**📝 The History of Java:**

**History & Philosophy** Java was initiated by **James Gosling** at Sun Microsystems and released in **1995**. Originally called **'Oak'**, it was later renamed Java. Its core philosophy is **"Write Once, Run Anywhere" (WORA)**, enabled by its platform independence. In 2006, much of Java was released as free and open-source software under the **GNU General Public License (GPL)**.

**1. Origin and Naming**

The project began as an initiative by the **'Green' team** to create a language for various electronic devices.

* **GreenTalk**: The initial name for the language. Its file extension was .gt.
* **Oak**: Renamed after an oak tree outside Gosling's office. This name was chosen to represent solidarity and strength, but it was already a trademark of Oak Technologies.
* **Java**: The final name, selected from a list of options. It was inspired by the **Java coffee bean** and was chosen for its uniqueness and dynamic, revolutionary feel.

**2. Key Milestones and Philosophy**

* **Java 1.0 (1995)**: The first public release. It established the core philosophy of **"Write Once, Run Anywhere" (WORA)**, promising no-cost runtimes on popular platforms.
* **Open Sourcing (2006-2007)**: Sun Microsystems released the majority of Java's core code as free and open-source software under the **GNU General Public License (GPL)**, making it accessible to a wider community.
* **Configurations**: Over time, Java was adapted for different platforms, including **J2EE** for enterprise applications and **J2ME** for mobile devices.

**3. Evolution of Versions**

Java has undergone continuous development, introducing significant new features with each major release.

* **The Early Years (JDK 1.0 - JDK 1.4)**
  + **JDK 1.1**: Added fundamental features like **JavaBeans** and **JDBC**.
  + **JDK 1.2**: A major success, introducing **Swing**, the **JIT Compiler**, and the **Collections Framework**.
  + **JDK 1.5 (J2SE 5)**: A highly impactful release that brought key language enhancements like **Generics**, foreach loops, and var-args.
* **Modern Java (Java SE 6 - Present)**
  + **Java SE 8 (2014)**: A monumental release that introduced **functional programming** with **Lambda Expressions** and **Streams**, as well as the new **Date-Time API**.
  + **Java SE 9 (2017)**: Introduced the **Java Module System** (Jigsaw) for better application scaling and security.
  + **Recent Versions (SE 14+)**: The rapid release cadence has delivered many modern features, including **Records**, **Pattern Matching for instanceof and switch**, **Sealed Classes**, and **Text Blocks**. The latest major releases mentioned are **Java SE 22** and **Java SE 23**.

**Key Characteristics**

* **Object-Oriented**: Everything in Java is an object, making it easily extensible.
* **Platform Independent**: The Java compiler generates **bytecode**, not machine code. This bytecode is run by the **Java Virtual Machine (JVM)** on any platform.
* **Secure**: Java enables the development of virus-free and tamper-free systems using authentication based on public-key encryption.
* **Robust**: It reduces error-prone situations by performing strong compile-time and run-time error checking.
* **Multithreaded**: This feature allows programs to perform many tasks simultaneously, which is great for interactive applications.
* **High Performance**: Uses **Just-in-Time (JIT) compilers** to enable high performance.
* **Portable**: It is architecture-neutral, with no implementation-dependent aspects, making it highly portable.
* **Simple**: Designed to be easy to learn, especially if you understand the basic concepts of OOP.

**Configurations** Java has evolved into multiple configurations to suit different platforms:

* **Java SE** (Standard Edition): For general-purpose and desktop applications.
* **Java EE** (Enterprise Edition): For large-scale enterprise applications.
* **Java ME** (Micro Edition): For mobile and embedded systems.

**"Hello World" Example** A basic Java program to print "Hello World" looks like this:

public class MyFirstJavaProgram {

public static void main(String []args) {

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

}

}

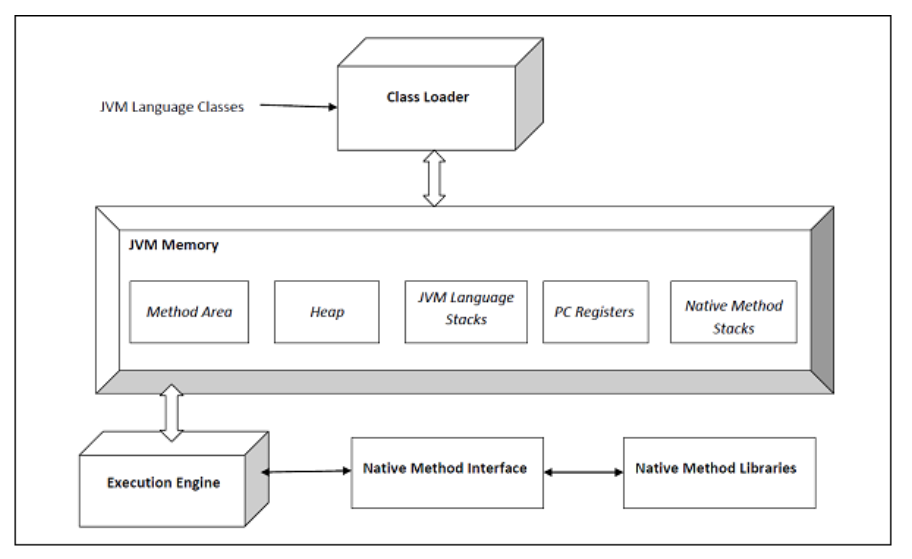
**What is the Java Virtual Machine (JVM)? 💻**

The **Java Virtual Machine (JVM)** is a virtual, abstract computer that serves as the runtime environment for Java bytecode. It's a key component of Java's "Write Once, Run Anywhere" philosophy, as it allows compiled Java code to run on any operating system. The JVM itself is a specification, meaning there are many different implementations (like Oracle's HotSpot JVM or IBM's J9 JVM), as long as they follow the defined rules.

The JVM manages and executes Java programs by handling tasks such as class loading, memory management (heap and stack), and garbage collection.

**JVM Architecture and Components 🏗️**

The JVM's architecture is composed of several key components that work together to execute Java applications.



**1. Class Loader**

The Class Loader is responsible for dynamically loading, linking, and initializing classes. It follows a hierarchy of three loaders:

* **BootStrap Class Loader**: Loads core JDK classes.
* **Extension Class Loader**: Loads classes from the jre/lib/ext directory.
* **Application Class Loader**: Loads classes from the classpath.

**2. Runtime Data Areas**

These are the memory areas used by the JVM during program execution.

* **Method Area**: Stores per-class structures like the constant pool, method data, and constructor code. It is shared among all threads.
* **Heap**: Shared among all threads and stores all objects, arrays, and classes. It is the primary area for garbage collection.
* **Stack**: A local memory area for each thread. It stores method calls, local variables, and return addresses.
* **PC (Program Counter) Register**: A local register for each thread that holds the address of the currently executing JVM instruction.
* **Native Method Stacks**: A separate stack created for each thread that calls a native (non-Java) method.

**3. Execution Engine**

The Execution Engine is the core of the JVM, responsible for executing the bytecode. It contains:

* **Interpreter**: Reads and executes bytecode instruction by instruction. It's fast at interpreting but slow in execution.
* **JIT (Just-In-Time) Compiler**: Compiles frequently executed bytecode into native machine code at runtime, significantly improving performance.
* **Garbage Collector (GC)**: An automatic memory management system that removes unused or "dead" objects from the Heap to free up memory. This process helps prevent memory leaks.

**4. Other Components**

* **Java Native Interface (JNI)**: Allows Java code running in the JVM to interact with native applications and libraries written in other languages, like C and C++.
* **Native Method Libraries**: The collection of native libraries that JNI interacts with to provide functionality not available in pure Java.

**JDK vs. JRE vs. JVM**

The **JDK (Java Development Kit)**, **JRE (Java Runtime Environment)**, and **JVM (Java Virtual Machine)** are three core components of the Java platform that work together to develop and run Java programs. Their relationship can be visualized as a layered system where the JDK contains the JRE, and the JRE contains the JVM.

**1. JDK (Java Development Kit) 🛠️**

The **JDK** is a complete software development kit for Java developers. It's the highest-level component and is used for creating, compiling, and debugging Java applications.

* **Primary Function**: Development and execution of Java code.
* **Contents**: Includes everything needed for development, such as the compiler (javac), debugger, and other development tools, in addition to a complete **JRE**.
* **Platform Dependency**: The JDK is **platform-dependent**, meaning you need to download a specific version for your operating system (e.g., Windows, macOS, or Linux).

**2. JRE (Java Runtime Environment) ⚙️**

The **JRE** is the runtime environment that allows you to run compiled Java applications. It's an integral part of the JDK but can also be installed separately on a system that only needs to execute Java programs, without the need for development tools.

* **Primary Function**: Providing the runtime environment for code execution.
* **Contents**: Contains the **JVM** and the necessary class libraries and support files required to run a Java program. It does **not** include development tools like a compiler.
* **Platform Dependency**: Like the JDK, the JRE is also **platform-dependent**.

**3. JVM (Java Virtual Machine) 🖥️**

The **JVM** is an abstract, virtual machine that provides a runtime environment for executing Java bytecode. It's the lowest-level component and the core of Java's platform independence.

