machine to get rid of objects which are not being used by a Java application anymore.
- There are exception handling and the type checking mechanism in Java. All these points make Java robust.

Architecture-neutral

→ When Java is compiled, it is not compiled into platform specific machine, rather into platform-independent byte code. This byte code is distributed over the web and interpreted by the Virtual Machine (JVM) on whichever platform it is being run on.

→ Thus when we write a piece of Java code in a particular platform and generated an executable code .class file. We can execute/run this .class file on any system the only condition is that the target system should have JVM (JRE) installed in it.

→ In short, Java compiler generates an architecture-neutral object file format, which makes the compiled code executable on many processors, with the presence of Java runtime system.

Portable

→ Java is portable because it facilitates we to carry the Java bytecode to any platform. It doesn't require any implementation.

High-performance

→ Java is faster than other traditional interpreted programming languages because Java bytecode is “close” to native code. It is still a little bit slower than a compiled language (e.g., C++). Java is an interpreted language that is why it is slower than compiled languages, e.g., C, C++, etc.

Distributed

→ Java is distributed because it facilitates users to create distributed applications in Java. RMI and EJB are used for creating distributed applications. This feature of Java makes us able to access files by calling the methods from any machine on the internet.

Multi-threaded

→ Java allows us to develop a program that can do multiple task simultaneously using thread.

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

Dynamic

→ Java is a dynamic language. It supports dynamic loading of classes. It means classes are loaded on demand. It also supports functions from its native languages, i.e., C and C++.

→ Java supports dynamic compilation and automatic memory management (garbage collection).

Interpreted (line by line code checking)

→ JVM interpreted the byte code into machine instruction during run time.

Principles of Java

- It must be simple, object oriented and familiar.
- It must be robust and secure.
- It must be architecture neutral and portable.

- It must be execute with high performance.
- It must be interpreted, multi threaded and dynamic.

JDK, JRE, JVM

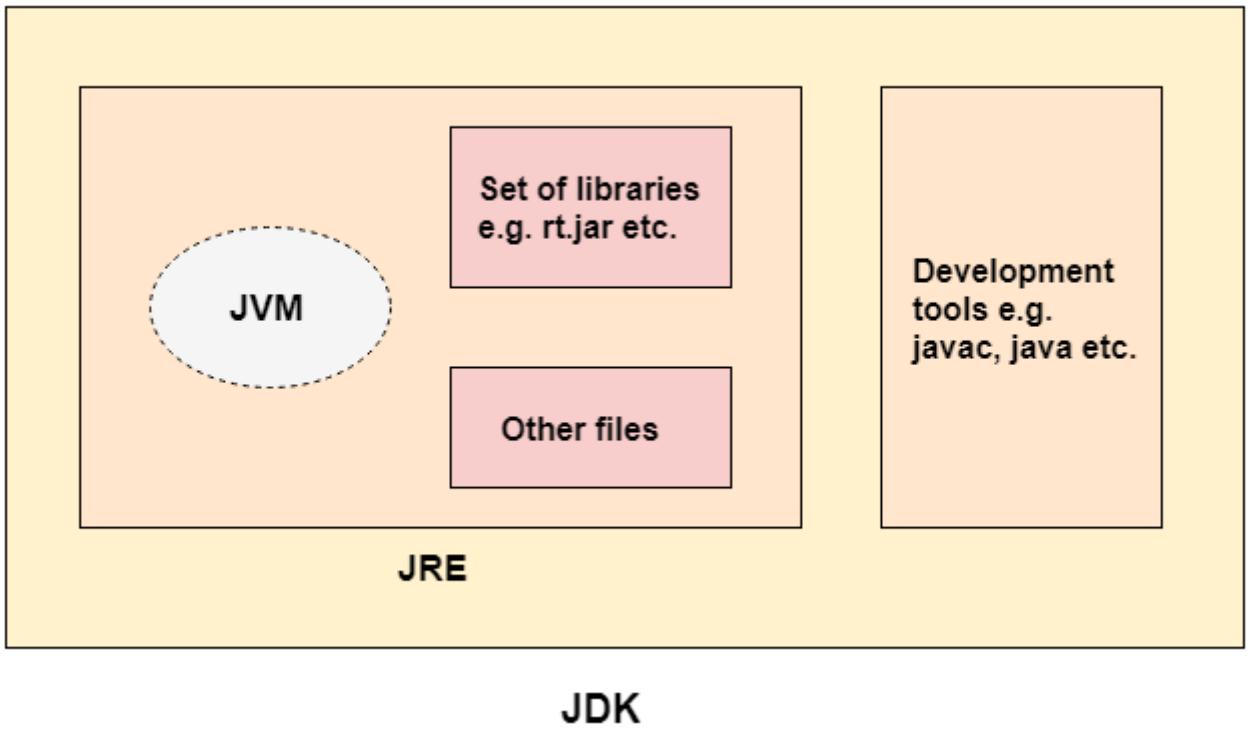
JDK

→ JDK is an acronym for Java Development Kit. The Java Development Kit (JDK) is a software development environment which is used to develop Java applications and applets. It physically exists. It contains JRE + development tools.

→ JDK is an implementation of any one of the below given Java Platforms released by Oracle Corporation:
- Standard Edition Java Platform
- Enterprise Edition Java Platform
- Micro Edition Java Platform

→ The JDK contains a private Java Virtual Machine (JVM) and a few other resources such as an interpreter/loader (java), a compiler (javac), an archiver (jar), a documentation generator (Javadoc), etc. to complete the development of a Java Application.

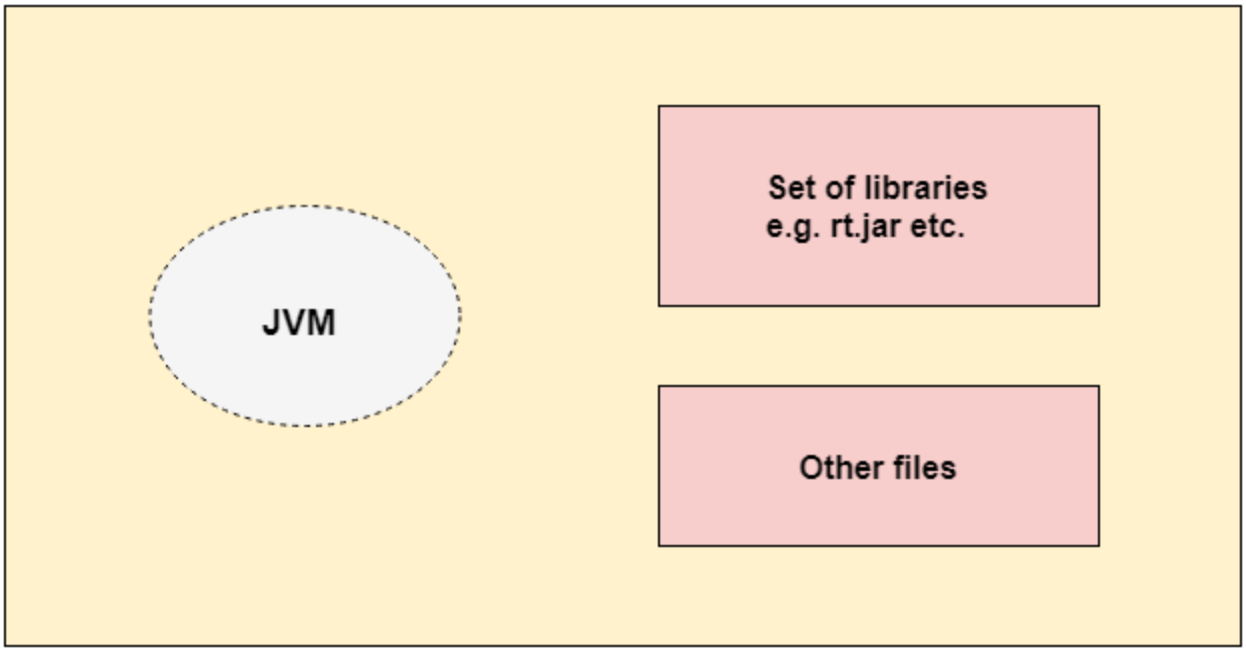
→ JDK Tools(Components)
- Appletwiever
- javac(Java Compiler)
- java (Kava Interpreter)
- javap(java dissembler)
- javah(for c header file)
- JDB(Java debugger)
- RMIC(Remote Method Invocation Compiler)
- RMI registry
- JAR(JAVA archiver)



JRE

→ JRE is an acronym for Java Runtime Environment. It is also written as Java RTE. The Java Runtime Environment is a set of software tools which are used for developing Java applications. It is used to provide the runtime environment. It is the implementation of JVM. It physically exists. It contains a set of libraries + other files that JVM uses at runtime.

→ The implementation of JVM is also actively released by other companies besides Sun Micro Systems.



JRE

JVM

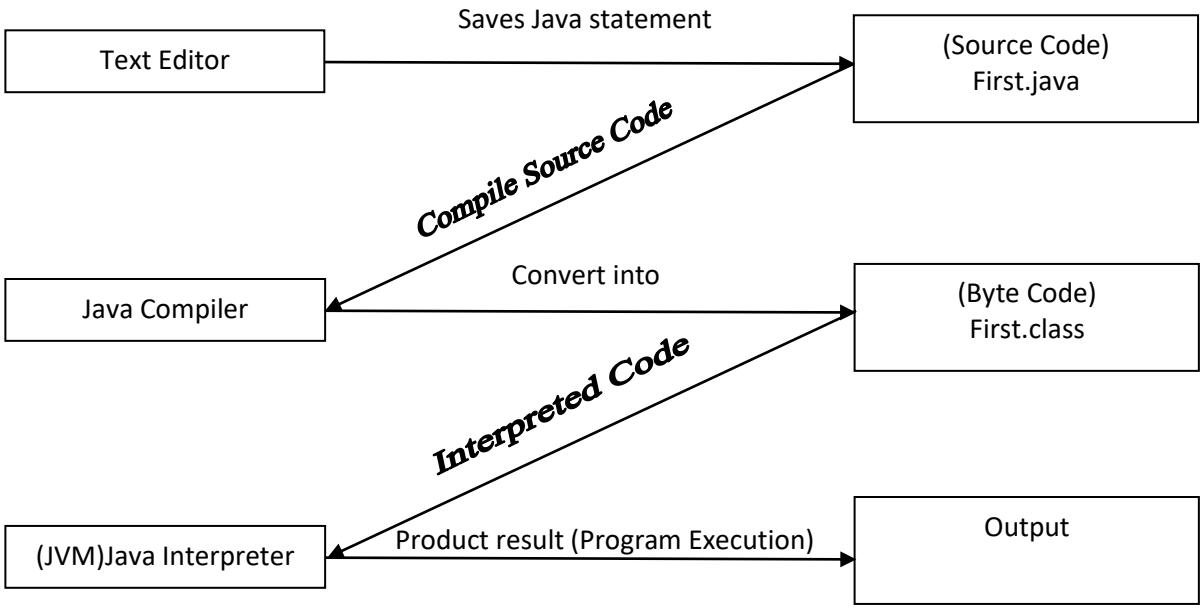
→ JVM (Java Virtual Machine) is an abstract machine. It is called a virtual machine because it doesn't physically exist. It is a specification that provides a runtime environment in which Java bytecode can be executed. It can also run those programs which are written in other languages and compiled to Java bytecode.

→ JVMs are available for many hardware and software platforms. JVM, JRE, and JDK are platform dependent because the configuration of each OS is different from each other. However, Java is platform independent. There are three notions (Concept) of the JVM: *specification*, *implementation*, and *instance*.

