Don't Fear the OOP!

-or-

A java tutorial that shows you why Coding Java
(or any other object-oriented programming)
is just like writing a trashy Western novel.

-or-

How to understand Java by looking at pretty colors.

The analogy of this tutorial is simple: think of a java programmer as a writer, composing a stock novel. All of the characters and settings are "off-the-shelf", and need be only modified slightly to fit into a new book. All that's left to write a bestseller is to come up with a plot that pulls all those pre-existing elements together.

That, in a nutshell, is java programming. Think of it as Dean Koontz for smart people. Now, that might be all you want to know. If so, thanks for stopping by! If things still could use some clearing up (perhaps by way of a couple dozen pages of examples), then read on!

When I first started learning how to program Java, I was left totally confused about this whole "object-oriented" thing. What books I had explained the concept poorly, and then went straight on to advanced programming tips. I felt frustrated and lost. Not being particularly math-oriented, I needed a good analogy to help me understand the nature of Java.

I have created this brief tutorial not in order to be an exhaustive Java resource, but rather to introduce readers to the concepts of object oriented programming in a way that is non-threatening. If all goes well, we'll have you all in pocket protectors before the end of the hour.

Don't Fear the OOP!

There are three different levels of this tutorial, coded by color. Green is for those readers who want the most basic introduction. It is targeted at those who are unsure what object-oriented programming is, and could use a good analogy to make things clearer. Yellow is for those who want to be able to understand object-oriented programming just enough to be able to read and follow it, but are not yet ready to learn the intricacies of coding Java. And finally, the third level, red, is for you daredevils who want to be able to program in Java, but just want to ease into it slowly.

In short, the green text gives a "plain English" version of the code that would be necessary, the yellow uses that English in a way that more closely resembles the format of code, and the red is the actual code that would be necessary for the program to work. Readers of all levels are encouraged to skip between the colors to deepen their understanding. Finally, although this tutorial operates mostly through analogy, innuendo, and intrigue, those words that appear in **boldface** are the *actual terms used by Java programmers* (ooooh!), so try to remember them as you go along.

Meet Eunice. Eunice writes Westerns. Her books about life in the American Wild West have become so popular that everybody rushes to buy them. Her readers are never disappointed, because Eunice's books are very consistent. How does she manage this?

Well, before Eunice writes anything, she sits down at her desk, sweeps aside the clutter, and places a single piece of paper down on her desk. It is on this one page that she writes her whole story.

How can Eunice get the whole story onto one page? Simple. Everything she writes on the page is just a sequence of events, or plot. Setting, character development, and how her characters interact is all taken care of elsewhere. How does she do that? Let's find out. You see, Eunice, more than anything else, is orderly and consistent. It's what has made her books so popular. Let's say that she has just sat down to write "Gunfight at Old West Saloon", her thirty-fifth book.

The first thing that Eunice needs is a setting. So she pulls down a binder she has marked "Western Towns", opens to the first page, and reads the first few lines:

Don't Fear the OOP!

Every Western town has a few key ingredients: stables, saloons, sheriffs, and a couple troublemakers. A standard town would have three stables and be located West of the Mississippi sometime around 1850.

This description of a Western town, while not very profound, did two important things: 1) It established the key ingredients for a Western Town and 2) It gave several values that might occur in a default Western Town. (Any more than three stables and the place starts to stink)

WesternTown
- has a certain number of stables
- has a certain number of saloons
- has a certain number of sheriffs
- has a certain number of troublemakers
- is located somewhere
- exists at a certain time

a typical WesternTown would have
- number of stables = 3
- location = Western America
- time period = 1850

It may seem that Eunice has a strange way of writing, but to her, it's straightforward. The first line on the page states that Eunice is defining a Western town. You may have noticed that when Eunice needs to write two words, she won't use a space. There is a reason for this. If she puts spaces in, her editor in New York, a stickler for details, gets confused.

