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 scary Fantasy novel.

- or -

How to understand Java by looking at pretty colours.

> 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 "objectoriented" 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.

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 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 books. Her books about Theme Parks 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 "Adventures at Fantasy Fair", her thirty-fifth book.

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

> Every Theme Park has a few key ingredients: rollercoasters, giftshops, employees, and a couple of wild children. A standard theme park would have seven roller coasters and be located south of Georgia sometime around 2003.

This description of a Theme Park, while not very profound, did two important things: 1) It established the key ingredients for a Theme Park and 2) It gave several values that might occur in a default Theme Park. (Any more than seven rollercoasters and the place starts to go wild)

```
Theme Park
        has a certain number of rollercoasters
        has a certain number of giftshops
        has a certain number of employees
        has a certain number of wildchildren
        is located somewhere
        exists at a certain time


a typical Theme Park would have
        number of roller coasters = 7
        location = South East America
        time period = 2003
```

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 Theme Park. 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 park looks) that a class of Theme Parks might have. Employees, roller coasters, wild children, etc. That being said, all that was left to do was to **construct** a sample park, with default values for her variables. Whew!

```java
public class ThemePark
{
        int rollerCoasters;
        int giftShops;
        int employees;
        int wildChildren;
        String location;
        int time;

        public ThemePark()
        {
                rollerCoasters = 7;
                location = "South East America";
                time = 2003;
        }
}
```

In making her Theme Park, Eunice is defining a class, not the park 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 Theme Park, 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 park **object** eventually in her book, her editor will look to the same Theme Parks class to determine what that park object should look like (Eunice will send the binders along with her manuscript). But where will her editor find out the number of rollercoasters 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 ThemePark(). The statements that follow are the default values for her variables; how Eunice's park **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 Theme Park section. 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 Thievery at the Cotton Candy Store
> This story takes place in a candy store at a theme park called Fantasy Fair. Fantasy Fair has six employees, two gift shops, and five wild children.

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 "Theme Park" in the binders she sends along and fill in the date, location, and number of rollercoasters. In addition, Eunice has added information about the employees, gift shops, and wild children. But that's okay, because her editor was expecting it. Because her binder stated that every theme park 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 Thievery
>
> Theme Park fantasyFair is a new Theme Park.
>   the number of giftShops in fantasyFair is two.
>   the number of employees is six.
>   the number of wildChildren is five.

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

```
public class Thievery {

        public static void main (String arguments[]) {
                ThemePark fantasyFair= new ThemePark;
                fantasyFair.giftShops = 2;
                fantasyFair.employees = 6;
                fantasyFair.wildChildren = 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** (giftShops, etc) that were not specified in the ThemePark **class**, Eunice first states the name of the **object** (in this case, fantasyFair. Remember that fantasyFair is an object of ThemePark) 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 Theme Park, then turned to her **main** plot page and created (**instantiated**) a theme park **object**, which she then called fantasyFair.

Now, for her children thief, Eunice needs a different binder. So she puts "Theme Parks" back on the shelf, and goes to pick up a new binder. Her hand passes over many different titles, like "Food", "Toys", "Weather", "Music", all the essentials for a good Park, 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 best friend with a name, and have different preferences in candy. 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 wholesome stories that has drawn so many readers to Eunice's particular brand of fairy tale. You'll note that she left the gender, name, pet name, and candy

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 gender
> > have a pet with a name
> > have a strong preference in candy
>
> 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.

```java
public class Humans {
        int legs;
        int arms;
        int eyes;
        int nose;
        int mouth;
        String name;
        String gender;
        String petName;
        String candyPreference;

        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 "Children Thieves".

> Children thieves are based on the idea of humans. They are identical, except that they have some additional qualities, namely a large grin, a dirty shirt, a certain "look", some level of craziness, and a certain quantity of candy in their possession. Your standard child thief will look innocent, start the day out as not crazy, and not yet have stolen any cotton candy.

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, child thieves, **extended** the idea of humans.

