CSSE2002 — Week 1 Notes

Table of contents

Course Overview and Java Basics

See csse2002[1] for staff, assessment, and course logistics — this note covers the technical content from Lecture 1 (the live lecture; there's also a **recorded** lecture this week, see 2026-02-26-java-

[1] courses/csse2002/csse2002.pdf

basics-part-02-collections-and-strings[2]).

Today's 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

Programming in the large

Code from an intro course (CSSE1001/ENGG1001) has likely been small, written solely by you, for a limited purpose, and to exist for a limited time — but large software projects are usually none of these. CSSE2002 teaches the individual discipline and programming practices needed to write software suitable for integration with large software systems: documenting, debugging, and testing code so that programming scales.

This is not just a Java course — Java is the vehicle, not the destination.

Java overview

Java is a general-purpose programming language that is:

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

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 on the JVM.

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

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

Homework (from the slides). 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 an explicit type declaration for every piece of data (variable, parameter, return value); dynamic languages instead infer (or guess) the type in use.

```
>>> 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 doesn't restrict a variable to one type over its lifetime, even though it tracks each value's own (runtime) type. Java is the opposite:

```java
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 delimit blocks and `;` to terminate statements.

Data types in Java

See java-primitive-and-reference-types[3] for the full primitive-vs-reference-types table introduced in this lecture. 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.

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);
}
```

This is a compile error: `z` is declared inside the `if/else` blocks, so it's out of scope by the time `System.out.println(z)` runs.

Answer. Declare the variable inside the same (or a higher) block as where it's used — e.g. declare `z` before the `if/else` (outer scope), or move the `println` inside each branch.

Control flow: sequence, selection, and iteration

Sequence

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

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)`.

Selection: if / switch statements

Java `if`/`else` and `switch` work like their Python counterparts, just with Java's block/statement syntax (`{}`, `;`, and `case`/`break` for `switch` rather than Python's `match`).

Exercise. Write a method `printGrade` that takes a student's mark and determines their grade:

Mark	Grade
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

A `while` loop has an **initialiser** (set up a counter), a **test/condition** (check the counter), and an **update** (increment/decrement the counter):

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

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

A `for` loop bundles all three parts together:

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

```python
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. Write a method `factorial` that calculates it.

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

Java arithmetic

- `+`, `-`, `*` work the same way as Python — **except** integer division: `/` between two `int`s in Java truncates towards zero (integer division), rather than always returning a float like Python's `/`.
- Comparison and logical operators are semantically identical to Python, just different syntax: `==`, `!=`, `<`, `>`, `<=`, `>=` are the same spelling, but Java's logical operators are `&&`, `||`, `!` where Python uses `and`, `or`, `not`.
- Augmented assignment (`+=`, `-=`, `*=`, `/=`) and increment/decrement (`++`, `--`) work as in many C-like languages — increment/decrement only apply to primitive types.

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[4].
- ☐ Install IntelliJ IDEA & Java Development Kit (JDK) 21.

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

Java Basics Part 02 - Collections and Strings

See csse2002[5] for staff, assessment, and course logistics. This is the **recorded** lecture flagged as a "Tasks this week" item in 2026-02-26-course-overview-and-java-basics[6] — watch/read this alongside the live Lecture 1.

Today's outline

- Java's memory model: stack vs heap
- Arrays and their limitations
- Strings: indexing, substrings, equality, immutability
- The `Stack`, `List`, `Set`, and `Map` collections

Java's memory model

Runtime memory splits into:

- **The stack** — local variables and method parameters.
- **The heap** — values with a dynamic size (objects and shared data).

Worked example: stack frames

```
public class StackHeap {
    public static void main(String[] args) {
        double hourlyRate = 54.0;
        int hoursWorked = 16;

        double salary = calculateSal(hourlyRate, hoursWorked);

        printSalary(salary);
    }
    public static double calculateSal(double hourlyRate, int hoursWorked) {
        return hourlyRate * hoursWorked;
    }
    public static void printSalary(double sal) {
        System.out.println("Salary: " + sal);
    }
}
```

[5] courses/csse2002/csse2002.pdf
[6] courses/csse2002/2026-02-26-course-overview-and-java-basics.pdf

Tracing the stack: `main` pushes a frame holding `hourlyRate = 54.0` and `hoursWorked = 16`. Calling `calculateSal` pushes a new frame on top (with its own `hourlyRate`/`hoursWorked` parameters); it computes `864.0` and returns, popping its frame and storing the result in `main`'s `salary`. `main` then calls `printSalary`, which pushes a frame holding `sal = 864.0`, prints `Salary: 864.0`, and pops. Each method call gets its own frame, and frames are popped in the reverse order they were pushed (last in, first out).

