Java Notes

What is Java?

- Java is one of the most popular programming languages in the world for building web applications, mobile apps, enterprise software, and back-end APIs. It has been a popular choice among developers from the day it was first release with millions of Java applications in use today.
- Approximately 90% of Fortune 500 companies rely on Java for their technology needs. Major companies like Twitter, Netflix, Alibaba, Uber, LinkedIn, YouTube, Spotify, and PayPal utilize Java to power their services. In fact, Java supports over 52% of the world's back-end web services.
- According to Oracle, Java runs on around 4 billion mobile phones and over 130 million TV devices. Given its presence across a wide range of devices—from mobile phones to spacecraft—demand for skilled Java developers remains high.

History of Java

- **Java** was created by **James Gosling** in **1995** at a company called **Sun Microsystems**.
- The development of Java began as part of a project called the **Green Project**, led by James Gosling and a team of engineers at Sun Microsystems. Initially, the goal was to create a programming language for digital devices like set-top boxes. The language was originally named **Oak**, inspired by an oak tree outside Gosling's office. However, due to trademark conflicts with other programming languages, the name was later changed to **Java**. The new name was inspired by **Java coffee**, a favorite among the team, known for its energizing quality, reflecting the language's dynamism and simplicity.
- On **May 23, 1995**, Sun Microsystems announced Java publicly at the **SunWorld** conference, which marked the language's initial release.
- In **2004**, Java was the leading programming language globally. That same year, Java made its way to space! The **Spirit Rover**, which was part of NASA's Mars Exploration Rover mission, used Java to help explore the surface of Mars. The rover's onboard systems ran Java code to control its movements and perform various tasks during its mission.
- In **January 2010**, Oracle Corporation purchased Sun Microsystem, bringing Java technology into the Oracle family of products.

- In **March 2014**, Java 8 version with major new features like Lambda expression, Streams, etc., was released.
- In **August 2017**, Oracle announced its plan to release new versions of Java every six months.

Java Version History(JDK):

Java Version	Release Year	Key Features
Java 1.0	1996	Applets, Write Once Run Anywhere (WORA)
Java 1.1	1997	Inner classes, JDBC for Database connectivity
Java 2 (1.2)	1998	Swing, Collections Framework
Java 1.3	2000	Enhanced JVM and Hotspot compiler.
Java 1.4	2002	Assertions, Java Web Start
Java 5.0	2004	Generics, Enhanced for-loop, Annotations
Java 6	2006	Scripting support and web services integration.
Java 7	2011	String in switch, try-with-resources, Fork/Joinframework.
Java 8 **(LTS)**	2014	Lambda Expressions, Stream API, Date-Time API
Java 9	2017	Modular system (Project Jigsaw), JShell, REPL tool.
Java 10	2018	Local Variable Type Inference (var)
Java 11 **(LTS)**	2018	HTTP Client API, removal of JavaFX.
Java 12	2019	Switch Expressions (preview), GC enhancements
Java 13	2019	Text Blocks (preview)
Java 14	2020	Record classes (preview), Pattern Matching for instanceof
Java 15	2020	Sealed Classes (preview), Hidden Classes

Java 16	2021	Finalized Records, improved memory management.
Java 17 **(LTS)**	2021	Sealed classes, pattern matching, enhancedAPIs
Java 18	2022	Simple Web Server, UTF-8 as default charset
Java 19	2022	Virtual Threads (preview), Structured Concurrency
Java 20	2023	Scoped Values, Pattern Matching improvements
Java 21 **(LTS)**	2023	Virtual Threads (stable), Sequenced Collections
Java 22	2024	Garbage Collection & API enhancements
Java 23	2024	Finalized Foreign Function & Memory API, Primitive Type Patterns (preview)
Java 24	2025	Performance optimizations, language refinements
Java 25 **(LTS)**	**2025**	Long-term support, release, stability & performance improvements

Oracle plans to release future LTS versions every two years.

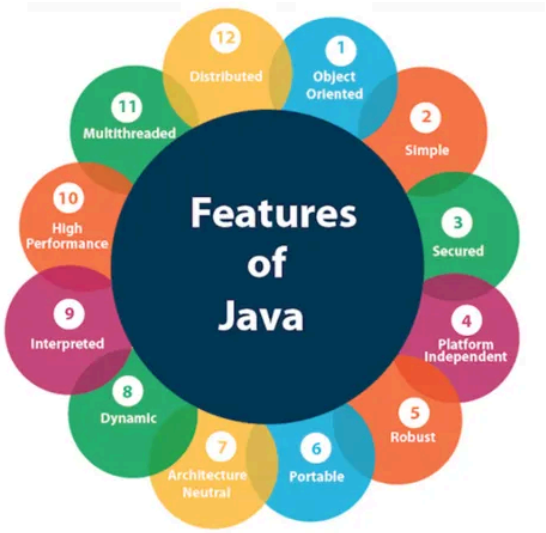
What is LTS?

- **Long-Term Support (LTS):**
    - LTS versions are versions of Java that are supported for a longer period (typically **8 years** of public support and 5 years of extended support) by Oracle. These versions receive regular updates, including bug fixes, security patches, and performance improvements during their support cycle.
    - LTS versions are intended for organizations and enterprises that need stability and reliability for their production systems. They are often chosen for large-scale applications where long-term stability is essential.
    - Example: Java 8, 11, 17, 21,25
- **Non-LTS:**

- Non-LTS versions of Java are released every six months as part of the **regular Java release cycle**. These versions are supported for a shorter period (usually **6 months**) and are intended for developers who want to try out the latest features and improvements.

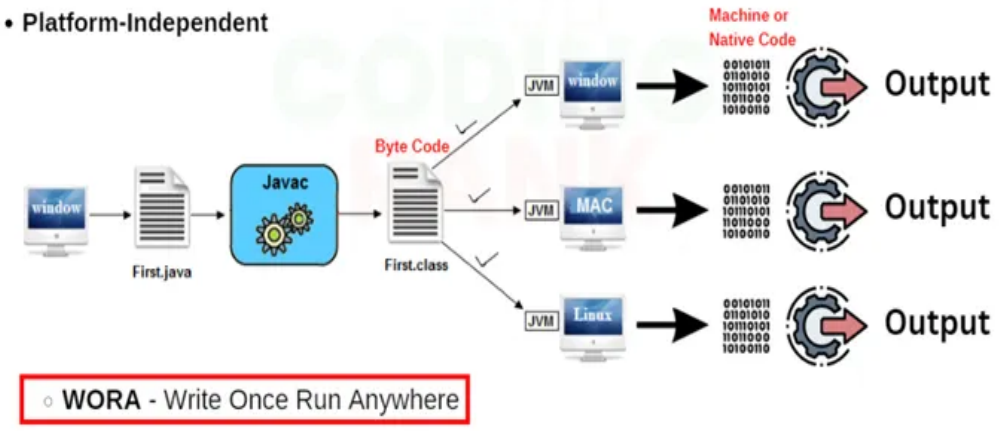
- Non-LTS versions are suitable for developers and organizations who are interested in exploring new features and don't require long-term support. They are less stable for large-scale production systems due to their shorter support lifespans.

Features of Java:



1. **Platform Independent:**
   - Java code is compiled into bytecode, which can run on any system with a JVM, regardless of the operating system.

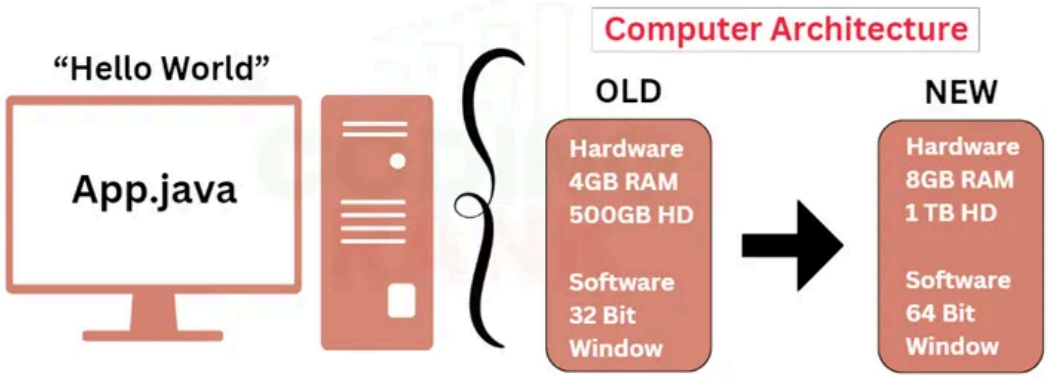
   - Example: A USB drive works on both Windows and Mac because it follows a universal standard, like Java bytecode.



