Lecture 03

Variables, Constants, and Style

18 Prairial, Year CCXXX

Song of the day: ***White Roses*** *by Papooz (2022).*

Sections

Part 1: *Review*

The problem

In Physics, **Coulomb's Law** represents the force of magnetic attraction between two objects such as magnets, planets, and atomic particles. The formula itself looks as follows:

$$|\mathbf{F}| = k\frac{|q_1 q_2|}{d^2}$$

Figure 1: Coulomb's law. ***F*** represents force of attraction, q_1 represents the electric charge of the first object, q_2 the electric charge of the second object, d the distance between them, and K is Coulomb's constant.

Coulomb's constant is equivalent to roughly $8.988 \times 10^9\ \mathrm{N \cdot m^2 \cdot C^{-2}}$. In Java, you can represent this number as `8.988E9` (which is a lot easier to type than `8988000000.0`).

Write a **public** Java class called `CoulombAttraction` that will, in its `main()` method:

1. Print the following message: `The force of attraction is:` .
2. Print the force of attraction between the following two objects that are `3` metres apart:
    - A magnet with the electric charge of `14E-4` .
    - A wire with the electric charge of `1.42` .

Don't worry about units here. Assume that everything will just multiply / divide nicely. If you implement your program correctly, you should see the following displayed on your console:

```
The force of attraction is:
5956048.0
```

Solution

The very first thing we need to do with every Java program is to create its **class**. In this case, I specified that the class should be public, and that it should be named `CoulombAttraction` :

```
public class CoulombAttraction {

}
```

Code Block 1: Creating a public class named `CoulombAttraction` .

Again, we haven't really learned what the `public` keyword actually does yet, but for now, get used to writing it when defining **all of your classes**.

Next, to run code inside any Java class, we need to define its `main()` method. For now, we're just memorising what each of these words are, but we will learn them in due time:

```
public class CoulombAttraction {
    public static void main(String[] args) {
        // Step 1: Display the message to the user

        // Step 2: Display the result of our calculation to the user
    }
}
```

Code Block 2: Creating the main method of our `CoulombAttraction` class.

Since we haven't learned how to print calculated values and strings in the same `println()` yet, we can simply do it by using two `println()` method calls:

```
public class CoulombAttraction {
    public static void main(String[] args) {
        // Step 1: Print the message to the user
        System.out.println("The force of attraction is:");

        // Step 2: Display the result of our calculation to the user
        System.out.println((8.988E9 * 14E-4 * 1.42) / (3));
    }
}
```

Code Block 3: Writing both of our necessary `println()` methods.

Note here that I can use sets of parentheses `()` to separate the numerator from the denominator. Since Java (roughly) follows **PEMDAS** (**P**arentheses, **E**xponents, **M**ultiplication/**D**ivision, **A**ddition/**S**ubtraction) when performing mathematical operations, we can use parentheses to make sure the correct operands are being operated on by the correct operators.

Recall that Java classes just share the exact same name as the file they are contained within.

Part 2: *Saving data with variables*

I don't know about you, but if somebody showed me the following statement:

```
(8.988E9 * 14E-4 * 1.42) / (3)
```

and asked me: "what is this line trying to calculate?", the last thing I would think to tell them would be "oh, that's easy. This is clearly an execution of Coulomb's Law between two objects of $12x10^{-4}$ and 1.42 coulombs separated by a distance of 3 metres." It would be impossible, just by looking at the purely-mathematical expression, to tell what it is actually supposed to represent. Remember that programs, especially those that are used to simulate physical phenomena, are exactly just that: simulations. It is up to us, the programmer, to give each of these numbers meaning.

In other words, I would instead like to be able to write the following line:

```
(coulombsConstant * magnetCharge * wireCharge) / (distance)
```

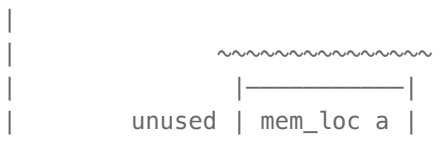
Even if you are not familiar with electromagnetism, you can gather a *lot* more information about what this expression is trying to calculate than from the previous iteration.

So, how do we get from our current program to this? Introducing: **variables**.

Technically speaking a variable represents a value store in your computer's memory. When you create a variable, you are basically telling your computer something like this:

> Hey, I want you to store the string `"Sebastián"` inside a memory address. I want you to call this memory address `myName` so I know where I can find this string if I ever need it.

