# Introduction to java programming

Java programming refers to the process of writing computer programs using the Java programming language. Java is a high-level, object-oriented programming language developed by Sun Microsystems (now owned by Oracle Corporation) in the mid-1990s. It is designed to be platform-independent, meaning that Java programs can run on any device or operating system that has a Java Virtual Machine (JVM) installed.

Java is widely used for developing a variety of applications, including web applications, mobile apps (Android), enterprise software, desktop applications, and more. Its popularity stems from its simplicity, portability, robustness, and security features.

Key features of Java programming include:

**Object-oriented**: Java is based on the object-oriented programming paradigm, which emphasizes the organization of code into reusable objects that interact with each other.

**Platform independence**: Java programs are compiled into bytecode, which can run on any device or operating system with a compatible JVM. This "write once, run anywhere" capability is one of Java's most significant advantages.

**Automatic memory management**: Java uses garbage collection to automatically manage memory, reducing the likelihood of memory leaks and simplifying memory management for developers.

**Rich standard library**: Java comes with a comprehensive standard library (Java API) that provides pre-built modules and classes for common tasks, such as networking, I/O, data structures, and more.

**Security**: Java includes built-in security features, such as sandboxing and class loading mechanisms, to prevent unauthorized access and execution of malicious code.

**Multi-threading support**: Java provides built-in support for concurrent programming through its multi-threading capabilities, allowing developers to create applications that can perform multiple tasks simultaneously.

# Keywords of java



# Building blocks of java

Building blocks are combined and utilized to create Java programs that accomplish various tasks and solve different problems. Understanding these fundamentals is essential for Java developers to write efficient and maintainable code.The building blocks of a Java program include:

**Classes and Objects**: Java is an object-oriented programming language, so classes and objects are fundamental building blocks.

***Object*** *-* Objects have states and behaviors.

Example: A dog has states - color, name, breed as well as behaviors – wagging the tail, barking, eating. An object is an instance of a class.

***Class -*** A class can be defined as a template/blueprint that describes the behavior/state that the object of its type support.

* A class is defined using the **class** keyword followed by the name of the class. Example:

|  |
| --- |
| public class MyClass {  // Class body  } |

A basic class may contains **Variable, methods, constructor.**

# Building blocks of Class in java

**Variables**: Variables are used to store data within a program. In Java, variables have a ***specific data type (e.g., int, double, String)*** and can hold different values.

**Data Types**: Java supports various data types, including primitive data types (e.g., int, double, boolean) and reference data types (e.g., String, arrays, objects).

A diagram of a data flow

Description automatically generated

**Methods (Functions)**: Methods are blocks of code that perform a specific task. They encapsulate behavior and can accept parameters and return values. Methods are defined within classes.

**Control Flow Statements**: Control flow statements determine the order in which statements are executed in a program. Examples include if-else statements, loops (for, while, do-while), and switch-case statements.

**Modifiers:** Modifiers are keywords that modify the behavior of classes, variables, and methods. Examples include public, private, protected, static, final, abstract, etc.

**Packages:** Packages are used to organize classes and interfaces into namespaces. They help in avoiding naming conflicts and provide a way to structure large-scale Java applications.

**Inheritance:** Inheritance allows a class (subclass) to inherit properties and behavior from another class (superclass). It promotes code reuse and supports the concept of "is-a" relationships.

**Interfaces:** Interfaces define a contract for classes to implement. They contain method signatures without implementations and allow for achieving abstraction and multiple inheritance in Java.

**Exception Handling:** Exception handling allows for graceful handling of runtime errors and exceptional conditions. Java provides try-catch blocks to handle exceptions.

**Arrays:** Arrays are used to store multiple values of the same data type in contiguous memory locations. They provide a way to work with collections of data efficiently.

**Comments:** Comments are used to document code and improve its readability. Java supports single-line (//) and multi-line (/\* \*/) comments.