Worked example: heap allocation

```
static float lastMark(int n) {
    float[] marks = new float[n];
    marks[0] = 75.5f;
    marks[n - 1] = 88.0f;
    return marks[n - 1];
}
```

Called as `lastMark(5)` from `main`: the array itself (`float[5]`, initially `[0.0, 0.0, 0.0, 0.0, 0.0]`) is allocated **on the heap**. The stack frame for `lastMark` only holds a *reference* (`marks`) pointing at that heap object, plus the parameter `n = 5`. Assigning `marks[0] = 75.5f` and `marks[n-1] = 88.0f` mutates the heap array directly through the reference, giving `[75.5, 0.0, 0.0, 0.0, 88.0]`.

Java never clears heap memory just because a method ends — it's only reclaimed once no references to it exist and the garbage collector decides to run.

Arrays

Recall: an array is an ordered, fixed-length, mutable sequence of homogeneous items.

```
int[] numbers = {1, 2, 3, 4, 5};
numbers.length;         // 5
numbers[0] = 0;         // OK
numbers[0] = "zero";    // illegal -- all elements must be the same type (int)
```

Two ways to create an array:

```
// Approach 1: array literal
float[] marks = {60.3, 62, 70.1, 65.8, 80.3};

// Approach 2: allocate then assign each index
float[] marks = new float[5];
```

```
marks[0] = 60.3;
marks[1] = 62;
marks[2] = 70.1;
marks[3] = 65.8;
marks[4] = 80.3;
```

Querying: `marks[3]` and `marks[0]` are valid, but `marks[marks.length]` throws `ArrayIndexOutOfBoundsException` — valid indices are `0` to `length - 1`.

Iterating: an indexed `for` loop, or an enhanced `for`-each loop:

```
for (int i = 0; i < marks.length; i++) {
    System.out.println(marks[i]);
}
for (float mark : marks) {
    System.out.println(mark);
}
```

Try at home.

- `max(int[])` — returns the maximum value in an array of integers.
- `contains(int[], int)` — returns `true` if the array contains the given integer.
- `reverse(int[])` — returns a new array with the elements in reverse order.

Limitations of arrays

Arrays have a **fixed size at creation** (you must know the space you need up-front), and don't automatically close gaps when an element is removed from the middle. This motivates the built-in collections below.

Strings

A `String` is a sequence of characters:

```
String course = "Programming in the Large";
System.out.println(course.length());     // 24
System.out.println(course.charAt(0));     // 'P'
System.out.println(course.charAt(11));
System.out.println(course.charAt(23));    // 'e'
```

`substring(start, end)` returns a substring with an **inclusive** start index and an **exclusive** end index (if `end` is omitted, it defaults to the end of the string):

```
System.out.println(course.substring(0, 11));  // "Programming"
```

Equality and immutability

Equality means something different for primitive types vs reference types:

	Primitive types	Reference types
`x = y`	make `x` store a copy of `y`'s value	make `x` refer to the same object `y` refers to
`x == y`	check if `x` stores the same value as `y`	check if `x` refers to the same object as `y`
`x != y`	check if `x`'s value differs from `y`'s	check if `x` and `y` refer to different objects

```
String name  = "Jack";
String name2 = "Jack";
String name3 = new String("Jack");

System.out.println(name == name2);        // true  -- same pooled literal
System.out.println(name == name3);        // false -- different object

System.out.println(name.equals(name2));   // true
System.out.println(name.equals(name3));   // true  -- .equals() compares content

Object obj1 = new Object();
Object obj2 = new Object();
System.out.println(obj1.equals(obj2));    // false -- Object's default .equals() is
↪  identity
```

String objects are immutable. Reassigning `name = "Jill"` doesn't mutate the original `"Jack"` object — it just repoints the `name` reference to a different (or newly pooled) string. String literals with the same value are shared via the **string pool** (`name` and `name2` above both point at the same pooled `"Jack"`); `new String("Jack")` opts out of the pool and allocates a distinct object.

