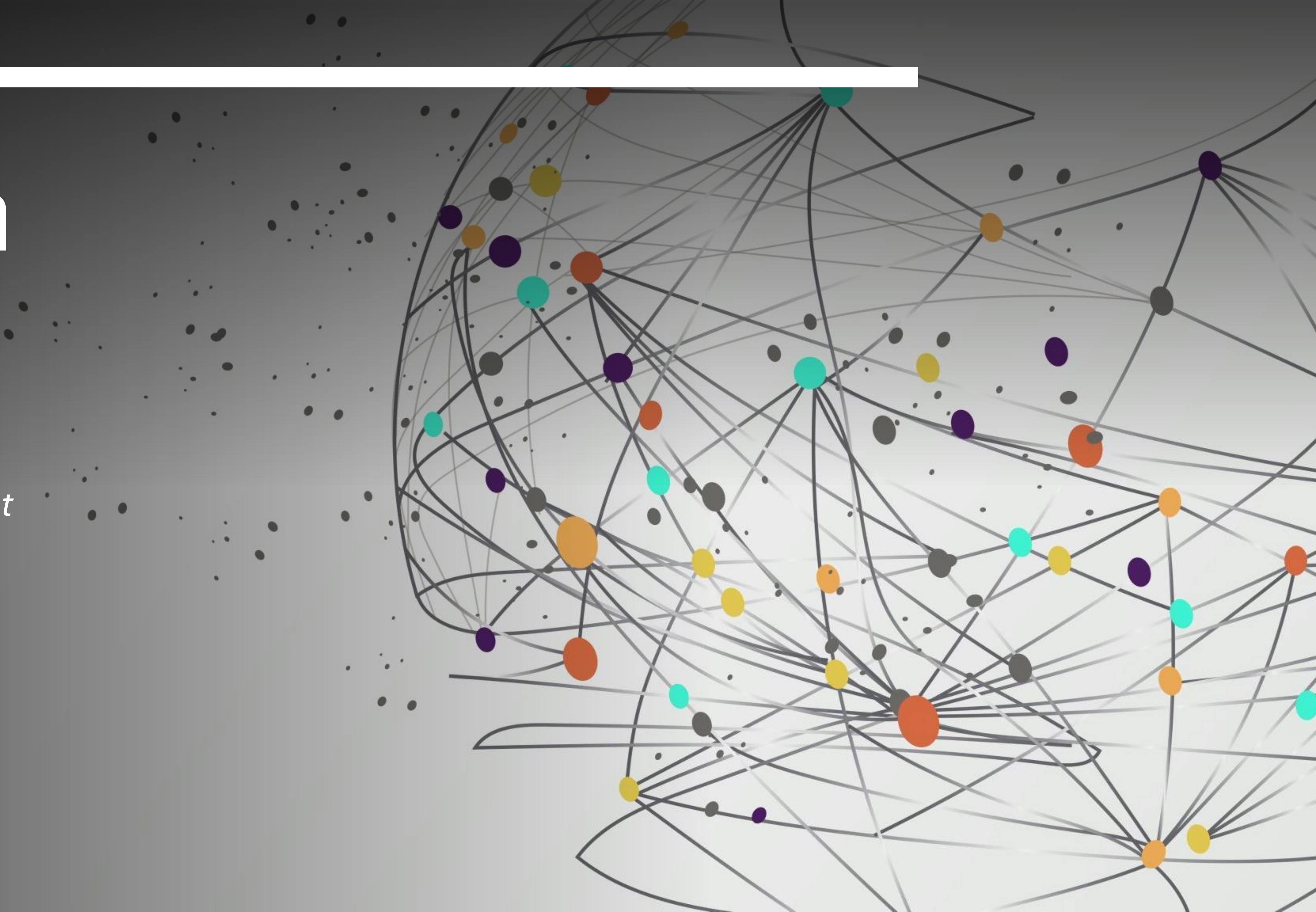
Java

Er. Piyush Pant



Installing Required Software

IntelliJ Idea

- Download community edition from this link : https://www.jetbrains.com/idea/download/?section=windows

Using VS code

Follow following tutorial

- https://www.youtube.com/watch?v=VUcI3Y1Nnak

Install git

Git

GitHub

Go to the following git hub link and pull the repo to your computer . [epiyushpant/MRCJava]

