Review Object-Oriented Design

Hide tutorial navigation

This is a review of what we covered in this tutorial on object-oriented design.

When we create programs, we often find that we want to create many different objects that all share similar properties - like many cats, that have slightly different fur color and size, or many buttons, with different labels and positions. We want to be able to say "this is generally what a cat is like" and then say "let's make this specific cat, and this other cat, and they'll be similar in some ways and different in a few ways as well." In that case, we want to use ***object-oriented design*** to define object types and create new instances of those objects.

To define an object type in JavaScript, we first have to define a "constructor function". This is the function that we'll use whenever we want to create a new instance of that object type. Here's a constructor function for a `Book` object type:

```
var Book = function(title, author, numPages) {
  this.title = title;
  this.author = author;
  this.numPages = numPages;
  this.currentPage = 0;
};
```

The function takes in arguments for the aspects that will be different about each book - the title, author, and number of pages. It then sets the initial properties of the object based on those arguments, using the `this` keyword. When we use `this` in an object, we are referring to the current instance of an object, referring to itself. We need to store the properties on `this` to make sure we can remember them later.

To create an instance of a `Book` object, we declare a new variable to store it, then use the `new` keyword, followed by the constructor function name, and pass in the arguments that the constructor expects:

```
var book = new Book("Robot Dreams", "Isaac Asimov", 320);
```

We can then access any properties that we stored in the object using dot notation:

```
println("I loved reading " + book.title); // I loved reading Robot Dreams
println(book.author + " is my fav author"); // "Isaac Asimov" is my fav
author
```

Let's contrast this for a minute, and show what would have happened if we didn't set up our constructor function properly:

```
var Book = function(title, author, numPages) {
};
var book = new Book("Little Brother", "Cory Doctorow", 380);
println("I loved reading " + book.title); // I loved reading undefined
println(book.author + " is my fav author"); // undefined is my favorite
author
```

If we pass the arguments into the constructor function but do not explicitly store them on `this`, then we will *not* be able to access them later! The object will have long forgotten about them.

When we define object types, we often want to associate both properties *and behavior* with them - like all of our cat objects should be able to meow() and eat(). So we need to be able to attach functions to our object type definitions, and we can do that by defining them on what's called the **object prototype**:

```
Book.prototype.readItAll = function() {
  this.currentPage = this.numPages;
  println("You read " + this.numPages + " pages!");
};
```

It's like how we would define a function normally, except that we hang it off the `Book`'s prototype instead of just defining it globally. That's how JavaScript knows that this is a function that can be called on any `Book` object, and that this function should have access to the `this` of the book that it's called on.

We can then call the function (which we call a ***method***, since it's attached to an object), like so:

```
var book = new Book("Animal Farm", "George Orwell", 112);
book.readItAll(); // You read 112 pages!
```

Remember, the whole point of object-oriented design is that it makes it easy for us to make multiple related objects (***object instances***). Let's see that in code:

```
var pirate = new Book("Pirate Cinema", "Cory Doctorow", 384);
var giver = new Book("The Giver", "Lois Lowry", 179);
var tuck = new Book("Tuck Everlasting", "Natalie Babbit", 144);

pirate.readItAll(); // You read 384 pages!
giver.readItAll(); // You read 179 pages!
tuck.readItAll(); // You read 144 pages!
```

That code gives us three books that are similar - they all have the same types of properties and behavior, but also different. Sweet!

Now, if you think about the world, cats and dogs are different types of objects, so you'd probably create different object types for them if you were programming a `Cat` and a `Dog`. A cat would `meow()`, a dog would `bark()`. But they're also similar- both a cat and dog would `eat()`, they both have an `age`, and a `birth`, and a `death`. They're both mammals, and that means they share a lot in common, even if they're also different.

In that case, we want to use the idea of **object inheritance**. An object type could inherit properties and behavior from a parent object type, but then also have its own unique things about it. All the `Cat`s and `Dog`s could inherit from `Mammal`, so that they wouldn't have to invent `eat()`ing from scratch. How would we do that in JavaScript?

Let's go back to our book example, and say that `Book` is the "parent" object type, and we want to make two object types that inherit from it - `Paperback` and `EBook`.

A paperback is like a book, but it has one main thing different, at least for our program: it has a cover image. So, our constructor needs to take four arguments, to take in that extra info:

```
var PaperBack = function(title, author, numPages, cover) {
  // ...
}
```

Now, we don't want to have to do all the work that we already did in the `Book` constructor to remember those first three arguments - we want to take advantage of the fact that the code for that would be the same. So we can actually call the `Book` constructor from the `PaperBack` constructor, and pass in those arguments:

```
var PaperBack = function(title, author, numPages, cover) {
  Book.call(this, title, author, numPages);
  // ...
};
```

We still need to store the `cover` property in the object though, so we need one more line to take care of that:

```
var PaperBack = function(title, author, numPages, cover) {
  Book.call(this, title, author, numPages);
  this.cover = cover;
};
```

Now, we have a constructor for our `PaperBack`, which helps it share the same properties as `Book`s, but we also want our `PaperBack` to inherit its methods. Here's how we do that, by telling the program that the `PaperBack` prototype should be based on the `Book` prototype:

```
PaperBack.prototype = Object.create(Book.prototype);
```

We might also want to attach paperback-specific behavior, like being able to burn it, and we can do that by defining functions on the prototype, after that line above:

```
PaperBack.prototype.burn = function() {
  println("Omg, you burnt all " + this.numPages + " pages");
  this.numPages = 0;
};
```

And now we can create a new paperback, read it all, and burn it!

```
var calvin = new PaperBack("The Essential Calvin & Hobbes", "Bill
Watterson", 256, "http://ecx.images-amazon.com/images/I/61M41hxr0zL.jpg");

calvin.readItAll(); // You read 256 pages!
calvin.burn(); // Omg, you burnt all 256 pages!
```

(Well, we're not really going to burn it, because that's an amazing book, but perhaps if we were stuck in a glacial desert and desperate for warmth and about to die.)

And now you can see how we can use object-oriented design principles in JavaScript to create more complex data for your programs and model your program worlds better.