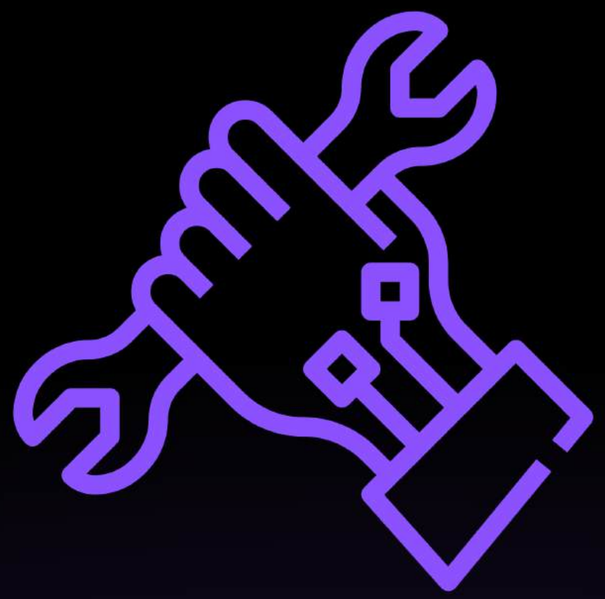
Java

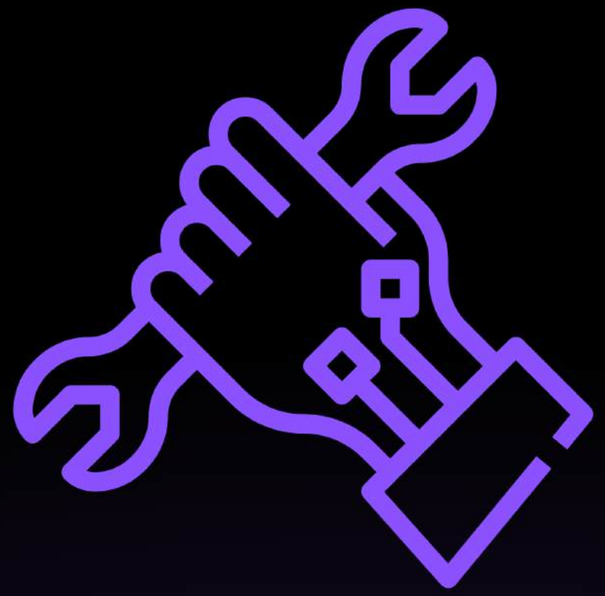
Java is an *object-oriented programming language.*

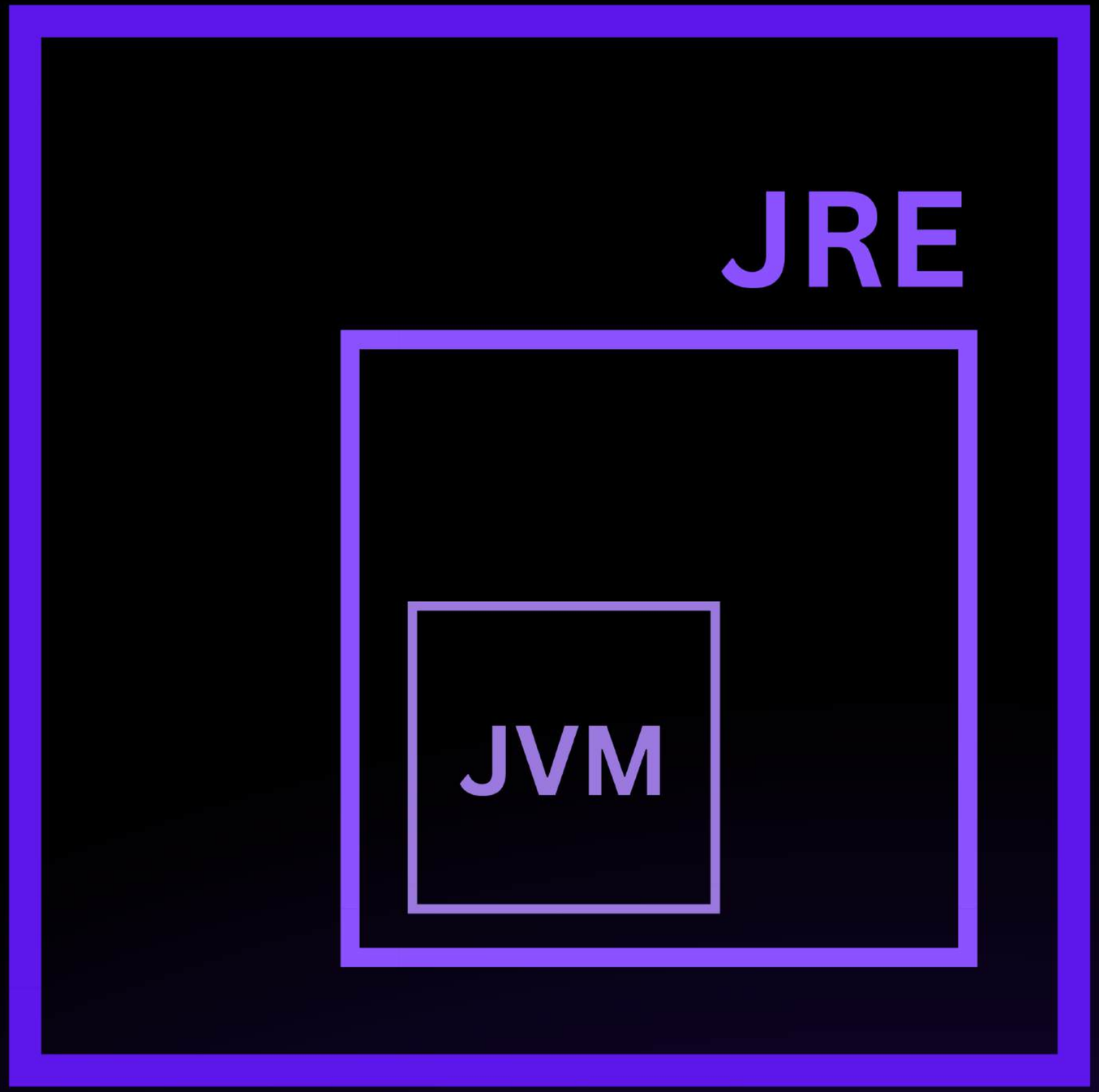
Java is known for its *"Write Once, Run Anywhere"* capability, meaning Java code can run on any platform that has a Java Virtual Machine (JVM) installed. This platform independence is one of Java's biggest strengths.

Install Java



Install Java = Install JDK





When you install JDK,
you are also installing JVM & JRE

JDK

The Java Development Kit is a comprehensive software development environment used for developing Java applications. It includes everything you need to develop Java programs, including:

1. The Java compiler (javac) to convert your Java source code (.java files) into bytecode (.class files)
2. Development tools like debugger, documentation generator (javadoc)
3. Source files for Java core classes
4. The JRE (which includes the JVM)

JRE

The Java Runtime Environment is what's needed to run Java applications on a computer. It consists of:

1. The Java Virtual Machine (JVM)
2. Compiled Core classes (like String, Integer, ArrayList, etc) and supporting files (Configuration files that tell Java how to run)
3. No development tools (unlike JDK) - it's meant for end users who just want to run Java applications
4. JRE (Java Runtime Environment) can be installed without JDK (Java Development Kit) since JRE is only needed to run Java applications, while JDK is needed to develop them.

