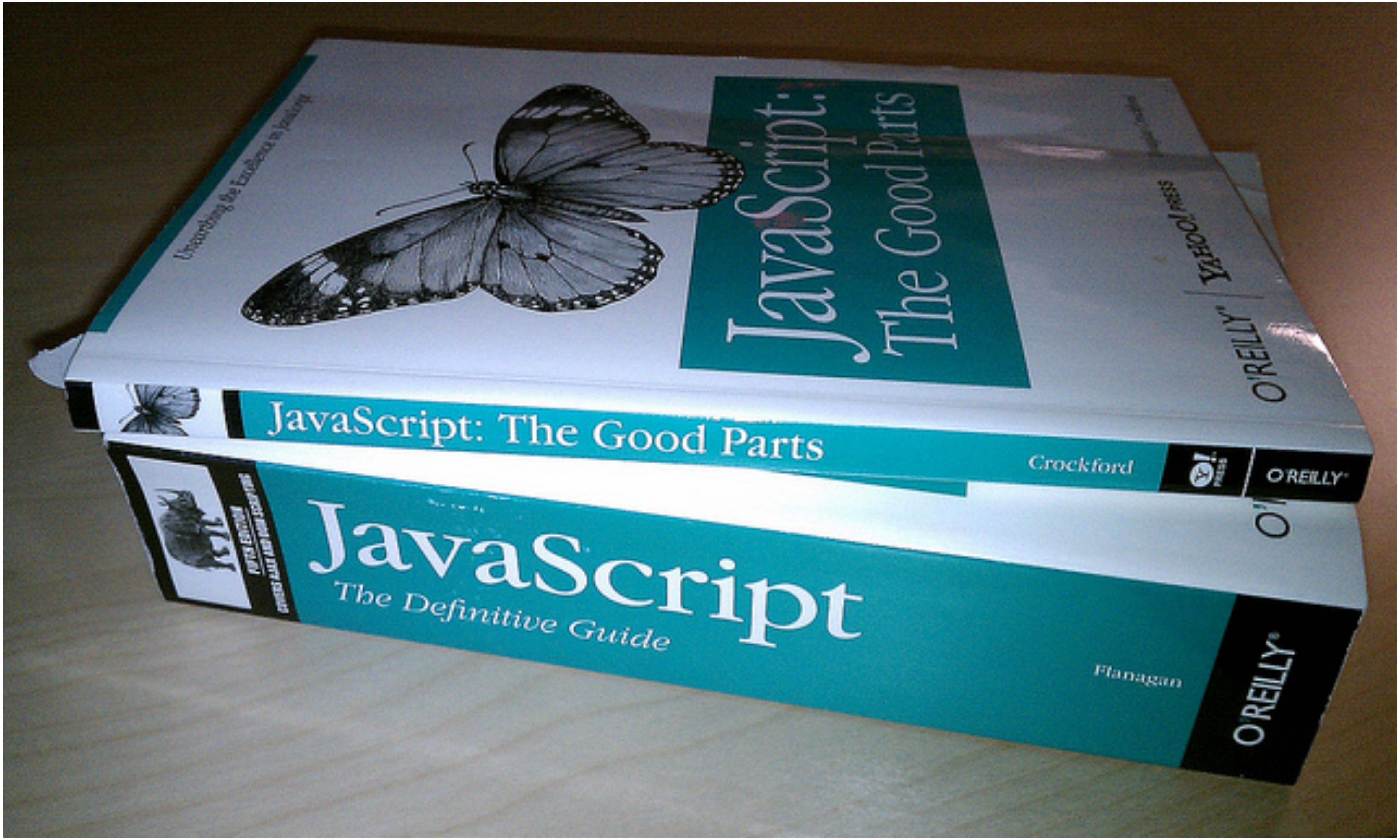
JavaScript Concepts


Unearthing the Excellence in JavaScript
JavaScript: The Good Parts
O'REILLY
YAHOO! PRESS
JavaScript: The Good Parts
Crockford
O'REILLY
FIFTH EDITION
JavaScript
The Definitive Guide
Flanagan
O'REILLY

JavaScript is pretty hard to escape if you want to do anything for the web

C of the Internet

“Take JavaScript for instance. It's widely criticized in the POPL community for getting many things wrong. But it must have gotten a ton of things right too, otherwise it wouldn’t be so popular. ”

– *Nikhil Swamy, MS Research*

Javascript is:

- Dynamically Typed
- Object-Oriented-ish
- Functional-ish

node
JS
®

Basically everything in JavaScript is an Object

Why do people dislike JavaScript?

