Lec 5: Wrapping Up Java

Sam McCauley

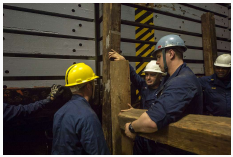
February 18, 2026

Admin

- Lab 2 today
  - Start early; look over before lab. Let me know if you see typos. (O
    code called out in the text.)
  - If you're already finished, please come to lab to look it over with an instructor
- Friday: first quiz (practice quiz is out; available at TA hours)
- Today: last day about Java! Lots of small topics; be sure to reference the handout
- I'm going to try to solidify my names knowledge

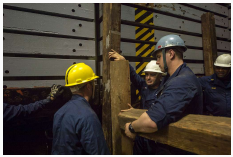
Classes and Objects

Constructors



- Special methods that call every time an object is instantiated
- (Rough) equivalent of `__init__` in Python
- Do work to “set up” the class
- Always have the same name as the class; no return value (not `void`—no return value at all)
- May have parameters; can have different constructors with different parameter types

Revisiting Object Instantiation



```java
public Student(String inName) {
    name = inName;
}
```

```java
Student s1 = new Student("Sam")
```

When an object is instantiated, we call its constructor. That is why the notation for instantiation uses parentheses.

What goes in a constructor

- Anything required to set up the object for the first time

- If computation is required, should be split off into a *helper method*

Access Modifiers

Access Modifiers for Methods

- We can also write `public` or `private` for methods
- `private` is for methods that others shouldn't use
  - Helper methods that no one else has a reason to access
  - Methods that do "dangerous" things (Example: moving a value in what is supposed to be a sorted array)
  - Helper methods are usually private; signal that they are just internal helper methods
- No hard rules: the programmer needs to decide if a method is `public` or `private`
- Let's look briefly at `OptimalAI.java`

Access Modifiers

Java has the following access modifiers. For now, we'll only use the first two.

- `public` : the method or variable is accessible from anywhere
- `private` : the method or variable can only be accessed from within the class itself
- `protected` : the method or variable can only be accessed from within the class, or from classes that *inherit* from it [We'll see inheritance right after spring break]
- *default* [meaning no access modifier specified] : the method or variable can only be accessed from a class within the same "package" [We won't cover packages in this course]

Access Modifiers

Java has the following access modifiers:

- `publ`
- `priv` hin the class itself
- `prot` within the class, ght after spring
- *defau* ariable can only b n't cover packa

Takeaway:

All instance variables should be `private` or `protected`.

Methods should be `private`, `protected`, or `public`.

You should only use `private` or `public` for now.

Let’s work on a larger piece of code together.

Anagram Solver

Let's do the following in `Anagrams.java`:

- Prompt user for a `String`
- Find all anagrams of the input `String` that are an English word
- Two classes to help us: `WordList.java` and `Permutations.java`
- Let's look at the `public` methods of those classes and see if they can help us fill in our code
- Note: The code here is not efficient, and in some places the style choices are questionable: our goal is getting the task done with the tools we have

Wrapping Up Types and Control Flow

Short-Circuit Evaluation

- If the first part of an && expression is false, the expression is always false.
  - Java will skip it!
- If the first part of an || expression is true, the expression is always true.
  - Java will skip it!

Classic example is for checking for validity before doing the “real operation”:

```
1  if(x != 0 && total/x > 4) //even if x = 0, no error due to
       short-circuit evaluation
```

Types and Operations

Mixing Types

```
1       int x = 4;
2       double y = 2.1;
3       System.out.println(x + y);
```

- When an expression has multiple types, Java needs to choose what type to use
- Chooses the “most general” type (see handout for link to full rules):

$$\texttt{String} > \texttt{double} > \texttt{int} > \texttt{char}$$

- In the above code, `x + y` causes `x` to be converted to a double; 6.1 is printed.