JVM

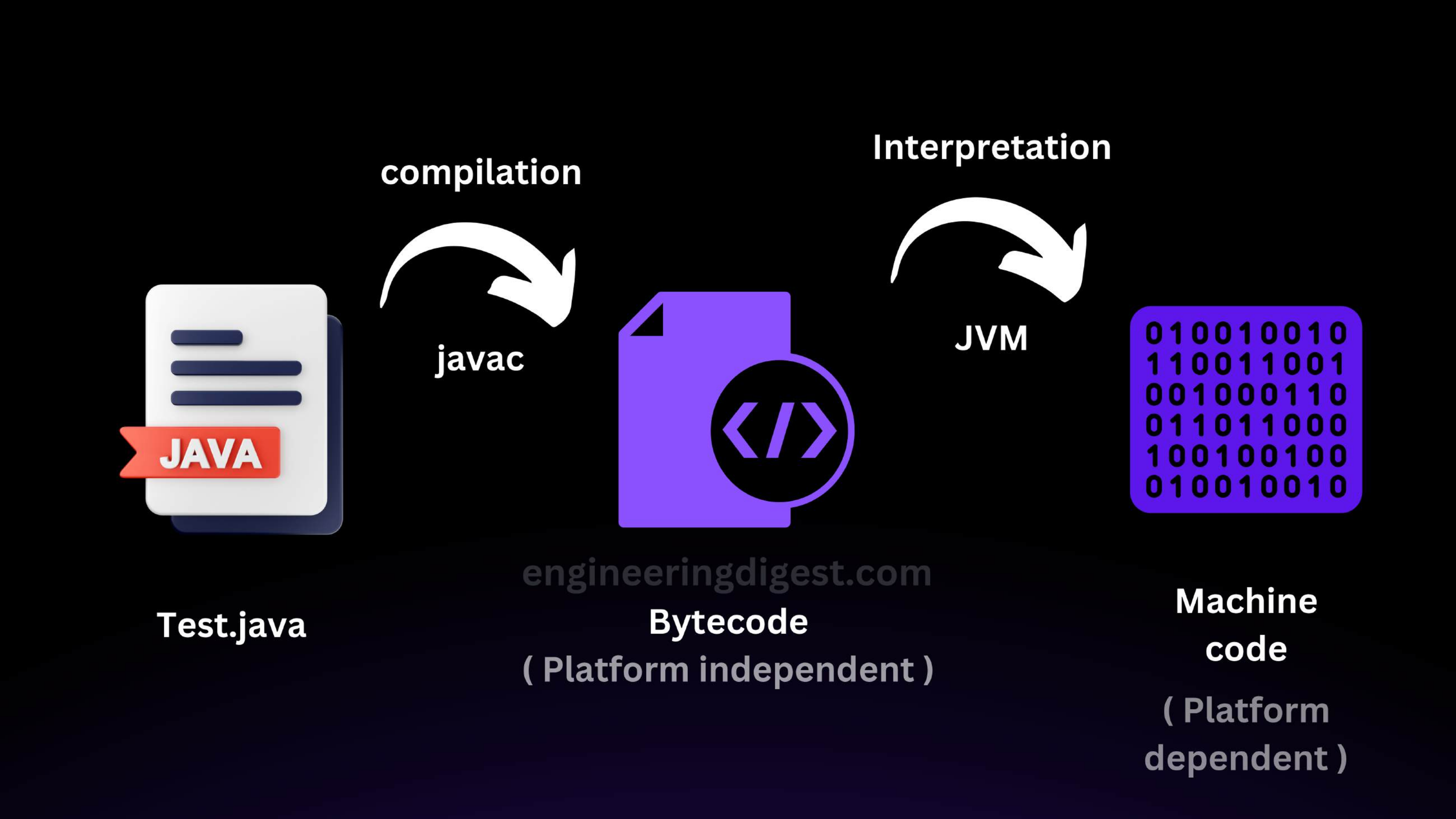
JVM is the actual engine that runs the Java bytecode on any platform

To summarize their relationship:

- JDK contains JRE, which contains JVM
- Developers need JDK to create Java applications
- End users only need JRE to run Java applications
- JVM is the actual engine that runs the Java bytecode on any platform

A Java program runs through a clear pipeline:

1. Source Code is written in *.java* files.
2. Compilation transforms it into platform-independent bytecode (*.class*).
3. Execution is handled by the JVM, which interprets or compiles the *bytecode into native code* for the specific operating system and hardware, allowing the program to run seamlessly across different environments.



```java
class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

```java
class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

This is the main method. In Java, the main method is the entry point of a program. It has a specific signature:

public: Access modifier indicating that the method can be accessed from outside the class.

static: Indicates that the method belongs to the class rather than an instance of the class.

void: Specifies that the method does not return any value.
main: The name of the method.

String[] args: The method accepts an array of strings as parameters. This is where command-line arguments can be passed to your program.

```java
class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

This line prints the string "Hello world!" to the console. Breakdown:

System: A class in the java.lang package that provides access to the system, including the console.

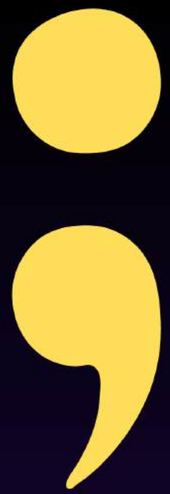
out: An instance of the PrintStream class within the System class, representing the console.

println: A method used to print a line of text to the console.

"Hello world!": The string to be printed.

```java
class Test {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

The semicolon in Java is a crucial element to indicate the end of a statement. It helps the compiler understand the structure of your code by marking the boundaries between different statements.



Data Types

Integral Numbers

byte (1 byte)
short (2 bytes)
int (4 bytes)
long (8 bytes)

Decimal Numbers

float (4 bytes)
double (8 bytes)

True/false

boolean (1 bit)

Characters

char (2 bytes)

String

String Creation Methods

```
String s1 = "Hello";                // String literal
String s2 = new String("Hello");  // Using constructor
```

String Pool vs Heap

```java
String str1 = "Hello";
String str2 = "Hello";
String str3 = new String("Hello");

System.out.println(str1 == str2);      // true (same reference in pool)
System.out.println(str1 == str3);      // false (different objects)
System.out.println(str1.equals(str3)); // true (same content)
```

String Immutability

```java
String name = "John";
name.toUpperCase();
System.out.println(name);   // Still prints "John"

// Correct way
name = name.toUpperCase();  // Creates new string
```

Common String Operations

```java
String text = "Hello World";

// Length
System.out.println(text.length());  // 11

// Accessing characters
System.out.println(text.charAt(0));  // 'H'

// Substring
System.out.println(text.substring(0,5));  // "Hello"

// Contains, startsWith, endsWith
System.out.println(text.contains("World"));  // true

// Replace
String newText = text.replace("World", "Java");
```

Arithmetic Operators

+ - / * %

++

--

Bitwise Operators

&

|

^

!

<<

>>

A	B	A & B	A | B	A ^ B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Conditional Statements

Relational Operators

Logical Operators

Relational Operators

They return a boolean result

<

>

<=

>=

==

!=

Logical Operators

logical operators are used to combine multiple boolean expressions or conditions

&&

||

!

