Course Overview and Java Basics

Table of contents

Outline 1
Programming in the large 2
Java overview 2
Data types in Java 4
Scope 5
Control flow: sequence, selection, and iteration 6
Java arithmetic 10
Summary 11
Reminders 11

See csse2002[1] for staff, assessment, and course logistics. This is the technical content of Lecture 1, the live lecture. There's also a **recorded** lecture this week, see 2026-02-26-java-basics-part-02-collections-and-strings[2].

Outline

- Programming in the large vs. programming in the small — why this course isn't just "a Java course"
- Java vs Python: compiled vs interpreted, statically vs dynamically typed
- Java's primitive and reference types
- Scope
- Sequence, selection, and iteration in Java
- Java arithmetic

[1] courses/csse2002/csse2002.pdf
[2] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf

Programming in the large

Code from an intro course (CSSE1001/ENGG1001) has likely been:

1. Small.
2. Written solely by you.
3. For a limited purpose.
4. To exist for a limited time.

…but large software projects are often **none of these**.

CSSE2002 teaches *individual discipline and programming practices such that graduates can design and write software suitable for integration with large software systems.* Professionally you will need to:

- **Document** your code — because someone who didn't write it (or doesn't remember it) may need to work on it.
- **Debug** your code — the more code you write, the more opportunity to make mistakes.
- **Test** your code — it is far better for you to find bugs than for someone else to. You want a reason for confidence in your work beyond "I hope it is okay".

  CSSE2002 teaches practices that support documenting, debugging, and testing of code to allow programming to scale to large software projects.

This is not just a Java course. Knowing everything (or even most things) about Java is *not the only goal.* Java is the vehicle. As for why Java: in the Stack Overflow 2025 developer survey, Java is used by 29.6% of professional developers.

Where the course sits in the software lifecycle

In a seven-phase software development lifecycle, CSSE2002 lives in two of the phases, Phase 4 (Development) and Phase 5 (Testing):

Java overview

Java is a general-purpose programming language that is:

- **Compiled**
- **Statically typed**
- **Object-oriented**
- **Memory safe**

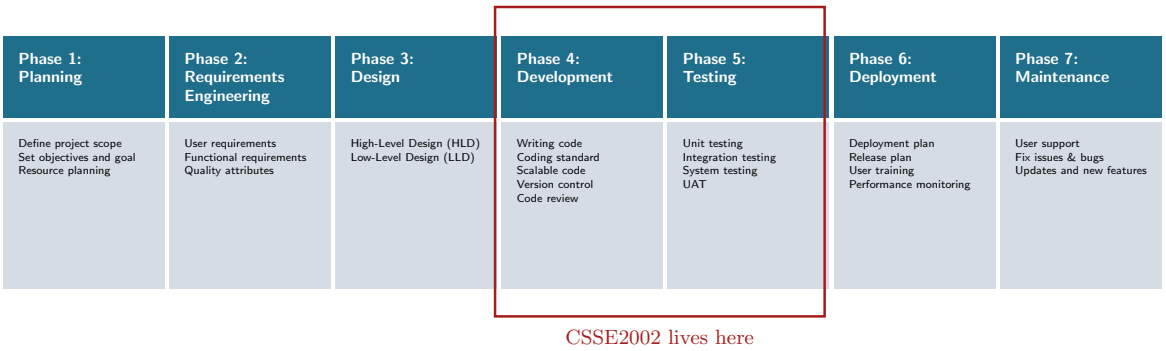

Figure 1: A seven-phase software development lifecycle (SDLC), with a red rectangle around Phases 4–5: CSSE2002 is a *development and testing* course, not a requirements or deployment course.

Java vs Python: compiled vs interpreted

Python is *interpreted*: you run the Python interpreter directly on the source.

```
$ python hello_world.py
```

Java is *compiled*: the source is first compiled to **bytecode**, then run.

