## What is Java?

What is Java?

General-purpose, class-based, object-oriented language. Aims for minimal implementation dependencies. "Write once, run anywhere" concept means code can run on diverse platforms without recompilation. Created by Sun Microsystems in 1991, now owned by Oracle. Known for simplicity, modularity, and reusability. Widely used, powering over 3 billion devices as per Oracle.

Java Features:

1. **Platform Independence:** Java code runs anywhere with a JVM.

Allows broad device and platform compatibility.

1. **Object-Oriented:** Organizes code around objects.

Encourages reusable, scalable, and clean code.

1. **Simple:** Easy to learn and use with a clear syntax.

Reduces complexity for faster development.

1. **Robust:** Strong memory management, exception handling.

Enhances reliability, reducing crashes or errors.

1. **Secure:** Built-in security features like bytecode verification.

Safeguards against unauthorized access and attacks.

1. **Multithreading:** Allows concurrent execution of tasks.

Enhances performance, especially in handling multiple operations.

Advantages

1. Platform independence: Code runs on various platforms without modification.
2. Robustness: Strong memory management, exception handling, and type checking.
3. Object-oriented: Encourages modular and reusable code.
4. Huge ecosystem: Rich libraries, frameworks, and tools available.
5. Security: Built-in features for secure execution.

Disadvantages

1. Performance: Slower execution compared to languages like C or C++.
2. Verbose syntax: Requires more code for simple tasks compared to some languages.
3. Garbage collection: Automatic memory management can impact performance unpredictably.
4. Limited low-level access: Less control over hardware compared to some languages.
5. Learning curve: Complexity in learning due to its vast ecosystem and concepts like threading.

Java Components

1. **Java Code (.java):** Your written code saved in .java file(s).
2. **Javac Compiler:** Translates Java code into bytecode saved in a class file.
3. **Bytecode:** Produced by the compiler, ready for execution.
4. **Java Virtual Machine (JVM):** Executes bytecode, ensuring platform independence.

Program Execution Phases

1. **Writing:** Java programmers create the code.
2. **Compilation:** Javac compiles code into bytecode.
3. **Execution: JVM runs the bytecode.**

Java Development Kit (JDK)

1. Includes JRE, compilers, and tools like JavaDoc and debugger.
2. Needed for creating, compiling, and running Java programs.

Java Runtime Environment (JRE)

1. Allows running Java programs, containing JVM and plugins.
2. For running compiled Java programs only; cannot compile.
3. A diagram of a computer code

   Description automatically generatedRequires JDK for compiling Java programs.

## Java Programming Structure

Package Declaration: Defines the directory/module for the Java class.

Uses "**package**" keyword.

Import Statements: Brings in classes from other directories/packages.

Uses "**import**" keyword.

Comments: Single line (//) or multiline (/\* \*/) for explanations.

Provides information about code elements.

Class Definition: Names the Java class, essential for object creation.

Uses the "**class**" keyword.

Main Method: Starting point for Java program execution.

Identified by "**public static void main(String[] args)**".

Methods/Behaviors: Set of instructions forming specific functionalities.

**Encapsulates code** to avoid repetition.

Can accept variable values for execution.

## Variables

A variable represents a memory location in a program. It stores values that can change during program execution. All operations on a variable affect that specific memory location.

A diagram of a memory

Description automatically generatedIn Java, **variables must be declared** before use.

Declaration

**type name;** // declaration

Assigning values to variables:

1. Variable Initialization
2. Assigning value by taking input

**float simpleInterest;** // Declaring a float variable

**int myAge = 19;** // Declaring and initializing an integer variable

**char firstLetter = 'h';** // declare and initializing a char variable

Types of Variables

Local Variables

Defined within a block, method, or constructor. Created when the block is entered and destroyed upon exiting. Scope is limited to the block where declared.

Instance Variables

Non-static variables declared in a class. Created with object creation, destroyed when the object is. May have access specifiers.

* Each object has its copy.
* Changes made in one object do not affect others.
* Accessed through object references.

Static Variables

Also called Class variables. Declared with the static keyword outside methods.

Only one copy per class, exist throughout program execution.

* Single copy per class, shared by all objects.
* Changes made are reflected across all objects.
* Accessed using the class name.

Rules for variable Names in Java

* Start with a letter, **$**, or **\_**.
* Followed by any combination of characters.
* Avoid using keywords.
* Case-sensitive.
* Legal: age, $salary, \_value, \_\_1\_value.
* Illegal: 123abc, -salary.

public class HelloWorld {  
 public String myVar = "Instance variable"; // instance variable  
 public static String *myClassVar* = "class or static variable"; // static variable  
  
 public void printHelloWorld() {  
 String myVar = "Variable inside Method"; // local variable  
 }  
 public static void main(String[] args) {  
 printHelloWorld();  
 }  
}

## Data Types

Data types define the values a variable can hold. In Java, they're statically typed, known at compile time.