→ The JVM performs the following main tasks:
- Loads code
- Verifies code
- Executes code
- Provides runtime environment

→ JVM provides definitions for the:
- Memory area
- Class file format
- Register set
- Garbage-collected heap
- Fatal error reporting etc.

Compiling and Executing Basic Java Program:



1) Written the source code of java using any text editor and save it with ".java" extension.
   Example:
   => **Creating First.java file**

```
class First{
   public static void main(String args[]){
    System.out.println("Welcome TO Java");
   }
}
```

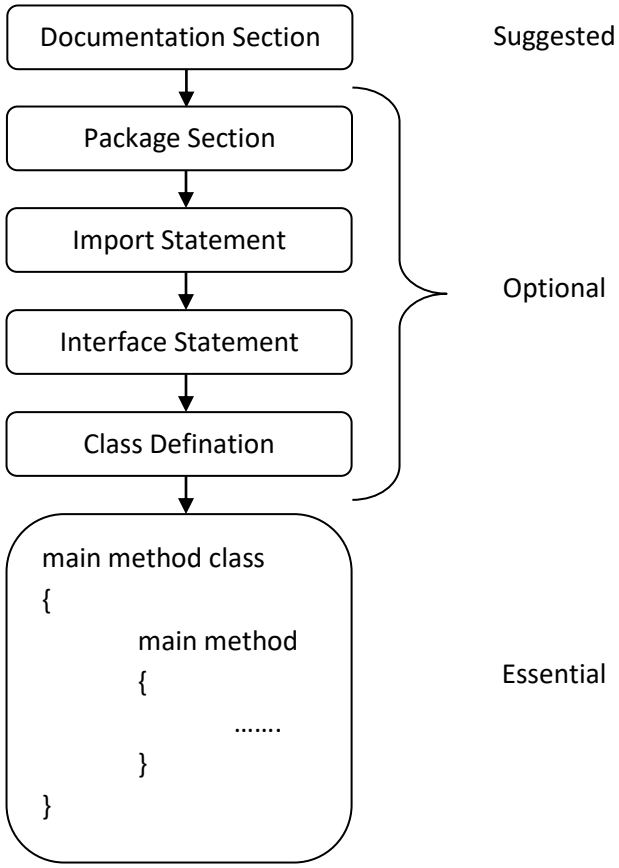
   - ✓ **class** keyword is used to declare a class in java.
   - ✓ **public** keyword is an access modifier which represents visibility. It means it is visible to all.
   - ✓ **static** is a keyword. If we declare any method as static, it is known as the static method. The core advantage of the static method is that there is no need to create an object to invoke the static method. The main method is executed by the JVM, so it doesn't require to create an object to invoke the main method. So it saves memory.
   - ✓ **void** is the return type of the method. It means it doesn't return any value.
   - ✓ **main** is method of java program. represents the starting point of the program.
   - ✓ **String[] args** is used for command line argument.
   - ✓ **System.out.println()** is used to print statement. Here, System is a class, out is the object of PrintStream class, println() is the method of PrintStream class. We will learn about the internal working of System.out.println statement later.

2) Compile Source code with java compiler that is "javac", which generate class file.
   To compile: javac First.java

3) A class file contain byte code which is interpreted by java interpreter that is "java" and produce result as a output.
   To execute: java First

Structure of Java Program

Java is an object-oriented programming, platform-independent, and secure programming language that makes it popular. Using the Java programming language, we can develop a wide variety of applications. So, before diving in depth, it is necessary to understand the basic structure of Java program in detail. In this section, we have discussed the basic structure of a Java program. At the end of this section, we will able to develop the Hello world Java program, easily.



1) Documentation Section

→ The documentation section is an important section but optional for a Java program. It includes **basic information** about a Java program. The information includes the **author's name, date of creation, version, program name, company name,** and **description** of the program. It improves the readability of the program. Whatever we write in the documentation section, the Java compiler ignores the statements during the execution of the program. To write the statements in the documentation section, we use **comments**. The comments may be **single-line, multi-line,** and **documentation** comments.

- ✓ **Single-line Comment:** It starts with a pair of forwarding slash (//). For example:
  - //First Java Program
- ✓ **Multi-line Comment:** It starts with a /* and ends with */. We write between these two symbols. For example:
  - /*It is an example of
    multiline comment*/
- ✓ **Documentation Comment:** It starts with the delimiter (/**) and ends with */. For example:
  - /**It is an example of documentation comment*/

2) Package Declaration

→ The package declaration is optional. It is placed just after the documentation section. In this section, we declare the package name in which the class is placed. Note that there can be only one package statement in a Java program. It must be defined before any class and interface declaration. It is necessary because a Java class can be placed in different packages and directories based on the module they are used. For all these classes package belongs to a single parent directory. We use the keyword package to declare the package name.

→ For example:

- ✓ **package firstpkg;** //where firstpkg is the package name
- ✓ **package first.firstpkg;** //where first is the root directory and firstpkg is the subdirectory

3) Import Statements

→ The package contains the many predefined classes and interfaces. If we want to use any class of a particular package, we need to import that class. The import statement represents the class stored in the other package. We use the import keyword to import the class. It is written before the class declaration and after the package statement. We use the import statement in two ways, either import a specific class or import all classes of a particular package. In a Java program, we can use multiple import statements.

→ For example:

```
import java.util.Scanner; //it imports the Scanner class only
import java.util.*; //it imports all the class of the java.util package
```

4) Interface Section

→ It is an optional section. We can create an interface in this section if required. We use the interface keyword to create an interface. An interface is a slightly different from the class. It contains only constants and method declarations. Another difference is that it cannot be instantiated. We can use interface in classes by using the implements keyword. An interface can also be used with other interfaces by using the extends keyword.

→ For example:

```
interface car
{
    void start();
    void stop();
}
```

5) Class Definition

→ In this section, we define the class. It is vital part of a Java program. Without the class, we cannot create any Java program. A Java program may conation more than one class definition. We use the class keyword to define the class. The class is a blueprint of a Java program. It contains information about user-defined methods, variables, and constants. Every Java program has at least one class that contains the main() method.

→ For example:

```
class Student //class definition
{
}
```

6) Main Method Class

→ In this section, we define the main() method. It is essential for all Java programs. Because the execution of all Java programs starts from the main() method. In other words, it is an entry point of the class. It must be inside the class. Inside the main method, we create objects and call the methods. We use the following statement to define the main() method:

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

→ For example:

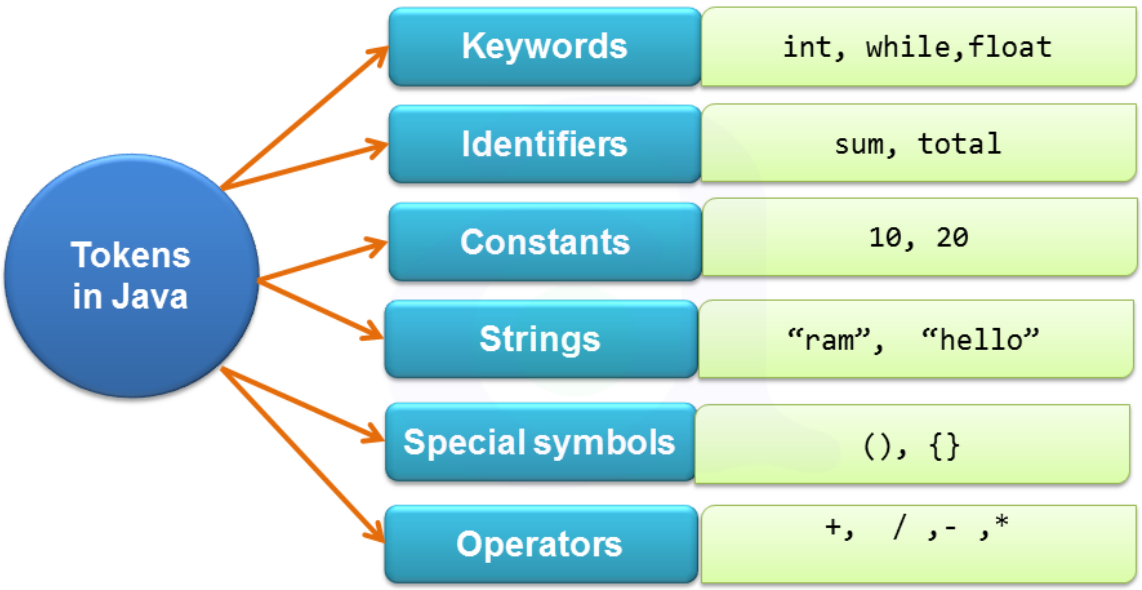
```
public class Student //class definition
{
    public static void main(String args[])
    {
        //statements
    }
}
```

Java Tokens

→ In Java, the program contains classes and methods. Further, the methods contain the expressions and statements required to perform a specific operation. These statements and expressions are made up of tokens. In other words, we can say that the expression and statement is a set of tokens. The tokens are the small building blocks of a Java program that are meaningful to the Java compiler. Further, these two components contain variables, constants, and operators. In this section, we will discuss what is tokens in Java.

→ **What is token in Java?**

The Java compiler breaks the line of code into text (words) is called Java tokens. These are the smallest element of the Java program. The Java compiler identified these words as tokens. These tokens are separated by the delimiters. It is useful for compilers to detect errors. Remember that the delimiters are not part of the Java tokens.



Java token includes the following:

1. Keywords
2. Identifiers
3. Literals
4. Operators
5. Separators
6. Comments

1) Keywords:

These are the **pre-defined** reserved words of any programming language. Each keyword has a special meaning. It is always written in lower case. Java language has reserved 50 words as keywords Java provides the following keywords: Abstract,Boolean,break,continue,int,byte,short,float,double,String,chat,try,catch,finally,final,interface,class,switch,case etc…

2) Identifier:

Identifiers are used to name a variable, constant, function, class, and array. It usually defined by the user. It uses letters, underscores, or a dollar sign as the first character. The label is also known as a special kind of identifier that is used in the goto statement. Remember that the identifier name must be different from the reserved keywords.

There are some rules to declare identifiers are:

- The first letter of an identifier must be a letter, underscore or a dollar sign. It cannot start with digits but may contain digits.
- The whitespace cannot be included in the identifier.
- Identifiers are case sensitive.
- Identifier name must be different from reserved words.
- Length of 65535

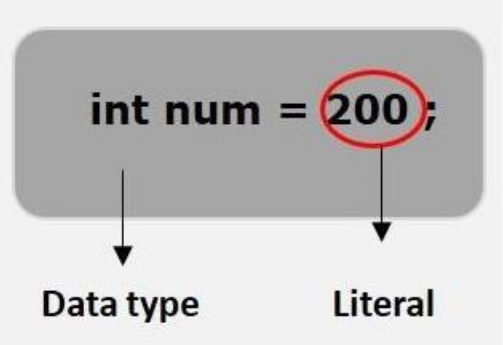
3) Whitespace:

It consist of space,tab,newline etc…

All occurrences of space, tab, newline are remove by java compiler.

4) Literals:

In programming literal is a notation that represents a fixed value (constant) in the source code. It can be categorized as an integer literal, string literal, Boolean literal, etc. It is defined by the programmer. Once it has been defined cannot be changed.