Operand 1	Operand 2	Operand1 && Operand2
true	true	true
true	false	false
false	true	false
false	false	false

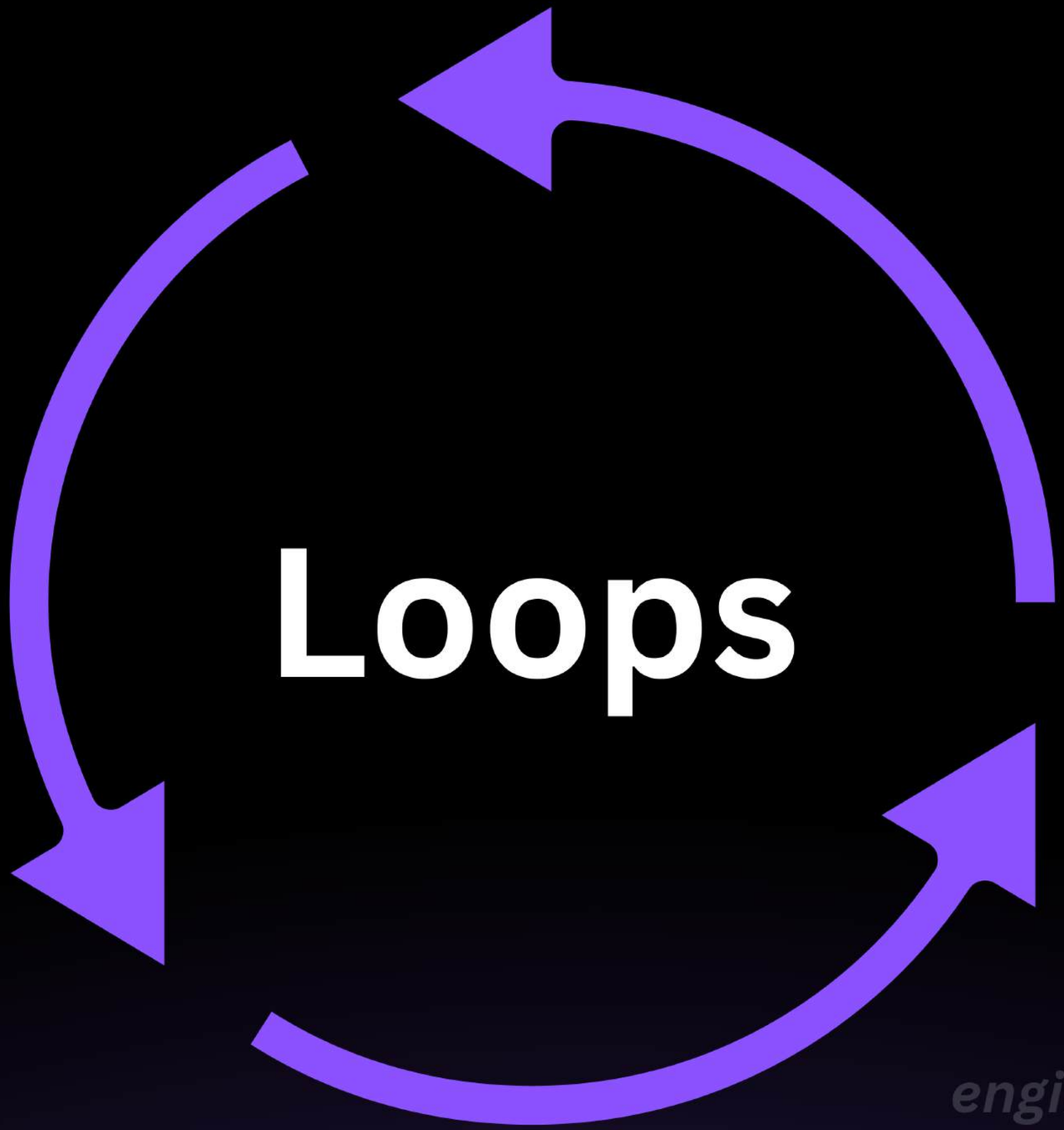
Operand 1	Operand 2	Operand1 || Operand2
true	true	true
true	false	true
false	true	true
false	false	false

Conditional Statements

Conditional statements in Java allow the program to make decisions based on conditions and execute specific blocks of code depending on the outcome.

1. if Statement
2. if-else Statement
3. if-else if Ladder
4. switch Statement (works with int, char, String, and enum types)
5. Ternary Operator (variable = (condition) ? value_if_true : value_if_false;)

Statement	Use Case
if	When a single condition needs to be evaluated.
if-else	When there are two conditions: one for true and one for false.
if-else if	When there are multiple conditions to evaluate in sequence.
switch	When you have multiple discrete values for a single variable to compare.
Ternary	When you need a compact way to represent simple if-else logic.



1. for Loop
2. Enhanced for Loop (for-each Loop)
3. while Loop
4. do-while Loop

Arrays

An array in Java is a data structure that stores a fixed-size *sequential collection of elements of the same type*

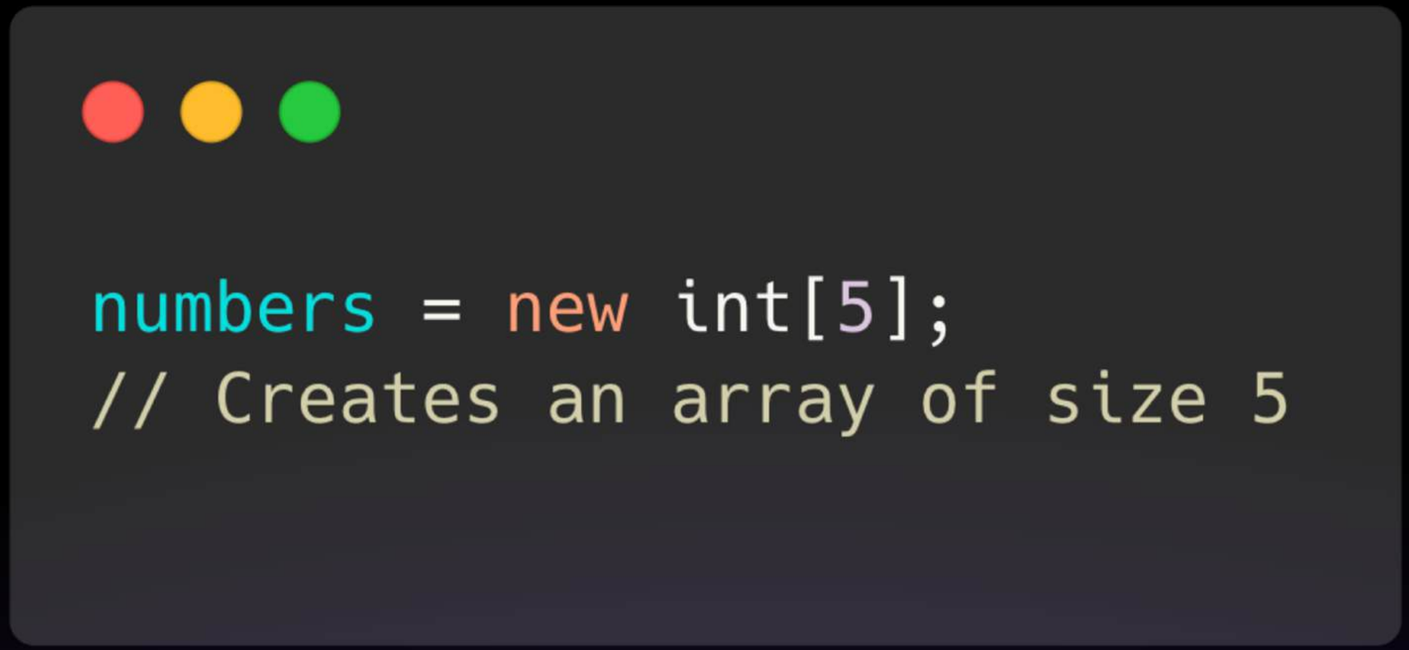
Declaration

An array must be declared with a specific data type.

```
int[] numbers;  // Recommended style
// OR
int numbers[];  // Allowed, but less common
```

Creation

Arrays are created using the new keyword, and their size is specified.