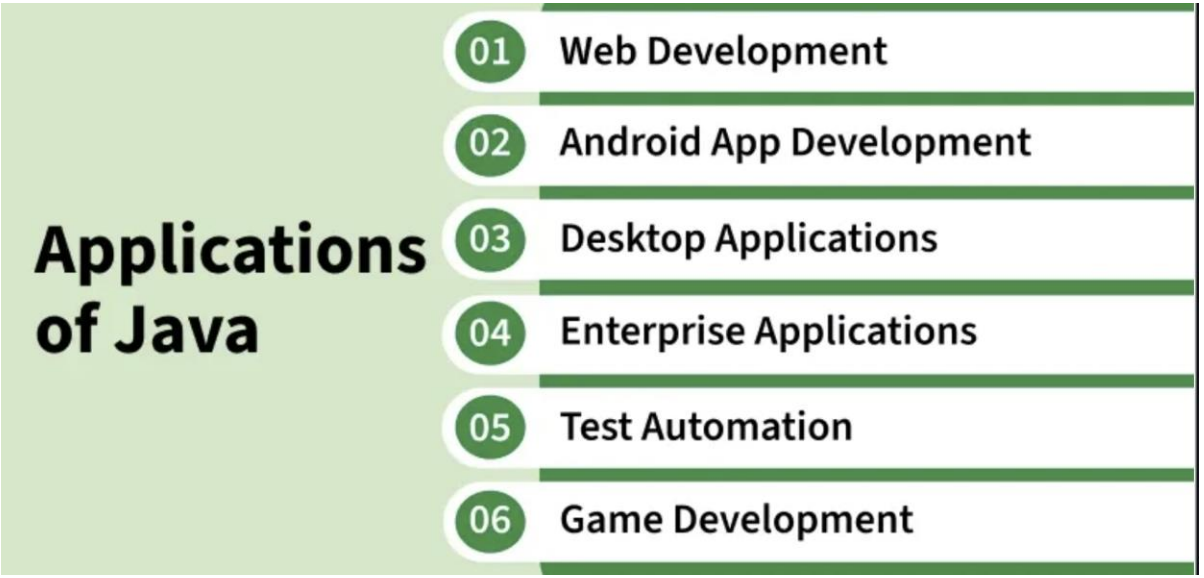
Follow the instruction provided by teacher or check the readme file in repo.

Note : always push your code in your branch.

Java

- Java is a high-level, object-oriented programming language used to build web apps, mobile applications, and enterprise software systems.
- Known for its Write Once, Run Anywhere capability, which means code written in Java can run on any device that supports the Java Virtual Machine (JVM).
- Syntax and structure is similar to C-based languages like C++ and C#.

Introduction Link



Applications of Java

01 Web Development

02 Android App Development

03 Desktop Applications

04 Enterprise Applications

05 Test Automation

06 Game Development

History of Java

- Developed by James Gosling and his team at sun Microsystems in early 1990s .
- Initially goal was to build a platform independent language for embeded devices such as set-top-box and televisions.
- C++ was not entirely platform independent language . (Could built on windows can't run on linux). SO, there was need of exploration in new language, which invented java.
- Firstly, named **Green** (extension .gt)
- Renamed to **Oak** inspired by Oak tree outside gosling office.
- Since, Oak name was already patient by another company , Oak Technologies , again renaming was needed.
- Then, finally marketing team of sun technologies renamed it to Java (after the Indonesian coffee bean) .

Java History

1991 – The Beginning

- James Gosling and his team at Sun Microsystems start “The Green Project,” creating a new language for embedded systems — the foundation of Java.

1995 – Java 1.0 Launched

- Official release with the slogan **“Write Once, Run Anywhere.”**
  JVM made cross-platform development possible.

1997 – Java Becomes a Standard

Recognized by **ISO/ANSI**, boosting reliability and global adoption.

1999 – Java 2 Platform Introduced

Java split into three editions:

- **J2SE** – Standard Edition
- **J2EE** – Enterprise Edition
- **J2ME** – Mobile/Embedded

2004 – Java 5 Released

- Major enhancements: **Generics, enhanced for-loop, annotations**.

2006 – Java Becomes Open Source

- Sun releases **OpenJDK**, inviting global community contributions.

2010 – Oracle Acquires Sun

- Oracle takes over Java, continues open-source development.

2011 – Java 7

- Project Coin features: **try-with-resources, multi-catch**, cleaner switch.

2014 – Java 8

- A milestone release introducing **Lambdas, Stream API, functional interfaces**.

2017 – Java 9

- Introduces the **Module System (Project Jigsaw)** for modular apps.

