Chapter 6
Classes and Objects

Hello!

Today we will focus on creating classes and objects. Now that our practice problems will tend to generate multiple files, I strongly suggest you create a folder for each one, and then just add files to that folder.

Classes and Objects: The Big Idea

A type can be thought of as representing a set of values.

> `byte` is the set of values {-128,-127,...,-1,0,1,...,127}.
> `int` is the set of values { -2147483648..,-2,-1,0,1,2,... 2147483647}.
> `boolean` is the set of values {`true, false`}.

We say that the `false` value is an "element of the `boolean` type", "false is of type `boolean`", "`false` has type `boolean`", or more simply, "`false` is a `boolean`." Similarly, "13 is an `int`." If we require a value whose type is an `int`, we may say that we "need an `int`." It is always important to differentiate between the values (like 5, "hi", `true`) and the types (like `int, String, boolean`).

A programming language comes with some specific built-in types. Java has the eight primitive types (`boolean, byte, char, short, int, long, float, double`), as well as array types. Any other types (like Strings) that seem to be provided in Java are actually present in the `java.lang` package, which is implicitly imported. (Much like the 'built-in' definitions of Python).

But the world around us has a lot more structure than just numbers, booleans, and arrays or Strings. There are cars, people, traffic lights, trees, instruments, all kinds of things that are the objects involved in the system that is our real world. If we want to solve real world problems, we'd like to mimic the structure of the real world with values that exhibit the same structure.

So if we don't want to be limited to just values of those types, and we want to get any more abstraction in our data representations, how can we create new types of values? How can we group values together in a meaningful fashion, and then define how to use them?

A class definition defines a new type of values in Java. It gives us two main capabilities.

> First, it allows us to meaningfully group different types of values together to represent some more complex structure, such as a `String` and an `int` used to represent a person (for their name and age), or two `double`s representing a rectangle (for the length and width). We may call these sub-values by many names: ***instance variables*** (because each object is an instance of the class); ***attributes*** (because these are the salient parts of any object of the class); ***field members*** (a bit more of a traditional name for Java documentation).

Second, we get to define the acceptable behaviors of these values by defining methods ('***behaviors***'). It gives us the chance to not just represent the world through values, but animate that world-representation through interactions (method calls).

An object represents a specific value of a class. The object has its own value for each attribute listed in the class, and is stored in its own unique place in memory.

When we create a variable to hold an object, we actually store a reference to the object: a 'pointer' into memory that is only allowed to refer to values of a particular type.

We can create multiple objects from one class. Each object resides in its own location in memory. When we ask a particular object to run a method from its class, the object uses only its own instance variables when running the code.

Defining a Class

When we create a class, we can create various members of the class. We can declare variables, to indicate that each value of this newly defined type should have its own value for each of these variables. We can declare methods, each defining the way in which these values can behave.

Think of the methods as actions that outsiders can request the object to perform. These actions can be simple, like reporting some quantity or value; they can be complex, like simulating some process, generating new structures, or other tasks like searching/sorting through the instance variables. The statements in a method can be any procedural-style statement – calling other methods, defining variables, using control structures, and performing exception handling.

Once we've got a class definition, we can then use it as a recipe to create individual values of this new type, which we call objects. Each object is guaranteed (required) to have its own copy of all the declared variables, and thus we can call all the methods on that object.

- (This is a simplification – once we add the `static` modifier to methods or variables, and once we discuss what the `private` modifier does to methods or variables, we will also see how that qualifies the description above).

Class Example

Here is a very simple example class definition, with no methods:

```
public class Person {
      public int age;
      public String name;
      public String email;
}
```

Your Turn!

- Copy that class definition into a file named `Person.java`.

Creating Objects

We could then create Person objects like this:

```
Person p1 = new Person();
p1.age = 18
p1.name = "Billy";
p1.email = "billy99@billinc.com";

Person p2 = new Person();
p2.age = 19;
p2.name = "Sally";
p2.email = "sally.sue@masonalive.gmu.edu";

System.out.println("p1: age: "+p1.age+", name: "+p1.name+", email:"+p1.email);
System.out.println("p2: age: "+p2.age+", name: "+p2.name+", email:"+p2.email);
```

First of all, **where** should this code go? A class definition is just a definition, not a specific request for Java to perform any action. A method does nothing, but a method *call* runs the statements in the method. We learned that running a Java program is synonymous with running the main method of a class, so we want to put this code in a class's main method (or somewhere that will be reached by a class's main method).

We can just add a main method to this class, or we can create an entirely separate class, and add the code to that class's main method. We don't want to get in the habit of thinking the code using a class is required to be in just that class's main method, so let's create a separate class (in the file TestLab2.java):

```
public class TestLab2 {
      public static void main(String[] args) {
            Person p1 = new Person() ;
            …
      }
}
```

Now, when we want to run our program, we can just compile TestLab2.java, and run TestLab2, and it will automatically deal with compiling Person.java, and integrating the compiled Person.class file with the TestLab2.class code.

```
javac TestLab2.java
java TestLab2
```

Even though we don't explicitly execute `javac Person.java`, notice that the `Person.class` file has been generated. Incidentally, even if the Person class had its own main method, by executing `java TestLab2`, we are not using Person's main method – we are only executing TestLab2's main method. It is far more common to have many classes involved with only one main method, so that is the usage pattern we are trying to establish.

Your Turn!

