A Drip of JavaScript

Using Duck Typing to Avoid Conditionals in JavaScript

Originally published in the A Drip of JavaScript newsletter.

As I've mentioned before, one of the best ways to simplify your code is to eliminate unnecessary `if` and `switch` statements. In this drip, we're going to look at using duck typing to do that.

Consider the example of a library filing system which has to deal with books, magazines, and miscellaneous documents:

```
function Book (title) {
    this.title = title;
}

function Magazine (title, issue) {
    this.title = title;
    this.issue = issue;
}

function Document (title, description) {
    this.title = title;
    this.description = description;
}

var laic = new Book("Love Alone is Credible");
var restless = new Magazine("Communio", "Summer 2007");
var le = new Document("Laborem Exercens", "encyclical");
```

```
// A simple array where we keep track of things that are filed.
filed = [];

function fileIt (thing) {
    // We conditionally file in different places
    // depending on what type of thing it is.

    if (thing instanceof Book) {
        console.log("File in main stacks.");
    } else if (thing instanceof Magazine) {
        console.log("File on magazine racks.");
    } else if (thing instanceof Document) {
        console.log("File in document vault.");
    }

    // Mark as filed
    filed.push(thing);
}

// Outputs: "File in main stacks.""
fileIt(laic);

// Outputs: "File on magazine racks."
fileIt(restless);

// Outputs: "File in document vault."
fileIt(le);
```

That may not look like a lot of conditional logic, but once the library starts handling museum pieces, photographs, CDs, DVDs, etc. it can become quite unwieldy. That's where duck typing can come to our rescue.

The term "duck typing" comes from the saying, "If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck." The implication is that in our code we don't need to test that something is a duck if we know that it can quack.

Here's how a duck typed version of our code might look:

```
function Book (title) {
    this.title = title;
}

Book.prototype.file = function() {
    console.log("File in main stacks.");
}

function Magazine (title, issue) {
    this.title = title;
    this.issue = issue;
}

Magazine.prototype.file = function() {
    console.log("File on magazine racks.");
}

function Document (title, description) {
    this.title = title;
    this.description = description;
}

Document.prototype.file = function() {
    console.log("File in document vault.");
}

var laic = new Book("Love Alone is Credible");
var restless = new Magazine("Communio", "Summer 2007");
var le = new Document("Laborem Exercens", "encyclical");

// A simple array where we keep track of things that are filed.
filed = [];

function fileIt (thing) {
    // Dynamically call the file method of whatever
    // `thing` was passed in.
    thing.file();
```

```
    // Mark as filed
    filed.push(thing);
}

// Outputs: "File in main stacks.""
fileIt(laic);

// Outputs: "File on magazine racks."
fileIt(restless);

// Outputs: "File in document vault."
fileIt(le);
```

Notice that for each different type of object we might file, we just define a `file` method on the prototype to encapsulate the logic specific to that type of object. The only thing that `fileIt` cares about is whether the `file` method exists.

If it quacks it must be a duck, and if it has a `file` method, then it must be fileable.

That means that not only is our conditional logic eliminated, but adding new types of objects to our filing system is a snap. For example:

```
function CompactDisc (title) {
    this.title = title;
}

CompactDisc.prototype.file = function() {
    console.log("File in music section.");
}

var coc = new CompactDisc("A Ceremony of Carols");

// Outputs: "File in music section."
fileIt(coc);
```

No modifications to `fileIt` were necessary. All we had to do was define an appropriate `file` method for `CompactDisc` objects.

Less conditional logic plus easy extensibility equals a win in my book.

Those of you with a background in more classical languages might be wondering about the advantages of using duck typing instead of more traditional subtype polymorphism. Here is an illustration:

```
var weirdModernArtPiece = {
    title: "différance",
    file: function() {
        console.log("File this where no one can see it.");
    }
}

// Outputs: "File this where no one can see it."
fileIt(weirdModernArtPiece);
```

In this example, we created an *ad hoc* object and `fileIt` handled it just fine. Duck typing lets us use any object, regardless of its type, or even whether it has a specific type at all. The ability to use *ad hoc* objects means that our system can more easily adapt to changing requirements.
