Java Programming Essentials - Day 1

Jansen Ang

Get to know

- What is your name?
- What is your background?
- Why did you want to learn about Java?
- What are your expectations for this training?

Jansen Marson Ang

- Java Developer with over 5 years of experience in the industry, specializing in Java technologies like Java 17 and Spring Boot.
- Has worked for various companies and industries, including IBM, Oracle, and Maya
- One of the Leaders of the Java User Group Philippines.
- In his free time, he organizes and speaks at community events, contributes to open-source libraries, and enjoys playing badminton and swimming.

Java User Group Philippines

Java User Group Philippines is a group for all Java advocates and enthusiasts. Members can talk about Java, JVM, and the latest frameworks within the Java ecosystem. Everyone is welcome to join their events whether one is a developer from a different tech stack, architect or student.

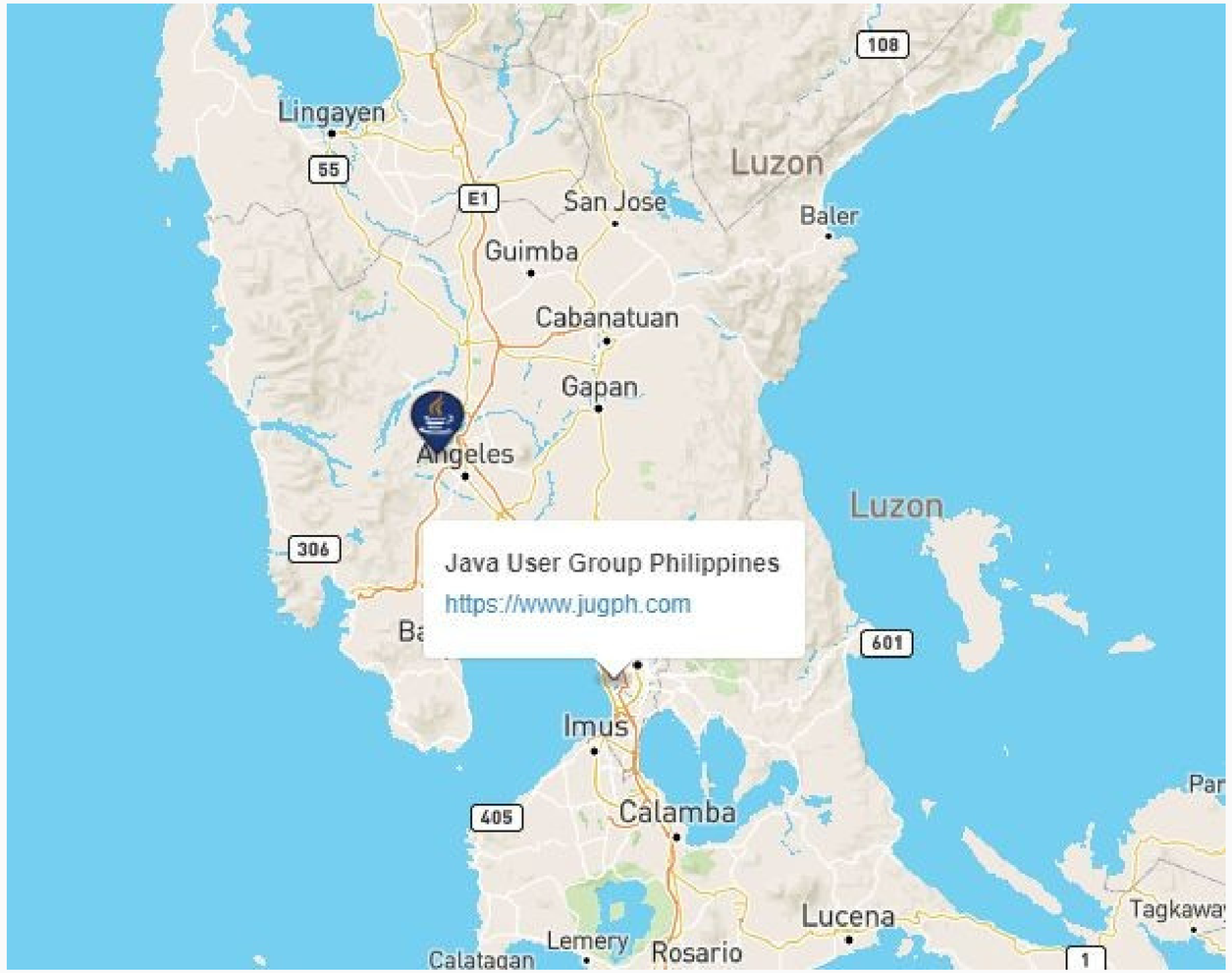
Revitalized JUG since April 2023.

904 Facebook Members

422 Meetup Members

Recognized by Oracle as the only **ACTIVE** JUG* in the Philippines.

*Java User Groups (JUGs) are volunteer organizations that strive to distribute Java-related knowledge around the world.



What are we learning today?

- Introduction to Java
  - What is Java
  - Uses of Java
  - Features of Java
  - History of Java and Java Today
  - Java Version Release Cycle and LTS Version
- Installing Java and IDE
  - System Requirements
  - Setting up the Java Environment
  - Setting up the IDE
  - Uses of an IDE
  - IntelliJ IDE
- Writing and Running your first Java Application
- Java Compilation Process
- Java Portability Model

What are we learning today?

- Java Syntax and Semantics
  - Comments
  - Data Types
  - Variables
  - Identifier Rules
  - The var Keyword
  - Printing Output
  - Understanding Java Operators
- Making Decisions
  - If-else
  - Switch-Case
- Repeating Functionality
  - For loops, while loops, do-while loops
- Methods
  - Types of Methods
  - Passing data among Methods

What are we learning today?

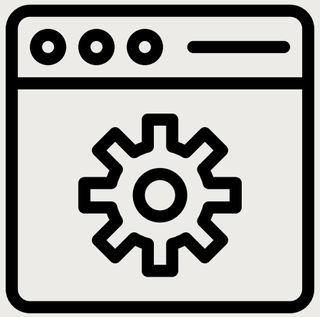
- Methods Cont.
  - Method Scopes
- Object-Oriented Programming I
  - OOP
  - Class
  - Objects
  - How Objects Work
- Strings
  - String Methods (Length, trim, equals, charAt, indexOf, toLowerCase, toUpperCase, split)
- Arrays
- Getting Input
  - Scanner class

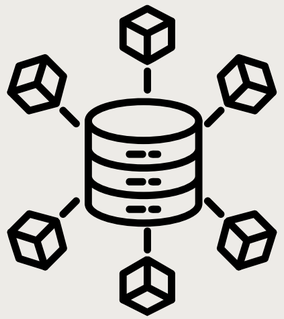
What is Java?

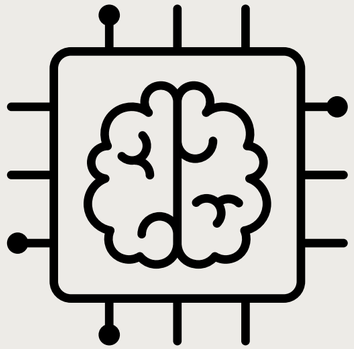
- **Widely-Used Programming Language**: Java is extensively used for coding web applications.
- **Popular Among Developers**: It has been a favorite choice for developers for over two decades.
- **Billions of Applications**
- **Multi-Platform Capability**
- **Object-Oriented**
- **A Platform in Itself:** JVM, Libraries and Java Runtime Environment
- **Fast, Secure, and Reliable**
- **Versatile Uses**: Used for mobile apps, enterprise software, big data applications, and server-side technologies.

What is Java used for?

- Android App Development
- Enterprise Applications
- Web Applications
- Big Data Technologies
- Artificial Intelligence
- Scientific and Research Applications
- Embedded Systems
- Gaming
- Desktop GUI Applications







Features of Java

- **Platform Independence -** Write-Once Run Anywhere
- **Strong Memory Management -** handles memory allocation and garbage collection automatically.
- **Multithreading Support**
- **Object-Oriented Programming**
- **Robust Standard Library** - comprehensive collection of ready-to-use libraries and APIs
- **High Security** - secure runtime environment with features like bytecode verification
- **Large Community and Ecosystem** - Has a vast global community and a rich ecosystem of frameworks and tools
- **Scalability**
- **Enterprise Integration**
- **Stable and Mature**
- **Strong Development Tools -** Supported by powerful IDEs and development tools

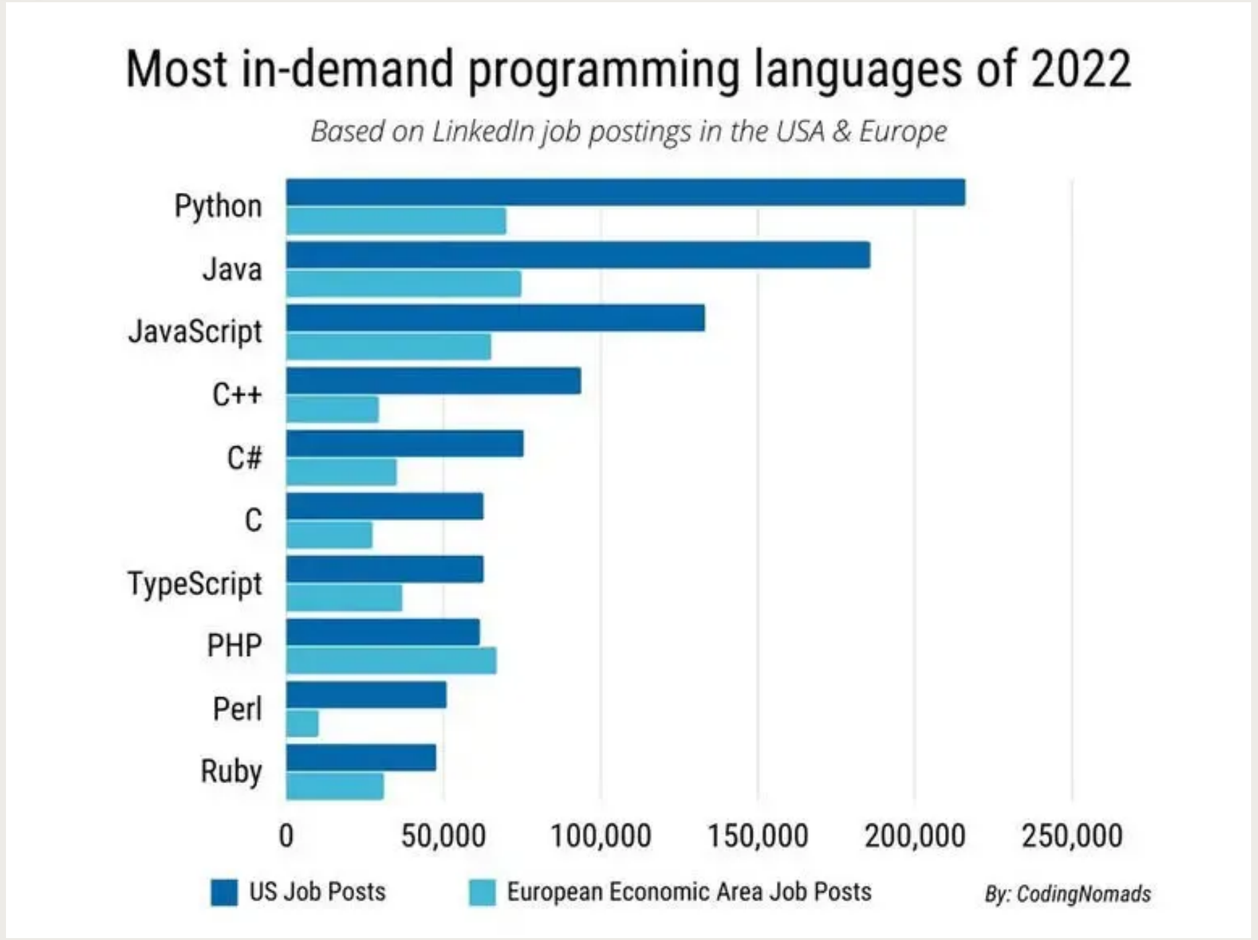
History of Java

- Originally called Oak, then Green, then finally to Java
- Developed by James Gosling and his team in 1991 at Sun MicroSystems
- Designed with C/C++ syntax in mind
- Released as Java 1.0 in 1996
- OpenJDK started on 2006 by Sun MicroSystems
- Oracle Corporation acquired Sun MicroSystems and Java with it in 2009
- Oracle updated the Java version release cycle starting in Java 9