- Implicit Conversions, it’s hard to make JS **crash**
  - Easily leads to strange behavior, unpredictible
  - So you have to test your code a lot
- Weird behavior of builtins, == vs ===, etc…
- Javascript uses **prototypical inheritance**
  - If you think about it like C++/Java, you will be terribly wrong

Numbers, Strings, Booleans, `null`, and `undefined`

Everything else is an object!

The hardest thing to get your head around in JS is that objects don’t belong to a **class** per-se. Classes still exist, but they’re more like **recipes** for objects

These are all **objects** in JS

```
{a: 23}
```

```
{foo: 12, bar: (x) => x}
```

```
{a: “hello”}
```

Observations: JS “objects” are mostly dictionaries

```
{
  speed: 12,
  distance: 13
}
```

```
x.speed
```

```
x[“speed”]
```

Write functions using the **function** keyword

```
function dist(x0, x1) {
   return Math.sqrt(
       (x1[0]-x0[0])**2
     + (x1[1]-x0[1])**2);
 }
```

```
> dist
[Function: dist]
```

Like Racket, JavaScript has a fairly functional flavor to it…

```
var x = function(x) { return x**2; }
```

```
function twice(f) { return function(x) { return f(f(x)); }; }
```

```
> [1,2,3].map(function (x) { return x.toString(); })
[ '1', '2', '3' ]
```

Another way to write functions in JS

```
> [1,2,3,4].reduce ((acc,next) => acc + " " + next.toString());
'1 2 3 4'
```

JavaScript has **closures**

```
function countUpFrom(x) {
    var counter = x;
    return function() {
       var cur = counter;
       counter = cur+1;
       return cur;
} };
```

```
> var startingAtFive = countUpFrom(5);
undefined
> startingAtFive()
5
> startingAtFive()
6
> startingAtFive()
7
```

Currying…

```
var myFirstCurry = function(word) {
     return function(user) {
            return [word , ", " , user].join("");
     };
};

var HelloUser = myFirstCurry("Hello");
HelloUser("Aadhya"); // Output: "Hello, Aadhya”
```

Q: What does this look like in Racket?

Classes

```
function Apple (typeofapple) {
    this.typeofapple = typeofapple;
    this.color = "red";
    this.getInfo = function() {
        return this.color + ' ' + this.typeofapple + ' apple';
    };
}
```

Critical question: what gets passed in for **this**?

```
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = function() {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

Observation: if I don’t explicitly specify, it goes to the **default** object

new Apple(3)

- Creates an **empty object**, let’s call it x
- Binds this to x
- Runs the Apple function using x as this

```
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = function() {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

new Apple(3)

This: {}

```
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = function() {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

```
Apple { type: 3, color: 'red', getInfo: [Function] }
```

One really crummy thing about JS: it **silently fails**

```
> new Apple
Apple { type: undefined, color: 'red', getInfo: [Function] }
```

E.g., `Apple` needed an argument, but we didn't pass it one. So JS just fills in `undefined`

I can **explicitly** specify `this` by using the `call` builtin function

```
> var x = {};
> Apple.call(x)
undefined
> x
{ type: undefined, color: 'red', getInfo: [Function] }
```

Note: not idiomatic JS

JS has a **strange** take on inheritance…

Object literal