2018 → Present – New Release Cycle

6-month release model with LTS versions:

- **Java 11 (2018)** – LTS
- **Java 17 (2021)** – LTS
- **Java 25 (2025)** – Latest LTS

Philosophy of Java

Java was designed with a set of guiding principles to make software development simpler, safer, and platform-independent. Its core philosophy includes:

- **Simple**
  Easy to learn and write, with a clean and readable syntax.
- **Object-Oriented**
  Everything is treated as an object, making code modular, reusable, and maintainable.
- **Platform Independent**
  “Write Once, Run Anywhere” through the JVM.
- **Secure**
  Strong security features like bytecode verification and a restricted runtime environment.
- **Robust**
  Emphasis on error checking, memory management, and exception handling.
- **Portable**
  Programs behave consistently across different operating systems and hardware.
- **High Performance**
  Uses Just-In-Time (JIT) compiler to execute code efficiently.
- **Multithreaded**
  Built-in support for concurrent programming.
- **Dynamic & Extensible**
  Can load classes at runtime and adapt to new libraries easily.

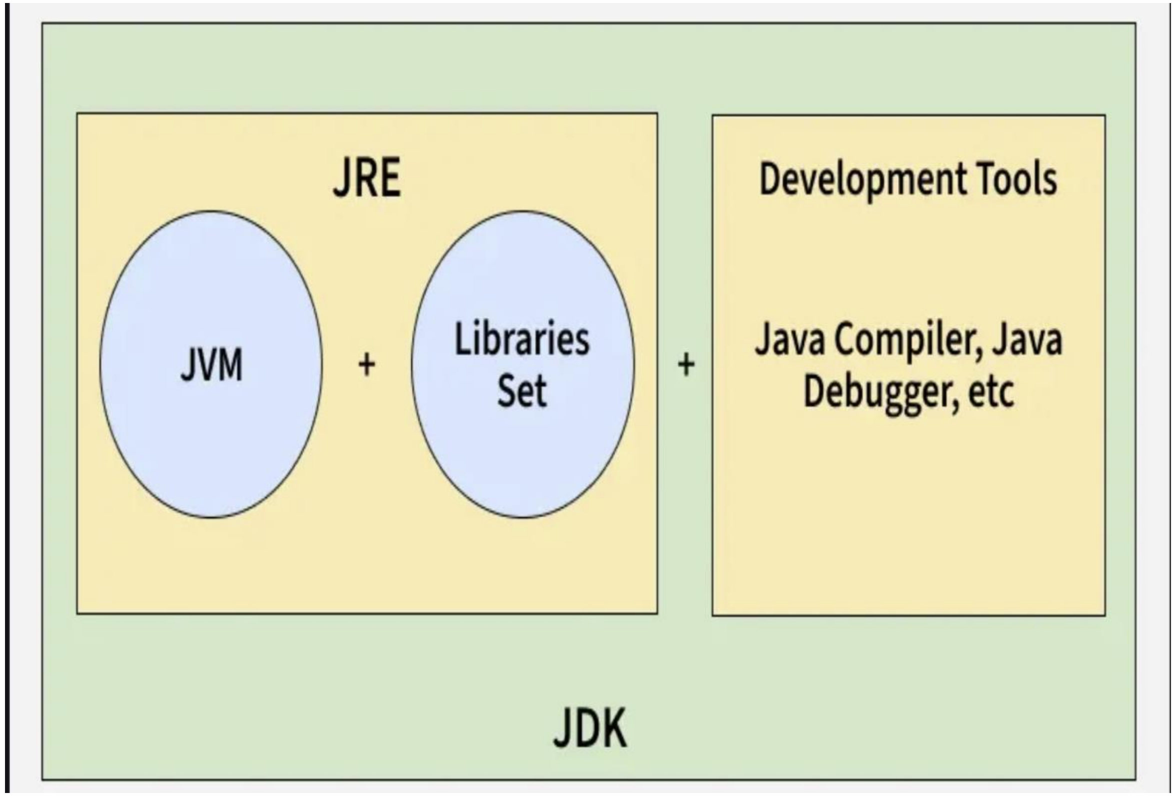
JDK, JVM and JRE

JDK (Java Development Kit) is a software development kit used to build Java applications. It contains the JRE and a set of development tools. JDK is the complete package that allows you to. **develop**, **compile**, and **run** Java programs.

- Includes compiler (javac), debugger (jdb), and utilities like jar (build tools) and javadoc.
- Required by developers to write, compile, and debug code.
- Includes JRE (which includes JVM)

Working of JDK

- Source Code (.java): Developer writes a Java program.
- Compilation: The JDK’s compiler (javac) converts the code into bytecode stored in .class files.
- Execution: The JVM executes the bytecode, translating it into native instructions.



Java Runtime Environment

JRE provides an environment to run Java programs but does not include development tools. It is intended for end-users who only need to execute applications.