Mixing Types

- Java will give an error if you are going to *lose* information
- Must cast the value explicitly (we'll discuss in a minute)

```
1  double x = 4.2;
2  int y = x - 1; //the .2 is dropped; the compiler gives an
       error
```

```
1  double x = 4.2;
2  int y = (int)x - 1; //OK!  y is 3
```

Casting

- Allows us to change the type of data
- Note: cannot change the type of a variable itself
- Notation: put the new variable type in parentheses

```
1 double x = 4.2;
2 int y = (int)x - 1;
3 System.out.println(y);
4 System.out.println(x);
```

Bear in mind: constants have types!

```
1  int feet = 10;
2  double yards = feet/3;
3  System.out.println(yards);  //prints 3.0
```

One fix: use a double constant (with a decimal point)

```
1  int feet = 10;
2  double yards = feet/3.0;
3  System.out.println(yards);  //prints 3.33333...
```

Better(?) fix: use casting

```
1  int feet = 10;
2  double yards = (double)feet/3;
3  System.out.println(yards);  //prints 3.33333...
4  System.out.println(feet);  //prints 10; feet is still an int
```

Getting Input From the User

Getting User Input

Several steps in Java:

- At the *top* of the file (before class declaration), write:
  `import java.util.Scanner;`
- Instantiate the reader object: before asking for input, write:
  `Scanner reader = new Scanner(System.in);`
- Get the next integer the user inputs using the method:
  `reader.nextInt();`
- Can also get a `String` using `reader.nextLine();` or a `double` using `reader.nextDouble()`
- Let's write a simple program that averages 10 user-input integers

Wrapping Up Control Flow

break and continue

- `break`: exit current loop
- `continue`: go to next iteration of current loop
- Use *very sparingly*. But if it makes the code clearer or simpler, it's OK.
- In short: if you are going to use a `break` or `continue` statement, think for a moment if you can refactor the code to avoid it. If you can, do so; if you can't (or if it would obfuscate the code), then don't

Example of Good Code Using `break`

```
for(int x = 0; x < arr.length; x++) {
    System.out.print(arr[x] + " ");
    if(arr[x] == -1) {
        break;
    }
}
```

How could we refactor this?

Attempt to Refactor Without `break`

```
1 boolean foundANeg1 = false;
2 for(int x = 0; x < arr.length && !foundANeg1; x++) {
3     System.out.print(arr[x] + " ");
4     if(arr[x] == -1) {
5         foundANeg1 = true;
6     }
7 }
```

We used a flag variable instead of a `break`. The code is less clear! So a `break` is good to use here.

do-while loop

- Like a `while` loop, but condition is checked at the *end* of the loop
- So the loop always runs at least once
- Note the semicolon! (Only loop with a semicolon i Java)

```
do {
    //this code is executed once,
    //and then continues to be executed while the
    //condition is true
}while (/*condition*/);
```

switch statements

- Useful for directing flow based on which of several cases a variable is *equal* to
- Let's say we want to translate a playing card `suit` stored as an integer 1–4 into a `String` representing the suit
- Let's first look at how to do it using an `if/else if/else` statement; then a `switch` statement

```java
public static void printSuit(int suit) {
    if(suit == 1) {
            System.out.println("Spades");
    } else if (suit == 2) {
            System.out.println("Clubs");
    } else if (suit == 3) {
            System.out.println("Diamonds");
    } else if (suit == 4) {
            System.out.println("Hearts");
    } else {
            System.out.println("Invalid suit!");
    }
}
```

```java
public static void printSuit(int suit) {
    switch(suit) {
        case 1:
            System.out.println("Spades");
            break;
        case 2:
            System.out.println("Clubs");
            break;
        case 3:
            System.out.println("Diamonds");
            break;
        case 4:
            System.out.println("Hearts");
            break;
        default:
            System.out.println("Invalid suit!");
    }
}
```

Don't forget the `break` statements! If you don't include them, execution will continue through the remaining cases in the `switch` statement.

Scope

What is Scope?

- The “lifetime” of a variable
- Inside the variable’s scope, you can access it
- Outside of the scope you cannot
- Outside of the scope, can create a new variable with the same name

```
1  int x = 0;
2  int x = 10; //Error!
3  double x = 10.0; //Error!
```

Scope for Methods

A variable declared in a method is completely local to the method: the lifetime of the variable cannot extend beyond the method.

```
1  public void doSomethingInteresting() {
2      int x = 0;
3      if(x < 5) {
4          System.out.println("x is " + x);  //OK!
5      }
6      //this is the last line where x is available
7  }
8  public void doSomethingElse() {
9      doSomethingInteresting();  //let's call the other method
10     System.out.println("x is " + x); //Error! Lifetime of x
           ended
11 }
```

Scope for If Statements

The scope of a variable declared in a loop or conditional is limited to the loop or conditional. This can be counterintuitive!

```
1  public void printSize(int y) {
2      if(y < 100) {
3          String response = "small y";
4          //reached the end of the if
5      } else {
6          String response = "large y";
7          //reached the end of the else
8      }
9      System.out.println(response); //Error!
10 }
```

Question: how can we fix this?