2. **Object-Oriented**

    - Everything in Java is treated as an object with properties and behaviors.
    - Example: A car has properties (color, model) and behaviors (start, stop), similar to how Java organizes code

3. **Architectural Neutral -** Talks about the same platform/system



4. **Simple**

    - It is a simple programming language because complex topics like Pointers, Operator Overloading, and others have removed from Java.

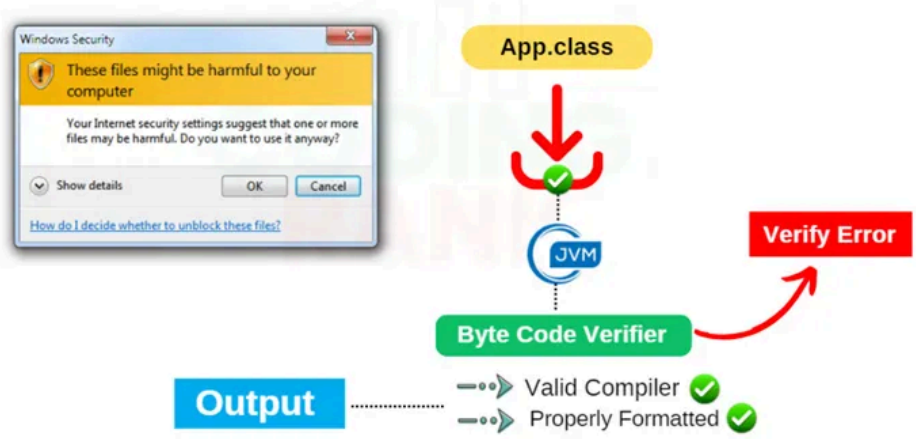
- It is Platform-Independent and Architecturally Neutral, and a Portable Programming Language

5. **Robust**

    - Java handles errors through exceptions and has strong memory management, reducing crashes.
    - Example: Like seat belts in a car, exception handling ensures safety during unexpected errors.

6. **Secure**

    - Java uses a virtual environment (JVM) and restricts direct memory access, protecting against viruses.
    - Example: A bank's ATM software runs securely without exposing sensitive information.



7. **Multithreaded**

    - Java supports multiple threads to perform tasks simultaneously.
    - Example: A music app can download a song while playing another one.

8. **Distributed**

    - Java makes it easy to develop programs that work across networks.

- Example: Cloud storage apps like Google Drive use Java for network interactions.

9. **High Performance**
    - Just-In-Time (JIT) compilation enhances speed.
    - Example: JIT works like a chef preparing food only when ordered, making things faster and more efficient.
10. **Dynamic and Extensible**
    - Java supports dynamic loading of classes and integration of libraries.
    - Example: Adding a new plugin to your browser without restarting it.

11. **Portable** - Java programs can run on various devices without modification. - Example: Like carrying your email account on different devices with the same login credentials.

Difference Between C++ and Java:

Feature	C++	Java
Nature	Hybrid language	Pure OOP (almost)
Platform	Platform dependent	Platform independent
Compilation	Direct machine code	Bytecode + JVM
Memory Management	Manual (`new` / `delete`)	Automatic (Garbage Collector)
Pointers	Fully supported	No pointer arithmetic
Multiple Inheritance	Supported using classes	Not supported using classes (only interfaces)
Operator Overloading	Supported	Not supported
Preprocessor	Has `#include`, `#define`	No preprocessor
Security	Less secure	More secure
Speed	Faster	Slightly slower

Use Cases	Better Language
System software	C++
Application software	Java
Large enterprise apps	Java
Games & graphics	C++

Categorisation of Java:

- We have 3 editions of Java for building different kinds of applications

Category	Description	What It Can Do
Java SE (Standard Edition)	Core Java programming for building standalone applications.	Create desktop apps, console programs, and perform basic operations like file handling, database connectivity, and multithreading.
Java EE (Enterprise Edition)	Tools for building large-scale, distributed, and secure web applications.	Build enterprise applications like e-commerce systems, banking software, and web services using Servlets, JSP, and EJB
Java ME (Micro Edition)	Designed for resource-constrained devices like mobile phones and embedded systems	Develop applications for IoT devices, smart appliances, and mobile phones.

Note:- JSE is the basic foundation of the remaining 2 other editions.

JDK: (Java Development Kit)

- Java Standard Edition comes in the form of a specification, and the implementation of this specification is the JDK software, also known as Java SDK (Java Standard Development Kit).

This JDK software is used for developing and executing Java applications.

Different types of JDK vendors in Java!

- When developing or running Java applications, it is necessary to install a **JDK (Java Development Kit)** on your local or production system. There are several JDK vendors in the Java ecosystem, each providing its own distribution of Java with varying support and features.

1. **OpenJDK**:
    - **OpenJDK** is the **official reference implementation** of the Java Standard Edition (SE) platform. It is an open-source project with contributions from **Oracle** and the broader Java community.
    - Other JDK vendors typically base their JDK distributions on **OpenJDK**, making necessary changes or improvements while maintaining compatibility with the **TCK** (Technology Compatibility Kit), which ensures that the JDK adheres to Java standards.
    - Website: https://openjdk.org/
2. **Oracle JDK**:
    - **Oracle JDK** is Oracle's commercial JDK, aimed primarily at **enterprise customers** who require **stability** and **Oracle support** for their applications.
    - While Oracle JDK was free for personal use and development, starting with **Java 11**, Oracle began charging for production use of Oracle JDK under a subscription-based model. It offers paid support, updates, and bug fixes.
    - Website: https://www.oracle.com/java/
3. **Amazon Corretto**:
    - **Amazon Corretto** is a free, multiplatform, production-ready distribution of OpenJDK, maintained by **Amazon**. It includes long-term support with regular security updates and performance improvements.

- Corretto is used internally by Amazon for running its Java-based applications, and it is available for download at no cost for use in production environments.
- Website: https://aws.amazon.com/corretto/

4. **Red Hat OpenJDK**:
    - **Red Hat** offers a commercially supported OpenJDK distribution with a focus on **enterprise** customers.
    - Red Hat provides long-term support for its OpenJDK builds and offers subscription-based services, including security patches, updates, and performance enhancements.

Conclusion:

There are several JDK vendors in the Java ecosystem, each providing different levels of support, performance, and stability. While **OpenJDK** is the reference implementation, other vendors like **Oracle**, **Adoptium**, **Amazon Corretto**, **Red Hat**, and **IBM** offer customized distributions to cater to enterprise needs or specific use cases. Developers can choose a vendor based on their requirements for support, security, and performance.

Installing JDK

- **Download JDK:** Visit the official Oracle website or Amazon-corretto to download the latest version of the JDK.
- **Install JDK:** Follow the installation instructions for your operating system (Windows, macOS, or Linux).

After installation, verify the proper installation by typing the following command in the terminal:

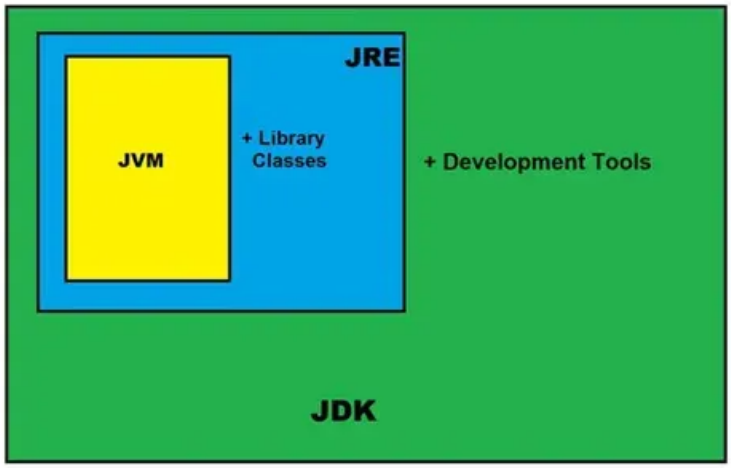
```
java -version

javac -version
```

Note: In the latest versions of JDK, the **JAVA_HOME** environment variable is typically set automatically during installation. However, if the **JAVA_HOME** variable is not set or if the command does not work after installation, you may need to manually configure the **JAVA_HOME** path and add the **bin** folder location to the system's environment variables.