#Basic syntax of Java Program

**Package Declaration :** A Java program may start with an optional package declaration, which organizes the classes into a package. If omitted, the classes belong to the default package.

|  |
| --- |
| package com.example.mypackage; |

**Import Statements:** Import statements are used to import classes or entire packages from other packages, making them accessible within the current program.

|  |
| --- |
| import java.util.Scanner; |

**Class Declaration:** Every Java program must contain at least one class. The class declaration includes the class keyword followed by the class name.

|  |
| --- |
| public class MyClass {  // class body  } |

**Main Method:** The main method is the entry point of a Java program. It has a specific signature: public static void main(String[] args). This is where the execution of the program begins.

|  |  |  |
| --- | --- | --- |
| public static void main(String[] args) {  // main method body  }   |  | | --- | | A diagram of a diagram  Description automatically generated | | A diagram of a method  Description automatically generated | |

**Statements and Expressions:** Inside the main method or other methods, statements and expressions are written to perform actions and computations.

|  |
| --- |
| int x = 5; // **Expression**  System.out.println("Hello, world!");// **Statement**  #Note :  In Java, **System.out.print** and **System.out.println** are both used to display output to the console, but they have differences in behavior. |

**Comments:** Comments are used to document the code for better understanding. Java supports single-line comments (//) and multi-line comments (/\* \*/).

|  |
| --- |
| // This is a single-line comment  /\* This is a  multi-line comment \*/ |

**Whitespace:** Whitespace (spaces, tabs, line breaks) is used to separate tokens and improve code readability. It is ignored by the compiler.

**Semicolons:** Statements in Java must end with semicolons (;). This indicates the end of a statement.

|  |
| --- |
| int x = 5; // Semicolon terminates the statement |

# Java basic programming

Putting it all together, a basic Java program syntax mentioned above, might look like this:

This program declares a class named MyClass with a main method, inside of which it declares an integer variable x and prints "Hello, world!" to the console.

|  |
| --- |
| package com.example.mypackage;  import java.util.Scanner;  public class MyClass {  public static void main(String[] args) {  int x = 5;  System.out.println("Hello, world!");  }  } |

# Variables : Detailed description

|  |
| --- |
| In Java, variables can be categorized into several types based on their scope, lifetime, and usage. Here are the main types of variables in Java:   1. **Local Variables**:    * Local variables are declared within a method, constructor, or block and are accessible only within that scope.    * They are created when the method, constructor, or block is entered and destroyed when it exits.    * Example:   public void myMethod() { int x = 5; // Local variable // x can only be accessed within this method }   1. **Instance Variables (Non-static Variables)**:    * Instance variables are declared within a class but outside any method, constructor, or block.    * They are associated with instances (objects) of the class and have a separate copy for each instance.    * Instance variables are initialized with default values if not explicitly initialized.    * Example:   public class MyClass { int x; // Instance variable // x is associated with each instance of MyClass }   1. **Static Variables (Class Variables)**:    * Static variables are declared with the **static** keyword within a class but outside any method, constructor, or block.    * They are associated with the class rather than with any particular instance (object) of the class.    * Static variables are initialized with default values if not explicitly initialized.    * Example:   public class MyClass { static int count; // Static variable // count is associated with the class MyClass }   1. **Final Variables**:    * Final variables, declared with the **final** keyword, are constants whose values cannot be changed once initialized.    * They must be initialized either at the time of declaration or within the constructor of the class.    * Example:   public class MyClass { final int MAX\_VALUE = 100; // Final variable // MAX\_VALUE cannot be changed once initialized } |

