book

2025-11-22

Generated 2025-11-22 01:26

Contents

- intro_to_java
- linking_java_file
- values_and_variables_java

intro_to_java

A Brief Introduction to Java (Java 21)

Introduction to Java

Java is a high-level, object-oriented, multi-paradigm programming language. In this introduction, we set the stage for understanding what Java is and what we can do with it.

Programming Language

A programming language is a tool we use to write instructions that a computer can execute. Java programs compile into bytecode that runs on the Java Virtual Machine (JVM), allowing the same program to work on many systems.

High-Level Language

Java is a high-level language, meaning we do not need to worry about lower-level details like memory manipulation. The JVM handles memory management, garbage collection, and optimizations, making Java easier to learn and safer to use.

Object-Oriented

Java is primarily object#oriented. Everything revolves around classes and objects, and most data structures or components are built using this model. Understanding classes, objects, inheritance, and encapsulation is essential for mastering Java.

Multi-Paradigm

While Java is strongly object#oriented, it also supports other paradigms such as functional-style programming through lambda expressions and the Stream API introduced in modern versions. This flexibility allows developers to choose the most effective style for their applications.

Role of Java in Software Development

Java is one of the most widely used technologies in software development. It enables the development of desktop applications, backend systems, enterprise-level services, Android applications, cloud systems, and more.

Java, combined with tools, frameworks, and libraries, powers servers, APIs, distributed systems, and large#scale business applications used worldwide.

Real Example: Enterprise Web Applications

Many enterprise systems use Java on the server to receive requests, process data, interact with databases, and send responses back to clients. Java can manage complex workflows, background tasks, message queues, and large datasets. It is optimized for reliability and high performance, making it ideal for enterprise development.

Recap and Further Possibilities

Java enables the creation of powerful applications across many platforms. With Java 21, developers gain access to modern features, performance improvements, and long#term support.

Frameworks built on Java (e.g., Spring, Jakarta EE) help developers build large, scalable applications efficiently, and all of these rely on solid core Java knowledge.

Mastering Java fundamentals is essential before working with these advanced technologies.

Java Beyond the Server

Java is not limited to backend systems. It runs on desktops, embedded devices, cloud environments, and Android systems. Thanks to the JVM, Java can be used anywhere the runtime is available.

Java for Desktop and Mobile

Developers can build desktop interfaces using JavaFX or Swing. For mobile development, Java is used heavily in the Android ecosystem, where it powers apps, services, and mobile logic.

Java Releases and Versions

Java uses a structured release cycle managed by Oracle and the OpenJDK community. Java 21 is a Long-Term Support (LTS) release, meaning it receives years of updates and is designed for production environments.

Older versions of Java introduced major features (e.g., generics, lambdas, records, switch improvements), and Java 21 continues with enhancements such as virtual threads, pattern matching, and performance upgrades.

This course content aligns with Java 21 features wherever applicable. If older Java versions differ in behavior or capabilities, those differences will be noted.

Key Takeaways

- Java is a high#level, object#oriented, multi#paradigm programming language.
- It powers backend systems, enterprise applications, Android apps, and desktop applications.
- Java 21 provides modern, stable features and long-term support.
- Mastering core Java concepts is essential before working with advanced frameworks and tools.

Linking a Java File (Java 21)

Introduction

In this lecture, we will learn how to write Java code in a separate file, organize it properly, and run it using standard Java tools. This mirrors how real Java development is done.

Downloading the Starter Code

To get started, download the starter project from the provided repository for this course. You will receive a ZIP file that contains folders for each section. Inside each section, you will find both a **starter** folder and a **final** folder.

The **final** folder contains the completed version of the code as it should look when you finish the section. If something goes wrong, you can compare your work with the final version.

The **starter** folder is your working folder. Use this folder as your project directory.

Extracting and Organizing the Starter Code

After downloading the ZIP file, extract it. Move the starter folder for the current section to your desktop. This folder contains your Java source files, project structure, and configuration files.

Inside the starter folder, you will typically find a `src` directory containing your Java files.

Opening the Project in VS Code

Open the starter folder in VS Code. Java is not tied to HTML files, so you work directly with `.java` files inside the `src` folder.

When you open the project, you will see at least one Java file, commonly named `Main.java`. This is the entry point of your Java application.

Writing Java in a File

Java code must live inside a `.java` file and inside a class. You cannot execute Java directly in a browser console or HTML file.

Let’s write a simple example in `Main.java`.

Java Code Sample

```
public class Main {
    public static void main(String[] args) {
        String status = "amazing";

        if (status.equals("amazing")) {
            System.out.println("Java is fun!");
        }
    }
}
```

To run this program, you must compile it and then execute the compiled bytecode.

Running Java Code Using the Terminal

Java does not run in the browser, so you use the terminal.

To compile:

```
javac Main.java
```

To run:

```
java Main
```

This will print `"Java is fun!"` to the terminal.