```
var Car = {
  name: “plain old car”
}
```

Object literal

```
var car = {
wheels:
  function() { return "I have "
               + this.numWheels(); }
}
```

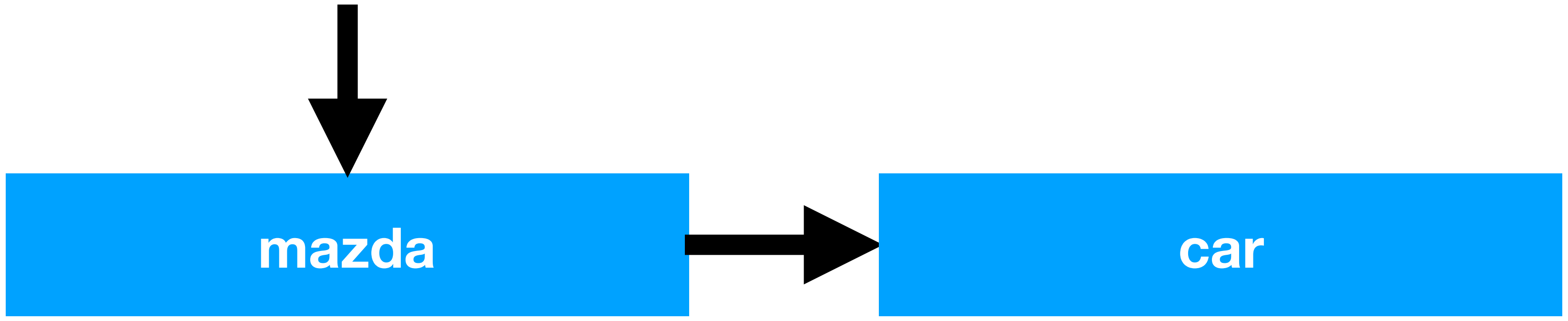
Note: runtime error if I call wheels

```
var car = {
wheels:
  function() { return "I have "
               + this.numWheels(); }
}
```

```
var mazda = {
  numWheels: function() { return 4; }
 };
```

```
> mazda.__proto__ = car;
{ wheels: [Function: wheels] }
> mazda
{ numWheels: [Function: numWheels] }
> mazda.__proto__
{ wheels: [Function: wheels] }
> mazda.wheels()
'I have 4'
```

When I want to look up **wheels**

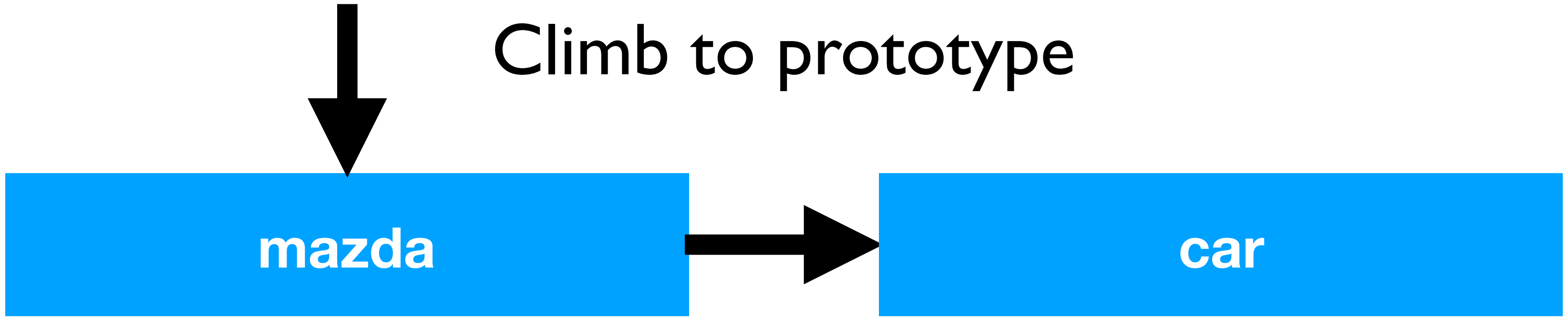


mazda

car

Not here!!!

When I want to look up **wheels**



Climb to prototype

mazda

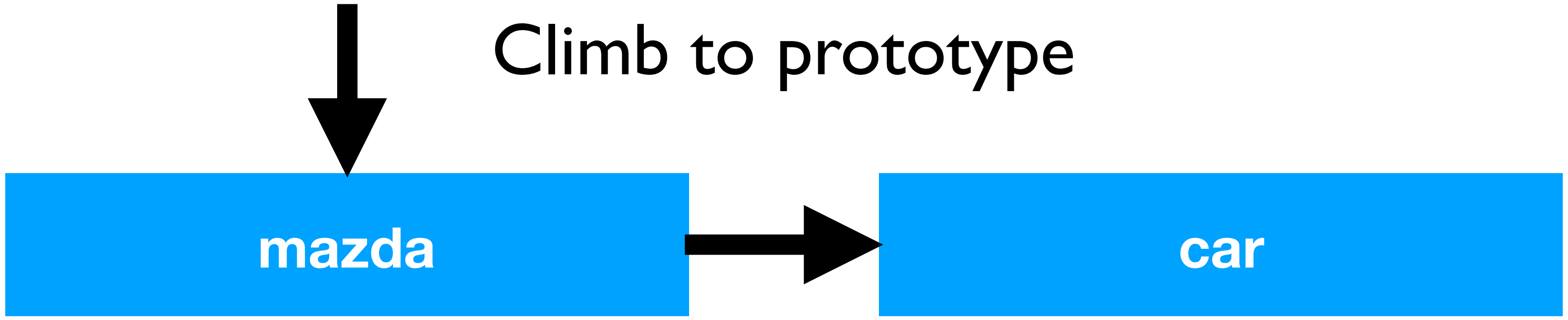
car

Not here!!!

Found **wheels** here