Initialization

Array elements can be initialized individually or in bulk.

```
numbers[0] = 10;  // Initialize individual elements
numbers[1] = 20;

int[] values = {1, 2, 3, 4, 5};  // Combined declaration, creation, and initialization
```

OOPS

Object-Oriented Programming System

A programming paradigm that uses objects and classes to design and implement software solutions.

Key Concepts of OOPS in Java

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

Class

A class is a blueprint for creating objects. It defines the structure (fields) and behavior (methods) of objects.

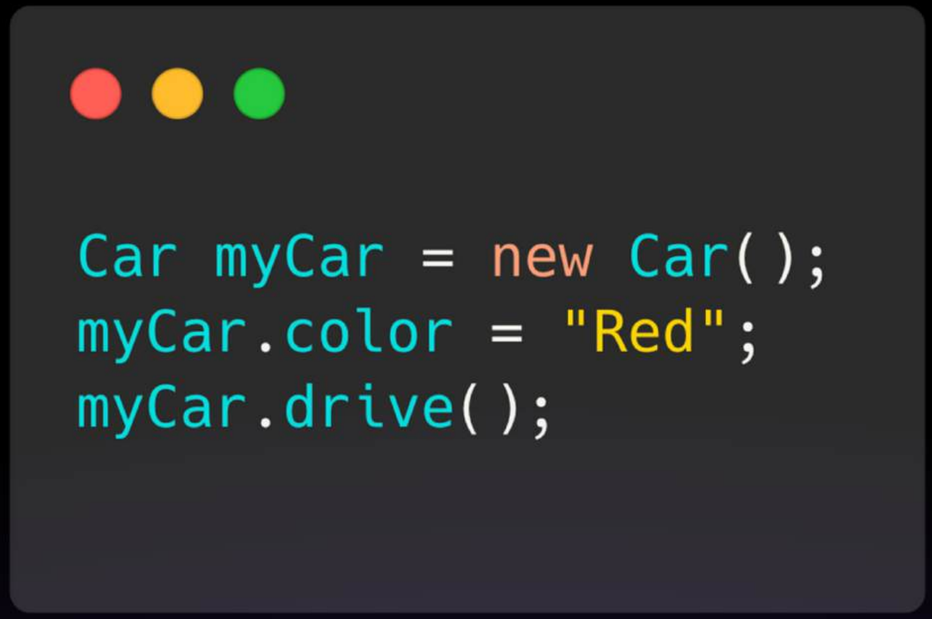
Class

```
class Car {
    String color;
    int speed;

    void drive() {
        System.out.println("Car is driving");
    }
}
```

Object

An object is an instance of a class. It represents a real-world entity and has attributes and behaviors.

Object



```
Car myCar = new Car();
myCar.color = "Red";
myCar.drive();
```

Encapsulation

Encapsulation is the practice of bundling data (fields) and methods (functions) that operate on the data into a single unit (class). It also involves restricting direct access to some components using access modifiers (e.g., private, protected).

Encapsulation

```java
class Employee {
    private String name;
    private int salary;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Inheritance

Inheritance allows a class to acquire properties and methods of another class. It supports code reusability.

Inheritance

```
class Animal {
    void eat() {
        System.out.println("This animal eats food.");
    }
}

class Dog extends Animal {
    void bark() {
        System.out.println("Dog barks.");
    }
}
```

Inheritance

Single

Multilevel

Hierarchical

Java Doesn’t support Multiple inheritance

Polymorphism

Polymorphism allows methods to perform different tasks based on the object that calls them.

It can be achieved via:

- Runtime Polymorphism (Method Overriding)
- Compile-time Polymorphism (Method Overloading)

Method overloading

(Compile-Time Polymorphism)

Method Overloading in Java is a feature that allows a class to have multiple methods with the same name but different parameter lists. It enables a method to perform different tasks depending on the arguments passed to it.

Different Number of Parameters

```java
class Calculator {
    // Add two integers
    int add(int a, int b) {
        return a + b;
    }

    // Add three integers
    int add(int a, int b, int c) {
        return a + b + c;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));          // Output: 5
        System.out.println(calc.add(1, 2, 3));       // Output: 6
    }
}
```

Different Types of Parameters

```java
class Printer {
    void print(String s) {
        System.out.println("String: " + s);
    }

    void print(int num) {
        System.out.println("Integer: " + num);
    }

    void print(double d) {
        System.out.println("Double: " + d);
    }
}

public class Main {
    public static void main(String[] args) {
        Printer printer = new Printer();
        printer.print("Hello, World!");  // Output: String: Hello, World!
        printer.print(100);             // Output: Integer: 100
        printer.print(3.14);            // Output: Double: 3.14
    }
}
```

Different Order of Parameters

```java
class Display {
    void show(String s, int num) {
        System.out.println("String: " + s + ", Number: " + num);
    }

    void show(int num, String s) {
        System.out.println("Number: " + num + ", String: " + s);
    }
}

public class Main {
    public static void main(String[] args) {
        Display display = new Display();
        display.show("Java", 101);  // Output: String: Java, Number: 101
        display.show(202, "OOPS");  // Output: Number: 202, String: OOPS
    }
}
```

Run-Time Polymorphism (Dynamic Polymorphism)

Run-time polymorphism is achieved through method overriding, where a subclass provides a specific implementation of a method already defined in its parent class. The method to be called is determined at runtime based on the object.

Rintime Polymorphism

```java
class Animal {
    void sound() {
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    void sound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    void sound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal animal1 = new Dog(); // Upcasting
        Animal animal2 = new Cat(); // Upcasting

        animal1.sound(); // Calls Dog's overridden method: "Dog barks"
        animal2.sound(); // Calls Cat's overridden method: "Cat meows"
    }
}
```

Abstraction

Abstraction focuses on showing only essential details while hiding the implementation. It is achieved through abstract classes and interfaces.

Abstract Class

- Declared using the abstract keyword.
- Can include both abstract methods (methods without a body) and concrete methods (methods with a body).
- Cannot be instantiated directly.
- Acts as a blueprint for subclasses, which must implement the abstract methods.

