Lecture 1: Introduction to CS 18 and Java

Contents

Motivating Problem

We want to manage information about our collection of pet armadillos. How do we create data for information about individual armadillos (or other real-world items) in Java?

Objectives

By the end of this lecture, you will know:

- our goals for the semester
- what classes and objects are in Java

By the end of this lecture, you will be able to:

- create simple classes and objects in Java

Conceptual Study Questions

- What is the difference between a class and an object?
- What is the role of the constructor?
- What are the areas of memory in a language?
- What operations in Java affect each of the areas of memory?

1 Welcome to CS18

Take yourself back in time to the early 1960s. None of us have been born (even the professor). The US and Russia have started the space race, but have not yet gotten to the moon. Microchips have just been invented. They replace large vacuum tubes (that used to be use in fat-backed TVs and monitors). The Apollo-mission computers are among the first to use the new microchips (which allow computers to start becoming smaller than this classroom). The amount of memory on the Apollo-mission computers is less than what you need to store 5% of a picture taken with a modern smartphone.[1]

And they navigated to and landed on the moon with that!

Okay, neat history, but why are we starting CS 18 this way?

Because if you want to understand a new technology or a new idea, you need to know the *context* of what was going on when it arose. Historians know this, but it's something we often overlook in CS. That's a pity, because knowing the context provides a lot of the texture in any field, and some ideas about why the new thing was designed the way it was.

In the mid 1960s, a group of computer scientists in Norway were trying to figure out how to use computers to tackle other new problems as well. They wanted to create software that could simulate physical systems that responded to events (like people pushing elevator buttons, or trains arriving at signal crossings). The programming languages of the day weren't proving up to the task. So these scientists, Kristen Nygaard and Ole-Johan Dahl (with help from some others) invented a new language called SIMULA. It was the first so-called "object-oriented" language. They won the Turing award (the "Nobel prize" of Computer Science) for this work.

And this semester, we are going to teach you object-oriented programming using Java, which descends from SIMULA.

But we aren't here just to teach you object-oriented programming. We're here to teach you fundamental principles of computer science, including data structures, analysis, software design, and testing. All of which are driven by needs to build computing systems that are efficient, robust, and maintainable. We start this course tipping our hats to the early pioneers of computing who did amazing things with limited technology, as we help prepare all of you to do amazing things with powerful technology.

Because the designers of SIMULA did one of the most fundamental tasks in computing: they tried to tackle a new kind of problem, realized they couldn't express the problems cleanly, and built a new language to make their lives easier. One of the most powerful and beautiful things about computing is that we get to define how our systems work. We aren't like the physical scientists, stuck to working with what nature gave us. We build our tools, our systems, our tools (sometimes, those tools cause new problems—that's another important part of the story).

To help prepare you to do that, we need to expose you to different ways of expressing problems. So we switch styles of languages in CS 18, from functional to object-oriented and imperative programming (in Java), then pull those two styles back together (in Scala in the second half of the semester). Most of you probably don't really understand why you learned functional programming in the first semester. Many of you likely aren't sure what "functional programming" even means. As the course goes along, we'll return to these questions of language, data structure, and software design

[1]The Apollo computer had less than 80K bytes of memory: `https://en.wikipedia.org/wiki/Apollo_Guidance_Computer`

(including details of the problem that objects solve), giving you a foundation for whatever else you plan to do with computing from here.

Concretely, this semester you will

- start learning two new languages, Java and Scala, that enable programming with objects
- learn how to use and implement several new data structures
- learn how to choose among data structures and algorithms for common computing problems
- see some neat applications of computer science, building a machine learning classifier, the core of a search engine, and both a web server and browser
- think about how to assess algorithms for their performance around time, space, and social impacts

All of the core information you need—lecture notes, assignments, course policies, staff contact information, and so on—are on the course webpage. The course will have a weekly 2-hour lab, 3 projects, 8 homeworks, and both a midterm and a final exam. The first homework, which asks you to think about an emerging issue in computer science and tell us about yourself and your background, is posted.

In other words, we're all ready to get started, and hope you are too!

2 Migrating to Java

Our first task in this course is to help you migrate what you learned in 111/112/17/19 to Java. A few new concepts will come up this week, but mostly we are shifting you to a new language and new way of organizing code.

We'll show the migration to Java by writing code that creates and operates on (arma)dillos, a kind of data that we want to create (looking ahead to writing programs to manage a small zoo). Let's assume two pieces of information matter about each dillo: how long it is, and whether or not it is dead. We will also write a function called `canShelter` that consumes a dillo and determines whether the dillo is both dead and large enough that someone could take shelter in its shell (this isn't hypothetical: a relative of the armadillo, the Glyptodon (`https://en.wikipedia.org/wiki/Glyptodon`), could grow as large as a Volkswagen and was believed to be used for human shelter).

Those with prior Java experience likely learned different ways to do some of what we cover here. Hold tight – we'll explain why we are doing things differently as we go.