```
function() { return "I have "
                + this.numWheels(); }
```

When I want to look up **wheels**



Climb to prototype

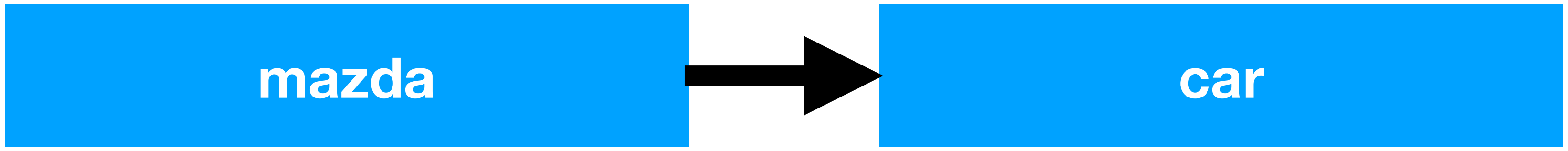
Not here!!!

Found **wheels** here

```
function() { return "I have "
                  + this.numWheels(); }
```

Now, **this** is `mazda`

Need to lookup `numWheels`



Finds it here!

Lookups are **dynamic**

```
car.wheels = function() { return “I have some number”; }
```

`mazda.wheels()` now gives `“I have some number”`

Using `__proto__` directly is terrible form

Instead, use `Object.create(car)`

`Object.create(car)`

- Creates a new object