Java provides five types of literals are as follows:

- Integer
- Floating Point
- Character
- String

- Boolean

I. Integral literals

For Integral data types (byte, short, int, long), we can specify literals in 4 ways:-

- **Decimal literals (Base 10):** In this form the allowed digits are 0-9.
  ```
  int x = 101;
  ```
- **Octal literals (Base 8):** In this form the allowed digits are 0-7.
  ```
  // The octal number should be prefix with 0.
  int x = 0146;
  ```
- **Hexa-decimal literals (Base 16) :** In this form the allowed digits are 0-9 and characters are a-f. We can use both uppercase and lowercase characters. As we know that java is a case-sensitive programming language but here java is not case-sensitive.
  ```
  // The hexa-decimal number should be prefix with 0X or 0x.
  int x = 0X123Face;
  ```
- **Binary literals :** In this form the allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.
  ```
  int x = 0b1111;
  ```

II. Floating-Point literal

For Floating-point data types, we can specify literals in only decimal form and we can't specify in octal and Hexa-decimal forms.

- **Decimal literals(Base 10) :** In this form the allowed digits are 0-9.
  ```
  double d = 123.456;
  ```

III. Char literal

For char data types we can specify literals in 4 ways:

- **Single quote :** We can specify literal to char data type as single character within single quote.
  ```
  char ch = 'a';
  ```
- **Char literal as Integral literal :** we can specify char literal as integral literal which represents Unicode value of the character and that integral literals can be specified either in Decimal, Octal and Hexadecimal forms. But the allowed range is 0 to 65535.
  ```
  char ch = 062;
  ```
- **Unicode Representation :** We can specify char literals in Unicode representation '\uxxxx'. Here xxxx represents 4 hexadecimal numbers.
  ```
  char ch = '\u0061';// Here /u0061 represent "a".
  ```
- **Escape Sequence :** Every escape character can be specify as char literals.
  ```
  char ch = '\n';
  ```

IV. String literal

Any sequence of characters within double quotes is treated as String literals.

```
String s = "Hello";
```

V. boolean literals

Only two values are allowed for Boolean literals i.e. true and false.

```
boolean b = true;
```

5) Operators:

In programming, operators are the special symbol that tells the compiler to perform a special operation. Java provides different types of operators that can be classified according to the functionality they provide. There are eight types of operators in Java, are as follows:

Operator	**Symbols**
Arithmetic	+ , - , / , * , %
Unary	++ , - - , !
Assignment	= , += , -= , *= , /= , %= , ^=

Relational	==, != , < , >, <= , >=
Logical	&& , ||,!
Ternary	(Condition) ? (Statement1) : (Statement2);
Bitwise(Boolean)	& ,	, ^ , ~
Shift	<< , >> , >>>
Special Operator	new,.,(),(type)

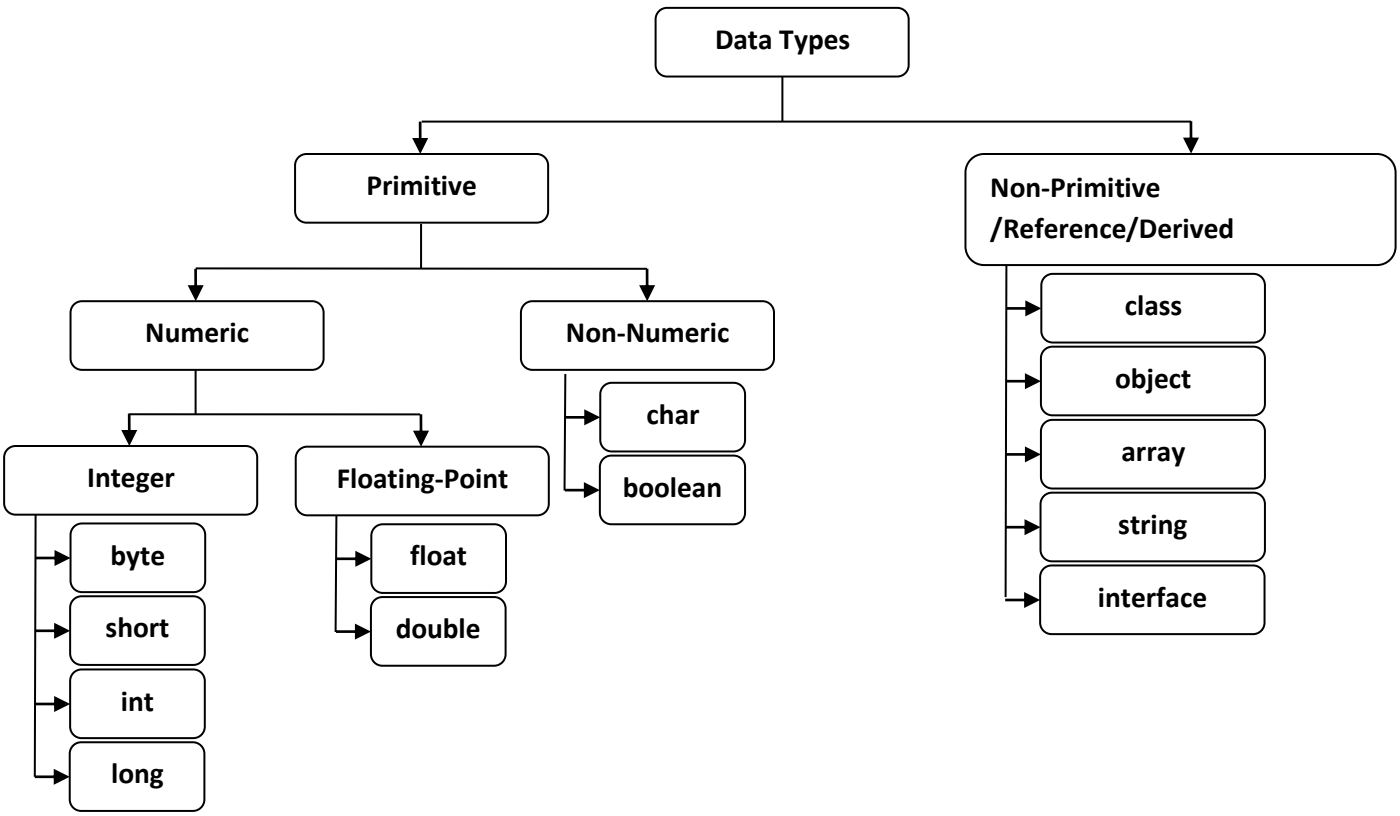
6) **Separators:** The separators in Java is also known as **punctuators**. There are nine separators in Java, are as follows: separator <= ; | , | . | (|) | { | } | [|]
   - **Square Brackets []:** It is used to define array elements. A pair of square brackets represents the single-dimensional array, two pairs of square brackets represent the two-dimensional array.
   - **Parentheses ():** It is used to call the functions and parsing the parameters.
   - **Curly Braces {}:** The curly braces denote the starting and ending of a code block.
   - **Comma (,):** It is used to separate two values, statements, and parameters.
   - **Assignment Operator (=):** It is used to assign a variable and constant.
   - **Semicolon (;):** It is the symbol that can be found at end of the statements. It separates the two statements.
   - **Period (.):** It separates the package name form the sub-packages and class. It also separates a variable or method from a reference variable.

7) **Comments:** Comments allow us to specify information about the program inside our Java code. Java compiler recognizes these comments as tokens but excludes it form further processing. The Java compiler treats comments as whitespaces. Java provides the following two types of comments:
   - Line Oriented: **It begins with a pair of forwarding slashes (//).**
   - Block-Oriented: **It begins with /* and continues until it founds** */**.**

Data Types

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

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



Primitive Data Types

→ There are eight primitive datatypes supported by Java. Primitive datatypes are predefined by the language and named by a keyword. Let us now look into the eight primitive data types in detail.

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

Boolean Data Type

→ The Boolean data type is used to store only two possible values: true and false. This data type is used for simple flags that track true/false conditions.
→ The Boolean data type specifies one bit of information, but its “size” can't be defined precisely.
→ **Example:** Boolean one = false

Byte Data Type

→ The byte data type is an example of primitive data type. It isan 8-bit signed two's complement integer. Its value-range lies between -128 to 127 (inclusive). Its minimum value is -128 and maximum value is 127. Its default value is 0.
→ The byte data type is used to save memory in large arrays where the memory savings is most required. It saves space because a byte is 4 times smaller than an integer. It can also be used in place of “int” data type.
→ **Example:** byte a = 10, byte b = -20

Short Data Type

→ The short data type is a 16-bit signed two's complement integer. Its value-range lies between -32,768 to 32,767 (inclusive). Its minimum value is -32,768 and maximum value is 32,767. Its default value is 0.
→ The short data type can also be used to save memory just like byte data type. A short data type is 2 times smaller than an integer.
→ **Example:** short s = 10000, short r = -5000

Int Data Type

→ The int data type is a 32-bit signed two's complement integer. Its value-range lies between - 2,147,483,648 (-2^31) to 2,147,483,647 (2^31 -1) (inclusive). Its minimum value is - 2,147,483,648and maximum value is 2,147,483,647. Its default value is 0.
→ The int data type is generally used as a default data type for integral values unless if there is no problem about memory.
→ **Example:** int a = 100000, int b = -200000

Long Data Type

→ The long data type is a 64-bit two's complement integer. Its value-range lies between -9,223,372,036,854,775,808(-2^63) to 9,223,372,036,854,775,807(2^63 -1)(inclusive). Its minimum value is - 9,223,372,036,854,775,808and

maximum value is 9,223,372,036,854,775,807. Its default value is 0. The long data type is used when we need a range of values more than those provided by int.

→ **Example:** long a = 100000L, long b = -200000L

Float Data Type

→ The float data type is a single-precision 32-bit IEEE 754 floating point.Its value range is unlimited. It is recommended to use a float (instead of double) if we need to save memory in large arrays of floating point numbers. The float data type should never be used for precise values, such as currency. Its default value is 0.0F.

→ **Example:** float f1 = 234.5f

Double Data Type

→ The double data type is a double-precision 64-bit IEEE 754 floating point. Its value range is unlimited. The double data type is generally used for decimal values just like float. The double data type also should never be used for precise values, such as currency. Its default value is 0.0d.

→ **Example:** double d1 = 12.3

Char Data Type

→ The char data type is a single 16-bit Unicode character. Its value-range lies between '\u0000' (or 0) to '\uffff' (or 65,535 inclusive).The char data type is used to store characters. [Note: Why char uses 2 byte in java ? It is because java uses Unicode system not ASCII code system. The \u0000 is the lowest range of Unicode system.]

→ **Example:** char letterA = 'A'

Type Casting

→ In Java, **type casting** is a method or process that converts a data type into another data type in both ways manually and automatically. The automatic conversion is done by the compiler and manual conversion performed by the programmer.

→ there are two types of casting:

- Widening Casting (automatically) - converting a smaller type to a larger type size
  byte -> short -> char -> int -> long -> float -> double

- Narrowing Casting (manually) - converting a larger type to a smaller size type
  double -> float -> long -> int -> char -> short -> byte

1) Widening or Automatic Type Conversion

→ Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign value of a smaller data type to a bigger data type.

→ For Example, in java the numeric data types are compatible with each other but no automatic conversion is supported from numeric type to char or boolean. Also, char and boolean are not compatible with each other.

→ Example:

```
class WideningTypeCastingEx
{
        public static void main(String[] args)
        {
                int i = 100;
                    // automatic type conversion
                long l = i;
```

```
                // automatic type conversion
                float f = l;
                System.out.println("Int value "+i);
                System.out.println("Long value "+l);
                System.out.println("Float value "+f);
        }
}
```

2) Narrowing or Explicit Conversion

→ Converting a higher data type into a lower one is called **narrowing** type casting. It is also known as **explicit conversion** or **casting up**. It is done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

→ If we want to assign a value of larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, target-type specifies the desired type to convert the specified value to.