**A diagram of data types

Description automatically generated**

Two Categories of Data Types:

1. **A screenshot of a black and white list

   Description automatically generatedPrimitive Data Types:** Specify size and type of variable values, no additional methods.
2. **Non-Primitive Data Types:** (Arrays and Strings)

* Represented by instances like objects (reference variables).
* Mainly classes, arrays, strings, or interfaces.

## Operators

**Operators** in Java represent actions, like arithmetic or logical operations.

Arithmetic Operators: Perform mathematical operations.

A screenshot of a computer

Description automatically generated

Relational Operators: Compares values.

A screenshot of a computer

Description automatically generated

A black and white list with white text

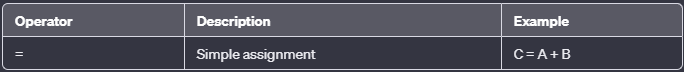
Description automatically generatedBitwise Operators: Perform bit-level operations.

Logical Operators: Operate on Boolean values.

A black and white text box

Description automatically generated with medium confidence

Assignment Operator: Assigns values.



## Classes and Objects in Java

**Class**: A template or blueprint describing the behavior and state that objects of its type support.

**Creation**: Declared using the **class** keyword.

Components of a class

**Constructor**: Method called when creating an object.

* Default constructor provided if none explicitly defined.
* Same name as the class.
* Can have multiple constructors.

**Methods**: Functions describing object behavior. Accesses and manipulates object values.

**Variables/Properties**: Attributes describing objects, hold object characteristics.

**public class Person {**

// Variables

**String name;**

**int age;**

**double height;**

// Constructor

**public Person() {**

// Constructor logic here

**}**

// Methods

**public void run() {**

// Method logic here

**}**

**}**

Object

* Instances of a class.
* Have states/attributes and behaviors/methods.

**Person personObject = new Person();**

## Object Oriented Programming

Object-Oriented Programming (OOP)

Object-oriented programming (OOP) is popular for its **modular**, **reusable** **approach**, organizing code into manageable entities (objects) that encapsulate data and methods.

**Concept:** Based on "objects" containing data and methods.

**Purpose:** Enhances program flexibility and maintainability.

**Integration:** Unites data and behaviour within objects for clarity.

**Java's Role:** Mature language embracing OOP principles.

Object Features

**Self-Access:** Objects access and modify their own data fields ("this" or "self").

Program Design: Built by interconnecting interacting objects.

OOP Languages

**Diversity:** Wide array including Java, C++, C#, Python, and more.

**Class-Based:** Most popular OOP languages follow a class-based structure.

**Key Languages:** Java, C++, C#, Python, R, PHP, JavaScript, and others embrace object-oriented principles in diverse ways.

## Features of OOPs

A diagram of a pie chart

Description automatically generatedA diagram of a diagram

Description automatically generated

Abstraction

Hides implementation details, focuses on functionality.

Achieved in Java via **Abstract classes** and **interfaces**.

Encapsulation

Bundles data (variables) and methods as a single unit.

Variables hidden (private) and accessed via public methods (getters/setters).

Inheritance

Enables one class to **acquire** properties of **another**.

Subclass inherits from a superclass using the extends keyword.

**class Super {**

**.....**

**.....**

**}**

**class Sub extends Super {**

**.....**

**.....**

**}**

**Polymorphism**

Processes objects differently based on their data types.

Achieved through multiple implementations of a generic interface.

Implemented as method **overloading** (compile-time) and **overriding** (runtime).

// Method overloading

**class Calculator {**

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

**return a + b;**

**}**

**double add(double a, double b) {**

**return a + b;**

**}**

**}**

// Method overriding

**class Parent {**

**void display() {**

**System.out.println("Parent's Display");**

**}**

**}**

**class Child extends Parent {**

**void display() {**

**System.out.println("Child's Display");**

**}**

**}**

Coupling

Coupling represents the degree of dependency between classes.

* Strong Coupling: Exists when classes are highly aware of each other's details.

**Java Usage:** Modifiers like private, protected, public limit visibility. Interfaces facilitate weaker coupling by avoiding concrete implementations.

Cohesion

Cohesion measures how well a component focuses on a single task.

* **High Cohesion:** Methods that perform a single, well-defined task.
* **Weak Cohesion:** Methods that split tasks into unrelated parts.

Example: java.io package (highly cohesive) deals with I/O-related classes, while java.util package (weakly cohesive) contains unrelated classes.

Association

Describes relationships between objects.

