Lecture 6: Mutating Fields and Equality of Objects

Contents

Motivating Question

How can we update data in ways that are visible across all names that refer to an object?

Objectives

By the end of this lecture, you will know:

- How changing the value of fields reflects in the environment and heap
- That the = symbol has two very different meanings in Java

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

- Update the values in fields in objects
- Show how the environment and heap change with = operations
- Write an equals method for a class (notes here – will do next class)

1 Reviewing Lists in Memory

We started by reviewing the heap and environment contents after using our list classes from last week to construct a three element list. Watch the lecture capture for details. We will return to this example in the next lecture.

2 An Initial Example – Course Locations

Let's look at problems in a new domain: course scheduling. Here are some initial classes to get us started:

```
public class Course {
  String dept;
  int number;
  String room;

  public Course(String dept, int num, String rm) {
    this.dept = dept;
    this.number = num;
    this.room = rm;
  }
```

```
}

public class Student {
  String name;
  Course course1;

  public Student(String name, Course c) {
    this.name = name;
    this.course1 = c;
  }
}
```

As enrollments shift around during shopping period, the registrar sometimes needs to move courses to different rooms. So let's add a `newRoom` method to the `Course` class. Use only the concepts we have covered in CS18 so far:

```
  /**
   * produce same course but in a different room
   * @param newRoom -- the new room for the course
   */
  public Course newRoom(String newRm) {
    return new Course(this.dept, this.number, newRm);
  }
```

Let's check out how our new method works with a short example:

```
public class RegistrarTest {

 public static void main(String[] args) {
   Course cs18 = new Course("csci", 18, "BERT 130");
   Student mary = new Student("mary smith", cs18);
   Student ari = new Student("ari aman", cs18);
   cs18.newRoom("SAL 001");
   Student li = new Student("wang li", cs18);
   System.out.println(li.course1.room);
 }
}
```

If we run this code, what result do you want to see printed? What does get printed? What do the environment and heap look like just before the `println` command?

Since `newRoom` returns a new object, the heap (which you can see on the lecture capture video) now has two objects for `cs18`, but none of the students refer to that object. What if we instead handled the new room as follows:

```
   Course cs18new = cs18.newRoom("SAL 001");
   Student li = new Student("wang li", cs18new);
   System.out.println(li.course1.room);
```

Now `li` has the new room, but what about `mary` and `ari`? They still have the original room. Don't we want all students going to the right room for class? Of course! We need a better way to manage changes such as changing classrooms.

3 Updating Fields

For this example, we want to end up with a heap in which there is one object for `cs18` and it always contains the current room in which the class is meeting. We achieve this by having `newRoom` change the value in the current field, rather than by returning a new object:

```
/**
  * produce same course but in a different room
  * @param newRm -- the new room for the course
  */
 public Course newRoom(String newRm) {
   this.room = newRm;  // the = sign changes the field value
   return this;
 }
```

This new code might seem rather obvious. After all, our `Test` classes have been full of = as we associate names with examples of data. This is no different, right?

Actually, under the hood, it is hugely different, because the same symbol (=) plays two very different roles depending on where it occurs in your code.

- When you write `Student li = new Student("wang li", cs18);`, you are using = akin to the `let` construct from Racket or OCaml – you are adding a name to the environment, and binding that name to a value.
- When you write `this.room = newRm`, you are locating the field of an object in the heap and changing the value associated with it. The environment isn't involved at all. *You did not have the ability to do this operation in CS17.*

This is a big point, and one we will return to many times this semester.

3.1 Should we Return at All?

But wait – if we are changing the field, why return the object at all? Couldn't we just do the following?

```
/**
  * produce same course but in a different room
  * @param newRm -- the new room for the course
  */
 public void newRoom2(String newRm) {
   this.room = newRm;
 }
```

Either of these works for the purpose of changing the room, but sometimes it is much more convenient to return the object to the code that is calling the method. For example, if we also tracked the time of the course and wanted to change the hour, we might want to write:

```
cs18.newRoom2("SAL 001").newHour("A");
```

To write this, we need to return the course object. We'll come back to this point again in the next lecture.

Trace through our code example again, this time with the revised `newRoom` method (see the lecture capture for the details). Now, all three students access the same (updated) course location.

4 Equality

Now that we are changing the contents of objects in the heap, an interesting question arises: when are two objects considered "equal"? We often ask questions about equality in programming, such as:

- when searching for an item in a list
- when eliminating duplicates from a collection
- when checking that we have the right student before changing a course grade

So, here's a question: which of the following objects should we view as equal?

```
Course cs18a = new Course("csci", 18, "BERT 130");
Course cs18b = new Course("csci", 18, "SAL 001");
Course cs18c = new Course("csci", 18, "SAL 001");
```

Doesn't it depend on context? Sometimes, the different room is important and sometimes it isn't. Sometimes, having the same object at the same spot in memory is important and sometimes it isn't. There is no single uniliateral decision on which of these is "better".

So you need ways to express each of these in writing programs.

Java gives programmers control of how equality is defined for each class. Every class has a method called `equals`, that takes another (arbitrary) argument as an input and returns boolean indicating whether the two objects should be considered equal. By default, the `equals` method checks whether the `this` object and the given object are the same object inside the heap.

To demonstrate `equals` methods, assume we want two classes to be considered equal if they have the same dept and course number (ignoring the room). This could be useful when checking whether two students had both taken cs18, without caring about when they took it.

We present the method first, then explain the new constructs that are used here:

```
@Override
public boolean equals(Object other) {
   if (other instanceof Course) {
     Course c = (Course)other; // view (cast) object as a Course
     return (this.dept.equals(c.dept) &&
               this.number == c.number);
   } else {
       return false; // other isn't even a Course object
   }
 }
```

The core equality computation is in the return statement in lines 5 and 6. The expression `this.dept.equals(c.dept)` checks that the `dept` Strings in this and the other object are the same (using the built-in `equals` method on strings).

The type of the input to the method is `Object`. In Java, every class extends from a basic class called `Object` that defines default methods on all objects (like `equals`). This also means that `Object` is available as a type that refers to "any object from any class". While we might want to have the input be a `Course`, in theory any object can be compared to any other, so Java expects the `Object` type instead.

The if statement uses `instanceof` to ask "is this object actually a Course?". If it isn't, then we just return false (in the **`else`** case). If the object is a course, then we have to tell Java to view it as a course (the first line within the if case); The use of `Course` in parentheses before `other` does this (using a technique called *casting* that we will discuss more later this semester).

Note: `instanceof` can only be used in two situations in this course, with `equals` method being one of them. In general, asking what class an object is from is poor OO practice (we'll come back to why after we've covered more material).

5 A Warning

You might not appreciate it just yet, but with one little = sign, you now have incredible power. You can change the world (inside the heap). You can share new information with other parts of a program with ease. You can destroy the behavior of someone else's code. You create history, and make it impossible to rewind time if you made a mistake. That's a lot of responsibility for one little symbol.

Mutation is serious stuff. It's incredibly powerful, but carries risks. As we go forth in the course, we'll be talking about when to mutate objects and when not to. And of course it affects space/time tradeoffs. The moral for today is that we'll need to give equality the respect that it deserves.

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`.