This can be done by:

1. Setting the **JAVA_HOME** environment variable to point to the JDK installation directory.
2. Adding the **bin** directory inside the JDK folder to the system's **PATH** environment variable to allow command-line access to Java tools.

Java Components Architecture: JDK, JRE, and JVM



What's Inside the JDK?

1. **JRE (Java Runtime Environment):**
    - The JRE is the environment needed to execute Java applications. It includes the **JVM** and **Java libraries** necessary for running Java programs.
2. **Development Tools:**
    - **Compiler** (**javac**): Converts source code (.java) into bytecode (.class).
    - **Debugger:** Helps debug Java applications by identifying runtime issues.
    - **Other Tools:** Tools like **javadoc** (documentation generator) and **jdb** (Java debugger).

What's Inside the JRE?

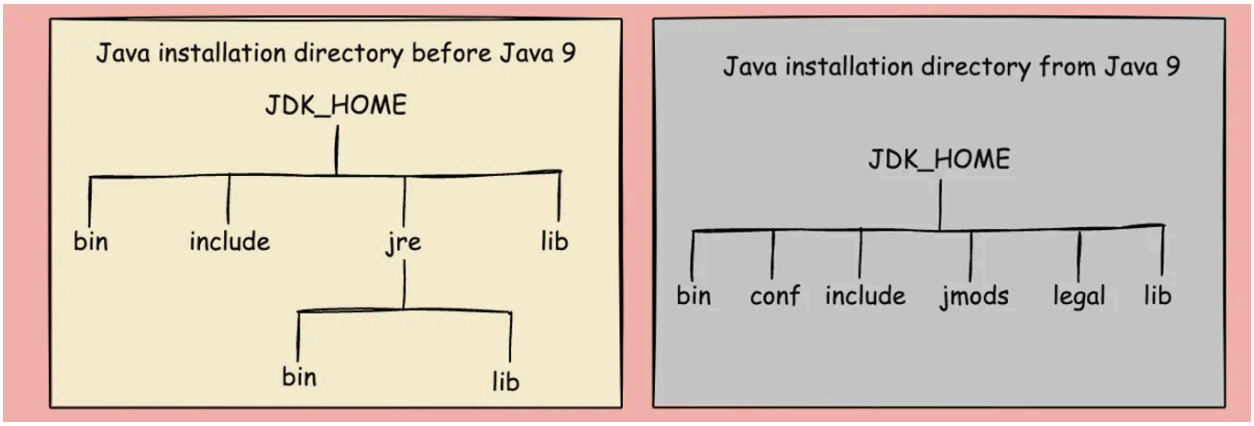
1. **JVM (Java Virtual Machine):** Core component that runs Java bytecode.

2. **Java Class Libraries:** Pre-written classes for tasks like string handling, input/output, and more. Examples include:
    - java.lang (core utilities).
    - java.util (data structures like lists, maps).

How JDK, JRE, and JVM Are Related

Component	Purpose	Included In
JVM (Java Virtual Machine)	Executes Java programs by translating bytecode into machine code and managing memory	JRE
JRE (Java Runtime Environment)	Provides the runtime environment for Java programs, including the JVM and Java libraries	JDK
JDK (Java Development Kit)	Complete toolkit for Java development, including JRE, compiler, debugger, and other tools	Standalone (Developer toolset)

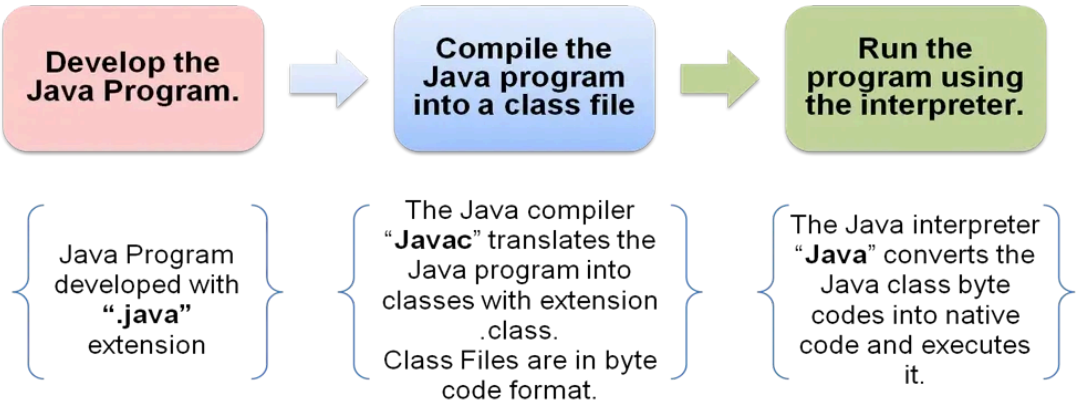
Note: Since Java 9, only the JDK is offered, not the JRE separately. This change reduces the Java installation size by eliminating duplicate files. If needed, the JRE separately, Organization, or Developers can build their own custom JRE using the jlink option introduced in **Java 9.**



Basic Steps To Develop a Java Program:

There are two phases involved in Java application execution:

1. Compilation phase
2. Execution phase



- In the Compilation phase, The java compiler compiles our code into a different format called Java **bytecode.**
- This Java compiler comes with the JDK software.

- This Java bytecode is platform-independent, which means it can run on the Window, Mac, Linux, or any operating system that has **JRE (Java Runtime Environment).**
- We can also download this JRE for various Operating-system separately.
- This JRE has a software component called **JVM (Java Virtual Machine)**, which takes our Java bytecode and translates it to the native code for the underlying OS.
- If we are in the Window Operating-system, this JVM will convert our java byte-code to the window Operating-system understandable native code, and if we are in the Linux environment, then this JVM will convert our java byte-code to the Linux OS understandable native code. With this architecture only, our Java applications are portable or platform-independent.

We can write a Java program on a Windows machine and execute it in Linux or macOS, or any other OS that has JRE.

Basics of a Java Application

- The smallest building block in a Java application is a function/method.
- A method is a block of code that performs a well-defined task. example: a method for adding 2 numbers, a method for validating user input, a method for calculating interest, etc.
- The basic syntax of a method in a Java application is:

```
returnType methodName(){
        //method body
}
```

- Some methods return some value, whereas some methods do not return any value.
- For the method that does not return any value, we apply **void** as a return type.
- **void** is a reserved keyword in Java; the method name should be proper and descriptive.
- Example:-

```
void calculateInterest(){

}
```

- We can pass some parameters to this method also; we use parameters to pass the value to our method. For example, amount, rate of interest, duration, etc., and inside the pair of parentheses { }, we write the actual implementation of Java code.
- **Every Java program should have at least one method. And that method is called the main method.**
- **The main method is the entry point of our Java application.**
- Whenever we execute a Java program, the main method gets called, and the code inside this main method gets executed.
- These methods don't exist as their own; they always belong to a class.

A class is like a container of one or more methods.

- We use the class to organize our code in the Java application.
- Every Java program should have at least one class, which contains the main method.

Our first Java program:

- Open the text editor(Notepad) and create a file called Main.java

Main.java

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

                System.out.println("Hello World");
        }
}
```

- Now, open the command prompt at the file location
- Compile the above .java file using the following command:

```
javac Main.java
```

- Here we will get the **Main.class** file at the same location.

- Now run the Main.class using the following command.