* One to One
* One to Many
* Many to One
* Many to Many

Real-life Examples: A country having one president (one to one), ministers serving a president (one to many), etc.

Direction: Can be unidirectional or bidirectional.

Aggregation

A form of association that creates a one-way relationship between classes.

Example: Passengers having a car (Passenger HAS-A Car),

BUT a **Car doesn’t necessarily** have a Passenger.

Java Usage: Represents a **HAS-A** relationship between classes, one class depends on the other.

Composition

A stricter form of aggregation where two classes are mutually dependent and can't exist without each other.

Example: Car and Engine classes, where a Car cannot function without an Engine and vice versa.

Java Usage: **PART-OF** relationship between classes, both classes are interdependent; if one ceases to exist, the other can't survive alone.

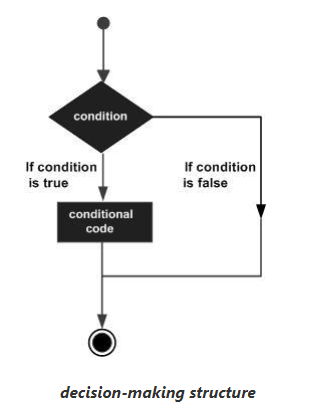
## Decision Control

Control Flow statements

Control flow statements in programming allow us to execute specific sets of code based on certain conditions. For instance, if we want to print "Positive Number" when a value is greater than zero and "Negative Number" when it's less than zero, control flow statements help determine which print statement runs based on the input.

Decision-making structures

Decision-making structures in programming involve evaluating one or more conditions. If a condition is found to be true, specific statements associated with that condition are executed. Optionally, when the condition is false, other predefined statements can be executed instead. This way, control statements enable the execution of tailored code based on varying conditions within a program.

If Statement

Consists of Boolean expression followed by one or more statements.

**If (Boolean\_expression) { Statements; } // statement executes if Boolean is true.**

Else statement

An if statement can be followed by an optional else statement, which executes when the Boolean expression is false. The diagram below shows the flow of an if else statement.

**if (num < 50) {**

**System.out.println("num is less than 50”);**

**}**

**else {**

**Sout("num is greater than or equal 50")**

**}**

Nested if Statement

Combines if, else if, else to test multiple conditions. Executes the code block associated with the first condition that evaluates to true; subsequent conditions are not evaluated after one succeeds.

**if (x == 10) {**

// Code block for x equals 10

**} else if (x == 20) {**

// Code block for x equals 20

**} else if (x == 30) {**

// Code block for x equals 30

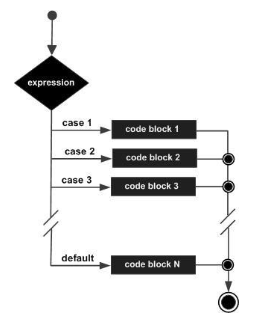
**} else {**

// Code block if none of the conditions are true

**}**

Switch Statement

Tests a variable against a list of values and executes code based on matching cases.



Rules:

* Used for integers, convertible integers (byte, short, char), strings, and enums.
* Cases followed by values to compare and a colon.
* When a match is found, statements execute until a **break** is encountered.
* Optional **default** case at the end, executed when no cases are true.

**switch (variable) {**

**case value1:**

// Code block for value1

**break;**

**case value2:**

// Code block for value2

**break;**

**default:**

// Code block for default

**}**

**Int num=2;**

**Switch(num+2) {**

**case 1:**

**sout(“value is: “ + num);**

**case 2:**

**sout(“value is: “ + num);**

**default:**

**sout(“value is: “ + num);**

**}**

Ternary Operator

Offers a compact way to handle two outcomes, akin to an IF-ELSE statement.

(expression1) ? expression2 : expression3;

If **expression1** is true, **expression2** becomes the result of the whole expression.

If **expression1** is false, **expression3** becomes the result of the whole expression.

## Java Identifiers

**Definition:** Identifiers are names for variables, methods, classes, packages, and interfaces in Java.

**Composition:** They comprise letters, numbers, underscore (\_), and dollar sign ($). They should start with a letter, underscore, or dollar sign.

Convention: Java uses CamelCase for identifiers, where compound words or phrases begin with a lowercase letter (except the first word) followed by capitalizing each new word.

Example: In the code snippet, HelloWorld, args, main, and println are identifiers.

Java Modifiers

Modifiers: Keywords altering the meaning of identifiers.

Types

* **Access Modifiers:** Regulate accessibility (public, protected, private).
* **Non-Access Modifiers:** Define properties or functionalities (final, static, abstract).
* **Usage:** Modifiers are placed before the class, method, or variable they modify.

Example: In code, modifiers appear as keywords preceding the identifier, such as public or static.