Unit 5: Information Hiding

After taking this unit, students should:

- understand the drawback of breaking the abstraction barrier
- understand the concept of information hiding to enforce the abstraction barrier
- understand how Java uses access modifiers to enforce information hiding
- understand what is a constructor and how to write one in Java

Breaking the Abstraction Barrier

In the ideal case, the code above the abstraction barrier would just call the provided interface to use the composite data type. There, however, may be cases where a programmer may intentionally or accidentally break the abstraction barrier.

Consider the case of `Circle` above, where we modify the radius `r` directly with `c.r = 10`. In doing so, we, as the client to `Circle`, make an explicit assumption of how `Circle` implements a circle. The implementation details have been leaked outside the abstraction barrier. Now, if the implementer wishes to change the representation of the `Circle`, to say, store the diameter, instead. This small implementation change would invalidate the code that the client has written! The client will have to carefully change all the code that makes the assumption, and modify accordingly, increasing the chances of introducing a bug.

Data Hiding

Many OO languages allow programmers to explicitly specify if a field or a method can be accessed from outside the abstraction barrier. Java, for instance, supports `private` and `public` access modifiers. A field or a method that is declared as `private` cannot be accessed from outside the class, and can only be accessed within the class. On the other hand, as you can guess, a `public` field or method can be accessed, modified, or invoked from outside the class.

Such a mechanism to protect the abstraction barrier from being broken is called *data hiding* or *information hiding*. This protection is enforced by the *compiler* at compile time.

In our original `Circle` class (v0.1) in Unit 4, we did not specify any access modifier -- this amounts to using the *default* modifier, the meaning of which is not our concern right now[1] For a start, we will explicitly indicate `private` or `public` for all our methods and fields.

```
1  // Circle v0.2
2  class Circle {
3    private double x;
4    private double y;
5    private double r;
6
7    public double getArea() {
8      return 3.141592653589793 * r * r;
9    }
10 }
```

Now the fields `x`, `y`, and `r` are hidden behind the abstraction barrier of the class `Circle`. Note that these fields are not accessible and modifiable outside of the class `Circle`, but they can be accessed and modified within `Circle` (inside the abstraction barrier), such as in the methods `getArea`.

Breaking Python's Abstraction Barrier

Python tries to prevent *accidental* access to internal representation by having a convention of prefixing the internal variables with `_` (one underscore) or `__` (two underscores). This method, however, does not prevent a lazy programmer from directly accessing the variables and possibly planting a bug/error that will surface later.

Constructors

With data hiding, we completely isolate the internal representation of a class using an abstraction barrier. But, with no way for the client of the class to modify the fields directly, how can the client initialize the fields in a class? To get around this, it is common for a class to provide methods to initialize these internal fields.

A method that initializes an object is called a *constructor*.

A constructor method is a special method within the class. It cannot be called directly but is invoked automatically when an object is instantiated. In Java, a constructor method *has the same name as the class* and *has no return type*. A constructor can take in arguments just like other functions. Let's add a constructor to our `Circle` class:

```
1  // Circle v0.3
2  class Circle {
3    private double x;
```

```
4     private double y;
5     private double r;
6
7     public Circle(double x, double y, double r) {
8       this.x = x;
9       this.y = y;
10      this.r = r;
11    }
12
13    public double getArea() {
14      return 3.141592653589793 * this.r * this.r;
15    }
16  }
```

Now, to create a `Circle` object, we need to pass in three arguments:

```
1  Circle c = new Circle(0.0, 0.5, 10.0);
```

Constructor in Python and JavaScript

In Python, the constructor is the `__init__` method. In JavaScript, the constructor is simply called `constructor`.

The `this` Keyword

The code above also introduces the `this` keyword. `this` is a reference variable that refers back to self, and is used to distinguish between two variables of the same name. In the example above, `this.x = x` means we want to set the field `x` of this object to the parameter `x` passed into the constructor.

Now that you have been introduced to `this`, we have also updated the method body of `getArea` and replaced `r` with `this.r`. Although there is nothing syntactically incorrect about using `r`, sticking to the idiom of referring to members through the `this` reference makes the code easier to understand to readers. We are making it explicit that we are referring to a field in the class, rather than a local variable or a parameter.

---

1. The other access modifier is `protected`. Again, we do not want to worry about this modifier for now. ↵