Here is the ReasonML code, showing the design recipe steps, that we will convert to Java in these notes:

```
/* Data Definition */
type dillo = { length : int, is_dead : bool };

/* Examples of Data */
let baby_dillo = {length: 8, is_dead: false};
let adult_dillo = {length: 24, is_dead: false};
let huge_dead_dillo = {length: 65, is_dead: true};

/* A Function */
let can_shelter = (d) =>
    d.length > 60 && d.is_dead ;

/* Test cases */
can_shelter(baby_dillo) /* should be false */
can_shelter(huge_dead_dillo) /* should be true */
```

Our task is to migrate the ideas in this code one step at a time, starting with the *data definition*, the part of the code that indicates that we want to be able to create dillos.

2.1 Migrating Data Definitions

In Java, if you want a kind of data that isn't something simple like a number, string, or boolean, you create a *class*. Classes let you specify *compound data*, which is data that has different components (such as a phone contact having both a name and a phone number).

The following Java code defines the Dillo class with components for length and death status:

```
public class Dillo {
  public int length;
  public boolean isDead;

  public Dillo (int len, boolean isD) {
    this.length = len;
    this.isDead = isD;
  }
}
```

The first three lines capture the class name (`Dillo`), field names (`length` and `isDead`), and *types* for each field (`int` and `boolean`).

The next three lines of code define the what is called the *constructor*: the function you call to create armadillos. The name of the class comes first, followed by a list of parameters, one for each field name (on the first line). The second and third lines store the parameter values in the actual fields.

2.2 Migrating Examples of Data

After creating classes, you should always create some *examples of data* from your classes. These examples both show how to use your classes and give you ready-made data inputs to use in testing your functions. In general, your examples should cover the interesting options within your data: for dillos, this means having examples of each of live and dead dillos, including of different lengths.

Here, we will make three dillos, a live one of length 8, a live one of length 24, and a dead one of length 65. Here's what the live dillo of length 8 looks like in ReasonML.

```
let baby_dillo = { length = 8; is_dead = false} ;
```

In each of Racket, ReasonML, and Pyret, you could simply put these definitions in your file (at the so-called "top level"). In Java, all definitions must lie inside classes. We therefore need a class in which to put our examples. We will create a class called `AnimalTest` to hold all our examples of data (and eventually our tests) for our zoo application:

```java
public class AnimalTest {
    // if the constructor needs no parameters, it can be omitted.
    // we included it here to help you see the pattern.
    public AnimalTest() {} ;

    Dillo babyDillo = new Dillo (8, false);
    Dillo adultDillo = new Dillo (24, false);
    Dillo hugeDeadDillo = new Dillo (65, true);
}
```

The first line inside the class is the constructor for the class. Since this class has no fields, the constructor is trivial. Technically you could omit the constructor in this case, but we include it for completeness (so you can see what an empty construtor looks like).

The next three lines create Dillos and store them under names that we can use to retrieve them later. To create a Dillo in Java, we use the `new` construct, followed by the name of the class you want to create and the parameters required by the constructor for that class. To save a value (like a Dillo) under a name, we write the type of the value, the name, an = sign, and the value to assign to the name.

When you use the `new` operator, Java performs the operations in the constructor: so `new Dillo(8, false)` creates a Dillo, sets the new Dillo's `length` to 8 and its `isDead` to false.

Things to note about creating examples of data:

- Names in Java cannot contain hyphens or other punctuation. The convention in Java is to use something called *camel case*: string the words together by using a capital letter for each word after the first. Thus, `huge_dead_dillo` in ReasonML becomes `hugeDeadDillo` in Java.

2.3 Running Programs

We now have a program containing two classes: `Dillo` and `AnimalTest`. How do we actually run the program?

We're going to hold this thought until the next lecture. In Racket, ReasonML, or Pyret, you could simply evaluate an expression at the REPL prompt. IntelliJ, our Java programming environment, doesn't offer this interactive feature. Instead, you start off a program by running a function, but we haven't learned how to write functions yet! We will do that next time, at which point we'll show you how to run a program.

3 Key Terminology: Objects

So far, our `AnimalTest` class mainly has uses of `new` whose results are stored under names. Whenever you use `new` on a class, you get something called an *object*. An object represents a particular value