- Sets its prototype to be `car`
- Now all lookups go through `car`
  - Unless you set otherwise, of course

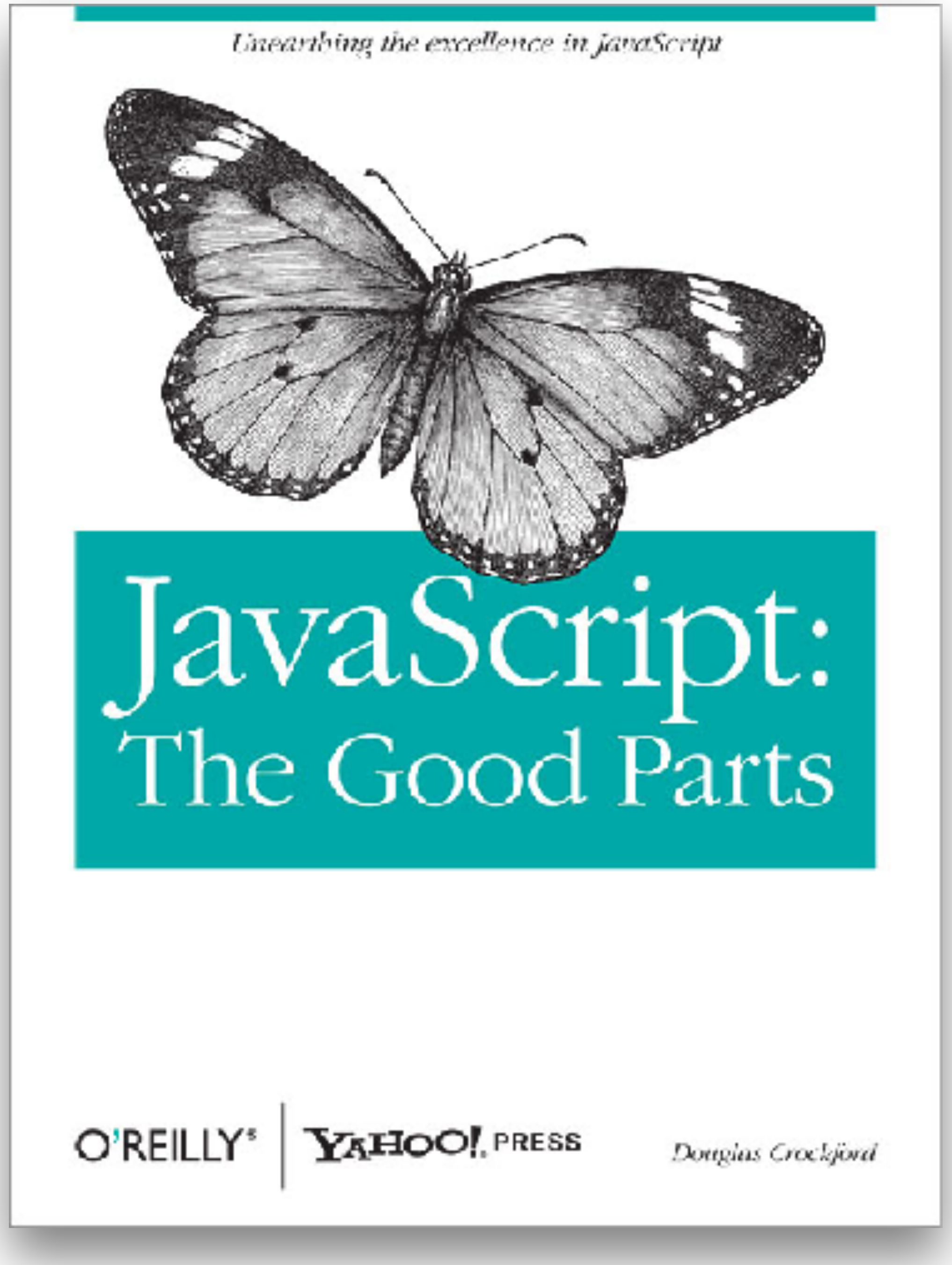
`Object.create(car)`

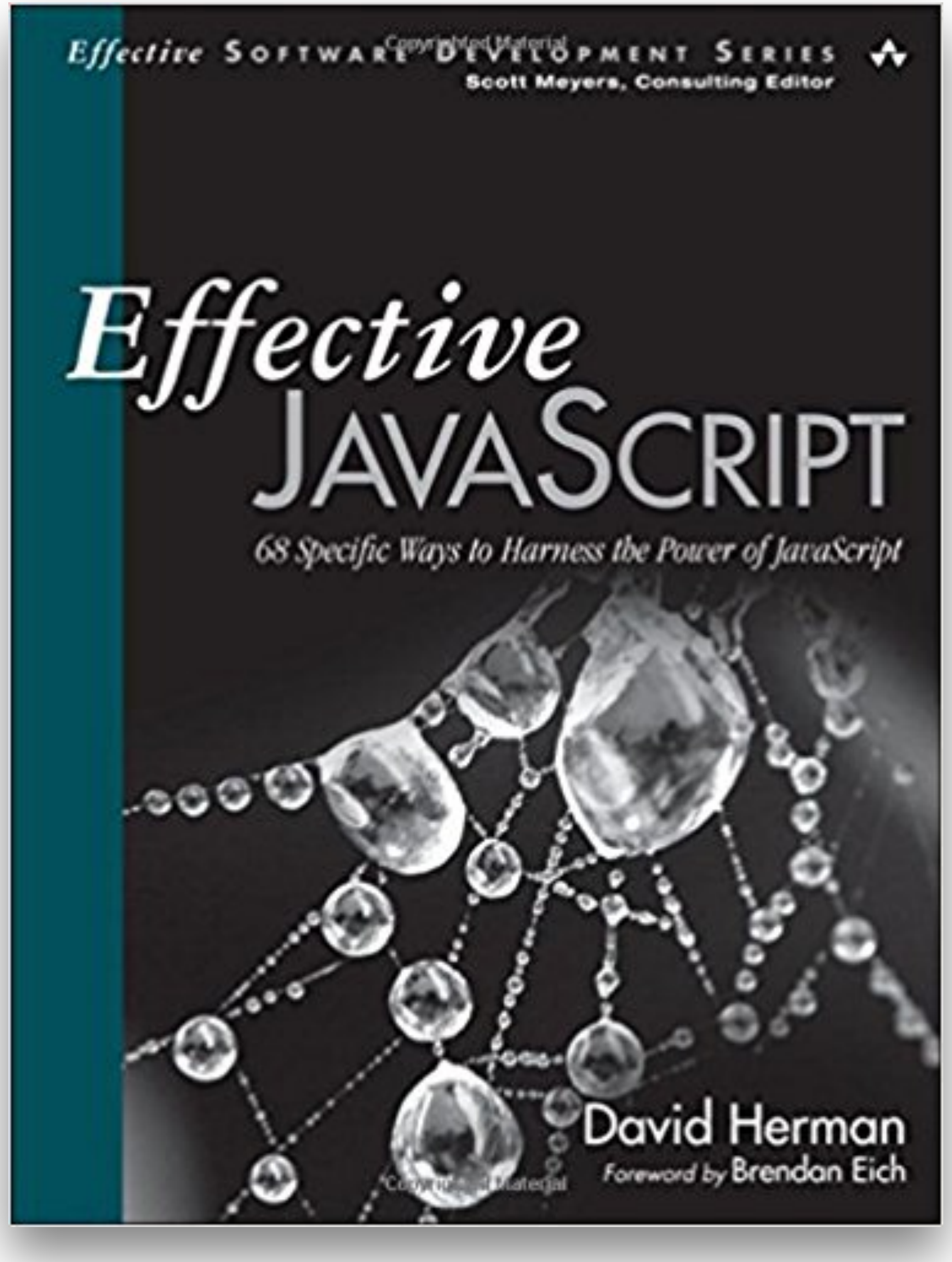
- Creates a new object
- Sets its prototype to be `car`
- Now all lookups go through `car`
  - Unless you set otherwise, of course

This effectively enables using car as a **class**

In the sense that a class is a blueprint for an object https://www.infoworld.com/article/3196070/node-js/10-javascript-concepts-nodejs-programmers-must-master.html

http://sporto.github.io/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/


Unearthing the excellence in JavaScript
JavaScript:
The Good Parts
O'REILLY
YAHOO! PRESS
Douglas Crockford


Effective Software Development Series
Scott Meyers, Consulting Editor
Effective
JavaScript
68 Specific Ways to Harness the Power of JavaScript
David Herman
Foreword by Brendan Eich

Why do people dislike JavaScript?

- Implicit Conversions, it’s hard to make JS **crash**
  - Easily leads to strange behavior, unpredictible
  - So you have to test your code a lot
- Weird behavior of builtins, == vs ===, etc…
- Javascript uses **prototypical inheritance**
  - If you think about it like C++/Java, you will be terribly wrong