2. Defining and Using Classes

If you do not have prior Java experience, we recommend that you work through the exercises in HW0 before reading this chapter. It will cover various syntax issues that we will not discuss in the book.

Static vs. Non-Static Methods

Static Methods

All code in Java must be part of a class (or something similar to a class, which we'll learn about later). Most code is written inside of methods. Let's consider an example:

```
public class Dog {
    public static void makeNoise() {
        System.out.println("Bark!");
    }
}
```

If we try running the `Dog` class, we'll simply get an error message:

```
$ java Dog
Error: Main method not found in class Dog, please define the main method as:
       public static void main(String[] args)
```

The `Dog` class we've defined doesn't do anything. We've simply defined something that `Dog` **can** do, namely make noise. To actually run the class, we'd either need to add a main method to the `Dog` class, as we saw in chapter 1.1. Or we could create a separate `DogLauncher` class that runs methods from the `Dog` class. For example, consider the program below:

```
public class DogLauncher {
    public static void main(String[] args) {
        Dog.makeNoise();
    }
}
```

```
$ java DogLauncher
Bark!
```

A class that uses another class is sometimes called a "client" of that class, i.e. `DogLauncher` is a client of `Dog`. Neither of the two techniques is better: Adding a main

method to `Dog` may be better in some situations, and creating a client class like `DogLauncher` may be better in others. The relative advantages of each approach will become clear as we gain additional practice throughout the course.

Instance Variables and Object Instantiation

Not all dogs are alike. Some dogs like to yap incessantly, while others bellow sonorously, bringing joy to all who hear their glorious call. Often, we write programs to mimic features of the universe we inhabit, and Java's syntax was crafted to easily allow such mimicry.

One approach to allowing us to represent the spectrum of Dogdom would be to create separate classes for each type of Dog.

```
public class TinyDog {
    public static void makeNoise() {
        System.out.println("yip yip yip yip");
    }
}

public class MalamuteDog {
    public static void makeNoise() {
        System.out.println("arooooooooooooooo!");
    }
}
```

As you should have seen in the past, classes can be instantiated, and instances can hold data. This leads to a more natural approach, where we create instances of the `Dog` class and make the behavior of the `Dog` methods contingent upon the properties of the specific `Dog`. To make this more concrete, consider the class below:

```
public class Dog {
    public int weightInPounds;

    public void makeNoise() {
        if (weightInPounds < 10) {
            System.out.println("yipyipyip!");
        } else if (weightInPounds < 30) {
            System.out.println("bark. bark.");
        } else {
            System.out.println("woof!");
        }
    }
}
```

As an example of using such a Dog, consider:

```
public class DogLauncher {
    public static void main(String[] args) {
        Dog d;
        d = new Dog();
        d.weightInPounds = 20;
        d.makeNoise();
    }
}
```

When run, this program will create a `Dog` with weight 20, and that `Dog` will soon let out a nice "bark. bark.".

Some key observations and terminology:

- An `Object` in Java is an instance of any class.
- The `Dog` class has its own variables, also known as *instance variables* or *non-static variables*. These must be declared inside the class, unlike languages like Python or Matlab, where new variables can be added at runtime.
- The method that we created in the `Dog` class did not have the `static` keyword. We call such methods *instance methods* or *non-static methods*.
- To call the `makeNoise` method, we had to first *instantiate* a `Dog` using the `new` keyword, and then make a specific `Dog` bark. In other words, we called `d.makeNoise()` instead of `Dog.makeNoise()`.
- Once an object has been instantiated, it can be *assigned* to a *declared* variable of the appropriate type, e.g. `d = new Dog();`
- Variables and methods of a class are also called *members* of a class.
- Members of a class are accessed using *dot notation*.

Constructors in Java

As you've hopefully seen before, we usually construct objects in object oriented languages using a *constructor*:

```
public class DogLauncher {
    public static void main(String[] args) {
        Dog d = new Dog(20);
        d.makeNoise();
    }
}
```

Here, the instantiation is parameterized, saving us the time and messiness of manually typing out potentially many instance variable assignments. To enable such syntax, we need only add a "constructor" to our Dog class, as shown below:

```
public class Dog {
    public int weightInPounds;

    public Dog(int w) {
        weightInPounds = w;
    }

    public void makeNoise() {
        if (weightInPounds < 10) {
            System.out.println("yipyipyip!");
        } else if (weightInPounds < 30) {
            System.out.println("bark. bark.");
        } else {
            System.out.println("woof!");
        }
    }
}
```

The constructor with signature `public Dog(int w)` will be invoked anytime that we try to create a `Dog` using the `new` keyword and a single integer parameter. For those of you coming from Python, the constructor is very similar to the `__init__` method.