Scope for If Statements

Fix:

```
1  public printSize(int y) {
2      String response;
3      if(y < 100) {
4          response = "small y";
5      } else {
6          response = "large y";
7      }
8      System.out.println(response); //works!
9  }
```

Scope for Loops

The scope of a variable declared in a loop or conditional is limited to the loop or conditional.

```
1  for(int x = 0; x < 10; x++) {
2      System.out.println(x);
3  }
4  for(int x = 0; x > -10; x--) { //OK!
5      System.out.println(x);
6  }
```

In the above code, we declared `int x` twice. This is OK, because the lifetime of `x` is limited to the `for` loop.

Scope for Loops

Sometimes we would like to access a variable after the loop finishes! How can we fix this?

```
for(int x = 0; x < arr.length; x++) {
    System.out.print(arr[x] + " ");
    if(arr[x] == -1) {
        break;
    }
}
System.out.println("-1 was found at " + x); //Error!
```

```
int x = 0;
for( ; x < arr.length; x++) {
    System.out.print(arr[x] + " ");
    if(arr[x] == -1) {
        break;
    }
}
System.out.println("-1 was found at " + x); //OK!
```

Rule for Scope in Java

- The scope of a variable is limited to the *surrounding curly braces*
- (Covers methods, conditions, loops...)
- You are allowed to add extra curly braces to your code to limit variable scope if you want.

Scope, Names, and Instance Variables

- Remember that methods can access instance variables of a class directly

- Nonetheless, you may (but usually *shouldn't*) create a variable in the method with the same name.

```java
public class Student{
  private int graduationYear;

  public void exampleParameter(int graduationYear) {
    System.out.println(graduationYear);  //parameter
  }

  public void exampleLocal() {
    graduationYear = -1;
    System.out.println(graduationYear);  //local variable
  }

  public Student(int newGraduationYear) {
    graduationYear = newGraduationYear;
  }

  public static void main(String[] args) {
    Student s1 = new Student(2026);

    s1.exampleParameter(0); //prints 0
    s1.exampleLocal(); //prints -1
  }
}
```

Disambiguating Instance Variables

- If you *do* have a parameter or local variable with the same name as an instance variable, can access the instance variable using the `this` keyword
- This is sometimes used intentionally in constructors

```java
/*This class is legal Java code.
 * using a constructor like this is valid but controversial
 * code style
 */
public class Student{
    String name;
    int graduationYear;

    public Student(String name, int graduationYear) {
        //this.name is instance variable; name is parameter
        this.name = name;
        this.graduationYear = graduationYear;
    }
}
```

Classes and Things Like Scope

Accessing Classes From Each Other

- In Lab 2, we access the `Move` class from the `CoinStrip` class (let's take a look)
- We didn't import anything. Why is this OK?
- (long story short:) In Java, can access other classes in the *same directory* (stored in the same folder)
- There are more advanced ways to control this; we won't cover in Data Structures

Class Compilation

- When you compile a file, the Java compiler automatically compiles any other classes it accesses

- Example: let's compile `CoinStrip.java`. We'll see that `Move.java` and `OptimalAI.java` are also compiled

- This can be confusing! It's always OK to run `javac` on each file individually.

Object Considerations

`this` keyword

- (Rough) equivalent to `self` in Python
    - Unlike python: do not need to have as an argument to methods!
- Refers to the current object
- We saw: also works to access instance variables
- Let's look at how this is used in Lab 2

Testing object equality

- Do not use ==! This tests the *memory address* of the objects, rather than their *value*
    - We will use == for objects later in the course
- Use `.equals()` instead
- For now, this is most important for `Strings`

Object Equality: Incorrect Version

```
1  String s1 = "hello";
2  String s2 = "hello";
3  if(s1 == s2) { //don't do this!!!  Could give unexpected
       results
4      System.out.println("These strings are equal:");
5      System.out.println(s1);
6      System.out.println(s2);
7  } else {
8      System.out.println("These strings are not equal:");
9      System.out.println(s1);
10     System.out.println(s2);
11 }
```

Object Equality: Correct Version

```
String s1 = "hello";
String s2 = "hello";
if(s1.equals(s2)) { //compares the contents of s1 and s2
    System.out.println("These strings are equal:");
    System.out.println(s1);
    System.out.println(s2);
} else {
    System.out.println("These strings are not equal:");
    System.out.println(s1);
    System.out.println(s2);
}
```