# JAVA PROGRAM STRUCTURE

*From the entry point method to the full anatomy of a class – a guided tour of how Java programs are organized and why.*

## 01: The Entry Point

Every Java application begins execution at a single, specific method. That method has a fixed signature the Java Virtual Machine (JVM) looks for when it launches your program.

```
public static void main(String[] args)
```

This is called the **main method signature** – or more broadly, a method declaration. Each word in it carries a specific meaning.

Let’s break it into smaller pieces.

| | |
|---|---|
| `Public` | **Access modifier.** Makes this method visible from anywhere. The JVM must be able to call it from outside your class, so public is required. |
| `Static` | **Class-level method.** Belongs to the class itself, not to any object Instance. The JVM can call it without creating an object first. |
| void | **Return type.** Means the method returns nothing when it finishes. Other methods might return an int, a String, etc. |
| `main` | **Method name.** This exact name is the JVM's designated entry point. When you run a  Java program, execution begins here. |
| String[] args | **Parameter.** An array of Strings passed in from the command line. Running java MyApp hello world gives args[0] = "hello" and args[1] = "world". |

## 02: The Anatomy of a Java Program

Java code is organized into a strict hierarchy. From the outermost container down to individual logic, every element has a defined place.

```
Package Declaration Import Statements Class Declaration {
     Fields (variables)
     Constructors
     Methods {
          Statements & Logic
     }
     Nested Classes (optional)
}
```

### Package Declaration

Optionally the very first line of a file. It places the class inside a named namespace, helping organize large codebases and prevent naming conflicts. Example: package com.myapp.animals;

### Import Statements

Tell the compiler which classes from other packages you want to use. Without an import, you'd have to write the full path every time. Example: import java.util.ArrayList;

### Class Declaration

In Java, *everything* lives inside a class. The file must be named after the public class it contains. The class is the blueprint; objects are the instances built from it.

### Fields

Variables declared at the class level. They represent the *state* of an object — things it knows about itself, like a name, age, or count.

### Constructors

Special methods that run when a new object is created with new. They share the class name and have no return type. If you don't write one, Java provides a default empty constructor automatically. Multiple constructors with different parameters are allowed — this is called *constructor overloading*.

### Methods

Where behavior lives. Methods define what an object can *do*. The main method is just one example — programs typically contain many methods, each responsible for a specific task.

## 03: A Complete Example: dog.java

```
// PACKAGE DECLARATION
package com.myapp.animals;

// IMPORT STATEMENTS
import java.util.ArrayList;
import java.util.List;

// CLASS DECLARATION
public class Dog {

    // FIELDS
    private String name;
    private String breed;
    private int age;

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

    // METHODS
    public String getName() {
        return name;
    }

    public void bark() {
        System.out.println(name + " says: Woof!");
    }

    public String toString() {
        return name + " is a " + breed + " who is " + age + "
years old.";
    }

    // MAIN METHOD — entry point
    public static void main(String[] args) {
        Dog myDog = new Dog("Rex", "Labrador", 4);
        myDog.bark();
        System.out.println(myDog.toString());
    }
}
```

**Output:** Running this program prints:

```
Rex says: Woof!
Rex is a Labrador who is 4 years old.
```

A few things worth noting in this example.

The **this** keyword inside the constructor refers to the current object — it distinguishes between the field and the parameter when they share the same name.

The **toString()** method is inherited from Java's base Object class and returns a string representation of the object.

And **myDog** is an *instance* of Dog, created using the new keyword and the constructor.

## 04: Core Java Keywords

Java reserves certain words for the language itself. These are the keywords you'll encounter most often when writing a program like the one above:

**Access Modifiers:**

| | |
|---|---|
| `public` | Accessible from anywhere |
| `private` | Accessible only within the same class |
| `protected` | Accessible within the same package and subclasses |

**Class & Object Keywords:**

| | |
|---|---|
| `class` | Declares a class |
| `new` | Creates a new instance of a class |
| `this` | Refers to the current object |
| `extends` | Inherits from another class |
| `Implements` | Implements an interface |

**Method & Variable Modifiers:**

| | |
|---|---|
| `static` | Belongs to the class, not an instance |
| `final` | Makes a variable constant; prevents method or class from being overridden |
| `void` | Indicates a method returns no value |
| `return` | Exits a method and optionally returns a value |

**Primitive Data Types:**

| | |
|---|---|
| `int` | Whole number |
| `double` | Decimal number |
| `boolean` | True or false |
| `char` | A single character |
| `long` | Other numeric types |
| `float` | “ |
| `byte` | “ |
| `short` | “ |

**Control Flow:**

if / elseConditional branching
for / while / doLoops
switch / case / breakBranching based on a value
continueSkips to the next loop iteration

(continued on next page)

**Exception Handling:**

| | |
|---|---|
| try / | Handle errors gracefully |
| catch / | “ |
| finally | “ |
| throw | Manually throws an exception |
| throws | Declares a method might throw an exception |

**Package & Import:**

| | |
|---|---|
| package | Declares which package the class belongs to |
| import | Brings in external classes or packages for use |

## 05: The Four Pillars of Object Oriented Programming

Java is fundamentally an *Object-Oriented Programming* language. The structure of every Java program — classes, fields, methods, access modifiers — exists to support four core principles:

**Encapsulation**
Bundling data (fields) and behavior (methods) inside a class, and controlling access with modifiers like private. Protects internal state from unintended modification.

**Inheritance**
One class can extend another, gaining its fields and methods. A GoldenRetriever class extending Dog inherits everything from Dog automatically.

**Polymorphism**
Objects of different classes can be treated as instances of a shared parent type. A method can behave differently depending on the actual object it's called on.

**Abstraction**
Hiding implementation complexity behind clean interfaces. Interfaces and abstract classes define *what* a class must do, without specifying *how* it does it.

Understanding these four principles explains *why* Java is structured the way it is. Access modifiers exist for encapsulation. extends and implements exist for inheritance and abstraction. Everything connects.

## 06: Important Distinctions to Know:

### Primitives vs. Reference Types:

Java has two categories of data types. *Primitive types* — like int, boolean, and char — store their values directly in memory. *Reference types* — like String and all objects — store a reference (pointer) to where the data actually lives. This distinction affects how variables are copied, compared, and passed into methods.

### Classes vs. Interfaces vs. Abstract Classes:

A *class* is a full blueprint for objects. An *interface* defines a contract — a list of methods that any implementing class must provide — but contains no data of its own. An *abstract class* is a middle ground: it can have both implemented methods and abstract ones that subclasses must fill in. These tools are central to designing flexible, maintainable Java programs.

### The Java Standard Library:

When you write System.out.println(), you're calling into Java's built-in standard library — a vast collection of pre-written classes for things like math, dates, file I/O, networking, data structures, and more. The import statements at the top of a file are your gateway to this library. You don't have to write everything from scratch; Java ships with an enormous toolkit ready to use.

## 07 — Where to Go Next:

With a solid grasp of Java's structure, these topics are natural next steps:

- **Interfaces & Abstract Classes** — the backbone of flexible Java design
- **Collections & Data Structures** — ArrayList, HashMap, and the Java Collections Framework
- **Exception Handling in Depth** — try/catch/finally with real code examples
- **Inheritance & Polymorphism** — extending classes and overriding methods
- **Constructor Overloading** — multiple constructors for flexible object creation
- **The Java Standard Library** — exploring what's available out of the box
- **Generics** — writing classes and methods that work with any data type safely