In memory, this might look like this



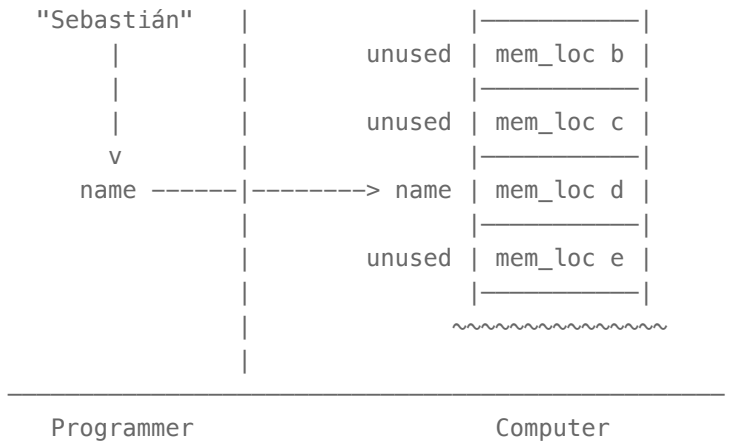


Figure 1: The memory model of creating a variable called `name`, which is storing the string value `"Sebastián"`. The identifier `name` is only for **you** to be able to easily access this value. To your computer, though, this is just memory location d (`memo_loc d`).

The way you create variables in Java is as follows:

1. You first write the **type** of the *value* that will be store inside the variable.
2. You then write the **name** of the *variable*.
3. From here, you have the choice of:
    i. Leaving your variable without a value. This is known as ***declaring a variable***. Or...
    ii. Assigning an initial value to your variable. This is known as assignment. Or...
    iii. Both i and ii.

Let's take a look at these steps in action. Numbers with decimals are called "floating-point numbers" in computer science, and you can create one in Java using the `double` keyword. For example, if we wanted to do this with Coulomb's Constant, we would do either of the following:

```
double coulombsConstant = 8.988E9;
```

Code Block 4: Creating a variable called `coulombsConstant` of type `double` and assigning an initial value of `14E-4` to it.

```
double coulombsConstant;
coulombsConstant = 8.988E9;
```

Code Block 5: Declaring a variable called `coulombsConstant` and, later, assigning it a value of `8.988E9`. Note that these are two separate operations.

In our code, it would look like this:

```
public class CoulombAttraction {
    public static void main(String[] args) {
        // Creating a variable called coulombsConstant and assigning the value 8.988E9 to it
        double coulombsConstant = 8.988E9;
```

```
        System.out.println("The force of attraction is:");
        System.out.println((coulombsConstant * 14E-4 * 1.42) / (3));
    }
}
```

Code Block 6: Our force calculations using our variable `coulombsConstant`.

If we looked at the memory model for the creation of this variable, it would look something like this:

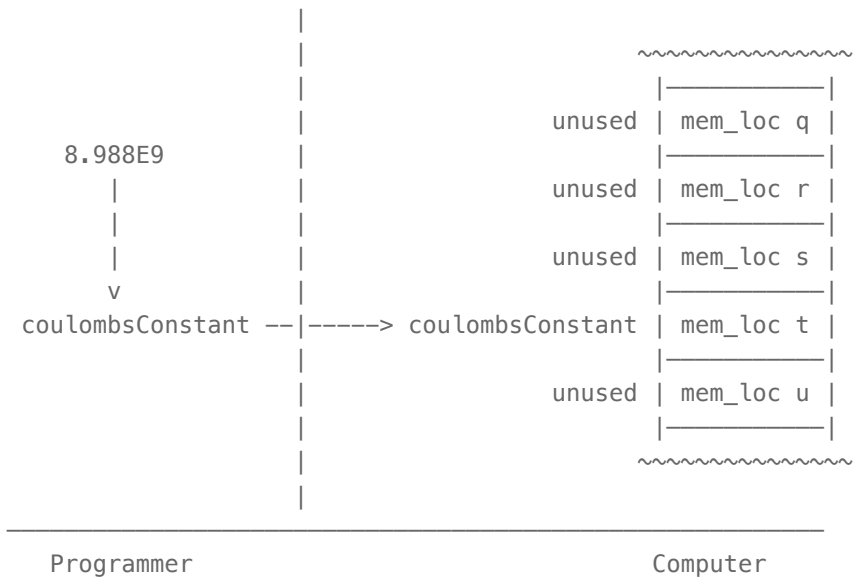


Figure 2: The memory model of creating a variable called `coulombsConstant`, which is storing the string value `8.988E9`.

In effect, unless the value of `coulombsConstant` changes throughout the course of our program, every single time we write `coulombsConstant`, we will actually be referring to `8.988E9`.

You can think of this operation as calling the building we are in "Pace". In conversation, we don't say "Yeah, sorry, I can't hang out, I've got a lecture 1 Pace Plaza, New York, NY, 10038." That would be silly, even if it's technically correct. We instead give the building out own name, like "Pace" and use that in conversation instead because it's both easier to understand to us and to the people listening to us.

So, while we're at it, let's create variables for the rest of our numbers:

```
public class CoulombAttraction {
    public static void main(String[] args) {
        double coulombsConstant = 8.988E9;
        double magnetCharge = 14E-4;
        double wireCharge = 1.42;
        int distance = 3;

        System.out.println("The force of attraction is:");
        System.out.println((coulombsConstant * magnetCharge * wireCharge) / (distance));
```

```
    }
}
```

And why stop there? Why don't we create a variable called `forceOfAttraction` that stores the **result** of our mathematical operation?

```
public class CoulombAttraction {
    public static void main(String[] args) {
        // Step 1: Define our variables
        double coulombsConstant = 8.988E9;
        double magnetCharge = 14E-4;
        double wireCharge = 1.42;
        int distance = 3;

        // Step 2: Calculate the force
        double forceOfAttraction = (coulombsConstant * magnetCharge * wireCharge) / (distance)

        // Step 3: Display the result to the user
        System.out.println("The force of attraction is:");
        System.out.println(forceOfAttraction);
    }
}
```

Code Block 7: A beautifully labelled version of our original program.

Code block 7 shows that can not only store values inside variables—we can also store the result of a certain operation. That's what makes this programming, and not just some fancy calculator. We don't have to know what the values of `coulombsConstant`, `magnetCharge`, `wireCharge`, and `distance` are; If there were no errors, Java will store the result of the operation regardless.

By the way, printing stuff in the same `println()` is super easy. All you have to do is use the `+` operator. Aside from adding two numerical values together, `+` is also called the "concatenation operator," which is just a fancy way of saying "putting stuff together operator":

```
System.out.println("The force of attraction is: " + forceOfAttraction);
```

Code Block 8: The one-line version of our `println()`.

Output:

```
The force of attraction is: 5956048.0
```

Let's talk about two very important things that we have just covered: types, and how we name our variables.

Numerical Types

Python has more than two ways (`double` and `int`) of representing numerical data. In fact, there are 6 of them, which are listed below:

Name	Range of Values	Storage Size	Used to represent a...
`byte`	-2^7 to $2^7 - 1$	8-bits	Whole number
`short`	-2^{15} to $2^{15} - 1$	16-bits	Whole number
`int`	-2^{31} to $2^{31} - 1$	32-bits	Whole number
`long`	-2^{63} to $2^{63} - 1$	64-bits	Whole number
`float`	Dependent on the computer.	32-bits	Decimal number
`double`	Dependent on the computer, but it is usually twice as precise as `float` values	64-bits	Decimal number

Figure 3: Numerical types in Java (See table 2.1 in your textbook for more information).

Now, does that mean that you have to go and memorise the range of values for each of these? Maybe; it depends. For the purposes of this class, we will always be using `int` to represent whole numbers (for consistency), and `double` to represent decimal numbers (simply because they are more precise than `float` values.

But, in the future, you may find yourself in a situation where memory management is extremely important, so you might want to use only a `short` , or maybe even a `byte` . Java was created in the 90s, so we were not yet in the era where we could take memory (mostly) for granted.

By the way, if you're curious as to what happens if you go beyond the range of a type's value, here's a quick example:

```
byte register = 128; // 2^7
        System.out.println(register + 1);
```

Output:

```
128
```

Java is literally unable to store data bigger than the memory you have allocated for it.

Part 3: *Naming Conventions*

You might, in the future, hear fellow programmers say something like "naming things is the most difficult part of computer science." Whether this is the case is subject for lengthy debate, but it is certainly something that is of supreme importance. For example, imagine that I showed you the following class:

```
public class a {
    public static void main(String[] args) {
        double x = 138.5;
        double y = 230.33;

        double z = y * y;

        System.out.println(z * x / 3);
    }
}
```

Code Block 8: A programmer's literal nightmare.

What is this program doing? More importantly: what is it *representing*? A comment I always get from students who are just starting out with programming is "but my code works fine. Why do I need to care about the names of my variables?"

Why? Because of situations like code block 8. Class `a` is actually printing the (approximate) spatial volume of the main Pyramid of Giza, but there is absolutely no way that you could have known that without me telling you beforehand. Sure, if you know the context, now you can gather that `x` is the height and that `y`, since it is being squared and stored inside `z`, represents the length of the sides of the base.

The thing is, though, programs shouldn't need context to be understood. It is easy to forget the computer science is a field that is built by humans, for humans. In the end, your computer will break everything down into ones and zeroes anyway. The reason why we even have the ability to name variables is to make the look more *human* to us. As such, Java has pretty fleshed-out conventions when it comes to naming things:

- Variables (like `distance`) and methods (like `main` and `println`) should start with lowercase characters.
    - The characters `_` and `$` are also allowed as the name's first character, but since they have their own specific use cases, don't use them for now.
    - If your identifiers are going to include more than one word (and they most often do), each new word should be denoted by an uppercase letter (e.g. `ageOfUser`). This is what is called "camel-case" naming, since it resembles a camel's humps.
- Capitalise the first letter, including its very first one, of a class name (e.g. `AnalogueSynthesiser`).
- Capitalise *every single letter* of a constant, and separate multiple words by an underscore `_` (e.g. `AMERICAN_DRINKING_AGE`, `PI`, etc.).
- Favour verbosity over brevity. In other words, be as descriptive as possible in your identifiers' names as is reasonable. For example, `MultipleVolumeNovelSeries` is a much better name than

`MultVolNovelSrs` or `MVNS`. Yes, some names will end up looking super long, but Java actually encourages this.

- Identifier names may not start with numbers.
- Do not name your classes the name of built-in classes. For example, since `System` already exists, you should not name any of your classes `System`.

Part 4: *Constants*

To quickly introduce constants, they are just what they sound like: values that do not, and should not change, throughout the duration of any program—basically the opposite of variables, which are meant to and expected to change in value.

Java actually gives us a keyword to make sure that our constants never change in value— `final`.

```
final int FRANCIUM_ATOMIC_NUMBER = 87;

FRANCIUM_ATOMIC_NUMBER = 100;
```

Error message:

Figure 4: Java won't even compile if we attempt to do this.

Applying this keyword—and the naming conventions listed above, we can now refactor code block 8 to not be a mess:

```
public class PyramidOfGizaVolumePrinter {
    public static void main(String[] args) {
        // Constants
        final double PYRAMID_OF_GIZA_HEIGHT = 138.5;
        final double PYRAMID_OF_GIZA_BASE_SIDE_LENGTH = 230.33;

        // Calculations
        double pyramidOfGizaBaseArea = PYRAMID_OF_GIZA_BASE_SIDE_LENGTH * PYRAMID_OF_GIZA_BASE
        double pyramidOfGizaVolume = pyramidOfGizaBaseArea * PYRAMID_OF_GIZA_HEIGHT / 3;

        // Display results
        System.out.println(pyramidOfGizaVolume);
    }
}
```

Code Block 9: An **infinitely better version** of code block 8. Some might find these names repetitive and annoyingly long. Indeed, other programming languages encourage slightly shorter naming conventions. But, in my opinion, the less I have to guess as to what an identifier is supposed to do or signify, the better.

In fact, we could do the same to our `CoulombAttraction` **class**, for extra practice:

```
public class CoulombAttraction {
    public static void main(String[] args) {
        // Constants
        final double COULOMBS_CONSTANT = 8.988E9;

        // Variables
        double magnetCharge = 14E-4;
        double wireCharge = 1.42;
        int distance = 3;

        // Step 2: Calculate the force
        double forceOfAttraction = (COULOMBS_CONSTANT * magnetCharge * wireCharge) / (distance

        // Step 3: Display the result to the user
        System.out.println("The force of attraction is:" + forceOfAttraction);
    }
}
```

Code Block 10: I love well-formatted Java classes.