The next step, logically, is to then declare what sorts of **variables** (those things which define how the town looks) that a **class** of Western towns might have. Sheriffs, stables, troublemakers, etc. That being said, all that was left to do was to **construct** a sample town, with default values for her variables. Whew!

```
public class WesternTown
{
	int stables;
	int saloons;
	int sheriffs;
	int troublemakers;
	String location;
	int time;


	public WesternTown()
	{
		stables = 3;
		location = "Western America";
		time = 1850;
	}
}
```

In making her Western Town, Eunice is defining a **class**, not the town itself (that would be an **object**). A **class** is like a (really mixing metaphors here) recipe without any measurements. It says what elements should be in a Western Town, but does not say in what amounts.

This is the process of **declaring** what the **variables** will be called, and what sort of values they will contain. For now, just worry about "int"s and "String"s. "**int**" stands for integer, and "**String**" for a string of letters.

The semi-colon in Eunice's way of letting her editor know that she is done with the preceding statement and that he can move on to the next one. For the most part, you can think of it like a period.

Finally, you'll notice that whenever Eunice wants to group a set of statements, she uses a curly brace, "{". Even though Eunice knows her characters very well, this lets her editor see where a group of statements begins and ends. Any **class** will always end with all the open curly braces being closed.

If Eunice does create (**instantiate**) a town **object** eventually in her book, her editor will look to the same Western Towns **class** to determine what that town object should look like (Eunice will send the binders along with her manuscript). But where will her editor find out the number of stables and such? This is where we get into the constructor.

The constructor comes after all of the variables have been declared. It starts with public WesternTown(). The statements that follow are the default values for her variables; how Eunice's town **object** would look if she simply **instantiated** it in her plot without specifying any of its values.

So now we've sneaked a peak at one of Eunice's binders. She has reviewed the Western Town section and thinks that this western should take place in, not surprisingly, a Western Town. So she's ready to write the first part of her plot. Eunice turns to the **main** paper that is sitting in the middle of her desk and starts to write the story.

The Gunfight at the Old West Saloon
This story takes place in a Western town called Sweaty Post. Sweaty Post has one sheriff, two saloons, and five troublemakers.

Just a couple lines, but Eunice has, in fact, said quite a bit. When she sends the story to her editor, he will look up "Western Town" in the binders she sends along and fill in the date, location, and number of stables. In addition, Eunice has added information about the sheriff, saloons and troublemakers. But that's okay, because her editor was expecting it. Because her binder stated that every western town would have a certain number of these, but didn't say how many, her editor was quite happy to see them specified in her plot. Even though her main plot page only has two lines, because of the reference to her binder, her story already contains quite a bit of information.

Main Gunfight

Western Town sweatyPost is a new Western Town.
the number of saloons in sweatyPost is two.
the number of sheriffs is one.
the number of troublemakers is five.

Eunice takes two very important steps early on in her plot. First, she **instantiates an object** of type WesternTown. Having created her first **object**, Eunice then goes on to fill out the information that was initially lacking in the WesternTown **class**: (number of saloons, sheriffs, troublemakers)

```
public class Gunfight {

	public static void main (String arguments[]) {
		WesternTown sweatyPost = new WesternTown;
		sweatyPost.saloons = 2;
		sweatyPost.sheriffs = 1;
		sweatyPost.troublemakers = 5;
	}
}
```

Eunice sure has a strange way of saying things, doesn't she? It would be possible to explain what each word in those first two lines does, but it wouldn't make much sense to you at this point, and it would really tax my small brain, so let's skip it for now, eh? What's important to know is that that line is how Eunice let's her editor know that this piece of paper is her **main routine**, or plot.

When she decides to specify those **variables** (saloons, etc) that were not specified in the WesternTown **class**, Eunice first states the name of the **object** (in this case, sweatyPost. Remember that sweatyPost is an object of WesternTown) followed by a period and then by the name of the **variable**. Having done this, Eunice can then provide a value for the variable, such as two.

So, believe it or not, you've just seen object-oriented programming in action. Our author Eunice first created a class (that was the binder) that roughly described a Western Town, then turned to her **main** plot page and created (**instantiated**) a western town **object**, which she then called sweatyPost.

Now, for her villain, Eunice needs a different binder. So she puts "Western Towns" back on the shelf, and goes to pick up a new binder. Her hand passes over many different titles, like "Animals", "Weapons", "Weather", "Music", all the essentials for a good Western, until she finally comes to the binder she wants: "Humans".
She opens up to the introduction:

> "All humans start out with two legs, two arms, eyes, a nose, and a mouth. They are either male or female, have a name, have a horse with a name, and have different preferences in whiskey. If someone asks, humans can respond with their name."