The Collections framework

See java-collections-framework[7] for the full `Stack`/`List`/`Set`/`Map` interface reference — the rest of this section walks through the lecture's worked traces. All of these live in `java.util.*`.

[7] courses/csse2002/java-collections-framework.pdf

Stack

LIFO (Last In, First Out): `empty()`, `peek()`, `pop()`, `push(obj)`.

```
letters.empty();       // true
letters.push("A");
letters.empty();       // false
letters.push("B");
letters.push("C");
letters.peek();        // "C"
letters.push("D");

letters.pop();         // "D"
letters.pop();         // "C"
letters.pop();         // "B"
letters.pop();         // "A"
letters.pop();         // EmptyStackException
```

Creating a stack (must import `java.util.Stack`):

```
Stack<Integer> stacks = new Stack<>();
Stack<String> stacks = new Stack<>();
Stack<Cat> stacks = new Stack<>();
```

Collections only store objects, so `Stack<int>` is illegal — Java provides a wrapper class for each primitive type (`Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`; see java-primitive-and-reference-types[8]).

Exercise. Implement `int sum(Stack)` that returns the sum of all integers in the given stack.

List

Lists hold items in sequential order like an array, but grow/shrink automatically, have no fixed size limit, support inserting/removing an item anywhere, and are indexed from zero.

`List` is an **interface**, not a particular implementation — you can declare a variable as `List`, but can't do `new List()`. Popular implementations: `ArrayList` (better for random access) and `LinkedList` (better for modifying the middle of the list).

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

```
List<String> courses = new ArrayList<>();
courses.add("CSSE1001");          // [CSSE1001]
courses.add("DECO3801");          // [CSSE1001, DECO3801]
courses.get(0);                   // "CSSE1001"
courses.add(1, "CSSE2310");       // [CSSE1001, CSSE2310, DECO3801]
courses.add(1, "CSSE2002");       // [CSSE1001, CSSE2002, CSSE2310, DECO3801]
courses.remove(2);                // removes/returns "CSSE2310" -> [CSSE1001,
↪  CSSE2002, DECO3801]
```

Set

Sets store unique items (no duplicates); don't assume any iteration order.

```
Set<String> farm = new HashSet<>();
farm.add("Fox");       // true
farm.add("Farmer");    // true
farm.add("Chicken");   // true
farm.add("Fox");       // false -- already present
farm.add("Grain");     // true
farm.size();           // 4
```

Implementations: `TreeSet<E>` (`E` must implement `Comparable`, e.g. `String`) and `HashSet<E>` (`E` must have sensible `hashCode()`/`equals()`).

Map

Maps store key → value pairs, like a Python dict.

```
Map<String, Integer> farm = new HashMap<>();
farm.put("Fox", 5);
farm.put("Farmer", 10);
farm.put("Chicken", 0);
farm.get("Farmer");                        // 10
farm.put("Fox", 18);                       // overwrites Fox's value
farm.put("Grain", 15);
farm.put("Grain", farm.get("Grain") + 5);  // Grain -> 20
// final map: {Fox: 18, Farmer: 10, Chicken: 0, Grain: 20}
```

Implementations: `TreeMap<K,V>` (`K` must implement `Comparable`) and `HashMap<K,V>` (`K` must have sensible `hashCode()`/`equals()`).

The hashCode/equals contract

For `HashSet`/`HashMap` to behave correctly, elements/keys need:

1. `x.equals(y)` $\iff$ `y.equals(x)` (symmetric).
2. `x.equals(y)` $\implies$ `x.hashCode() == y.hashCode()`.

`hashCode()` returns an integer generated by a hashing algorithm for the object, e.g. `"Thilina".hashCode()` $\rightarrow$ `318621125`.

Reference material

Java Collections Framework

Introduced in 2026-02-26-java-basics-part-02-collections-and-strings[9] (Lecture 1, Week 1) — see that lecture for the worked push/pop/add/remove traces. All of these live in `java.util.*`, and (unlike arrays) grow/shrink automatically.

