ZOLAXTECH

zolaxtech.com.et

THE COMPLETE JAVA PROGRAMMING GUIDE

From Your First Line of Code to Advanced Software Engineering

Beginner • Intermediate • Advanced

Table of Contents

Welcome to the Zolaxtech Java Guide

This guide takes you from a completely blank editor to writing real, working Java software. It is organized into five parts that build on each other, so work through them in order if you are new to programming or jump to the part you need if you already have some experience.

Every chapter combines a Plain-English explanation, a runnable code example, and practical tips drawn from real-world Java development. By the end, you will understand not just Java syntax, but how professional Java programs are structured, tested, and shipped.

What You Will Learn

- The fundamentals: variables, data types, operators, and control flow
- Object-Oriented Programming: classes, objects, inheritance, polymorphism, abstraction
- Core data structures: arrays, Strings, and the Java Collections Framework
- Robust programs: exception handling and file I/O
- Modern Java: lambda expressions, the Stream API, and functional-style code
- Concurrency: threads and the basics of multithreaded programs
- Advanced topics: generics, design patterns, and working with databases (JDBC)
- Professional habits: naming conventions, project structure, and where to go next

> **TIP**
> Install a Java Development Kit (JDK 17 or later) and an editor such as IntelliJ IDEA Community Edition or VS Code before you begin. Chapter 2 walks through this step by step.

PART ONE

Getting Started

Understanding what Java is and preparing your machine to write and run it.

1. Introduction to Java

1.1 What Is Java?

Java is a general-purpose, object-oriented programming language created by Sun Microsystems in 1995 (now maintained by Oracle). It was designed around one guiding idea: "Write Once, Run Anywhere." Java code is compiled into an intermediate form called bytecode, which runs on the Java Virtual Machine (JVM) meaning the same compiled program runs unchanged on Windows, macOS, or Linux.

Java is used to build almost everything: Android apps, enterprise back-end systems, banking software, e-commerce platforms, big-data tools (like Hadoop and Kafka), and cloud services.

1.2 Why Learn Java?

- Huge job market Java remains one of the top languages requested by employers worldwide.
- Strong typing catches many mistakes before your program even runs.
- A massive standard library and ecosystem (Spring, Hibernate, Maven, Gradle).
- Excellent for learning solid OOP fundamentals that transfer to other languages.
- Runs anywhere the JVM is installed servers, desktops, and Android devices.

1.3 How Java Code Runs

Understanding the Java workflow helps everything else make sense:

1. You write source code in a file ending with .java
2. The Java compiler (javac) turns it into bytecode (a .class file)
3. The JVM loads and executes that bytecode, translating it into instructions your specific computer understands

```
MyProgram.java  --(javac)-->  MyProgram.class  --(JVM)-->  Program runs
```

2. Setting Up Your Environment

2.1 Install the JDK

Download the latest Long-Term Support (LTS) release of the Java Development Kit for example JDK 21 from Oracle or the free Eclipse Temurin distribution. The JDK includes the compiler (javac), the runtime (JVM), and the core libraries.

2.2 Verify the Installation

Open a terminal or command prompt and run:

```
java -version
javac -version
```

If both commands print a version number, you're ready to go.

2.3 Choose an Editor

- IntelliJ IDEA (Community Edition) the most popular free Java IDE, excellent autocomplete and debugging
- Visual Studio Code with the "Extension Pack for Java" lightweight and flexible
- Eclipse a long-standing, free, full-featured IDE

2.4 Your First Program

Create a file named HelloWorld.java with the following content:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, Zolaxtech!");
    }
}
```

Compile and run it from the terminal:

```
javac HelloWorld.java
java HelloWorld
```

> **COMMON MISTAKE**
> The file name must exactly match the public class name (HelloWorld.java for `public class HelloWorld`), including capitalization this is one of the most common beginner errors.

PART TWO

Java Fundamentals

Variables, data types, operators, and the logic that controls your program.

3. Variables and Data Types

3.1 Declaring Variables

A variable is a named container for a value. Java is statically typed, meaning you must declare the type of data a variable will hold.

