

Lesson - 26

'this', call/apply, bind and prototype



Lesson Plan

- HW Review
- What is 'this' ?
- call/apply and bind
- What is prototype ?

What is 'this' in JavaScript?



Ölbaum

@oscherler



JavaScript makes me want to flip the table and say “Fuck this shit”, but I can never be sure what “this” refers to.

7:03 AM · Oct 30, 2015 · [Twitterrific for Mac](#)

1.2K Retweets **1.3K** Likes



‘this’

this is a special key word in javascript that refers to an object it belongs to or in other words refers to invoker object (parent object)

this keyword refers to an object, that object which is executing the current bit of javascript code.

In other words, every javascript function while executing has a reference to its current execution context, called **this**.

Like in real conversation, the value of ‘this’ is inferred from **context**

Execution context means here is how the function is called.

To understand this keyword, only we need to know how, when and from where the function is called, does not matter how and where function is declared or defined.

‘this’

It has different values depending on where it is used:

- In a method, **this** refers to the **owner object**.
- Alone, **this** refers to the **global object**.
- In a function, **this** refers to the **global object**.
- In a function, in strict mode, **this** is **undefined**.
- In an event, **this** refers to the **element** that received the event.
- Methods like **call()**, and **apply()** can refer **this** to **any object**.


this in a Method

It's common that an object method needs to access the information stored in the object to do its job.

For instance, the code inside `user.sayHi()` may need the name of the user.

To access the object, a method can use the `this` keyword.

The value of `this` is the object “before dot”, the one used to call the method.



```
let user = {  
  name: "John",  
  age: 30,  
  
  sayHi() {  
    // "this" is the "current object"  
    alert(this.name);  
  }  
  
};  
  
user.sayHi(); // John
```

this in a Method

Here during the execution of `user.sayHi()`, the value of `this` will be `user`.

Technically, it's also possible to access the object without `this`, by referencing it via the outer variable:

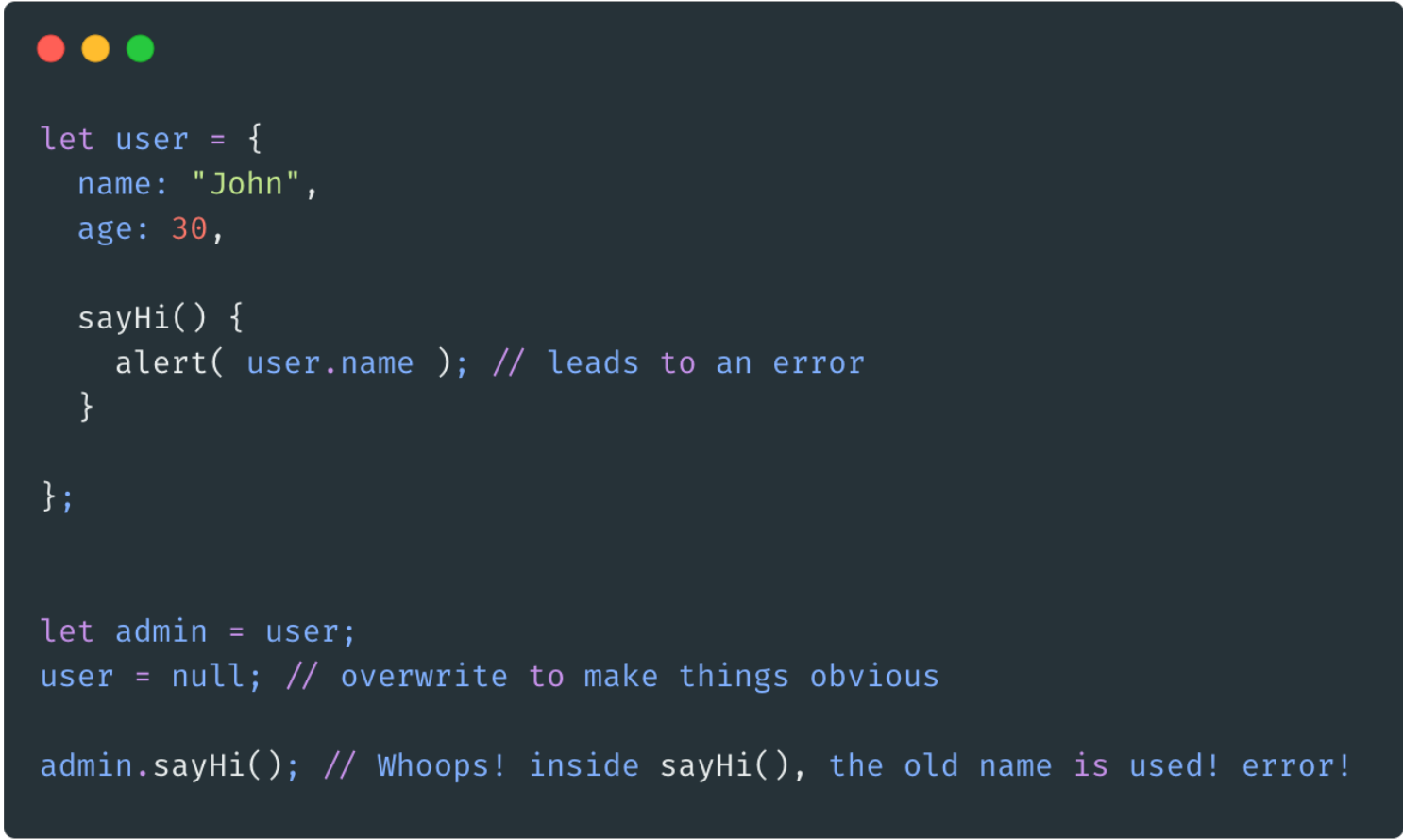


```
let user = {  
  name: "John",  
  age: 30,  
  
  sayHi() {  
    alert(user.name); // "user" instead of "this"  
  }  
  
};
```

this in a Method

But such code is unreliable. If we decide to copy `user` to another variable, e.g. `admin = user` and overwrite `user` with something else, then it will access the wrong object.