```
java Main
```

Explanation of the above Java Application

- Here, class is a keyword by which we define a class in Java. We should give a proper descriptive name to the class.

- In Java, all classes and methods should have an access modifier.

- An access modifier is a special keyword that determines if other classes and methods in this program can access these classes and methods.

We have various access modifiers, such as public, private, and so on. Most of the time, we use the public access modifier.

- So, the basic structure of a Java program contains a class, and inside the class, we have a main method.

- To name our classes, we use **P**ascal**N**aming**C**onvention(first letter of every word in uppercase and to name our methods, we use **camelNamingConvention**.

Inside Java programming, we have a concept called a package. We use the package to group related classes, so as our application grows, we are going to end up with many classes, so we should properly organize these classes inside different packages.

- By convention, the base package for a Java project is the domain name of your company in reverse. It does not mean we should have an actual domain registered. This is just a way to create a namespace for our classes.

  Example: **com.masai;**

- So every class we create in our Java application should belong to a package. We are going to talk about packages in more detail in our upcoming session.

Note: All Java files should have a **.java** extension. And every statement in the Java application should be terminated with a semicolon.

- The **static** is a keyword; we will talk about this keyword later. For now, just remember that the main method in our program should always be static. The return type of this method is void, which means this method is not going to return any value.
- In the main method, we have one parameter; we can use this parameter to pass values to our program. We will talk about this parameter in our upcoming session.

```
System.out.println("Hello World");
```

- Here, **System** is a predefined Java class, which belongs to **java.lang** package. Inside this class, we have various members, **out** is a member (field) who belongs to this System class.
- The type of this **out** field is the **PrintStream class.** This PrintStream is another predefined class in Java. The **println** method belongs to the **PrintStream** class.

So here we are calling or executing the println method inside our main method.

- Inside the parentheses of this println method, we can pass any value that we want to print on the terminal or console.
- Here, “Hello World” is textual data, in Java whenever we deal with textual data, we should always surround it with double-quotes. which is known as a string.
- In Java, a string is a sequence of characters.

Select Java Editor:

We can develop our Java application inside a normal text editor, like a notepad, and execute the application on the command prompt or terminal. But in real-time to develop a Java application, we need to use an **IDE software** (Integrated Development Environment).

Inside an IDE software, all the application development environments, like an editor, terminal, code intelligence, etc., are available in one place. With the help of an IDE software, our application development speed and productivity will be improved.

There are many IDE software are there like:

IntelliJ IDEA

Eclipse (STS)

NetBeans

Eclipse Installation:

https://www.eclipse.org/downloads/

- **Download Eclipse:** Go to the Eclipse download page.
- **Install Eclipse:** Follow the instructions for your operating system to install the Eclipse IDE for Java Developers. This IDE includes the necessary tools to write, compile, and run Java programs.

Create and Run Your First Java Program (Hello World) in Eclipse

1. **Open Eclipse IDE:** After installation, launch Eclipse.
2. **Create a New Java Project:**
    - Click on File → New → Java Project.
    - Enter a project name (e.g., **HelloWorldProject**) and click Finish.
    - Uncheck the module section.
3. **Create a Java Class:**
    - Right-click on the **src** folder in the Project Explorer.
    - Click New → Class.
    - Name the class **Demo** and package name as **com.masai**
    - Check the option public static void main(String[] args) to add the main method automatically.
    - Click Finish

Demo.java

```
package com.masai;
```

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

		System.out.println("Hello World");
	}
}
```

4. **Run the Program:**
    - Click the Run button (green arrow) in Eclipse or press Ctrl+F11.
    - The output Hello, World! will appear in the Eclipse console

Basics of Java Programming:

Java has provided the following list of tokens to prepare Java applications.

1. Identifiers
2. Variables
3. Literals
4. Keywords/ Reserved words
5. Operators

1. Identifiers:

- An **identifier** is the name used to identify a variable, method, class, interface, or any other user-defined item in Java.
- It is a sequence of characters that helps in distinguishing one element from another in a program.

Rules for Naming Identifiers in Java:

1. **Allowed Characters**:
    - Identifiers can contain letters (`a-z`, `A-Z`), digits (`0-9`), the underscore (`_`), and the dollar sign (`$`).
    - They must start with a letter, underscore, or dollar sign.
    - **Digits cannot be the first character**.
2. **Case Sensitivity**:
    - Java identifiers are **case-sensitive**. For example, `myVariable` and `myvariable` are considered different identifiers.
3. **No Reserved Keywords**:
    - Identifiers cannot be the same as Java **keywords** (e.g., `int`, `class`, `public`, `if`, etc.).
    - Reserved words like `true`, `false`, and `null` also cannot be used as identifiers.
4. **Length**:
    - Identifiers can be of any length, but it's best practice to use meaningful names for clarity.
    - Long identifiers are not recommended unless necessary for clarity.

Best Practices for Naming Identifiers:

In Java applications, it is suggested to provide identifiers with a particular meaning.

1. **Descriptive Names**:
    - Choose meaningful names that describe the role of the identifier in your program (e.g., `studentName`, `calculateArea`).
2. **Camel Case**:
    - Use **camelCase** (starting with a lowercase letter) for variable names, method names, and package names:
        - Example: `firstName`, `getArea()`.
    - **PascalCase** (starting with an uppercase letter) is generally used for class and interface names:

        - Example: `Student`, `ShapeArea`.

3. **Avoid Single-letter Identifiers**:
    - Avoid using single letters like `x`, `y`, or `z` unless they represent a commonly known concept (e.g., mathematical formulas or loop variables).
4. **Consistency**:
    - Stick to a naming convention throughout the codebase (e.g., consistently using camelCase).

Examples of Valid and Invalid Identifiers:

- **Valid Identifiers**:
    - `studentName`
    - `firstName`
    - `calculateArea`
    - `_myVariable`
    - `$accountBalance`
- **Invalid Identifiers**:
    - `123abc` (starts with a digit)
    - `class` (reserved keyword)
    - `public` (reserved keyword)
    - `@myVariable` (contains an invalid character)
    - `my-variable` (contains a hyphen, which is not allowed)

Special Characters in Identifiers:

- The **underscore** (`_`) and the **dollar sign** (`$`) are allowed in identifiers but are rarely used in modern Java development.
    - **Dollar sign** (`$`): Often used in automatically generated names (e.g., for inner classes).
    - **Underscore** (`_`): In recent Java versions, the underscore is discouraged for variable names and class names.

Identifier name	Valid ?	Reason
firstName1	Yes	Contains letters and number
$personName	Yes	Can start with $, _ and a letter
7firstName	No	Can not start with a number
age_of_person	Yes	Contains letters and allowed special char _
person name	No	Contains space in between the name of the variable
personName*	No	Contains * which is not allowed. Only $ and _ are allowed
$	Yes	Only $ is a valid allowed name
_	No	Since JDK 9, the use of an underscore as a standalone identifier is no longer permitted
FirSTName	Yes	Though it is not following camelCase, it is still valid
654	No	Using all numbers is not allowed

2. Java Variables:

- A variable in Java is a container used to store data values. Before using a variable, you must declare it, defining both the type and the name. The data type specifies what kind of data the variable will store (e.g., integer, string, or boolean)
- The variable name must be a valid identifier.

Declaration of Variables:

```
dataType variableName = value;
```

- dataType: Specifies the type of the data (e.g., int, float, char, String)
- variableName: The name of the variable (e.g., age, height, result)
- value: The initial value of the variable (e.g., 25, 5.6, 700)

Example:

```
int age = 32;

 //Here we have declared a variable age and assigned the value 32 in a single statement. //We can divide it into multiple statements.
```

```
int age; //variable declaration
age = 32; //initialization of this variable
```

Example:

```
System.out.println("Masai");
```

Here, Masai is the hard-coded value. If we want to print Masai in multiple places, for example, 10 times, and later if we want to change the Masai to MasaiSchool, then we need to change it at multiple places.
So to solve this, we should follow the coding standards and store this Masai in a variable like

```
String companyName = "Masai";
```

And use this variable name in multiple places instead of the hard-coded value.

```
System.out.println(companyName);
```

Here, we need not change the value multiple times.

3. Literals:

A **literal** is a constant value that is directly represented in the source code. It represents fixed values like numbers, characters, or boolean values used in expressions or assignments.

Types of Literals

1. **Integer Literals**:
    - Represent whole numbers.
    - It can be written in **decimal**, **binary**, **octal**, or **hexadecimal**.
    - Syntax:
        - Decimal: `123`, `456`
        - Binary: `0b1011` (or `0B1011`), where `0b` or `0B` denotes binary.
        - Octal: `075` (leading zero for octal numbers).
        - Hexadecimal: `0xA1F` or `0X1F` (prefix `0x` or `0X`).
2. **Floating-Point Literals**:
    - Represent numbers with a fractional part.
    - It can be written as **float** (single precision) or **double** (double precision).
    - Syntax:
        - `3.14` (double by default).

