Interfaces

What are interfaces?

A **interface** is a list of methods. Any class that `implements` an interface **pinky promises to provide code for all methods in the interface**.

Interfaces will:

- Provide method signatures
- Do not provide any implementation
- Contain no instance variables or static methods.

Using interfaces

An example:

```
interface Countable {
        int getCount(); //Promising the use of getCount()
} //End Countable Interface
```

And to `implement`:

```
public class StudentPopulation implements Countable {
        int population = 7000;

        public getCount() {
                return this.population;
        } //End getCount()
} //End Student Population
```

Why use interfaces

Should use to implement general cases. In the above example, I could implement the `Countable` interface for *anything* that is countable. This lets user know that, since i've implemented `Countable`, they should be able to use `getCount()`. Above, I used the `Countable` interface for `StudentPopulation()`, but I can also use it for `CatPopulation()`, and `BatmanPopulation()`. They dont necessarly relate, but the user will know that `getCount()` method will be available because they all implement `Countable`.

Abstract Classes

What are Abstract Classes

Abstract classes may contain **abstract methods** and **instance variables**. They **cannot be instantiated** (Like `Human batman = new Human()`). Ho wever, they can be **subclassed**. They may have signatures of methods we promise to define, or pre-defined methods we can override.

Its main purpose is **to be extended by subclasses**:

```
public abstract class Superhero {

        private String company;

        //All superheroes (unfortunately) are owned by a company
        public Superhero(String company) {
                this.company = company;
        } //End Superhero()

        //All superheros must have superpowers
        abstract void superPowers();
```

```
12
13          //All superheroes must have a cool backstory, so they gotta tell people
14          abstract void tellBackStory();
15
16          //All superheroes have a sense for justice
17          void justiceSpeech() {
18                  System.out.println("Being bad is bad!");
19          } //End justiceSpeech()
20
21  } //End Superhero()
```

Now that we have a abstract class, let's have two examples classes that `extend` this superclass: `Superman()` and `Batman()`

```
1   public class Batman extends Superhero {
2
3           //The abstract class has a constructor, so we must have a constructor,
4   too
5           public Baman(String company) {
6                   super(company);
7           } //End constructor
8
9           //An abstract method means we have to define it
10          public void superPowers() {
11                  System.out.println("I can fight with bat tools!");
12          } //End superPowers()
13
14          pubic void tellBackStory() {
15                  System.out.println("I swear revenge for my parents!");
16          } //End tellBackStory()
17
18          //justiceSpeech() was already defined in the abstract class
19
    } //End Batman()
```

```
1   public class Superman extends Superhero {
2
3           //Constructor
4           public Superman(String company) {
5                   super(company);
6           } //End constructor
7
8           public void superPowers() {
9                   System.out.println("I can fly!!");
10          } //End superPowers()
11
12          public void tellBackStory() {
13                  System.out.println("I come from another planet");
14          } //End tellBackStory()
15
16          //This time, we want to overrice justiceSpeech()
17          public void justiceSpeech() {
18                  System.out.println("I protect people!");
19          } //End justiceSpeech()
20
21  } //End Superman()
```

Why use Abstract Classes

We should use abstract classes when we are making a class that is logically a subclass of the abstract class, and are related and share values in the same way. In this case, since Supermand and Batman are both superheroes and have their own superpowers and back stories, we can use the `Superhero` class.

Abstract Classes & Interfaces

The surface similarity between these two are that abstract classes and interfaces are both used to declare signatures you're going to use.

In Abstract classes:

- Can extend a class.
- Can define public, private, or protected methods/variables.
- Classes can only designate one superclass

In Interfaces:

- All fields are automatically public, static, and final.
- We can implement more than one interface per class. (as opposed to only one superclass)

When to use which type

Interfaces should be used **as a rule**. If I was working at a game company, they'd probably tell me all game characters **must** `implement` the `Playable` interface, because thats **what the character needs to be functional in the video game**. If we were making Batman, he would have to `implement Playable`. In addition to this, Batman can also use a class that the game company has provided called `Superhero()` which provides most of the functions that someone who's making Batman would want to use. Abstract classes should be used **To help categorize and flesh out a class**.