- Contains the JVM and standard class libraries.
- Provides all runtime requirements for Java applications.
- Does not support compilation or debugging.

Working of JRE:

- Class Loading: Loads compiled .class files into memory.
- Bytecode Verification: Ensures security and validity of bytecode.
- Execution: Uses the JVM (interpreter + JIT compiler) to execute instructions and make system calls.

Java Virtual machine (JVM)

JVM is the core execution engine of Java. It is responsible for converting bytecode into machine-specific instructions.

- Part of both JDK and JRE.
- Performs memory management and garbage collection.
- Provides portability by executing the same bytecode on different platforms.
- Handles **runtime tasks** like loading classes, verifying code, and managing threads

Working of JVM

- JVM implementations are platform-dependent.
- Bytecode is platform-independent and can run on any JVM.
- Modern JVMs rely heavily on Just-In-Time (JIT) compilation for performance.

- Details

JVM
Runs Java bytecode
Part of JRE

JRE
Runs Java applications
JVM + Libraries

JDK
Develops + runs Java programs
JRE + development tools

Feature / Component	**JVM**	**JRE**	**JDK**
Full Form	Java Virtual Machine	Java Runtime Environment	Java Development Kit
Main Purpose	Runs Java bytecode	Provides environment to run Java programs	Provides tools to develop and run Java programs
Contains JVM	✓ (itself)	✓	✓
Contains Core Libraries	✗	✓	✓
Includes Development Tools	✗	✗	✓ (javac, javadoc, jdb, jar, etc.)
Can Run Java Programs	✓	✓	✓
Can Compile Java Programs	✗	✗	✓
Includes JRE	✗	✗	✓
Target Users	Runtime system	End users who only want to run Java apps	Developers who write Java code
Example Tools Included	None	JVM + libraries	Compiler (javac), debugger (jdb), archiver (jar), etc.

Object Oriented Programming

Object-Oriented Programming (OOP) is a programming paradigm in which a software system is organized around **objects** rather than functions or logic. An object is a self-contained entity that contains **data (attributes)** and **methods (functions)** that operate on the data. OOP emphasizes **encapsulation, inheritance, polymorphism, and abstraction** to improve code modularity, reusability, and maintainability.

Problems with Structural Programming like : C

- Code becomes **large and hard to manage** as the program grows.
- **Reusing code** is difficult; functions often need to be rewritten.
- Hard to **model real-world entities** like Student, Car, or Bank Account.
- **Maintenance is tricky**; changes in one part may break other parts.
- **Data is not secure**; global variables can be accessed and changed anywhere.
- Difficult to **scale programs** for large projects or teams.
- **Adding new features** often requires rewriting existing code.

Problems that OOP Solves

Problem	How OOP Solves It
Programs become too big and confusing	OOP divides the program into **objects**, each handling its own data and actions.
Rewriting the same code again and again	OOP uses **inheritance** so we can reuse existing code.
Making changes breaks the program	**Encapsulation** keeps data safe inside objects, so changes don’t affect everything else.
Hard to model real-life things	OOP uses **classes and objects** to represent real-world entities like Student, Car, or Bank Account.
Adding new features is difficult	**Polymorphism** allows new features without changing existing code.
Data is not safe	**Data hiding** protects information inside objects, giving access only through safe methods.

Java Hello World

```
public class Main {
 public static void main(String[] args) {
 System.out.println("Hello World");
 }
}
```

