CS 200 fall 2022 Lecture 2 (from OO): Records, tests, map/filter

Milda Zizyte

records

A record in Java is a lightweight, immutable class that lets you group data together. “Immutable” means that the data cannot be changed after it’s created.

To create a record, create a new Java file (e.g. Boa.java) and use the following syntax:

```java
public record Boa(String name, int length, String eats) { }
```

After we instantiate a record, we can print the record and access its fields (this code was in the main method of Lec02.java):

```java
Boa fredBoa = new Boa("Fred", 20, "lettuce");

System.out.println(fredBoa);
System.out.println("Fred eats: " + fredBoa.eats());
```

What we **cannot** do is change any value of fredBoa’s fields after we have already created fredBoa:

```java
// NONE OF THESE WILL WORK because fredBoa (and all record instantiations)
is immutable:
// fredBoa.length = 40
// fredBoa.length() = 40
// fredBoa.setLength(40)
```

Instead, we can make new Boas based on the values in fredBoa:

```java
Boa longFred = new Boa(fredBoa.name(), fredBoa.length() * 2,
fredBoa.eats());
System.out.println(longFred);
```

We can also write methods to work with Boas (we created these methods in Lec02.java):

```java
// returns true if the input Boa eats lettuce or carrots
public static boolean isVeg(Boa b) {
   return b.eats().equals("lettuce") || b.eats().equals("carrots");
}
```

```java
// returns a new Boa, triple the length of the input Boa
public static Boa growBoa(Boa b) {
   return new Boa(b.name(), b.length() * 3, b.eats());
}
```

Tests

One thing we will practice in CS 200 is using tests as examples that show that our methods are working (i.e. that the expected values match the actual results of the method call). The following tests were written in Lec02Test.java:

```java
@Test
public void simpleBoaTest() {
   Boa testBoa = new Boa("Test", 12, "Java");
   Assert.assertEquals(12, testBoa.length());
}

@Test
public void testIsVeg() {
   Boa b1 = new Boa("B1", 2, "lettuce");
   Boa b2 = new Boa("B2", 3, "profs");
   Boa b3 = new Boa("B3", 900, "carrots");

   Assert.assertTrue(Lec02.isVeg(b1));
   Assert.assertFalse(Lec02.isVeg(b2));
   Assert.assertTrue(Lec02.isVeg(b3));
   // the test below would have failed:
   //Assert.assertFalse(Lec02.isVeg(b3));

}

// test for Lec02.growBoa, which is supposed to triple a boa's length
@Test
public void testGrowBoa() {
   Boa b1 = new Boa("Fred", 20, "lettuce");
   Boa b2 = new Boa("Rude guy", 10, "profs");

   Assert.assertEquals(60, Lec02.growBoa(b1).length());
   Assert.assertEquals(10 * 3, Lec02.growBoa(b2).length());

   Assert.assertEquals("Rude guy", Lec02.growBoa(b2).name());
}
```

Dealing with lists of Boas

We can put Boas into a linked list (in the main method of Lec02.java):

```
LinkedList<Boa> boaList = new LinkedList<Boa>();
boaList.add(fredBoa);
boaList.add(longFred);
boaList.add(new Boa("Lily", 80, "ants"));

System.out.println(boaList);
```

In CS 15, if you had wanted to see which Boas in this list were vegetarian, you might have written something like (in the main method of Lec02.java):

```
System.out.println("with loop:");
for (Boa b: boaList) {
   if (isVeg(b)) {
       System.out.println(b);
   }
}
```

The idea here is that we are “filtering” through the list and only outputting the Boas for whom isVeg returns true. We can also do filtering using a built-in list mechanism (seen in many programming languages) called filter.

filter

A way of getting back a list of just the vegetarian Boas would be (in the main method of Lec02.java):

```
List<Boa> vegBoas =
       boaList.stream().filter(b ->
isVeg(b)).collect(Collectors.toList());

System.out.println("with filter:");
System.out.println(vegBoas);
```

The expression `b -> isVeg(b)` is called a “predicate.” It is saying: *As you go through the boaList, give every item the name b, in turn. Now call isVeg on b.* The filter only keeps the elements of boaList for which the predicate evaluates to true.

map

Instead of using predicates to decide whether to keep elements from a list, we can also use them with a mechanism called “map” that applies a method to every item in the list and returns a new list with the result. To see an example of this, say we wanted to triple the length of every

Boa in a list. We could write a predicate that applies growBoa to every element and put that predicate as an input to map (in the main method of Lec02.java):

```
List<Boa> looongBoas =
       boaList.stream().map(bo ->
growBoa(bo)).collect(Collectors.toList());
System.out.println("map to grow the boas:");
System.out.println(looongBoas);
```

Writing examples for map/filter

We can write tests (examples) for map/filter using the same idea of expected vs. actual results. Here is an example of doing that to make sure that our filter expression only gets the odd numbers in a list (in Lec02Test.java). Arrays.asList is just a shortcut way of quickly initializing a list for testing.

```
@Test
public void testOdd() {
   List<Integer> numList = new LinkedList<Integer>(Arrays.asList(0, 8, 3,
5));

   List<Integer> oddList = numList.stream().filter(n -> n % 2 ==
1).collect(Collectors.toList());

   Assert.assertEquals(new LinkedList<Integer>(Arrays.asList(3, 5)),
oddList);
}
```

Here, we put a simple mathematical expression inside of the predicate, instead of calling a method.