Java Today

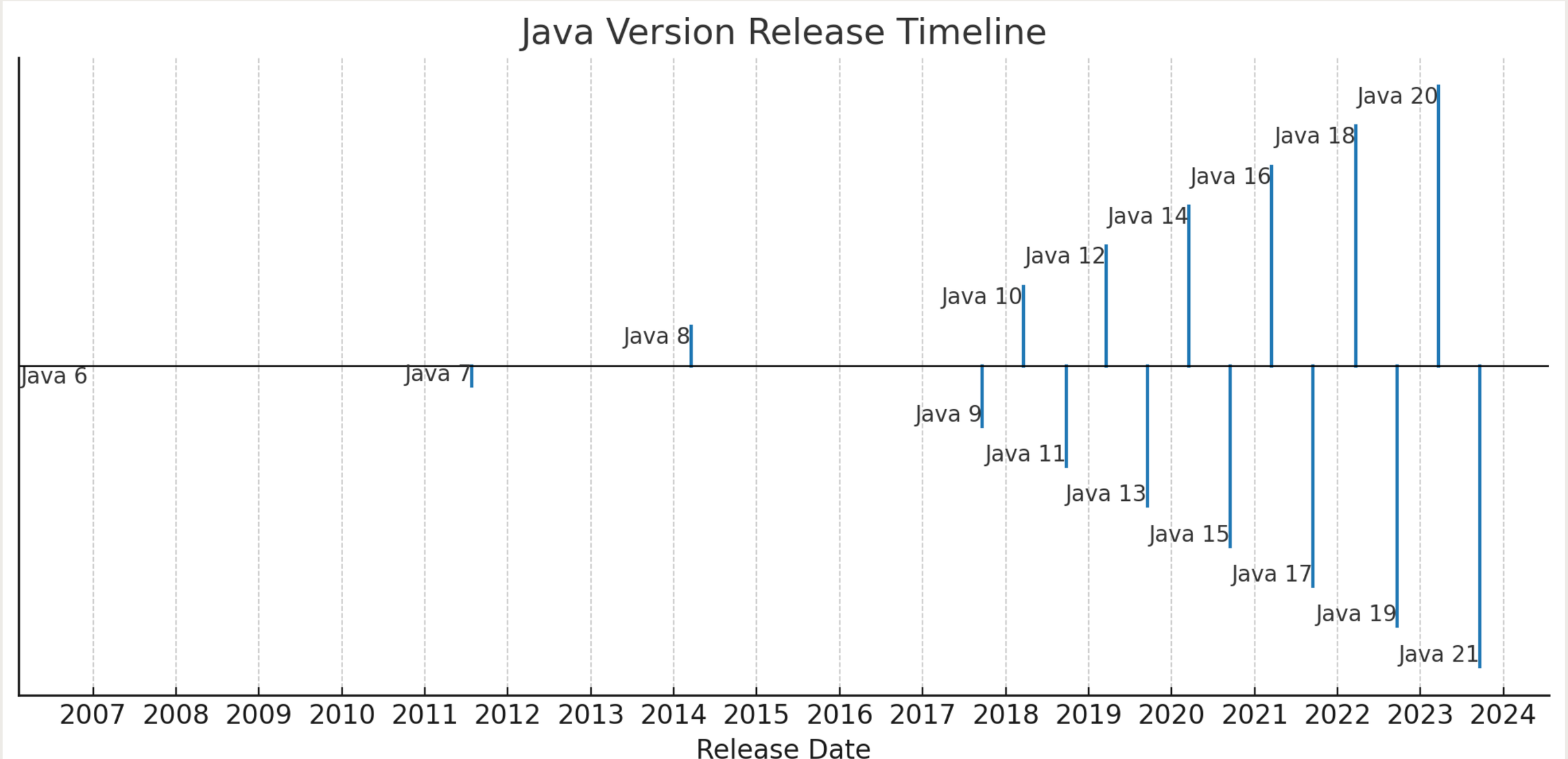
Most Popular Programming Languages by HackerRank 2023

Language	Rank 2022	Mvmt	Abs Incr
Java	1	-	155%
Python	2	-	168%
C++	3	-	171%
Javascript	4	-	157%
C#	5	▲ 1	173%
SQL	6	▼ -1	133%
C	7	-	160%
PHP	8	-	145%
Go	9	▲ 1	190%
Swift	10	▼ -1	127%
Kotlin	11	▲ 1	147%
Ruby	12	▼ -1	124%
Typescript	13	▲ 2	2788%
Scala	14	-	96%
R	15	▼ -2	59%



Taken from invozone.com

Java Version Release Cycle



Java LTS Versions and Important Features

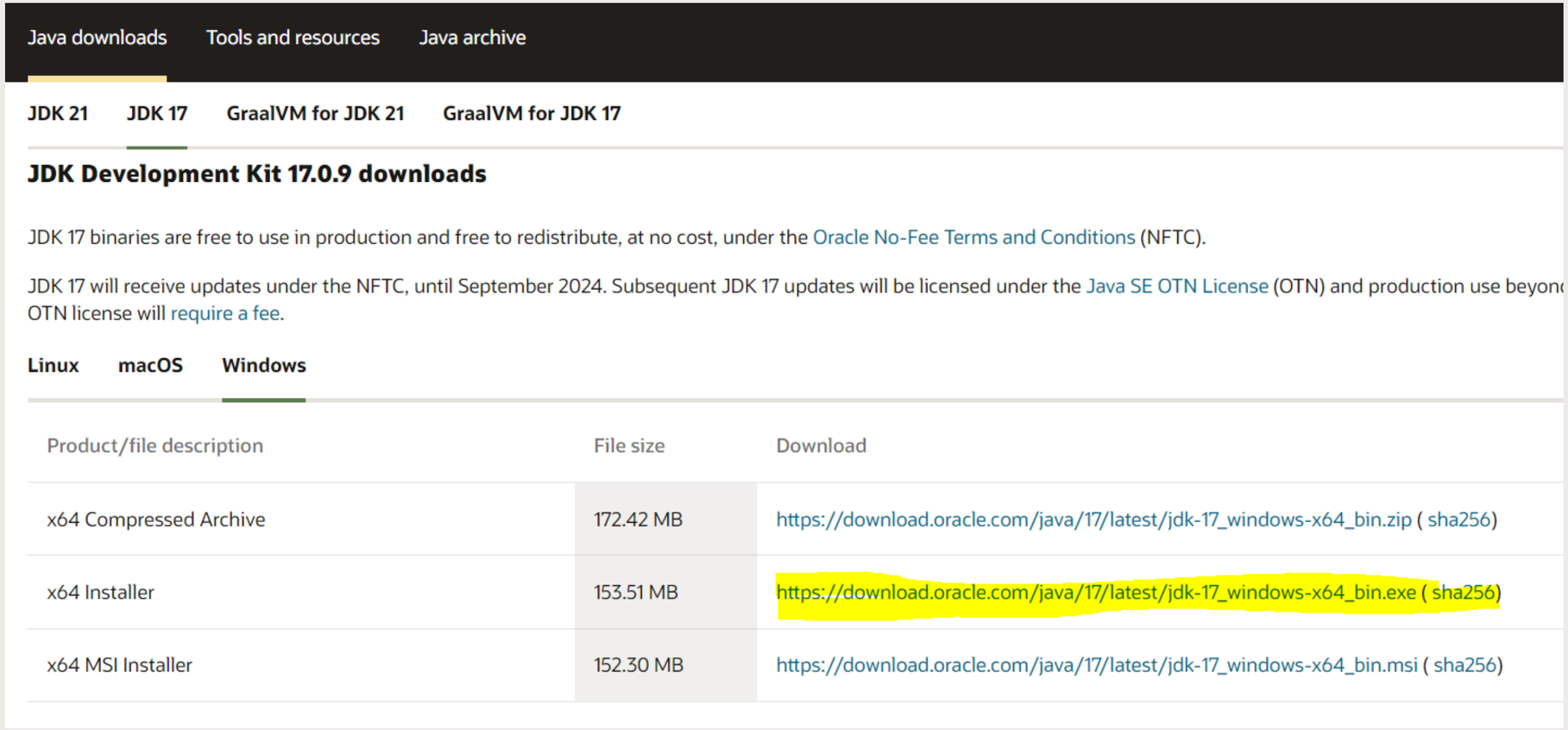
- **Java 8**
  - Lambda Expressions
  - Stream API
  - New Date and Time API
- **Java 11**
  - New HTTP Client (Standard)
  - Flight Recorder
- **Java 17**
  - Sealed Classes (Standard)
  - New macOS Rendering Pipeline
- **Java 21** (Latest version)
  - Virtual Threads (Incubator)
  - Foreign Function & Memory API (Incubator)

System Requirements

- https://www.java.com/en/download/help/sysreq.html
- Can be installed on Windows, Linux and MacOS