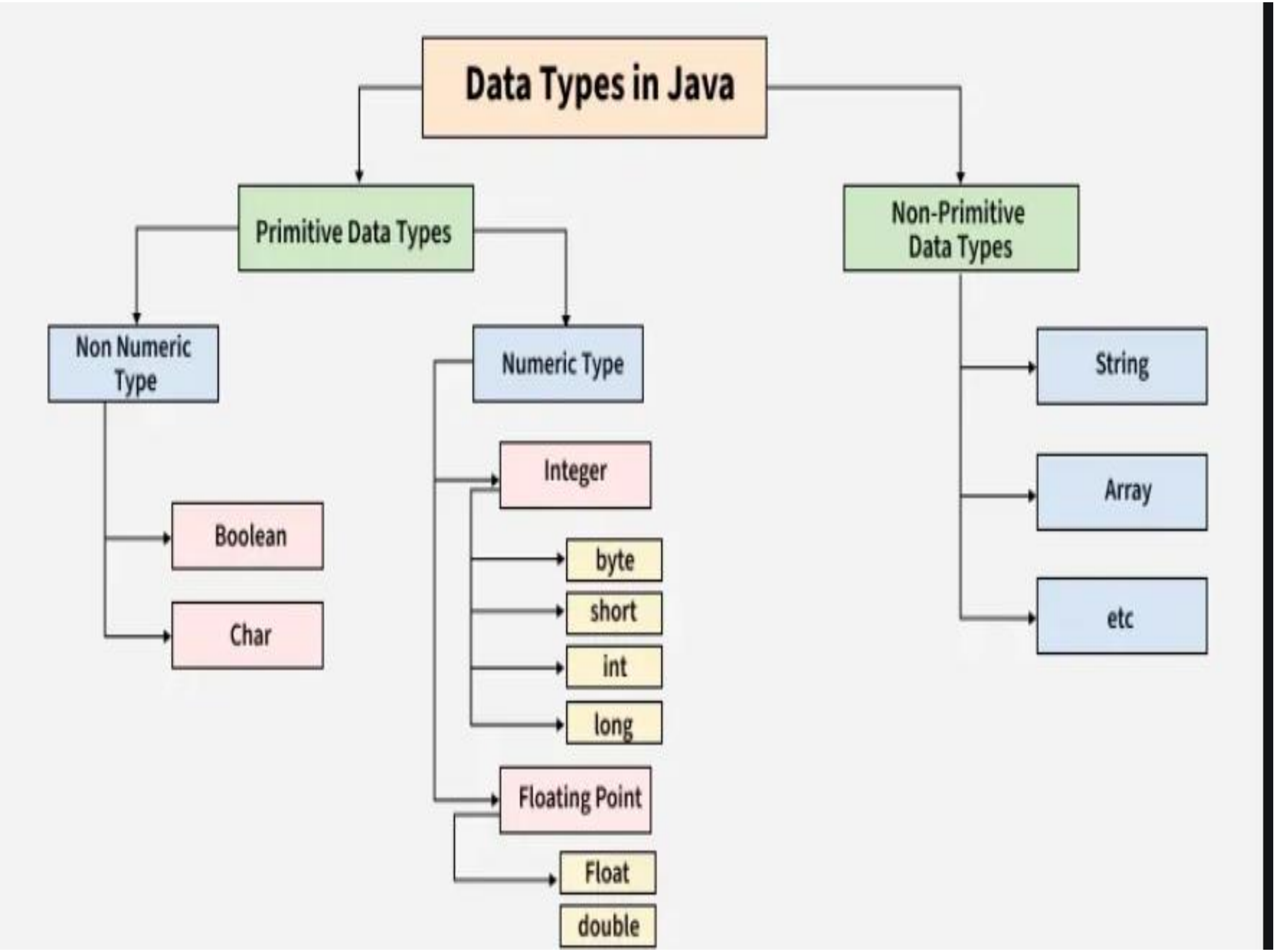
- Every line of code that runs in Java must be inside a class.
- The class name should always start with an uppercase first letter. In our example, we named the class Main.
- Note: Java is case-sensitive. MyClass and myclass would be treated as two completely different names.
- The name of the Java file must match the class name. So if your class is called Main, the file must be saved as Main.java.
- This is because Java uses the class name to find and run your code. If the names don't match, Java will give an error and the program will not run.

Data Types

Data types in Java define the kind of data a variable can hold and the memory required to store it.

Primitive Data Types: Store simple values directly in memory.

Non-Primitive (Reference) Data Types: Store memory references to objects.



Primitive Data Types

TYPE	DESCRIPTION	DEFAULT	SIZE	EXAMPLE	RANGE
boolean	Logical values	false	JVM-dependent (typically 1 byte)	true, false	—
byte	8-bit signed integer	0	**1 byte**	**10**	**-128 to 127**
char	16-bit Unicode character	\u0000	**2 bytes**	'A', '\u0041'	0 to 65,535
short	16-bit signed integer	0	**2 bytes**	**2000**	**-32,768 to 32,767**
int	32-bit signed integer	0	**4 bytes**	1000, -500	**-2,147,483,648 to 2,147,483,647**
long	64-bit signed integer	0L	**8 bytes**	123456789L	**±9.22e18** **-2^63 to 2^63 –1**
-float	32-bit floating point	0.0f	**4 bytes**	3.14f	**~6–7 digits precision**
double	64-bit floating point	0.0d	**8 bytes**	3.14159d	**~15–16 digits precision**

Data Types

Data Type	Description
byte	Stores whole numbers from -128 to 127
short	Stores whole numbers from -32,768 to 32,767
int	Stores whole numbers from -2,147,483,648 to 2,147,483,647
long	Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
float	Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits
double	Stores fractional numbers. Sufficient for storing 15 to 16 decimal digits
boolean	Stores true or false values
char	Stores a single character/letter or ASCII values

Non-Primitive Data Type

Non-primitive data types are created by the programmer or provided by Java libraries, and they store references (addresses) instead of actual values.