That's demonstrated below



```
let user = {  
  name: "John",  
  age: 30,  
  
  sayHi() {  
    alert( user.name ); // leads to an error  
  }  
};  
  
let admin = user;  
user = null; // overwrite to make things obvious  
  
admin.sayHi(); // Whoops! inside sayHi(), the old name is used! error!
```


this alone

When used alone, the **owner** is the Global object, so **this** refers to the Global object.

In a browser window the Global object is **[object Window]**




```
let x = this;  
console.log(this) // window
```

this in function

When used alone, the **owner** is the Global object, so **this** refers to the Global object.


In a browser window the Global object is **[object Window]**



```
function showMeThis() {  
    console.log(this)  
}
```

This in eventHandlers

In HTML event handlers, `this` refers to the HTML element that received the event:



```
<button
  onclick="this.innerText = 'Click has changed me'; this.style.color = 'green'">
  Change me
</button>
```

This in eventHandlers

!!! Be careful, very common mistake:



```
element.onclick = someKindOfFunction;
```

```
// this would refer to the element object. But be careful,  
// a lot of people make this mistake.
```


```
<element onclick="someKindOfFunction()">
```

In the second case, you merely reference the function, not hand it over to the element. Therefore, this will refer to the window object.

Explicit binding

The `call()` and `apply()` methods are predefined JavaScript methods.

They can both be used to call an object method with another object as argument.



```
const person1 = {
  name: 'Mike',
  lastName: 'Doqe',
}

const person2 = {
  name: 'Joqhn',
  lastName: 'Smithg',
}

function showFullName() {
  console.log(`${this.name} - ${this.lastName}` );
}

showFullName.call(person1, 'hey', 'hou');
showFullName.apply(person2, ['hey', 'hou']);
```

this with bind()

Functions provide a built-in method **bind** that allows to fix this.

The result of `func.bind(context)` is a special function-like “exotic object”, that is callable as function and transparently passes the call to `func` setting `this=context`.

In other words, calling `boundFunc` is like `func` with fixed `this`.



```
const person1 = {
  name: 'Mike',
  lastName: 'Doqe',
}

function showFullName() {
  return this.name + " " + this.lastName;
}

let person1FullName = showFullName.bind(person1);
console.log(person1FullName());
```

Prototypes

In the previous modules, we've covered objects, properties, and mutation. It's tempting to jump to other topics, but we're not quite done with objects yet!

Here is a small riddle to check our mental model:

```
let pizza = {};  
console.log(pizza.taste); // "pineapple"
```

Ask yourself: is this possible?

Prototypes

We have just created an empty object with `{ }`. We definitely didn't set any properties on it before logging. So it seems like `pizza.taste` can't point to `"pineapple"`. We would expect `pizza.taste` to give us `undefined` instead. (We usually get `undefined` when a property doesn't exist, right?)

And yet, it *is* possible to add some code before these two lines that would cause `pizza.taste` to be `"pineapple"`! This may be a contrived example, but it shows that our mental model of the JavaScript universe is incomplete.

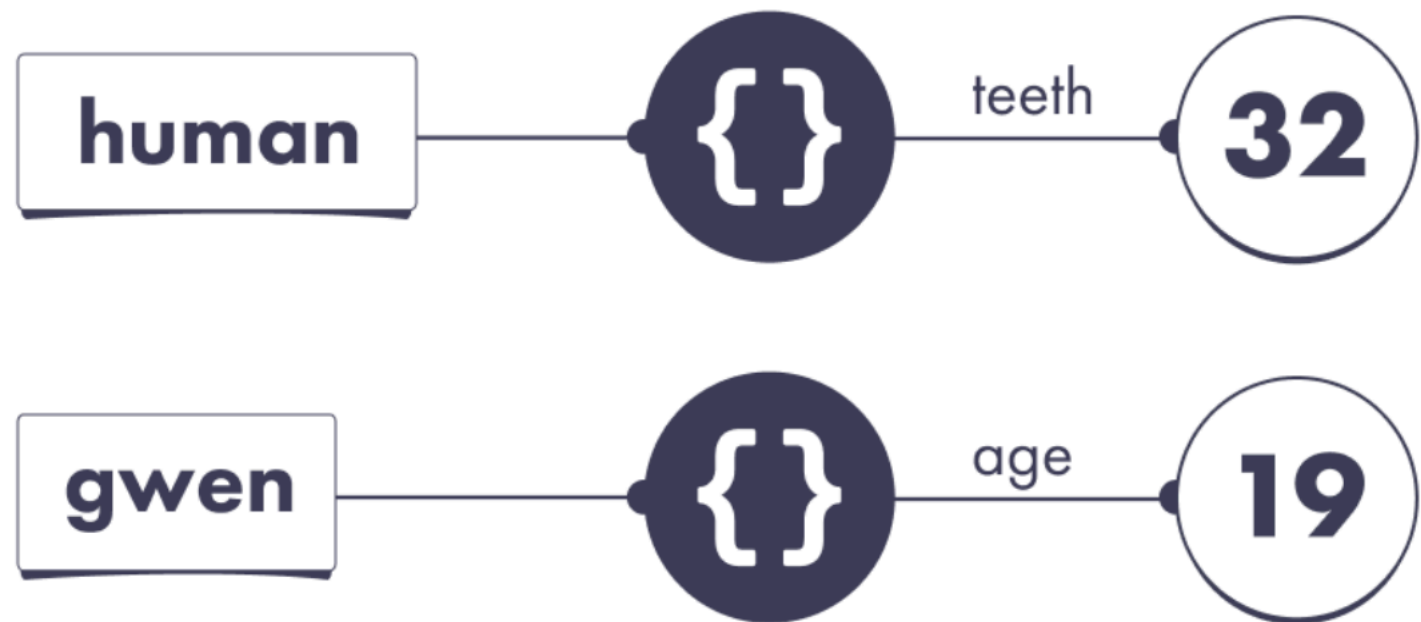
In this module, we'll introduce the concept of *prototypes*. Prototypes explain what happens in this puzzle. More importantly, prototypes are at the heart of several other JavaScript features. Occasionally people neglect learning them because they seem too unusual. However, the core idea is remarkably simple.

