Chapter 2

Object-Oriented Modeling

Previously, we studied programming-in-the-small in Java, mapping our knowledge of C into this new, C-like language. This week, we will explore *object-oriented programming* in Java and how this style of programming is fundamentally different from writing programs in C.

2.1 What is an Object?

You may have heard the term "object" used in Racket or C as a synonym for "data". However in an object-oriented programming language like Java, *object* refers to a very specific kind of program entity:

Definition 1. *An object possess* state *and* behavior.

So far we have seen two types of objects in Java:

Strings Its state is the (immutable) sequence of characters it contains. An example of its behavior is the `trim()` method which produces a new string that is identical to the old string except the whitespace on either end is removed.

Arrays Its state is the sequence of elements it contains (accessed with index notation) along with its length (accessed via dot notation, *i.e.*, `arr.length`). It has no associated behavior.

In Java, we realize the state of an object as *fields* or *instance variables* of an object. Recall that a C `struct` is a collection of a data fields. The instance variables of an object are analogous to these data fields of a `struct`.

We realize the behavior of an object with *methods*. A method is a function that we "call on" a particular object. For example with the `trim()` method above, if we have a variable `s` of type `String`, then we can call its `trim` method as follows: `s.trim()`.

2.1.1 An Example of OO Design: Student

When we design a program in an object-oriented style, we think about modeling the data of our program as a collection of objects that interact with each other through their methods. For example, in a program that manages a course registration database, we will likely need to model a student. In this context, what are the *state* and *behavior* of a student? Here are some examples:

State First name, last name, birthday, gender, age.

Behavior Register (for a course), drop (from a course), withdraw (from the semester).

And we can consider other pieces of state and behavior as our problem demands it. The state of the student becomes fields or instance variables of our student objects. The behavior becomes methods, *e.g.*, we might realize the register behavior as the method `void register(Course c)` which takes a `Course` object (yet to be defined) as input, registers the student for the course, and returns nothing. But which student is registered for the course? The student that we *call the method on*, for example:

```
Student s = /* Some initialization... */
Course csc207 = /* Some more initialization... */
s.register(csc207);
```

We use *dot notation* to invoke a method on an object, just like how we use dot notation to access a field of an object, *e.g.*, the `length` field of an array. Unlike field access, however, we do not provide only the name of the method of interest, we also provide the list of arguments to pass to the method. In effect, the student `s` is the *implicit* first argument to the method—it is neither specified in the method declaration nor provided in list of arguments to the method.

Note that we have not chosen all the possible pieces of state and behavior for our student objects. For example, a student certainly has a height. However we likely do not need to know a person's height to manage their course registration. This is the fundamental *design* question of object-oriented programming: how do we best and most concisely represent the salient features of the data in our program? To answer this question, we must take into account the context in which we are designing the object, what types best capture our requirements, and whether we can be clever about our representation to avoid unneeded complexity.

2.1.2 The Author and Client Perspective on Objects

Another way to look at objects is from two dual perspectives. Someone is responsible for designing the objects we use, *i.e.*, deciding what the state and behavior of that object is. We call this party the *author* of the object. Once we have a definition of an object, people can then use those objects in their own program. We'll call these people *clients* of the object.

These two perspectives on objects are especially important for understanding the relationship between author and client when *interfaces* are involved. There are potential obligations on either side of the object line—author or client—that must be met to guarantee that the program works correctly. We'll discuss these various sorts of *pre-conditions*, *post-conditions*, and *invariants* in the next chapter.

For now, this division is important for our purposes because it outlines our trajectory for discussing objects. We'll first talk about using objects as clients: creating already-defined objects, accessing their state, and invoking their behavior. All of this will feel eerily like using `struct`s in C. Then we'll talk about all the details of authoring objects—defining templates for objects called *classes* as well as all of the features we can throw into our objects to make them robust abstractions for others to use.

2.2 The Client Perspective on Objects

As a client of objects, there are three fundamental operations we can perform on them:

1. We create or *instantiate* objects.
2. We access the state or *instance variables* of objects.

3. We invoke the behavior or *methods* of objects.

We'll explore how we do each of these three operations in Java.