→ Example:

```
public class NarrowingTypeCastingExample
{
        public static void main(String args[])
        {
                double d = 166.66;
                //converting double data type into long data type
                long l = (long)d;
                //converting long data type into int data type
                int i = (int)l;
                System.out.println("Before conversion: "+d);
                //fractional part lost
                System.out.println("After conversion into long type: "+l);
                //fractional part lost
                System.out.println("After conversion into int type: "+i);
        }
}
```

Operators in Java

→ Operators constitute the basic building block to any programming language. Java too provides many types of operators which can be used according to the need to perform various calculation and functions be it logical, arithmetic, relational etc. They are classified based on the functionality they provide. Here are a few types:

- Arithmetic Operators
- Unary Operators
- Assignment Operator
- Relational Operators
- Logical Operators
- Ternary Operator
- Bitwise Operators
- Shift Operators

1) Arithmetic Operators

→ These operators involve the mathematical operators that can be used to perform various simple or advance arithmetic operations on the primitive data types referred to as the operands. These operators consist of various unary and binary operators that can be applied on a single or two operands respectively.

Operator	Description
+ (Addition)	Adds values on either side of the operator.
- (Subtraction)	Subtracts right-hand operand from left-hand operand.
* (Multiplication)	Multiplies values on either side of the operator.
/ (Division)	Divides left-hand operand by right-hand operand.
% (Modulus)	Divides left-hand operand by right-hand operand and returns remainder.
++ (Increment)	Increases the value of operand by 1.
-- (Decrement)	Decreases the value of operand by 1.

Example:

```
class OperatorExample{
public static void main(String args[]){
int a=10;
int b=5;
System.out.println(a+b);//15
System.out.println(a-b);//5
System.out.println(a*b);//50
System.out.println(a/b);//2
System.out.println(a%b);//0
}}
```

2) Unary Operators

→ Java unary operators are the types that need only one operand to perform any operation like increment, decrement, negation etc. It consists of various arithmetic, logical and other operators that operate on a single operand.They are used to increment, decrement or negate a value.

Operator	Description
-	Unary minus, used for negating the values.
+	Unary plus, indicates positive value (numbers are positive without this, however). It performs an automatic conversion to int when the type of its operand is byte, char, or short. This is called unary numeric promotion.
++	Increment operator, used for incrementing the value by 1. There are two varieties of increment operator. Post-Increment : Value is first used for computing the result and then incremented. Pre-Increment : Value is incremented first and then result is computed.
--	Decrement operator, used for decrementing the value by 1. There are two varieties of decrement operator. Post-decrement : Value is first used for computing the result and then decremented. Pre-Decrement : Value is decremented first and then result is computed.
!	Logical not operator, used for inverting a boolean value.
~	**Bitwise Complement Operator** returns the one's complement representation of the input value or operand, i.e, with all bits inverted, means it makes every 0 to 1, and every 1 to 0.

Example:

```
class Unary {
   public static void main(String[] args)
   {
      // variable declaration
      int n1 = 6;

      // Displaying numbers
      System.out.println("First Number = " + n1);

      // Performing bitwise complement
      System.out.println(n1 + "'s bitwise complement = " + ~n1);
   }
}
```

3) Assignment Operator :

→ These operators are used to assign values to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data-type of the operand on the left side otherwise the compiler will raise an error. This means that the assignment operators have right to left associativity. General format of assignment operator is,
variable *operator* value;

→ '=' Assignment operator is used to assign a value to any variable. It has a right to left associativity, i.e value given on right hand side of operator is assigned to the variable on the left and therefore right hand side value must be declared before using it or should be a constant.

→ In many cases assignment operator can be combined with other operators to build a shorter version of statement called Compound Statement. For example, instead of a = a+5, we can write a += 5.

Operator	Description	Example : int a=5
+=	for adding left operand with right operand and then assigning it to variable on the left.	a+=10
-=	for subtracting left operand with right operand and then assigning it to variable on the left.	a-=10
*=	for multiplying left operand with right operand and then assigning it to variable on the left.	a*=10
/=	for dividing left operand with right operand and then assigning it to variable on the left.	a/=10
%=	for assigning modulo of left operand with right operand and then assigning it to variable on the left.	a%=10

4) Relational Operators

→ These operators are used to check for relations like equality, greater than, less than. They return boolean result after the comparison and are extensively used in looping statements as well as conditional if else statements.

→ General format is,
variable relation_operator value

→ Some of the relational operators are-

Operator	Description
==	Equal to : returns true if left hand side is equal to right hand side.
!=	Not Equal to : returns true if left hand side is not equal to right hand side.
<	less than : returns true if left hand side is less than right hand side.
<=	less than or equal to : returns true if left hand side is less than or equal to right hand side.
>	Greater than : returns true if left hand side is greater than right hand side.
>=	Greater than or equal to: returns true if left hand side is greater than or equal to right hand side.

5) Logical Operators

→ These operators are used to perform "logical AND" and "logical OR" operation, i.e. the function similar to AND gate and OR gate in digital electronics. One thing to keep in mind is the second condition is not evaluated if the first one is false, i.e. it has a short-circuiting effect. Used extensively to test for several conditions for making a decision.

→ Conditional operators are-

Operator	Description	Syntax
&&	Logical AND: This operator returns true when both the conditions under consideration are satisfied or are true. If even one of the two yields false, the operator results false. For example, cond1 && cond2 returns true when both cond1 and cond2 are true (i.e. non-zero).	condition1 && condition2
||	Logical OR: This operator returns true when one of the two conditions under consideration are satisfied or are true. If even one of the two yields true, the operator results true. To make the result false, both the constraints need to return false.	condition1 || condition2
!	Logical NOT: Unlike the previous two, this is a unary operator and returns true when the condition under consideration is not satisfied or is a false condition. Basically, if the condition is false, the operation returns true and when the condition is true, the operation returns false.	!(condition)

6) Ternary operator

→ Ternary operator is a shorthand version of if-else statement. It has three operands and hence the name ternary.

→ Java ternary operator is the only conditional operator that takes three operands. It's a one-liner replacement for if-then-else statement and used a lot in Java programming. We can use the ternary operator in place of if-else conditions or even switch conditions using nested ternary operators. Although it follows the same algorithm as of if-else statement, the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.

→ General format is-
condition ? if true : if false

→ The above statement means that if the condition evaluates to true, then execute the statements after the '?' else execute the statements after the ':'.

7) Bitwise Operators

→ These operators are used to perform manipulation of individual bits of a number. They can be used with any of the integer types. They are used when performing update and query operations of Binary indexed tree.

Operator	Description
&	Bitwise AND operator: returns bit by bit AND of input values.
|	Bitwise OR operator: returns bit by bit OR of input values.
^	Bitwise XOR operator: returns bit by bit XOR of input values.
~	Bitwise Complement Operator: This is a unary operator which returns the one's compliment representation of the input value, i.e. with all bits inversed.

8) Shift Operators

Operator	Description
<<	Left shift operator: shifts the bits of the number to the left and fills 0 on voids left as a result. Similar effect as of multiplying the number with some power of two.
>>	Signed Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit depends on the sign of initial number. Similar effect as of dividing the number with some power of two.
>>>	Unsigned Right shift operator: shifts the bits of the number to the right and fills 0 on voids left as a result. The leftmost bit is set to 0.

→ These operators are used to shift the bits of a number left or right thereby multiplying or dividing the number by two respectively. They can be used when we have to multiply or divide a number by two.

→ General format-
number shift_op number_of_places_to_shift;

9) Special Operators[Class and Object Operators]:

Operator	Description
instance of operator	Instance of operator is used for type checking. It can be used to test if an object is an instance of a class, a subclass or an interface.
new	Class instantiation
.	Class member Access
()	Method Invocation
(type)	Object Cast

Control Statements

❖ If-else Statement

The Java *if statement* is used to test the condition. It checks boolean condition: *true* or *false*. There are various types of if statement in Java.

- if statement
- if-else statement
- if-else-if ladder
- nested if statement

1. if Statement

→ The Java if statement tests the condition. It executes the *if block* if condition is true.

→ **Syntax:**

```
if(condition){
//code to be executed
}
```

2. if-else Statement

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

→ **Syntax:**

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

3. if-else-if ladder Statement

→ The if-else-if ladder statement executes one condition from multiple statements.

→ **Syntax:**

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
```

```
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

4. Nested if statement

→ The nested if statement represents the if block within another if block. Here, the inner if block condition executes only when outer if block condition is true.

→ **Syntax:**

```
if(condition){
    //code to be executed
        if(condition){
          //code to be executed
    }
}
```

❖ Switch Statement

→ The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, we can use strings in the switch statement.

→ In other words, the switch statement tests the equality of a variable against multiple values.

→ Points to Remember

- There can be *one or N number of case values* for a switch expression.
- The case value must be of switch expression type only. The case value must be *literal or constant*. It doesn't allow variables.
- The case values must be *unique*. In case of duplicate value, it renders compile-time error.
- The Java switch expression must be of *byte, short, int, long (with its Wrapper type),* enums *and string*.
- Each case statement can have a *break statement* which is optional. When control reaches to the break statement, it jumps the control after the switch expression. If a break statement is not found, it executes the next case.
- The case value can have a *default label* which is optional.

→ **Syntax:**

```
switch(expression){
case value1:
 //code to be executed;
 break;  //optional
case value2:
 //code to be executed;
 break;  //optional
......
default:
 code to be executed if all cases are not matched;
}
```

Looping Statements

1. do-while Loop

→ The Java *do-while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed and we must have to execute the loop at least once, it is recommended to use do-while loop.

→ The Java *do-while loop* is executed at least once because condition is checked after loop body.

→ **Syntax:**

```
do{
//code to be executed
}while(condition);
```

→ **Example:**

```
public class DoWhileExample {
public static void main(String[] args) {
   int i=1;
   do{
      System.out.println(i);
   i++;
   }while(i<=10);
}
}
```

2. while Loop

→ The Java *while loop* is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop.

→ **Syntax:**

```
while(condition){
//code to be executed
}
```

→ **Example:**

```
public class WhileExample {
public static void main(String[] args) {
   int i=1;
   while(i<=10){
      System.out.println(i);
   i++;
   }
}
}
```

3. For Loop

→ The Java for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop.

→ There are three types of for loops in java.

- Simple For Loop
- For-each or Enhanced For Loop
- Labeled For Loop

1) Simple For Loop

→ A simple for loop is the same as C/C++. We can initialize the variable, check condition and increment/decrement value. It consists of four parts:

1. **Initialization**: It is the initial condition which is executed once when the loop starts. Here, we can initialize the variable, or we can use an already initialized variable. It is an optional condition.
2. **Condition**: It is the second condition which is executed each time to test the condition of the loop. It continues execution until the condition is false. It must return boolean value either true or false. It is an optional condition.
3. **Statement**: The statement of the loop is executed each time until the second condition is false.
4. **Increment/Decrement**: It increments or decrements the variable value. It is an optional condition.

→ **Syntax:**

```
for(initialization;condition;incr/decr){
//statement or code to be executed
}
```

→ **Example:**

```
//Java Program to demonstrate the example of for loop
//which prints table of 1
public class ForExample {
public static void main(String[] args) {
   //Code of Java for loop
   for(int i=1;i<=10;i++){
      System.out.println(i);
   }
}
}
```

2) for-each Loop

→ The for-each loop is used to traverse array or collection in java. It is easier to use than simple for loop because we don't need to increment value and use subscript notation.

→ It works on elements basis not index. It returns element one by one in the defined variable.