You'll notice that Eunice likes to make all body parts **variables**. This gives her the freedom to change their values later in the story. It is her well-known propensity for gore that has drawn so many readers to Eunice's particular brand of Western. You'll note that she left the sex, name, horse name, and whiskey preference unspecified. Just so all of her humans don't look identical, Eunice prefers to specify those **variables** when she writes the plot.

```
Humans
        have a certain number of legs
        have a certain number of arms
        have a certain number of eyes
        have a certain number of noses
        have a certain number of mouths
        have a name
        have a certain sex
        have a horse with a name
        have a strong preference in whiskey

A standard human would start with
        two legs
        two arms
        two eyes
        one nose
        one mouth

When someone asks for your name
        tell them your name
```

You, being as sharp as you are, notice that Eunice's human has some interactivity. She can ask its name, and it will respond. This is called a **method** and is your key to a good time. We'll get to it in just a few pages.

```
public class Humans {
	int legs;
	int arms;
	int eyes;
	int nose;
	int mouth;
	String name;
	String sex;
	String horseName;
	String whiskeyPreference;

	public Humans() {
		legs = 2;
		arms = 2;
		eyes = 2;
		nose = 1;
		mouth = 1;
	}

	public String whatIsYourName() {
		return name;
	}
}
```

Boy, it all makes sense except that bit about that "public String whatIsYourName()" business, eh? Don't worry. We'll get to all that in just a little bit.

As interesting as these descriptions of humans may be, however, they're not very specific. Eunice promised her editor a new story by Friday, so she decides to get down to business. Flipping through the "Humans" binder, she comes to the first chapter, entitled "Villains"

Villains are based on the idea of humans. They are identical, except that they have some additional qualities, namely a mustache, a hat, a certain "look", some level of drunkenness, and a certain quantity of damsels in their possession. Your standard villain will look mean, start the day out sober, and not yet have captured any damsels.

Nothing really new here, except that we have peered deeper into Eunice's binders (**classes**) to see one of the subsections (**subclasses**). This particular subsection, villains, **extended** the idea of humans.

Villain extends the idea of Humans.

A Villain
- has a mustache.
- has a hat.
- has a "look".
- has some level of drunkenness.
- has a certain number of damsels tied up.

For a given Villain,
- He will look mean.
- He will start out sober.
- He will start the day without having any damsels tied up.

```
public class Villain extends Humans {
	String mustacheColor;
	String hatColor;
	String look;
	int drunkenness;
	int numberOfDamsels;
	Humans damsel;

	public Villain() {
		look = "Mean";
		drunkedness = 0;
		numberOfDamsels = 0;
	}
}
```

We have introduced here the idea of subclasses. "Humans"; was a class, and "Villains" a subclass of it.

You may also be wondering why Eunice declared "Humans damsel" in this class. Think of it this way. If the villain is going to tie up a damsel, Eunice's editor has to know what a damsel is. By declaring "Humans damsel", her editor will know that a damsel is a type of human.

After Eunice had described what her villain looked like, she decides to move on to some of the **methods** (remember those from a few pages ago?) that villains employ to achieve their dastardly deeds. Being a teetotaler, Eunice wants to focus on how villains drink. So she writes:

```
Whenever the main plot says that a villain drinks whiskey, his level of drunkenness will
go up by one.
```

While this may not look any different than any of the things that we were doing before, you should note that Eunice is specifying how one of her characters acts, rather than just how he/she/it looks. You should also note that she is altering one of her variables here, the variable "drunkenness". Since her binders (**classes**) can contain information about how her characters act, Eunice can create quite a bit of character development without ever even touching her main plot page.

```
drinkWhiskey
        drunkenness increases by one
```

Notice the pattern than has begun to emerge in Eunice's writing. What she has done with this **method** is similar to how she treated variables in the past. First, she notes what she is going to describe (in this case a **method** called drinkWhiskey) and then, on the next line, what that method will do when she **calls** it in her **main** plot (**routine**).

```
public void drinkWhiskey() {
        drunkenness ++;
}
```

Although Eunice used quite a bit of strange symbols, her approach was straightforward and consistent. The first thing that she did was to name the method "drinkWhiskey". Since the method name is drinkWhiskey, Eunice figures that she should modify the villain's level of drunkedness, in this case by adding one to it. That is what the "++" after "drunkedness" does.