2.2.1 Object Instantiation

For the two objects that we've seen so far, arrays and strings, we saw special syntax for their creation. For arrays, we could either:

1. Instantiate an array of a given size and type whose elements are initialized to be the default values of that type or
2. Instantiate an array to an initial set of contents with special *array initialization syntax*.

```
int[] arr1 = new int[20];                    // initialized to an array of 20 zeros.
int[] arr2 = new int[] { 0, 1, 2, 3, 4, 5 }; // initialized to an array containing 0--5
```

For strings, we used string literals to create strings, *e.g.*, `"Hello World!"`. However, we have already learned of an alternative way to create a string. We can create a string from an array of characters as follows:

```
char[] arr = new char[] { 'h', 'e', 'l', 'l', 'o' };    // Note: no null character!
String s = new String(arr);                             // The string "hello"
```

This is an example of the primary way of creating objects: *new expressions*. In general, a new expression has the following form:

```
new <class name>( <parameters> )
```

This syntax invokes the *constructor* of an object which:

1. Allocates the memory for that object.
2. Initializes that object given the arguments to the constructor.

Here, the expression `new String(arr)` creates a new string object by invoking the `String` constructor that takes a `char[]` as input. The result is a string containing the characters found in that array.

Another example of object instantiation that we've seen is the `Scanner`:

```
Scanner in = new Scanner(System.in);
```

`new Scanner(System.in)` creates a new `Scanner` object by invoking the constructor that takes an `InputStream` object as input. (It turns out that `System.in` is an object of type `InputStream`. More specifically, `in` is the object, and it is a *static* member of the `System` class, an important distinction we'll discuss shortly.)

2.2.2 Accessing State

To access the state of an object, we use *dot notation*. It has the following syntax:

```
<object>.<field name>
```

An example of this syntax is accessing the length of an array:

```
int[] arr = new int[5];
System.out.println(arr.length);
```

Think of the dot ('.') as a *binary* operation like addition, except that the left-hand side of the dot is an expression that must evaluate to an object and the right-hand side is the name of the field of the object that we would like to access.

Normally, we would be able to *mutate* or change a field that we have access to, just like how we can modify the fields of `struct`s in C. However, if we try to do this to an array we get the following error:

```
arr.length = 6
/* ERROR: cannot assign a value to final variable length */
```

This is because the `length` field of the array is constant or `final`; it cannot be changed.

2.2.3 Invoking Behavior

Recall that the behavior of an object corresponds to the set of *methods* that the object exposes to a client. To invoke a method on an object, we also use dot notation:

```
<object>.<method name>( <parameters> )
```

For example, we used the `charAt` method of String objects extensively in previous labs and homework:

```
String isbn = "123456789X";
System.out.println(isbn.charAt(9));   // 'X'
```

`s.charAt(n)` fetches the nth character from the string `s`. Here, the object is `s`, a string, and the name of the method is `charAt`. The single parameter to `charAt` is the index of the `char` we wish to fetch.

Method invocation looks a lot like function calls and indeed they are very similar. If we're not being ultra-precise about our language, we may call `charAt` a function call or function call-looking thing a method invocation, and that is fine when discussing code casually. However, let's make the distinction between the two explicit, especially since we are coming from a C background:

1. A *method invocation* is always invoked on a particular instance of an object, namely, the object that appears to the left of the dot.
2. A *function call* or *application* is not invoked on a particular object. It looks like the normal C function calls that you are used to.

As we transition into the world of objects, it'll be important to keep the two types of function-like calls straight in our head.

Exercise (String Methods) In a class called `StringMethods`, write a `static` method called `endsWithRep` with method signature:

```
public static boolean endsWithRep(String s1, String s2, int n);
```

that returns `true` if `s1` ends with `n` repetitions of `s2`.

Here are some example invocations of `endsWithRep`:

```
endsWithRep("foobarbar", "bar", 2);    /* returns true  */
endsWithRep("foobarbar", "baz", 1);    /* returns false */
```

To write this method, you can use string concatenation (the `+` operator) to build up a string that is `n` repetitions of `s2` and the `boolean endsWith(String suffix)` method of the `String` class which returns `true` if `suffix` is indeed a `suffix` of the string the method is called on. In the `main` method of `StringMethods`, demonstrate that your method works by printing the results of the two examples above to the console.

2.3 Blueprints for Objects

Now that we have seen how to interact with objects from the client perspective, let's now discuss how to specify objects. To do this, we'll formally introduce the *class* construct that has been in every one of our Java programs so far.

2.3.1 Anatomy of a Class

Recall that an object is a programming entity that contains state and behavior. To specify what kinds of state and behavior an object contains, we use *classes*. We say that an object is an *instance* of some class. For example, recall from the previous section we defined a Student and defined the following pieces of state and behavior for it:

State First name, last name, and age.

