Lec 2: Java Basics

Sam McCauley

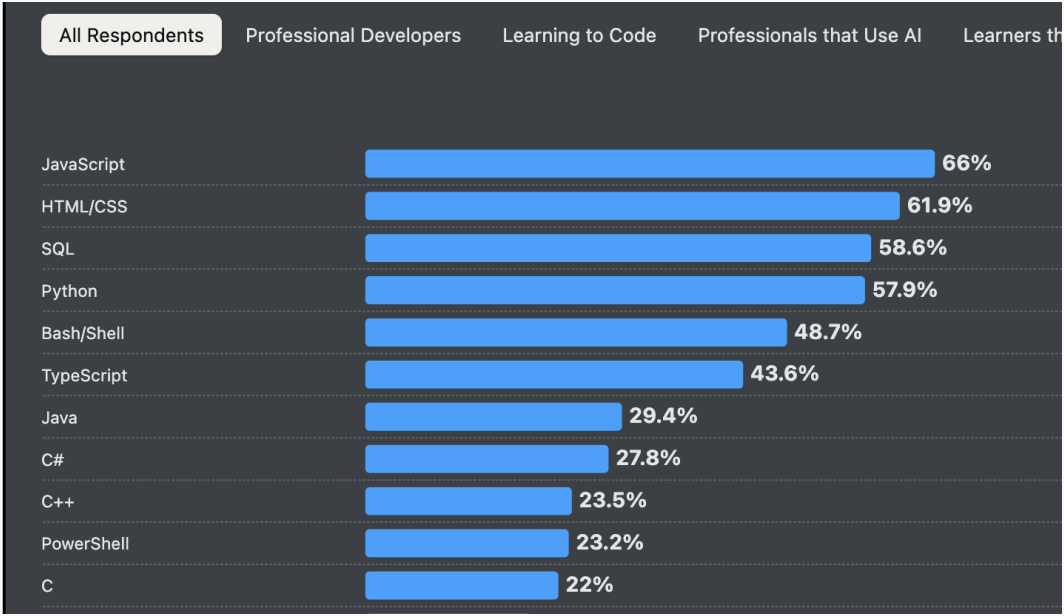
February 9, 2026

Admin

- Welcome back!
- Lab 0 due tomorrow (get started early!)
  - Office Hours/TA Hours today and tomorrow (see calendar on website)
- Lab 1 in person on Wednesday at 1pm
  - Covers basic Java: loops, arrays, functions
- Any questions?

Why Java?

Java is Still Popular!



Why Java?

- Java is (still) popular!
- Object-oriented—good for large systems
- Good support for abstraction, extension, modularization
- Automatically handles low-level memory management
- Very portable

How to Think about Java

- Containers lead to scalability, flexibility, portability
- But they come with overhead!
- Java is the same way
- This class is all about *scalang* your CS knowledge

Java Experience

- It’s OK (in fact completely expected) if you’re new to Java, or very rusty
- (I am also quite rusty)
- Let me know if you see any mistakes!
- Goal for this class: figure it out together

Java Intro

Let’s take a look at Java programming

- Next couple lectures: (re)introduction to Java
- Some of you have Java experience; some don’t
- Goal: quickly get everyone up to speed for the first couple labs
- We will go fast! So ask questions.
- If you are less comfortable with Java, be sure to keep up with readings and ask questions early on!

Elements of a Java Program

```
1 public class Hello {
2       public static void main(String[] args) {
3
4
5
6
7       }
8 }
```

- First line: the “class” the program lives in (basically the box we’re working in). Must match the name of the file!
- Second line: the “main” function where execution starts

Hello World

```java
/*
 * Hello.java
 * Author: Sam
 * Prints a welcome message to the terminal
 */
public class Hello {

    public static void main(String[] args) {
        System.out.println("Hello, CS136!") ;
    }
}
```

Let’s test out the hello world program

- Open it in VSCode
  - You want to always use “Open *Folder*”
- To compile:

```
> javac Hello.java
```

- To run:

```
> java Hello
```

Compiling and Running Code

- You need to compile the code every time you change it.
- Long story short: the computer does a pass over the code, translating it into a form that's more useful for it internally
    - Makes a `.class` file, which is what it's actually running later
- In pairs: What are some disadvantages of compiling code before it is run? What do you think some advantages are?

Playing around with Hello World

- How can we change what is printed?

- What happens if we make a mistake?

Semicolons

- Every command needs a semicolon at the end
- (Don’t need at the end of loops, function declarations, classes, etc.)
- It’s annoying. The compiler will usually tell you if you missed one

Java Comments

- Comments are ignored by the compiler (they do not affect what your code does)
- Multi-line comments: `/*` and `*/`
- Single-line comments: `//`
  - Anything after `//` on a line is ignored by Java
- You should use comments to explain any code that is not self-explanatory
  - `x = x - 1` is self explanatory
  - `if(n & (n - 1)) == 0)` needs a comment
- We'll discuss this more in the next couple weeks

Java Variables

Variables in Java

- Variables *must* be declared before they are used
- Basically: tell Java that you are going to use a variable; tell Java what *type* it is
  - `int age; // An integer value`
  - `double speed; // A number that may have a ‘decimal’ part`
  - `char grade; // A single character`
  - `boolean loggedIn; // Either true or false`
- Must also initialize (assign a value) before it is accessed.

Simple Operations

- Assigning variable values, addition, subtraction, multiplication, division work probably the way you expect
- Let’s do a quick example: write a program to convert feet to miles
- Use % for modulo (remainder)
- Note that ^ is not exponent! (Nor is **). We’ll come back to exponents later

Primitive Types Cont.