Now, the villains in Eunice's stories are famous for being able to hold their liquor. Hence, it's very difficult for an onlooker to gauge how drunk one of her villains really is. Eunice decides that it's a good idea to allow her villain to state how drunk he is, so, she writes a new method:

If someone asks a villain how drunk he is, the villain will always respond with his level of drunkenness.

```
howDrunkAmI
        tell them how drunk I am
```

```
public int howDrunkAmI() {
        return drunkenness;
}
```

If you're wondering about that "int" in front of the "howDrunkAmI()", Eunice had to put that in because this **method** has a **return value**. In order to let the editor know what sort of value was being returned, she specified "int" (for integer) in front of the name of the method. Remember, her editor doesn't like surprises, and if she didn't tell him what sort of variable this method was going to return, he'd get all hot and flustered. (Now, we don't want that, do we?)

At this point Eunice is pretty proud of herself, having created a villain that can experience pretty much the full range of villainous activities. However, just to make for a good closing, Eunice decides to have the villain have the ability to tie up a damsel. She sets out to write one more method. What makes this different than her other methods is that it needs information about someone other that the villain, namely the damsel to be tied up. To allow for flexibility (and for a variety of damsels), Eunice decides to leave the identity of the damsel blank for now.

If the villain is supposed to tie up a damsel, tie up the specified damsel, then add one to the number of damsels he has tied up. Then print out "Oh my gosh! (the specified damsel) has been tied up!"

So now you can see the flexibility that Eunice has in writing her stories; she can leave certain things to be specified only when the plot is written. And by writing that the damsel will have a name, but not stating what it will be, that's just what she has done. She has also used a print statement that will print out the information gathered by the method as an event in her book. How easy!

```
tieUpDamsel (name)
        add one to the number of damsels this villain has tied up.
        print "Oh my Gosh! (the specified damsel) has been tied up!
```

I have nothing to say to you people. If you feel bad about understanding everything so far, then you can read the red section for this one.

```
public void tieUpDamsel (Humans damsel) {
        this.damsel = damsel;
        numberOfDamsels++;
        System.out.println("The Villain has tied up " + damsel.whatIsYourName());
}
```

"Geez! I should have stayed with the green type," you're probably saying to yourself. Don't worry. It looks much worse than it is, and you probably understand most of it already. First, we have the name of the method: "tieUpDamsel". No surprise there. Then, between those parentheses, we have "Humans damsel". What comes between those parentheses is called an argument. It helps make the method more specific. When Eunice starts writing her plot, she will at some point want her villain to tie up a damsel (what sort of Western would it be without it?). But if she just writes in her plot, "villain.tieUpDamsel()", that wouldn't be very exciting for the reader. Who has been tied up? What is her name? Priding herself on always having strong female characters, Eunice wants to add the name of the damsel who is being tied up. Hence, when her plot says "tieUpDamsel", it will also say the name of the damsel being tied up as an argument. "Humans", which comes before "damsel", tells her editor that the thing being tied up is a human. Her editor doesn't like surprises, so that knowing that it is a human being tied up, rather than say, a sheep, reduces the amount of guesswork on his end.

The next line is another attempt by Eunice to please her rather persnickety editor, who is even more picky than she is! Even though she has supplied the name of the damsel as an argument, her editor won't let her use it in the method until she has stated that the "damsel" that she declared as a "Villains" variable is equal to the value given in the method's argument.

Having finally jumped through all of her editor's hoops (don't expect to understand all of that right away, it took Eunice a couple of weeks!), Eunice gets down to finishing the rest of her method. Once the villain has tied up a damsel, the variable "tiedUpDamsels" will surely go up by one, so Eunice puts the same "++" after tiedUpDamsels that she had put after "drunkenness" in the "drinkWhiskey" example.

The stage being set, Eunice decides that it is time to let her readers know what is going on. Hence, the "System.out.println" statement. A bit wordy, but it let's her editor know that what comes between those parentheses goes into the final text of the book: all the text she puts in between quotation marks. Why is the variable 'damsel.whatIsYourName' then not in quotation marks? Because it is not a literal, but rather an **object's method**. By keeping it out of the quotation marks, Eunice lets her editor know that he has to go look for the damsel object (he would find out that it was of the "Humans" class) then for the method "whatIsYourName" (he would see that it returned the damsel's name). All of that for a damsel's name!

Had she put damsel.whatIsYourName inside of the quotation marks, her editor would have printed out in the final copy of the book : "The villain has tied up + damsel.whatIsYourName!". No Pulitzers for that. In order for her editor to substitute the variable name for the word name, Eunice closes the quotation marks before she writes it. The plus "+" lets her editor know that she wants to join the two statements.

Lucky for you, the next page is just review.

Don't Fear the OOP!

At this point, it might be a good idea to review what Eunice's overall description of a villain looks like. None of this material is new, just a compilation of what we've gone over in the past few pages.

Villains are based on the idea of humans. They are identical, except that they have some additional qualities, namely a mustache, a hat, a certain "look", some level of drunkenness, and a certain quantity of damsels in their possession. Your standard villain will look mean, start the day out sober, and not yet have captured any damsels.

Whenever the main plot says that a villain drinks whiskey, his level of drunkedness will go up by one.

If someone asks a villain how drunk he is, the villain will always respond with his level of drunkenness.

If the villain is supposed to tie up a damsel, tie up the specified damsel, then add one to the number of damsels he has tied up. Then print out "Oh my gosh! (the specified damsel) has been tied up!"

Villain extends the idea of Humans.
    Every villain has a mustache.
    Every villain has a hat.
    Every villain has a "look".
    Every villain will have some level of drunkenness.
    Every villain will tie up a certain number of damsels.

For a given Villain,
    He will look mean.
    He will start out sober.
    He will start the day without having any damsels tied up.

drinkWhiskey
    drunkedness increases by one

howDrunkAmI
    tell them how drunk I am

tieUpDamsel (name)
    add one to the number of damsels this villain has tied up.
    print "Oh my Gosh! (the specified damsel) has been tied up!"

```
public class Villains extends Humans {
	String mustacheColor;
	String hatColor;
	String look;
	int drunkenness;
	int numberOfDamsels;
	Humans damsel;

	public Villains() {
		look = "Mean";
		drunkenness = 0;
		numberOfDamsels = 0;
	}

	public void drinkWhiskey() {
		drunkenness ++;
	}

	public int howDrunkAmI() {
		return drunkenness;
	}

	public void tieUpDamsel (Humans damsel) {
		this.damsel = damsel;
		numberOfDamsels++;
		System.out.println("The Villain has tied up " +
damsel.whatIsYourName());
	}
}
```

Don't Fear the OOP!

All this work with damsels and trains is making Eunice feel a bit run down, so she decides to stop working on her characters for a little while, and to work a bit on the main plot.

Accordingly, she sets her pen to the sheet of paper labeled **main routine**.

Here is the main plot of the Gunfight at the Old West Saloon: There is a Western Town called Sweaty Post. Sweaty Post has two saloons, one sheriff, and five troublemakers. There is a male villain named Maurice. Maurice has a black hat, a red mustache, and a horse named "Beer Gut". Maurice prefers Jack Daniels whiskey. Mary is a female human. She has a horse named "Midnight" and she prefers her whiskey straight. In our story, Maurice starts out by drinking whiskey. He then lets everyone know how drunk he is, and then ties up a woman named Mary.

```
Here is the main plot of Gunfight at the Old West Saloon;
In the novel Gunfight;

There is a town called sweatyPost;
        sweatyPost has two saloons;
        sweatyPost has one sheriff;
        sweatyPost has five troublemakers;

There is a new villain named maurice;
        maurice has a black hat;
        maurice has a red mustache;
        maurice is male;
        maurice has a horse named "Beer Gut";
        maurice prefers Jack Daniels whiskey;

There is a new Human named Mary;
        mary is female;
        mary has a horse named "Midnight";
        mary prefers her whiskey straight;

maurice drinks some whiskey;
maurice tells us how drunk he is;
maurice ties up a damsel named Mary
```

```
public class Gunfight {
        public static void main(String arguments[]) {

                WesternTown sweatyPost = new WesternTown();
                        sweatyPost.saloons = 2;
                        sweatyPost.sheriffs = 1;
                        sweatyPost.troublemakers = 5;

                Villains maurice = new Villains();
                        maurice.hatColor = "black";
                        maurice.mustacheColor = "red";
                        maurice.sex = "Male";
                        maurice.horseName = "Beer Gut";
                        maurice.whiskeyPreference = "Jack Daniels";

                Humans mary = new Humans();
                        mary.sex = "female";
                        mary.horseName = "Midnight";
                        mary.whiskeyPreference = "Straight";
                        mary.name = "Mary";

                maurice.drinkWhiskey();
                System.out.println(maurice.howDrunkAmI());
                maurice.tieUpDamsel(mary);
        }
}
```