Behavior Register, drop, and withdraw.

Here is how we would take these pieces of state and behavior and define them in a Java class:

```
public class Student {
    public String firstName;
    public String lastName;
    public int age;

    public Student (String firstName, String lastName, int age) {
        /* ... */
    }

    public boolean register(String course) {
        /* ... */
```

```
    }

    public void drop(String course) {
        /* ... */
    }

    public void withdraw() {
        /* ... */
    }
}
```

For the time being, we've elided the implementations of the constructor and methods. But regardless, we can use this class as follows:

```
Student s = new Student("Ada", "Lovelace", 23);
s.register("csc 207");
```

The state of an object translates into *field* or *instance variable declarations* in our class. You should recognize these from C; these look and behave the like the field declarations of a `struct` definition. The behavior of an object translates into *method* definitions. These are like function declarations but appear within the class definition *without the* `static` *modifier*. Adding `static` no longer associates the function with an instance of the class—a distinction that made the function a method—but rather with the class itself. We'll explore this distinction in more detail later as it is one of the greatest points of confusion for students transitioning from the programming-in-the-small world of C to the object-oriented world of Java.

The `public` annotations on the class, the fields, and the methods determine the *visibility* of that particular program entity. Keeping the client versus author distinction in mind, an entity marked `public` is usable to everyone—clients and author alike. In contrast, a entity marked `private` is usable only by the author. This is useful for hiding fields and methods that concern the *implementation* of the class, *e.g.*, auxiliary functions and state that we don't want the outside world to know about.

A class is a sort of swiss-army knife in object-oriented languages like Java; it does a bunch of stuff:

- Classes act as a blueprints for objects as we discussed before.
- Classes are the way of defining *user-defined* types in Java.
- Classes act as a *namespace* for collections of `static` functions and variables.
- Classes act as an abstraction mechanism separating *features* from *implementation* through *interfaces*.

It is easy to conflate all of these features together, especially if Java is your first language. However, with multiple languages under our fingertips, we can see that all of these features serve their own distinct purposes.

2.3.2 Class Declaration Syntax

Class declarations take on the following form:

```
<visibility modifier> class <class name> {
    <field and method declarations>
}
```

A program in Java is defined to be a collection of class declarations. The class declaration may optionally be preceded by a visibility modifier, *e.g.*, `public` or `private`. The visibility modifier may be left out which turns out to have a different meaning from `public` or `private` ("package''-protected which we'll discuss later when we talk about Java's package system).

A class definition contains a number of *declarations*:

1. Field declarations.
2. Method declarations.
3. Constructor declarations.

Field declarations look like local variable declarations but exist outside of any particular method but inside a class definition:

```
<visibility modifier> <type> <name>;
```

For example, `public String name;` in the declaration of `Student` above declares a field of type `String` named `name`. Every instance of a `Student` has their own `name` field.

Method declarations look a lot like the function declarations we have seen so far:

```
<visibility modifier> <type> <name>( <arguments> ) {
    <statements>
}
```

except that there is no `static` in the signature of the method. As discussed ealrier, this is the distinction between a *method*, a function tied to a particular object, and a *static function*, a function not tied to a particular object but the overall class.

2.3.3 Constructors

Recall that fields and methods gives us way of using objects. Constructors give us ways of creating objects, a process called *instantiation*. The constructor defines how we should *initialize* a freshly-created object of the given class. We define a constructor in a class similarly to how we define a method except:

1. The name of the method is the name of the class.
2. There is no return type.

For example, recalling the constructor of our `Student` class above:

```
public Student (String firstName, String lastName, int age) {
    /* ... */
}
```

This is a constructor that takes three arguments: two strings corresponding to the person's name as well as their age. This constructor allows us to create a `Student` using a `new` expression as follows: `new Student("Ada", "Lovelace", 23)`.

2.3.4 this

In the example above, we have omitted the implementation of the constructor. What should the constructor do to initialize a new `Student`? A sensible sketch of an approach is:

```
public Student (String firstName, String lastName, int age) {
    // Initialize the firstName field of Student with the parameter firstName
    // Initialize the lastName field of Student with the parameter lastName
    // Initialize the age field of Student with the parameter age
}
```

However, how do we access the fields of the object that we are creating? Recall that the syntax of a field access is `<object>.<field name>`, but what goes to the left-hand side of the dot? If we need to refer to the object that we are currently instantiating (in a constructor) or called the method on, we use the `this` keyword:

```
public Student (String firstName, String lastName, int age) {
    this.firstName = firstName;
    this.lastName  = lastName;
    this.age       = age;
}
```

In a method or constructor, `this` is an expression that evaluates to the object that is the subject of the constructor or method call.