Examples : String , Arrays . Classes , Objects, Interfaces, Enums.

Note:

- Primitive types in Java are predefined and built into the language, while non-primitive types are created by the programmer (except for `String`).
- Primitive types start with a lowercase letter (like `int`), while non-primitive types typically starts with an uppercase letter (like `String`).
- Primitive types always hold a value, whereas non-primitive types can be `null`.

Java Type Casting

Type casting means converting one data type into another. For example, turning an `int` into a `double`.

In Java, there are two main types of casting:

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

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

Widening Casting

```
int myInt = 9;
double myDouble = myInt; // Automatic casting: int to double

System.out.println(myInt); // Outputs 9
System.out.println(myDouble); // Outputs 9.0
```

Narrow Casting

```
double myDouble = 9.78d;

int myInt = (int) myDouble; // Manual casting: double to int


System.out.println(myDouble); // Outputs 9.78

System.out.println(myInt);    // Outputs 9
```

Java Operators

- Arithmetic Operators
- Assignment Operators
- Comparison Operators
- Logical Operators
- Java Operator Precedence

Arthmetic Operators

Operator	Name	Description	Example
+	Addition	Adds together two values	x + y
-	Subtraction	Subtracts one value from another	x - y
*	Multiplication	Multiplies two values	x * y
/	Division	Divides one value by another	x / y
%	Modulus	Returns the division remainder	x % y
++	Increment	Increases the value of a variable by 1	++x
--	Decrement	Decreases the value of a variable by 1	--x

Assignment Operators

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
&=	x &= 3	x = x & 3
|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Operator	Name	Example
==	Equal to	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

Operator	Name	Description	Example
&&	Logical and	Returns true if both statements are true	x < 5 && x < 10
||	Logical or	Returns true if one of the statements is true	x < 5 || x < 4
!	Logical not	Reverse the result, returns false if the result is true	!(x < 5 && x < 10)

Precedence

When there are more than one operation java follows following order .

( ) - Parentheses

*, /, % - Multiplication, Division, Modulus

+, - - Addition, Subtraction

>, <, >=, <= - Comparison

==, != - Equality

&& - Logical AND

|| - Logical OR

= - Assignment

Java Literals

In Java, a Literal is a value of boolean, numeric, character, or string data. Any constant value that can be assigned to the variable is called a literal.

// Here 100 is a constant/literal.
int x = 100;

Integral Literals in Java

For Integral data types (byte, short, int, long), we can specify literals in four ways, which are listed below:

1.1 Decimal literals (Base 10): In this form, the allowed digits are 0-9.

int x = 101;

1.2 Octal literals (Base 8): In this form, the allowed digits are 0-7.

// The octal number should be prefix with 0.

int x = 0146;

1.3 Hexadecimal 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;

1.4. Binary literals: From 1.7 onward, we can specify literal value even in binary form also, allowed digits are 0 and 1. Literals value should be prefixed with 0b or 0B.

int x = 0b1111;

Note: *By default, every integral literal is of int type. To specify it as long, add the suffix L or l. There's no explicit way to define byte or short literals, but if an integral value assigned is within their range, the compiler treats it automatically as a byte or short literal.*

Refer code also

Java Literals

2. Floating-Point Literal in Java

For Floating-point data types, we can specify literals in only decimal form, and we cannot specify in octal and Hexadecimal forms.

2.1 Decimal literals(Base 10): In this form, the allowed digits are 0-9.

double d = 123.456;

Note: *By default, floating-point literals are of double type. To assign them to a float, use the suffix f or F. You may optionally use d or D for double. Hexadecimal floating-point literals are not supported in Java.*

Refer code also

Java Literals

3. Char Literals in Java

For char data types, we can specify literals in four ways which are listed below:

1 Single quote: We can specify literal to a char data type as a single character within the single quote.

char ch = 'a';

2. Char literal as Integral literal: we can specify char literal as integral literal, which represents the Unicode value of the character, and that integral literal can be specified either in Decimal, Octal, and Hexadecimal forms. But the allowed range is 0 to 65535.

```
  char ch1 = 65;    // Decimal → 'A'
char ch2 = 0101;  // Octal → 'A'
char ch3 = 0x41;  // Hexadecimal → 'A'
char ch4 = 062;   // Octal → Unicode 50 → '2'
```

3. Unicode Representation: We can specify char literals in Unicode representation '\uxxxx'. Here xxxx represents 4 hexadecimal numbers.

char ch = '\u0061';// Here \u0061 represent a.