# Methods(functions) in Java : Detailed description

|  |
| --- |
| In Java, methods are blocks of code that perform specific tasks or operations. They are encapsulated within classes and are used to define the behavior of objects or to provide reusable functionality. Here are the key aspects of methods in Java:   1. **Method Signature**:    * A method signature consists of the method name and its parameter list (if any). It specifies the unique identifier for the method within its class.    * Example:   javaCopy code  public void myMethod(int x, String y) { // Method body }   * + In this example, **myMethod** is the method name, and **(int x, String y)** is the parameter list.  1. **Return Type**:    * Methods in Java may return a value of a specific data type, or they may return **void**, indicating that they do not return any value.    * Example:   javaCopy code  public int calculateSum(int a, int b) { return a + b; }   1. **Parameters**:    * Parameters are variables that are declared in the method signature and are used to pass data to the method.    * Methods can have zero or more parameters, and each parameter must have a data type and a name.    * Example:   javaCopy code  public void printMessage(String message) { System.out.println(message); }   1. **Method Body**:    * The method body contains the code that defines the behavior of the method. It is enclosed within curly braces **{}**.    * This is where the actual logic of the method is written.    * Example:   javaCopy code  public int calculateSum(int a, int b) { int sum = a + b; return sum; }   1. **Access Modifiers**:    * Access modifiers control the visibility of methods. They determine which classes can access the method.    * The main access modifiers in Java are **public**, **private**, **protected**, and package-private (default).    * Example:   javaCopy code  public void myPublicMethod() { // Public method accessible from anywhere } private void myPrivateMethod() { // Private method accessible only within the same class }   1. **Static Methods**:    * Static methods belong to the class rather than to any specific instance of the class.    * They can be called directly on the class without creating an instance of the class.    * Example:   javaCopy code  public static void myStaticMethod() { // Static method implementation }   1. **Method Overloading**:    * Method overloading allows a class to have multiple methods with the same name but different parameter lists.    * The compiler determines which method to call based on the number and types of arguments passed.    * Example:   javaCopy code  public int calculateSum(int a, int b) { return a + b; } public double calculateSum(double a, double b) { return a + b; } |

# constructor

# java modifiers

#Object: Detailed Discription

1: Selecting a Class:

Determine which class you want to instantiate to create objects. Classes define the structure and behavior of objects.

2 : Instantiation:

To create an object, use the new keyword followed by the class name and parentheses (). This invokes the constructor of the class to initialize the object.

|  |
| --- |
| ClassName objectName = new ClassName(); |

3: Constructor Invocation using object

* When an object is created, its constructor is invoked to initialize the object's state. Constructors may accept parameters to initialize the object with specific values.
* If the class has multiple constructors, you can choose the appropriate constructor based on the initialization requirements.

|  |
| --- |
| public class Car {  String color;  // Constructor with parameters  public Car(String initialColor) {  color = initialColor;  }  }  // Creating an object with constructor parameters  Car myCar = new Car("blue"); |

4: Method Invocation using object

|  |
| --- |
| public class MyClass {  public void displayMessage() {  System.out.println("Hello, world!");  }  public int add(int a, int b) {  return a + b;  }  public static void main(String[] args) {  // Creating an object of MyClass  MyClass obj = new MyClass();  // Invoking displayMessage() method  obj.displayMessage(); // Output: Hello, world!  // Invoking add() method with arguments  int sum = obj.add(3, 5);  System.out.println("Sum: " + sum); // Output: Sum: 8  }  } |

#Object oriented concepts

The main ideas behind Java's Object-Oriented Programming, OOP concepts include abstraction, encapsulation, inheritance and polymorphism. Basically, Java OOP concepts let us create working methods and variables, then re-use all or part of them without compromising security.

A diagram of a class

Description automatically generated

Tutorial 2 (next week)

# Encapsulation:

Encapsulation is often introduced first because it forms the foundation of OOP principles. It emphasizes the bundling of data (attributes) and methods (behaviors) into a single unit (class), allowing for data hiding and controlled access.

Understanding encapsulation helps learners grasp the importance of data abstraction and how to create classes with well-defined interfaces.

#Inheritance:

Inheritance allows classes to inherit properties and behaviors from other classes, promoting code reuse and establishing "is-a" relationships between classes.

Understanding inheritance builds on encapsulation by showing how classes can be organized hierarchically, with more specialized classes inheriting from more general classes.

#Polymorphism:

Polymorphism can be introduced after encapsulation and inheritance because it builds on these concepts. Polymorphism allows objects of different classes to be treated as objects of a common superclass, enabling flexibility and extensibility in code.

Understanding polymorphism helps learners see how different classes can share common interfaces and be used interchangeably, enhancing code readability and maintainability.