Setting up the Java Environment

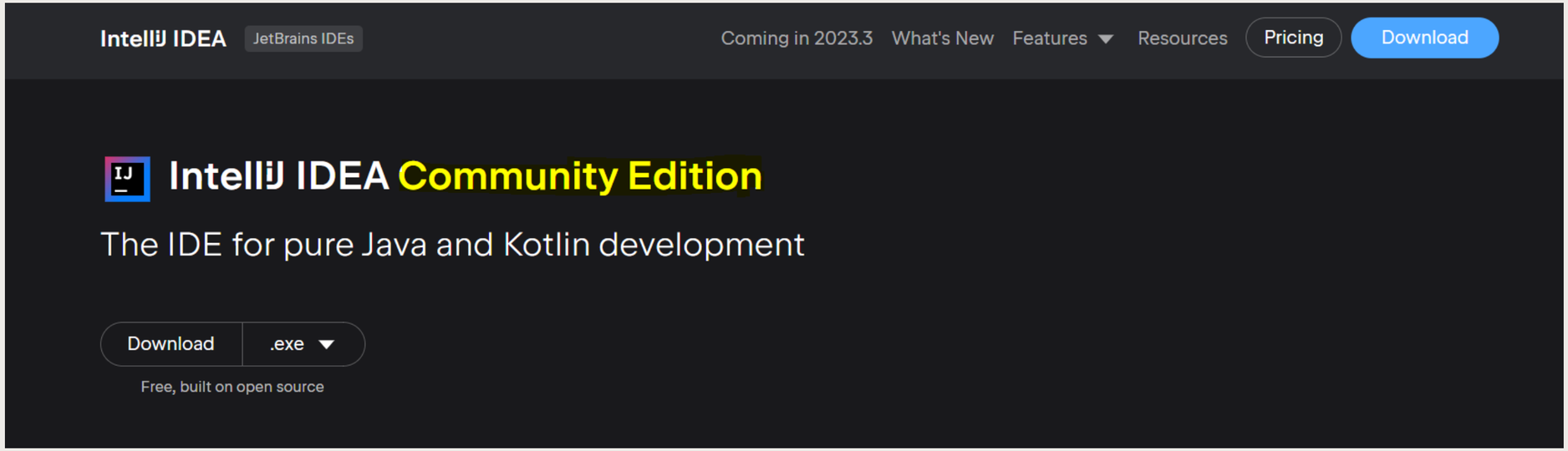
- Download from https://www.oracle.com/java/technologies/downloads/#jdk17-windows



- Install after downloading and follow installation instructions

Setting up the IDE

- Download from https://www.jetbrains.com/idea/download/?section=windows
- Choose Community Edition



- Install after downloading and follow installation instructions

What is an IDE and why do we need it?

- is a software application that provides comprehensive facilities to computer programmers for software development. An IDE typically consists of:
  - Source Code Editor: A text editor for writing and editing code.
  - Build Automation Tools: Compile and build applications.
  - Debugger: Used to test and debug software.

Reasons for using an IDE:

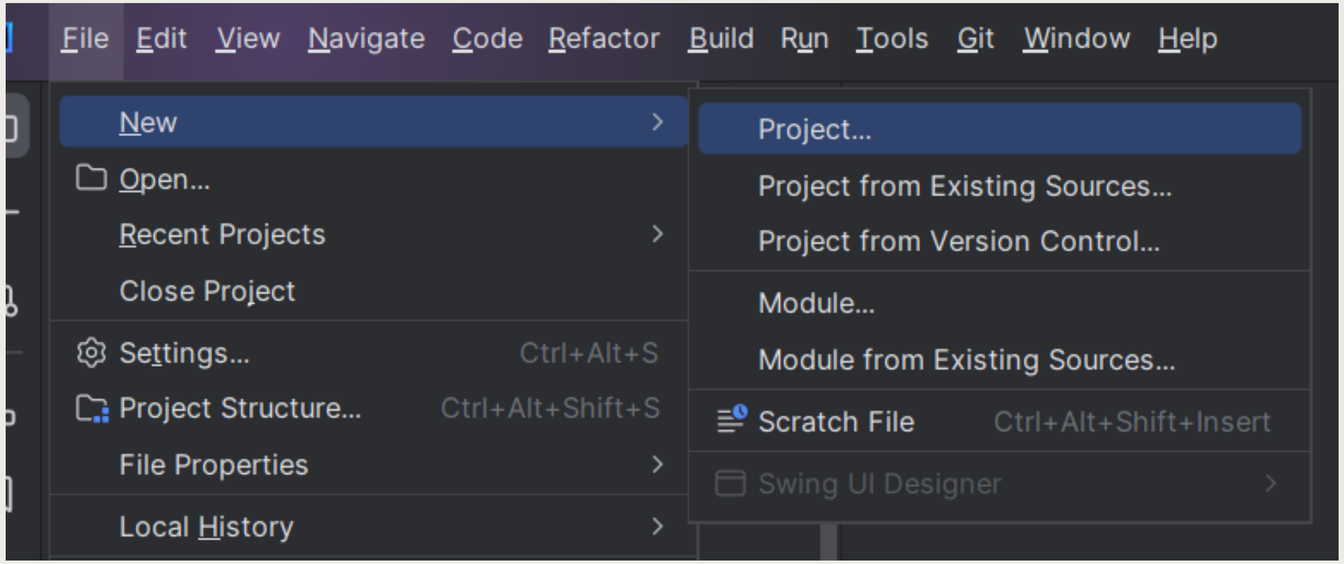
1. Efficiency and Speed
2. Simplifies the Development Process
3. Enhances Code Quality
4. Facilitates Learning
5. Project Management
6. Support for Multiple Languages and Frameworks

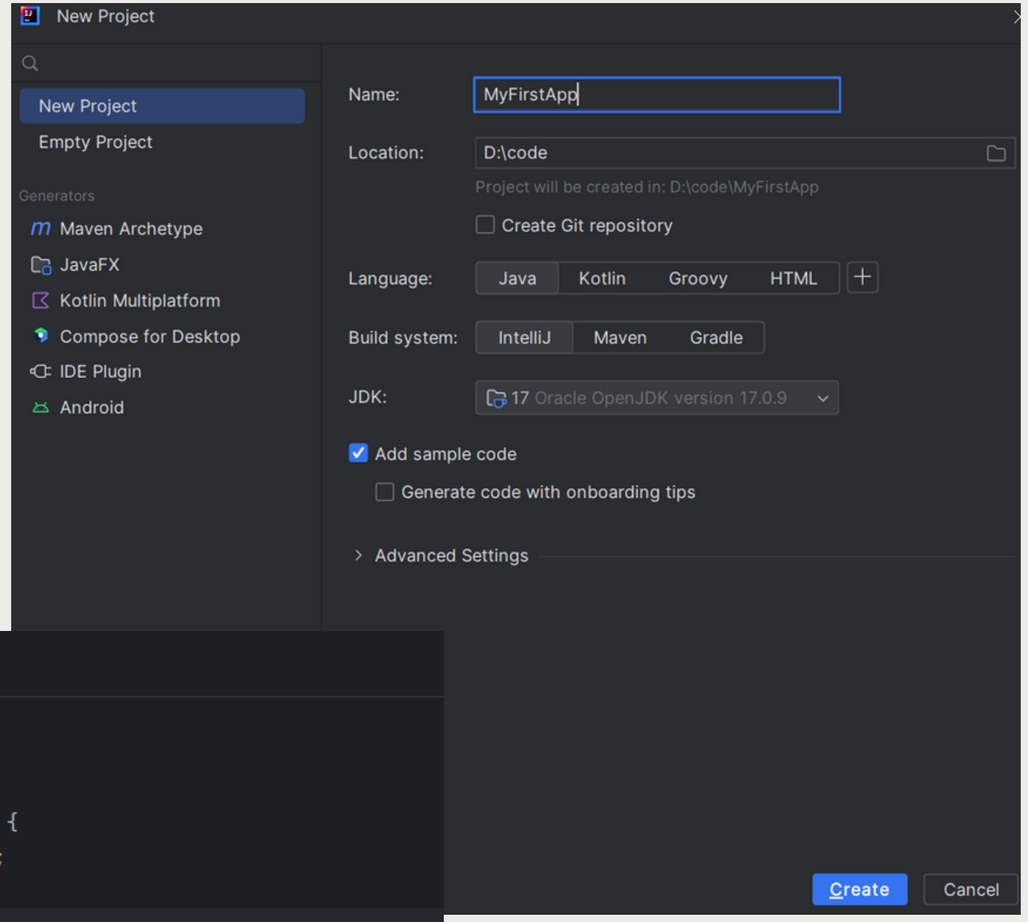
IntelliJ IDE

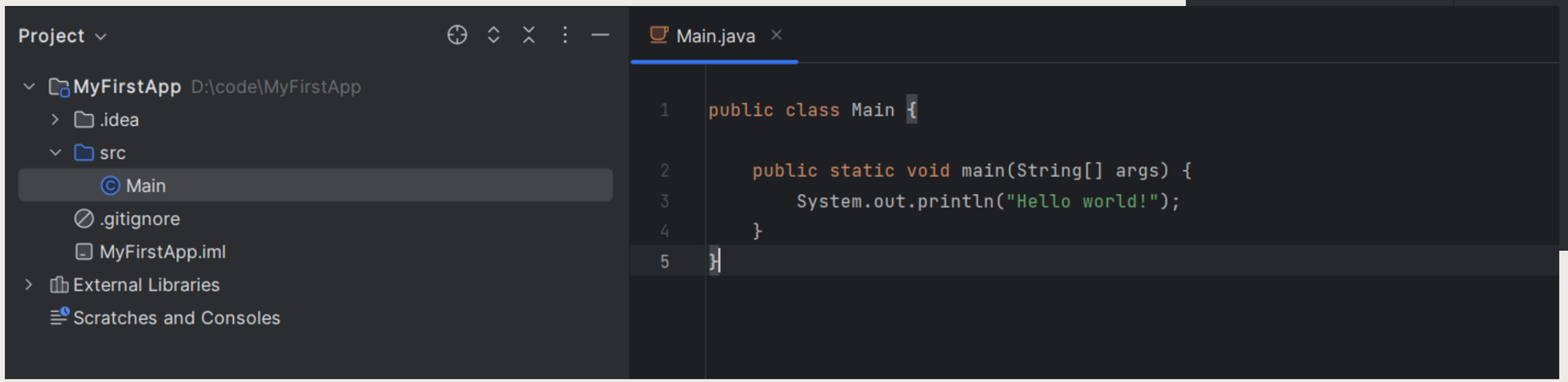
is a powerful and versatile Integrated Development Environment (IDE) primarily used for Java development. It also supports other languages like Kotlin, Groovy, Scala, and a variety of web development languages.

1. **Smart Code Completion**: Offers more intelligent code completion that understands context.
2. **Refactoring Tools**: Provides powerful refactoring options which are both safe and reliable.
3. **User Interface**: Clean and intuitive UI, with a focus on reducing clutter and providing a more efficient coding environment.
4. **Performance**: Tends to be more responsive and less resource-intensive than Eclipse.
5. **Framework Support**: Better out-of-the-box support for modern frameworks and technologies.
6. **Inspections and Quick Fixes**: Offers advanced code inspections and quick-fix suggestions.
7. **Integrated Tools**: Built-in tools for profiling, debugging, decompiling, and database management.