```java
public class Test {
    public static void main(String[] args) {
        Animal bob = new Dog(); Animal bobyy = new Cat();
        bob.sayBye();
        bobyy.sayBye();
        bob.sleep();
        bobyy.sleep();
    }
}
abstract class Animal{
    public abstract void sayHello();
    public abstract void sayBye();
    public void sleep(){
        System.out.println("zzzzz...");
    }
}
class Dog extends Animal{
    public void sayHello() {
        System.out.println("Woof");
    }
    public void sayBye() {
        System.out.println("Woof Woof");
    }
}
class Cat extends Animal{
    public void sayHello() {
        System.out.println("Meow");
    }
    public void sayBye() {
        System.out.println("Meow Meow");
    }
}
```

Access Modifiers

Access Modifier	Scope within the Class	Scope within the Package	Scope in Subclasses (Different Package)	Scope Everywhere
public	✅ Yes	✅ Yes	✅ Yes	✅ Yes
protected	✅ Yes	✅ Yes	✅ Yes	❌ No
default (no keyword)	✅ Yes	✅ Yes	❌ No	❌ No
private	✅ Yes	❌ No	❌ No	❌ No

Interface

Class --> Blueprint for object

Interface --> Blueprint for class

Interface

By Interface, we acheive abstraction and multiple inheritance.

Interface

It can have abstract methods, static constants, static methods and default methods.

Static Methods in Interface:

- Used for utility operations that are RELATED to the interface but don't need instance state
- Cannot be overridden by implementing classes
- Called directly on the interface (not through instance)

```java
interface PaymentValidator {
    boolean validatePayment(Payment payment);

    // Static utility method - helper functions related to validation
    static boolean isValidCreditCard(String cardNumber) {
        // Luhn algorithm check
        return cardNumber.length() == 16;
    }

    static boolean isValidAmount(double amount) {
        return amount > 0 && amount < 1000000;
    }
}

class PayPalValidator implements PaymentValidator {
    @Override
    public boolean validatePayment(Payment payment) {
        // First use static utility method
        if (!PaymentValidator.isValidAmount(payment.getAmount())) {
            return false;
        }
        // Then do PayPal specific validation
        return true;
    }
}
```

Default Methods in Interface

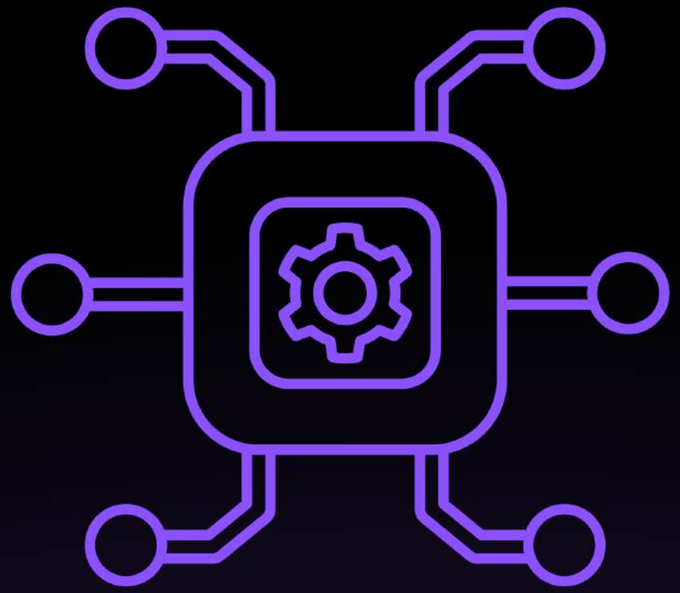
- Provide optional functionality to implementing classes
- Can be overridden if needed
- Can use other interface methods (abstract or default)
- Called through instance

```java
interface PaymentProcessor {
    void processPayment(Payment payment);

    // Default method using abstract method
    default void processPayments(List<Payment> payments) {
        for(Payment payment: payments) {
            processPayment(payment);
        }
    }

    // Default method with common implementation
    default void validateAndProcess(Payment payment) {
        if(payment.getAmount() <= 0) {
            throw new IllegalArgumentException("Invalid amount");
        }
        processPayment(payment);
    }
}

class StripeProcessor implements PaymentProcessor {
    @Override
    public void processPayment(Payment payment) {
        // Stripe specific implementation
    }

