# CS200: Migrating to Java – Classes, Methods, and Tests

Kathi Fisler

January 28, 2022

### 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?

### Conceptual Study Questions

- What is the difference between a class and an object?
- What is the role of the constructor?
- How does having a function inside of class (as a method) rather than as a top-level function change what data the function can access?

## 1 Migrating to Java

Our first task in this course is to help you migrate what you learned in 111/112/17/19 to Java. 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, 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 are the ReasonML and Pyret versions of the code, showing the design recipe steps, that we will convert to Java in these notes:

```
/* ReasonML -- 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 */
```

---

```
# Pyret -- data definition

data Dillo:
  | dillo(length :: Number, is-dead :: Boolean)
end

# examples of data
baby-dillo = dillo(8, false)
adult-dillo = dillo(24, false)
huge-dead-dillo = dillo(65, true)

# a function
fun can-shelter(d :: Dillo) -> Boolean:
  (d.length > 60) and (d.is-dead)
where:
  # test cases / examples of use
  can-shelter(baby-dillo) is false
  can-shelter(huge-dead-dillo) is true
end
```

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.

## 1.1 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.

## 1.2 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 AnimalsTest to hold all our examples of data (and eventually our tests) for our zoo application:

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

    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 Key Terminology: Objects

So far, our AnimalsTest 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 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 Pyret and 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.

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

    // the constructor
    public Dillo (int length, boolean isDead) {
        this.length = length ;
        this.isDead = isDead ;
    }

    /**
     * determines whether dillo is dead and longer than 60
     */
    public boolean canShelter() {
        return (this.isDead && this.length > 60);
    }
}
```

Figure 1: A sample method on Dillos

# 3 Extracting Field Values

We've seen how to create objects (using new), but not how to extract information from them. What if I wanted to know whether babyDillo is dead? In Java, we would write:

```
babyDillo.isDead
```

The pattern here is OBJECT.FIELD – you write down the object whose data you want, followed by a period, followed by the name of the field.

# 4 Migrating Functions

Now we will write the canShelter function over Dillos in Java. In object-oriented programming (OOP), functions are placed in the class for the primary data on which they operate: this is one of OOP's hallmark features (we will talk about why in a couple of weeks). OOP uses the term **method** instead of function.

As a reminder, the method we are trying to write determines whether a Dillo is dead and large enough to shelter a person (the code is in the lecture 1 notes). For the second criterion, we will check whether the Dillo has length longer than 60 (inches).

The Java method for this appears in Figure 1. The lines with the asterisks are comments (in the style used to document the purpose of a method). The first line of the definition states the type of data returned (boolean), the method name (canShelter), and parameters (none in this case). The second line contains the body of the method, prefixed with the keyword return (required). The body of the method shows the && notation for writing and in Java.

But wait – didn't we initially say that the canShelter method should take a dillo? It did in the Reason/Pyret code. Why isn't there a parameter then? This is one of the essential traits of object-oriented programming. Every method goes inside a class. That means the only way you can "get to" a method in order to call it is to go through an object (in this case, a dillo). *Since you need an object to even call a method, you don't need that object as a parameter.*

Which brings us to the this that you see before the fields in the method body. This says "take the field values from this object" (as opposed to some other object). An example of calling the function will help us explain this more clearly.

Assume you wanted to know whether babyDillo can shelter a human. You would write the expression:

```
babyDillo.canShelter();
```

You can read this code as "Run the canShelter method using babyDillo as the this object. What you are really asking Java to do is go inside *babyDillo*, lookup the canShelter method, and run it (on no arguments, as the method requires).

When you run this expression, Java will evaluate the body of the method, which contains

```
this.isDead && this.length > 60
```

Here, this refers to the object that you used to get to the method. So it is as if you typed

```
babyDillo.isDead && babyDillo.length > 60
```

Java doesn't actually rewrite your code to replace "this" with "babyDillo", but that is the essence of what happens under the hood. If you had called

