

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

Kathi Fisler

January 29, 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:

```
1 /* ReasonML -- Data Definition */
2 type dillo = { length : int, is_dead : bool };
3
4 /* Examples of Data */
5 let baby_dillo = {length: 8, is_dead: false};
6 let adult_dillo = {length: 24, is_dead: false};
7 let huge_dead_dillo = {length: 65, is_dead: true};
8
9 /* A Function */
10 let can_shelter = (d) =>
11     d.length > 60 && d.is_dead ;
12
13 /* Test cases */
14 can_shelter(baby_dillo) /* should be false */
15 can_shelter(huge_dead_dillo) /* should be true */
```

```

1 # Pyret -- data definition
2
3 data Dillo:
4   | dillo(length :: Number, is-dead :: Boolean)
5 end
6
7 # examples of data
8 baby-dillo = dillo(8, false)
9 adult-dillo = dillo(24, false)
10 huge-dead-dillo = dillo(65, true)
11
12 # a function
13 fun can-shelter(d :: Dillo) -> Boolean:
14   (d.length > 60) and (d.is-dead)
15 where:
16   # test cases / examples of use
17   can-shelter(baby-dillo) is false
18   can-shelter(huge-dead-dillo) is true
19 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:

```

1 public class Dillo {
2   public int length;
3   public boolean isDead;
4
5   public Dillo (int len, boolean isD) {
6     this.length = len;
7     this.isDead = isD;
8   }
9 }

```

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

Lines 5-8 of the 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). Lines 6 and 7 store the parameter values in the actual fields. The term **this** in lines 6 and 7 indicate that we are setting the fields of the object that is currently being created (as opposed to some other `Dillo` object that we may have already created). We'll see more about **this** in the next class or two.

In Racket, Reason, and Pyret, constructors get defined for you automatically when you make a datatype. Java instead requires that the programmer write the constructor. There are two reasons for this:

- Sometimes, we want to do something to process or check the constructor arguments before creating the data value. In the case of the `Dillo`, for example, we might want to check that the given `len` value is a reasonable length, rather than a negative number or a large positive number.

```

1 public class Dillo {
2     public int length;
3     public boolean isDead;
4
5     // here is the original constructor with error checking added
6     public Dillo (int len, boolean isD) {
7         if ((len < 5) || (len > 80)) {
8             throw new IllegalArgumentException("invalid_length");
9         } else {
10            this.length = len;
11            this.isDead = isD;
12        }
13    }
14
15    // here is the second constructor
16    public Dillo (int len) {
17        this.length = len;
18        this.isDead = false;
19    }
20 }

```

Figure 1: Customizing Java constructors

- Sometimes, we want to have different constructors with different numbers of arguments, to allow for default values. For example, our default might be that all dillos are alive unless the person creating the data values wants to say otherwise.

Figure 1 shows examples of both of these. It also shows how we report invalid inputs on line 8 (by “throwing” an exception, which is another term for an error). If the data are valid, the constructor sets the fields as before.

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.

```

1 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:

```

1 public class AnimalsTest {
2     // if the constructor needs no parameters, it can be omitted.
3     // we included it here to help you see the pattern.
4     public AnimalsTest() {} ;
5
6     Dillo babyDillo = new Dillo (8, false);
7     Dillo adultDillo = new Dillo (24, false);
8     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 constructor 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.

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:

```
1 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

```

1 public class Dillo {
2     public int length ;
3     public boolean isDead ;
4
5     // the constructor
6     public Dillo (int length, boolean isDead) {
7         this.length = length ;
8         this.isDead = isDead ;
9     }
10
11     /**
12      * determines whether dillo could shelter a human
13      * @result true if dillo is dead and longer than 60
14      */
15     public boolean canShelter() {
16         return (this.isDead && this.length > 60);
17     }
18 }

```

Figure 2: A sample method on Dillos

(**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:

```

1 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

```

1 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

```

1 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

```

1 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).

Methods that take parameters: What if we want to write a method that takes inputs? What might that look like? As an example, here's a method that checks whether one `Dillo` is larger than a given `dillo`. In your previous languages, you'd have a function header like:

```
1 fun is-bigger(d1 :: Dillo, d2 :: Dillo) -> Boolean: ...
```

In Java, we rethink this as asking a question about a `dillo`: is it larger than another `dillo`. In Java, the method looks like:

```
1  /**
2   * Determines whether the dillo is larger than another one
3   *
4   * @param the Dillo to compare to
5   * @return true if this dillo is longer than given one
6   */
7  public boolean isBigger(Dillo other) {
8      return this.length > other.length;
9  }
```

Here, we start to reinforce the meaning of **this**: it refers to the fields of a specific `Dillo` object, in contrast to the fields of the same name in another `Dillo` object.

Javadocs: Documenting Methods: You've previously learned to put descriptive comments on your code. Java programmers use a specific format of comments, known as Javadocs. The benefit of this is that there are other programs that process Javadocs and produce readable documentation from them. We'll use a few common forms of Javadocs in this course.

The `isBigger` method shows an example of how we annotate a function with Javadocs. The block comment above the function (marked by the slash-and-asterisk pattern) contains a brief high level summary, a line that explains what each parameter represents, and a line describing what the returned value means.

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`, let's 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`.

```
1 import org.junit.Assert;
2 import org.junit.Before;
3 import org.junit.FixMethodOrder;
4 import org.junit.Test;
5 import org.junit.runners.MethodSorters;
6
7 @FixMethodOrder(MethodSorters.NAME_ASCENDING)
8
9 public class AnimalsTest {
```

```

10 AnimalTest(){};
11
12 Dillo babyDillo = new Dillo (8, false);
13 Dillo adultDillo = new Dillo (24, false);
14 Dillo hugeDeadDillo = new Dillo (65, true);
15
16 /**
17  * checks computations on the length of adultDillo
18  */
19 public void testExample() {
20     // example syntax for assertEquals
21     Assert.assertEquals(adultDillo.length * 2, 48);
22 }
23 }

```

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:

```

1 /**
2  * check canShelter on small live dillos
3  */
4  @Test
5  public void testBabyShelter() {
6      Assert.assertEquals(babyDillo.canShelter(), false);
7  }
8
9  /**
10   * check canShelter on large dead dillos
11   */
12   // if no expected answer is given, the tester compares to true
13   public void testHugeDeadShelter() {
14       Assert.assertEquals(hugeDeadDillo.canShelter(), true);
15   } }

```

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

```

1 /**
2  * check canShelter on multiple dillos
3  */
4  // the ! here is the Java operator for not/negation
5  public void testShelter() {
6      Assert.assertEquals(babyDillo.canShelter(), false);
7      Assert.assertEquals(hugeDeadDillo.canShelter(), true);
8  }
9  }

```

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.

```
1 public class TestRunner {
2     public static void main(String[] args) {
3         Result result = JUnitCore.runClasses(AnimalsTest.class);
4         for (Failure failure : result.getFailures()) {
5             System.out.println(failure.toString());
6         }
7         if (result.wasSuccessful()) {
8             System.out.println("all_tests_passed!");
9         }
10    }
11 }
```

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.