* **Primary Function**: Responsible for converting Java bytecode into machine-specific code, which allows the "Write Once, Run Anywhere" philosophy to work. It loads, verifies, and executes the code.
* **Nature**: The JVM is a **specification**. It can have multiple implementations (e.g., Oracle's HotSpot JVM), but all must adhere to the same set of rules.
* **Platform Independence**: The JVM specification is **platform-independent**, but its actual implementation (which is the JRE) is platform-dependent.

**Summary of Key Differences**

| Key Aspect | JDK (Development Kit) | JRE (Runtime Environment) | JVM (Virtual Machine) |
| --- | --- | --- | --- |
| **Purpose** | To develop and execute Java applications | To run Java applications | To provide a runtime environment for bytecode execution |
| **Contents** | JRE + Development tools | JVM + Libraries | Runtime environment only |
| **Platform** | Platform-dependent | Platform-dependent | Platform-independent (as a specification) |
| **Relationship** | The complete package for developers | An implementation of the JVM | The abstract specification |

A "Hello World" program is the foundational first step in learning any programming language. In Java, it helps you understand the basic structure of a program, including classes, methods, and printing output to the console.

**The "Hello World" Program in Java 💻**

Java

public class MyFirstJavaProgram {

public static void main(String []args) {

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

}

}

**Writing and Running the Program 🚀**

1. **Write**: Open a text editor like Notepad and type the code exactly as shown above.
2. **Save**: Save the file with the same name as the public class, followed by a .java extension. In this case, save it as **MyFirstJavaProgram.java**.
3. **Compile**: Open your command prompt or terminal. Navigate to the directory where you saved the file. To compile the code, type javac MyFirstJavaProgram.java and press Enter. This command uses the **Java compiler (javac)** to translate your source code into bytecode, creating a file named MyFirstJavaProgram.class.
4. **Run**: To run the compiled bytecode, type java MyFirstJavaProgram and press Enter. This command invokes the **Java Virtual Machine (JVM)** to execute your program. The output "Hello World" will be displayed on the console.

**Understanding the Code 🧠**

* **public class MyFirstJavaProgram**: This line declares a **public class** named MyFirstJavaProgram. In Java, the file name must match the public class name.
* **public static void main(String []args)**: This is the **main method**, the entry point of the program. The JVM looks for and executes this method to start the application.
  + public: The method is accessible from outside the class (the JVM can call it).
  + static: The method can be called without creating an object of the class.
  + void: The method does not return any value.
  + main: The method's name, a standard convention.
* **System.out.println("Hello World");**: This statement prints the string "Hello World" to the console.
  + System.out: Represents the standard output stream (the console).
  + println(): A method that prints its argument to the console and moves the cursor to the next line.

**COMMENTS**

Java comments are non-executable text notes within the source code used for explanation and documentation. The Java compiler ignores them. There are three types of comments in Java, similar to C and C++.

**1. Single-Line Comments ( // )**

Single-line comments start with two forward slashes (//) and extend to the end of the line. They are ideal for brief, in-line explanations of a single line of code or a specific variable.

**Example**:

Java

// This line of code calculates the total price

double totalPrice = itemPrice \* quantity;

**2. Multi-Line Comments ( /\* ... \*/ )**

Multi-line comments begin with a forward slash and an asterisk (/\*) and end with an asterisk and a forward slash (\*/). Everything in between is considered a comment, regardless of how many lines it spans. These are useful for long explanations or for commenting out blocks of code temporarily.

**Example**:

Java

/\*

This method takes a user's input,

validates it, and then saves it

to the database.

\*/

public void processUserData() {

// ... method logic here

}

**3. Documentation Comments ( /\*\* ... \*/ )**

Documentation comments start with a forward slash and two asterisks (/\*\*) and end with an asterisk and a forward slash (\*/). They are specifically designed for use with the **Javadoc tool**, which generates professional, HTML-based API documentation directly from the source code. These comments often contain special tags like @param, @return, and @throws to describe a method's parameters, return value, and potential exceptions.

**Example**:

Java

/\*\*

\* Calculates the square of a given number.

\* \* @param number The input number to be squared.

\* @return The square of the input number.

\*/

public double calculateSquare(double number) {

return number \* number;

}

**What are the fundamental components of a Java program? 🖥️**

A Java program is built upon the principles of Object-Oriented Programming (OOP) and is a collection of interacting objects. The foundational components that make this possible are classes, objects, methods, and instance variables.

**Core OOP Concepts**

* **Object**: An object represents a real-world entity that has a **state** (data/properties) and **behavior** (actions). For example, a dog object can have states like color and name and behaviors like barking and eating. An object is an instance of a class.
* **Class**: A class is a blueprint or template from which objects are created. It defines the state and behavior that all objects of that type will have.
* **Methods**: A method is a set of instructions that defines an object's behavior. Methods contain the logic for manipulating data and performing actions.
* **Instance Variables**: These variables hold the unique state for each individual object. The values assigned to an object's instance variables define its unique state.

**Basic Syntax and Program Structure**

* **Case Sensitivity**: Java is case-sensitive. Hello and hello are considered different identifiers.
* **Class and Method Names**: Class names should start with an uppercase letter (MyClass), while method names should start with a lowercase letter (myMethod).
* **File Name**: The name of the Java source file must exactly match the name of the public class it contains, with a .java extension (e.g., MyClass.java).
* **main() Method**: Program execution always begins from the public static void main(String[] args) method. It is the mandatory entry point for every Java application.

**Identifiers, Modifiers, and Keywords**

* **Identifiers**: These are names given to classes, variables, and methods. They must start with a letter, \_, or $. After the first character, they can include any combination of letters and numbers.
* **Modifiers**: Keywords that modify the behavior of classes, methods, and variables. They are categorized as:
  + **Access Modifiers**: public, private, protected, and default. They control the visibility of a class or its members.
  + **Non-access Modifiers**: final, abstract, static, etc. They provide special functionality.
* **Keywords**: A set of predefined, reserved words (e.g., class, public, void) that cannot be used as identifiers. There are 50 reserved keywords in Java, each with a specific meaning to the compiler.

**Other Core Concepts**

* **Arrays**: Objects that store a fixed-size collection of variables of the same data type.
* **Enums**: Introduced in Java 5.0, enums (enumerated types) restrict a variable to a predefined set of constant values, helping to prevent bugs.
* **Inheritance**: A mechanism where a new class (**subclass**) can inherit properties and behaviors from an existing class (**superclass**). This allows for code reuse.
* **Interfaces**: A contract that a class can sign, defining a set of methods that the class must implement. Interfaces define the "what" (the methods) but not the "how" (the implementation).

**What is a Java Variable?**

In Java, a variable is a named storage location in memory that holds a specific type of data. The **data type** of a variable determines the size of its memory allocation, the kind of values it can store, and the operations that can be performed on it. Before a variable can be used, it must be **declared** by specifying its data type and name, and it can be **initialized** with a value using the assignment operator (=).

Java

int myAge; // Declaration

String name = "John"; // Declaration and Initialization

**Types of Variables 📦**

Java has three main types of variables, each with its own scope and lifecycle.

**1. Local Variables**

* **Location**: Declared inside a method, constructor, or a specific code block.
* **Scope**: They are only accessible within the block in which they are declared.
* **Lifecycle**: Created when the method or block is entered and destroyed when it exits.
* **Initialization**: **Must be explicitly initialized** before they are used. They do not have a default value.

**2. Instance Variables**

* **Location**: Declared inside a class, but outside of any method, constructor, or block.
* **Scope**: They belong to a specific object (instance) of a class. Each object has its own unique copy of the instance variables.
* **Lifecycle**: Created when a new object is instantiated with the new keyword and destroyed when the object is garbage collected.
* **Initialization**: They are assigned a **default value** if not explicitly initialized (e.g., 0 for numbers, false for booleans, null for objects).

**3. Static (Class) Variables**

* **Location**: Declared inside a class with the static keyword.
* **Scope**: They belong to the class itself, not to any individual object. There is only one copy of a static variable, which is shared by all instances of the class.
* **Lifecycle**: Created when the program starts and the class is loaded, and they exist for the entire duration of the program.
* **Initialization**: They are assigned a **default value** if not explicitly initialized. They are often declared as final to be used as constants.

**What are the Data Types in Java? 🔢**

Java has two main categories of data types: **primitive** and **reference (non-primitive)**. Data types are essential as they determine the type of data a variable can hold and the amount of memory allocated to it.

**Primitive Data Types**

Primitive data types are fundamental, predefined data types that hold a single value. Java has eight of them.

* **byte**: An 8-bit integer, ranging from -128 to 127. Useful for saving memory in large arrays.
* **short**: A 16-bit integer, ranging from -32,768 to 32,767. Also used for memory efficiency.
* **int**: The most commonly used 32-bit integer, with a wide range of values. This is the default data type for whole numbers.
* **long**: A 64-bit integer, used for values that exceed the range of an int. You must append an L to the value (e.g., 100000L).
* **float**: A 32-bit single-precision floating-point number. Use f at the end of the value (e.g., 3.14f). Not recommended for precise calculations like currency.
* **double**: A 64-bit double-precision floating-point number. This is the default type for decimal values. Not recommended for precise calculations.
* **boolean**: Represents a single bit of information and can only be true or false.
* **char**: A 16-bit Unicode character, used to store a single character. Values are enclosed in single quotes (e.g., 'A').