→ **Syntax:**

```
for(Type var:array){
//code to be executed
}
```

→ **Example:**

```
//Java For-each loop example which prints the elements of the array
public class ForEachExample {
public static void main(String[] args) {
   //Declaring an array
   int arr[]={12,23,44,56,78};
   //Printing array using for-each loop
   for(int i:arr){
      System.out.println(i);
   }
}
}
```

3) Labeled For Loop

→ We can have a name of each Java for loop. To do so, we use label before the for loop. It is useful if we have nested for loop so that we can break/continue specific for loop.

→ Usually, break and continue keywords breaks/continues the innermost for loop only.

→ **Syntax:**

```
labelname:
for(initialization;condition;incr/decr){
//code to be executed
}
```

→ **Example:**

```
//A Java program to demonstrate the use of labeled for loop
public class LabeledForExample {
public static void main(String[] args) {
   //Using Label for outer and for loop
   aa:
      for(int i=1;i<=3;i++){
         bb:
            for(int j=1;j<=3;j++){
```

```
                if(i==2&&j==2){
                   break aa;
                }
                System.out.println(i+" "+j);
            }
        }
    }
    }
```

Jumping Statements

1) break Statement

→ When a break statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.

→ The Java *break* statement is used to break loop or switch statement. It breaks the current flow of the program at specified condition. In case of inner loop, it breaks only inner loop.

→ We can use Java break statement in all types of loops such as for loop, while loop and do-while loop.

2) continue Statement

→ The continue statement is used in loop control structure when we need to jump to the next iteration of the loop immediately. It can be used with for loop or while loop.

→ The Java *continue statement* is used to continue the loop. It continues the current flow of the program and skips the remaining code at the specified condition. In case of an inner loop, it continues the inner loop only.

→ We can use Java continue statement in all types of loops such as for loop, while loop and do-while loop.

3) return Statement

→ return is a reserved keyword in Java i.e, we can't use it as an identifier. It is used to exit from a method, with or without a value.

→ return can be used with methods in two ways:

- **Methods returning a value:** For methods that define a return type, return statement must be immediately followed by return value.
- **Methods not returning a value:** For methods that **don't** return a value, return statement can be skipped.

Array

→ 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.

→ 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.

→ **Advantages**

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

→ **Disadvantages**

- **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.

❖ Declaring Array Variables

→ To use an array in a program, we must declare a variable to reference the array, and we must specify the type of array the variable can reference. Here is the syntax for declaring an array variable –

→ **Syntax:**

dataType[] arrayRefVar;// preferred way.

or

dataType arrayRefVar[];// works but not preferred way.

→ **Example:**

double[] myList;// preferred way.

or

double myList[];// works but not preferred way.

❖ Creating Arrays

→ We can create an array by using the new operator with the following syntax −

→ **Syntax:**

arrayRefVar = new dataType[arraySize];

The above statement does two things –

- **It creates an array using new dataType[arraySize].**
- **It assigns the reference of the newly created array to the variable arrayRefVar.**

→ Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement, as shown below −

dataType[] arrayRefVar = new dataType[arraySize];

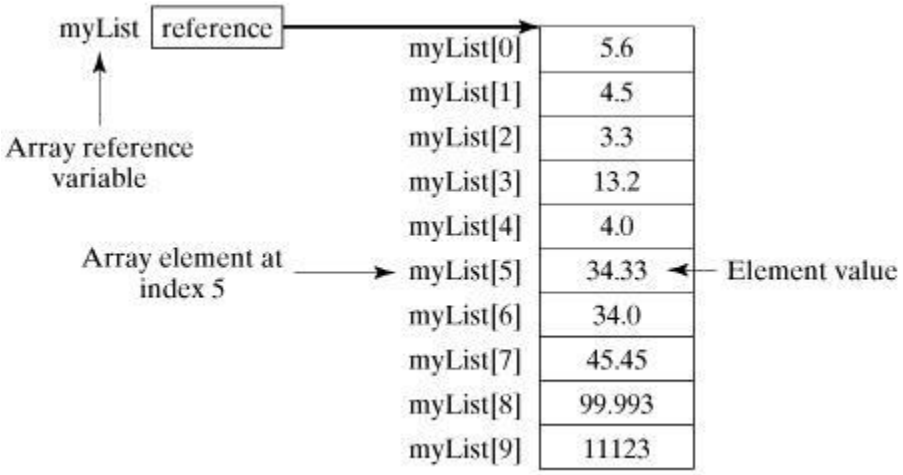
❖ Alternatively we can create arrays as follows –

dataType[] arrayRefVar = {value0, value1, ..., valuek};

→ The array elements are accessed through the index. Array indices are 0-based; that is, they start from 0 to arrayRefVar.length-1.

→ **Example:**

double[] myList = new double[10];



❖ Processing Arrays

→ When processing array elements, we often use either for loop or foreach loop because all of the elements in an array are of the same type and the size of the array is known.

→ **Example:**

- Using for Loop

```
double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (int i = 0; i < myList.length; i++) {
      System.out.println(myList[i] + " ");
    }
```

- Using foreach Loop

```
double[] myList = {1.9, 2.9, 3.4, 3.5};

    // Print all the array elements
    for (double element: myList) {
      System.out.println(element);
    }
```

Types of Array in java

- Single Dimensional Array
- Multidimensional Array
- Jagged Array

1) Single Dimensional Array:

→ Single dimensional arrays are used to store list of values of same datatype. In other words, single dimensional arrays are used to store a row of values. In single dimensional array, data is stored in linear form. Single dimensional arrays are also called as one-dimensional arrays, Linear Arrays or simply 1-D Arrays.

→ **Syntax:**

data_type[] array_name = new data_type[size];

where:

- **data_type: Type of data to be stored in the array. For example: int, char, etc.**
- **array_name: Name of the array**
- **size: Size of the Array.**

→ **Example:**

```
int[] arr = new int[10];
```

2) Multi Dimensional Array:

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

→ **Syntax:**

data_type[1st dimension][2nd dimension][]..[Nth dimension] array_name = new data_type[size1][size2]….[sizeN];

where:

- **data_type: Type of data to be stored in the array. For example: int, char, etc.**
- **dimension: The dimension of the array created. For example: 1D, 2D, etc.**
- **array_name: Name of the array size1, size2, …, sizeN: Sizes of the dimensions respectively.**

→ **Example:**

Two dimensional array:

```
int[][] twoD_arr = new int[10][20];
```

Three dimensional array:

```
int[][][] threeD_arr = new int[10][20][30];
```

→ Size of multidimensional arrays: The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

3) Jagged Array in Java

→ A jagged array is an array of arrays such that member arrays can be of different sizes, i.e., we can create a 2-D array but with a variable number of columns in each row. These types of arrays are also known as Jagged arrays.

→ **Syntax:**

```
data_type array_name[][] = new data_type[n][];  //n: no. of rows
array_name[] = new data_type[n1] //n1= no. of colmuns in row-1
array_name[] = new data_type[n2] //n2= no. of colmuns in row-2
array_name[] = new data_type[n3] //n3= no. of colmuns in row-3
.
.
.
array_name[] = new data_type[nk]  //nk=no. of colmuns in row-n
```

Alternative, ways to Initialize a Jagged array :

```
int arr_name[][] = new int[][]  {
                        new int[] {10, 20, 30 ,40},
                        new int[] {50, 60, 70, 80, 90, 100},
                        new int[] {110, 120}
                           };

                    OR

            int[][] arr_name = {
                new int[] {10, 20, 30 ,40},
                new int[] {50, 60, 70, 80, 90, 100},
                new int[] {110, 120}
                   };

                    OR

            int[][] arr_name = {
                 {10, 20, 30 ,40},
                 {50, 60, 70, 80, 90, 100},
                 {110, 120}
                   };
```

→ **Example:**

```
class TestJaggedArray{
   public static void main(String[] args){
      //declaring a 2D array with odd columns
      int arr[][] = new int[3][];
      arr[0] = new int[3];
      arr[1] = new int[4];
      arr[2] = new int[2];
      //initializing a jagged array
      int count = 0;
      for (int i=0; i<arr.length; i++)
         for(int j=0; j<arr[i].length; j++)
```

```
            arr[i][j] = count++;

        //printing the data of a jagged array
        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
        }
    }
}
```

Command Line Argument Array

→ The java command-line argument is an argument i.e. passed at the time of running the java program.

→ The arguments passed from the console can be received in the java program and it can be used as an input.

→ So, it provides a convenient way to check the behavior of the program for the different values. We can pass N (1,2,3 and so on) numbers of arguments from the command prompt.

→ **Example 1:**

```
class CommandLineExample{
public static void main(String args[]){
System.out.println("first argument is: "+args[0]);
}
}
```

compile: javac CommandLineExample.java

run: java CommandLineExample BCA4

→ **Example 2:**

```
class CmdEx{
public static void main(String args[]){

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

}
}
```

compile : javac CmdEx.java

run : java CmdEx Sarvodaya 1 3 BCA6 45.05

final Keyword

→ The final keyword in java is used to restrict the user. The java final keyword can be used in many context. Final can be:

- variable
- method
- class

→ The final keyword can be applied with the variables, a final variable that have no value it is called blank final variable or uninitialized final variable. It can be initialized in the constructor only. The blank final variable can be static also which will be initialized in the static block only. We will have detailed learning of these. Let's first learn the basics of final keyword.

1) final variable

→ If we make any variable as final, we cannot change the value of final variable(It will be constant).

→ **Example:**

```
class FinalVar{
    public static void main(String[] args){
            final int no=100;//final variable
            System.out.println("Number is : "+no);
            no=150;//generate error
            System.out.println("Number is : "+no);
    }
}
```

2) final method

→ If we make any method as final, we cannot override it.

→ **Example:**

```
class Test{
  final void disp(){System.out.println("Display from Test");}
}

class FinalMethodEx extends Test{
   //generate an error
   void disp(){System.out.println("Display from Main Class");}

   public static void main(String args[]){
   FinalMethodEx f= new FinalMethodEx();
   f.disp();
   }
}
```

3) final class

→ If we make any class as final, we cannot extend it.

→ **Example:**

```
final class Test{
  final void disp(){System.out.println("Display from Test");}
}
//final class can not inherit
class FinalClassEx extends Test{
   public static void main(String args[]){
   FinalClassEx f= new FinalClassEx();
   f.disp();
   }
}
```

finally block

→ Java finally block is a block that is used *to* ***execute important code*** such as closing connection, stream etc.

→ Java finally block is always executed whether exception is handled or not.

→ Java finally block follows try or catch block.

→ **Example:**

```
class TestFinallyBlock{
  public static void main(String args[]){
  try{
   int data=25/5;
   System.out.println(data);
  }
```

```
 catch(NullPointerException e){System.out.println(e);}
 finally{System.out.println("finally block is always executed");}
 System.out.println("rest of the code...");
 }
}
```

finalize() method

→ Finalize() is the method of **Object class**. This method is called just before an object is garbage collected. finalize() method overrides to dispose system resources, perform clean-up activities and minimize memory leaks.

→ **Syntax:**

```
protected void finalize() throws Throwable
```

→ **Example**

```
public class finalizeExample {
    public static void main(String[] args)
    {
       finalizeExample  obj = new finalizeExample();
       System.out.println(obj.hashCode());
       obj = null;
       // calling garbage collector
       System.gc();
       System.out.println("end of garbage collection");

    }
    protected void finalize()
    {
       System.out.println("finalize method called");
    }
}
```

Java Keywords(assert,strictfp,enum)

1) Assertion:

→ Assertion is a statement in java. It can be used to test our assumptions about the program.

→ While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.

→ Advantage of Assertion is it provides an effective way to detect and correct programming errors.