> ChildThief extends the idea of Humans.
>
>   A ChildThief
>   has a large grin.
>   has a dirty shirt.
>   has a "look".
>   has some level of craziness.
>   has a certain number of stolen cotton candy.
>
>   For a given ChildThief,
>   He/she will look innocent.
>   He/she will start out as not being crazy at all.
>   He/she will start the day without having any stolen cotton candy.

```
public class ChildThief extends Humans {
        String shirtColor;
        String look;
        int grinLength;
        int craziness;
        int numberOfStolenCandy;
        Humans tourist;

        public ChildThief() {
                look = "Innocent";
                craziness= 0;
                numberOfStolenCandy= 0;
        }
}
```

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

You may also be wondering why Eunice declared "Humans tourist" in this class. Think of it this way. If the child thief is going to steal cotton candy from a tourist, Eunice's editor has to know what a tourist is. By declaring "Humans tourist", her editor will know that a tourist is a type of human.

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

> Whenever the main plot says that a child thief eats candy, his/her level of craziness 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 "craziness". 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.

> eatCandy
>   craziness 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 eatCandy) and then, on the next line, what that method will do when she **calls** it in her **main** plot (**routine**).

```
public void eatCandy() {
        craziness++;
}
```

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 "eatCandy". Since the method name is eatCandy, Eunice figures that she should modify the villain's level of craziness, in this case by adding one to it. That is what the "++" after "craziness" does.

Now, the child thieves in Eunice's stories are famous for being able to hide their craziness. Hence, it's very difficult for an onlooker to gauge how crazy one of her child thieves really is. Eunice decides that it's a good idea to allow her child thief to state how crazy he is, so, she writes a new method:

If someone asks a child thief how crazy he/she is, the child thief will always respond with his/her level of craziness.

```
howCrazyAmI
        tell them how crazy I am
```

```
public int howCrazyAmI ()
{
        return craziness;
}
```

If you're wondering about that "int" in front of the "howCrazyAmI()", 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 flustered. (Now, we don't want that, do we?)

At this point Eunice is pretty proud of herself, having created a child thief that can experience pretty much the full range of child thieving activities. However, just to make for a good closing, Eunice decides to have the child thief have the ability to steal cotton candy from a tourist. 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 child thief, namely the tourist to be stolen from. To allow for flexibility (and for a variety of tourists), Eunice decides to leave the identity of the tourist blank for now.

If the child thief is supposed to steal a piece of cotton candy, steal the specified cotton candy, then add one to the number of cotton candies he/she has stolen. Then print out "Oh my gosh! (the specified cotton candy) has been stolen!"

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 tourist 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!

stealCottonCandy (name)
        add one to the number of cottonCandy this childThief has stolen.
        print "OMG! The cotton candy of (the specified tourist) has been stolen!"

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 stealCandy (Humans tourist) {
        this.tourist = tourist;
        numberOfStolenCandy++;
        System.out.println("The ChildThief has stolen from " +
tourist.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: "stealCandy". No surprise there. Then, between those parentheses, we have "Humans tourist". 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 child thief to steal cotton candy from a tourist (what sort of theme park story would it be without it?). But if she just writes in her plot, "childThief.stealCandy()", that wouldn't be very exciting for the reader. Who did the child thief steal from? What is their name? Priding herself on always having strong characters, Eunice wants to add the name of the tourist whose cotton candy was stolen by the child thief. Hence, when her plot says "stealCandy", it will also say the name of the tourist whose cotton candy was stolen. "Humans", which comes before "tourist", tells her editor that the tourist who had their candy stolen is a human. Her editor doesn't like surprises, so that knowing that it is a human being stolen from, 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 tourist as an argument, her editor won't let her use it in the method until she has stated that the "tourist" that she declared as a "ChildThief" 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 childThief has stolen candy from a tourist, the variable "numberOfStolenCandy" will surely go up by one, so Eunice puts the same "++" after numberOfStolenCandy that she had put after "craziness" in the "eatCandy" 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 lets 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 'tourist.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 tourist object (he would find out that it was of the "Humans" class) then for the method "whatIsYourName" (he would see that it returned the tourist's name). All of that for a tourist's name!

Had she put tourist.whatIsYourName inside of the quotation marks, her editor would have printed out in the final copy of the book : "The child thief has stolen cotton candy from + tourist.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.

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

Child thieves are based on the idea of humans. They are identical, except that they have some additional qualities, namely a large grin, a dirty shirt, a certain "look", some level of craziness, and a certain quantity of stolen cotton candy in their possession. Your standard child thief will look innocent, start the day out as not crazy, and not yet have stolen any cotton candy. Whenever the main plot says that a child thief eats candy, his level of craziness will go up by one. If someone asks a child thief how crazy he is, the child thief will always respond with his level of craziness. If the child thief is supposed to steal cotton candy from a tourist, take cotton candy from the specified tourist, then add one to the number of candy he has stolen. Then print out "Oh my gosh! The cotton candy of (the specified tourist) has been stolen!"