```
int age = 25;
double price = 19.99;
char grade = 'A';
boolean isActive = true;
String name = "Selam";
```

3.2 Primitive Data Types

- byte 8-bit whole number (-128 to 127)
- short 16-bit whole number
- int 32-bit whole number (most commonly used for integers)
- long 64-bit whole number (append L, e.g. 10000000000L)
- float 32-bit decimal number (append f)
- double 64-bit decimal number (default for decimals)
- char a single 16-bit character, e.g. 'A'
- boolean true or false

3.3 Reference Types

Everything that is not a primitive is a reference type objects such as String, arrays, and instances of your own classes. Reference variables store a pointer to an object in memory, not the object itself.

> **TIP**
> Use meaningful variable names (totalPrice, not tp) and follow Java's camelCase convention for variables and methods, and PascalCase for class names.

4. Operators

4.1 Arithmetic Operators

```
int sum = 10 + 5;   // 15
int diff = 10 - 5;  // 5
int prod = 10 * 5;  // 50
int quot = 10 / 3;  // 3  (integer division)
int rem  = 10 % 3;  // 1  (modulus / remainder)
```

4.2 Comparison and Logical Operators

```
a == b   // equal to
a != b   // not equal to
a > b, a < b, a >= b, a <= b
a && b   // logical AND
a || b   // logical OR
!a       // logical NOT
```

4.3 Assignment Shortcuts

```
x += 5;   // x = x + 5
x -= 5;   // x = x - 5
x *= 2;   // x = x * 2
x++;      // increment by 1
x--;      // decrement by 1
```

5. Control Flow

5.1 if / else

```
int score = 82;
if (score >= 90) {
    System.out.println("Grade: A");
} else if (score >= 80) {
    System.out.println("Grade: B");
} else {
    System.out.println("Grade: C or below");
}
```

5.2 switch Statement

```
int day = 3;
switch (day) {
    case 1 -> System.out.println("Monday");
    case 2 -> System.out.println("Tuesday");
    case 3 -> System.out.println("Wednesday");
    default -> System.out.println("Another day");
}
```

5.3 Loops

The for loop, ideal when you know how many times to repeat:

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

The while loop, ideal when repetition depends on a condition:

```
int n = 5;
while (n > 0) {
    System.out.println(n);
    n--;
}
```

The enhanced for-each loop, ideal for iterating over collections and arrays:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int num : numbers) {
    System.out.println(num);
}
```

6. Arrays and Strings

6.1 Arrays

An array is a fixed-size, ordered collection of values of the same type.

```
int[] scores = {90, 85, 77, 92};
System.out.println(scores[0]);      // 90
System.out.println(scores.length);  // 4

int[][] grid = new int[3][3];       // 2D array
```

6.2 Strings

A String represents text. Strings in Java are immutable once created, their content never changes; methods like replace() return a new String.

```
String greeting = "Hello";
String full = greeting + ", World!";  // concatenation
System.out.println(full.length());     // 13
System.out.println(full.toUpperCase());
System.out.println(full.substring(0, 5));
System.out.println(full.contains("World"));
```

> **TIP**
> For building strings inside loops, use StringBuilder instead of + concatenation it is far more efficient because it avoids creating a new String object on every step.

PART THREE

Object-Oriented Programming

The heart of Java modeling real-world things as classes and objects.

7. Classes and Objects

7.1 The Core Idea

A class is a blueprint that defines properties (fields) and behaviors (methods). An object is a specific instance created from that blueprint. If Car is the class, your specific red Toyota is the object.

```
public class Car {
    // Fields
    String model;
    int year;

    // Constructor
    public Car(String model, int year) {
        this.model = model;
        this.year = year;
    }

    // Method
    public void displayInfo() {
        System.out.println(year + " " + model);
    }
}
```

Creating and using an object:

```
Car myCar = new Car("Corolla", 2023);
myCar.displayInfo();   // 2023 Corolla
```

7.2 Constructors

A constructor is a special method that runs when an object is created, typically used to initialize its fields. A class can have multiple constructors (constructor overloading) as long as their parameter lists differ.

8. The Four Pillars of OOP

8.1 Encapsulation

Encapsulation means keeping a class's fields private and exposing controlled access through public getter and setter methods. This protects an object's internal state from invalid changes.

```
public class BankAccount {
    private double balance;

    public double getBalance() {
        return balance;
    }

    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }
}
```

8.2 Inheritance

Inheritance lets one class (the subclass) reuse and extend the fields and methods of another (the superclass) using the extends keyword.