Prototypes

Here's a couple of variables pointing at a couple of objects:

We can represent them visually in a familiar way:

```
let human = {  
  teeth: 32  
};  
  
let gwen = {  
  age: 19  
};
```



Prototypes

In this example, gwen points at an object without a `teeth` property. According to the rules we've learned, if try to read it, we get `undefined`:

```
console.log(gwen.teeth); // undefined
```

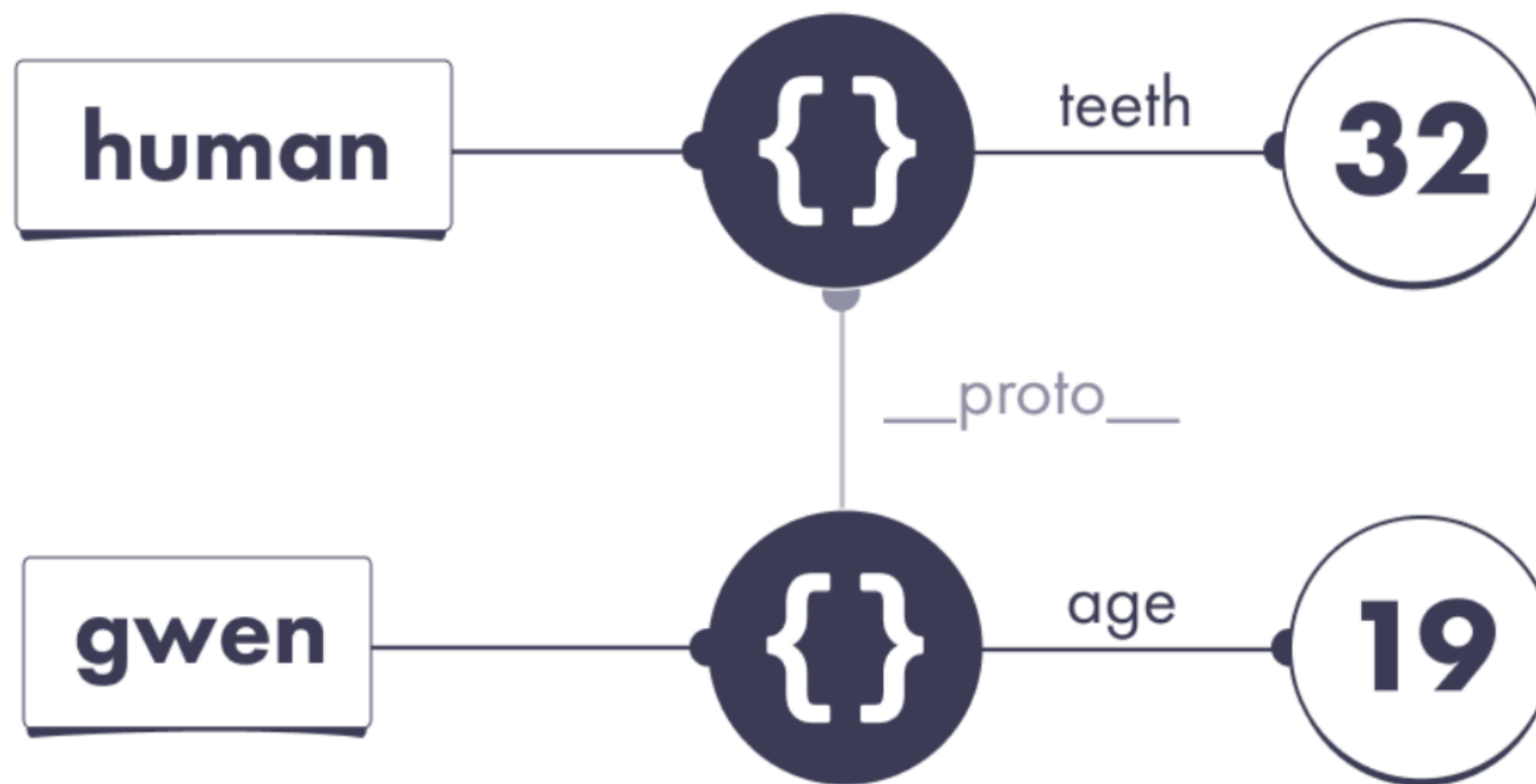
But the story doesn't have to end here. Instead of the default behavior of returning `undefined`, we can instruct JavaScript to *continue searching for our missing property on another object*. We can do it with one line of code:



```
let human = {  
  teeth: 32  
};  
  
let gwen = {  
  // We added this line:  
  __proto__: human,  
  age: 19  
};
```

What is `__proto__`?

It represents the JavaScript concept of a *prototype*. Any JavaScript object may choose another object as a prototype. We will discuss what that means in practice very soon. For now, let's think of it as a special `__proto__` wire:



Prototypes in action

Earlier, when we went looking for `gwen.teeth`, we got `undefined` because the `teeth` property doesn't exist on the object that `gwen` points at.