**Reference (Non-Primitive) Data Types**

Reference data types are not predefined and are created by the programmer. They are used to access objects. A reference variable holds the memory address of an object, not the value itself. The default value for any reference variable is null.

Examples of reference data types include:

* **Classes**: User-defined types that serve as blueprints for objects (e.g., String, Employee).
* **Arrays**: Objects that can store multiple values of the same type (e.g., int[]).
* **Interfaces**: Abstract types that define a contract for a class to implement.

**Type Casting In Java**

Type casting, or type conversion, is the process of converting a variable from one data type to another. It's an important technique used in Java to ensure compatibility between different types of data. There are two types of type casting: widening and narrowing.

**Widening Type Casting (Implicit)**

**Widening type casting** is the automatic conversion of a smaller data type to a larger one. This process is handled by the Java compiler and is also known as implicit type casting. It is considered "safe" because there is no risk of data loss, as the larger type can fully accommodate the smaller one.

* **Hierarchy**: The conversion follows a specific order from smallest to largest size: byte > short > char > int > long > float > double.
* **When it's used**: When you assign a value of a smaller type to a larger type, the compiler automatically performs the conversion for you. For example, assigning an int to a double.

Java

int myInt = 100;

double myDouble = myInt; // The int is implicitly converted to a double

**Narrowing Type Casting (Explicit)**

**Narrowing type casting** involves converting a larger data type to a smaller one. This process is not done automatically by the compiler because there is a potential for data loss. Therefore, it requires the programmer to perform the conversion manually using a cast operator (). This is also known as explicit type casting.

* **Syntax**: You must place the target data type in parentheses before the variable you want to convert. For example: (int) myDouble.
* **Risk**: If the value of the larger type exceeds the range of the smaller type, data will be lost. For example, converting a double with a decimal value to an int will truncate the decimal part.

Java

double myDouble = 9.75;

int myInt = (int) myDouble; // Explicitly casting the double to an int

System.out.println(myInt); // Output: 9 (decimal part is lost)

**Working with Unicode in Java 📜**

Java has built-in support for **Unicode**, a universal character set that represents characters from all of the world's languages. Unlike older encoding standards that were limited to specific regions (like ASCII), Unicode uses 2 bytes to represent each character, ensuring every character has a unique, consistent value. This makes Java ideal for creating multilingual applications.

**Approaches to Storing Unicode Characters**

There are two primary ways to store Unicode characters in a Java program. The approach you choose depends on whether the character can be easily typed or displayed in your source code.

**1. Using Unicode Escape Sequences**

This approach uses a special format called a **Unicode escape sequence** to represent a character. An escape sequence starts with \u followed by four hexadecimal digits that correspond to the character's Unicode code point. This method is useful for characters that can't be typed directly on a standard keyboard.

Java

// \u0041 is the Unicode escape sequence for the letter 'A'

char letterA = '\u0041';

System.out.println(letterA); // Output: A

**2. Storing Characters Directly**

This is the more common and convenient approach. If a character can be directly typed and displayed in your source code editor, you can simply enclose it in single quotes and assign it to a char variable.

Java

// Storing the character 'A' directly

char letterA = 'A';

System.out.println(letterA); // Output: A

Both methods are valid and produce the same result. The choice depends on practicality: use direct storage for simple, typable characters, and use escape sequences for special or non-typable characters.

**User Input in Java ⌨️**

In Java, you can get user input from the console using the **Scanner class**, which is part of the java.util package. This class provides a simple and convenient way to read various types of data, such as integers, floating-point numbers, and strings.

**How to Use the Scanner Class**

To read user input, you need to follow three simple steps:

1. **Import the class**: Add import java.util.Scanner; at the beginning of your Java file. This makes the Scanner class available for use.
2. **Create a Scanner object**: Instantiate a Scanner object, typically linking it to the standard input stream (System.in).

Java

Scanner input = new Scanner(System.in);

1. **Read the input**: Use the object's built-in methods to read the desired data type. For example, nextInt() reads an integer, and nextLine() reads a full line of text.

**Common Scanner Methods**

The Scanner class includes a variety of methods for different data types.

| Method | Description | Example |
| --- | --- | --- |
| nextInt() | Reads and returns the next integer. | int num = input.nextInt(); |
| nextFloat() | Reads and returns the next float. | float num = input.nextFloat(); |
| nextDouble() | Reads and returns the next double. | double num = input.nextDouble(); |
| next() | Reads and returns the next single word (token). | String word = input.next(); |
| nextLine() | Reads and returns the entire line of input. | String line = input.nextLine(); |

**The Date Class in Java 📅**

In Java, the java.util.Date class is used to represent a specific instant in time, with millisecond precision. It's a fundamental class for handling dates and times.

The Date class provides two primary constructors:

* **Date()**: Initializes a Date object with the current date and time.
* **Date(long millisec)**: Initializes a Date object with a value representing the number of milliseconds that have elapsed since midnight on January 1, 1970 UTC.

The class also includes a variety of methods for date manipulation and comparison, such as before(), after(), equals(), and getTime().

**Formatting Dates**

Formatting dates is crucial for displaying them in a human-readable way. Java offers two primary approaches for this:

**SimpleDateFormat**

The SimpleDateFormat class is a concrete class for formatting and parsing dates in a locale-sensitive manner. You can define custom patterns using a set of format codes (e.g., yyyy for year, MM for month, dd for day).

Java

// Example using SimpleDateFormat

import java.util.Date;

import java.text.SimpleDateFormat;

Date date = new Date();

SimpleDateFormat formatter = new SimpleDateFormat("E yyyy.MM.dd 'at' hh:mm:ss a zzz");

String formattedDate = formatter.format(date);

System.out.println("Current Date: " + formattedDate);

**printf**

You can also format dates using the printf method with format specifiers starting with t. This method is often more concise, especially when formatting multiple parts of a date.

Java

// Example using printf

import java.util.Date;

Date date = new Date();

System.out.printf("Current Date/Time: %tc", date);

**Comparing and Parsing Dates**

* **Comparison**: Dates can be compared using the before(), after(), and equals() methods or by comparing their millisecond values obtained with getTime().
* **Parsing**: The SimpleDateFormat class can also **parse** a string and convert it into a Date object using its parse() method, provided the string's format matches the one defined by the SimpleDateFormat object.

**Measuring Time**

You can measure the elapsed time of a process using System.currentTimeMillis(). This method returns the current time in milliseconds since January 1, 1970, allowing you to calculate the difference between a start and end time.

Java

long start = System.currentTimeMillis();

// code to be timed

long end = System.currentTimeMillis();

long duration = end - start;

**The GregorianCalendar Class**

The GregorianCalendar class is a concrete implementation of the abstract Calendar class. It provides a more robust and flexible way to manipulate dates than the Date class. You can use it to perform date-specific calculations, such as adding a certain number of days or checking for a leap year. GregorianCalendar objects are initialized with the current date and time in the default locale and time zone.

**Java Operators**

In Java, operators are special symbols that perform specific operations on variables and values. They are categorized based on the type of operation they perform.

**Arithmetic Operators 🔢**

Arithmetic operators are used to perform mathematical calculations like addition, subtraction, multiplication, division, and modulus. They are the same as those used in algebra.

* + (Addition)
* - (Subtraction)
* \* (Multiplication)
* / (Division)
* % (Modulus/Remainder)
* ++ (Increment)
* -- (Decrement)

Java

int a = 10, b = 5;

System.out.println(a + b); // Output: 15

**Assignment Operators ✅**

Assignment operators are used to assign a value to a variable. The most common one is =, but Java also provides compound assignment operators that combine an arithmetic operation with an assignment (e.g., +=, -=).

* = (Simple assignment)
* += (Add and assign)
* \*= (Multiply and assign)
* /= (Divide and assign)
* %= (Modulus and assign)
* etc.

Java

int x = 10;

x += 5; // Same as x = x + 5;

System.out.println(x); // Output: 15

**Relational Operators ⚖️**

Relational operators are used to compare two values and return a **boolean** result (true or false). They are commonly used in conditional statements.

* == (Equal to)
* != (Not equal to)
* > (Greater than)
* < (Less than)
* >= (Greater than or equal to)
* <= (Less than or equal to)

Java

int a = 10, b = 5;

System.out.println(a > b); // Output: true

**Logical Operators 🧠**

Logical operators are used to perform logical operations on boolean values. They are essential for building complex conditions.

* && (Logical AND): Returns true if both operands are true.
* || (Logical OR): Returns true if at least one operand is true.
* ! (Logical NOT): Reverses the boolean state of the operand.

Java

boolean isJavaFun = true;

boolean isCodingEasy = false;

System.out.println(isJavaFun && isCodingEasy); // Output: false

**Bitwise Operators ⚙️**

Bitwise operators perform operations on individual bits of an integer or character. They are often used in low-level programming for efficiency.