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

public class Dog extends Animal {
    public void bark() {
        System.out.println("Woof!");
    }
}

Dog d = new Dog();
d.eat();   // inherited from Animal
d.bark();  // defined in Dog
```

8.3 Polymorphism

Polymorphism means "many forms" a subclass can override a method to provide its own behavior, and Java decides at runtime which version to call.

```
public class Animal {
    public void makeSound() {
        System.out.println("Some generic sound");
    }
}
public class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}

Animal a = new Cat();
a.makeSound();  // "Meow"  decided at runtime
```

8.4 Abstraction

Abstraction hides implementation detail behind a simple interface. Java provides two tools for this: abstract classes and interfaces.

```
public interface Shape {
    double area();   // no implementation  a contract
}

public class Circle implements Shape {
    private double radius;
    public Circle(double radius) { this.radius = radius; }

    @Override
    public double area() {
        return Math.PI * radius * radius;
    }
}
```

COMMON MISTAKE

A class can extend only one other class, but it can implement multiple interfaces. When designing shared behavior across unrelated classes, prefer interfaces.

PART FOUR

Working with Data and Errors

Collections, exception handling, and reading and writing files.

9. The Collections Framework

9.1 Why Collections?

Arrays have a fixed size. The Collections Framework provides flexible, resizable data structures for storing groups of objects.

9.2 ArrayList

A resizable, ordered list the most commonly used collection in Java.

```
import java.util.ArrayList;
import java.util.List;

List<String> names = new ArrayList<>();
names.add("Abel");
names.add("Hana");
names.remove("Abel");
System.out.println(names.size());  // 1
for (String n : names) {
    System.out.println(n);
}
```

9.3 HashMap

Stores key-value pairs for fast lookup by key.

```
import java.util.HashMap;
import java.util.Map;

Map<String, Integer> ages = new HashMap<>();
ages.put("Selam", 28);
ages.put("Dawit", 34);
System.out.println(ages.get("Selam"));  // 28
```

9.4 HashSet

Stores unique values with no defined order duplicate additions are silently ignored.

```
import java.util.HashSet;
import java.util.Set;

Set<Integer> uniqueNumbers = new HashSet<>();
uniqueNumbers.add(1);
uniqueNumbers.add(1);   // ignored, already present
System.out.println(uniqueNumbers.size());  // 1
```

> **TIP**
> Choose List when order and duplicates matter, Set when you need uniqueness, and Map when you need fast key-based lookup.

10. Exception Handling

10.1 Why Exceptions Matter

Exceptions are Java's mechanism for handling runtime errors gracefully, without crashing the entire program.

```
try {
    int result = 10 / 0;
} catch (ArithmeticException e) {
    System.out.println("Error: " + e.getMessage());
} finally {
    System.out.println("This always runs.");
}
```

10.2 Checked vs. Unchecked Exceptions

- Checked exceptions (e.g. IOException) must be caught or declared with throws the compiler enforces this.
- Unchecked exceptions (e.g. NullPointerException, ArithmeticException) extend RuntimeException and are not compiler-enforced.

10.3 Custom Exceptions

```
public class InvalidAgeException extends Exception {
    public InvalidAgeException(String message) {
        super(message);
    }
}

public void setAge(int age) throws InvalidAgeException {
    if (age < 0) {
        throw new InvalidAgeException("Age cannot be negative");
    }
}
```

11. Working with Files

11.1 Reading a File

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;

List<String> lines = Files.readAllLines(Path.of("data.txt"));
for (String line : lines) {
    System.out.println(line);
}
```

11.2 Writing a File

```
import java.nio.file.Files;
import java.nio.file.Path;

Files.writeString(Path.of("output.txt"), "Hello from Zolaxtech!");
```

> **COMMON MISTAKE**
> File operations can throw a checked IOException always handle it with try-catch or declare it with throws.

PART FIVE

Modern and Advanced Java

Lambdas, streams, concurrency, generics, and where to go from here.