→ Syntax

- There are two ways to use assertion.
  - First way is:
    assert expression;
  - and second way is:
    assert expression1 : expression2;

→ **Example**

```
import java.util.Scanner;

class AssertionExample{
 public static void main( String args[] ){

  Scanner scanner = new Scanner( System.in );
  System.out.print("Enter ur age ");

  int value = scanner.nextInt();
  assert value>=18:" Not valid";
```

```
 System.out.println("value is "+value);
 }
}
```

→ If we use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, **-ea** or **-enableassertions** switch of java must be used.

→ Compile it by: **javac AssertionExample.java**

→ Run it by: **java -ea AssertionExample**

→ Where not to use Assertion:

→ There are some situations where assertion should be avoid to use. They are:

- According to Sun Specification, assertion should not be used to check arguments in the public methods because it should result in appropriate runtime exception e.g. IllegalArgumentException, NullPointerException etc.
- Do not use assertion, if we don't want any error in any situation.

2) strictfp

→ Java strictfp keyword ensures that we will get the same result on every platform if we perform operations in the floating-point variable. The precision may differ from platform to platform that is why java programming language have provided the strictfp keyword, so that we get same result on every platform. So, now we have better control over the floating-point arithmetic.

→ Legal code for strictfp keyword:

- The strictfp keyword can be applied on methods, classes and interfaces.
  - **strictfp applied on class**
    ```
    strictfp class A{}
    ```
  - **strictfp applied on interface**
    ```
    strictfp interface M{}
    ```
  - **stictfp applied on method**
    ```
    class A{
        strictfp void m(){}//strictfp applied on method
    }
    ```

→ Illegal code for strictfp keyword:

- The strictfp keyword **cannot** be applied on abstract methods, variables or constructors.

```
class B{
    strictfp abstract void m();//Illegal combination of modifiers
}

class B{
    strictfp int data=10;//modifier strictfp not allowed here
}

class B{
    strictfp B(){}//modifier strictfp not allowed here
}
```

3) enum

→ The **Enum in Java** is a data type which contains a fixed set of constants.

→ It can be used for days of the week (SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, and SATURDAY) , directions (NORTH, SOUTH, EAST, and WEST), season (SPRING, SUMMER, WINTER, and AUTUMN or FALL), colors (RED, YELLOW, BLUE, GREEN, WHITE, and BLACK) etc. According to the Java naming conventions, we should have all constants in capital letters. So, we have enum constants in capital letters.

→ Java Enums can be thought of as classes which have a fixed set of constants (a variable that does not change). The Java enum constants are static and final implicitly. It is available since JDK 1.5.

→ Enums are used to create our own data type like classes. The **enum** data type (also known as Enumerated Data Type) is used to define an enum in Java. Unlike C/C++, enum in Java is more *powerful*. Here, we can define an enum either inside the class or outside the class.

→ Java Enum internally inherits the *Enum class*, so it cannot inherit any other class, but it can implement many interfaces. We can have fields, constructors, methods, and main methods in Java enum.

→ Points to remember for Java Enum

- Enum improves type safety
- Enum can be easily used in switch
- Enum can be traversed
- Enum can have fields, constructors and methods
- Enum may implement many interfaces but cannot extend any class because it internally extends Enum class

→ Example:

```
class EnumEx
{
        //defining the enum inside the class
        public enum Season { WINTER, SPRING, SUMMER, FALL }
        //main method
        public static void main(String[] args)
        {
                //traversing the enum
                for (Season s : Season.values())
                        System.out.println(s);
        }
}
```

→ Methods:

- **values()**

  The Java compiler internally adds the values() method when it creates an enum. The values() method returns an array containing all the values of the enum.

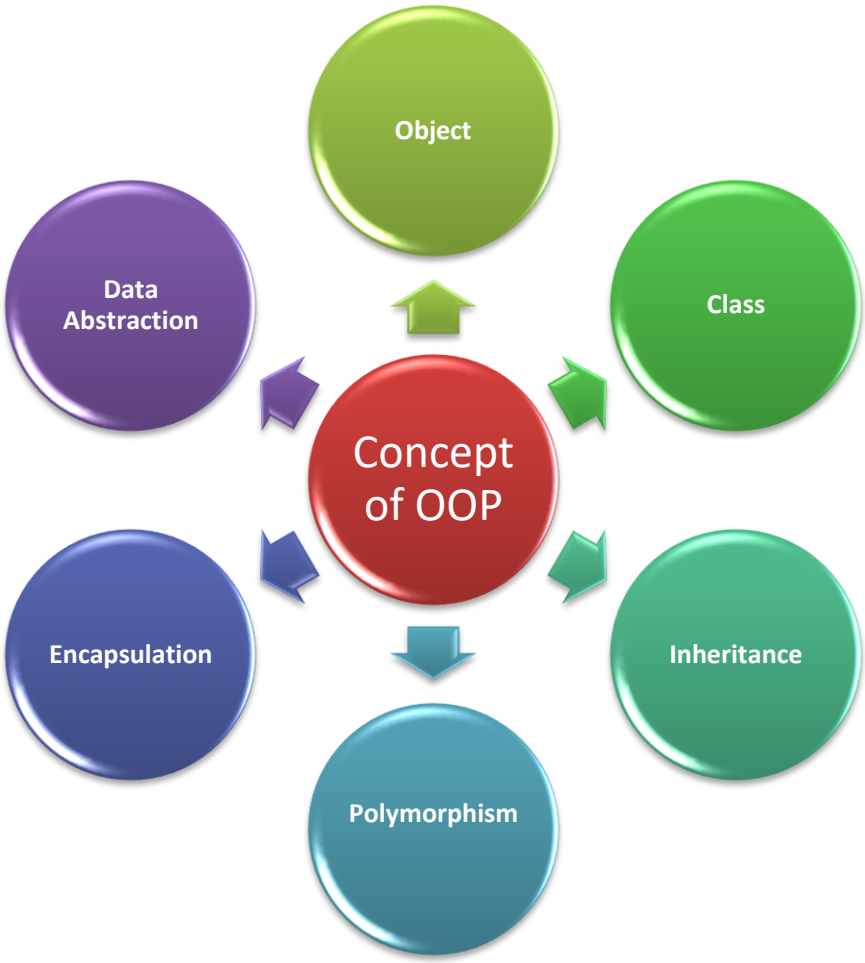
- **valueOf()**

  The Java compiler internally adds the valueOf() method when it creates an enum. The valueOf() method returns the value of given constant enum.

- **ordinal()**

  The Java compiler internally adds the ordinal() method when it creates an enum. The ordinal() method returns the index of the enum value.

OOPs Concept



1) Object

→ Any entity that has state and behavior is known as an object. For example, a chair, pen, table, keyboard, bike, etc. It can be physical or logical.

→ An Object can be defined as an instance of a class. An object contains an address and takes up some space in memory. Objects can communicate without knowing the details of each other's data or code. The only necessary thing is the type of message accepted and the type of response returned by the objects.

→ An object consists of :

- **State:** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

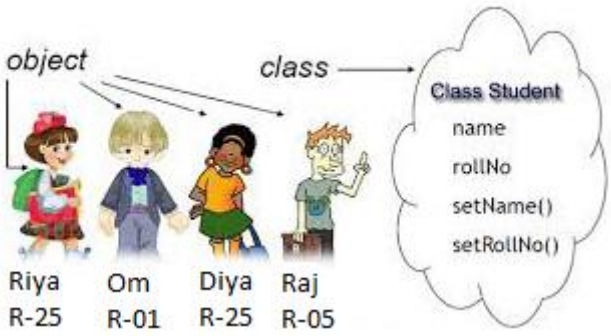
→ **Example:** A dog is an object because it has states like color, name, breed, etc. as well as behaviors like wagging the tail, barking, eating, etc.

2) Class

→ A class is a user defined blueprint or prototype from which objects are created. It represents the set of properties or methods that are common to all objects of one type. In general, class declarations can include these components, in order:

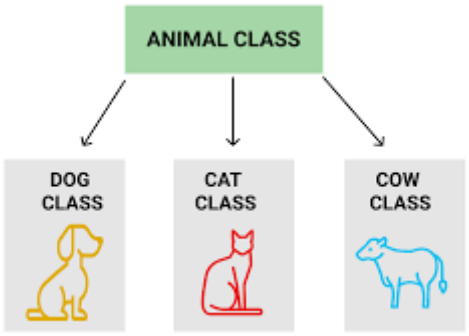
- **Modifiers**: A class can be public or has default access.
- **class keyword:** class keyword is used to create a class.
- **Class name:** The name should begin with an initial letter (capitalized by convention).

- **Superclass(if any):** The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- **Interfaces(if any):** A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- **Body:** The class body surrounded by braces, { }.



3) Inheritance

→ Inheritance is the process by which one object acquire the properties and behaviors of another object. With the use of inheritance the information is made manageable in a hierarchical order. It provides code reusability. It is used to achieve runtime polymorphism.



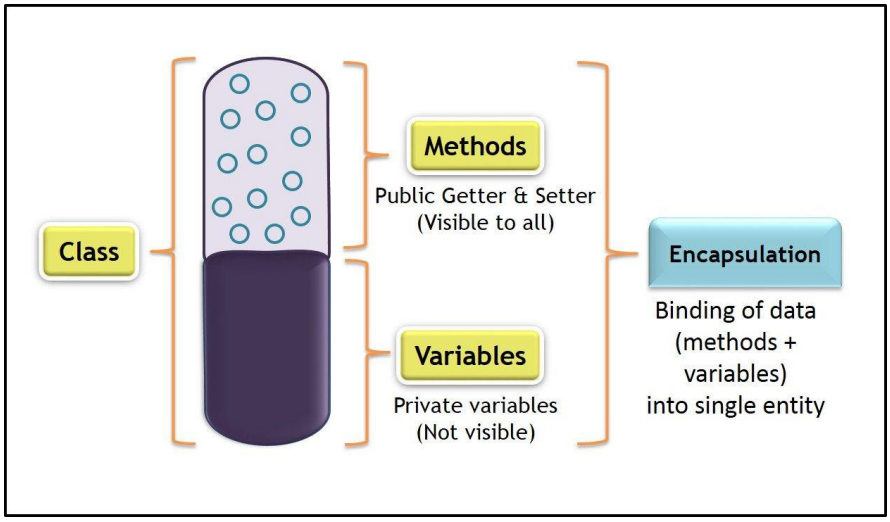
4) Polymorphism

→ Polymorphism (from Greek, meaning "many forms") is a feature that allows one interface to be used for a general class of actions. For example: Can you speak something? a cat speaks meow, dog barks woof, etc.

→ In Java, we use method overloading and method overriding to achieve polymorphism.

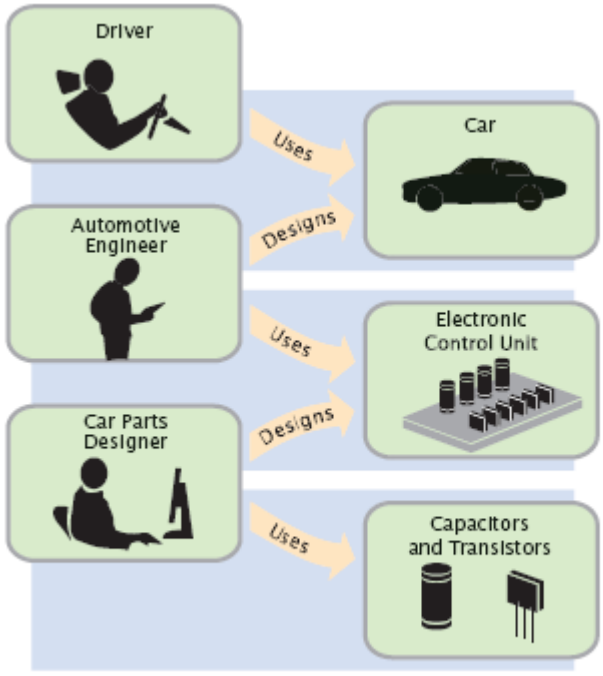
5) Encapsulation

→ Encapsulation is the mechanism that binds together code and the data it manipulates, and keeps both safe from outside interference and misuse
→ For example, a capsule, it is wrapped with different medicines.
→ A java class is the example of encapsulation. Java bean is the fully encapsulated class because all the data members are private here.