- Variables *can* be *initialized* at the same time as they are declared. (Initialize just means assign a value for the first time)

  - `int age = 21;`
  - `float speed = 47.25;`
  - `char grade = ‘A’;`
  - `boolean loggedIn = true;`

Conditionals

if statements

- The code in the if statements is only executed if the if statement is true
- Looks like below
    - Don’t forget your parentheses and curly braces!

```
1 if(x < 0) {
2     System.out.println("x is negative.");
3 }
```

When is an if statement over?

```
1 if(x < 0) {
2     x = -x;
3     System.out.println("x is negative.");
4 }
5 //execution goes here if x < 0 is false
```

- The braces show Java what to execute when the if statement is true (and what to jump past if the statement is false)
- Indentation is *only visual help* in Java (does not affect what the code does)
- You should indent anyway!
    - Option-Shift-F or Alt-Shift-F automatically indents code if you installed the optional Java backage

What goes in the conditional

- Expression that evaluates to `true` or `false` (can be stored in a `boolean`)
- Note that to check equality, you need == (two equals signs)
- Can also use <, >, <=, >=, or `!=` (not equal to)

else

- Executed only if the if statement is *false*
    - Don’t forget your curly braces!

```
1 if(x == 0) {
2     System.out.println("x is zero.");
3 } else {
4     System.out.println("x is not zero.");
5 }
```

else if

- Each statement is only executed if previous ones were false, and this one is true

```
if(x < 0) {
    System.out.println("x is negative.");
} else if (x == 0) {
    System.out.println("x is zero.");
} else {
    System.out.println("x is positive.");
}
```

Loops

while Loops

```
1 int x = 10;
2 while(x < 1000) {
3     x = x * 2;
4     System.out.println(x);
5 }
```

- If the condition in a `while` loop is true, the code inside the loop is executed
- Then the condition is checked again; if it is still true, the code is executed again
- The code is not run at all if the condition is false to begin with
- Let's compile and run the above code

for loops

- Java `for` loops are substantially *different* to python's
- In Java, `for` loops are just shorthand for a common kind of while loop
- They are good for iterating over lists, but they can be used for other purposes

for loops

- In the parentheses, we'll have three parts separated by semicolons
    - The first part is executed once, *before* any code is run
    - The second part is the loop condition (just like a while loop: the code in the braces will execute so long as this condition is true)
    - The third part is executed after each loop iteration. It usually consists of incrementing a variable, but it doesn't have to.

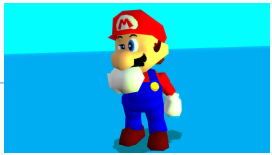
for loops

These two loops are *exactly* the same in Java:

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

```
int x = 0;
while(x < 10) {
    System.out.print(x);
    System.out.print(" ");
    x++;
}
System.out.println();
```

for loop practice



- How could we count multiples of 3 using a for loop?

- How could we count *backwards* using a for loop?

Methods

Methods in Java

- Often called “functions” in other languages
- Snippets of code we can call repeatedly
- May or may not have *parameters* (way for us to give information to the method)
- May or may not have a *return value* (way for the method to pass back information)
- These are just the basics of methods. We’ll learn much more when we cover objects and classes

Method Syntax

```java
public static int addThreeInts(int x, int y, int z) {
    int sum = x + y + z;
    return sum;
}

public static void main(String[] args) {
    int total = addThreeInts(1, 2, 4);
    System.out.println(total);
    System.out.println(addThreeInts(10, 20, 30));
}
```

- Use `public static` at beginning of method (for now)
- State the *type* of value being returned: `int` here
- Then name of the method: `addThreeInts`
- List the parameters in the parentheses. Don't forget their types!
- The `return` keyword tells Java to go back to where the function was called. The original function call is "replaced" with the return value

Method Syntax

Let's trace execution through the following code.

```
1  public static int addThreeInts(int x, int y, int z) {
2      int sum = x + y + z;
3      return sum;
4  }
5
6  public static void main(String[] args) {
7      int total = addThreeInts(1, 2, 4);
8      System.out.println(total);
9      System.out.println(addThreeInts(10, 20, 30));
10 }
```

Method Practice

- Let’s write some code to print all of the prime numbers from 1 to 100
- (Prime means: not evenly divisible by any positive integer other than 1 and itself)

Arrays (First data structure!)

What is an array?

- Sequence of items, kind of like a list
- Access any one of the items using brackets: `[` and `]`
- The first item is index `0`
- *Fixed* length!
    - If you want to “change” the length, you need to make a brand new array and copy everything over

An Array

Declaring and Using Arrays

- Like variables, specify the type up front: array of `int`, `float`, `char`, `String`, etc.
- Use brackets after the type to specify that it is an array
- Need to initialize it using the “new” keyword with its size (we’ll come back to this)

```
1 int[] arr = new int[3];
2 arr[0] = 4;
3 arr[1] = 3;
4 System.out.println(arr[0] + arr[1]);
```

Notes on Arrays

- Arrays are not primitive types in Java, they are class types (an array is therefore an *object* in Java)
- As a result, an uninitialized array (before the “new” part is assigned) holds the special object value `null`. This means
  - It is an error to attempt to index into an uninitialized array (no new)

```
1 int[] scores; // Uninitialized array
2 scores[0] = 100; // Error!
```

  - It is an error to access any instance variable or method of an uninitialized array
    - int size = scores.length; // Error!