But thanks to that `__proto__: human` line, the answer is different now:

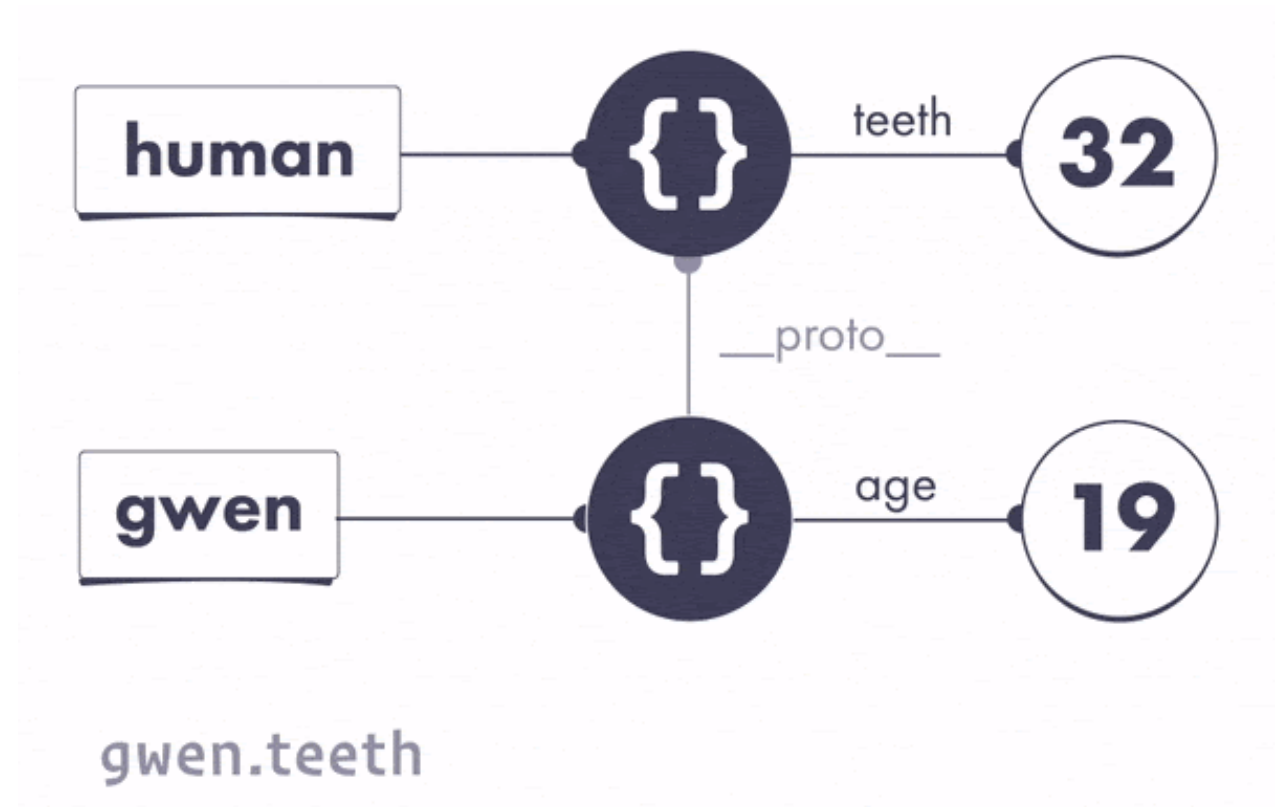
```
let human = {  
  teeth: 32  
};
```

```
let gwen = {  
  // "Look for other properties here"  
  __proto__: human,  
  age: 19  
};
```

```
console.log(gwen.teeth); // 32
```

Prototypes in action

1. Follow the `gwen` wire. It leads to an object.
2. Does this object have a `teeth` property?
 - No.
 - **But it has a prototype.** Let's check it out.
3. Does *that* object have a `teeth` property?
 - Yes, it points at 32.
 - Therefore, the result of `gwen.teeth` is 32.



Prototypes in action

This is similar to how you might say at work: “I don’t know, but Alice might know”. With `__proto__`, you instruct JavaScript to “ask another object”.

To check your understanding so far, write down your answers:

```
let human = {  
  teeth: 32  
};
```

```
let gwen = {  
  __proto__: human,  
  age: 19  
};
```

```
console.log(human.age); // ?  
console.log(gwen.age); // ?
```

```
console.log(human.teeth); // ?  
console.log(gwen.teeth); // ?
```

Prototypes in action

The `human` variable points at an object that doesn't have an `age` property, so `human.age` is `undefined`. The `gwen` variable points at an object that *does* have an `age` property. That wire points at 19, so the value of `gwen.age` is 19:

```
console.log(human.age); // undefined  
console.log(gwen.age); // 19
```

The `human` variable points at object that has a `teeth` property, so the value of `human.teeth` is 32. The `gwen` variable points at an object that doesn't have a `teeth` property. However, that object has a prototype, which *does* have a `teeth` property. This is why the value of `gwen.teeth` is also 32.

```
console.log(human.teeth); // 32  
console.log(gwen.teeth); // 32
```

Prototypes in action

Neither of our objects have a `tail` property, so we get `undefined` for both:

```
console.log(human.tail); // undefined  
console.log(gwen.tail); // undefined
```

Note how although the value of `gwen.teeth` is 32, it doesn't mean `gwen` has a `teeth` property! Indeed, in this example, `gwen` *does not* have a `teeth` property. But its prototype object — the same one `human` points at — does.

This serves to remind us that `gwen.teeth` is an *expression* — a question to the JavaScript universe — and JavaScript will follow a sequence of steps to answer it. Now we know that these steps involve looking at the prototype.

Prototype chain

A prototype isn't a special "thing" in JavaScript. A prototype is more like a *relationship*. An object may point at another object as its prototype.

This naturally leads to a question: but what if my object's prototype has its own prototype? And that prototype has its own prototype? Would that work?

The answer is that this is *exactly* how it works!

```
let mammal = {  
  brainy: true,  
};
```