"That's her book?" you ask yourself. "Ten pages of slogging through ugly yellow text for this?"

So what is so special about this plot? Well, it's not everything that's happening in the main routine, but rather everything that happens behind the scenes. When Eunice created that town called Sweaty Post, she created (**instantiated**) an **object** of type Western Town. This object had all the characteristics of a standard Western Town (do you remember the binders?) The same thing goes for the Maurice object and the (pardon me for this) Mary object. While some of the traits of these objects were specified at the time of creation, most of their traits were specified back in the binders, or classes. This allowed Eunice to say a good deal while only saying a little bit in her **main routine**.

So Eunice's main plot (routine) turns out to be nothing more than a collection of references to objects, which in turn are references to classes. It is, to mix metaphors again, literary federalism, with everything being dealt with at the lowest level possible.

So what are the benefits of this? Well, imagine that instead of one page, Eunice's editor wanted a book of five hundred pages. Or let's say that Eunice decided that Maurice the villain shouldn't be so cruel. Instead of going back and laboriously changing all her instances of Maurice eating kittens and tripping old people (or whatever it is that bad folks do), Eunice could just make some changes to the Maurice object at the time of instantiation. Or she could change the Villain class to be friendlier. Or she could even create a *new* class, called FriendlyVillain with some aspects of the Samuel L. Jackson character in Pulp Fiction. She's got a lot of options; the point is that she only needs to make the changes in one place.

But that's not Eunice's only reason for writing her books in Java. She has a keen idea for the future of Westerns. She sees interactivity. And if, instead of being bound in pages, her characters live in objects and classes, Eunice is free to create a Virtual Sweaty Post. Readers (on her website) could be prompted for their own actions and her characters could respond in a variety of ways, according to what was written in their classes. Using Java, Eunice could finally bring the Old West back to life. And hey, I'm sure she's not the only one yearning for the days of outhouses, blood, and dust.

So who is this editor fellow?

You've seen many references to Eunice's editor in New York. Who is this fellow? Well, he knows the world of Westerns in and out, and he knows that certain things will just not work in a good Western. That is why her editor takes all of Eunice's scripts, reads them, and then returns them with all the errors that she has made (remember, we said that he was a bit persnickety). On the computer, this is called a **compiler.** All Java programs must be **compiled** before they can be run. The compiler will patiently (and repeatedly) tell you everything that you did wrong. And then you get to go back and do it again. Hey, that's the literary life.

So Where To From Here?

Now you're set to go and make your way in the exciting world of Java programming. You're probably saying to yourself, "No, I don't know enough yet!" Don't be silly. Java is so hyped that employers who **have no idea what it is** are listing it as a "job requirement". So give 'em what they deserve. Put "java" on your resume. If they call you on it, say something like, "Why yes, I was just instantiating some objects this morning." Then ask them something that will make them feel stupid, like "Say, if I've got a problem with vectors from the awt class, would you be the one to talk to about that?" They'll never ask again, and you'll be sipping free Odwalla by Monday. Trust me.

If, however, you just don't feel comfortable **lying to potential employers**, you may want to try some exercises. You probably noticed that this tutorial had all sorts of variables that it didn't use. Steal my code. Make it your own. Write new commands into the main routine. Create new characters. Write the sequel. Use those variables.

You of course would want to get a decent java book before you started any of this, but please feel free to come back to your home here at Don't Fear the Oop, and augment my code any day.

That's all. And if you **do** get a job, try to remember the little people who helped you along the way.