2.3.5 An Example: The Counter Class

As a complete example to study, let's consider creating a class that represents a simple counter that we can increment. What is the state and behavior of a counter?

State The current value of the counter, an integer.

Behavior Incrementing the counter.

Now let's translate this into a simple Java implementation:

```
public class Counter {
    public int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value += 1;
    }
}
```

Here's an example of using this `Counter` class:

```
Counter c1 = new Counter();
Counter c2 = new Counter();
System.out.println(c1.value);   // 0
System.out.println(c2.value);   // 0
c2.increment();
c1.increment();
c1.increment();
c2.increment();
c2.increment();
System.out.println(c1.value);   // 2
System.out.println(c2.value);   // 3
```

Note that each instance of the `Counter` possesses a distinct `value` field. So each call to `increment()` increments the `value` field of the counter that the method is called on.

A final note: because we annotated `value` with the `public` visibility modifier, anyone can change the value of a counter. For example:

```
c1.value = 5;
System.out.println(c1.value);   // 5
System.out.println(c2.value);   // 3
```

This may be fine for our simple purposes, but we may want to *hide* this field so that non-counter code cannot change the value directly. We can accomplish this by marking the field `private` instead of `public` although there are design considerations to make in choosing one modifier over the other. We'll discuss these considerations in more detail shortly.

Exercise (An Example Class: Dogs) Define a class called `Dog` in an appropriately named Java file that defines a class that represents dogs. Define your `Dog` so that it has at least three fields, a constructor, and a method. You may define whatever fields and methods for your `Dog` class that you would like. (I didn't give you any additional parameters so you can design your class in whatever context you desire). If you are at a loss for creativity, one recommendation is to define your method to be a `bark` method which returns nothing and makes the dog bark the value of its properties to the console.

2.4 Thinking with Objects

In Java, we decompose our problems not in terms of mathematical functions—a *functional* style—as in Racket, not in terms of procedures (functions with side-effects)—a *procedural* style—as in C but in terms of objects—an *object-oriented style*. At a first glance, the differences between functional or procedural and the object-oriented programming seem insignificant. We still have to think about both data and functions in either a functional or procedural style, and we frequently reason about them together. However, in Java, we unite data and methods together under the all-encompassing object construct. This simple change in code organization fundamentally changes the way that we approach program design in Java.

The classes that we identify, design, and implement in order to solve our problem become little packages of code. Ideally, these little packages satisfy a few properties:

1. They are *small and as simple as possible.* It would defeat the purpose of decomposing a problem into classes if the classes were as complex as the original problem!

2. Related, they are *limited in scope*, ideally, serving only a single distinct purpose in the overall program.
3. They function as *independent units* (as much as possible). This allows us to reason about the packages for correctness independently, either during debugging or testing.
4. They possess a *well-defined* interface with clear guarantees about inputs and outputs (when appropriate).
5. We are able to *hide* the details of the package that are unnecessary for clients of the package to know about, *i.e.*, its implementation details.

The process of bundling data and behavior together into packages that satisfy these properties is called *encapsulation*. Java allows us to accomplish this with the `class` construct. However, classes alone only allow us to bundle code; it is up to us to enforce these properties through good object-oriented design principles.

2.4.1 Abstraction

The first three properties imply that our classes should be kept small and specific in their purpose. The final two properties deal with *abstraction*, the hiding of a system's implementation through an interface. For example, consider the `Counter` class we've used as our running example so far:

```
public class Counter {
    public int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value +=1 ;
    }
}
```

The interface that a class specifies (for one of its instances) contains all of the (accessible) fields and methods of that class. For example, a `Counter` exposes:

- A way to construct an instance via a no-argument constructor,
- A `value` field that is the current value of the counter, and
- An `increment()` method to increment the counter.