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

Hello World, with the canonical entry point:

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello world!");
    }
}
```

Running `javac` on it produces a `.class` file of bytecode. As raw hex, the file begins with the JVM's `cafebabe` magic number:

```
cafebabe00000041001d0a000200030700040c000500060100106a617661
2f6c616e672f4f626a6563740100063c696e69743e010003282956090008
...
```

Homework. What is the benefit of having bytecode? What are the differences between bytecode and machine code?

Java vs Python: static vs dynamic typing

Python is *dynamically* typed — whether an appropriate type is used is determined when the program runs. Java is *statically* typed — types are checked for every scenario at compile time.

> Static typed languages require explicit definition of a data type when they create a piece of data (e.g. variable, parameter, return value). Dynamic languages are the opposite and can infer, or at least try to guess, the type that we're using.

```
x = 1
print(x, type(x))    # 1 <class 'int'>
x = 1.7
print(x, type(x))    # 1.7 <class 'float'>
x = "Hello"
print(x, type(x))    # Hello <class 'str'>
```

Python knows that `1` and `1.7` are different types of value (different "dynamic"/"runtime" types) but doesn't restrict variables on that basis. Java is the opposite — the *variable* carries the type:

```
int x;
x = 1;      // OK
x = -5000;  // OK
x = 1.0;    // Illegal: not a whole number
x = "15";   // Illegal: cannot convert from a string to a number
```

Python also uses indentation to delimit code blocks, while the Java compiler uses `{}` to define code blocks and `;` to terminate statements.

Data types in Java

See java-primitive-and-reference-types[3] for the full primitive-vs-reference-types table. In short: Java distinguishes **primitive types** (built-in, fixed representations like `int` and `boolean`) from **reference types** — everything else, i.e. classes (including `String` and arrays).

[3] courses/csse2002/java-primitive-and-reference-types.pdf

Scope

A variable's scope is where it can be used. In Java, a variable's scope ends at the end of the block in which it's declared. Python has no such rule, so the same code behaves differently in the two languages:

```python
# z does not exist
if 5 > 4:
    z = 2
else:
    z = 3
print(z)          # This works!
```

```java
if (5 > 4) {
    int z = 2;
} else {
    int z = 3;
}
System.out.println(z);   // This does not!
```

Question. What happens when this executes?

```java
public static void main(String[] args) {
    if (5 > 4) {
        int z = 2;
    } else {
        int z = 3;
    }
    System.out.println(z);
}
```

It never executes — it's a *compile* error: `z` is declared inside the `if`/`else` blocks, so it's out of scope by the time `System.out.println(z)` is reached.

Answer. Declare the variable inside the same (or a higher) block as the print — hoist the declaration out and leave only the assignments in the branches:

```java
public static void main(String[] args) {
    int z;
    if (5 > 4) {
        z = 2;
    } else {
        z = 3;
    }
    System.out.println(z);
}
```

Control flow: sequence, selection, and iteration

Sequence

1. Each line must end with a semicolon (`;`).
2. Declarations must come before other assignments.
3. Instructions are executed sequentially.

```java
public static void main(String[] args) {
    int x = 1;
    int y = 2;
    int z = x + y;
    System.out.println(z);
}
```

Methods are Java's equivalent of Python functions — but all code must live inside a class:

```python
def add_two_numbers(num1, num2):
    result = num1 + num2
    return result
```