ChildThief extends the idea of Humans.
        Every ChildThief has a largeGrin.
        Every ChildThief has a dirtyShirt.

Every ChildThief has a "look".
Every ChildThief will have some level of craziness.
Every ChildThief will steal a certain number of cotton candy.

For a given ChildThief,
He will look innocent.
He will start out as not crazy at all.
He will start the day without having any cotton candy stolen.

eatCandy
craziness increases by one

howCrazyAmI
tell them how crazy I am

stealCottonCandy (name)
add one to the number of cottonCandy this childThief has stolen.
print "Oh my Gosh! The cotton candy of (the specified tourist) has been stolen!"

```java
public class ChildThief extends Humans {
        String shirtColor;
        String look;
        int grinLength;
        int craziness;
        int numberOfStolenCandy;
        Humans tourist;

        public ChildThief () {
                look = "Innocent";
                craziness = 0;
                numberOfStolenCandy = 0;
        }

        public void eatCandy() {
                craziness++;
        }

        public int howCrazyAmI() {
                return craziness;
        }

        public void stealCandy (Humans tourist) {
                this.tourist = tourist;
```

```
            numberOfStolenCandy++;
            System.out.println("The ChildThief has stolen from " +
            tourist.whatIsYourName());
    }
}
```

All this work with child thieves and stolen cotton candy 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 Thievery at the Cotton Candy Store: There is a Theme Park called Fantasy Fair. Fantasy Fair has six employees, two gift shops, and five wild children. There is a boy named Jack. Jack has a black dirty shirt, 7cm wide grin, and a best friend named "Marcus". Jack prefers strawberry flavoured candy. Mia is a girl. She has a best friend named "Molly" and she prefers grape flavoured candy. In our story, Jack starts out by eating a strawberry candy. He then lets everyone know how crazy he is, and then steals Mia's candy.

Here is the main plot of Thievery at the Cotton Candy Store; In the novel Thievery;

There is a theme park called fantasyFair;
        fantasyFair has six employees;
        fantasyFair has two giftShops;
        fantasyFair has five wildChildren;

There is a new child thief named jack;
        jack has a black dirty shirt;
        jack has a 7 cm wide grin;
        jack is a boy;
        jack has a best friend named "Marcus";
        jack prefers strawberry flavoured candy;

There is a new girl named Mia;
        mia is a girl;
        mia has a best friend named "Molly";
        mia prefers grape flavoured candy;

jack eats a strawberry flavoured candy;
jack tells us how crazy he is;
jack steals Mia's candy

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

                ThemePark fantasyFair = new ThemePark();
                        fantasyFair.giftShops = 2;
                        fantasyFair.employees = 6;
                        fantasyFair.wildChildren = 5;

                ChildThief jack = new ChildThief();
                        jack.shirtColor = "black";
                        jack.grinLength = 7;
                        jack.gender = "Boy";
                        jack.bestFriendName = "Marcus";
                        jack.candyPreference = "Strawberry";
                Humans mia = new Humans();
                        mia.gender = "Girl";
                        mia.bestFriendName = "Molly";
                        mia.candyPreference = "Grape";
                        mia.name = "Mia";

                jack.eatCandy();
                System.out.println(jack.howCrazyAmI());
                jack.stealCandy(mia);
        }
}
```

"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 Theme Park called Fantasy Fair, she created (**instantiated**) an **object** of type Theme Park. This object had all the characteristics of a standard Theme Park (do you remember the binders?) The same thing goes for the Jack object and the (pardon me for this) Mia 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 Jack the child thief shouldn't be so cruel. Instead of going back and laboriously changing all her instances of Jack stealing cotton candy and going crazy (or whatever it is that bad folks do), Eunice could just make some changes to the Jack object at the time of instantiation. Or she could change the ChildThief class to be friendlier. Or she could even create a *new* class, called FriendlyChildThief 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 Theme Parks. 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 Fantasy Fair. 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 Fun Parks back to life. And hey, I'm sure she's not the only one yearning for the days of trivia, rollercoasters, and cotton candy.

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 Theme Parks in and out, and he knows that certain things will just not work in a good Theme Park. 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 you did wrong. And then you get to go back and do it again. Hey, that's the literary life.