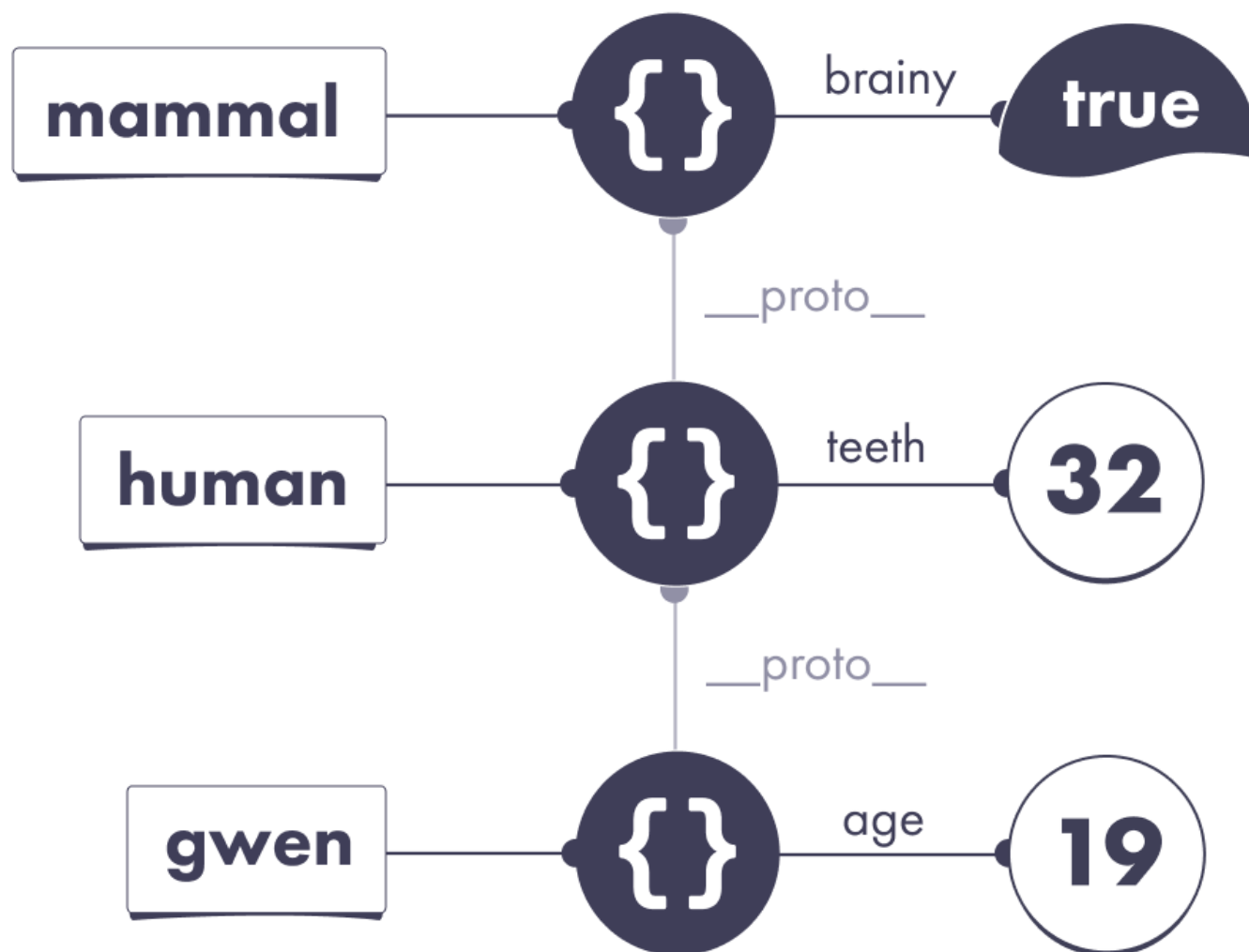
```
let human = {  
  __proto__: mammal,  
  teeth: 32  
};
```

```
let gwen = {  
  __proto__: human,  
  age: 19  
};
```

```
console.log(gwen.brainy); // true
```

Prototype chain

We can see that JavaScript will search for the property on our object, then on its prototype, then on *that* object's prototype, and so on. We would only get undefined if we ran out of prototypes and still haven't found our property.



Shadowing

Consider this slightly modified example:

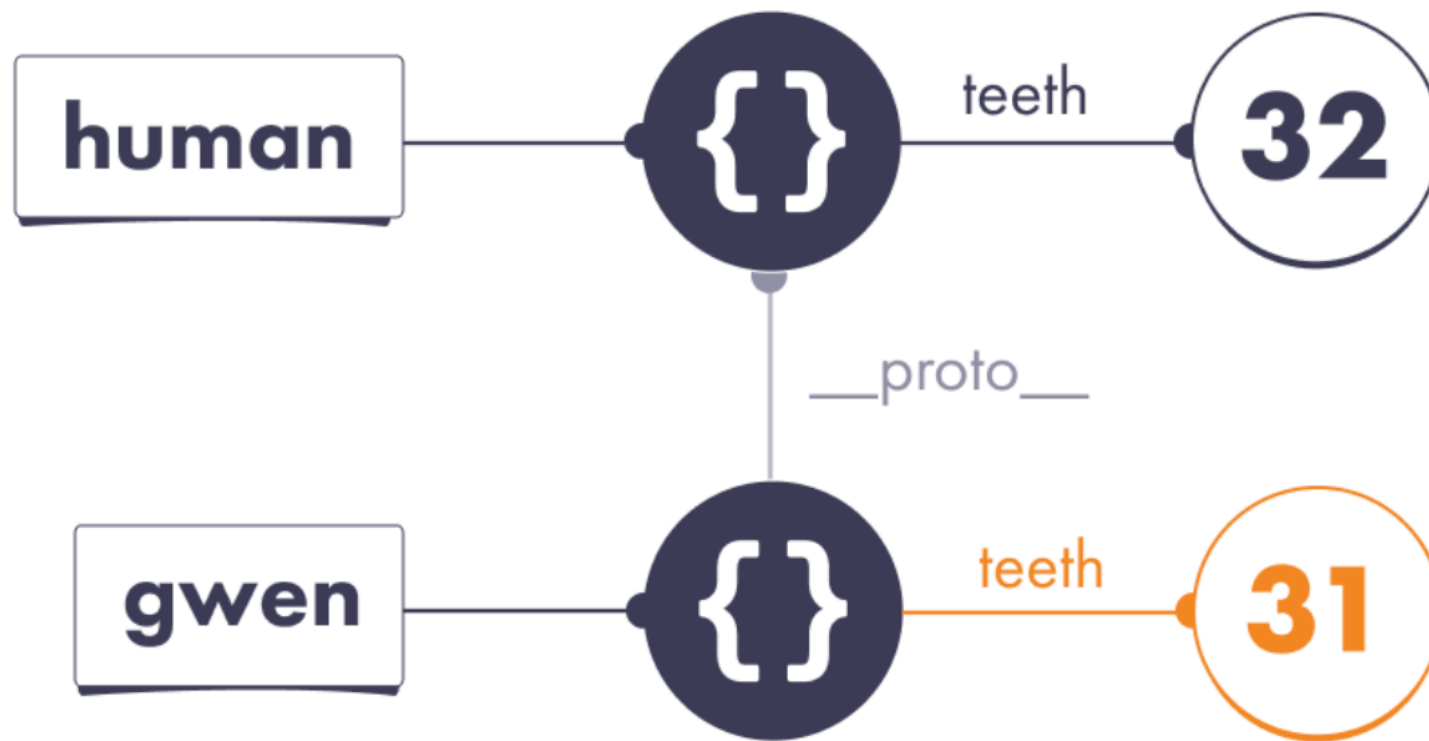
```
let human = {  
  teeth: 32  
};  
  
let gwen = {  
  __proto__: human,  
  // This object has its own teeth property:  
  teeth: 31  
};
```