    // Can use default processPayments() as is
    // Can override validateAndProcess() if needed
}
```

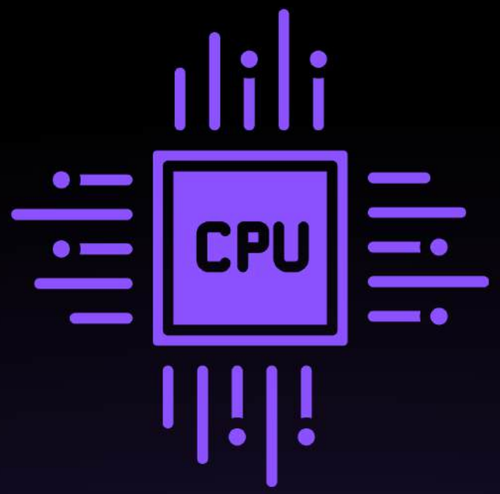
Interface

Abstract Class

Multithreading

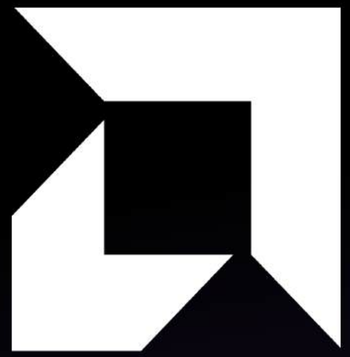


The CPU, often referred to as the brain of the computer, is responsible for executing instructions from programs. It performs basic arithmetic, logic, control, and input/output operations specified by the instructions.

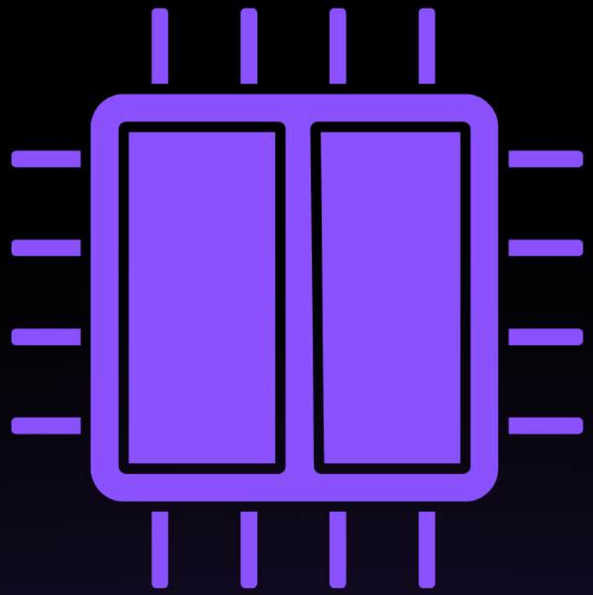


Example

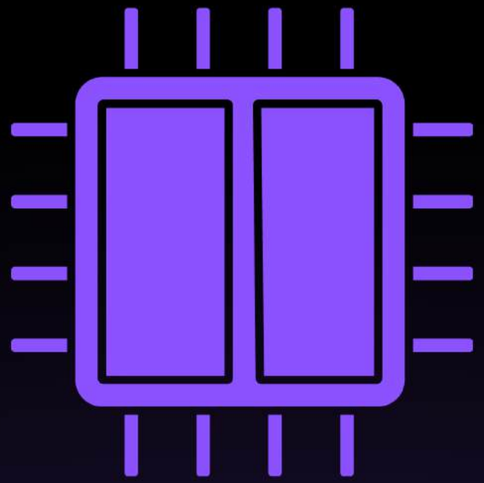
A modern CPU like the Intel Core i7 or AMD Ryzen 7



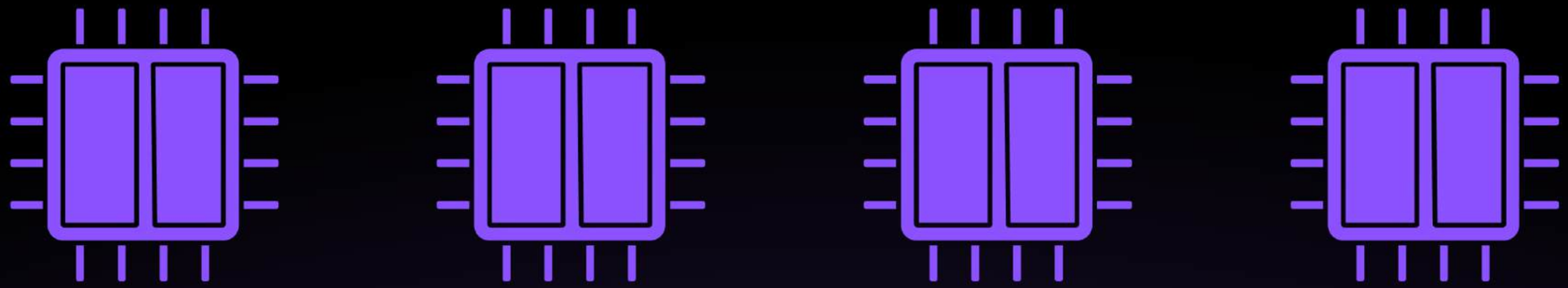
Core



A core is an individual processing unit within a CPU. Modern CPUs can have multiple cores, allowing them to perform multiple tasks simultaneously.



A quad-core processor has four cores, allowing it to perform four tasks simultaneously. For instance, one core could handle your web browser, another your music player, another a download manager, and another a background system update.

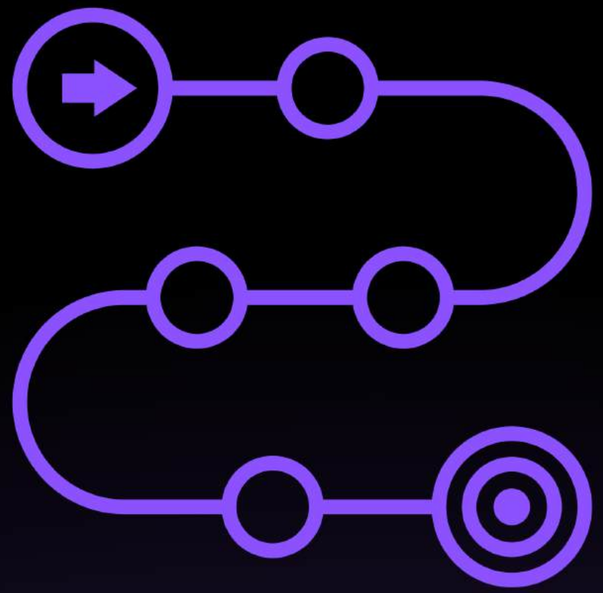


Program

A program is a set of instructions written in a programming language that tells the computer how to perform a specific task

Microsoft Word is a program that allows users to create and edit documents.

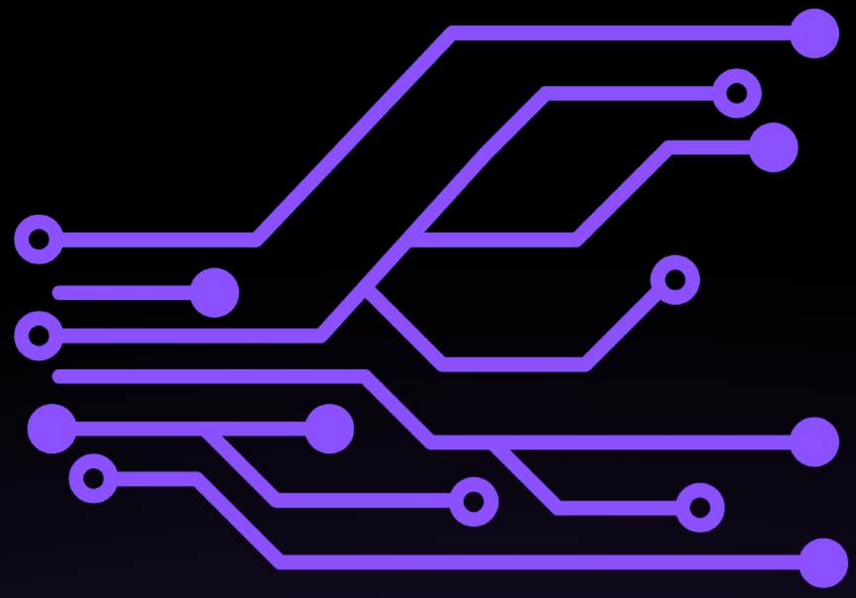
Process



A process is an instance of a program that is being executed. When a program runs, the operating system creates a process to manage its execution.

When we open Microsoft Word, it becomes a process in the operating system.

Thread

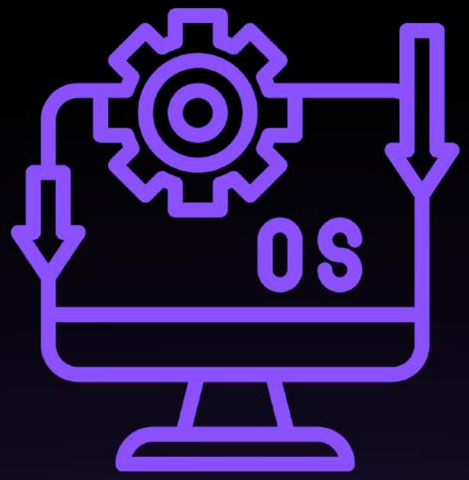


A thread is the smallest unit of execution within a process. A process can have multiple threads, which share the same resources but can run independently.

A web browser like Google Chrome might use multiple threads for different tabs, with each tab running as a separate thread.

Multitasking

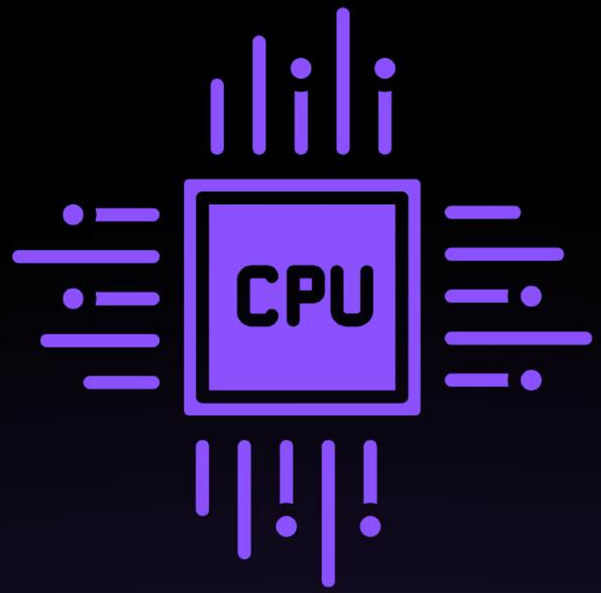
Multitasking allows an operating system to run multiple processes simultaneously. On single-core CPUs, this is done through rapidly switching between tasks. On multi-core CPUs, true parallel execution occurs, with tasks distributed across cores. The OS scheduler balances the load, ensuring efficient and responsive system performance.