12. Lambda Expressions and the Stream API

12.1 Lambda Expressions

Introduced in Java 8, lambdas let you write compact, function-like blocks of code, most often used with functional interfaces (interfaces with a single abstract method).

```
Runnable task = () -> System.out.println("Running!");
task.run();

// A lambda implementing Comparator
List<String> names = new ArrayList<>(List.of("Zara", "Abel", "Meles"));
names.sort((a, b) -> a.compareTo(b));
```

12.2 The Stream API

Streams let you process collections in a declarative, pipeline style: filter, transform, and collect data without writing manual loops.

```
import java.util.List;
import java.util.stream.Collectors;

List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8);

List<Integer> evenSquares = numbers.stream()
        .filter(n -> n % 2 == 0)
        .map(n -> n * n)
        .collect(Collectors.toList());

System.out.println(evenSquares);  // [4, 16, 36, 64]
```

13. Multithreading Basics

13.1 Creating a Thread

Multithreading lets a program perform multiple tasks concurrently, which is essential for responsive applications and efficient use of multi-core processors.

```
Runnable task = () -> {
    for (int i = 1; i <= 3; i++) {
        System.out.println("Working: " + i);
    }
};

Thread thread = new Thread(task);
thread.start();
```

13.2 Synchronization

When multiple threads access shared data, use the synchronized keyword to prevent race conditions.

```
public synchronized void increment() {
    counter++;
}
```

> **TIP**
> For most real applications, prefer the higher-level java.util.concurrent package (ExecutorService, CompletableFuture) over managing raw Thread objects directly.

14. Generics

Generics let you write classes and methods that work with any type while still catching type errors at compile time, instead of at runtime.

```
public class Box<T> {
    private T content;

    public void set(T content) { this.content = content; }
    public T get() { return content; }
}

Box<String> stringBox = new Box<>();
```

```
stringBox.set("Hello");
System.out.println(stringBox.get());

Box<Integer> intBox = new Box<>();
intBox.set(42);
```

15. Connecting to a Database with JDBC

JDBC (Java Database Connectivity) is the standard API for connecting Java applications to relational databases such as MySQL or PostgreSQL.

```
import java.sql.*;

String url = "jdbc:mysql://localhost:3306/mydb";
try (Connection conn = DriverManager.getConnection(url, "user", "password");
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery("SELECT * FROM users")) {

    while (rs.next()) {
        System.out.println(rs.getString("name"));
    }
} catch (SQLException e) {
    e.printStackTrace();
}
```

> **TIP**
> In real projects, most teams use an ORM framework such as Hibernate, or Spring Data JPA, to avoid writing raw SQL and JDBC boilerplate by hand.

16. Design Patterns You Should Know

Singleton

Ensures a class has only one instance and provides a single, global access point to it commonly used for configuration objects and connection managers.

Factory

Centralizes object creation logic in one place, so calling code doesn't need to know the exact class being instantiated.

Observer

Let's objects (observers) subscribe to and react to events from another object (the subject) the foundation of most event-driven and GUI systems.

Builder

Constructs complex objects step by step, useful when a class has many optional constructor parameters.

17. Best Practices and Professional Habits

- Follow naming conventions: PascalCase for classes, camelCase for methods and variables, UPPER_SNAKE_CASE for constants.
- Keep methods short and focused on a single responsibility.
- Favor composition over inheritance when it produces a simpler design.
- Write unit tests (JUnit) as you build, not after.
- Use a build tool Maven or Gradle to manage dependencies and builds.
- Use version control (Git) for every project, from day one.
- Read and write Javadoc comments for anything other developers will use.
- Avoid catching exceptions you don't intend to handle meaningfully never leave an empty catch block.

18. Where to Go Next

You now have a solid foundation across the entire Java language, from syntax to advanced, real-world topics. Here is a suggested path to keep growing:

1. Build small projects a to-do list app, a simple bank simulator, a text-based game.
2. Learn a build tool deeply: Maven or Gradle.
3. Learn the Spring Framework (Spring Boot) for building web applications and REST APIs.
4. Practice data structures and algorithms this pays off in interviews and in writing efficient code.
5. Contribute to an open-source Java project on GitHub.
6. Keep exploring the resources and tutorials on zolaxtech.com.et.

Thank you for learning with
ZOLAXTECH

zolaxtech.com.et