Lecture 2: Methods

Contents

1 A Quick Review **1**

2 Extracting Field Values **2**

3 Migrating Functions **2**

4 Migrating Test Cases **4**

4.1 Running Programs 6

5 Summary **6**

Motivating Problem

We want to write functions to process our Armadillos. How do we write and test functions in Java?

Objectives

By the end of this lecture, you will know:

- how to write methods (functions) in Java
- how to test methods in Java
- how OO languages organize methods and data

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

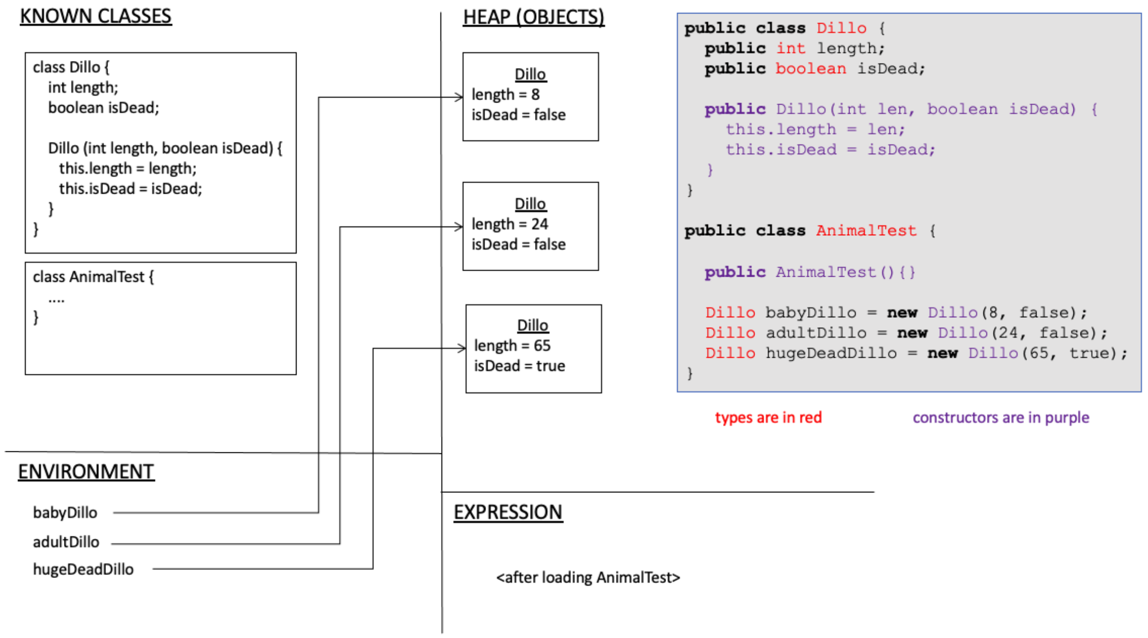
- write and test simple Java methods

1 A Quick Review

In the last lecture, we saw how to create a *class* in Java, which is how we create new forms of data that have fields. We created a class for armadillos that have two fields: a length and a boolean indicating whether or not a dillo is dead.

We also saw how to create *objects*, which are values that represent specific elements of a class. Classes describe the structure of a kind of data, while classes fill in specific values for the fields listed in a class. The keyword `new` is used to create objects from classes.

We also saw how memory changes as we create objects. Here is a diagram showing how memory would look after creating the three dillos from our `AnimalTest` class:



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

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

```
class AnimalTest {
   ....
}
```

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

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

public class AnimalTest {

  public AnimalTest(){}

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

We'll build on all of these today as we learn how to write functions in Java.

2 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.

3 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

```
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

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 OCaml 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).

4 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 *Tester* that you will practice during the first lab.

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 `AnimalTest`. Before we test `canShelter`, lets write a test method that checks that whether twice the length of adultDillo is 48 (we'll see why this is conceptually easier in a moment):

```java
import tester.Tester;

public class AnimalTest {
    AnimalTest(){};

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

    /**
      * checks computations on the length the adultDillo
      */
    public void testCheckLen(Tester t) {
        t.checkExpect(adultDillo.length * 2, 48);
    }
}
```

The inner part of the test is a `checkExpect`, 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 method names must start with “test” (otherwise the testing library won’t find them)
- test methods always take one argument (a `Tester` object, which here is named `t`)
- test methods return `void` (they can also return a boolean indicating whether a test passed or failed, but that can get cumbersome when you want to put multiple `checkExpects` in the same method, as you will do in lab). The type `void` 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.
- If you want to use the testing library, you need to include the `import` line that you see here at the top of the file.

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

```
 /**
    * check canShelter on small live dillos
    */
 // the ! here is the Java operator for not/negation
 public void testBabyShelter(Tester t) {
   t.checkExpect(!babyDillo.canShelter());
 }

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

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(Tester t) {
   t.checkExpect(!babyDillo.canShelter());
   t.checkExpect(hugeDeadDillo.canShelter());
 }
}
```

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.

4.1 Running Programs

How do we actually run our `Dillo` and `AnimalTest` 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. Since our goal is to run tests, we will put the main method into the `AnimalTest` class. Here is the method:

```
public static void main(String[] args) {
  Tester.run(new AnimalTest());
}
```

All of this is boilerplate, except for the name of the class whose tests you want to run (where `AnimalTest` appears in the method body).

Note: If you want less verbose reports from `Tester`. you can replace the `Tester.run` line with the following:

```
  Tester.runReport(new AnimalTest(), false, false);
```

Here's what happens when Java runs this method:

- it creates a new `AnimalTest`, which runs the `AnimalTest` constructor (that doesn't do anything in this case, but any code in the constructor would get run at this point).
- It evaluates all the expressions for creating examples of data (`babyDillo`, `adultDillo`, etc) from top to bottom.
- It runs each method in the class whose name starts with "test". There is no guarantee on the order in which the test methods run (does that matter?).

Under the hood, evaluating expressions and running tests can modify or consult the memory map. We'll demonstrate this in a future lecture. The result of running the program will show up in the Eclipse console window.

5 Summary

And we're off! You can create data, write methods, and write tests. We have a few more core Java ideas to show you, but you have enough to practice with now. If you want to practice, see the practice problems posted on the website.

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`.