4. Escape Sequence: Every escape character can be specified as char literals.

```
char newline = '\n';  // Newline
char tab     = '\t';  // Tab
char quote   = '\"';  // Double quote
char backslash = '\\'; // Backslash
```

Java Literals

4. String Literals in Java

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

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

String literals may not contain unescaped newline or linefeed characters. However, the Java compiler will evaluate compile-time expressions, so the following String expression results in a string with three lines of text.

```
String text = "This is a String literal\n"

+ "which spans not one and not two\n"

+ "but three lines of text.\n";
```

5. Boolean Literals in Java

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

```
boolean b = true;

boolean c = false;
```

Java Identifier

- All Java **variables** must be **identified** with **unique names**.
- These unique names are called **identifiers**.
- Identifiers can be short names (like x and y) or more descriptive names (age, sum, totalVolume).

Note: It is recommended to use descriptive names in order to create understandable and maintainable code

- `// Good`
- `int minutesPerHour = 60;`
-
- `// OK, but not so easy to understand what m actually is`
- `int m = 60;`
-

RULES

Names can contain letters, digits, underscores, and dollar signs
Names must begin with a letter
Names should start with a lowercase letter, and cannot contain whitespace
Names can also begin with $ and _
Names are case-sensitive ("myVar" and "myvar" are different variables)
Reserved words (like Java keywords, such as `int` or `boolean`) cannot be used as names

Invalid Ones

```
int 2ndNumber = 5;   // Cannot start with a digit
int my var = 10;     // Cannot contain spaces
int int = 20;        // Cannot use reserved keywords
```

Variable and Constants in Java

Variable

Variables are containers for storing data values. It can change , we can update variables.

```
type variableName = value;
```

```
String name = "John";
```

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

Constant

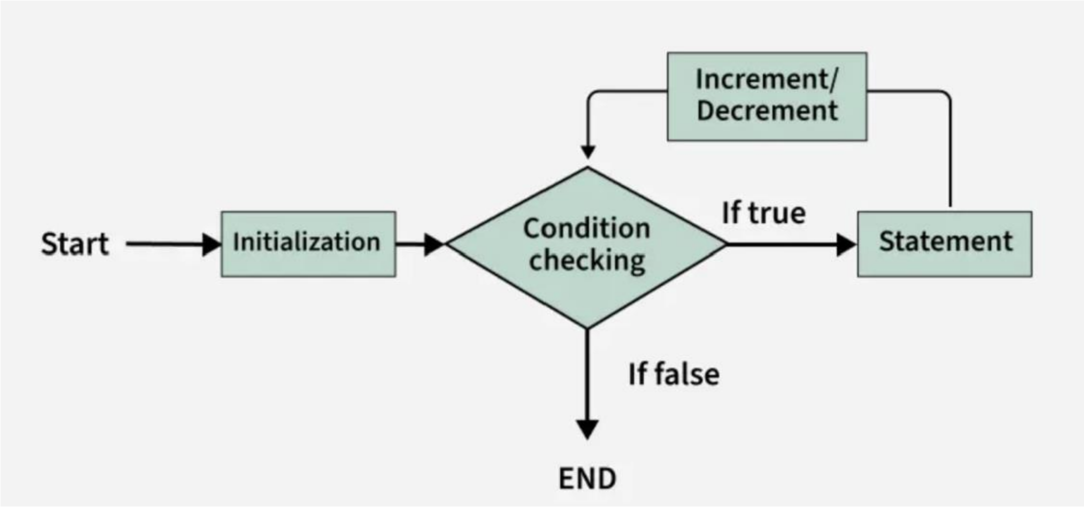
Constant cannot change after declaration . USed for storin values that are not changed frequently .

```
final int MINUTES_PER_HOUR = 60;
```

```
final int BIRTHYEAR = 1980;
```

Loops Java

For Loops



Code

```
for (int i = 0; i <= 10; i++) {
        System.out.print(i + " ");
     }
```

- Use for when you know **exactly how many times** the loop should run.