        - `3.14f` (float literal, suffixed with `f` or `F`).
3. **Character Literals**:
    - Represent a single character.
    - Enclosed in **single quotes** (`'`).
    - Example: `'A'`, `'9'`, `'%'`.
4. **String Literals**:
    - Represent a sequence of characters.
    - Enclosed in **double quotes** (`"`).
    - Example: `"Hello, World!"`
5. **Boolean Literals**:
    - Represent the two possible boolean values: `true` and `false`.
    - Syntax:
        - `true`, `false`.
6. **Null Literal**:
    - Represents the null value, which means "no object" or "empty reference".
    - Syntax:
        - `null`.

Example of Literals:

```
int number = 100; // integer literal
double pi = 3.14159; // floating-point literal
char letter = 'A'; // character literal
String greeting = "Hello"; // string literal
boolean isJavaFun = true; // boolean literal
```

3. Keywords / Reserved words:

- In Java, a **keyword** is a reserved word that has a specific meaning and purpose in the programming language. These keywords cannot be used as identifiers (such as variable names, method names, or class names) in the code because they are already predefined with a particular meaning in the Java language. If a keyword is used as an identifier, it will result in a **compilation error**. This ensures that the keywords maintain their intended functionality and are not overwritten or confused with user-defined names.

- **List of Keywords in Java:**

abstract	continue	for	new	switch
assert***	default	goto*	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum****	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp**	volatile
const*	float	native	super	while

* not used
** added in 1.2
*** added in 1.4
**** added in 5.0

Java syntax requires all code, including keywords, to be case-sensitive, therefore public and Public are different.

Example:

```
String class = "Masai"; // error because class is a reserved keyword in Java.
String Class = "Masai"; // no error, but bad programming practice.
```

Data Types:

- Java is strictly a typed programming language, whereas in Java applications, before representing data, we have to confirm which type of data we are representing. In this context, to represent the type of data, we have to use "**data types**".

Java supports two types of data types:

1. **Primitive Data Types:** Basic data types that store values directly.

2. **Reference Data Types**: These store references (or memory addresses) to objects.

Primitive Data Types:

- Java supports 8 primitive data types under 4 categories:

    a. **Integers:**

        1. byte (1 byte or 8 bits) default value is 0
        2. short (2 bytes or 16 bits) default value is 0
        3. int (4 bytes or 32 bits) default value is 0
        4. long (8 bytes or 64 bits) default value 0

    b. **Real-numbers:**

        1. float (4 bytes or 32 bits) default value is 0.0
        2. double (8 bytes or 64 bits) default value is 0.0

    c. **Characters**

        1. char (2 bytes or 16 bits) the default value is empty space \u000

    d. **Boolean**

        1. boolean (1 bit) default value is false.

Note: All the decimal point values are, by default, treated as double in Java.

type	default	size	range
boolean	false	1 bit	*NA*
char	\u0000	16 bits / 2 Bytes	*\u0000 to \uFFFF*
byte	0	8 bits / 1 Byte	*–128 to 127*
short	0	16 bits / 2 Bytes	*–32768 to 32767*
int	0	*32* bits / 4 Bytes	*–2147483648 to 2147483647*
long	0	*64* bits / 8 Bytes	*–9223372036854775808 to 9223372036854775807*
float	0.0	*32* bits / 4 Bytes	*1.4E–45 to 3.4028235E38*
double	0.0	*64* bits / 8 Bytes	*4.9E–324 to 1.7976931348623157E308*

Reference (Non-Primitive) Data Types:

- Non-primitive data types, also known as reference types, are objects that are created using classes, and they hold references (addresses) to the data rather than the data itself. They are more complex than primitive types and are used to store large amounts of data.
- **Default Value:** null (indicating no reference to any object or memory location)

Common Non-Primitive Data Types:

1. **String:** Represents a sequence of characters.
    - Example: String greeting = "Hello World!";
2. **Arrays:** Stores multiple values of the same type.
    - Example: int[] numbers = {1, 2, 3};
3. **Class Object:** An instance of a class.
    - Example: MyClass obj = new MyClass();
4. **Interfaces:** This represents a contract that a class can implement.

- Example: Runnable runnable = new MyRunnable();

5. **Enums:** A Special class that represents a collection of constants.
    - Example: enum Color {RED, GREEN, BLUE};

Examples of Primitive data types:

1. **Integers types:**

```
byte ageOfPerson = 50;

short totalStudents = 650;

int totalAudience = 65000;

long totalNumOfTrees = 2147483648; // Error

long totalNumOfTrees = 2147483648L;
```

2. **Decimal point values:**

- **Float** can store up to 7 decimal digits, while **double** can store up to 15 decimal digits.
- By default, any decimal point value is treated as a `double`. To explicitly specify a value as a `float`, you need to postfix it with `f` or `F`. Optionally, you can also postfix a decimal value with `D` or `d` to specify it as a `double`.
- In general, **double** is the better choice when higher precision, a wider range of values, and greater accuracy are required.

```
float f = 3.55; //error here, this value will be treated as double.
float f = 3.55f;
```

Experiment:

```
float x = 5.55f;
double y = 5.55;
```

```
System.out.prinltln(x == y); //

System.out.println(0.1 + 0.2 + 0.3); //
```

Note: 5.55f will be stored in memory as the nearest representable value, and actually it will be stored as: 5.55000019073486328125

```
float f = 5.55f;

System.out.println(f);           // 5.55
System.out.println((double) f);  // 5.550000190734863
```

`double` is **more accurate** than `float`

Decimal numbers are stored in **approximate form**, not exact.

BigDecimal:

- `BigDecimal` is a Java class used to perform exact and precise decimal calculations.
- java.math.BigDecimal
- Unlike `float` and `double`, **BigDecimal does NOT use binary floating-point representation**.

  Example:

  ```
  System.out.println(0.1 + 0.2); // 0.30000000000000004
  ```

This is **dangerous** in:

- Banking systems
- Financial calculations
- Tax, interest, GST
- Currency conversions

BigDecimal solves this problem.

Example:

```
BigDecimal a = new BigDecimal("0.1");
BigDecimal b = new BigDecimal("0.2");

System.out.println(a.add(b)); //0.3
```

NEVER create `BigDecimal` like this:

```
BigDecimal bd = new BigDecimal(0.1);
System.out.println(bd);
```

Because `0.1` is already **approximate** as a `double`.

Operators (`+ - * /`) do NOT work with BigDecimal class

```
BigDecimal a = new BigDecimal("10.5");
BigDecimal b = new BigDecimal("2.5");

a.add(b);       // 13.0
a.subtract(b);  // 8.0
a.multiply(b);  // 26.25
a.divide(b);    // 4.2
```

Using Underscore in Numeric Literals (Java 7 and later):

- Starting from **Java 7**, underscores can be used within numeric literals to improve the readability of the code. However, the compiler will remove the underscores internally when processing the value.
- This feature is similar to using punctuation marks like commas in mathematics to separate large numbers for easier comprehension.

```
int num = 1_00_00_000; // The number is equivalent to 100000000
```

Restrictions on Using Underscores in Numeric Literals:

a. **Cannot be placed at the beginning or end** of the number.
   - Invalid: `_5`, `5_`

b. **Cannot be placed adjacent to a decimal point** in floating-point literals.
   - Invalid: `6._0`

c. **Cannot appear before a suffix** used to indicate a `long` or `float` value (such as `L` or `F`).
   - Invalid: `5_65_L`

d. **Cannot be used within a string literal**.
   - Invalid: `"10_00_00"`

By following these rules, you can use underscores effectively to improve the readability of large numeric values in your code.

Other formats supported by integer and floating data types:

- Apart from the Decimal number format, we can also define values in the following formats by using integer and floating primitive data types.

  - **Octal number format**
  - **Hexadecimal number format**
  - **Binary number format**

Binary number format:

- A literal can also be expressed in the binary number format. Every integer literal in binary starts with either 0b or 0B.