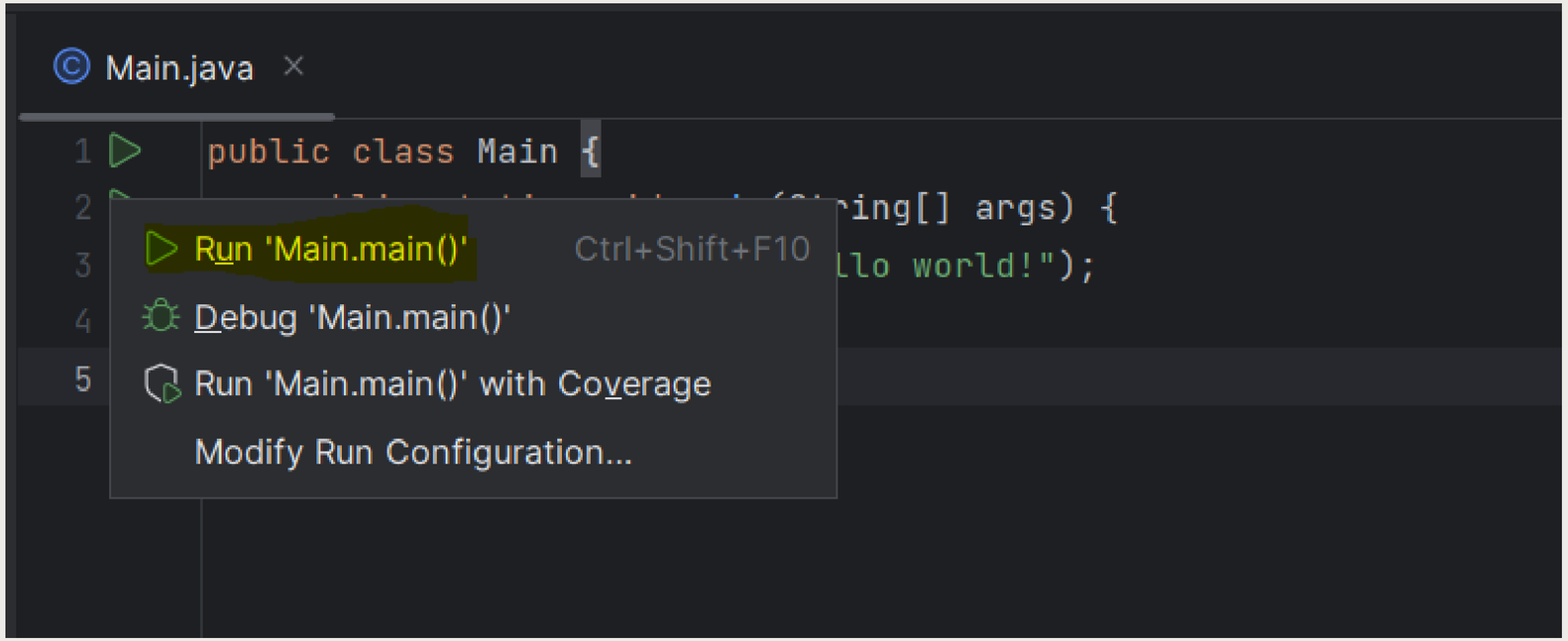
Writing your first Java Application

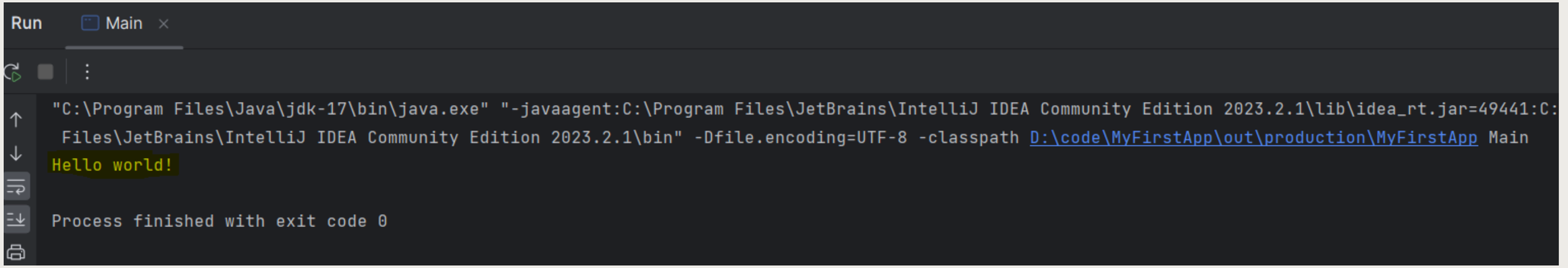






Running your first Java Application



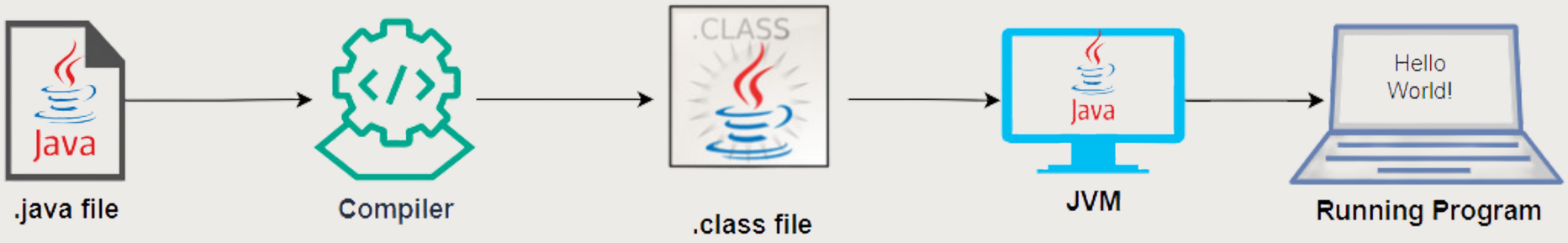


Can you run a Java application without an IDE?

- YES!
- Step 1: Ensure JDK is installed and configured properly
  - Ensure that the **java** and **javac** commands are in your system's PATH. You can verify this by typing *java -version* and *javac -version* in the command prompt or terminal. These commands should return the version of Java installed.
- Step 2: Write your code
  - Write your java program in a text editor like Notepad.
  - Name and Save the file with a .java file extension
- Step 3: Compile your Java program
  - Open a command prompt or terminal window.
  - Navigate to the directory where your **HelloWorld.java** file is located.
  - Compile the Java program using the **javac** command:
    - *javac HelloWorld.java*
- Step 4: Run the compiled Java program
  - Run the program using the java command followed by the class name:
    - *java HelloWorld*

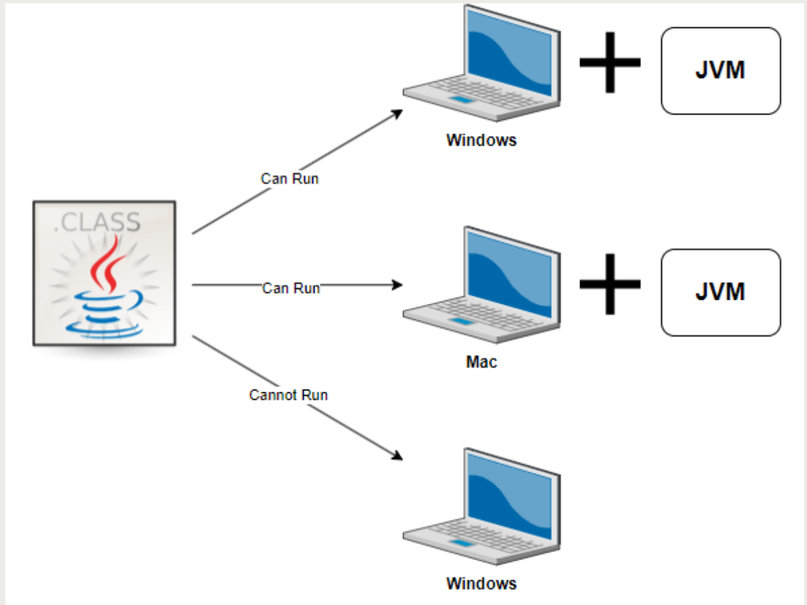
Java Compilation Process behind the scenes

- **Compilation to Bytecode:** The Java source code (**.java** files) is compiled by the Java compiler (**javac**). This compiler translates the human-readable Java code into bytecode, which is a lower-level, platform-independent code. This bytecode is saved in **.class** files.
- **Bytecode:** Bytecode is not machine code and cannot be directly executed by the CPU. It is a highly optimized set of instructions designed to be executed by the Java Virtual Machine (JVM).
- **Running on the JVM**: The Java Virtual Machine (JVM) is the runtime environment that executes the bytecode. When you run a Java program, the JVM reads the bytecode from the .class files.



Java Portability Model

- **Platform Independence**: Achieves platform independence through its use of the Java Virtual Machine (JVM). The JVM acts as an intermediary layer between Java bytecode and the underlying hardware or operating system.
- **Java Virtual Machine (JVM)**: Each operating system or platform has its own implementation of the JVM. The JVM reads and executes Java bytecode, translating it into machine code suitable for the host machine at runtime. This means that as long as a device has a JVM implementation, it can run Java bytecode.
- **Security and Robustness**: Also includes a strong focus on security and robustness, which is essential for applications that run on a wide range of devices with varying levels of security and performance capabilities.



Java Syntax and Semantics

Comment

Comments in Java are notes that you can add to your code to explain what it does. These comments are not executed as part of your program

Single-Line Comments:

- Start with //.
- Everything following // on that line is a comment.

```java
// This is a single-line comment
int number = 5; // This is a comment at the end of a line
```

Multi-Line Comments:

- Start with /* and end with */.
- Can span multiple lines.

```java
/* This is a multi-line comment
   that covers more than
   one line. */
int number = 10;
```

Documentation Comments:

- Start with /** and end with */.
- Used to create documentation for Java classes and methods (JavaDoc).

```java
/**
 * This function returns the sum of two numbers.
 * @param a First number
 * @param b Second number
 * @return Sum of a and b
 */
public int add(int a, int b) {
    return a + b;
}
```

Understanding Data Types

- Specify the size and type of values that variables can hold
- Has two types: **Primitive** and **Reference** Types
- Has 8 built-in data types called the **Java Primitive types**

Keyword	Type	Min Value	Max Value	Default Value	Example
boolean	true or false	n/a	n/a	false	true
byte	8-bit integral value	-128	127	0	123
short	16-bit integral value	-32,768	32,767	0	123
int	32-bit integral value	-2,147,483,648	2,147,483,647	0	123
long	64-bit integral value	-2^{63}	$2^{63}-1$	0L	123L
float	32-bit floating-point value	n/a	n/a	0.0f	123.45f
double	64-bit floating-point value	n/a	n/a	0.0	123.456
char	16-bit Unicode value	0	65,535	\u0000	'a'

- A **float** requires the letter **f or F** following the number so Java knows it is a float. Without an f or F, Java interprets a decimal value as a double. A **long** requires the letter **l or L** following the number so Java knows it is a long. Without an l or L, Java interprets a number without a decimal point as an int in most scenarios.
- What about **String**?