Terminology Summary

Array Instantiation, Arrays of Objects

As we saw in HW0, arrays are also instantiated in Java using the new keyword. For example:

```
public class ArrayDemo {
    public static void main(String[] args) {
        /* Create an array of five integers. */
        int[] someArray = new int[5];
        someArray[0] = 3;
        someArray[1] = 4;
    }
}
```

Similarly, we can create arrays of instantiated objects in Java, e.g.

```
public class DogArrayDemo {
    public static void main(String[] args) {
        /* Create an array of two dogs. */
        Dog[] dogs = new Dog[2];
        dogs[0] = new Dog(8);
        dogs[1] = new Dog(20);

        /* Yipping will result, since dogs[0] has weight 8. */
        dogs[0].makeNoise();
    }
}
```

Observe that new is used in two different ways: Once to create an array that can hold two `Dog` objects, and twice to create each actual `Dog`.

Class Methods vs. Instance Methods

Java allows us to define two types of methods:

- Class methods, a.k.a. static methods.
- Instance methods, a.k.a. non-static methods.

Instance methods are actions that can be taken only by a specific instance of a class. Static methods are actions that are taken by the class itself. Both are useful in different circumstances. As an example of a static method, the `Math` class provides a `sqrt` method. Because it is static, we can call it as follows:

```
x = Math.sqrt(100);
```

If `sqrt` had been an instance method, we would have instead the awkward syntax below. Luckily `sqrt` is a static method so we don't have to do this in real programs.

```
Math m = new Math();
x = m.sqrt(100);
```

Sometimes, it makes sense to have a class with both instance and static methods. For example, suppose want the ability to compare two dogs. One way to do this is to add a static method for comparing Dogs.

```
public static Dog maxDog(Dog d1, Dog d2) {
    if (d1.weightInPounds > d2.weightInPounds) {
        return d1;
    }
    return d2;
}
```

This method could be invoked by, for example:

```
Dog d = new Dog(15);
Dog d2 = new Dog(100);
Dog.maxDog(d, d2);
```

Observe that we've invoked using the class name, since this method is a static method.

We could also have implemented `maxDog` as a non-static method, e.g.

```
public Dog maxDog(Dog d2) {
    if (this.weightInPounds > d2.weightInPounds) {
        return this;
    }
    return d2;
}
```

Above, we use the keyword `this` to refer to the current object. This method could be invoked, for example, with:

```
Dog d = new Dog(15);
Dog d2 = new Dog(100);
d.maxDog(d2);
```

Here, we invoke the method using a specific instance variable.

Exercise 1.2.1: What would the following method do? If you're not sure, try it out.

```
public static Dog maxDog(Dog d1, Dog d2) {
    if (weightInPounds > d2.weightInPounds) {
        return this;
    }
    return d2;
}
```

Static Variables

It is occasionally useful for classes to have static variables. These are properties inherent to the class itself, rather than the instance. For example, we might record that the scientific name (or binomen) for Dogs is "Canis familiaris":

```
public class Dog {
    public int weightInPounds;
    public static String binomen = "Canis familiaris";
    ...
}
```

Static variables should be accessed using the name of the class rather than a specific instance, e.g. you should use `Dog.binomen`, not `d.binomen`.

While Java technically allows you to access a static variable using an instance name, it is bad style, confusing, and in my opinion an error by the Java designers.

Exercise 1.2.2: Complete this exercise:

- Video: link
- Slide: link
- Solution Video: link

public static void main(String[] args)

With what we've learned so far, it's time to demystify the declaration we've been using for the main method. Breaking it into pieces, we have:

-

- `public`: So far, all of our methods start with this keyword.
- `static`: It is a static method, not associated with any particular instance.
- `void`: It has no return type.
- `main`: This is the name of the method.
- `String[] args`: This is a parameter that is passed to the main method.

Command Line Arguments

Since main is called by the Java interpreter itself rather than another Java class, it is the interpreter's job to supply these arguments. They refer usually to the command line arguments. For example, consider the program `ArgsDemo` below:

```
public class ArgsDemo {
    public static void main(String[] args) {
        System.out.println(args[0]);
    }
}
```

This program prints out the 0th command line argument, e.g.

```
$ java ArgsDemo these are command line arguments
these
```

In the example above, `args` will be an array of Strings, where the entries are {"these", "are", "command", "line", "arguments"}.

Summing Command Line Arguments

Exercise 1.2.3: Try to write a program that sums up the command line arguments, assuming they are numbers. For a solution, see the webcast or the code provided on GitHub.

Using Libraries

One of the most important skills as a programmer is knowing how to find and use existing libraries. In the glorious modern era, it is often possible to save yourself tons of work and debugging by turning to the web for help.

In this course, you're welcome to do this, with the following caveats:

-

- Do not use libraries that we do not provide.
- Cite your sources.
- Do not search for solutions for specific homework or project problems.

For example, it's fine to search for "convert String integer Java". However, it is not OK to search for "Project 2048 Berkeley".

For more on collaboration and academic honesty policy, see the course syllabus.

Previous
1.4 Exercises

Next
3. References, Recursion, and Lists

Last updated 5 months ago