Why collections instead of arrays?

Arrays have a fixed size at creation and don't automatically close gaps when an element is removed from the middle. Collections solve both problems, at the cost of only being able to store objects (reference types) — see java-primitive-and-reference-types[10] for the primitive wrapper classes (`Integer`, `Double`, etc.) used to store primitives inside them.

Stack — LIFO

Method	Description
`empty()`	Is this stack empty?
`peek()`	Return the object at the top of the stack
`pop()`	Remove (and return) the object at the top of the stack
`push(obj)`	Put `obj` on the top of the stack

```
Stack<Type> stacks = new Stack<>();
```

`pop()` on an empty stack throws `EmptyStackException`.

[9] courses/csse2002/2026-02-26-java-basics-part-02-collections-and-strings.pdf
[10] courses/csse2002/java-primitive-and-reference-types.pdf

List

An interface, not a particular implementation — you can declare a variable as `List`, but can't do `new List()`.

- **`ArrayList`** — better for random access (`get(i)`).
- **`LinkedList`** — better for operations that modify the middle of the list.

Holds items in sequential order (like an array), 0-indexed, with no fixed size limit; supports inserting/removing at any position.

Set

Stores unique items (no duplicates); don't assume any iteration order.

```
interface Set<E> {
    int size();
    boolean contains(E item);
    boolean add(E e);
    boolean remove(E item);
}
```

- **`TreeSet<E>`** — `E` must implement `Comparable` (e.g. `String`).
- **`HashSet<E>`** — `E` must have sensible `hashCode()`/`equals()`.

Map

Stores key → value pairs (like a Python dict) — specify a type for both the key and the value, e.g. `Map<Integer, String>`.

```
interface Map<K, V> {
    int size();
    boolean containsKey(K key);
    boolean containsValue(V value);
    V get(K key);
    V put(K key, V value);
    V remove(K key);
    Set<K> keySet();
}
```

- **`TreeMap<K,V>`** — `K` must implement `Comparable`.
- **`HashMap<K,V>`** — `K` must have sensible `hashCode()`/`equals()`.

The hashCode/equals contract

`HashSet`/`HashMap` rely on their elements/keys satisfying:

1. `x.equals(y)` $\iff$ `y.equals(x)` (symmetric).
2. `x.equals(y)` $\implies$ `x.hashCode() == y.hashCode()`.

`hashCode()` returns an integer generated by a hashing algorithm for the object — two objects that are `.equals()` must hash the same, or a `HashSet`/`HashMap` won't be able to find them correctly.

Java Primitive and Reference Types

Introduced in 2026-02-26-course-overview-and-java-basics[11] (Lecture 1, Week 1).

Two categories of types

Java distinguishes two sorts of types:

1. **Primitive types** — built-in, fixed representations.
2. **Reference types** — everything else (i.e. classes), including `String`, arrays, and any user-defined class.

Primitive types

Type	Size	Stores
`boolean`	not specified	`true` or `false`
`byte`	1 byte / 8 bits	whole numbers from -128 to 127
`short`	2 bytes / 16 bits	whole numbers from -32,768 to 32,767
`int`	4 bytes / 32 bits	whole numbers from -2,147,483,648 to 2,147,483,647
`long`	8 bytes / 64 bits	whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`float`	4 bytes / 32 bits	fractional numbers, accurate to 6-7 decimal places

[11] courses/csse2002/2026-02-26-course-overview-and-java-basics.pdf

Type	Size	Stores
`double`	8 bytes / 64 bits	fractional numbers, accurate to 15 decimal places
`char`	2 bytes / 16 bits	a single character/letter

Reference types

Reference types are everything that isn't primitive — classes, including `String` and arrays. A variable of a reference type stores a reference to an object, not the object's data directly (see 2026-02-26-java-basics-part-02-collections-and-strings[12] for the stack/heap consequences of this, and how it changes the meaning of `=`, `==`, and `!=`).

Collections (`Stack`, `List`, `Set`, `Map`) can only store objects/reference types — see java-collections-framework[13] for the wrapper classes (`Boolean`, `Byte`, `Character`, `Double`, `Float`, `Integer`, `Long`, `Short`) Java provides so primitives can be stored in them.

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