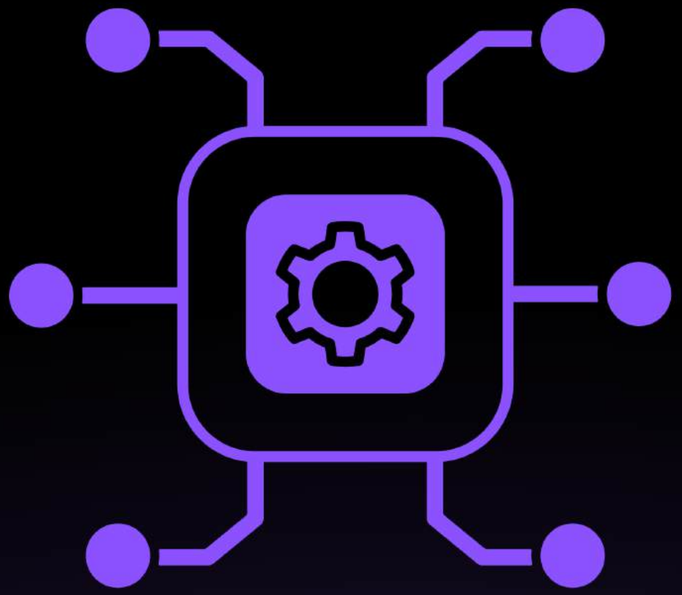
Example

We are browsing the internet while listening to music and downloading a file.

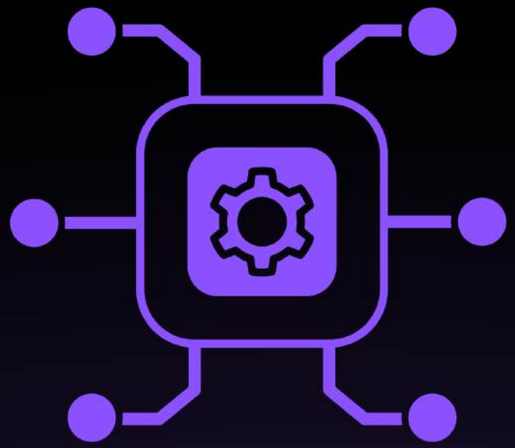
Multitasking utilizes the capabilities of a CPU and its cores. When an *operating system* performs multitasking, it can assign different tasks to different cores. This is more efficient than assigning all tasks to a single core.



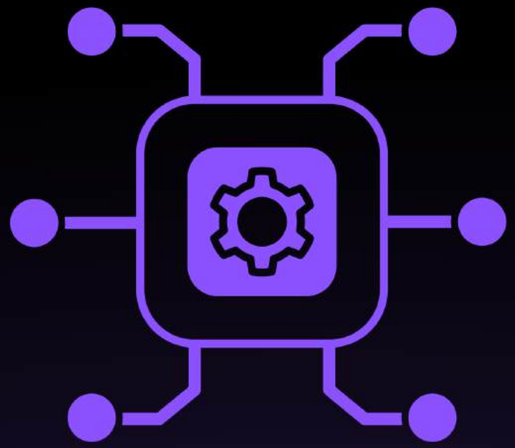
Multithreading



Multithreading refers to the ability to execute multiple threads within a single process concurrently.



A web browser can use multithreading by having separate threads for rendering the page, running JavaScript, and managing user inputs. This makes the browser more responsive and efficient.



Multitasking can be achieved through multithreading where each task is divided into threads that are managed concurrently.

While multitasking typically refers to the running of multiple applications, multithreading is more granular, dealing with multiple threads within the same application or process.

Multitasking operates at the level of processes, which are the operating system’s primary units of execution.

Multithreading operates at the level of threads, which are smaller units within a process

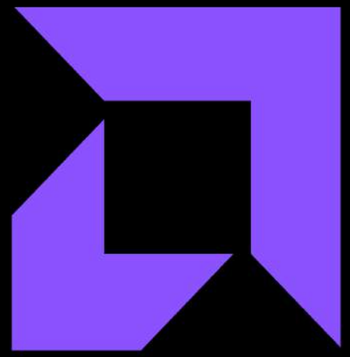
Multitasking allows us to run multiple applications simultaneously, improving productivity and system utilization.

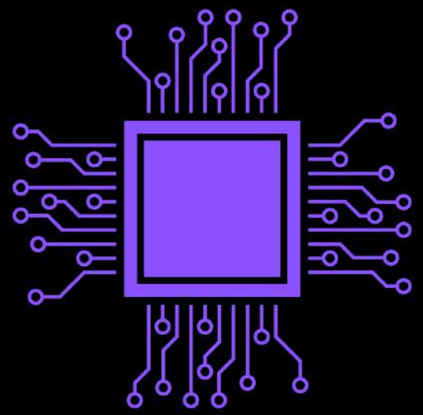
Multithreading allows a single application to perform multiple tasks at the same time, improving application performance and responsiveness.

The office manager (operating system) assigns different employees (processes) to work on different projects (applications) simultaneously. Each employee works on a different project independently.

Within a single project (application), a team (process) of employees (threads) works on different parts of the project at the same time, collaborating and sharing resources.

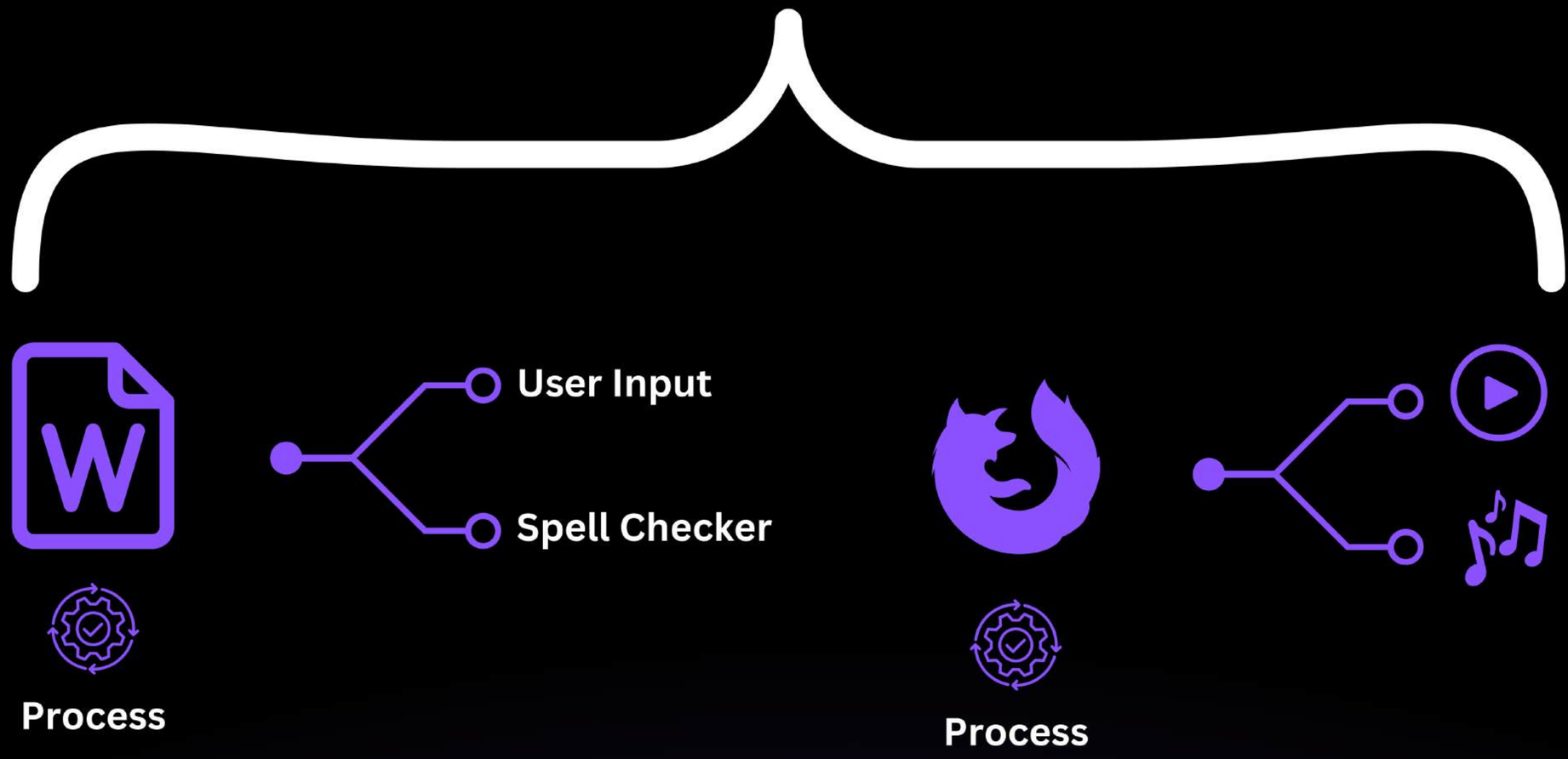
AMD Ryzen 9 5900HS





Octa core

Multitasking (Managed by OS)