Using System.out.println

If you write a mathematical expression in Java without printing it, nothing appears in the output. Just like JavaScript requires `console.log`, Java requires `System.out.println`.

Java Code Sample

```
System.out.println(40 + 8 + 23 - 10);
```

This will print the result to your terminal when you run the program.

Recap: Console vs Java Files

Unlike JavaScript, Java has **no script tag**, and it does not run inside HTML. Everything must be written in `.java` files and executed through the JVM.

Outputs appear only when you explicitly print them using `System.out.println`.

Moving Code to Another Java File

If you want to cleanly separate pieces of logic, you can create additional Java files.

Create a new file named `Helper.java` in the same folder as `Main.java`.

Java Code Sample

```
public class Helper {
    public static void showMessage() {
        System.out.println("Hello from another file!");
    }
}
```

Now call it from Main:

```
public class Main {
    public static void main(String[] args) {
        Helper.showMessage();
    }
}
```

Linking Java Files Together

Java files in the same folder automatically work together as long as:

- They are in the same package (or have no package statements)
- They are compiled together or exist in the same compiled output directory

When you compile:

```
javac Main.java Helper.java
```

Running `java Main` will execute your program and print the messages.

Conclusion

You now know how to organize Java files, separate logic into multiple files, and run Java code correctly from the command line or VS Code. This is the foundation of all Java application development.

Key Takeaways

- Java code must be written in `.java` files inside classes and run through the JVM.
- Use `System.out.println` to output results to the terminal.
- Separate logic by creating additional Java files and calling them from your main class.
- Compile multiple files together to build your application correctly.

values_and_variables_java

Values and Variables (Java 21)

Introduction to Values and Variables

In this section, we begin learning the Java language by understanding values and variables. These concepts form the foundation of every Java program. We will use examples such as a person's name, age, or job to illustrate how values and variables work.

Understanding Values

A value is a piece of data — the most fundamental unit of information in a program. For example, the text `"Jonas"` is a value, and the number `23` is also a value. Java can perform mathematical operations on numeric values to produce new values, such as `40 + 8 + 23 - 10`, which results in `61`.

Java Code Samples

```
System.out.println("Jonas");
```

```
System.out.println(23);
```

```
System.out.println(40 + 8 + 23 - 10);
```

Storing Values in Variables

One of the most useful things we can do with values is store them in variables so they can be reused throughout the program.

In Java, we declare a variable by specifying its type followed by its name:

```
String firstName = "Jonas";
```

This line creates a variable called `firstName` and stores the value `"Jonas"` inside it.

Another example:

```
String status = "amazing";
```

Variables act like labeled boxes — the label is the variable name, and the value is the content stored inside. To use the stored value, you reference the variable name:

```
System.out.println(firstName);
```

If you change the value of the variable later, all usages reflect the updated value:

```
firstName = "Matilda";
```

Variable Naming Conventions and Rules

Now that we understand what variables are, let's look at how to name them correctly. Java uses *camelCase* for variable names. This means the first word is lowercase, and each following word starts with an uppercase letter.

```
String firstNamePerson;
```

Other languages use styles like underscores, but in Java, camelCase is the standard.

Rules for Variable Names

Java variable names must follow these rules:

- They **cannot start with a number**:

```
// Invalid
int 3years = 3;
```

- They may contain **letters, numbers, underscores, or dollar signs**, but no other symbols:

```
String jonas_and_matilda = "JM";
```

- They **cannot use reserved Java keywords**, such as `class`, `new`, or `public`:

```
// Invalid
int new = 27;
int class = 1;
```

If necessary, an underscore or dollar sign may prefix the name:

```
int _class = 1;
int $value = 5;
```

The word `name` is allowed but not very descriptive:

```
String name = "Jonas";
```

More Naming Conventions

In Java, variable names should **not start with an uppercase letter** — by convention, uppercase names are reserved for **class names**.

```
String Person = "Jonas"; // Not recommended
```

Constants that never change should be written in **uppercase with underscores**:

```
final double PI = 3.1415;
```

Using meaningful, descriptive names makes your code easier to read:

```
String myFirstJob = "programmer";
String myCurrentJob = "teacher";
```

Avoid vague names like:

```
String job1 = "programmer";
String job2 = "teacher";
```

Recap: What is a Variable?

A variable is a box that stores a value. You give the box a name (like `firstName` or `myFirstJob`) and put a value inside it. Later in the program, you can reference that variable as many times as you want.

```
System.out.println(myFirstJob);
```

If you change the value:

```
myFirstJob = "coder";
```

The output changes accordingly. Variables are one of the most fundamental building blocks of programming, so take the time to understand them well before moving on.

Key Takeaways

- Values are the smallest units of information in programming and can be stored in variables.
- Variables are named containers that hold values and allow easy reuse.
- Java uses camelCase naming conventions and has strict rules for valid identifiers.
- Descriptive variable names improve readability and make programs easier to maintain.