```
hugeDeadDillo.canShelter()
```

Java would instead use the values of isDead and length that are stored inside hugeDeadDillo.

Note the similarity and differences between accessing fields and methods in Java objects. Both take the form object.<item>, but methods require a (possibly empty) list of arguments after the method name.

One more thing to note about the method definition:

- For multi-word method names, we use a convention in which we use lower case for the first word and upper case for the rest (unlike Racket, Java doesn't allow hyphens in method names).

# 5 Migrating Test Cases

Whenever you write a method, you should write some examples (or tests) showing how you expect the method to behave – this is the same practice you followed last semester. Normally, we write examples of method use *before* we write the method itself. In this introductory segment, we showed you how to write methods first as that provides useful context for writing test expressions for them. We'll be using a framework called *Junit* for writing tests.

Like all Java code, test cases must be placed in a class. Test cases are written as methods with a particular naming convention, input, and return type. We already have a class with the leading name AnimalsTest. Before we test canShelter, lets write a test method that checks whether twice the length of adultDillo is 48. Ignore all of the import lines at the top of the file for now, they are just things we need to use with JUnit.

```
import org.junit.Assert;
import org.junit.Before;
import org.junit.FixMethodOrder;
import org.junit.Test;
import org.junit.runners.MethodSorters;

@FixMethodOrder(MethodSorters.NAME_ASCENDING)

public class AnimalsTest {
    AnimalTest(){};

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

    /**
```

```
    * checks computations on the length of adultDillo
    */
  public void testExample() {
      // example syntax for assertEquals
      Assert.assertEquals(adultDillo.length * 2, 48);
  }
}
```

The inner part of the test is an Assert statement, with a computation to run and its expected answer. This is similar to what those coming from 111/112/17/19 have written previously. The rest is largely formatting:

- test methods must be annotated with @Test before the header line.
- test methods return void, which means that no value is returned, but the computation in the method is still done. The tester will write out the results of tests, so nothing needs to be returned.

How about the tests we wanted to write on canShelter? Here they are, written against JUnit:

```
/**
   * check canShelter on small live dillos
   */
 @Test
public void testBabyShelter() {
  Assert.assertEquals(babyDillo.canShelter(), false);
}

/**
   * check canShelter on large dead dillos
   */
// if no expected answer is given, the tester compares to true
public void testHugeDeadShelter() {
  Assert.assertEquals(hugeDeadDillo.canShelter(), true);
} }
```

If you wanted instead to write a single test method that covers multiple cases of sheltering, you could also have written these as follows:

```
/**
   * check canShelter on multiple dillos
   */
// the ! here is the Java operator for not/negation
public void testShelter() {
  Assert.assertEquals(babyDillo.canShelter(), false);
  Assert.assertEquals(hugeDeadDillo.canShelter(), true);
}
}
```

You have now seen your first complete Java program, along with the components you are expected to include (classes, examples of data, and test cases). You will have at least a Testing class and classes for the kind of data you are developing in every program you write for this course.

## 5.1 Running Programs

How do we actually run our Dillo and AnimalsTest program? In particular, how do we run our tests?

When you try to run a program, Java looks for a method named main. Java will run this method to run your program. As with all methods, main must live in a class. For the next few assignments, we will provide you with a class called TestRunner which will take care of running your tests.

```
  public class TestRunner {
    public static void main(String[] args) {
        Result result = JUnitCore.runClasses(AnimalsTest.class);
        for (Failure failure : result.getFailures()) {
            System.out.println(failure.toString());
        }
        if (result.wasSuccessful()) {
            System.out.println("all tests passed!");
        }
    }
}
```

For now, you don't have to understand how this works (we'll get there). All you need to do if you are writing your own test classes is copy over this file and change @codeAnimalsTest (on line 3 of the method) to the name of your class that holds your tests.

There are a lot of moving parts here, in part because Java was designed for writing larger programs in which all of this infrastructure is useful. For now, it will feel like (and is) overkill. But we'll get it sorted out soon.