These three things constitute a user's *interface* to a counter. One suspicious design decision here is that we have exposed the `value` field to the user by marking it `public` This may be undesirable because a user can set the value of a counter to any value that they want, *e.g.*,

```
Counter counter = new Counter();
counter.value = -42;
```

If we wished to restrict the counter from ever going negative or more specifically, only allow a user to change the value of the counter through `increment()` then this choice of interface is not sufficient. However,

we can't simply remove `value` from the class because it needs it to keep track of the number of calls to `increment()`! We need some mechanism to *hide* `value` from clients but still keep it around so that a counter can use it internally.

In Java, we accomplish this sort of hiding of members with *privacy modifiers*. So far, we have seen the `public` privacy modifier which makes a member visible to everyone. We fix this problem by modifying the `value` field with the `private` modifier which makes a member visible to only the class containing the member.

```
public class Counter {
    private int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value +=1 ;
    }
}
```

Now, with the `value` field marked as `private`, clients of the class can no longer access it. In particular, the code above that changes the counter's value to `-42` produces the following compiler error:

```
Counter.java:16: error: value has private access in Counter
        c.value = -42;
         ^
1 error
```

This is what we want! But there is one problem: we cannot access `value` at all! For example, if we wanted to print out `value`:

```
System.out.println(counter.value);
```

We receive the same error.

So how do we fix this problem? We create an alternative route to access `value`: a `public` method that simply returns `value`:

```
public class Counter {
    private int value;

    public Counter() {
        this.value = 0;
    }

    public void increment() {
        this.value +=1 ;
    }
```

```
    public int getValue() {
        return value;
    }
}
```

Now, we can use `getValue()` to retrieve `value` without exposing a way to change it via assignment.

```
System.out.println(counter.getValue());
```

Such a kind of method that simply returns the value of a field is so commonplace in Java that we have a special name for it: a *getter method*. If we also wanted to enable a client to change a field, we could create a corresponding *setter method*:

```
// In Counter...
public void setValue(int value) {
    this.value = value;
}
```

A getter and setter method combined provides all of the functionality that a `public` field provides but does not actually expose the field itself. So why would we want to create both a getter and setter? There may be an `invariant` of the counter—a property—that we would like to preserve, *e.g.*, that the counter should never be negative:

```
// In Counter...
public void setValue(int value) {
    if (value >= 0) {
        this.value = value;
    } else {
        throw new IllegalArgumentException();
    }
}
```

We can use `setValue` to enforce this property. If the user provides an inappropriate argument, then we signal an error with an exception. We'll discuss these mechanics and design considerations when we talk about interfaces in the next chapter.

2.4.2 Comments and Style

When designing abstractions, we rely on our type system (when available) to enforce those abstractions. Take a look at the signature of `setValue(value)` again:

```
public void setValue(int value) { /* ... */ }
```

Java enforces that we must call `setValue` with exactly one argument and that argument must be an `int`. However, the signature alone does not tell the user that they are not allowed to provide a non-negative argument. The exception we threw in the implementation signals to the user that they messed up, but at runtime. Ideally we would like to catch these sorts of errors at compile time, but we have no way of enforcing these properties with Java. Instead we must resort to documenting them with comments.

Java provides excellent facilities for commenting code: *Javadocs*. Here is an example of using the Javadoc facilities in Java:

```java
/**
 * Sets the value of the counter.  This value must be non-negative.
 *
 * @param value the new, non-negative value the counter.
 * @throws IllegalArgumentException if a non-negative value is given.
 */
public void setValue(int value) { /* ... */ }
```

Javadocs are special comments above declarations of program elements (*e.g.*, classes or methods). They start with `/**` and end with `*/`. They include special *tags* for various parts of the documentation. The most important of these for methods are:

- `@param <name> <description>`: Used to document a method parameter.
- `@return <description>`: Used to document a return value.
- `@throws <name> <description>`: Used to document an exception the method may throw.

When building rich abstractions, comments become a necessary tool to ensure people know how to use your code and what to expect from it!

Exercise (The Student Class Revisited) Fix the version of the `Student` class below so that it (a) does not expose its fields directly and (b) has appropriate Javadoc comments. Your updated class should use privacy modifiers and setter and getter methods to expose read/write access to fields as necessasry. Your Javadoc comments should contain tags for the return values and parameters of any methods or constructors that you document.

```java
public class Student {
    public String firstName;
    public String lastName;
    public int id;
    public int age;

    public Student (String firstName, String lastName, int id, int age) {
        this.firstName = firstName;
        this.lastName  = lastName;
        this.id        = id;
        this.age       = age;
    }
}
```