Java provides robust support for multithreading, allowing developers to create applications that can perform multiple tasks simultaneously, improving performance and responsiveness.

In Java, multithreading is the concurrent execution of two or more threads to maximize the utilization of the CPU. Java's multithreading capabilities are part of the java.lang package, making it easy to implement concurrent execution.

In a single-core environment, Java's multithreading is managed by the JVM and the OS, which switch between threads to give the illusion of concurrency.

The threads share the single core, and time-slicing is used to manage thread execution.

In a multi-core environment, Java's multithreading can take full advantage of the available cores.

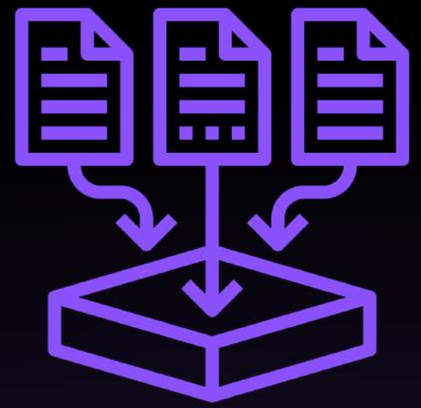
The JVM can distribute threads across multiple cores, allowing true parallel execution of threads.

A thread is a lightweight process, the smallest unit of processing. Java supports multithreading through its java.lang.Thread class and the java.lang.Runnable interface.

When a Java program starts, one thread begins running immediately, which is called the main thread. This thread is responsible for executing the main method of a program.

To create a new thread in Java, you can either extend the Thread class or implement the Runnable interface.

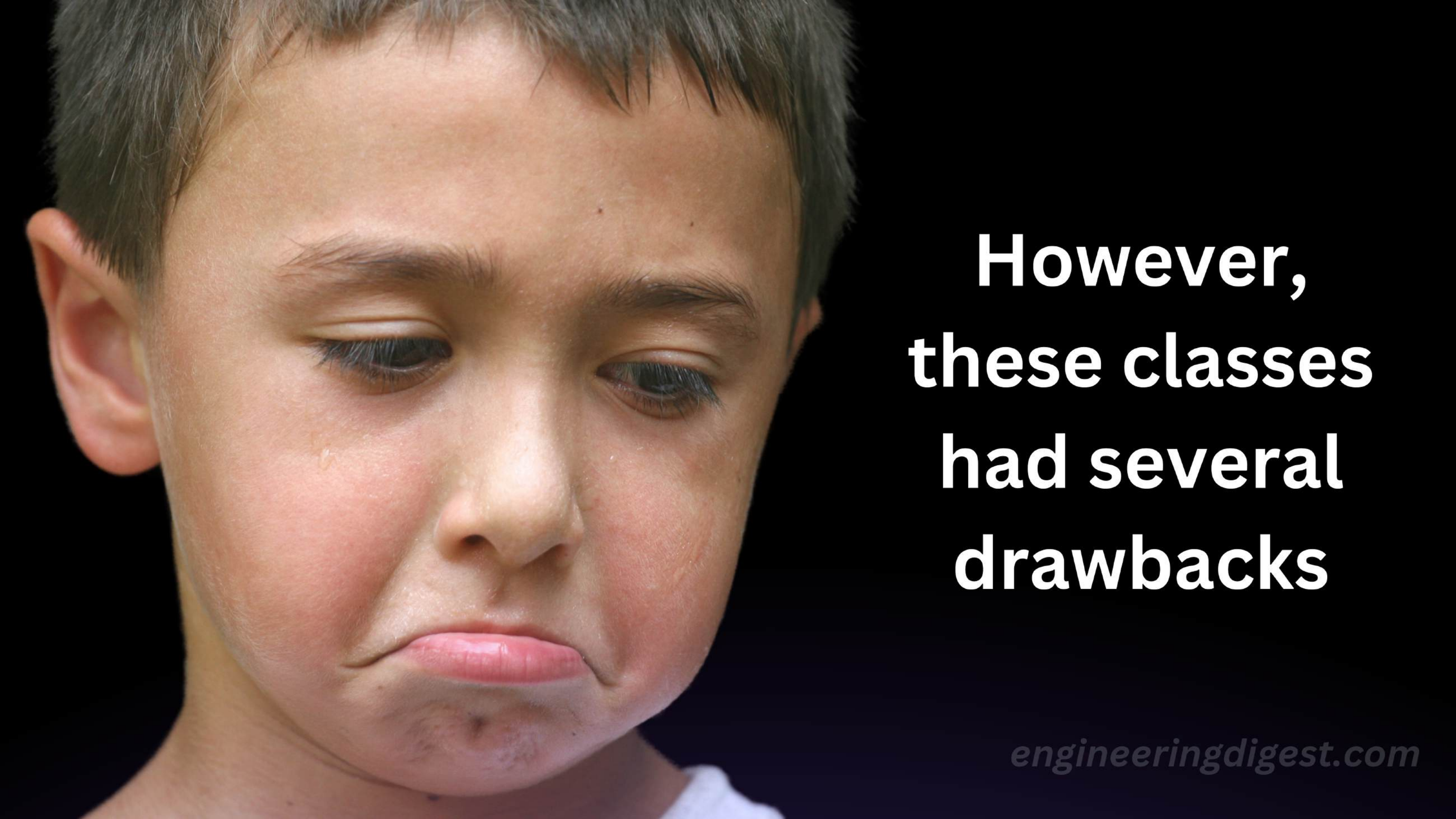
Collection Framework



What is a Collection Framework?

It provides a set of interfaces and classes that help in managing groups of object.

Before the introduction of the Collection Framework in JDK 1.2, Java used to rely on a variety of classes like *Vector*, *Stack*, *Hashtable*, and *arrays* to store and manipulate groups of objects.


However,
these classes
had several
drawbacks
engineeringdigest.com

- *Inconsistency:* Each class had a different way of managing collections, leading to confusion and a steep learning curve.

- *Lack of inter-operability:* These classes were not designed to work together seamlessly.

- *No common interface:* There was no common interface for all these classes, which meant you couldn't write generic algorithms that could operate on different types of collections.