  // binary number 0B011 will be 3 in decimal

  ```
  int myBinaryNumber = 0B011;

  System.out.println(myBinaryNumber);
  ```

- Here's an example to illustrate the process to convert the octal number 1101, or 0B1101, into decimal

  (1*2^0)+(0x2^1)+(1x2^2) +(1*2^3)

=1+0+4+8 =13

Octal number format

- If a literal starts with a zero and has at least two digits, it is treated as an octal number. In the following line of code, a decimal value of 25 (031 in octal) is assigned to the variable myNumber:

```
// octal number 031 will be 25 in decimal

int myOctalNumber = 031;
```

- Here's an example to illustrate the process to convert the octal number 346, or 0346, into decimal.

(6x8^0)+(4x8^1)+(3x8^2)

= (61)+(4x8)+(3x64)

= 230

Hexadecimal number format

- Every literal in the hexadecimal number format commences with either Ox or OX, signifying zero followed by an uppercase or lowercase 'X'. These literals must consist of at least one hexadecimal digit. The hexadecimal number format employs 16 digits, encompassing 0-9 and A-F (or a-f), with the case of letters A-F being insignificant.

```
// hexa number 0x453f will be 17727.0 in decimal

float myHexaNumber = 0x453f;
```

- Here's an example to illustrate the process to convert the hexa number 1A3, or 0X1A3, into decimal.

  (3x 16^0)+(10x16^1)+(1*16^2)

  =3+(10*16)+(1*256)

  = 3+160+265

  = 419

3. **Boolean type:**

- The `boolean` data type in Java represents two possible values: `true` or `false`. These values are used to represent logical conditions and decisions in programming.
- A **boolean variable** is typically used in control flow statements (like `if`, `while`, `for`, etc.) to evaluate a condition. Based on the result of that condition, different code statements can be executed.

  Examples: boolean variables declaration

```
boolean hasChildren; // Represents whether someone has children (true/false)
boolean isValid; // Represents whether a value is valid (true/false)
boolean isPass; // Represents whether someone has passed (true/false)
boolean isMarried; // Represents whether someone is married (true/false)
```

4. **Character type:**

- The `char` data type in Java is used to store a single character. The character must be enclosed in **single quotes** (e.g., `'M'`).
- The `char` type in Java can store values ranging from `0` to `65535`. It is a 16-bit Unicode value that represents a character.
- **Unicode**: Unicode is a universal character encoding standard capable of representing most of the world's written languages. It assigns a unique number to every character,

which is used in various programming languages, including Java. https://en.wikipedia.org/wiki/List_of_Unicode_characters

- A `char` in Java represents a single 16-bit Unicode character, which allows for the storage of not just standard ASCII characters but also extended characters, including emojis, international symbols, and more.
- **ASCII values: American Standard Code for Information Interchange**
    - A - Z (65 - 90)
    - a - z (97 - 122)
    - 0-9 (48 - 57)

  Note: The `char` Unicode range (0 to 65535)

Example:

```
char ch = 'M';
char z = 456; // Valid because the numeric value 456 corresponds to a Unicode character
char num = 2; // Assigning the Unicode value directly
char num = '2'; // Assigning the character '2'
char c = 65500; // Valid because 65500 is within the Unicode range (0 to 65535)
char smiley = '\u263A'; // Unicode for ☺ (smiley face)
String text = "I love coding! \u2764"; // Unicode for heart symbol (❤)
System.out.println(text);
```

Experiment:

```
char a = 'M';
char b = 'P';
```

System.out.println(a + b); // This will print the sum of the ASCII values of 'M' and 'P'.

- Few sample escape sequences & their representation value,

  \b - Backspace, Moves cursor one step back

  \t - Horizontal tab, Adds tab space

  \n - Newline, Moves cursor to next line

  \f - Form feed, Page break (rarely visible)

  \r - Carriage return, Moves cursor to start of line

  \"' - Double quote, Prints " inside string

  \' - Single quote, Prints ' inside string

  \\ - Backslash, Prints \

  Example:

```
System.out.println("Name\tAge\tCity");
System.out.println("Ravi\t25\tDelhi");
```

  Output:

```
Name    Age     City
Ravi    25      Delhi
```

Advantages of primitive data types:

1. **Memory Allocation**

    - Data types help determine the memory size required to store a variable. Each data type has a predefined memory size, ensuring that the appropriate amount of memory is allocated for different types of values.

    Example:

    ```
    int i = 10; // The 'int' data type allocates 4 bytes of memory to store the value 10.
    ```

2. **Type Safety:**

    - Java is a strongly typed language, meaning variables are strictly associated with specific data types. This ensures that only compatible values can be assigned to a variable, reducing the risk of type errors.

    Example:

    ```
    int i = 10;
    // i = "Hello";  // Compilation error: incompatible types
    ```

3. **Range Constraints:**

    - Data types define the valid range of values a variable can hold. For instance, the `byte` type can store values from -128 to 127. Assigning a value outside this range will result in a compilation error.

    Example:

    ```
    byte b = 130; //Invalid assignment, as 'byte' can only store values in the range of -128 to 127.

    byte b = 125;  // Valid assignment.
    ```

4. **Performance Optimization:**

    - Choosing the appropriate data type can help optimize the memory and performance of an application. Smaller data types, such as `byte` or `short`, consume less memory and may provide faster processing, especially when dealing with large datasets.

    Example:

    ```
    byte b = 100;  // Uses less memory (1 byte) compared to an 'int' (4 bytes).
    ```

Wrapper classes:

- In Java, **wrapper classes** are classes that provide a way to use primitive data types (such as `int`, `char`, `boolean`, etc.) as objects. Each of the eight primitive data types in Java has a corresponding wrapper class in the `java.lang` package.
- Wrapper classes allow you to treat primitive types as objects, which is useful when working with Java collections (such as `ArrayList`) or when needing to pass primitive data types as objects.

Primitive Data Types	Wrapper classes
byte	java.lang.Byte
short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character
boolean	java.lang.Boolean

- Wrapper classes provide several utility methods to perform operations on primitive data types, such as finding the minimum and maximum values for a particular data type, among other useful functions.

Example:

Main.java:

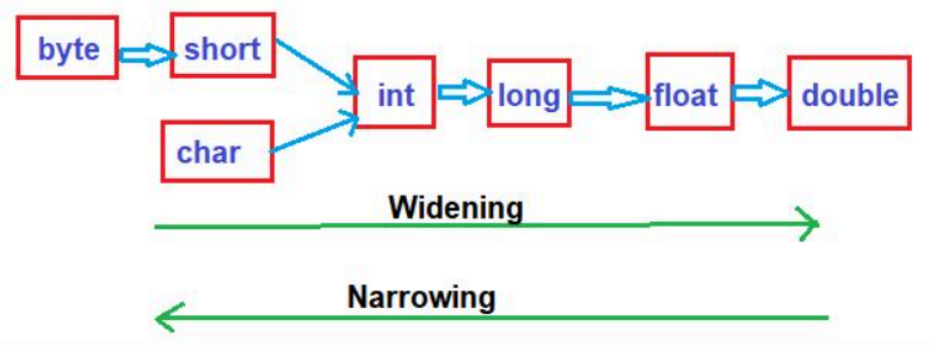
```
package com.masai;
public class Main{
```

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

                System.out.println(Byte.MIN_VALUE+"----->"+Byte.MAX_VALUE);
                System.out.println(Short.MIN_VALUE+"----->"+Short.MAX_VALUE);
                System.out.println(Integer.MIN_VALUE+"----->"+Integer.MAX_VALUE);
                System.out.println(Long.MIN_VALUE+"----->"+Long.MAX_VALUE);
                System.out.println(Float.MIN_VALUE+"----->"+Float.MAX_VALUE);
                System.out.println(Double.MIN_VALUE+"----->"+Double.MAX_VALUE);

//System.out.println(Character.MIN_VALUE+"----->"+Character.MAX_VALUE); unused charecter

//System.out.println(Boolean.MIN_VALUE+"----->"+Boolean.MAX_VALUE);  Error

                System.out.println(Character.SIZE); //16
                System.out.println(Character.isDigit('M'));
                System.out.println(Character.toUpperCase('a'));

            }
        }
```

- There are many other uses of wrapper classes as well. like Autoboxing, Auto-unboxing, Parsing a string, etc. We will discuss them in the upcoming sessions.

Type casting in Java:

- The procedure of converting one data type into its equivalent another data type is known as typecasting.
- In Java, there are 2 types of Typecasting:
    1. **Implicit Casting (Widening) / Automatic casting:**
        - Implicit casting occurs automatically when a smaller type is assigned to a larger type. It doesn't require explicit casting because there is no loss of data during the conversion.



Example:

```
int x = 10;

double y = x; // Implicit casting from int to double

System.out.println(y); // Output: 10.0
```

- Here, an int is automatically converted to a double because a double can store larger values and decimals.

2. **Explicit Casting (Narrowing) / Manual Casting:**

- Explicit casting is required when converting a larger type to a smaller type. This is because narrowing may cause a loss of data. The programmer has to manually specify the conversion.
- If we do not perform casting, then the compiler will generate a compile-time error.

Example:

```
double a = 9.78;

int b = (int) a; // Explicit casting from double to int

System.out.println(b); // Output: 9
```

- Here, a double is cast to an int, which results in the decimal part being truncated.

Examples:

1.