Variables

- A basic unit of storage that holds data which can be modified during program execution.
- Each variable in Java has a specific data type that dictates the size and layout of the variable's memory, the range of values that can be stored within that memory, and the set of operations that can be applied to the variable.
- Value of a variable can be changed during the execution of the program, and it acts as a placeholder to represent data that can vary or change.

Declaring Variables

- Defined by specifying its type, followed by its name, and optionally initializing it with a value:
- <Data Type> <Variable Name>;
- <Data Type> <Variable Name> = <Value>;
- int age = 30; String name;

```
byte aByte = 10; // Decimal
// Short
short aShort = 1000; // Decimal
// Int
int anInt = 10000; // Decimal
int million2 = 1_000_000;
int hisAge = 12, herAge = 15;
// Long
long aLong = 100000L; // Decimal, L suffix for long type
// Float
float aFloat = 10.99f; // Decimal, f suffix for float type
// Double
double aDouble = 12345.678; // Decimal
double underDecimal = 10_00.00;
// Boolean
boolean aBoolean = true; // true or false
// Char
char aChar = 'A'; // Single character
char unicodeChar = '\u0041'; // Unicode representation
```

Declaring non-base 10 literals

```
// Byte - Binary
byte binaryByte = 0b1010;

// Short - Octal
short octalShort = 01750;

// Int - Binary, Octal, and Hexadecimal
int binaryInt = 0b1010001101001;
int octalInt = 023641;
int hexInt = 0x2710;

// Long - Binary and Hexadecimal
long binaryLong = 0b1100101010110010101011100101010L;
long hexLong = 0xFFFFFFFFFFL;
```

Identifier Rules

- An Identifier is the name of a variable, method, class, interface, or package.
- Identifiers must begin with a letter, a currency symbol, or a _ symbol. Currency symbols include dollar ($), yuan (¥), euro (€), and so on.
- Identifiers can include numbers but not start with them. A single underscore _ is not allowed as an identifier.
- You cannot use the same name as a Java reserved word. A reserved word is a special word that Java has held aside so that you are not allowed to use it. Remember that Java is case sensitive, so you can use versions of the keywords that only differ in case. Please don't, though.

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

Identifier Examples

WORKS BUT UGLY

```
long okidentifier;
float $OK2Identifier;
boolean _alsoOK1d3ntifi3r;
char __SStillOkbutKnotsonice$;
```

DOES NOT COMPILE

```
int 3DPointClass; // identifiers cannot begin with a number
byte hollywood@vine; // @ is not a letter, digit, $ or _
String *$coffee; // * is not a letter, digit, $ or _
double public; // public is a reserved word
short _; // a single underscore is not allowed
```

JUST PERFECT

```
boolean hasPizza = true;
String myName;
int requiredAge = 21;
```

Best practice is to use camelCase and have your variables concise yet short, and captures your overall intention

Introducing **var**

- Allows you to declare a variable without explicitly stating its type
- The type of the variable is inferred by the compiler based on the value assigned to it
- Makes your code shorter, cleaner and readable

WITHOUT VAR

```
String message = "Hello, World!";
```

WITH VAR

```
var message = "Hello, World!";
```

Key Points

- var can only be used when declaring local variables (inside methods or blocks of code).
- The variable must be initialized at the time of declaration.
- Use var where it makes the code more readable and the type is obvious from the context. Avoid it in complex or unclear situations.

Printing Output

We have different ways to display text or information on the screen. The most common methods are **println**, **print**, and **printf** from the System class.

println - Print with a Line Break

Prints something and then moves to a new line.

```
System.out.println("Hello, world!"); // Prints "Hello, world!" and goes to a new line
System.out.println("Welcome to Java!"); // Prints on the next line
```

print - Print Without a Line Break

Prints something but stays on the same line

```
System.out.print("Hello, ");
System.out.print("world!"); // Prints "Hello, world!" on the same line
```

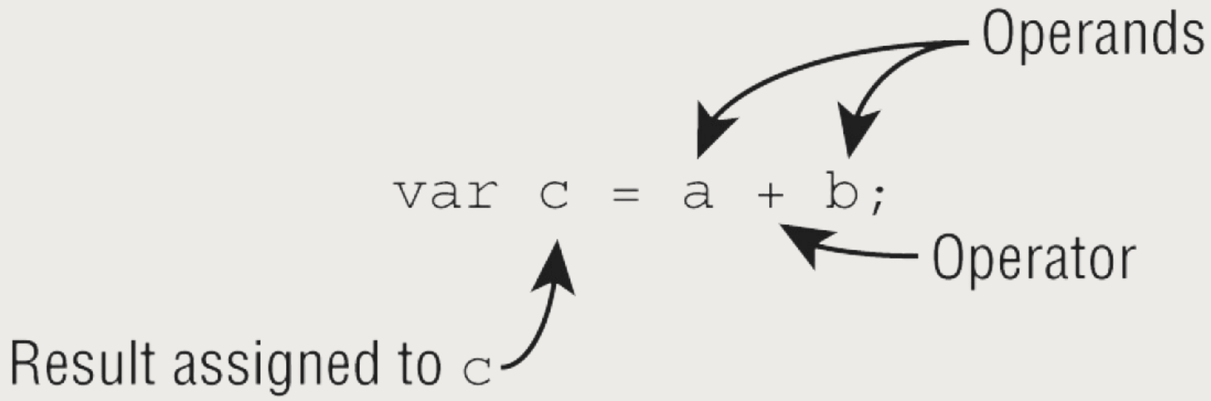
printf - Formatted Print

Lets you print with a format. This is like a more advanced printer where you can decide exactly how things should look.

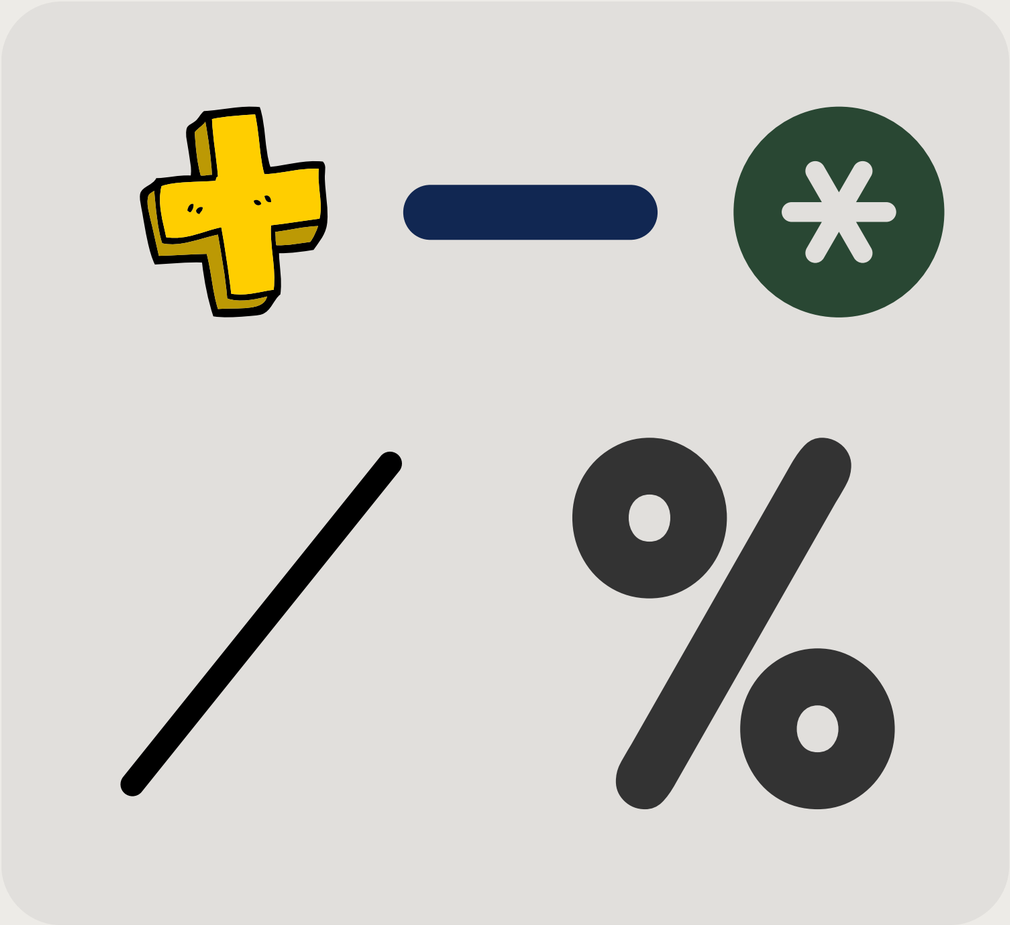
```
String name = "Java";
int version = 17;
System.out.printf("Welcome to %s version %d!", name, version); // Prints "Welcome to Java version 17!"
```

Understanding Java Operators

- is a special symbol that can be applied to a set of variables, values, or literals—referred to as operands—and that returns a result.



Arithmetic Operators



Perform basic mathematical operations like addition, subtraction, multiplication, division, and modulus (remainder)

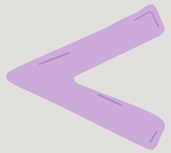
```
int a = 10, b = 5;
int sum = a + b; // sum = 15
int diff = a - b; // diff = 5
int prod = a * b; // prod = 50
int div = a / b; // div = 2
int mod = a % b; // mod = 0
```

Relational Operators

Checks if two values are equal.

Checks if two values are not equal

Checks if the value on the left is greater than the one on the right



Checks if the value on the left is less than the one on the right

Checks if the value on the left is greater than or equal to the one on the right

Checks if the value on the left is less than or equal to the one on the right

Compare two operands and return a boolean value (true or false) based on the comparison. They include equals, not equals, greater than, less than, greater than or equal to, and less than or equal to

```
int a = 5;
int b = 3;

boolean isEquals = (a == b); // false
boolean isNotEquals = (a != b); // true
boolean isGreaterThan = (a > b); // true
boolean isLessThan = (a < b); // false
boolean isGreaterOrEqual = (a >= b); // true
boolean isLessOrEqual = (a <= b); // false
```

Logical Operators

&&

Logical AND: Returns true if both operands are true.

||

Logical OR: Returns true if at least one operand is true.

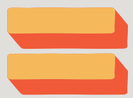
!

Logical NOT: Reverses the logical state of its operand.

Used with boolean (logical) values to perform logical AND, OR, and NOT operations. They determine the truth or falsehood of boolean expressions.

```
int a = 10, b = 5;
boolean andResult = (a > b) && (b > 0); // andResult = true
boolean orResult = (a < b) || (b > 0); // orResult = true
boolean flag = true;
boolean notFlag = !flag; // notFlag = false
```

Assignment Operators



Assigns a value to a variable.

Adds the right operand to the left operand and assigns the result to the left.

Subtracts the right operand from the left operand and assigns the result to the left.

Multiplies the left operand by the right operand and assigns the result to the left.

Divides the left operand by the right operand and assigns the result to the left.

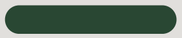
Applies modulus operation and assigns the result to the left operand.