* & (Bitwise AND)
* | (Bitwise OR)
* ^ (Bitwise XOR)
* ~ (Bitwise Complement)
* << (Left shift)
* >> (Right shift)
* >>> (Zero-fill right shift)

**Miscellaneous Operators ❓**

Java includes other specialized operators, such as:

* **Conditional (Ternary) Operator**: A shorthand for an if-else statement. It has the syntax (condition) ? value\_if\_true : value\_if\_false.

Java

int score = 85;

String result = (score > 80) ? "Pass" : "Fail";

* **instanceof Operator**: Used to check if an object is an instance of a particular class or interface. It returns a boolean value.

Java

String str = "hello";

boolean isString = str instanceof String; // Returns true

**Operator Precedence and Associativity 🪜**

**Operator precedence** determines the order in which operators are evaluated in an expression. For example, multiplication (\*) has a higher precedence than addition (+), so it is performed first. **Associativity** determines the order of evaluation when operators have the same precedence, usually from left to right. Parentheses () can be used to override precedence rules.

**Java Arithmetic Operators 🧮**

Java arithmetic operators are symbols used to perform basic mathematical calculations on numeric data types such as int, float, double, and long. They are fundamental for performing computations in programming. These operators adhere to standard mathematical precedence rules, meaning multiplication and division operations are executed before addition and subtraction.

**List of Java Arithmetic Operators**

Here's a list of the arithmetic operators available in Java:

| Operator | Description | Example (A=10, B=20) |
| --- | --- | --- |
| + | **Addition**: Adds two values. | A + B returns 30 |
| - | **Subtraction**: Subtracts the right operand from the left operand. | A - B returns -10 |
| \* | **Multiplication**: Multiplies two values. | A \* B returns 200 |
| / | **Division**: Divides the left operand by the right operand. | B / A returns 2 |
| % | **Modulus**: Returns the remainder of the division. | B % A returns 0 |
| ++ | **Increment**: Increases the value of a variable by 1. Can be pre-increment (++A) or post-increment (A++). | A++ makes A 11 |
| -- | **Decrement**: Decreases the value of a variable by 1. Can be pre-decrement (--A) or post-decrement (A--). | A-- makes A 9 |

**Examples of Arithmetic Operators**

Let's illustrate these operators with some simple Java code.

**Basic Operations**

This example demonstrates addition, subtraction, multiplication, and division.

Java

public class BasicArithmetic {

public static void main(String[] args) {

int a = 10;

int b = 20;

System.out.println("a + b = " + (a + b)); // Output: a + b = 30

System.out.println("a - b = " + (a - b)); // Output: a - b = -10

System.out.println("a \* b = " + (a \* b)); // Output: a \* b = 200

System.out.println("b / a = " + (b / a)); // Output: b / a = 2

}

}

**Modulus Operator**

This example shows how the modulus operator (%) calculates the remainder of a division.

Java

public class ModulusExample {

public static void main(String[] args) {

int a = 10;

int b = 20;

int c = 25;

System.out.println("b % a = " + (b % a)); // Output: b % a = 0 (20 divided by 10 has no remainder)

System.out.println("c % a = " + (c % a)); // Output: c % a = 5 (25 divided by 10 has a remainder of 5)

}

}

**Increment and Decrement Operators**

This example highlights the difference between pre-increment/decrement and post-increment/decrement.

Java

public class IncDecExample {

public static void main(String[] args) {

int a = 10;

int d = 25;

System.out.println("a++ = " + (a++)); // Output: a++ = 10 (a is used, then incremented to 11)

System.out.println("a (after a++) = " + a); // Output: a (after a++) = 11

System.out.println("d++ = " + (d++)); // Output: d++ = 25 (d is used, then incremented to 26)

System.out.println("++d = " + (++d)); // Output: ++d = 27 (d is incremented to 27, then used)

}

}

**Java Assignment Operators ➡️**

Java assignment operators are used to assign a value to a variable. The most basic one is the equals sign (=), which takes the value from the right-hand side and stores it in the variable on the left-hand side. However, Java also provides **compound assignment operators** as a shorthand for common operations.

**Types of Assignment Operators**

| Operator | Description | Example |
| --- | --- | --- |
| = | **Simple Assignment**: Assigns the value of the right operand to the left operand. | c = a + b; |
| += | **Add and Assign**: Adds the right operand to the left operand and assigns the result to the left operand. | c += a; is the same as c = c + a; |
| -= | **Subtract and Assign**: Subtracts the right operand from the left operand and assigns the result. | c -= a; is the same as c = c - a; |
| \*= | **Multiply and Assign**: Multiplies the left operand by the right operand and assigns the result. | c \*= a; is the same as c = c \* a; |
| /= | **Divide and Assign**: Divides the left operand by the right operand and assigns the result. | c /= a; is the same as c = c / a; |
| %= | **Modulus and Assign**: Calculates the modulus and assigns the remainder. | c %= a; is the same as c = c % a; |
| <<= | **Left Shift and Assign**: Shifts bits left and assigns the result. | c <<= 2; is the same as c = c << 2; |
| >>= | **Right Shift and Assign**: Shifts bits right and assigns the result. | c >>= 2; is the same as c = c >> 2; |
| &= | **Bitwise AND and Assign**: Performs a bitwise AND and assigns the result. | c &= a; is the same as c = c & a; |
| ^= | **Bitwise XOR and Assign**: Performs a bitwise XOR and assigns the result. | c ^= a; is the same as c = c ^ a; |
| ` | =` | **Bitwise OR and Assign**: Performs a bitwise OR and assigns the result. |

**Examples**

**Simple and Compound Arithmetic Assignment**

This example demonstrates how basic arithmetic assignment operators work.

Java

int a = 10, b = 20, c = 0;

c = a + b; // c is now 30

c += a; // c = c + a, so c is 30 + 10 = 40

c -= a; // c = c - a, so c is 40 - 10 = 30

c \*= a; // c = c \* a, so c is 30 \* 10 = 300

**Compound Bitwise and Modulus Assignment**

This example shows how compound assignment works with other operations.

Java

int a = 10;

int c = 15;

c /= a; // c = 15 / 10, so c becomes 1 (integer division)

c = 15;

c %= a; // c = 15 % 10, so c becomes 5 (remainder)

c = 15;

c &= a; // c = 15 & 10 (binary AND), so c becomes 10

c = 15;

c ^= a; // c = 15 ^ 10 (binary XOR), so c becomes 5

c = 15;

c |= a; // c = 15 | 10 (binary OR), so c becomes 15

**Java Relational Operators ⚖️**

Relational operators in Java are used to compare two values. They return a **boolean** result (true or false), making them essential for decision-making within programs, particularly in if statements and loops.

**List of Relational Operators**

| Operator | Description | Example (A=10, B=20) |
| --- | --- | --- |
| == | **Equal to**: Checks if the values of two operands are equal. | A == B returns false |
| != | **Not equal to**: Checks if the values are not equal. | A != B returns true |
| > | **Greater than**: Checks if the left operand is greater than the right. | A > B returns false |
| < | **Less than**: Checks if the left operand is less than the right. | A < B returns true |
| >= | **Greater than or equal to**: Checks if the left operand is greater than or equal to the right. | A >= B returns false |
| <= | **Less than or equal to**: Checks if the left operand is less than or equal to the right. | A <= B returns true |

**Examples of Relational Operators**

**Equality and Inequality**

This example demonstrates the use of the == and != operators.

Java

public class EqualityTest {

public static void main(String args[]) {

int a = 10;

int b = 20;

System.out.println("a == b = " + (a == b) ); // Output: a == b = false

System.out.println("a != b = " + (a != b) ); // Output: a != b = true

}

}

**Greater Than and Less Than**

This example shows how to compare two values using > and <.

Java

public class ComparisonTest {

public static void main(String args[]) {

int a = 10;

int b = 20;

System.out.println("a > b = " + (a > b) ); // Output: a > b = false

System.out.println("a < b = " + (a < b) ); // Output: a < b = true

}

}

**Greater Than/Less Than or Equal To**

This example demonstrates the use of >= and <=.

Java

public class InclusiveComparisonTest {

public static void main(String args[]) {

int a = 10;

int b = 20;

System.out.println("b >= a = " + (b >= a) ); // Output: b >= a = true

System.out.println("b <= a = " + (b <= a) ); // Output: b <= a = false

}

}

**Java Logical Operators 🧠**

Java logical operators are used to perform logical operations on **boolean** values (true or false). These operators are essential for creating complex conditions in decision-making statements like if conditions and loops, allowing you to control the flow of your program.

**List of Logical Operators**

| Operator | Description | Example (A=true, B=false) |
| --- | --- | --- |
| **&&** | **Logical AND**: Returns true only if **both** operands are true. If the first operand is false, the second one isn't evaluated (short-circuiting). | A && B returns false |
| \*\*` |  | `\*\* |
| **!** | **Logical NOT**: Reverses the logical state of the operand. It's a unary operator. | !A returns false and !B returns true |

**Examples**

**Logical AND (&&)**

The && operator checks if two conditions are simultaneously true.