Both objects define a property called `teeth`, so the results are different:

```
console.log(human.teeth); // 32  
console.log(gwen.teeth); // 31
```

Shadowing

Note that `gwen.teeth` is 31. If `gwen` didn't have its own `teeth` property, we would look at the prototype. But because the object that `gwen` points at has its *own* `teeth` property, we don't need to keep searching for the answer.



`gwen.teeth`



In other words, once we find our property, **we stop the search.**

The object prototype

We will now take a much closer at objects than ever before.

```
let obj = {};
```

This object doesn't have a prototype, right?

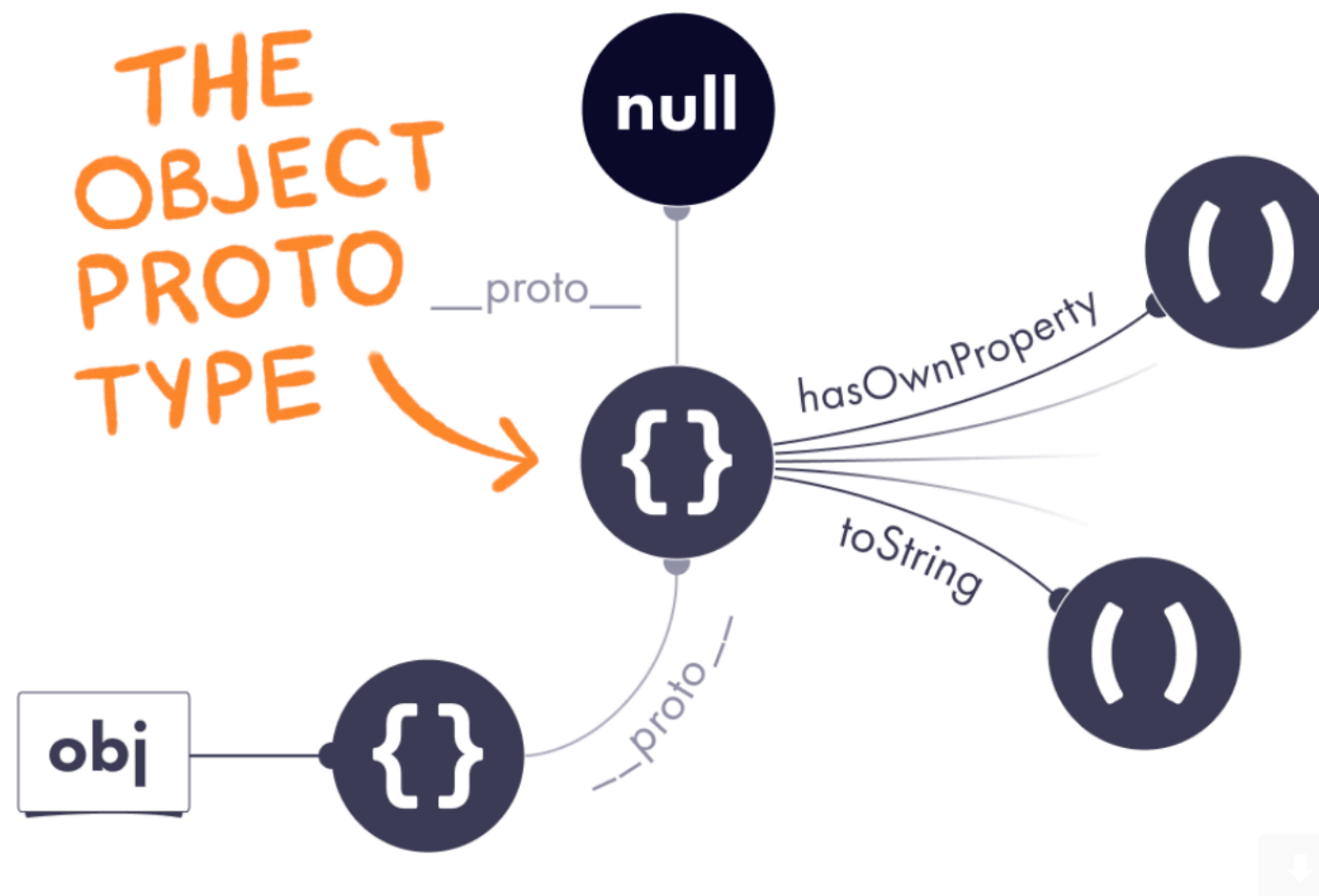
Try running this in your browser's console:

```
let obj = {};  
console.log(obj.__proto__); // Play with it!
```

Surprisingly, `obj.__proto__` is not `null` or `undefined`! Instead, you'll see a curious object with a bunch of properties, including `hasOwnProperty`.
(The “built-in” method we learned earlier turned out to be a regular property!)

The object prototype

We're going to call that special object the Object Prototype:



At first, this might be a bit mindblowing. Let that sink in. All this time we were thinking that `{ }` creates an “empty” object. But it’s not so empty, after all! It has a hidden `__proto__` wire that points at the Object Prototype by default.