- create a class named Player that has these instance variables in it: name, age, height, weight, position. What types will you choose for each instance variable?
- create three of your Player objects, and give them values for each instance variable.
- Print things out so that you are comfortable seeing them
- Arrays. Remember, arrays can hold anything. Make an array to hold Person values, and make it length three. Can you figure out how to put those three Person objects you just made into the array? Can you use the array to perform the printing? Especially useful may be the for-each loop.
- Create another class, Bottle. Each bottle should have state to represent how many ounces it stores, how many ounces of fluid are currently in it, whether it is insulated or not, and whether it has a lid or not.

Constructor Methods

We find that the first thing we want to do with a brand new object is to give values to all the instance variables; this is so common, that we can write a special method called a constructor to serve this exact purpose.

A constructor is a method for taking a brand new object and setting up all the instance variables, and then returning (a reference to) that object. Because a constructor must return that object, the return type is not specified. It is actually a compilation error to attempt to give a return type to a constructor. The name of a constructor method is identical to the class name, so that Java knows the method is a constructor. Just like any other method in Java, we can then require zero or more parameters.

Add the following example constructor method to the Person class:

```
public Person(int a, String nm, String e) {
	age = a;
	name = nm;
	email = e;
}
```

Before we add our own constructors, we need to point out a more common pattern of naming conventions in constructors:

```
public Person(int age, String name, String email) {
	this.age = age;
	this.name = name;
	this.email = email;
}
```

The **`this`** keyword functions as a reference to the current object running this method. Since we are creating a new object of this class type, **`this`** refers to the object being initialized. Why doesn't Java get confused between the instance variable age, and the parameter age? Usually Java chooses to disallow situations like this where there is any chance for confusion. But having to choose new parameter names for each of the instance variables is another way to introduce confusion or errors. We will discuss parameters and scope in better detail soon, so just look at the above code and know that **`this.age`** refers to the current object's age attribute, and **`age`** refers to the parameter (because the parameter is the 'closest' definition of `age`).

Your Turn!

- Add a constructor to your Player class. What parameters will you have? Will you use the this-notation or not? Do you need a return statement?
- Add a constructor to your Bottle class. Ask yourself the same questions as for the previous constructor.

What Types are Allowed?

Now that we are starting to create more structures, we might wonder if there are any limitations on what types we can use for instance variables. It turns out there are no restrictions: primitive types, arrays, other class types, we can even use the type of the class we are defining!

Your Turn!

- Add a Bottle attribute to the Player class. Hydration is important for our sport-generic athletes!

- In your Player constructor, you will now need to create the bottle. You are allowed to write things like `bottle = new Bottle(…);` inside a constructor.
- **Bonus (more difficult):** Matryoshka dolls are those "nesting dolls" that are popular in Russian folk art. Create a `Matryoshka` class, with these attributes: `canOpen, innerDoll`, and any others you might want to add. The issue of how to stop making inner dolls is tricky, and requires an understanding of recursion. The issue is solved by using the "not-a-value" value for the innermost doll: the `null` value can be stored into a reference of any class type, and means that there is no current value. Thus the purpose of the `canOpen` attribute is to serve as a safety check for whether we can access the inner doll. We could also just check `if (innerDoll==null)`.

Creating Objects (again)

We glossed over creating objects a bit. Now that we have constructors and a better understanding of classes, we can discuss a bit more the process of creating objects. Let's assume that our Bottle class has the following constructor:

```
public Bottle (int ounceMax, int ounces, boolean isInsulated, boolean hasLid) { …}
```

With this constructor, we can now create Bottle objects. The big distinction we need to make is between objects (the values in memory) and references (the variables/expressions that refer to a particular object value).

Objects versus References

When we create a new object, we are reserving a new space in memory. The **new** keyword is what actually creates the space, while the constructor method merely tells us how to set up that already-reserved chunk of memory.

```
new Bottle(32, 21, false, true)
```

We have seen the new keyword before, when we created space in memory for an array:

```
new int[10]
```

new serves the same purpose, reserving exactly enough space for this complex data structure.

So what is a reference? A **reference** is simply a "typed address" for a particular object value. It is roughly equivalent to variables for class types, though we can obtain a reference from a method call as well.

```
Bottle hydroHolder = new Bottle(32, 21, false, true)
Bottle coffeeCup = new Bottle(12, 5, true, false)
```

`hydroHolder` is a reference for the first created object (call it *32oz*), while `coffeeCup` is a reference to the second created object (call it *12oz*). But there is no restriction on a one-to-one reference-to-object relation:

```
Bottle bensBottle = hydroHolder;
Bottle jensCoffee = coffeeCup;
```

After these two extra references have been created, we still only have two objects, *32oz* and *12oz*. But `hydroHolder` and `bensBottle` both point to the *32oz* object, and `coffeeCup` and `jensCoffee` both point to the *12oz* object. When we have multiple references to the same object, we call them **aliases**. Just

as you have already learned about mutability and multiple names for the same spot of memory from CS 112, we will see that references in Java allow us to modify complex values 'from afar' by passing a reference.

Your Turn!

- Create the two objects and four references as above.
- Learn what happens when you try to print the reference itself (e.g., `System.out.println(jensCoffee)`). This is much like printing out objects in Python, where you get both a type and some sort of obfuscated memory address that uniquely identifies the location of the object (or array).
- Printing references, can you see that they are the exact same two objects despite there being four references?
- Print out all the information you can for the objects from all four references (yes, two of them will be repeats). Modify `hydroHolder` and `jensCoffee` in some way (such as `hydroHolder.ouncesMax = 42;`), and then re-print all of that information and note where your changes took place. You are seeing aliases in action!

Take away experience: Objects are the actual values in memory. References are our handles that help us find those objects. Just as many people can have your mailing address written down but there is only one house at that location, there can be many references to a single object. Keep this in mind, especially when surprising things are happening in your code.