```java
class Arithmetic {           // All code must be inside a class.
    int add_two_numbers(int num1, int num2) {
        int result = num1 + num2;
        return result;
    }
}
```

Calling a method looks the same as calling a Python function: `add_two_numbers(10, 20)`.

The Java method here keeps the Python name unchanged. Java's own convention, used everywhere else in the course (see csse2002-diagram-notation[4] §16), would be `addTwoNumbers`.

Selection: `if` and `switch`

`if` and `switch` are the same shape of control flow, so one diagram covers both:

The `if` chain:

[4] courses/csse2002/csse2002-diagram-notation.pdf

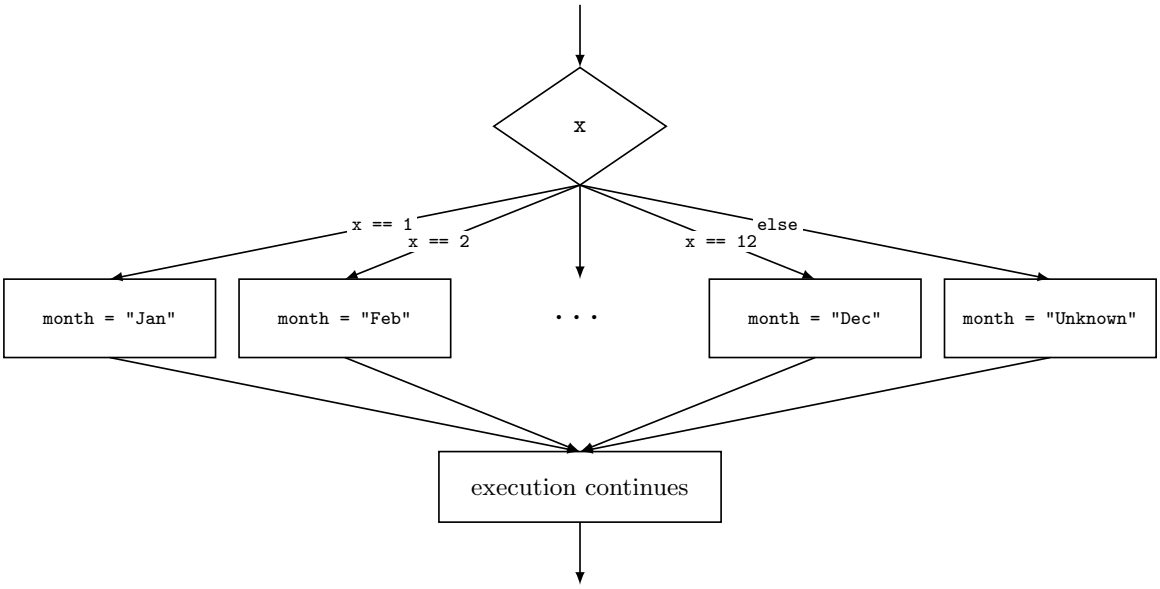

Figure 2: Selection as a flowchart: one test fans out into mutually exclusive branches that all rejoin at a single point. The same diagram describes the `if`-chain *and* the `switch`. The selector `x`, the guards and the assignments are those of the month code below, with one branch box per named outcome.

```
String month;
if (x == 1) {
    month = "Jan";
} else if (x == 2) {
    month = "Feb";
} ... {
    month = "Dec";
} else {
    month = "Unknown";
}
```

The same thing as a `switch`. This is the **arrow form** (`case … ->`), not the old colon-and-`break` form. An arrow case takes a block and does *not* fall through, so no `break` is needed:

```
String month;
switch (x) {
    case 1 -> {
        month = "Jan";
    }
    case 2 -> {
        month = "Feb";
    }
    ...
    default -> {
        month = "Unknown";
    }
}
```

Exercise. Write a method `printGrade` that takes the `mark` of a student and determines the grade:

Mark	Grade Description
83.00 – 100.00	High Distinction
73.00 – 82.99	Distinction
63.00 – 72.99	Credit
50.00 – 62.99	Pass
0.00 – 49.99	Fail

Iteration: `while` and `for` loops

Every counting loop has three moving parts:

- **Initialiser** — initialise a counter variable.

- **Test/condition** — check whether the counter has reached a condition.
- **Update** — increment or decrement the counter.

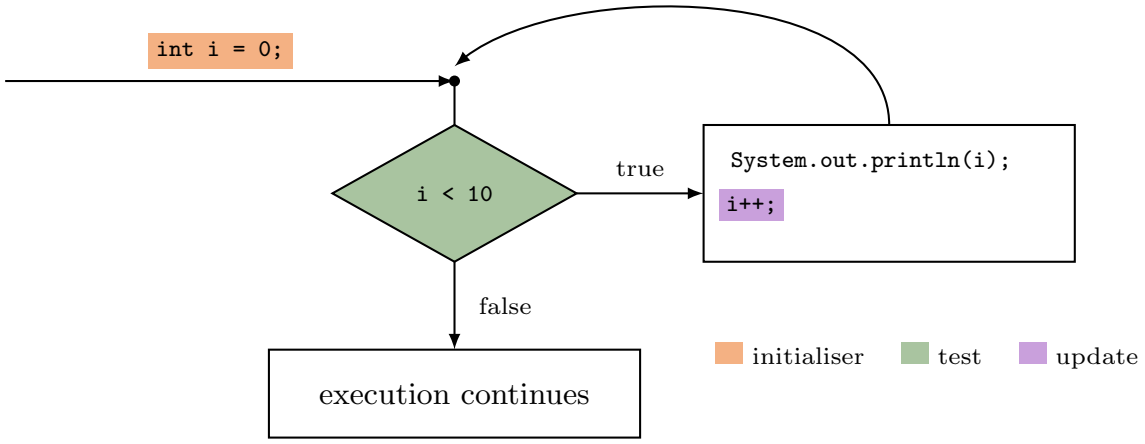

Figure 3: The `while` loop's control flow: the test is evaluated *before* each pass, and the update at the end of the body feeds the back-edge. The initialiser is orange, the test green and the update purple.

In a `while` loop the three parts are scattered across three lines:

```
int i = 0;              // initialiser
while (i < 10) {        // test
    System.out.println(i);
    i++;                // update
}
```

```
i = 0
while i < 10:
    print(i)
    i += 1
```

A `for` loop bundles all three into the loop header — the same three fragments, in the same order, separated by semicolons:

```
for (int i = 0; i < 10; i++) {
    System.out.println(i);
}
```

```
for i in range(10):
    print(i)
```

Exercise. A factorial of a number n, written $n!$, is the product of all numbers less than or equal to n:

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 = \prod_{k=1}^{n} k$$

with $0! = 1$, $1! = 1$, $2! = 2 \times 1 = 2$, $3! = 3 \times 2 \times 1 = 6$. Write a method `factorial` that calculates the factorial of its input.

Exercise. For all positive integers it holds that $n! = n \times (n-1)!$. Write an equivalent `factorialRec` method that uses recursion to calculate the result.

Java arithmetic

`+`, `-` and `*` work the same way as Python, except in a very special case.

Division is where the two languages diverge. Java has no `//` operator; instead, `/` *is* integer division whenever **both** operands are integers, and floating-point division as soon as either operand is a floating-point value:

Python	Result	Java	Result
`3 / 2`	`1.5`	`3 / 2`	`1`
`3 // 2`	`1`	*(no // operator)*	—
`3 / 2.0`	`1.5`	`3 / 2.0`	`1.5`
`3.0 / 2`	`1.5`	`3.0 / 2`	`1.5`
`3.0 / 2.0`	`1.5`	`3.0 / 2.0`	`1.5`

Comparison operators are identical in spelling to Python's — `x < y`, `x <= y`, `x > y`, `x >= y`, `x == y`, `x != y`. One caveat applies to `==` alone: this holds **for primitive types**. On reference types, `==` compares object identity, not contents. The equality section of 2026-02-26-java-basics-part-02-collections-and-strings[5] covers this.

Logical operators have the same semantics but different syntax:

Python	Java
`x and y`	`x && y`

[5] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf

Python	Java
`x or y`	x || y
`not x`	`!x`

Augmented assignment and increment/decrement:

Python	Java
`x += y`	`x += y`
`x -= y`	`x -= y`
`x += 1`	`x++`
`x -= 1`	`x--`

Summary

- The entry point for all Java programs is the `main` method: `public static void main(String[] args)`.
- Java is a compiled and statically typed language, compared to Python.
- **Most** arithmetic operations in Java function identically to Python.
- The same core principles of sequence, selection, and iteration apply to Java, but the syntax is different.

Reminders

- Your first Ed Lessons exercise is due by 1pm next Wednesday.
- Applied and practical classes start next week (Week 2).

Tasks this week:

- ☒ Attend the lecture.
- ☐ Week 1 Ed Lessons exercises.
- ☐ Ensure you can access Ed discussion.
- ☐ Enrol in your contact and practical classes.
- ☐ Watch the recorded lecture on Collections — see 2026-02-26-java-basics-part-02-collections-and-strings[6].
- ☐ Install IntelliJ IDEA & Java Development Kit (JDK) 21 — in IntelliJ's *Download JDK* dialog, choose **Version 21**, vendor **Eclipse Temurin (AdoptOpenJDK HotSpot)**.

[6] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf