# Unit 2 Review

Requested Review Topics:

- Incrementor order precedence (++value vs value++)
- Do-while vs while loop
- this, and when to use it
- Overloaded Constructors
- Substrings and indexOf
- File input into into ArrayLists
- Inheritance
  - Especially knowledge check on what is printed from stimulus question

## Your future in CS

I used to include this on my slides, but since these slides have changed - going to just leave it up here for every notebook. I get a lot of questions about more programming courses, the concentrations, and minors in computer science. Here is a brief reminder.

CS 165 – Next Course In Sequence, also consider CS 220 (math and stats especially)

- CO Jobs Report 2021 – 77% of *all* new jobs in Colorado require programming
- 60% of all STEM jobs requires *advanced* (200-300 level)
- 31% of all Bachelor of Arts degree titled jobs also required coding skills
- 2016 Report found on average jobs that require coding skills paid $22,000 more

- Concentrations in CS:
  - Computer science has a number of concentrations.
    - General concentration is the most flexible, and even allows students to double major or minor pretty easily.
    - Software Engineering
    - Computing Systems
    - Human Centered Computing
    - Networks and Security
    - Artificial Intelligence
    - Computer Science Education.
  - Minors:
    - Minor in Computer Science - choose your own adventure minor
    - Minor in Machine Learning - popular with stats/math, and engineering
    - Minor in Bioinformatics - Biology + Computer Science

# Incrementor

- `++value` - adds 1 to the value, and then uses the variable
- `value++` - uses the value, and then adds 1 to it.

Good to know, but can technically program without them.

For practice, Open an IDE - and create an class with a main method, and **follow** along with my example

(we will keep coming back to that main method)

What is printed (just write down the numbers of every step)?

In [6]:
```
int value = 10;
for(int i = 0; i < 10; i+=1) {
    System.out.println(String.format("The value is %d, and the incremented value is %c
}
```

```
The value is 10, and the incremented value is 11
The value is 11, and the incremented value is 12
The value is 12, and the incremented value is 13
The value is 13, and the incremented value is 14
The value is 14, and the incremented value is 15
The value is 15, and the incremented value is 16
The value is 16, and the incremented value is 17
The value is 17, and the incremented value is 18
The value is 18, and the incremented value is 19
The value is 19, and the incremented value is 20
```

> Discussion Topic
> As some folks are having trouble remembering the differences, what are some ways you can think of to help you remember the differences at your table.

## Do-While and While Loops

- **while** loops - checks condition before running
- **do while** loops - runs *once always*, and then checks the condition.

Really, it is in order of how you read it.

Follow along again:

In [8]:
```
int val = 10;

while(val < 10) {
    System.out.println("While: Does this print?");
}

do {
    System.out.println("Do-While: Does this print?");
}while(val < 10);
```

```
Do-While: Does this print?
```

## this.

`this` literally means "this instance / object", so used when you want to reference *instance* variables.

Another way to look at it, let's take the following code - how will I fix this code??

In [11]:
```
public class Student {
    private int id = 5;

    public void setID(int id) {
        id = id;
    }

    public String toString() {
        return String.format("ID:%d", id);
    }
}

Student stuie = new Student();
stuie.setID(10);
System.out.println(stuie);
```

```
ID:10
```

Let's step through each line of code looking at the memory.

```
Student stuie = new Student();
```

We see the *new* keyword, which means create a new table:

### Student@x131

| variable | value |
|---|---|
| id | 5 |

### The Stack (initial memory location)

| variable | value |
|---|---|
| stuie | Student@x131 |

Now, we call the following line of code

```
stuie.setID(10);
```

Which means, we are *inside* of stuie, giving us access to the following variables:

### .setID(10)

| variable | value |
|---|---|
| id | 10 |
| this.id | 5 |

this being the **instance** variable in the current active object.

Without this, how would java know which `id` to access? It really wouldn't, so it grabs the first one.

Using `this` helps us access the instance / object variable, to either use or set it to a different value, which is what `setID(int)` does.