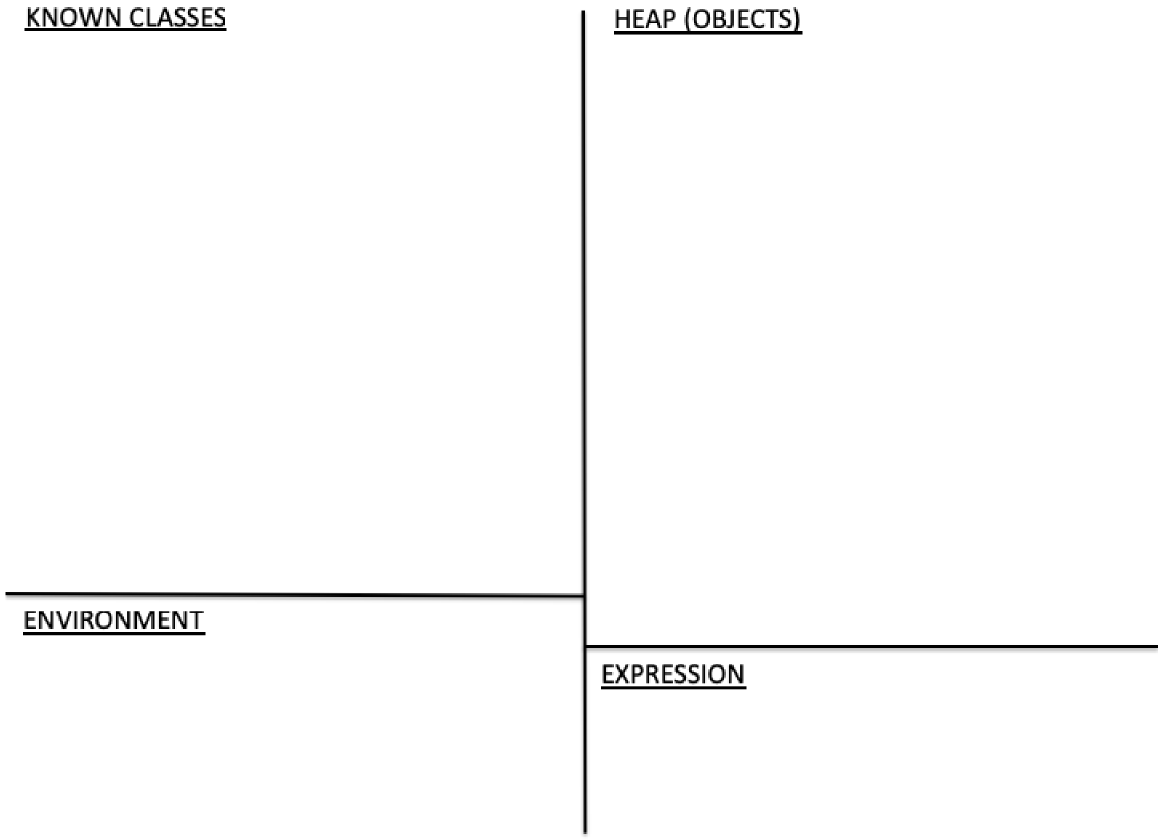

Figure 1: A blank map for tracking objects and names.

or entity within that class. For example, `new Dillo(8, false)` is a concrete live Dillo of length 8. A class, in contrast, describes a kind of data you want to create, but lacks specific values for the component data. Some people explain objects with a physical analogy: an item that exists in the physical world (such as a specific Dillo along the roadside) corresponds to an object, while a description of what information makes up a Dillo corresponds to a class. We say that an object is an *instance* of a class.

The difference between classes (descriptions of data) and objects (actual data) is an essential concept in programming. You've encountered this distinction before, but perhaps under different terminology. In ReasonML, you used the term `value` for concrete data, but that also included simple data like numbers and strings. Here, objects are values that are made by calling `new` for a class.

Other nuances of classes and objects will come up as we begin writing functions. Before we get to that, let's stop and make sure you understand what happens under the hood when you use the `class` and `new` constructs.

4 Under the Hood: Classes, Objects, Naming, and `new`

To understand how Java "works", we will map out what happens under the hood as you compile and run programs. Java organizes your programs and computations into four main "areas" under the hood: *known classes*, *existing objects*, *named values*, and the *current expression* that we are evaluating (sometimes called a *program counter*) . Before you compile a program, all of these are empty (well, mostly: Java has a bunch of built-in classes, but we haven't talked about those yet). Figure 1 shows an empty map from before you compile a program (a PDF version is at `https://brown-cs18-master.github.io/content/lectures/01intro/blank-memory-model.png`).

Different areas of the map get populated by different constructs and operators in Java. Roughly speaking, the `class` construct modifies the *known classes* area (in the compile step), `new` modifies the *existing objects* area (in the run or testing steps), and `=` modifies the *named values* area (in the run step).

Here's a link to a brief powerpoint presentation that illustrates an example of how running Java expressions against a class affects the memory map: `https://brown-cs18-master.github.io/content/lectures/01intro/notional-machine-classes-objects.pptx`

What should you understand from this?

- The *known classes* area gets modified when you *compile* the program – that's when Java finds out what classes your program knows about.
- The *existing objects* and *named values* areas get modified when you *run* or *test* the program. While we've only looked at defining examples so far, you can imagine from your prior programming experience that you could create objects and name values frequently while running a program. These areas are much more dynamic.
- Objects are only accessible through names. Under the hood, there are often objects that exist but aren't accessible. *This isn't a problem!*. What matters is that you have names for the objects that you need to get to. This is actually a fairly complex topic that we will revisit throughout the course.

We will return to this picture throughout the course. If you aren't comfortable with it, be sure to do the extra exercises (linked to the lecture notes page) for working with the program-execution map.

5 Wrapping Up

Next lecture, we'll come back and learn how to define functions in the context of classes. The first homework and first lab are posted and we'll be in touch with more details soon.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: `https://cs.brown.edu/courses/cs018/feedback`.