PROGRAMMING IN JAVA

Lecture 1

# Introduction to Java

Java Technologies, JDK, JRE, JVM, Java Platforms, First Program, and `main()`

**Lecturer: Arshyn Altybay**

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

## 1. Learning Objectives

By the end of this lecture, students should be able to:

- explain what Java is and identify major areas in which it is used;
- distinguish source code, bytecode, and machine-level execution;
- explain the roles of the JDK, JRE concept, and JVM;
- describe the compile–load–execute workflow of a Java program;
- identify the purpose of the Java compiler and launcher;
- explain the role of an IDE in Java development;
- write, compile, and run a simple Java program;
- explain the structure and purpose of `public static void main(String[] args)`;
- use comments and basic output statements;
- recognize common beginner compilation and runtime errors.

## 2. What Is Java?

Java is a high-level, class-based, strongly typed programming language and software platform. Java source code is written in human-readable form, compiled into platform-neutral bytecode, and executed by a Java Virtual Machine (JVM).

Java is widely used for:

- enterprise and business information systems;
- backend and server-side applications;
- desktop tools and utilities;
- data-processing systems;

- testing and automation tools;
- Android-related development through the broader Java ecosystem;
- scientific, educational, and research software.

> Java is not "platform independent" because the same native machine code runs everywhere. The portable part is primarily Java bytecode and the standard API contract; each operating system requires a compatible JVM implementation.

## 3. Why Java Became Important

Several design characteristics contributed to Java's long-term use.

| Characteristic | Meaning |
|---|---|
| **portability** | compiled bytecode can run on different systems with compatible JVMs |
| **static typing** | many type errors are detected before execution |
| **object orientation** | classes and objects support modular software design |
| **automatic memory management** | unreachable objects can be reclaimed by garbage collection |
| **rich standard library** | Java provides APIs for files, collections, networking, concurrency, databases, and more |
| **ecosystem** | IDEs, build tools, libraries, frameworks, and testing tools are widely available |
| **backward compatibility** | Java strongly values compatibility with existing applications |

## 4. A Brief Historical Context

Java was developed at Sun Microsystems in the 1990s. Early design goals included portability, safety, simpler memory management than lower-level languages, and support for networked software. Java later became part of Oracle's technology portfolio, while OpenJDK became the main open-source implementation project for the Java platform.

> **Historical note:** Browser applets played an important role in Java's early history, but they are obsolete today. Modern Java development focuses on standalone, server-side, desktop, cloud, data-processing, and other current application models.

## 5. The Java Development Environment

Three terms appear repeatedly in Java development: JVM, JRE, and JDK.

| Term | Role |
|---|---|
| **JVM** | executes Java bytecode and provides runtime services such as class loading and memory management |
| **JRE** | traditional conceptual name for the runtime environment: JVM plus runtime libraries needed to execute Java applications |
| **JDK** | Java Development Kit; includes the compiler, launcher, development tools, and runtime components |
| **IDE** | Integrated Development Environment used to edit, build, run, debug, and organize projects |

**Traditional relationship**

```
JDK
 |
 +-- development tools: javac, jar, javadoc, ...
 |
 +-- runtime components
      |
      +-- JVM
      +-- standard Java libraries
```

> In modern Java distributions, a separate end-user JRE is less commonly supplied. For teaching, the traditional JDK–JRE–JVM relationship remains useful as a conceptual model.

## 6. From Source Code to Program Execution

A Java source file normally has the extension `.java`. The compiler `javac` translates source code into bytecode, usually stored in a `.class` file. The `java` launcher starts the JVM and executes the compiled class.

**Java compilation and execution workflow**

```
Hello.java
   |
   |  javac Hello.java
```

```
   v
Hello.class
   |
   |  java Hello
   v
JVM loads, verifies, and executes bytecode
   |
   v
Program output
```

> **Key idea:** Compilation and execution are separate stages. A program can compile successfully and still fail later because of a runtime problem.

## 7. Checking the Java Installation

From a terminal or command prompt, the following commands are useful:

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

If both commands are available, the system can normally run and compile Java programs.

## 8. Your First Java Program

A Java application begins inside a class. The following program prints a message to the console.

**Example 1. Hello, Java**

```
1 public class Hello {
2     public static void main(String[] args) {
3         System.out.println("Hello, Java!");
4     }
5 }
```

**Expected output**

```
Hello, Java!
```

Save the file as:

```
Hello.java
```

Compile it:

```
javac Hello.java
```

Run it:

```
java Hello
```

## 9. Anatomy of the First Program

Consider the line:

```
public class Hello {
```

The keyword `class` declares a class. When a public top-level class is called `Hello`, the source file is normally named `Hello.java`.

The program entry point is:

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

Its parts are summarized below.

| Part | Meaning |
|---|---|
| `public` | allows the launcher to access the method |
| `static` | the method belongs to the class and can run without first creating an object |
| `void` | the method does not return a value |
| `main` | special method name used as the application entry point |
| `String[] args` | array of command-line arguments |

## 10. Console Output

The class `System` provides the standard output stream `System.out`.

**Example 2. Using `print` and `println`**

```
1 public class OutputExample {
2     public static void main(String[] args) {
3         System.out.print("Java ");
4         System.out.println("Programming");
5         System.out.println("Lecture 1");
6     }
7 }
```

**Expected output**

```
Java Programming
Lecture 1
```

`print()` keeps the cursor on the same line, while `println()` finishes the current line.