## Overloaded Constructors and this()

> Discussion
>
> What is an overloaded **method**?
>
> Why are they useful?

### Overloaded Constructors

- Constructors are specialized methods whose purpose is to 'build' the object
  - They can only be called via the 'new' keyword
- Overloading a Constructor is just like overloading a method!
  - It creates optional ways to create objects.

Let's take the following "student" sometimes we want to have just an id, and other times we want a name and id.

In [14]:
```java
public class Student {
    private int id = 5;
    private String name = null;

    public Student(int id) {
        setID(id);
    }
    public Student(String name, int id) {
        this(id);  // but wait! isn't this the same code in Student(int)!!
        setName(name);
    }

    public void setID(int id) { id = id;}
    public void setName(String name) { this.name = name;}
    public String toString() {return String.format("Name:%s, ID:%d", name, id);}
}

Student stuie = new Student("Stewie", 8);
stuie.setID(10);
System.out.println(stuie);
```

```
Name:Stewie, ID:5
```

Notice the 'but wait!' comment.

Just like overloaded methods, you often want to keep your code *DRY*. However, java requires a special keyword for that

```java
this(values);
```

Notice the parans (not the same as `this.`). It is saying, use the 'constructor' of this format, to help me build the instance of this object. Let's change the code above and see!

Caveat - Java requires it to always be the *first line* of another constructor (just how they built it).

## Substring and IndexOf

- substring(start, end) - takes the substring from index start (inclusive) to index end (exclusive)
  - Since it takes int values, it pairs well with indexOf.
- indexOf(string or char) - gives you the location where that item **first** shows up or -1 if it isn't in the string
- indexOf(string or char, int) - gives you the location of the item the first time it shows up after (including) the int location for starting
  - `indexOf(string or char)` is the equivalent of `indexOf(string or char, 0)`

So let's take the following code

```java
String knocker = "tattarrattat";
String knock = knocker.substring(knocker.indexOf("r")+1,
                 knocker.indexOf("t", knocker.length()-1)+1);
```

Step 1: I would write out the string, and put the indices under it!

| t | a | t | t | a | r | r | a | t | t | a | t |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Step 2: read the code and take it in **smaller parts** (divide-conquer-glue)

- knocker.indexOf("r") + 1 - i know this starts at 0, goes to r, but then adds 1 to it
  - 5+1 = 6
  - `knocker.substring(6, ...)`
- knocker.length()-1
  - 12 - 1 = 11
  - `knocker.indexOf("t",11)+1)`
    - 11 + 1 = 12
    - yes, just a way to grab the last T
- `knocker.substring(6,12);`
- I then remind myself inclusive, exclusive

| r | a | t | t | a | t |
|---|---|---|---|---|---|
| 6 | 7 | 8 | 9 | 10 | 11 |

The last part, I would often do just by circling or underlining.

### IMPORTANT!

Write it out with index numbers. **REALLY** easy to get an OB1 error without it.

```
In [16]:
// your turn - at the table, work it out with the same steps above!

String plant = "kinnikinnick";
String p2 = plant.substring(plant.indexOf("k"),
            plant.indexOf("k", plant.indexOf("i"))+1);

System.out.println(p2);
```

```
kinnik
```

# File Input:

File input into ArrayList?

## Small but very important difference!

File input is into **Strings** (for now). We *then* take those strings, and use them how we want - often putting them in ArrayLists.

Let's look at code, and build it up!

```
In [20]:
import java.util.ArrayList;
import java.io.File;
import java.io.IOException;

public ArrayList<String> readFileIntoArrayList(String filename) {
    ArrayList<String> lines = new ArrayList<>();

    try {
        Scanner in = new Scanner(new File(filename));

    }catch(IOException ex) {
        ex.printStackTrace();
    }
    return lines;
}


ArrayList<String> lines = readFileIntoArrayList("data/students.csv");
```

# Inheritance

We will switch over to a knowledge check for this one