Are used to assign values to variables. The most basic assignment operator is =, which directly assigns the value of the right operand to the left operand. In addition to the simple assignment, there are compound assignment operators like +=, -=, *=, /=, and %=, which combine arithmetic operations with assignment.

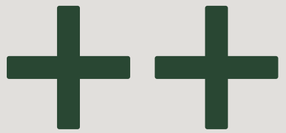
```
int a = 10, b = 5;     // Simple assignment
a += b;     // a = a + b; Now a is 15
a -= b;     // a = a - b; Now a is 10
a *= b;     // a = a * b; Now a is 50
a /= b;     // a = a / b; Now a is 10
a %= b;     // a = a % b; Now a is 0
```

Unary Operators

Indicates a positive value



Negates the value.



Increases a value by 1, pre and post.

Decreases a value by 1, pre and post

Operate on a single operand to perform various operations like negation, increment, decrement, and logical not

```java
int a = 5;
int b = -5;

// Unary Plus
int plus = +a;
System.out.println(plus); // plus = 5

// Unary Minus
int minus = -a;
System.out.println(minus); // minus = -5

// Post-Increment
System.out.println(a++); // a = 5 before incrementing
System.out.println(a); // a = 6 after incrementing

// Pre-Increment
System.out.println(++a); // a = 7 after this line

// Post-Decrement
System.out.println(b--); // b = -5 before decrementing
System.out.println(b); // b = -6 after decrementing

// Pre-Decrement
System.out.println(--b); // b = -7 after this line
```

Making Decisions

Conditional if-else Statements

Are used to perform different actions based on different conditions. They allow your program to choose different paths of execution based on the evaluation of boolean expressions.

- **if** Statement: This is used to execute a block of code if the specified condition is true.
- **else if** Statement: This is used after an if statement to define a new condition to test, if the previous condition is false.
- **else** Statement: This is used after an if statement (or else if statements) to define a block of code that will execute if all the previous conditions are false.

```
if (condition1) {
    // code to be executed if condition1 is true
} else if (condition2) {
    // code to be executed if condition1 is false and condition2 is true
} else {
    // code to be executed if both condition1 and condition2 are false
}
```

```
int number = 15;

 if (number < 10) {
     System.out.println("The number is less than 10");
 } else if (number > 10 && number < 20) {
     System.out.println("The number is between 10 and 20");
 } else {
     System.out.println("The number is 20 or more");
 }
```

Nested Conditional if-else Statements

Nested conditions in Java refer to the use of one or more if, else if, or else statements inside another if or else block. This structure allows for more complex decision-making processes by evaluating multiple layers of conditions.

```java
int a = 10;
int b = 20;

if (a > 5) {
    // Outer condition
    if (b > 15) {
        // Nested condition
        System.out.println("Both conditions are true.");
    } else {
        System.out.println("a is greater than 5 but b is not greater than 15.");
    }
} else {
    System.out.println("a is not greater than 5.");
}
```

Nested Conditional Statements Best Practices

```php
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) > 5) {
                        if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```

- Limit the Depth of Nesting
- Avoid Duplicating Code
- Combine Logical Operators instead of nesting if conditions
- Use Switch or Enum where applicable
- Consistent Formatting

Switch-Case Statements

Provide an efficient way to execute different parts of code based on the value of an expression. They are often used as an alternative to long, nested if-else statements, especially when you have a variable that can take one of many possible values.

Traditional Switch Case (before Java 12)

- **break** Statement: the break statement is used to prevent the flow of execution from falling through to the next case.
- **default** Case: This is executed if none of the case values match the expression. It's optional but useful for handling unexpected values.
- **Data Types**: The switch expression can be a byte, short, char, int, String (since Java 7), or an enum type.

```
switch (expression) {
  case value1:
    // code block
    break;
  case value2:
    // code block
    break;
  // more cases...
  default:
    // default code block
}
```

```
// Traditional Switch Case
int day = 4;
switch (day) {
  case 1:
    System.out.println("Monday");
    break;
  case 2:
    System.out.println("Tuesday");
    break;
  case 3:
    System.out.println("Wednesday");
    break;
  case 4:
    System.out.println("Thursday");
    break;
  case 5:
    System.out.println("Friday");
    break;
  case 6:
    System.out.println("Saturday");
    break;
  case 7:
    System.out.println("Sunday");
    break;
  default:
    System.out.println("Invalid day");
}
```

Switch-Case Statements

Enhanced Switch (Java 12 and later)

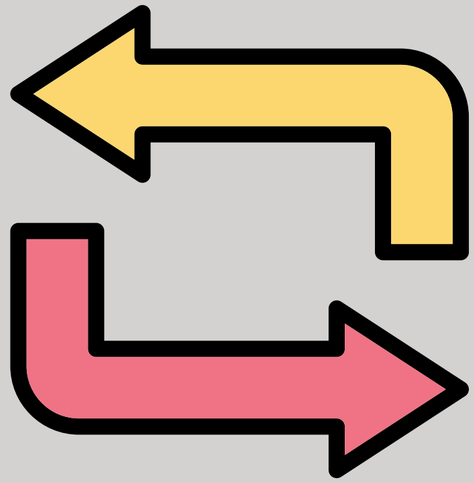
- More concise syntax, allowing multiple case labels and the yield statement for returning values.
- The enhanced switch statement, introduced in Java 12, allows a more concise syntax and avoids the need for **break** statements. It also supports returning values directly with **yield**.

```
switch (expression) {
  case value1, value2 -> // code block or single statement
  case value3 -> {
    // code block
  }
  // more cases...
  default -> // default code block or single statement
}
```

```
result = switch (expression) {
  case value1 -> {
    // code block
    yield returnValue1;
  }
  case value2 -> {
    // code block
    yield returnValue2;
  }
  // more cases...
  default -> {
    // default code block
    yield defaultValue;
  }
};
```

```
int month = 4;
String season = switch (month) {
  case 12, 1, 2 -> "Winter";
  case 3, 4, 5 -> {
    // Additional logic
    yield "Spring";
  }
  case 6, 7, 8 -> "Summer";
  case 9, 10, 11 -> "Fall";
  default -> "Unknown";
};
System.out.println(season);
```

Repeating Functionalities



Java Loops

Loops in Java are used to execute a block of code repeatedly until a specified condition is met. There are three main types of loops in Java: for, while, and do-while. Each serves a specific purpose and can be chosen based on the requirements of the task at hand.

for Loop

- Ideal for scenarios where you know in advance how many times you need to iterate.

```
for (initialization; condition; update) {
  // code block to be executed
}
```

```
for (int i = 0; i < 5; i++) {
  System.out.println(i);
}
```

```
while (condition) {
  // code block to be executed
}
```

```
int i = 0;
while (i < 5) {
  System.out.println(i);
  i++;
}
```

while Loop

- Useful when the number of iterations is not known beforehand, and you want to continue looping as long as a condition is true.

Java Loops

do-while Loop

- Similar to the **while** loop, but it ensures that the loop will be executed at least once.

```
do {
  // code block to be executed
} while (condition);
```

```
int i = 0;
do {
  System.out.println(i);
  i++;
} while (i < 5);
```

Choosing the Right Loop

- Use a **for** loop when the number of iterations is known or determinable at the beginning of the loop.
- Use a **while** loop when the continuation condition is dependent on something other than a straightforward iteration count.
- Use a **do-while** loop when the loop body needs to execute at least once, regardless of the condition's initial state.

Break and **Continue** Keywords

break and **continue** are two important keywords used within loops (for, while, do-while) to control the flow of the loop.

Break Keyword

- Usage: The break keyword is used to immediately exit the loop, regardless of the loop's condition.
- Common Use Case: It is often used in situations where you need to stop the loop once a certain condition is met.

```java
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break; // Loop exits when i reaches 5
    }
    System.out.println(i);
}
```

Break and **Continue** Keywords

Continue Keyword

- Usage: The continue keyword skips the current iteration of the loop and moves directly to the next iteration.
- Common Use Case: It's useful when you want to skip some specific condition within the loop but continue looping.

```java
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        continue; // Skips the rest of the loop body when i is 5
    }
    System.out.println(i);
}
```

Loops can be Nested as well

Nested for-loops in Java are loops within loops. They are commonly used for tasks that require iteration over multiple dimensions, like processing multi-dimensional arrays or creating patterns.

```java
public static void main(String[] args) {
    int size = 5; // Size of the multiplication table

    for (int i = 1; i <= size; i++) { // Outer loop
        for (int j = 1; j <= size; j++) { // Inner loop
            System.out.print(i * j + "\t"); // Multiply and print result
        }
        System.out.println(); // New line after each row
    }
}
```

Methods

Java Methods

Java methods are blocks of code designed to perform a specific task. They are fundamental for organizing and structuring code in a logical, reusable, and maintainable way. Methods allow you to break down complex problems into smaller, manageable subtasks.

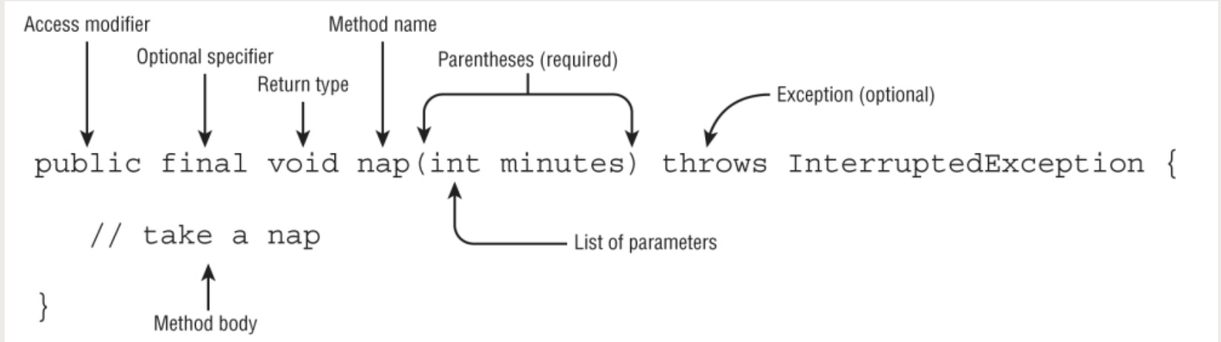
Why We Need Methods

- **Reusability**: Write once, use multiple times.
- **Modularity**: Break down complex problems into smaller chunks.
- **Maintainability**: Easier to update and debug.
- **Readability**: Enhances understanding of the code.

Methods allows for a clean, organized way to execute different tasks, making code easier to read and manage. Methods in the same class can be called directly, while methods in other classes require object creation or different access strategies, which can be discussed later.

In Java, methods are defined within a class.

