Inheritance

What is inheritance?

Inheritance is when you "inherit" or **take code from another class** through the `extends` code. In summary, inheritance allows us to re-use code, saving us the time of coding it again or the messiness of copy/pasting it. For example:

```
public class Batman extends Human {
        public void fly() {
                //The code to flying is here
                //But that's closed source
                //So I can't show you
                //(haha get it??)
        } //End fly()
} //End Batman()
```

In this case, `Batman` is the **subclass** or **child class**, inheriting all *public and protected* methods and instance variables from the **superclass** or the **parent class**, `Human`. In addition to everything `Human` can do, `Batman` can run the `fly()` method.

NOTE: Although Batman is a superhero, he is a SUBCLASS of the SUPERCLASS Human. This is because Batman can be categorized as a human, but not all humans can be categorized as Batmen.

Subsitution Principle

Anywhere you use a class, you may also use a subclass that `extends` the superclass. This is called the **subsitution principle**. Here, I'm recruiting people to join my gang of human crime-fighters:

```
Human[] inMyGang = new Human[5];
inMyGang[0] = new Batman();
inMyGang[1] = new Human();
```

In this I have an array of people I have recruited to my gang that only allows `Human` people (Sorry Yoshi, you're still my favorite dinosaur, though). `Batman` can be in here because he's a subclass of `Human` (Because he's technically a `Human`).

Overriding

We can replace methods in our original class, too. This is called **overriding** a method:

```
public class Batman extends Human {
        public void fight() {
                //normally, all humans can fight
                //but this (also closed source) code
                //allows Batman to beat people up better than the average human
        } //End fight()
} //End Batman()
```

Every human can `fight()`, but Batman has a better `fight()` because we modified, or overrided the `fight()` method in this block of code.

Super

Nobody knows Batman's identity. That's because when he goes to work and goes about his daily life, he needs to blend in. However, most of his methods are overridden to replace them with superhero methods. The solution to this is to use the `super` command. **Super will access the non-overridden command of a superclass**. So, to dress up like an average joe and go to work like one, he can use `super.dressUp()` and `super.goToWork()`. If he used his normal `dressUp()`, he'd dress up in his Batman suit. If he used his normal `goToWork()`, he would be looking for criminals to fight. This is how the Batman blends in and avoids mixing his two lives with eachother: he uses the `super` to memorize and execute normal, civilian, human methods. There is another way to use the `super` class: you can **call the superclass's constructor**, as we will discuss in the constructor section involving the creation of an enemy of Batman: the street thug.

Overloading

Overloading is when we use one method for two operations. Here, we encounter the class `Thug()`, who try to terrorize humans. Batman swoops in to save these civilians, but thugs won't let him save the day that easily! Thugs that are a subclass of humans have two methods to try to get Batman off their back:

```
public class Thug extends Human {
        public void negotiate(int money) {
                //Let your money do the talking.
        } //End negotiate (1/2)

        public void negotiate(String Threat) {
                //Threaten Batman with a mean string instead of bribing him.
        } //End negotiate (2/2)
} //End Thug()
```

Java will know which method to use by what you put in (an `int` or `String`).

Constructors

We can give names to people using a constructor. Say our human class constructor looked like this:

```
public class Human() {
        int willToHarm;

        public Human(int willToHarm) {
                this.willToHarm = willToHarm;
        } //End constructor
} //End Human()
```

This will set the Thug's will to harm variable equal to the humans will to harm. However, thugs usually have more will to harm people than the average human:

```
public class Thug extends Human {
        public Thug() {
                willToHarm = 100;
        } //End default constructor (1/2)

        public Thug(int willToHarm) {
                super(willToHarm);
        } //End constructor (2/2)
} //End Thug()
```

In this example, if we created a thug with just `new Thug()`, it would call on the **default constructor**, which would make a thug with a 100 will to harm rating. He's a petty thief at most, stealing from old women. However, say we made a thug like this:

```
Thug goKill = new Thug(9001);
```

This would create a thug named `goKill`, whose will to harm, according to our scouters, is over 9000!! The `super(willToHarm)` part of the code means that it will call the **superclass's constructor**, and give it the argument that we give it (in this case, we gave it 9001). This is another way to use the `super` accessor.

Static

Static variables are variables that will be created at the start of the program, and will **Be the same for every instance of the class made**. One way to illustrate this is giving `Human()` a `population` variable. This variable, when changed, will be the same for Batman, Thugs, or Humans. **Static methods** are a lot more complicated, and don't worry if you do not understand them at first. Static methods are methods that do not need to have an object created to call them. They also cannot access any variables that are not static. Unfortunately, I can't think of a way to tie in static methods into our little thug, Batman, and human story. If you would like to do a bit of reading on your own time, a nice explaination is located at http://www.leepoint.net/notes-java/flow/methods/50static-methods.html