6) Abstraction

→ Abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.
→ For example phone call, we don't know the internal processing.
→ In Java, we use abstract class and interface to achieve abstraction.



Levels of Abstraction in Automotive Design

Object and Class

→ An object in Java is the physical as well as a logical entity, whereas, a class in Java is a logical entity only.
→ **What is a class in Java**

- A class is a group of objects which have common properties. It is a template or blueprint from which objects are created. It is a logical entity. It can't be physical.
- A class in Java can contain:
  - **Fields**
  - **Methods**
  - **Constructors**
  - **Blocks**
  - **Nested class and interface**
- Syntax:

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

- Example:

```
class Student
{
    int rno;
    String name;
    void setStudent(int r,String nm)
    {
            rno=r;
            name=nm;
    }
    void disp()
    {
            System.out.println("Rollno :"+rno+",Name :"+name);
    }
}
```

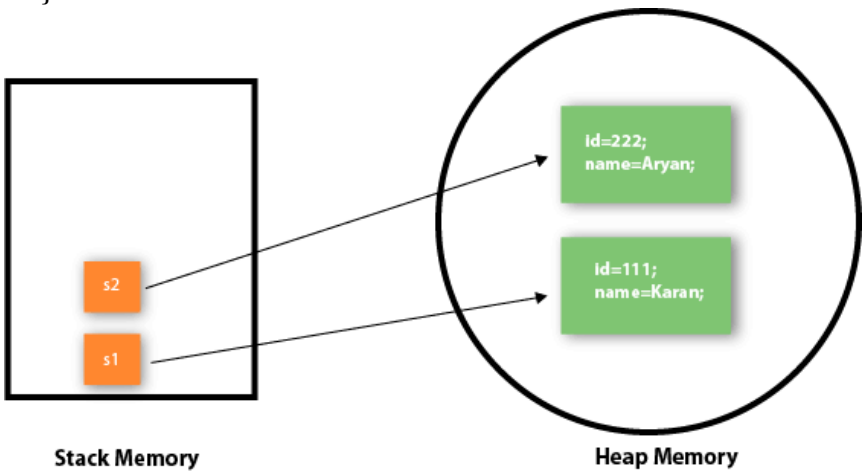
→ **What is an object**

- An entity that has state and behavior is known as an object e.g., chair, bike, marker, pen, table, car, etc. It can be physical or logical (tangible and intangible). The example of an intangible object is the banking system.
- An object has three characteristics:
  - State: represents the data (value) of an object.
  - Behavior: represents the behavior (functionality) of an object such as deposit, withdraw, etc.
  - Identity: An object identity is typically implemented via a unique ID. The value of the ID is not visible to the external user. However, it is used internally by the JVM to identify each object uniquely.
- For Example, Pen is an object. Its name is Reynolds; color is white, known as its state. It is used to write, so writing is its behavior.
- **An object is an instance of a class.** A class is a template or blueprint from which objects are created. So, an object is the instance(result) of a class.
- **Object Definitions:**
  - An object is a real-world entity.
  - An object is a runtime entity.
  - The object is an entity which has state and behavior.
  - The object is an instance of a class.
- **Declaring Objects (Also called instantiating a class)**
  - When an object of a class is created, the class is said to be **instantiated**. All the instances share the attributes and the behavior of the class. But the values of those attributes, i.e. the state are unique for each object. A single class may have any number of instances.
  - As we declare variables like (type name;). This notifies the compiler that we will use name to refer to data whose type is type. With a primitive variable, this declaration also reserves the proper amount of memory for the variable. So for reference variable, type must be strictly a concrete class name. In general, we **can't** create objects of an abstract class or an interface.
  - Syntax : classname objectname;

- Example : Student s1;

- **Initializing an object**
  - The new operator instantiates a class by allocating memory for a new object and returning a reference to that memory. The new operator also invokes the class constructor.
  - Example : s1=new Student();
- **Declaration and Initialization:**
  - Student s1=new Student();
- **Creating multiple objects by one type only**
  - Student s1=new Student(),s2=new Student();
- **There are 3 ways to initialize object in Java.**
  - By reference variable
    - Initializing an object means storing data into the object.
    - Example:
      Student s1=new Student();
      s1.rno=101;
      s1.name="Disha";
  - By method
    - we are creating the two objects of Student class and initializing the value to these objects by invoking the setStudent() method.
    - Example:

```
class Student{
        int rollno;
        String name;
        void setStudent(int r, String n){
                rollno=r;
                name=n;
        }
        public static void main(String args[]){
                Student s1=new Student();
                Student s2=new Student();
                s1.setStudent(111,"Karan");
                s2.setStudent(222,"Aryan");
                System.out.println(s1.rollno+" "+s1.name);
                System.out.println(s2.rollno+" "+s2.name);
        }
}
```



  - By constructor
    - we are creating the two objects of Student class and initializing the value to these objects by invoking the setStudent() method.
    - Example:

```
class Student{
```

```
        int rollno;
        String name;
        Student(int r, String n){
                rollno=r;
                name=n;
        }
        public static void main(String args[]){
                Student s1=new Student(111,"Karan");
                System.out.println(s1.rollno+" "+s1.name);
        }
}
```

→ **Instance variable[Field Variable, Member Variable]:**

- A variable which is created inside the class but outside the method is known as an instance variable. Instance variable doesn't get memory at compile time.
- It gets memory at runtime when an object or instance is created. That is why it is known as an instance variable.

→ **Method:**

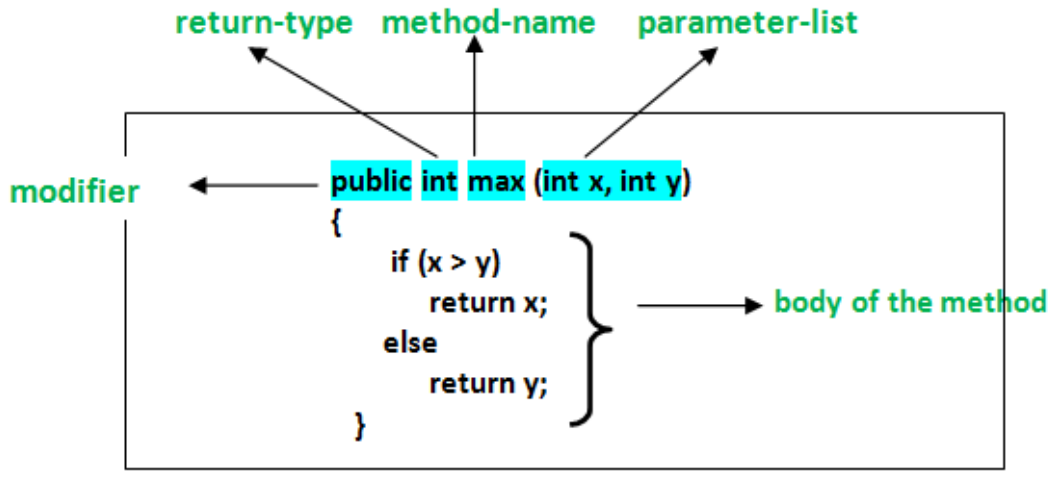
- A **method** is a block of code or collection of statements or a set of code grouped together to perform a certain task or operation. It is used to achieve the **reusability** of code.
- In general, a **method** is a way to perform some task. Similarly, the **method in Java** is a collection of instructions that performs a specific task. It provides the reusability of code. We can also easily modify code using **methods**.
- Advantages:
  - Code Reusability
  - Code Optimization.
- **Types of Method**

  There are two types of methods in Java:

  I. Predefined Method
  - In Java, predefined methods are the method that is already defined in the Java class libraries is known as predefined methods. It is also known as the 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 length(), equals(), compareTo(), sqrt(), etc.

  II. 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.
  - Method Declaration:

    The method declaration provides information about method attributes, such as visibility, return-type, name, and arguments. It has six components that are known as **method header.**



- **Method Signature:** Every method has a method signature. It is a part of the method declaration. It includes the method name and parameter list.
  - **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:
  - **Return Type:** Return type is a data type that the method returns. It may have a primitive data type, object, collection, void, etc. If the method does not return anything, we use void keyword.
  - **Method Name:** It is a unique name that is used to define the name of a method. It must be corresponding to the functionality of the method. Suppose, if we are creating a method for subtraction of two numbers, the method name must be subtraction(). A method is invoked by its name.
  - **Parameter List:** It is the list of parameters separated by a comma and enclosed in the pair of parentheses. It contains the data type and variable name. If the method has no parameter, left the parentheses blank.
- **Method Body:** It is a part of the method declaration. It contains all the actions to be performed. It is enclosed within the pair of curly braces.

- Example:

```
public static void findEvenOdd(int num)  {
        if(num%2==0)
                System.out.println(num+" is even");
        else
                System.out.println(num+" is odd");
}
```

→ **Anonymous object:**

- Anonymous simply means nameless. An object which has no reference is known as an anonymous object. It can be used at the time of object creation only.
- If we have to use an object only once, an anonymous object is a good approach.
  For example:
  new Student();//anonymous object
- Calling method through a reference:
  Student s=new Student();
  c.setStudent(5,"Raj");
- Calling method through an anonymous object
  new Student().setStudent(5,"Raj");

Constructor

→ In Java, a constructor is a block of codes similar to the method. It is called when an instance of the class is created. At the time of calling constructor, memory for the object is allocated in the memory.

→ It is a special type of method which is used to initialize the object.

→ Every time an object is created using the new() keyword, at least one constructor is called.

→ It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

→ We can use access modifiers while declaring a constructor. It controls the object creation. In other words, we can have private, protected, public or default constructor in Java.

→ Note: It is called constructor because it constructs the values at the time of object creation. It is not necessary to write a constructor for a class. It is because java compiler creates a default constructor if our class doesn't have any.

→ Rules for creating Java constructor

- Constructor name must be the same as its class name
- A Constructor must have no explicit return type
- A Java constructor cannot be abstract, static, final, and synchronized

❖ Types of Java constructors

→ There are two types of constructors in Java:
- Default constructor (no-arg constructor)
- Parameterized constructor

1) Default Constructor:
- A constructor that has no parameter is known as default constructor. If we don't define a constructor in a class, then compiler creates **default constructor(with no arguments)** for the class. And if we write a constructor with arguments or no-arguments then the compiler does not create a default constructor. Default constructor provides the default values to the object like 0, null, etc. depending on the type.
- Syntax
  <class_name>(){}
- Example

```
class Student
{
        Student(){System.out.println("Student Class is Created…");}
        public static void main(String args[]){
                Student s=new Student();
        }
}
```

2) Parameterized Constructor:
- A constructor that has parameters is known as parameterized constructor. If we want to initialize fields of the class with our own values, then use a parameterized constructor.
- Syntax
  <class_name>(Parameterlist){}
- Example:

```
class Student
{
        int rollno;
        String name;
        Student(int r, String n)
        {
                rollno=r;
                name=n;
        }
        public static void main(String args[]){
                Student s1=new Student(111,"Karan");
                System.out.println(s1.rollno+" "+s1.name);
        }
}
```

❖ Constructor Overloading:

→ In Java, we can overload constructors like methods. The constructor overloading can be defined as the concept of having more than one constructor with different parameters so that every constructor can perform a different task.