```
returnType methodName(parameterList) {
    // method body
    // optional return statement
}
```



Types of Methods

Methods Without Parameters

```java
public void showWelcomeMessage() {
    System.out.println("Welcome!");
}
```

Methods With Parameters

```java
public int addNumbers(int a, int b) {
    return a + b;
}
```

Methods That Don’t Return Anything

```java
public void displayNumber(int number) {
    System.out.println("Number: " + number);
}
```

Methods That Return a Value

```java
public double computeArea(double radius) {
    return 3.14 * radius * radius;
}
```

Combination

```java
public void bark(int size) {
     if (size > 60) {
        System.out.println("Wooof! Wooof!");
     } else if (size > 14) {
        System.out.println("Ruff! Ruff!");
     } else {
        System.out.println("Yip! Yip!");
     }
 }
```

How you call them inside own class

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

    int result = addNumbers(5, 7);

    displayNumber(15);

    double area = computeArea(10.0);
    System.out.println("Area: " + area);
}
```

Passing data among Methods

- Java is a pass-by-value language.
- Method arguments is only a copy and assignments made to those arguments inside the method do not affect the caller.

Passing Primitives to Methods

```java
public static void main(String[] args) {
    int num = 4;
    newNumber(num);
    System.out.print(num); // This will print 4
}

public static void newNumber(int num) {
    num = 8;
}
```

In the main method, when num is printed, it still has its original value 4, demonstrating that the changes in newNumber did not affect the original variable.

Passing Objects to Methods

```java
public static void main(String[] args) {
    String greeting = "Hello";
    modifyString(greeting);
    System.out.println(greeting); // This will print "Hello"
}

public static void modifyString(String str) {
    str = "Goodbye";
}
```

In the main method, when greeting is printed, it still has its original value "Hello", demonstrating that the original String object was not modified

Method Scopes

- **Local Scope**: Variables declared within a method are locally scoped and are only accessible within that method.

```java
// Method Scopes
public void myMethod() {
    int localVariable = 5; // Local to myMethod
    System.out.println(localVariable); // Accessible here
}
// System.out.println(localVariable); // Error: localVariable not accessible outside myMethod
```

- **Parameter Scope:** Method parameters are also locally scoped to the method and behave like local variables.

```java
// Parameter Scope
public void printNumber(int number) {
    System.out.println(number); // 'number' is a parameter and locally scoped
}
// System.out.println(number); // Error: 'number' not accessible
```

- **Block Scope:** Within a method, variables declared in a block (like in loops or conditional statements) are only accessible within that block.

```java
// Block Scope
public void loopMethod() {
    for (int i = 0; i < 5; i++) {
        System.out.println(i); // 'i' is accessible within this loop block
    }
    // System.out.println(i); // Error: 'i' not accessible outside the loop bloc
}
```

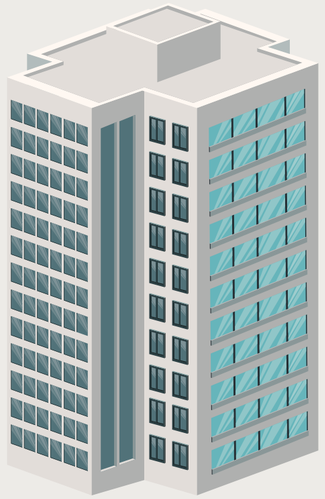
Method Scopes

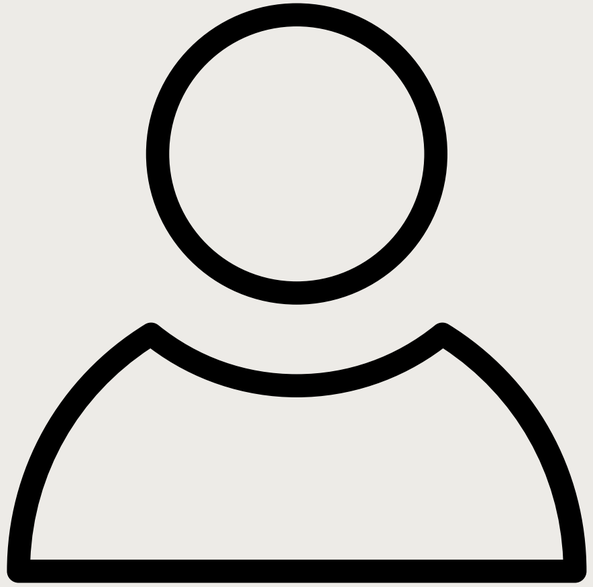
- **Shadowing:** If a local variable or parameter has the same name as a field, it shadows the field within the method.

```java
public class MyClass {
    int memberVariable = 10;

    public void methodWithShadowing() {
        int memberVariable = 5; // This local variable shadows class member variable
        System.out.println(memberVariable); // Prints 5, the local variable
    }

    public void methodWithoutShadowing() {
        System.out.println(memberVariable); // Prints 10, the member variable of the class
    }
}
```

Object Oriented Programming I

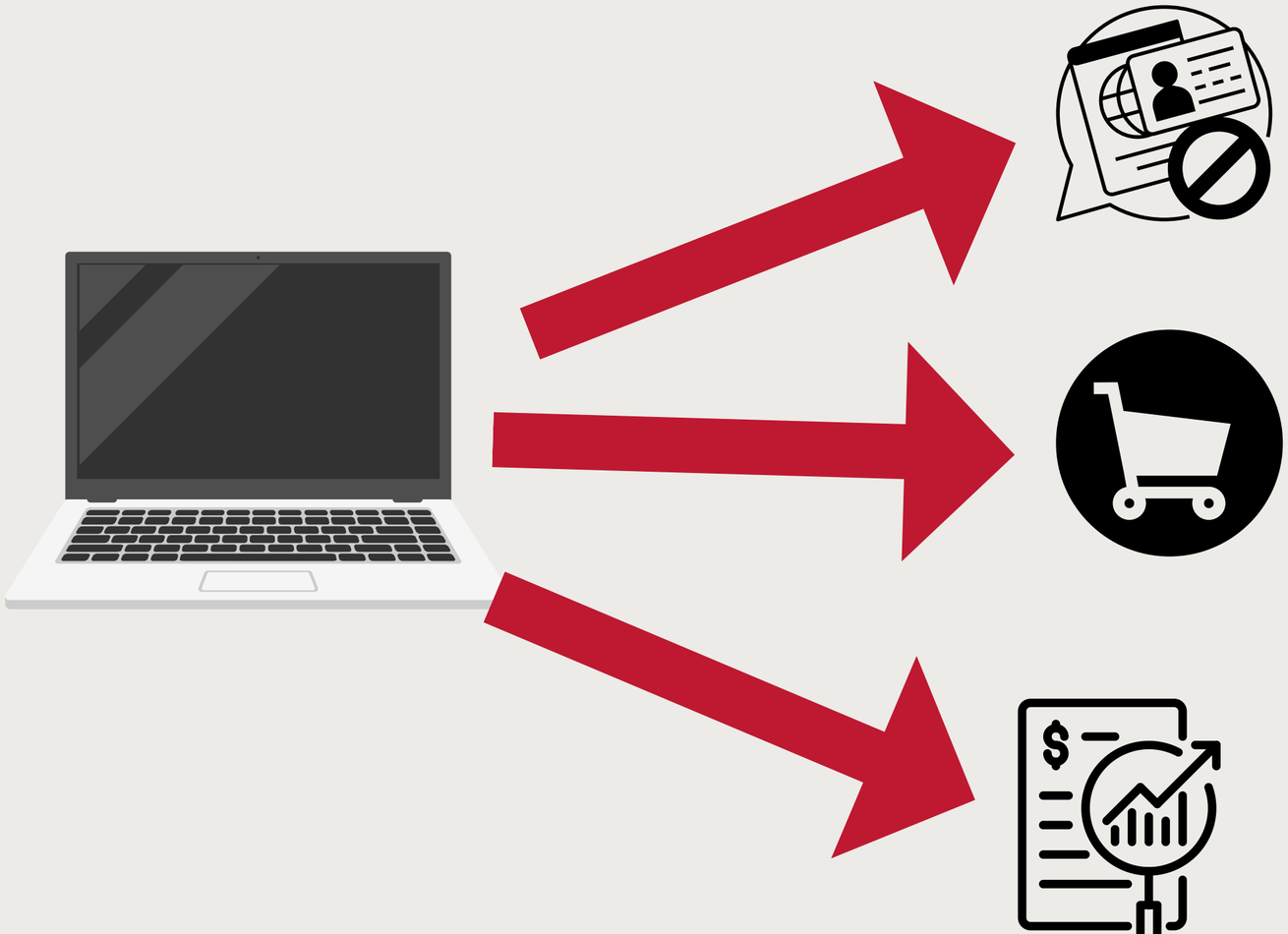
We are surrounded by Objects





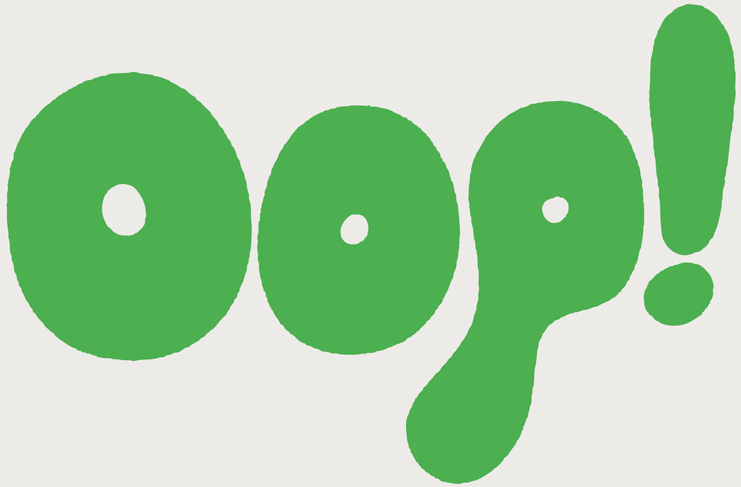
Objects are Data

- In programming, programs use and interact with data such as a person’s financial information in a financial system, employee information in a HR system or item information in an inventory system



Object-Oriented Programming

- Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic.
- Real world example is an HR system that comprises of Employees, and each employee has its own set of information such as their name, gender, address and other information.
- Its structure includes **objects**, **classes**, **methods** and **attributes**.



Why the need for OOP?