The object prototype

This explains why the JavaScript objects seem to have “built-in” properties:

```
let human = {  
  teeth: 32  
};  
console.log(human.hasOwnProperty); // function  
hasOwnProperty() { }  
console.log(human.toString); // function toString() { }
```

These “built-in” properties are nothing more than normal properties that exist on the Object Prototype. Our object’s prototype is the Object Prototype, which is why we can access them. (Their implementations are inside the JS engine.)

Object without prototype

We've just learned that all objects created with the `{ }` syntax have the special `__proto__` wire set to a default Object Prototype. But we also know that we can customize the `__proto__`. You might wonder: can we set it to `null`?

```
let weirdo = {  
  __proto__: null  
};
```

The answer is yes — this will produce an object that truly doesn't have a prototype, at all. As a result, it doesn't even have built-in object methods:

```
console.log(weirdo.hasOwnProperty); // undefined  
console.log(weirdo.toString); // undefined
```

You won't often want to create objects like this, if at all. However, the Object Prototype itself is exactly such an object. It is an object with no prototype.

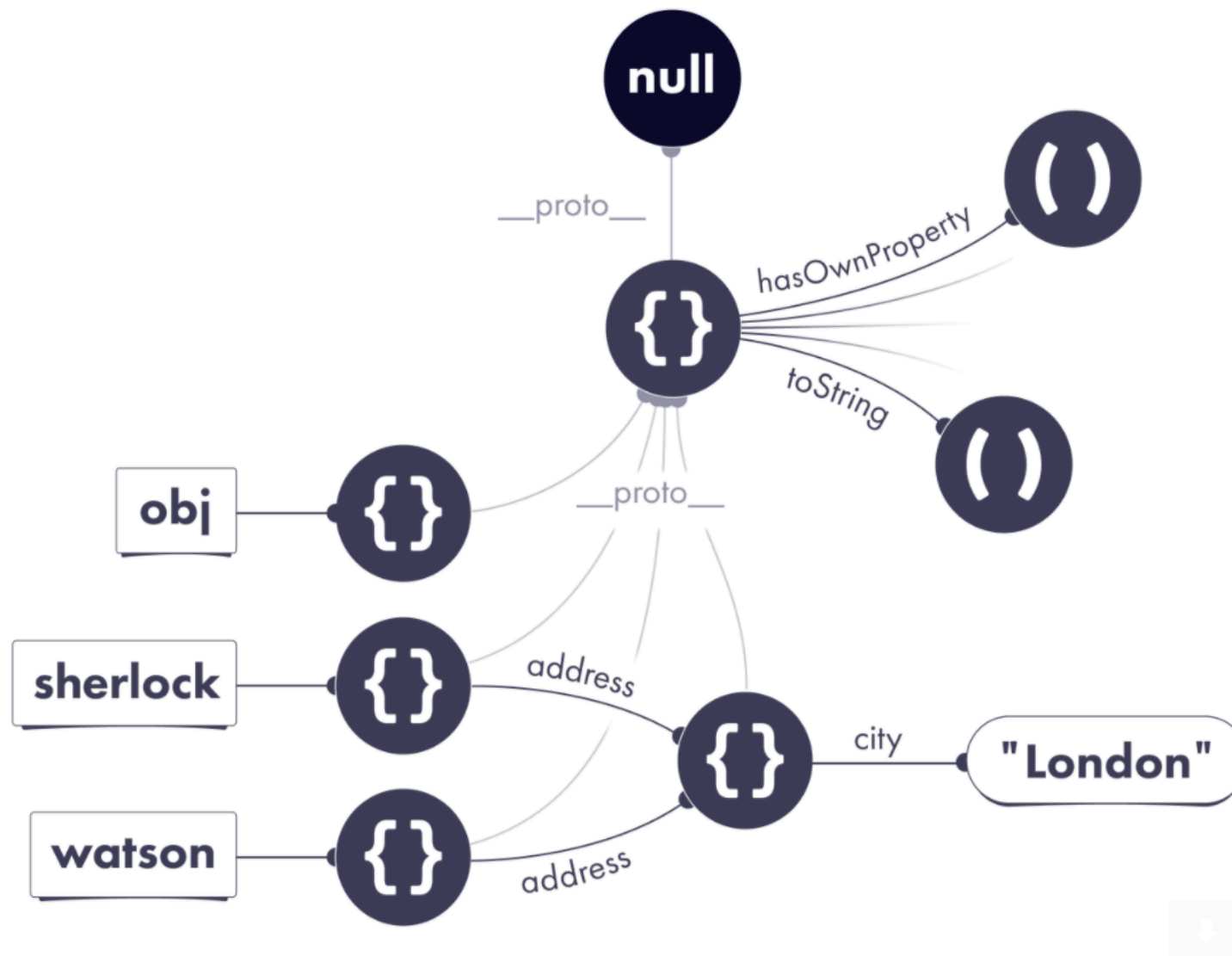
Polluting the prototype

Now we know that all JavaScript objects get the same prototype by default.

Let's illustrate our past example with a new level of detail:

```
let sherlock = {  
  address: {  
    city: 'London'  
  }  
};  
let watson = {  
  address: sherlock.address  
};
```

Polluting the prototype



This picture gives us an interesting insight. If JavaScript searches for missing properties on the prototype, and most objects share the same prototype, can we make new properties “appear” on all objects by mutating that prototype?

The answer is yes!