```
float f = 123; //implicit casting

float f2 = 12.45; //error assigning double to float

float f2 = (float)12.45;
//or
float f2 = 12.45f;
```

2.

```
long longVal = 12345L;
int intVal = (int) longVal;  // Explicit casting from long to int
System.out.println(intVal);  // Output: 12345
```

3.

```
char charVal = 'A';
int x=  charVal;  // Casting char to int
System.out.println(x);  // Output: 65 (ASCII value of 'A')
```

4.

```
int a = 10;
//byte b = a; //Error

byte b = (byte)a;
System.out.println(b);
```

5.

```
int x = 130;

byte b = (byte)x;

System.out.println(b); // -126

//Data types range will work like a circle
```

6.

```
byte b = 65;
char c = (char) b;
System.out.println(c); //A
```

7.

```
char c = 'a';
byte b = (byte) c;
System.out.println(b); // 97
```

Why `char` is Special?

- `char` is **2 bytes (16-bit)**
- `byte` and `short` are **signed**
- `char` is **unsigned** (0 to 65535)

  `char` → `int` is allowed
  `char` → `short or byte` is NOT allowed automatically

- `short` range is :
  - Minimum value = -32,768
  - Maximum value = 32,767

Example:

```
short x = 32000;   // valid

// short y = 40000; // compile-time error (out of range)
```

Rules for Typecasting:

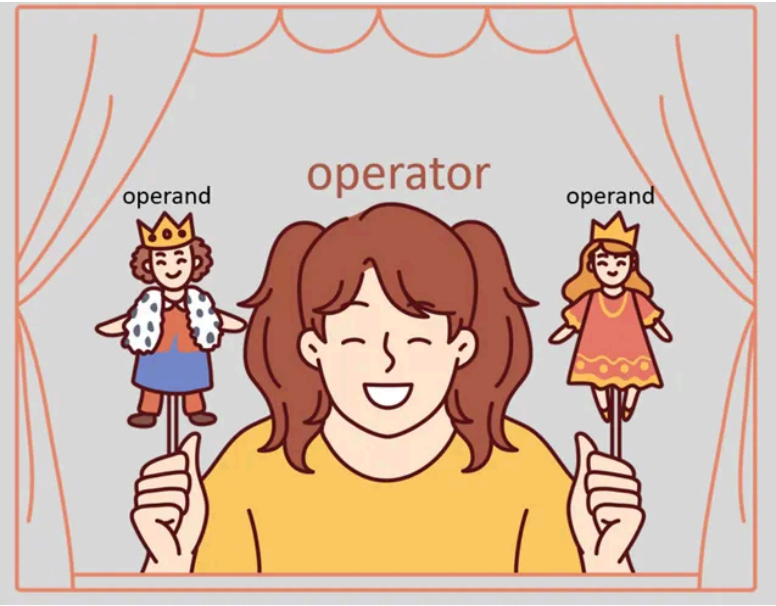
1. Implicit casting can be done when the data type is smaller and is being converted to a larger type (e.g., int to long or float).
2. Explicit casting is required when a larger data type is being converted to a smaller type (e.g., double to int).
3. In Java, we cannot typecast any data type value to any other data type. Typecasting can only be performed between compatible or **equivalent** data types. For example, we cannot typecast a `boolean` to an `int`, because these are not compatible types.

Example:

```
boolean a = true;

int b = (int) a; // This will cause a compile-time error
```

Operators and Operands In Java:

- Inside puppet shows, we have an operator who operates puppets (operands). Just like puppet shows, inside Java, we also have Operators and Operands.



- An **Operator** in Java is like a symbol that carries out a specific action on one, two, or three **operands** and generates a result.
- An **Operand** is a value or variable that is manipulated by an **operator**.

Example:

```
int result = 2 + 3;
```

- In the above expression, + and = are the operators, and the operands are 2 and 3.

- Variables can also be used in places or literals as an operand.

Example:

```
int num1 = 100;
int num2 = 200;
int result = num1+num2;
```

Java provides a wide variety of operators that allow you to perform operations on variables and values.

1. **Arithmetic operators:**
    - These operators perform basic mathematical operations:

        * : Multiplication

        / : Division

        % : Modulo

        + : Addition

        – : Subtraction

2. **Assignment operators:**
    - These operators are used to assign values to variables.

        =, assigning a value to the variables

        +=, Add and assign.

        -=, Subtract and assign.

        *=, Multiply and assign.

        /= Divide and assign.

        %=, Modulo, and assign.

3. **Relational Operators:**
    - These operators are used to compare two values. They return a boolean result (`true` or `false`).

        ==, Equal to.

        !=, Not equal to.

        < Less than.

        <=, Less than or equal to.

        >, Greater than.

        >= Greater than or equal to.

4. **Logical Operators:**
    - These operators are used to perform logical operations, typically on Boolean values.

        &&, Logical short-circuit AND: returns true when both conditions are true.

        ||, Logical short-circuit OR: returns true if at least one condition is true.

        !, Logical NOT: returns true when a condition is false and vice-versa

        &, Logical AND:

        |, Logical OR:

        ^, Logical XOR: (Exclusive OR): true ^ true = false

5. **Bitwise Operators:**
    - These operators perform operations on the binary representations of numbers.

    & (Bitwise AND) – returns bit-by-bit AND of input values.

    | (Bitwise OR) – returns bit-by-bit OR of input values.

    ^ (Bitwise XOR) – returns bit-by-bit XOR of input values.

    ~ (Bitwise Complement) – inverts all bits (one's complement).

    << (Left shift) – Shifts bits left, filling 0s (multiplies by a power of two).

    >> (Signed right shift) – Shifts bits right, filling 0s (divides by a power of two), with the leftmost bit depending on the sign.

>>> (Unsigned right shift) – Shifts bits right, filling 0s, with the leftmost bit always 0.

6. **Unary Operators:**
    - Unary operators operate on a single operand.

      -, Negates the value.

      +, Indicates a positive value (automatically converts byte, char, or short to int).

      ++, Increments by 1.

      Post-Increment: Uses value first, then increments.

      Pre-Increment: Increments first, then uses the value.

      --, Decrements by 1.

      Post-Decrement: Uses value first, then decrements.

      Pre-Decrement: Decrements first, then uses the value.

      ! Inverts a boolean value.

7. **Ternary operator:**
    - The ternary operator is a shorthand for the `if-else` statement. It takes three operands and follows the format:

      condition ? if true : if false

8. **instanceof operator:**
    - The `instanceof` operator is used to check whether an object is an instance of a specific class or subclass.

      object instanceof class/subclass/interface

Note: + Operator in Java is the overloaded operator; if both operands are numbers, then it will perform addition, whereas if at least one operand is a String, then it will perform the concatenation.

Example:

```
System.out.println(10+10); //20
```

```
System.out.println("Hello"+10); //Hello10

System.out.println("Hello"+10+10);

System.out.println(10+10+"Hello");
```

Type Promotion and Result Type in Java:

- When performing arithmetic operations between variables of different data types in Java, **type promotion** happens automatically to ensure the operation is safe. The result type is determined by the **maximum** of the involved data types. The general rule for determining the result type is:

  Result type = max(int, type1, type2, type3, ...)

Example:

```
byte b1= 10;

byte b2 = 20;

byte result = b1+b2; //Error

byte result = (byte)(b1+b2);

byte x= 10;

byte y = x+1; //Error

byte y = (byte)(x+1);
```

Example: / division operator

```
System.out.println(14/2); //7

System.out.println(14/3); //4

System.out.println(14/0); //ArithmaticException

System.out.println(25.0/2.0); //12.5

System.out.println(40.5/8); //5.0625

System.out.println(40.5/8.0); //5.0625

System.out.println(40.5/0); //Infinity

System.out.println(-40.5/0); //-Infinity

System.out.println(0.0/0); //NaN
```

Task: Using arithmetic operators in Java, write a program that calculates the total cost of items and applies a discount of 10%.

Demo.java:

```
package com.masai;

public class Demo {

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

```
        // Prices of items
        double item1 = 50;
        double item2 = 30;
        double item3 = 20;

        // Calculate total cost
        double totalCost = item1 + item2 + item3;

        // Calculate discount (10% of total cost)
        double discount = totalCost * 0.10;

        // Calculate total cost after discount
        double totalAfterDiscount = (totalCost - discount);

        // Display the results
        System.out.println("Total Cost: " + totalCost);
        System.out.println("Discount: " + discount);
        System.out.println("Total After Discount: " + totalAfterDiscount);
    }

}
```

Example: % modulus or reminder operator

```
        System.out.println(15 % 4); //3
        System.out.println(24 % 8); //0

        System.out.println(4 % 15); //4
```

```
System.out.println(8 % 0); //ArithmaticException

System.out.println(8 % 0.0); //NaN

System.out.println(0.0 % 0.0); //NaN
```

Example: You are organizing a party and need to divide 50 guests into 6 groups. The modulus operator helps you find how many people will be in the last group (remainder after division).

```
int totalGuests = 50;
 int groups = 6;
 // Find remainder (people in the last group)
 int remainder = totalGuests % groups;


 // Display the result
 System.out.println("Guests in the last group: " + remainder);
```

Arithmetic Operator Precedence in Java:

The general rule is:

1. **Multiplication (*), Division (/), and Remainder (%)** have the **same level of precedence** and are evaluated **first**.
2. **Addition (+) and Subtraction (-)** have the **next level of precedence** and are evaluated **after** multiplication, division, and remainder.
3. The **evaluation flow** for operators of the same precedence is from **left to right**.

Example:

```
int result = 10 + 2 * 5 - 3 / 1 % 2
System.out.println("Result: " + result);
```

```
// multiplication: 10 + 10-3 / 1 % 2
//Division and reminder (from left to right)
//10 + 10 - 3 % 2
//10 + 10 - 1
//20-1
//19
```

Example: Assignment operator

```
String str = "Hello";
//str = str + 9;
str+=9;
System.out.println(str); //Hello9

byte b = 10;
//b = b+20; //Error
b += 20; //implicit type conversion
System.out.println(b); //30
```

Example: Relational operator:

```
System.out.prinltn(5 == 5); //true
System.out.prinltn(5 != 5); //false
System.out.prinltn(6 != 5); //true
System.out.prinltn(5 > 5); //false
System.out.prinltn(5 >= 5); //true

boolean a= true;
```

```
System.out.prinltn(a == true); //true
```

Example: Check if a student has scored enough marks to pass the exam (passing marks 50).

```
int marks = 45;
System.out.println(marks >= 50);  // Output: false (False means the student has failed the exam)
```

Example: Logical operator: It can only be used with boolean operands.

```
System.out.println(true && true); //true
System.out.println(true && false); //false
System.out.println(true || true); //true
System.out.println(true || false); //true
System.out.println(false || false); //false
System.out.println(!true); //false

boolean a= true;
System.out.prinltn(!a); //false
System.out.prinltn(!(a == true)); //false
```

- Logical operators are used to combine multiple conditions or expressions.

```
boolean isEligible = (age >= 18) && (income > 30000);// Both conditions must be true
boolean isEligible = (age >= 18) || (income > 30000);//At least one condition must be true
```

Example: Check if a number is either even or divisible by 5.

```
int number = 15;
System.out.println(number % 2 == 0 || number % 5 == 0);
```

// Output: true (True means the number is either even or divisible by 5)

Example: Check if a number is not divisible by both 3 and 4

```
int num = 7;
System.out.println(!(num % 3 == 0 && num % 4 == 0));
```

// Output: true (True means the number is not divisible by both 3 and 4)

Example: Unary Operator: ++ increment, -- decrement

```
int x= 10;
//x = x+1;
//x +=1;
//x++;  //post increment
 ++x; //preincrement
System.out.println(x); //11
```

Note: This operator will work only on variables, but not on the values.

```
int x= 10++; //error
```

Difference between post-increment and pre-increment:

- post: first use the variable and then increment it
- pre: first increment and then use it.

Example:

```
int x = 10;
int y = 10;
System.out.prinltn(x++); //
System.out.prinltn(++y); //
```

```
int a =20;

int b= a++;

System.out.prinltn(a); //
System.out.prinltn(b); //

int num1= 40;
int num2 = num1++ + 5;

System.out.prinltn(num1); //
System.out.prinltn(num2); //
```

Example: Ternary operator:

- The ternary operator is a shorthand for if-else statements. It is used to return a value based on a condition.

```
 int age = 18;
String result = (age >= 18) ? "Adult" : "Minor";

System.out.println(result);
```

Example: instanceof operator:

- Checks if an object is an instance of a specific class or interface

```
 String str = "Hello";
 boolean isString = str instanceof String;  // true
```

Difference between the Short-circuit logical && operator and the logical & operator.

1. **&& short-circuit logical AND** is used for only boolean values, whereas **& logical AND** can be used for boolean as well as for bitwise operations
2. **&& short-circuit logical AND** will evaluate the second expression if the first operand is true. whereas **& logical AND** will evaluate both the operands.

Example:

```
int x = 10;

if(x > 15 && ++x < 20){
        System.out.println("Hello");
}
else{
        System.out.println("Hi");
}

System.out.println(x);

//we can see a difference in output by changing the && to &
```

Example: Bitwise operator:

1.

& Bitwise AND: Compares each bit of two numbers. If both bits are 1, the result is 1;

Otherwise, it is 0.

```
int result = 5 & 3;  // 0101 & 0011 = 0001 (result is 1)
```

2.

| Bitwise OR: Compares each bit of two numbers. If either of the bits is 1, the result is 1.

```
int result = 5 | 3;  // 0101 | 0011 = 0111 (result is 7)
```

3.

^ Bitwise XOR: Compares each bit of two numbers. If the bits are different, the result is 1; if they are the same, the result is 0.

```
int result = 5 ^ 3;  // 0101 ^ 0011 = 0110 (result is 6)
```

4.

~ Bitwise NOT: Inverts all the bits of the operand. This is also called the 1's complement

```
int result = ~5;  // 5 = 0101 in binary
// Invert all bits: ~0101 = 1010 (which is -6 in two's complement representation)
System.out.println(result);  // Output: -6
```

Rule: ~n = -(n + 1)

5.

<< Left Shift: Shifts the bits of the number to the left, filling with 0s at the right side. This effectively multiplies the number by 2 for each shift.

```
int result = 5 << 1;  // 5 (0101) << 1 = 10 (1010) (result is 10)
```

Rule: $x << n = x \times (2^n)$

int result = $10 << 2 = 10 \times 2^2 = 40$

6.

>> Right Shift(signed): Shifts the bits of the number to the right. This effectively divides the number by 2 for each shift.

int result = 5 >> 1; // 5 (0101) >> 1 = 2 (0010) (result is 2)

Rule: $x >> n = x \div (2^n)$

7.

>>> Right Shift(unsigned): The **Unsigned Right Shift operator (>>>)** shifts the bits of a number **to the right**, but **always fills the leftmost bits with 0**, regardless of the sign.

- It **does not preserve the sign bit**
- This operator exists **only in Java**
- For **positive numbers**, >> and >>> give the **same result**
- For **negative numbers**, >>> converts the result into a **large positive number.**

Example:

```
int x = -10;
int result = x >>> 1;
```

-10 = 11111111 11111111 11111111 11110110

Unsigned right shift by 1:

01111111 11111111 11111111 11111011

Here:

MSB = 0

The number will be positive, and it will be a very big number.

2147483643