→ Consider the following Java program, in which we have used different constructors in the class.

→ Example:

```
public class Student {
   //instance variables of the class
   int id,passoutYear;
   String name,contactNo,collegeName;

   Student(int id, String name ,String contactNo, String collegeName, int passoutYear){
```

```
        this.id = id;
        this.name = name;
        this.contactNo = contactNo;
        this.collegeName = collegeName;
        this.passoutYear = passoutYear;
    }
    Student(int id, String name){
        this.id = id;
        this.name = name;
    }
    public static void main(String[] args) {
        //object creation
        Student s = new Student(101, “Rohan”);
        Student s1 = new Student(102, “Rutvi”,”9899234455”, “IIT Kanpur”, 2021);
        System.out.println(“Printing Student Information: \n”);
        System.out.println(“Id: “+s.id+”\nName: “+s.name);
        System.out.println(“Name: “+s1.name+”\nId: “+s1.id+”\nContact No.: “+s1.contactNo+”\nCollege Name: “+s1.contactNo+”\nPassing Year: “+s1.passoutYear);
        }
    }
```

Static Keyword

→ The **static keyword** in Java is used for memory management mainly. We can apply static keyword with variables, methods, blocks and nested classes. The static keyword belongs to the class than an instance of the class.

→ The static can be:

1) Variable (also known as a class variable)
2) Method (also known as a class method)
3) Block
4) Nested class

1) Java static variable

→ When a variable is declared as static, then a single copy of variable is created and shared among all objects at class level. Static variables are, essentially, global variables. All instances of the class share the same static variable.

→ The static variable can be used to refer to the common property of all objects (which is not unique for each object), for example, the company name of employees, college name of students, etc.

→ The static variable gets memory only once in the class area at the time of class loading.

→ Advantages of static variable : It makes our program **memory efficient** (i.e., it saves memory).

→ problem without static variable :

```
class Student{
    int rollno;
    String name;
    String college=”ITS”;
}
```

→ Suppose there are 500 students in my college, now all instance data members will get memory each time when the object is created. All students have its unique rollno and name, so instance data member is good in such case. Here, "college" refers to the common property of all objects. If we make it static, this field will get the memory only once.

→ Example:

```
class Student{
        int rollno;//instance variable
        String name;
        static String college =”ITS”;//static variable
        //constructor
        Student(int r, String n){
```

```
        rollno = r;
        name = n;
        }
        void display (){System.out.println(rollno+" "+name+" "+college);}
}
public class TestStaticVariable{
        public static void main(String args[]){
                Student s1 = new Student(111,”Karan”);
                Student s2 = new Student(222,”Aryan”);
                //we can change the college of all objects by the single line of code
                //Student.college=”BBDIT”
                ;
                s1.display();
                s2.display();
        }
}
```

2) Java static method

→ When a method is declared with *static* keyword, it is known as static method. The most common example of a static method is *main( )* method. As discussed above, Any static member can be accessed before any objects of its class are created, and without reference to any object. Methods declared as static have several restrictions:

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to this or super in any way.

→ Example:

```
class Student{
        int rollno;
        String name;
        static String college = “ITS”;
        //static method to change the value of static variable
        static void change(){
        college = “BBDIT”;
}
Student(int r, String n){
        rollno = r;
        name = n;
}
void display(){System.out.println(rollno+” “+name+” “+college);}
}
public class TestStaticMethod{
        public static void main(String args[]){
                Student.change();//calling change method
                Student s1 = new Student(111,”Karan”);
                Student s2 = new Student(222,”Aryan”);
                Student s3 = new Student(333,”Sonu”);
                s1.display();
                s2.display();
                s3.display();
        }
}
```

3) Java static block

→ If we need to do computation in order to initialize our **static variables**, we can declare a static block that gets executed exactly once, when the class is first loaded. Consider the following java program demonstrating use of static blocks.

```
class A2{
    static{System.out.println("static block is invoked");}
    public static void main(String args[]){
        System.out.println("Hello main");
    }
}
```

4) Java static nested class

→ We cannot declare top-level class with a static modifier, but can declare nested classes as static. Such type of classes are called Nested static classes.

→ A static class i.e. created inside a class is called static nested class in java. It cannot access non-static data members and methods. It can be accessed by outer class name.

→ It can access static data members of outer class including private.

→ Static nested class cannot access non-static (instance) data member or method.

→ Example:

```
class Outer{
    static int data=30;
    static class Inner
    {
        void msg(){System.out.println("data is "+data);}
    }
    public static void main(String args[]){
        Outer.Inner obj=new Outer.Inner();
        obj.msg();
    }
}
```

Overloading

→ Overloading allows different methods to have the same name, but different signatures where the signature can differ by the number of input parameters or type of input parameters or both. Overloading is related to compile-time (or static) polymorphism.

→ Advantage of method overloading: Method overloading *increases the readability of the program*.

→ There are two ways to overload the method in java

- By changing number of arguments
- By changing the data type

1) Method Overloading: changing no. of arguments

→ In this example, we have created two methods, first add() method performs addition of two numbers and second add method performs addition of three numbers.

→ In this example, we are creating static methods so that we don't need to create instance for calling methods.

```
class Adder{
    static int add(int a,int b){return a+b;}
    static int add(int a,int b,int c){return a+b+c;}
}
class SimpleAdder{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(11,11,11));
    }}
```

2) Method Overloading: changing data type of arguments

→ In this example, we have created two methods that differs in data type. The first add method receives two integer arguments and second add method receives two double arguments.

→ **Example**

```
class Adder{
    static int add(int a, int b){return a+b;}
    static double add(double a, double b){return a+b;}
}
class SimpleAdder{
    public static void main(String[] args){
        System.out.println(Adder.add(11,11));
        System.out.println(Adder.add(12.3,12.6));
}}
```

Varargs

→ The varrags allows the method to accept zero or multiple arguments. Before varargs either we use overloaded method or take an array as the method parameter but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

→ Advantage of Varargs: We don't have to provide overloaded methods so less code.

→ Syntax of varargs:

```
return_type method_name(data_type... variableName){}
```

→ Example:

```
class VarArgsEx
{
   static void fun(int ...a)
   {
      System.out.println("Number of arguments: " + a.length);
      for (int i: a)
         System.out.print(i + " ");
      System.out.println();
   }
   public static void main(String args[])
   {
      fun(100);        // one parameter
      fun(1, 2, 3, 4);  // four parameters
      fun();           // no parameter
   }
}
```

IIB in Java

→ Instance Initializer block is used to initialize the instance data member. It run each time when object of the class is created. So firstly, constructor is invoked and the java compiler copies the instance initializer block in the constructor after the first statement super(). They run each time when object of the class is created.

→ The initialization of the instance variable can be done directly but there can be performed extra operations while initializing the instance variable in the instance initializer block.

→ Initialization blocks are executed whenever the class is initialized and before constructors are invoked.

→ They are typically placed above the constructors within braces.

→ It is not at all necessary to include them in our classes.

→ Example:

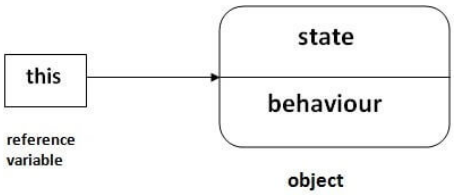
```
class Bike{
```

```
    int speed;
    Bike(){System.out.println("speed is "+speed);}
    {
        speed=100;
        System.out.println("IIB – Instance Initializer Block");
    }

   public static void main(String args[]){
   Bike b1=new Bike();
   Bike b2=new Bike();
   }
}
```

this keyword

→ There can be a lot of usage of java this keyword. In java, this is a reference variable that refers to the current object.



→ Usage of java this keyword

1) this can be used to refer current class instance variable.
2) this can be used to invoke current class method (implicitly)
3) this() can be used to invoke current class constructor.
4) this can be passed as an argument in the method call.
5) this can be passed as argument in the constructor call.
6) this can be used to return the current class instance from the method.

1) this: to refer current class instance variable

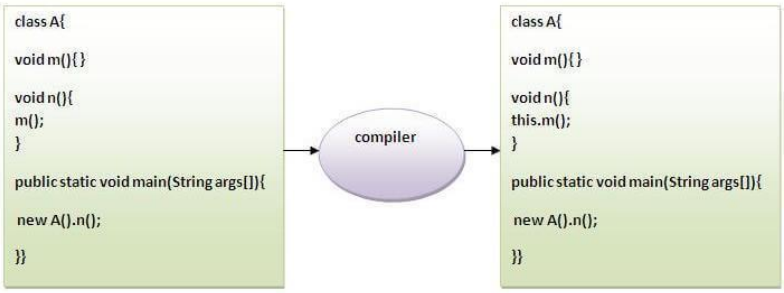
→ The this keyword can be used to refer current class instance variable. If there is ambiguity between the instance variables and parameters, this keyword resolves the problem of ambiguity.

```
class Student{
    int rollno;
    String name;
    float fee;
    Student(int rollno,String name,float fee){
        this.rollno=rollno;
        this.name=name;
        this.fee=fee;
    }
    void display(){System.out.println(rollno+" "+name+" "+fee);}
}

class ThisEx{
public static void main(String args[]){
    Student s1=new Student(111,"ankit",5000f);
    Student s2=new Student(112,"sumit",6000f);
    s1.display();
    s2.display();
}}
```

2) this: to invoke current class method

→ We may invoke the method of the current class by using the this keyword. If we don't use the this keyword, compiler automatically adds this keyword while invoking the method.



→ Example:

```
class A{
        void m(){System.out.println("hello m");}
        void n(){
                System.out.println("hello n");
                //m();//same as this.m()
                this.m();
        }
}
class ThisEx2{
        public static void main(String args[]){
        A a=new A();
        a.n();
        }
}
```

3) this() : to invoke current class constructor

→ The this() constructor call can be used to invoke the current class constructor. It is used to reuse the constructor. In other words, it is used for constructor chaining.

→ **Calling default constructor from parameterized constructor:**

```
class A{
    A(){System.out.println("hello a");}
    A(int x){
            this();
            System.out.println(x);
    }
}
class ThisEx3{
    public static void main(String args[]){
    A a=new A(10);
    }
}
```

4) this: to pass as an argument in the method

→ The this keyword can also be passed as an argument in the method. It is mainly used in the event handling.

```
class ThisEx4{
  void m(ThisEx4 obj){
        System.out.println("method is invoked");
  }
  void p(){
        m(this);
  }
```

```
 public static void main(String args[]){
 ThisEx4 s1 = new ThisEx4 ();
 s1.p();
 }
}
```

5) this: to pass as argument in the constructor call

→ We can pass the this keyword in the constructor also. It is useful if we have to use one object in multiple classes.

→ Example:

```
class B{
 A obj;
 B(A obj){
    this.obj=obj;
 }
 void display(){
    System.out.println(obj.data);//using data member of A4 class
 }
}

class A{
 int data=10;
 A(){
  B b=new B(this);
  b.display();
 }
 public static void main(String args[]){
  A a1=new A();
 }
}
```

6) this keyword can be used to return current class instance

→ We can return this keyword as an statement from the method. In such case, return type of the method must be the class type (non-primitive). Let's see the example:

→ Syntax of this that can be returned as a statement

```
return_type method_name(){
        return this;
}
```

→ Example:

```
class A{
        A getA(){
        return this;
        }
        void msg(){System.out.println("Hello java");}
}
class ThisEx6{
        public static void main(String args[]){
        new A().getA().msg();
        }
}
```