Java

public class LogicalAndExample {

public static void main(String[] args) {

boolean a = true;

boolean b = false;

System.out.println("a && b = " + (a && b)); // Output: a && b = false

}

}

**Logical OR (||)**

The || operator returns true if either of the conditions is met.

Java

public class LogicalOrExample {

public static void main(String[] args) {

boolean a = true;

boolean b = false;

System.out.println("a || b = " + (a || b)); // Output: a || b = true

}

}

**Logical NOT (!)**

The ! operator negates a boolean value.

Java

public class LogicalNotExample {

public static void main(String[] args) {

boolean a = true;

boolean b = false;

System.out.println("!a = " + !a); // Output: !a = false

System.out.println("!b = " + !b); // Output: !b = true

}

}

Java's bitwise operators perform operations directly on the individual bits of integer types (long, int, short, char, and byte). They are used in low-level programming for tasks like data compression and encryption because they are highly efficient.

**List of Java Bitwise Operators**

| Operator | Description | Example (A=60, B=13) |
| --- | --- | --- |
| & | **Bitwise AND**: Compares two bits and returns a 1 only if **both** bits are 1. | 60 & 13 results in 12 (0000 1100) |
| ` | ` | **Bitwise OR**: Compares two bits and returns a 1 if **either** bit is 1. |
| ^ | **Bitwise XOR**: Compares two bits and returns a 1 if the bits are **different**. | 60 ^ 13 results in 49 (0011 0001) |
| ~ | **Bitwise Complement**: Flips the bits of a single operand from 0 to 1 and 1 to 0. | ~60 results in -61 (due to two's complement representation) |
| << | **Left Shift**: Shifts the bits of the left operand to the left by the number of places specified by the right operand. Zeroes are added to the right. | 60 << 2 results in 240 (1111 0000) |
| >> | **Signed Right Shift**: Shifts the bits of the left operand to the right. The sign bit (the leftmost bit) is copied to fill the empty spaces. | 60 >> 2 results in 15 (0000 1111) |
| >>> | **Unsigned Right Shift**: Shifts the bits of the left operand to the right. Zeroes are used to fill the empty spaces, regardless of the original sign. | 60 >>> 2 results in 15 (0000 1111) |

**Examples**

**Bitwise AND (&) and OR (|)**

This example shows how & and | operate on the binary representations of numbers.

Java

int a = 60; // 0011 1100

int b = 13; // 0000 1101

int c;

c = a & b; // 0000 1100 (binary) -> 12 (decimal)

System.out.println("a & b = " + c);

c = a | b; // 0011 1101 (binary) -> 61 (decimal)

System.out.println("a | b = " + c);

**Bitwise XOR (^) and Complement (~)**

This example demonstrates the bitwise XOR and complement operators.

Java

int a = 60; // 0011 1100

int b = 13; // 0000 1101

int c;

c = a ^ b; // 0011 0001 (binary) -> 49 (decimal)

System.out.println("a ^ b = " + c);

c = ~a; // 1100 0011 (binary) -> -61 (decimal)

System.out.println("~a = " + c);

**Shift Operators (<<, >>, >>>)**

These operators efficiently multiply or divide a number by powers of two.

Java

int a = 60; // 0011 1100

int c;

c = a << 2; // Shift bits left by 2: 1111 0000 -> 240

System.out.println("a << 2 = " + c);

c = a >> 2; // Shift bits right by 2 (signed): 0000 1111 -> 15

System.out.println("a >> 2 = " + c);

c = a >>> 2; // Shift bits right by 2 (unsigned): 0000 1111 -> 15

System.out.println("a >>> 2 = " + c);

In Java, **operator precedence** determines the order in which operators in an expression are evaluated. Operators with higher precedence are executed before those with lower precedence. When multiple operators in an expression have the same precedence, **associativity** determines the direction of evaluation, typically from left to right.

**How Precedence and Associativity Work**

**Operator Precedence 📈**

Just like in mathematics, Java has a specific order for evaluating operators. For example, multiplication and division have a higher precedence than addition and subtraction. This means that in an expression like 5 + 2 \* 3, the multiplication 2 \* 3 is performed first, resulting in 6, and then the addition 5 + 6 is calculated, giving a final result of 11. Parentheses () can be used to override these rules, forcing a part of the expression to be evaluated first.

**Operator Associativity 🔄**

When you have multiple operators with the same precedence, associativity determines the direction of evaluation. For instance, the multiplicative operators \*, /, and % all have the same precedence and are **left-to-right associative**. This means an expression like 10 / 5 \* 2 is evaluated as (10 / 5) \* 2, which equals 4.

However, the assignment operator = is **right-to-left associative**. This allows for chained assignments like a = b = 5. Here, b = 5 is evaluated first, and its result is then assigned to a.

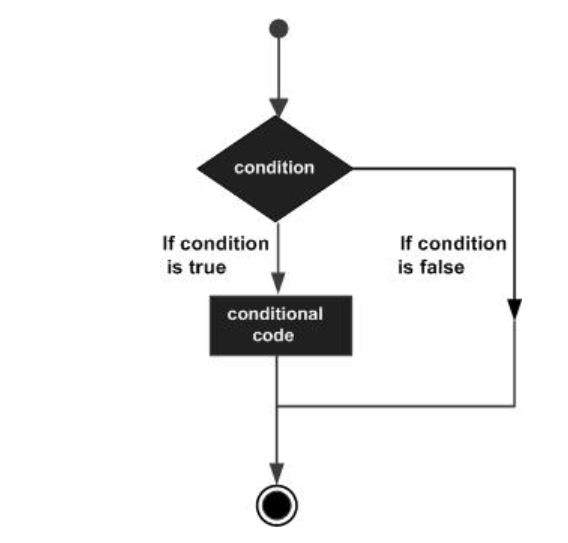
**Operator Precedence and Associativity Table**

This table summarizes the hierarchy of Java operators from highest to lowest precedence.

| Precedence | Category | Operator | Associativity |
| --- | --- | --- | --- |
| 1 | Postfix | (), [], . | Left to right |
| 2 | Unary | ++, --, +, -, ~, ! | Right to left |
| 3 | Multiplicative | \*, /, % | Left to right |
| 4 | Additive | +, - | Left to right |
| 5 | Shift | <<, >>, >>> | Left to right |
| 6 | Relational | <, >, <=, >=, instanceof | Left to right |
| 7 | Equality | ==, != | Left to right |
| 8 | Bitwise AND | & | Left to right |
| 9 | Bitwise XOR | ^ | Left to right |
| 10 | Bitwise OR | ` | ` |
| 11 | Logical AND | && | Left to right |
| 12 | Logical OR | ` |  |
| 13 | Conditional | ? : | Right to left |
| 14 | Assignment | =, +=, -=, etc. | Right to left |

**Java Control Statements**

A decision-making structure in Java is a control flow statement that allows a program to execute specific blocks of code based on whether a given condition is true or false. These structures, also known as conditional statements, enable the program to make decisions and follow different paths of execution.



**Types of Decision-Making Statements**

Java offers several types of decision-making structures to handle different scenarios:

**if statement**

The **if statement** is the most basic decision-making structure. It executes a block of code only if a boolean condition evaluates to true.

**if...else statement**

The **if...else statement** provides an alternative path of execution. The code block within the if is executed if the condition is true, while the code block within the else is executed if the condition is false.

**nested if statement**

A **nested if statement** is an if or else if statement placed inside another if or else if block. This allows for more complex, multi-layered conditions.

**switch statement**

A **switch statement** provides a way to test a variable for equality against a list of values. It is often a more efficient and readable alternative to a long chain of if...else if statements.