## 11. Comments

Comments document code and are ignored by the compiler.

```
// Single-line comment

/*
   Multi-line comment
*/

/**
 * Documentation comment.
 */
```

Use comments to explain *why* code exists or to clarify non-obvious logic. Avoid comments that simply repeat what an obvious statement already says.

## 12. Identifiers and Basic Naming Rules

An identifier is a programmer-defined name such as a class name, method name, or variable name.

Common conventions include:

- class names: `StudentRecord`, `BankAccount`;
- method and variable names: `calculateAverage`, `studentName`;
- constants: `MAX_SIZE`, `DEFAULT_PORT`;
- package names: lowercase, such as `edu.university.app`.

Java identifiers are case-sensitive. Therefore `Student` and `student` are different identifiers.

## 13. Compilation Errors and Runtime Errors

A compilation error prevents bytecode from being produced.

```
System.out.println("Hello")
```

The missing semicolon causes a compilation error.

A runtime error occurs after successful compilation.

```
int result = 10 / 0;
```

This compiles, but integer division by zero causes a runtime exception.

A logical error is different: the program runs but produces an incorrect result.

## 14. Working with an IDE

An IDE such as IntelliJ IDEA, Eclipse, or NetBeans typically provides:

- source-code editing and syntax highlighting;
- project organization;
- automatic compilation;
- run configurations;
- debugging;
- refactoring;
- integration with build and version-control tools.

> Students should understand the command-line workflow even if an IDE is used. Knowing what `javac` and `java` do makes IDE behavior much easier to understand.

# 15. Comprehensive Example: Program Information Card

**Example 3. A small structured console program**

```java
public class ProgramCard {
    public static void main(String[] args) {
        System.out.println("==============================");
        System.out.println("       PROGRAMMING IN JAVA");
        System.out.println("==============================");
        System.out.println("Language : Java");
        System.out.println("Compiler : javac");
        System.out.println("Runtime  : JVM");
        System.out.println("File     : ProgramCard.java");
        System.out.println("==============================");
    }
}
```

**Expected output**

```
==============================
       PROGRAMMING IN JAVA
==============================
Language : Java
Compiler : javac
Runtime  : JVM
File     : ProgramCard.java
==============================
```

# 16. Common Errors and Good Practices

- Match the public class name and source-file name.
- Remember that Java is case-sensitive.
- End ordinary statements with semicolons.
- Balance parentheses, braces, and quotation marks.
- Compile frequently so that errors are found early.
- Read compiler messages from the first reported error downward.
- Use meaningful names instead of unexplained abbreviations.
- Keep indentation consistent.

## 17. Lecture Summary

- Java source code is compiled into bytecode and executed by a JVM.
- The JDK provides the tools needed to develop Java applications.
- `javac` compiles source code and `java` launches compiled applications.
- A standard Java application begins in `public static void main(String[] args)`.
- Java is case-sensitive and uses classes as a fundamental program structure.
- Console programs commonly use `System.out.print()` and `System.out.println()`.
- Compilation, runtime, and logical errors are different categories of problems.

## 18. Review Questions

1. What is Java?
2. What is bytecode?
3. What is the role of the JVM?
4. What is the JDK?
5. How does the traditional JRE concept differ from the JDK?
6. What does `javac` do?
7. What does the `java` launcher do?
8. What is the purpose of `main()`?
9. Why is `main()` declared `static`?
10. What does `void` mean in the method declaration?
11. What is the difference between `print()` and `println()`?
12. Why must a public class normally match the source-file name?
13. What is a compilation error?
14. What is a runtime error?
15. What is a logical error?

## 19. Programming Exercises

1. Write a program that prints your name, university, and department.
2. Print a three-line welcome message using only one `println()` call.
3. Write a program that uses both `print()` and `println()`.
4. Create a class called `CourseInfo` and print the course title.
5. Compile and run a Java file from the command line.
6. Intentionally remove a semicolon, compile the program, and record the compiler message.
7. Intentionally misspell `System` and observe the compiler response.
8. Add single-line and multi-line comments to a Java program.
9. Print a simple text menu for a future student-management system.
10. Modify the comprehensive example to display your own application information.

## 20. Homework

Create a console application called `JavaProfile`. It must:

- print a formatted title;
- print at least six facts about Java;
- include at least one single-line and one multi-line comment;
- use both `print()` and `println()`;
- compile without warnings caused by syntax mistakes;
- be executed from the command line at least once.

Submit the source file and a screenshot or copied console output.

## 21. Key Terms

| Term | Meaning |
|---|---|
| **source code** | human-readable program text stored in a `.java` file |
| **compiler** | program that translates source code into another representation |
| **bytecode** | platform-neutral instructions produced by the Java compiler |
| **JVM** | Java Virtual Machine that loads and executes Java bytecode |
| **JDK** | development kit containing the compiler, launcher, tools, and runtime components |
| **class** | Java construct that defines a type and groups data and behavior |
| **method** | named block of executable code inside a class |
| **entry point** | method from which a standard Java application begins execution |
| **IDE** | integrated environment for editing, building, running, and debugging software |

> **Study recommendation:** Write, compile, and run every example yourself. The development workflow should become automatic before moving to more advanced language concepts.