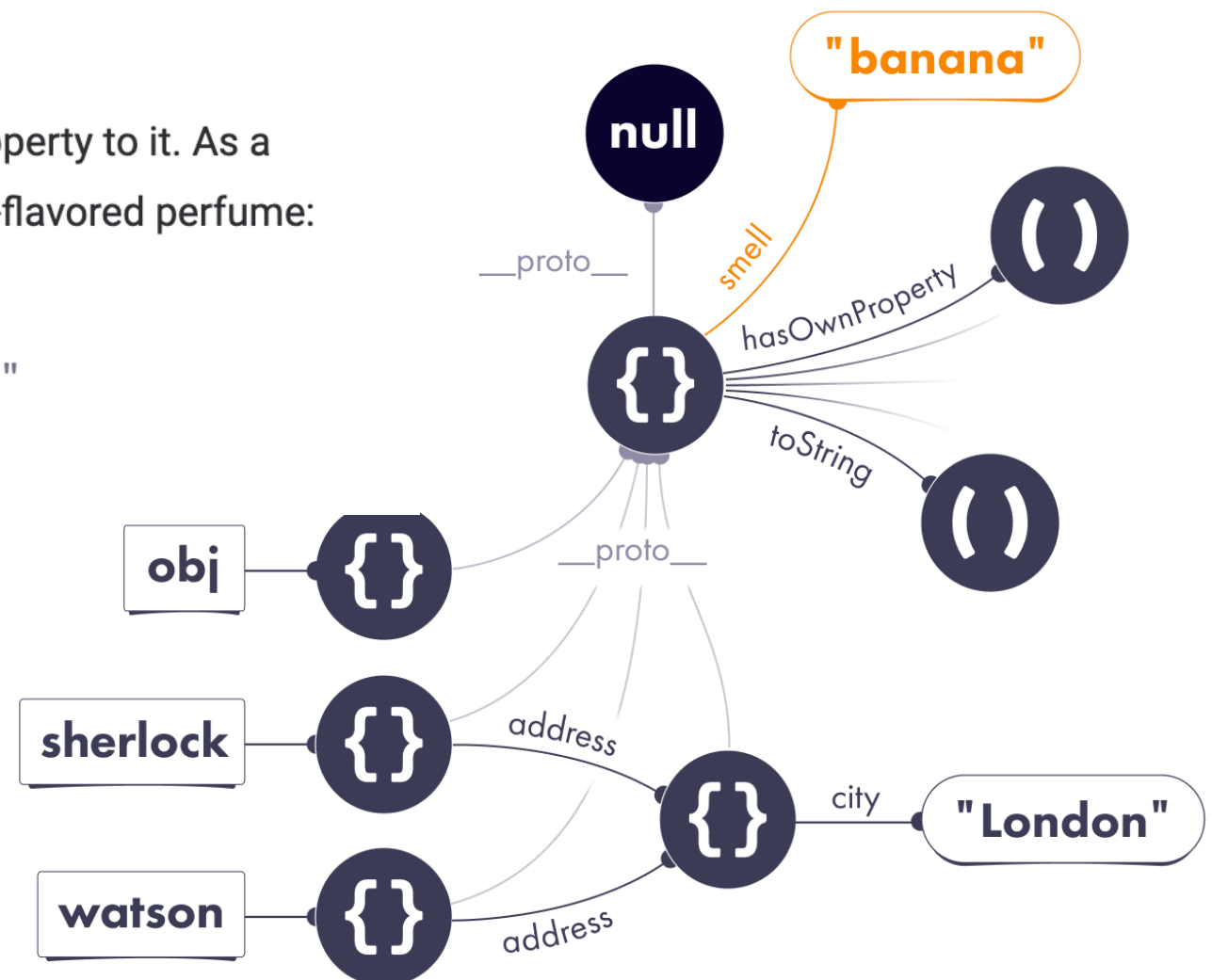
Polluting the prototype

Let's add these two lines of code:

```
let obj = {};  
obj.__proto__.smell = 'banana';
```

We mutated the Object Prototype by adding a `smell` property to it. As a result, both detectives now appear to be using a banana-flavored perfume:

```
console.log(sherlock.smell); // "banana"  
console.log(watson.smell); // "banana"
```



Why it matters?

You might be wondering: why care about prototypes at all? Will you use them much? In practice, you probably won't use them directly. Don't get into the habit of writing `__proto__`. These examples only illustrated the mechanics. (In fact, even using the `__proto__` syntax directly itself is [discouraged](#).)

Prototypes are a bit unusual, and most people and frameworks never really fully embraced them as a paradigm. Instead, people often used prototypes as mere building blocks for a traditional “class inheritance” model that's popular in other programming languages. In fact, it was so common that JavaScript added a class syntax as a convention that “hides” prototypes out of sight.

Still, you will notice prototypes hiding “beneath the surface” of classes and other JavaScript features. For example, here is a [snippet](#) of a JavaScript class rewritten with `__proto__` to demonstrate what's happening under the hood.

Personally, I don't use a lot of classes in my daily coding, and I rarely deal with prototypes directly either. However, it helps to know how those features build on each other, and what happens when I read or set a property on an object.

Recap

- When reading `obj.prop`, if `obj` doesn't have a `prop` property, JavaScript will look for `obj.__proto__.prop`, then it will look for `obj.__proto__.__proto__.prop`, and so on, until it either finds our property or reaches the end of the prototype chain.
- When writing to `obj.prop`, JavaScript will usually write to the object directly instead of traversing the prototype chain.
- We can use `obj.hasOwnProperty('prop')` to determine whether our object has an *own* property called `prop`. In other words, it means there is a property wire called `prop` attached to that object directly.
- We can “pollute” a prototype shared by many objects by mutating it. We can even do this to the Object Prototype — the default prototype for `{ }` objects! But we shouldn't do that unless we're pranking our colleagues.
- You probably won't use prototypes much directly in practice. However, they are fundamental to how JavaScript objects work, so it is handy to understand their underlying mechanics. Some advanced JavaScript features, including classes, can be expressed in terms of prototypes.

Home work

Create an **object** calculator with three methods:

- `read()` prompts(using `prompt` method, twice, for each value) for two values and saves them as object properties.
- `sum()` returns the sum of saved values.
- `multiply()` multiplies saved values and returns the result.

Learning Resources

1. This: <http://javascriptissexy.com/understand-javascripts-this-with-clarity-and-master-it/>
2. Object method (this): <https://javascript.info/object-methods>
3. This: https://www.w3schools.com/js/js_this.asp
4. Prototypes: <https://javascript.info/prototype-inheritance>