CSJ61B Lecture 2

Wednesday, January 29, 2020

> **What is CSJ61B?**
> CSJ61B is my collection of lecture notes for Professor Josh Hug's version of CS61B, which in my opinion is better taught, and has many more relevant lectures. While Professor Hilfinger's version teaches you software engineering better, I do think this version of 61B teaches conceptual foundations better.

Compilation

The most basic way to compile a Java program is to type `javac <Filename>.java`. It appears as if nothing has happened, but if you see your folder, there is now a file called `<Filename>.class`. Now, you can type `java <Filename>` to run your program!

What is this `<Filename>.class` file? If we try to read it, well you will see it appears to be a bunch of senseless symbols and numbers. Well, this `class` file is the compiled, machine-readable version of your code, which is then read by the interpreter to do things.

This isn't the only way to run Java files, but it is the most common way to do this.

Why do we compile code?

`.class` has been typechecked

We don't distribute our `.java` file; we distribute our `.class` file. When the code is compiled, it is typechecked, which is to say the interpreter makes sure the correct types of data are associated with variables. This makes our distributed code safer, since you can be guaranteed it won't run into type errors.

`.class` files are machine-readable

Since our `.class` file is machine-readable natively, it removes the extra step of the user's computer having to compile the code itself, which makes the distributed code faster.

Protects your source code

This is a minor benefit, but since you aren't distributing human-readable code, it protects your code from prying eyes. There are ways to get back a rough estimation of your `.java` file, and this isn't foolproof, but it's a small benefit nonetheless.

Classes

Defining and Instantiating Classes

One of the most critical parts of a Java program is the main method. The main method is the code outside of a function or class definition in Python:

```
def x():
    ...

s = 2
j = s

"""This is where the main method in Java is, roughly"""
```

If we take away the main method and write a different method instead, then try to run that file, the code will compile just fine, but the interpreter will complain and tell you it doesn't have a main method, so it doesn't know what to do.

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

The main method tells the interpreter what running that particular Java file does:

```
    public static void main(String[] args) {
        makeNoise();
```

That doesn't mean every Java class has a main method. Instead, we might have other classes that reference a particular class without a main method. For example:

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

Every method in Java is associated with some class – there is no way to write code without a class. And to run a class, we must define a main method; without a main method, the class can only be used when referenced from another class.

Data in Classes

Our cat is fine and dandy right now, but not every cat makes the same noise. What if we wanted every cat we make to have a slightly different sound? Well, classes can not only contain methods, but also data.

Classes can be instantiated as objects: we first create a single `Cat` class, then create instances of this `Cat`. The class provides a template for all `Cats` to follow.

```
public class Cat {
    public String noise;

    public void makeNoise() {
        if noise == "meow" {
            System.out.println("That's not a cat!");
```

```
        } else {
            System.out.println(noise);
        }
    }
}
```

If we now try to run this code, the compiler will complain that "a non-static variable cannot be referenced from a static context". The easiest way for us to fix this right now is to remove the `static` part, which tells the compiler that this function can change its value based on what noise it makes.

Now if we try to run `CatLauncher`, the same error will run. What we now need to do is to run this code with some specific instance of dog:

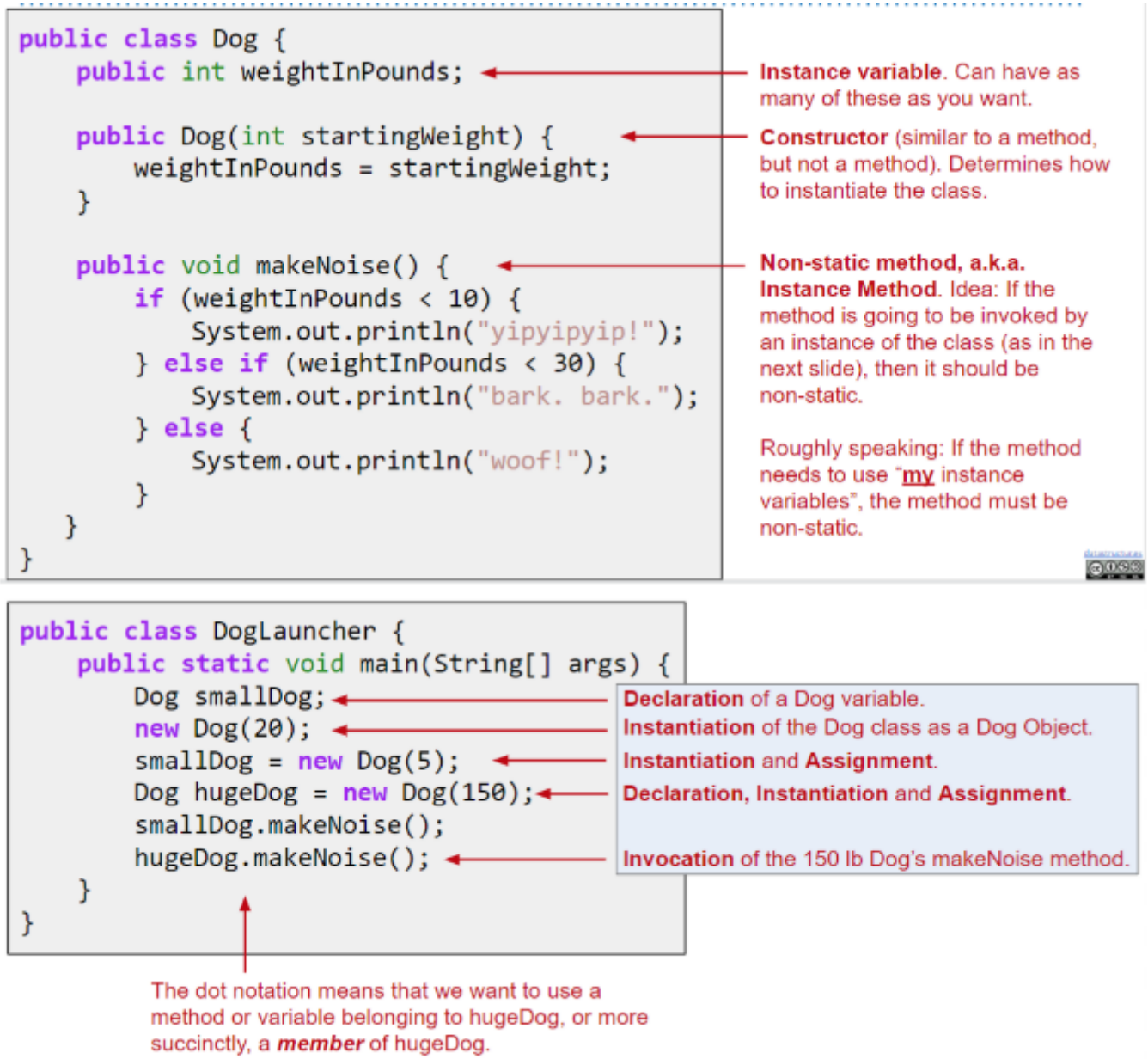
```
public class CatLauncher {
    public static void main(String[] args) {
        Cat c = new Cat();
        c.noise = "Meow!"
        c.makeNoise();
    }
}
```

This is slightly strange syntax. We can also directly pass in the value for noise when creating a Cat, which if you did 61A, was the `__init__` constructor. In Java, we can write the equivalent with:

```
public class Cat {
    public String noise;
    ...
    public Cat(String N){
        noise = N;
    }
```

And you can pass in some value of N in `CatLauncher`:

```
        Cat c = new Cat("woof");
```



Arrays

An array is like a Python list. To create an array of objects:

- First use the keyword to create the array.
- To use new again for each object you want to put in the array.

Example

```
Dog[] dogs = new Dog[2];
dogs[0] = new Dog(8);
dogs[1] = new Dog(20);
dogs[0].makeNoise();
```

You declare the size of your array with `new Dog[2]`, which creates the array and declares the spaces inside to be for dogs, but it hasn't actually created the dog yet. It names this array `dogs`, which we can access with indexing, at which point we create the dogs themselves.

Static vs. Instance Methods

We know that there are static methods, and non-static (a.k.a. instance) methods.