Example:

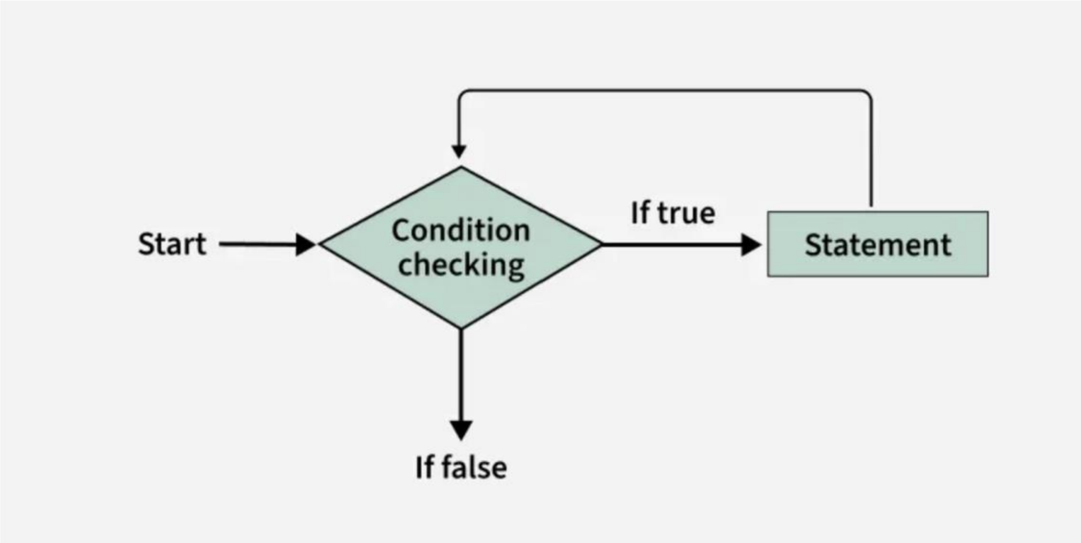
- Printing 1 to 100
- Looping through an array
- Executing code a fixed number of times (like 5 attempts)

Loops Java

Code

```
Scanner sc = new Scanner(System.in);
String password = "";
while (!password.equals("admin123")) {
   System.out.print("Enter password: ");
   password = sc.nextLine();
}
System.out.println("Login successful!");
```

While Loop



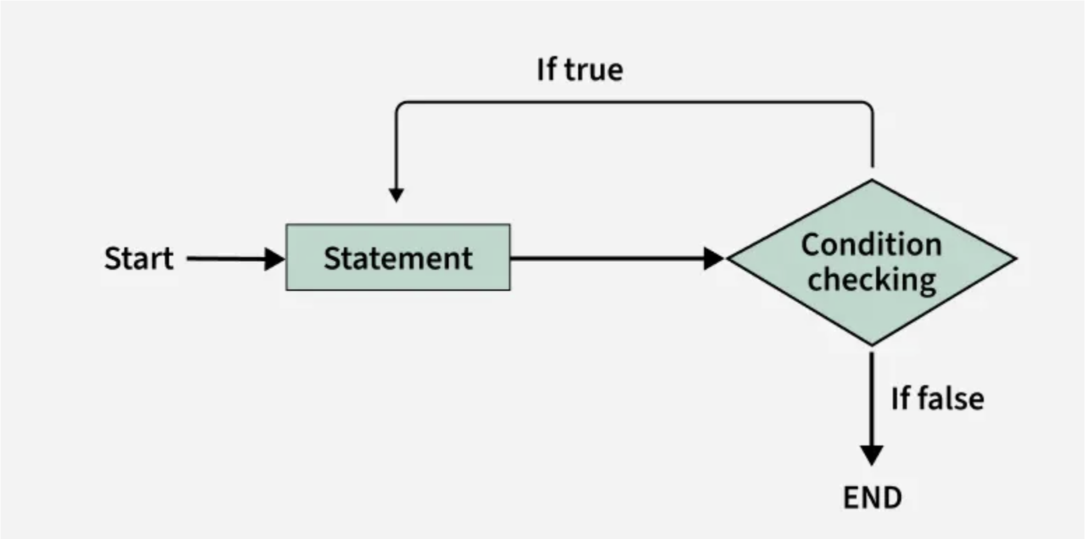
Use `while` when you **don't know** the number of iterations **in advance**, but you continue until a condition becomes false.

Example:

- Reading user input until they type "exit"
- Running until file ends
- Keep checking internet connection until it becomes available

Loops Java

While Loop



Code

```
//Example display menu at least once
Scanner sc = new Scanner(System.in);
int choice;
do {
   System.out.println("1. Start Game");
   System.out.println("2. Settings");
   System.out.println("3. Exit");
   System.out.print("Choose an option: ");
   choice = sc.nextInt();
} while (choice != 3);
System.out.println("Program Closed.");
```

Use `do-while` when you need the loop to run **at least once**, even if the condition is false.

Example:

- Display menu at least once
- Ask user to enter number at least once
- Retry operation at least one time