Lecture 15: The Origin of Objects
10:00 AM, Feb 28, 2018

Contents

1 Finishing Exceptions 1

2 Why Objects? 1
   2.1 Example: A Simple Rope-and-Pulley system 1
   2.2 Limitations of this Pattern 2
   2.3 Objects 3

3 Could you do the same thing in .... Racket? 3

Objectives

By the end of these notes, you will know:

- What problem inspires combining data and functions
- How objects relate to closures

1 Finishing Exceptions

The first half of class was spent on finishing the exception notes posted from the previous lecture ... look at those notes for details.

2 Why Objects?

In the first lecture, we talked about how OO languages arose in the late 1960s. The first OO language, SIMULA 67, is credited to Kristen Nygård and Oleg Dahl [FIX!][FIX!] (though many of their colleagues were also involved in the design and development). Both of them were working on CS projects for the Norweigen defense industry. Simulating the behavior of nuclear reactors, for instance, was an example of what computer scientists were trying to figure out how to do at the time. More generally, the focus was on modeling and simulating systems made of physical components.

2.1 Example: A Simple Rope-and-Pulley system

For simplicity, we will work with a basic physical system in which a mechanical arm has a hand-turned crank at one end, a pulley on the other, and a rope passing over the pulley to lift a bucket.
Someone would turn the crank to move the bucket up and down. Someone could also adjust the angle of the arm relative to the floor.

If we wanted to simulate the movement of the bucket as people turn the crank and rotate the arm, we could have variables for the angle of the arm and the amount of rope hanging from the pulley (these are variables because their values would change during the simulation). We also need two constants, the diameters of the pulley and the crank wheel, to compute how much the rope moves. Roughly, we’d have the setup in the following code (with made-up values for the variables and constants):

```java
import math;

crankDiameter = 5;
pulleyDiameter = 10;
leverDegrees = 45;
ropeHangLength = 10;

// adjust the degree of rotation of the lever
moveLever(byDegrees) {
    leverDegrees = math.max(-90,
                            math.min(90, leverDegrees + byDegrees));
    ropeHangLength = ...;
}

turnCrank(byDegrees) {
    ropeHangLength = ... crankDiameter ... pulleyDiameter;
}

// compute distance of bucket to floor
bucketToFloor() {
    ...
}
```

This code is not Java, though it leverages Java syntax. It isn’t in a class. It’s just code at the top level of a file (sort of like you did in Racket), which is roughly how one would program in the 1960s. We haven’t written the expressions that compute new values of ropeHangLength because that isn’t the point (and we don’t want to focus on the wrong details). What matters is that we have a collection of functions that read and write to the variables and constants that define the rope-pulley system.

### 2.2 Limitations of this Pattern

The core pattern here is (a) a collection of variables that (b) get updated and referenced by a collection of functions that capture stimuli or operations on the system. This pattern works fine ... as long as there is only one pulley system.

What happens if the device we are simulating has two rope-pulley systems? What might the code need to look like then? Start with the variables and constants:

```java
// rope-pulley 1
crankDiameter = 5;
```
Now the functions. Something like turnCrank has to know which rope-pulley system it is working on. There are two general options in the style of code that we’ve started with. One is to copy the method, but refer to the variables and constants for the second system:

```java
turnCrank(byDegrees) {
    ropeHangLength = ... crankDiameter ... pulleyDiameter;
}

turnCrank2(byDegrees) {
    ropeHangLength2 = ... crankDiameter2 ... pulleyDiameter2;
}
```

The other is to have one function that takes another argument that says which set of variables to use:

```java
turnCrank(byDegrees, whichSystem) {
    if whichSystem == 1
        ropeHangLength = ... crankDiameter ... pulleyDiameter;
    else if whichSystem == 2
        ropeHangLength2 = ... crankDiameter2 ... pulleyDiameter2;
}
```

Either way, significant code has to get copied. What if someone makes a mistake in making the copy and mixing up variables across the two systems (in a complex system with dozens of variables or many functions, debugging that will be painful)? What if the computation needs to change, so edits are needed to every copy of a computation? Neither of these approaches are viable for any realistic system.

### 2.3 Objects

What we really want is a way to write the functions once, but to easily customize the functions to different copies of the variables. This is exactly what classes and objects do for us: with the class, we state what variables will be needed and capture the functions over them, but we don’t create the actual variables in memory. Each object holds a fresh set of the variables (at distinct locations in memory), while connecting to the class that defines the functions. It’s a beautiful separation of information that changes across uses to information (the functions) that stay the same.\footnote{The class/object distinction is not the only way to achieve this. There are object oriented languages without classes, for example, that still achieve this idea. The separation, however, is the high level idea.}
3 Could you do the same thing in .... Racket?

At first glance, classes and objects seem to give you a distinct advantage over working in something like Racket. The code you wrote in Racket looked more like where we started (a bunch of defines for constants and variables, with functions at the top-level). Could we have written the class/object pattern in Racket itself?

Yep, and it mostly uses constructs you learned in CS17, aside from the operator (set!) that changes the values of variables!

The following code shows the gear-pulley class and object as a Racket function:

```
(define (make-gear-pulley crankDiameter pulleyDiameter)
  ;; set up the variables
  (let ([leverDegrees 45]
        [ropeHangLength 10])
    ;; set up the functions/methods
    (let ([move-lever (lambda (byDegrees)
                        (set! leverDegrees ...)
                        (set! ropeHangLength ...))]
          [turn-crank (lambda (byDegrees)
                        (set! ropeHangLength
                               (... crankDiameter ... pulleyDiameter ...)))])
      ;; the "object" -- a value that receives messages and acts accordingly
      (lambda (message . args)
                (cond [(equal? message 'moveLever) (move-lever (first args))]
                      [(equal? message 'turnCrank) (turn-crank (first args))]
                      ...)))))
```

The `make-gear-pulley` function is the constructor: it takes values for the constants (they get “saved” for free as values of the parameters), and returns a closure/function (the “object”) that has the variables and methods captured inside the class. The “object” takes as inputs a “message” indicating what method to run and the arguments to that method (the “. args” notation gathers all arguments to a function into a list – this is a good way to handle differing numbers of inputs when calling a function. The object checks with method has been requested, then calls the function corresponding to the method.

Put differently, objects are just closures that can respond to multiple messages. Syntactically, setting up objects as lambdas may seem a little clumsy, which is why languages provide custom syntax for objects and (possibly) classes. Even Racket has a class and object system! But conceptually, objects are made out of constructs you used in CS17, albeit in slightly more advanced ways.

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: [http://cs.brown.edu/courses/cs018/feedback](http://cs.brown.edu/courses/cs018/feedback)