Static methods are invoked using the class name (`Cat.makeNoise()`), while instance methods are invoked using the instance name (`whiskers.makeNoise()`). Static methods can't access any properties specific to an instance, like the noise a particular cat has been given, because it has no bearing of `self` (or as Java calls it, `this`).

```
/** Static method example */
public static void makeNoise(){
    System.out.println("Meow!");
}

/** Non-static method example */
public void makeNoise() {
    if noise == "meow" {
        System.out.println("That's not a cat!");
    } else {
        System.out.println(noise);
    }
}
```

Why Static Methods?

From how we've described it, non-static methods are more powerful, but there are real use cases for them. Well, here's a use case: in the Math class, we simply never instantiate the Math class, and we just use the class and its static methods to do things:

```
/** This is much nicer than the alternative. */
x = Math.round(5.6);

/** Which is: */
Math m = new Math();
x = m.round(x);
```

Most of the time though, classes just have a mix of static and non-static methods, depending on different purpose. For example, if we wanted to compare two instances, we don't want any particular instance to do the comparing, but rather, the static method to be doing it.

We can also make static variables, which is a property all dogs will share. For example, in the `Cat` class, we will establish that all cats have `int legs = 4;` , which we can access with either the actual class name, or the instance name itself:

```
System.out.println(Cat.legs);
System.out.println(whiskers.legs);
```

Both these commands will print out 4, but I should note the second is considered bad style.

Example

Here's an example of a completed class, which we will use to hammer home some points about static and non-static methods:

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

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

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

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

A class can have a mix of static and non-static members. A variable or method defined in a class is also called a member of that class. Static methods are accessed using class name, and static methods can only access instance variables through a specific instance. Non-static members cannot be invoked using the class name (`Dog.makeNoise();`).

`public static void main(String[] args)`

So we now know what this means, at least in pieces. We know that:

- `public static` means we're declaring a static method
- `void` tells us it is a method that returns nothing
- `main` is the name of the function.

The real question is what is `(String[] args)` . We're not calling this main method on anything, so why does it take in an array of strings? Well, the `args` method is how we can access **command line arguments**, and we will be using the `args` name.

For example:

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

$ java ArgsDemo hello some things
hello
```

This program prints out the zero-th command line argument, as we call it. An example of a command line argument is one we saw in CS61A:

```
$ python3 ok q -06
```

`q -06` is a command line argument, and this is how the program accesses them (well, in Java, and the above example is in Python, but that's besides the point).

Exercise

Goal: Create a program `ArgsSum` that prints out the sum of the command line arguments, assuming they are numbers. Tip: Search engines are our friend!

Solution

```
public class ArgsSum {
    public static void main(String[] args){
        int N = args.length;
        int i = 0;
        int sum = 0;
        while (i < N){
            sum = sum + args[i];
            i += 1
        }
        System.out.println(sum);
    }
}
```

This program should work, but Java will scream and give an incompatible type error. The problem is that our program takes in strings, and strings cannot be added up as numbers. Well, we can very quickly fix that by doing some Googling: change `sum = sum + args[i]` to `sum = sum + Integer.parseInt(args[i])`, and we're done.

Using Libraries

One of the big developments in computer science is that pieces of code are shareable, which makes it possible for innovations to be put into use by lots of people. The technique that makes this possible is called **libraries**.

There are probably millions of Java libraries out there. In this version of 61B, all needed libraries will be provided, which will include:

- built-in Java libraries (e.g. Math, String, Map)
- The Princeton standard library (e.g. StdDraw, StdAudio, In)

As a programmer, you want to leverage existing libraries whenever possible.

- Saves you the trouble of writing code
- Existing widely used libraries will probably be less buggy
- But you also have to spend some time getting acquainted with the library.

The best ways to learn how to use an unfamiliar library include:

- Find a tutorial on the web, YouTube etc. for the library
- Read the documentation for the library (Java docs are usually pretty good!)
- Look at example code snippets that use the library

For now, don't yet use other libraries that aren't provided by course staff, as it doesn't yet work with the grader system.

Going back to the Princeton standard library, which makes various things like getting user input, reading from files, and drawing to the screen much easier.