**The Ternary Operator (? :)**

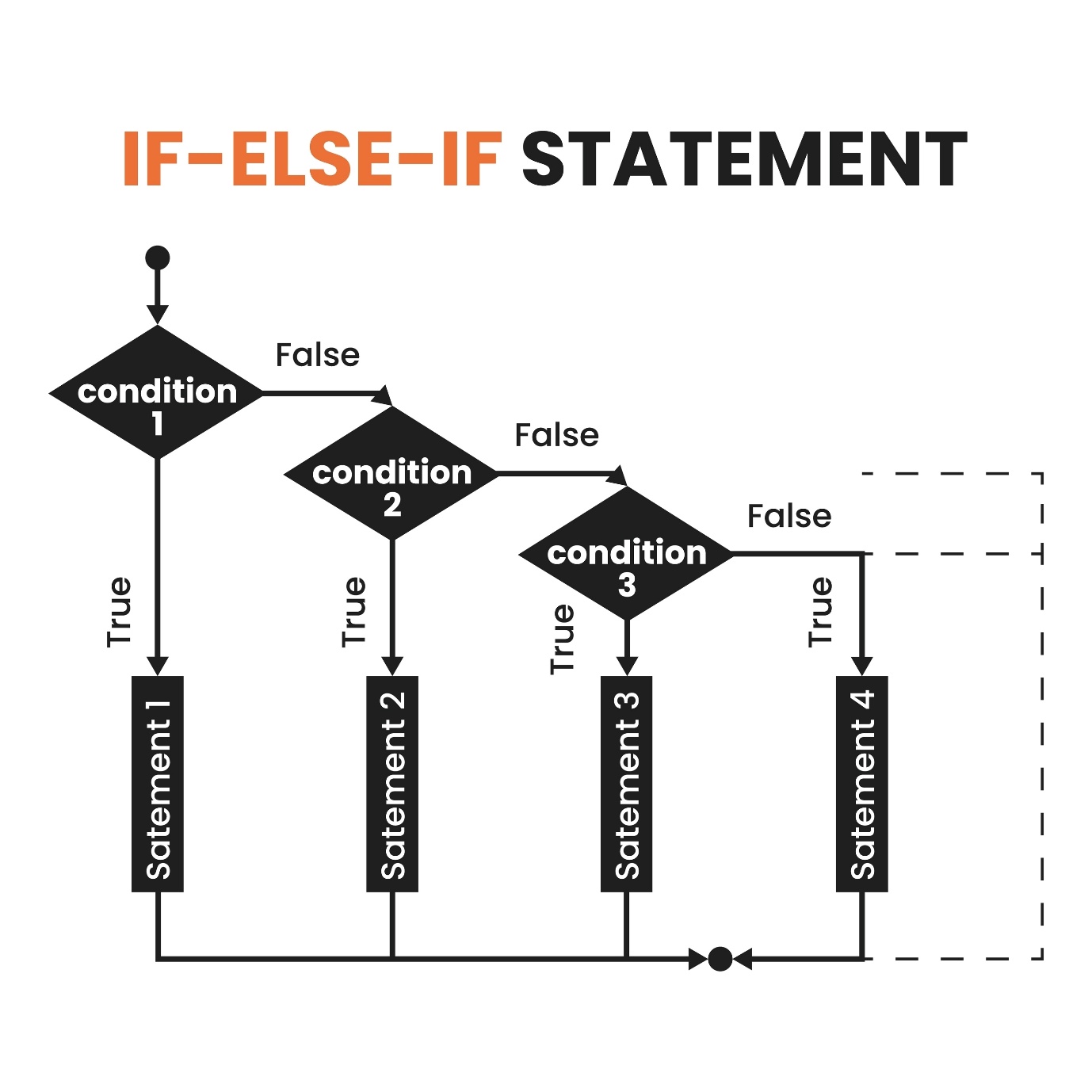
The **ternary operator** is a shorthand for a simple if...else statement. It's a conditional operator that takes three operands: a boolean expression, a value to return if the expression is true, and a value to return if it's false.

**Syntax:** Exp1 ? Exp2 : Exp3;

* **Exp1**: The boolean expression to be evaluated.
* **Exp2**: The value to be returned if Exp1 is true.
* **Exp3**: The value to be returned if Exp1 is false.

For example, the code int b = (a == 1) ? 20 : 30; checks if a is equal to 1. If it is, b is assigned the value 20; otherwise, it's assigned 30.

In Java, **if statements** are conditional statements that control program flow by executing a code block only when a specified boolean condition is true. The if...else statement allows for two-way branching: one code block executes if the condition is true, and another if it's false.



Licensed by Google

**The if Statement**

The simplest form of decision-making, the if statement, tests a single boolean expression. If the expression is true, the code within its block is executed. If it's false, the program simply skips the block.

**Syntax:**

Java

if (boolean\_expression) {

// Code to execute if the expression is true

}

**The if...else Statement**

The if...else statement provides a clear alternative path. If the initial condition is true, the if block runs. If the condition is false, the else block runs instead. This ensures that one of the two code blocks will always be executed.

**Syntax:**

Java

if (boolean\_expression) {

// Code for true condition

} else {

// Code for false condition

}

**Example:** If x is 30, the condition x < 20 is false, so the else block is executed.

Java

int x = 30;

if (x < 20) {

System.out.println("This is if statement");

} else {

System.out.println("This is else statement"); // Output

}

**The if...else if...else Ladder**

This structure, often called a ladder, is used for checking multiple conditions sequentially. Java checks each else if condition in order, and the first one that evaluates to true is executed. Once a condition is met, the rest of the ladder, including the final else, is skipped. The final else block is optional and serves as a default case if none of the preceding conditions are met.

**Syntax:**

Java

if (condition1) {

// Code for condition1

} else if (condition2) {

// Code for condition2

} else {

// Default code if no conditions are met

}

**Example:** If x is 30, the first two conditions (x == 10 and x == 20) are false. The third condition (x == 30) is true, so its code block is executed, and the final else is ignored.

Java

int x = 30;

if (x == 10) {

// ...

} else if (x == 20) {

// ...

} else if (x == 30) {

System.out.println("Value of X is 30"); // Output

} else {

// ...

}

**The nested if...else Statement**

A nested if...else statement is one where an if or else block contains another full if...else structure. This is useful for evaluating conditions that depend on a previous condition being true. It allows for more complex, multi-level decision-making.

**Example:** To find the largest of three numbers, a nested if...else can first compare x and y, then compare the larger of the two with z.

Java

int x = 10, y = 20, z = 30;

if (x >= y) {

if (x >= z)

System.out.println(x + " is the largest.");

else

System.out.println(z + " is the largest.");

} else {

if (y >= z)

System.out.println(y + " is the largest.");

else

System.out.println(z + " is the largest."); // Output

In Java, a **switch statement** provides a way to execute a code block based on a variable's value. It tests a variable for equality against a list of constant case values and then jumps to the first matching case. It's often a more readable alternative to a long chain of if-else if statements.

A diagram of a code block

AI-generated content may be incorrect.

**Syntax and Rules 📜**

The switch statement evaluates an expression and matches its value to one of the case values.

**Syntax:**

Java

switch(expression) {

case value1:

// statements

break;

case value2:

// statements

break;

default:

// statements (optional)

}

**Rules for Using switch**

* The expression can be an **int**, **byte**, **short**, **char**, **String**, or an **enum**.
* Each case value must be a **literal** or a **constant** and must be of the same data type as the expression.
* The break statement is crucial. Without it, the code will **fall through** to the next case block, executing all subsequent statements until a break or the end of the switch is reached.
* The **default** case is optional. It is executed if none of the case values match the expression. No break is needed after the default case.

<br>

**Examples 💻**

**Example 1: char and Fall-Through**

This example demonstrates how a switch statement can handle character values and how **fall-through** works. For a grade of 'C', the code falls through from case 'B' to case 'C', executing the "Well done" message. The break then prevents further execution.

Java

public class Test {

public static void main(String args[]) {

char grade = 'C';

switch(grade) {

case 'A' :

System.out.println("Excellent!");

break;

case 'B' :

case 'C' : // No break, so 'B' falls through to 'C'

System.out.println("Well done");

break;

default :

System.out.println("Invalid grade");

}

}

}

**Output:**

Well done

**Example 2: The default Keyword**

This example illustrates the use of the **default** case. If the month variable has a value that doesn't match any of the defined case statements, the code within the default block is executed.

Java

public class SwitchWithDefault {

public static void main(String[] args) {

int month = 13; // Invalid month

switch (month) {

case 1:

System.out.println("January");

break;

case 12:

System.out.println("December");

break;

default:

System.out.println("Invalid month"); // Executed because 13 is not a case

}

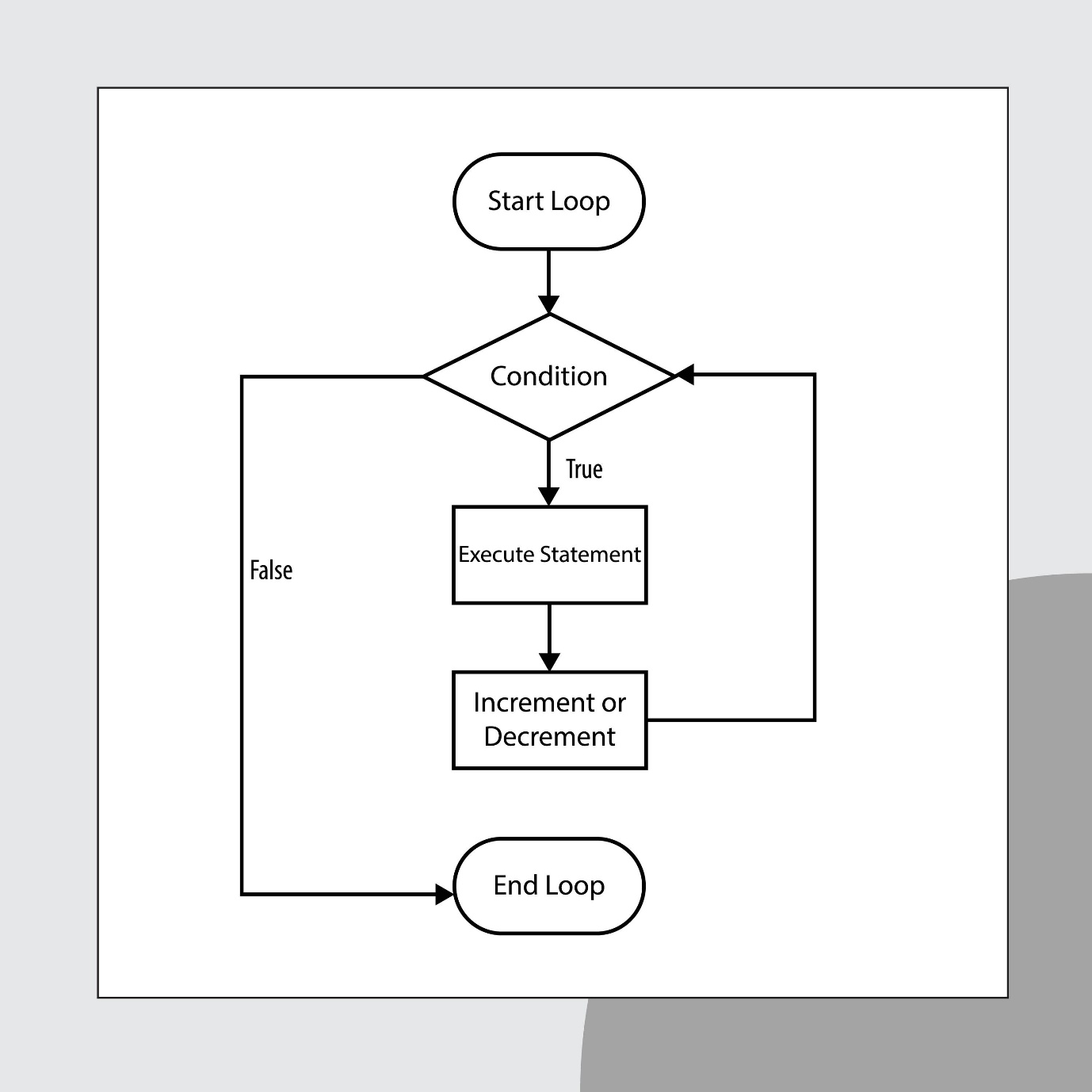
}

}

**Output:**

Invalid month

You need loops in programming when you want to execute a block of code multiple times. Instead of writing the same code over and over, a loop automates this repetition, making your code shorter, more organized, and easier to manage. Loops are a fundamental part of a program's control flow, allowing it to perform repetitive tasks efficiently.



Licensed by Google

**Types of Loops in Java**

Java provides several types of loops to handle different scenarios:

* **while loop**: This loop repeats a block of code as long as a condition is true. It **checks the condition before** executing the code, so the loop body might not run at all if the condition is initially false.
* **for loop**: This is a more structured loop often used when you know exactly how many times you want to iterate. It combines the initialization, condition, and increment/decrement of the loop counter into a single line.
* **do...while loop**: Similar to a while loop, but it **checks the condition at the end** of the loop. This guarantees that the loop body will always execute at least once, regardless of the initial condition.
* **Enhanced for loop**: Introduced in Java 5, this loop is designed for iterating over collections and arrays. It simplifies the syntax, making it easier to read and write code for traversing elements.

**Loop Control Statements**

These statements are used to alter the normal flow of a loop:

* **break statement**: This statement is used to **immediately terminate** a loop or a switch statement. When break is encountered, the program's control flow jumps to the statement right after the loop.
* **continue statement**: This statement **skips the rest of the current iteration** of a loop and immediately moves to the next one. The loop's condition is re-evaluated, and the loop continues from there.

A **for loop** in Java is a control structure used to execute a block of code a specific number of times. It's an **entry-control loop**, meaning it checks its condition before each iteration. Its concise syntax makes it an efficient and readable way to automate repetitive tasks, especially when the number of iterations is known.

**Structure of a for Loop**

A for loop has three essential components within its parentheses, separated by semicolons:

* **Initialization**: This part runs only once at the beginning. It's used to declare and initialize a loop counter variable.
* **Boolean Expression (Condition)**: Evaluated before each iteration. If true, the loop continues; if false, it terminates.
* **Update**: Executed at the end of each iteration. It typically modifies the loop counter (e.g., increments or decrements it).

The general syntax is:

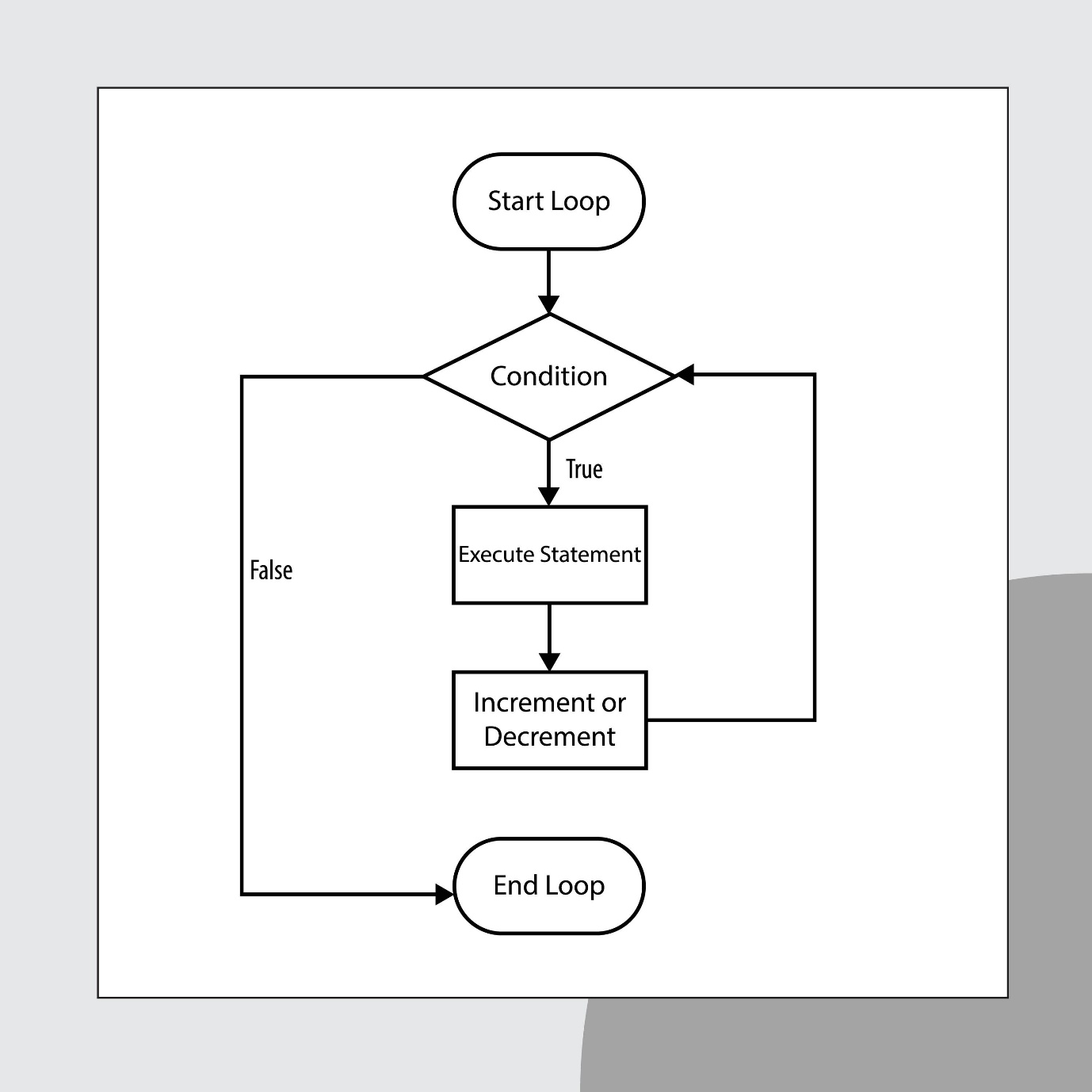
Java

for (initialization; condition; update) {

// Code to be executed repeatedly

}

For example, to print numbers from 1 to 5:



Licensed by Google

Java

for (int i = 1; i <= 5; i++) {

System.out.println("Number: " + i);

}

**Types of for Loops**

1. **Standard for Loop**: The most common type, used for iterating a fixed number of times.
2. **Enhanced for Loop (for-each)**: A simplified version for iterating over elements in arrays or collections. It abstracts away the index management.

Java

String[] fruits = {"Apple", "Banana", "Cherry"};

for(String fruit : fruits) {

System.out.println(fruit);

}

1. **Nested for Loops**: A for loop placed inside another for loop. This is useful for working with multi-dimensional data, like matrices. The inner loop completes all its iterations for each single iteration of the outer loop.
2. **Infinite for Loop**: Created by providing a condition that is always true or by leaving all three sections in the parentheses blank (for(;;)). It runs indefinitely until a break statement is encountered or the program is manually terminated.

The **for-each loop** is a special type of loop in Java, introduced in Java 5, designed for **iterating over elements of arrays and collections**. It provides a simplified, more readable syntax compared to a traditional for loop, as it abstracts away the need to manage an index counter.

**Syntax and Execution 🔄**

The for-each loop simplifies traversing arrays and collections.

**Syntax:**

Java

for (declaration : expression) {

// Statements to be executed for each element

}

* **declaration**: Declares a variable of a type compatible with the elements in the array or collection. This variable will hold the current element during each iteration.
* **expression**: The array or collection that you want to loop through.

The loop automatically iterates over each element in the expression, assigning the value of the current element to the declaration variable for each iteration. The loop terminates once all elements have been visited.