- **Modularity**. Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
- **Reusability**. Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
- **Productivity**. Programmers can construct new programs quicker through the use of multiple libraries and reusable code.
- **Scalability**. Programmers can implement system functionalities independently.
- **Interface descriptions**. Descriptions of external systems are simple, due to message passing techniques that are used for objects communication.
- **Security**. Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.
- **Flexibility**. Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

Procedural vs OOP

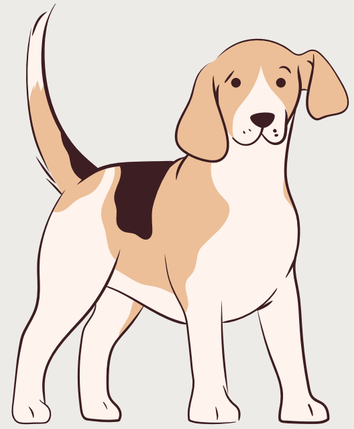
```java
public class LibraryOOP {

    public static void main(String[] args) {
        String book1Title = "1984";
        String book1Author = "George Orwell";
        int book1Year = 1949;

        String book2Title = "To Kill a Mockingbird";
        String book2Author = "Harper Lee";
        int book2Year = 1960;

        printBookDetails(book1Title, book1Author, book1Year);
        printBookDetails(book2Title, book2Author, book2Year);
    }

    public static void printBookDetails(String title, String author, int year) {
        System.out.println("Title: " + title + ", Author: " +
        author + ", Year: " + year);
    }
}
```

```java
public class Book {
    private String title;
    private String author;
    private int year;

    public Book(String title, String author, int year) {
        this.title = title;
        this.author = author;
        this.year = year;
    }

    public void printDetails() {
        System.out.println("Title: " + title + ", Author: " +
            author + ", Year: " + year);
    }
}

public class LibraryOOP {

    public static void main(String[] args) {
        Book book1 = new Book("1984", "George Orwell", 1949);
        Book book2 = new Book("To Kill a Mockingbird", "Harper Lee", 1960);

        book1.printDetails();
        book2.printDetails();
    }
}
```

Class

- **Blueprint**: A class is a blueprint for objects. You can create many objects from a class.
- **Encapsulation**: It encapsulates data and behavior. Data (attributes) and behavior (methods) related to the class are kept together.
- **Object Creation**: Use the **new** keyword to create instances (objects) of a class.
- **Customization**: Each object can have unique attribute values (e.g., different name, breed, and age for each Dog object).

```java
public class Dog {
    // Attributes (Fields)
    String name;
    String breed;
    int age;

    // Constructor
    public Dog(String name, String breed, int age) {
        this.name = name;
        this.breed = breed;
        this.age = age;
    }

    // Method to make the dog bark
    public void bark() {
        System.out.println(name + " says Woof!");
    }

}
```



Objects

Is an instance of a class. It's a concrete realization of the class's blueprint, with its own set of data and behavior.

Key Characteristics of Objects:

1. **Identity**: Each object has a unique identity that distinguishes it from other objects, even if they are of the same class.
2. **State**: An object has a state, determined by the values of its attributes (fields). For example, a Dog object might have a name, breed, and age.
3. **Behavior**: Objects have behavior, defined by the methods in their class. These methods operate on the object's state. For instance, a Dog object may have behaviors like bark or run.

How Objects work

- **Instantiation**: The process of creating an object from a class is called instantiation. This is done using the **new** keyword in Java.
- **Memory Allocation:** When an object is created, memory is allocated for storing its state (its fields).
- **Method Invocation**: Objects can have methods invoked on them to perform actions or operations, affecting their state or providing functionality.
- **Interaction**: Objects can interact with other objects. For example, a Dog object might interact with a Person object in a hypothetical feed method.

```java
public class Main {
    public static void main(String[] args) {
        // Creating instances of the Dog class
        Dog myDog = new Dog("Buddy", "Golden Retriever", 3);
        Dog neighborDog = new Dog("Rex", "German Shepherd", 4);

        // Calling methods on these instances
        myDog.bark(); // Buddy says Woof!

        neighborDog.bark(); // Rex says Woof!
    }
}
```

To call a method,
use the . (dot) operator

Strings

What are Strings?

Is a built-in object that represents a sequence of characters. It's one of the most widely used classes in Java

Key Characteristics of Strings:

1. **Immutable**: Once created, the contents of a **String** object cannot be altered. If you try to modify a **String**, a new **String** object is created instead.
2. **Concatenation**: You can combine Strings using the **+** operator, which creates a new **String** object containing the combined content.

```
String hello = "Hello, World!";
```

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName; // fullName: "John Doe"
```

String Methods

Strings come with a variety of methods for various operations like finding length, sub-strings, character at a specific index, and more.

length(): Returns the length of the string.

```
String str = "Hello";
int len = str.length(); // len = 5
```

trim(): This method returns a copy of the string, with leading and trailing whitespace omitted.

```
String str = "  Hello World!  ";
String trimmed = str.trim(); // trimmed = "Hello World!"
```

equals(Object anObject): Compares this string to the specified object. The result is true if and only if the argument is not null and is a String object that represents the same sequence of characters as this object.

```
String str1 = "Hello";
String str2 = "hello";
boolean result = str1.equals(str2); // result = false
```

String Methods

charAt(int index): Returns the char value at the specified index.

```
String str = "Hello";
char ch = str.charAt(1); // ch = 'e'
```

indexOf(String str): Returns the index within this string of the first occurrence of the specified substring.

```
String str = "Hello";
int index = str.indexOf("lo"); // index = 3
```

toLowerCase() and **toUpperCase()**: Converts all of the characters in this String to lower case or upper case, respectively.

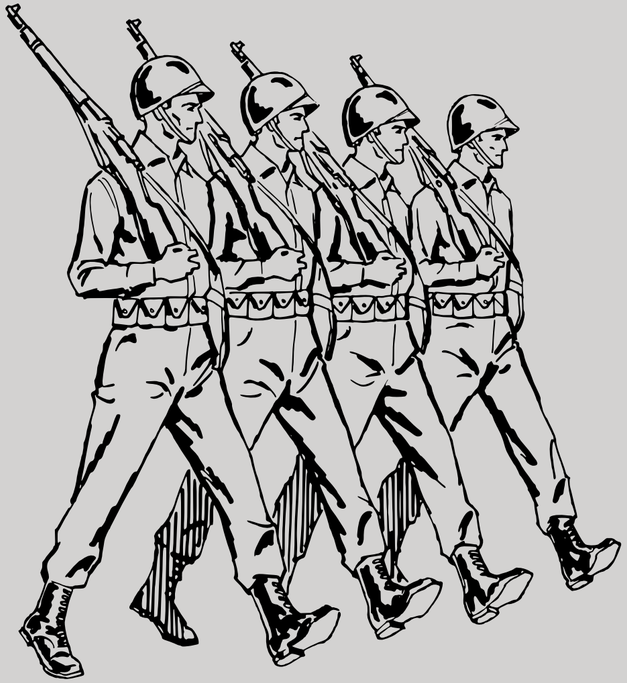
```
String str = "Hello";
String lower = str.toLowerCase(); // lower = "hello"
String upper = str.toUpperCase(); // upper = "HELLO"
```

split(String regex): This method splits the string around matches of the given regular expression. It returns an array of strings.

```
String str = "Java,Python,C++";
String[] languages = str.split(","); // languages = ["Java", "Python", "C++"]
```

More: https://docs.oracle.com/en/java/javase/17/docs/api/index.html

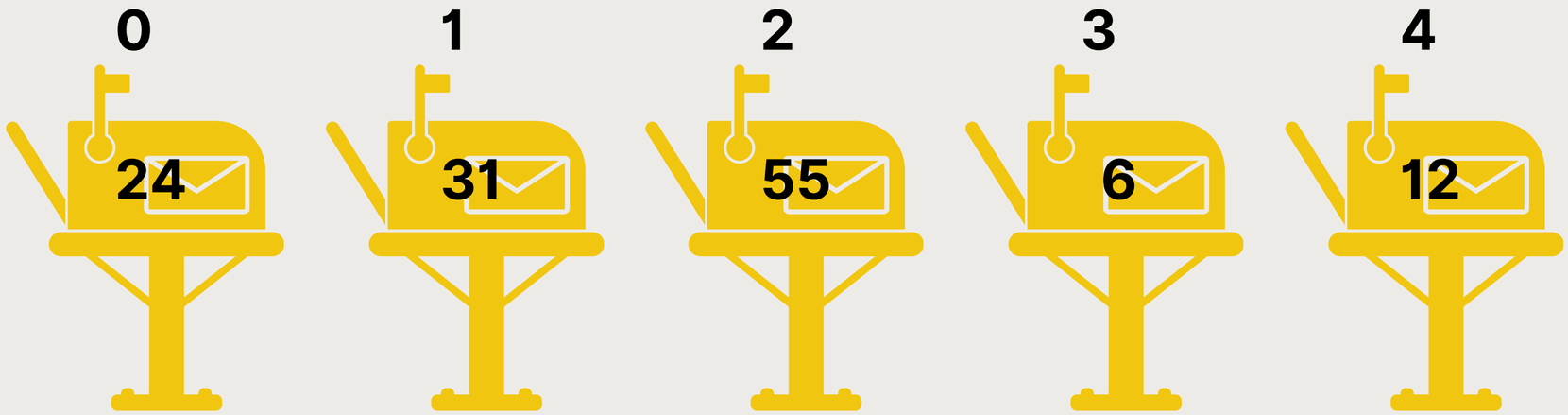
Arrays



Arrays

Are a way to store multiple items of the same type together. Think of them like a row of mailboxes, where each mailbox can hold a letter. In Java, each "mailbox" is an element of the array, and they all hold values of the same data type (like int, double, String, etc.).

- **Collection of Elements**: An array is a collection of elements, all of the same type.
- **Fixed Size**: Once created, the size of an array cannot be changed.
- **Indexed**: Each element in an array is accessed by an index. Indexes start at 0.



Array Examples

This creates an array named numbers that can hold 5 integers. Initially, all elements are set to 0.

```
int[] numbers = new int[5]; // An array to hold 5 integers
String[] names = {"Alice", "Bob", "Charlie"};
```

This creates and initializes an array of Strings with 3 names.

Assigning Values to an Array:

```
numbers[0] = 10; // First element (index 0) is now 10
numbers[1] = 20; // Second element (index 1) is now 20
```

Iterating Over an Array

```
for(int i = 0; i < numbers.length; i++) {
    System.out.println(numbers[i]);
}
```

Accessing Array Elements:

```
int firstNumber = numbers[0]; // Access the first element
```

```
int[] numbers = new int[5];
int length = numbers.length; // length will be 5
```

.length Property: Use **.length** on an array to get its size.

Getting Input

Introducing the Scanner class

The Scanner is a tool in Java that lets you read what the user types.

Importing: To use the Scanner class, you need to import it from **java.util**.

```
import java.util.Scanner;
```

Creating a Scanner Object: You create a Scanner object and associate it with a source like System.in (the standard input stream, typically the keyboard).

```
Scanner scanner = new Scanner(System.in);
```

Reading Input: The Scanner class provides various methods to read different types of input data (like nextInt, nextLine, nextDouble, etc.).

```
// Reading a string
String name = scanner.nextLine();

// Reading an integer
int age = scanner.nextInt();

// Reading a double
double salary = scanner.nextDouble();
```

Scanner Full Example

This program will ask the user to input their name and age, and then it will print these values back to the console.

```java
import java.util.Scanner;

public class InputExample {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        // Prompting and reading a string
        System.out.print("Enter your name: ");
        String name = scanner.nextLine();
        System.out.println("Hello, " + name + "!");

        // Prompting and reading an integer
        System.out.print("Enter your age: ");
        int age = scanner.nextInt();
        System.out.println("You are " + age + " years old.");

        // Closing the scanner
        scanner.close();
    }
}
```

Any questions?

Jansen Ang