**Examples 💻**

**1. Iterating Over a List**

This example demonstrates how to use a for-each loop to print each element from a list of integers.

Java

import java.util.Arrays;

import java.util.List;

public class Test {

public static void main(String args[]) {

List<Integer> numbers = Arrays.asList(10, 20, 30, 40, 50);

for (Integer x : numbers) {

System.out.print(x + ", ");

}

}

}

**Output:** 10, 20, 30, 40, 50,

**2. Iterating Over an Array of Objects**

The for-each loop also works seamlessly with arrays of objects, making it easy to access each object in the collection.

Java

public class Test {

public static void main(String args[]) {

Student[] students = {new Student(1, "Julie"), new Student(3, "Adam")};

for (Student student : students) {

System.out.print(student + ",");

}

}

}

// Assume Student class is defined elsewhere

class Student {

int rollNo;

String name;

Student(int rollNo, String name){

this.rollNo = rollNo;

this.name = name;

}

@Override

public String toString() {

return "[ " + this.rollNo + ", " + this.name + " ]";

}

}

**Output:** [ 1, Julie ],[ 3, Adam ],

A **while loop** in Java repeatedly executes a block of code as long as a given boolean condition is true. It's an **entry-control loop**, meaning it checks the condition at the beginning of each iteration. If the condition is initially false, the loop body will never execute.

A diagram of a code

AI-generated content may be incorrect.

**Syntax and Execution 🔄**

The while loop has a simple structure: a condition that's checked before the loop's body runs.

**Syntax:**

Java

while (Boolean\_expression) {

// Statements to be executed repeatedly

}

* The Boolean\_expression is evaluated.
* If the result is true, the code inside the loop's body is executed.
* After the body runs, the Boolean\_expression is re-evaluated. This cycle continues until the expression becomes false.
* When the condition is false, the program's control flow moves to the statement immediately following the loop.

**Examples 💻**

**1. Counting with a while Loop**

This example shows how to use a while loop to print numbers within a range.

Java

public class Test {

public static void main(String args[]) {

int x = 10;

while( x < 20 ) {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

}

}

}

**Output:**

value of x : 10

value of x : 11

...

value of x : 19

The loop starts with x = 10. The condition x < 20 is true, so the code runs, and x is incremented. This continues until x becomes 20, at which point the condition is false, and the loop terminates.

**2. Iterating an Array**

A while loop can also be used to traverse an array, as long as you manually manage an index counter.

Java

public class Test {

public static void main(String args[]) {

int [] numbers = {10, 20, 30, 40, 50};

int index = 0;

while( index < 5 ) {

System.out.print("value of item : " + numbers[index] );

index++;

System.out.print("\n");

}

}

}

**Output:**

value of item : 10

value of item : 20

...

value of item : 50

This loop continues as long as index is less than 5 (the array size). The index is incremented in each iteration to access the next element.

**3. Infinite while Loop**

An infinite loop can be created by providing a condition that never becomes false, such as a boolean literal.

Java

while(true) {

// This loop runs forever

}

You would typically use a break statement inside an infinite loop to terminate it based on some condition.

A **do-while loop** in Java is a repetition control structure that's similar to a while loop, but with one key difference: it's an **exit-control loop** that checks its condition at the end of each iteration. This guarantees that the loop's body will **execute at least once**, even if the condition is initially false.

A diagram of a code

AI-generated content may be incorrect.

**Syntax and Execution 🔄**

The do-while loop first executes its code block and then evaluates the condition.

**Syntax:**

Java

do {

// Statements to be executed repeatedly

} while (Boolean\_expression);

* The code inside the do block is executed first.
* After the code runs, the Boolean\_expression is evaluated.
* If the condition is true, control jumps back to the do statement, and the loop repeats.
* If the condition is false, the loop terminates, and the program moves to the next statement.

**Examples 💻**

**1. Printing a Range of Numbers**

This example demonstrates a basic do-while loop that prints numbers from 10 to 19.

Java

public class Test {

public static void main(String args[]) {

int x = 10;

do {

System.out.print("value of x : " + x );

x++;

System.out.print("\n");

} while( x < 20 );

}

}

**Output:**

value of x : 10

...

value of x : 19

The loop body executes first, printing 10, then the condition x < 20 is checked. Since it's true, the loop continues. This repeats until x is 20, at which point the condition becomes false, and the loop ends.

**2. Guaranteed Execution**

This example highlights the core advantage of a do-while loop. The code inside the loop will run once, even though the condition is initially false.

Java

public class GuaranteedExecution {

public static void main(String args[]) {

int x = 25; // x is already not less than 20

do {

System.out.println("This runs at least once.");

x++;

} while(x < 20);

}

}

**Output:**

This runs at least once.

Here, the loop body executes once, printing the message. Then, the condition x < 20 is checked with x as 26, which is false. The loop then terminates.

The Java **break statement** is a control flow statement used to terminate a loop or a switch statement immediately. When a break is executed, the program's control flow exits the innermost loop or switch block and continues with the next statement after it.

A diagram of a break

AI-generated content may be incorrect.

**Using break with Loops ➡️**

The break statement provides a way to exit a loop prematurely, regardless of the loop's condition. This is particularly useful when you've found the item you're looking for or an error condition has occurred.

**Example with for Loop**

In this example, the loop is designed to iterate through an array. However, the break statement terminates the loop as soon as the value 30 is encountered, preventing subsequent elements from being processed.

Java

int [] numbers = {10, 20, 30, 40, 50};

for(int index = 0; index < numbers.length; index++) {

if(numbers[index] == 30){

break; // Loop terminates here

}

System.out.println("value of item : " + numbers[index] );

}

**Output:**

value of item : 10

value of item : 20

**Example with while Loop**

The break statement can also be used to exit a while loop, including an infinite loop, based on a specific condition.

Java

int x = 10;

while(true) { // An infinite loop

System.out.print("value of x : " + x );

if(x == 14) {

break; // Loop terminates when x becomes 14

}

x++;

}

**Output:**

value of x : 10

value of x : 11

value of x : 12

value of x : 13

value of x : 14

**Using break with switch Statements 💡**

In a switch statement, the break keyword prevents **"fall-through"**. Without it, once a case is matched, the program would execute the code for that case and then continue to the next cases, ignoring their conditions.

**Example with switch**

Here, the break statements ensure that only the code block for the matching case is executed.

Java

int day = 3;

switch (day) {

case 1:

System.out.println("Monday");

break; // Exits the switch block

case 2:

System.out.println("Tuesday");

break;

case 3:

System.out.println("Wednesday"); // This code is executed

break; // Exits the switch block

case 4:

System.out.println("Thursday");

break;

default:

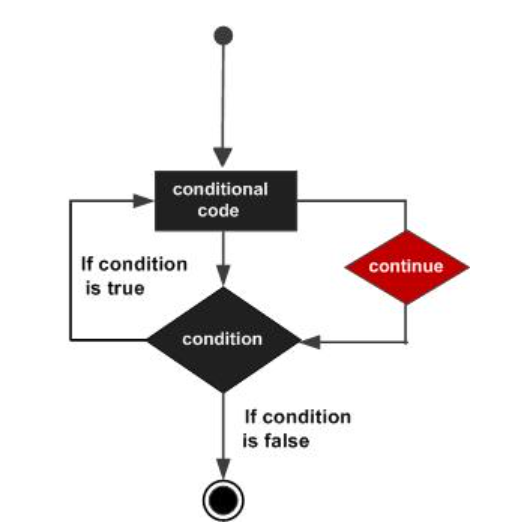
System.out.println("Invalid day");

}

**Output:**

Wednesday

The **continue statement** in Java is a control flow statement used inside loops to skip the current iteration and proceed to the next one. When continue is encountered, the rest of the code in the loop body for that specific iteration is ignored.



**Using continue with Loops ➡️**

* **for loop**: continue skips the rest of the loop body and jumps directly to the **update** expression.
* **while and do-while loops**: continue skips the rest of the loop body and jumps directly to the **boolean expression** for re-evaluation.

**Example with a for loop**

This example shows how continue is used to skip a specific element in an array. The loop is designed to print each number, but when it encounters the value 30, the continue statement is executed, and the println statement is skipped for that iteration. The loop then proceeds to the next element.

Java

int [] numbers = {10, 20, 30, 40, 50};

for(int index = 0; index < numbers.length; index++) {

if(numbers[index] == 30){

continue; // Skips the rest of this iteration

}

System.out.println("value of item : " + numbers[index] );

}

**Output:**

value of item : 10

value of item : 20

value of item : 40

value of item : 50

**Common Use Cases 💡**

The continue statement is useful for filtering data or handling exceptions within a loop without breaking out of it entirely.

1. **Skipping values**: You can use continue to filter out specific values while processing a range.
2. **Input validation**: It's a great way to handle invalid user input in a loop. If a user enters bad data, you can print an error message and continue to the next iteration to ask for input again.
3. **Nested loops**: In nested loops, continue only affects the innermost loop, allowing the outer loop to continue its normal execution.
4. **Data processing**: When reading data from a file or network stream, you can use continue to skip over empty lines or corrupted data records.