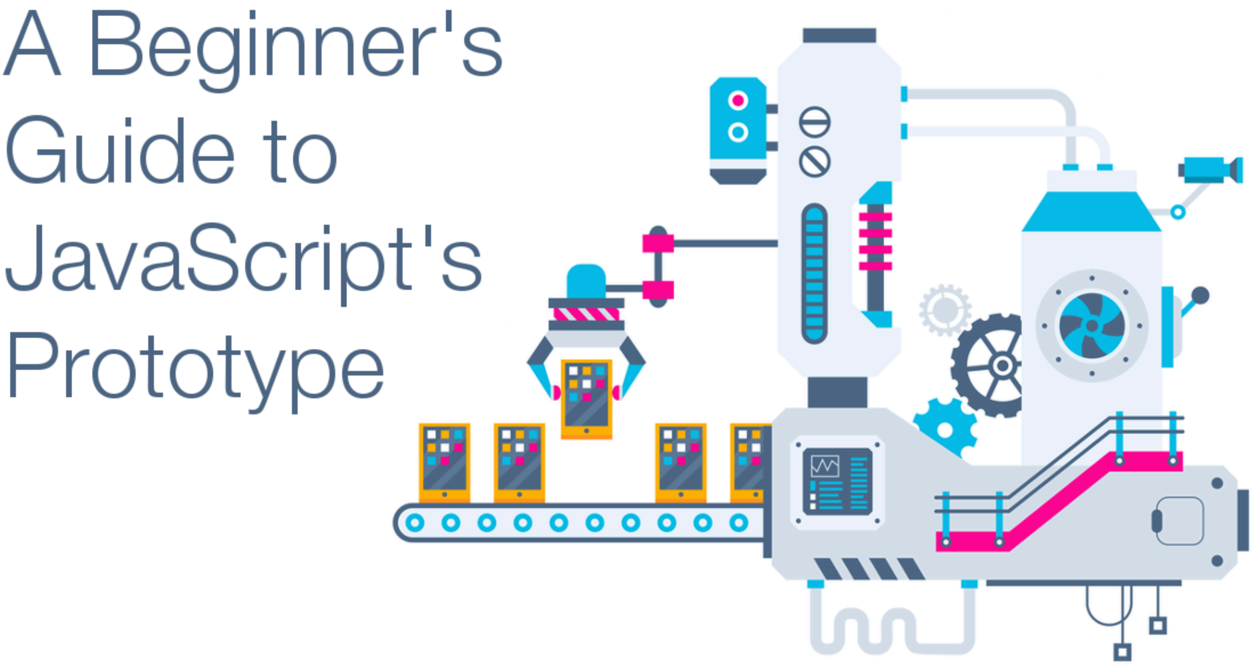
A Beginner’s Guide to JavaScript’s Prototype



You can't get very far in JavaScript without dealing with objects. They're foundational to almost every aspect of the JavaScript programming language. In fact, learning how to create objects is probably one of the first things you studied when you were starting out. With that said, in order to most effectively learn about prototypes in JavaScript, we're going to channel our inner Jr. developer and go back to the basics.

Objects are key/value pairs. The most common way to create an object is with curly braces `{}` and you add properties and methods to an object using dot notation.

```
let animal = {}
animal.name = 'Leo'
animal.energy = 10

animal.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

animal.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}

animal.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}
```

Simple. Now odds are in our application we'll need to create more than one animal. Naturally the next step for this would be to encapsulate that logic inside of a function that we can invoke whenever we needed to create a new animal. We'll call this pattern `Functional Instantiation` and we'll call the function itself a "constructor function" since it's responsible for "constructing" a new object.

Functional Instantiation

```
function Animal (name, energy) {
  let animal = {}
  animal.name = name
  animal.energy = energy

  animal.eat = function (amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
  }

  animal.sleep = function (length) {
    console.log(`${this.name} is sleeping.`)
    this.energy += length
  }

  animal.play = function (length) {
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }

  return animal
}

const leo = Animal('Leo', 7)
const snoop = Animal('Snoop', 10)
```

`"I thought this was an Advanced JavaScript course...?" - Your brain`

It is. We'll get there.

Now whenever we want to create a new animal (or more broadly speaking a new "instance"), all we have to do is invoke our `Animal` function, passing it the animal's `name` and `energy` level. This works great and it's incredibly simple. However, can you spot any weaknesses with this pattern? The biggest and the one we'll attempt to solve has to do with the three methods

- `eat`, `sleep`, and `play`. Each of those methods are not only dynamic, but they're also completely generic. What that means is that there's no reason to re-create those methods as we're currently doing whenever we create a new animal. We're just wasting memory and making each animal object bigger than it needs to be. Can you think of a solution? What if instead of re-creating those methods every time we create a new animal, we move them to their own object then we can have each animal reference that object? We can call this pattern `Functional Instantiation with Shared Methods`, wordy but descriptive ?♂️.

Functional Instantiation with Shared Methods

```
const animalMethods = {
  eat(amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
  },
  sleep(length) {
    console.log(`${this.name} is sleeping.`)
    this.energy += length
  },
  play(length) {
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }
}

function Animal (name, energy) {
  let animal = {}
  animal.name = name
  animal.energy = energy
  animal.eat = animalMethods.eat
  animal.sleep = animalMethods.sleep
  animal.play = animalMethods.play

  return animal
}

const leo = Animal('Leo', 7)
const snoop = Animal('Snoop', 10)
```

By moving the shared methods to their own object and referencing that object inside of our `Animal` function, we've now solved the problem of memory waste and overly large animal objects.

Object.create

Let's improve our example once again by using `Object.create`. Simply put, **Object.create allows you to create an object which will delegate to another object on failed lookups**. Put differently, Object.create allows you to create an object and whenever there's a failed property lookup on that object, it can consult another object to see if that other object has the property. That was a lot of words. Let's see some code.

```
const parent = {
  name: 'Stacey',
  age: 35,
  heritage: 'Irish'
}

const child = Object.create(parent)
child.name = 'Ryan'
child.age = 7

console.log(child.name) // Ryan
console.log(child.age) // 7
console.log(child.heritage) // Irish
```

So in the example above, because `child` was created with `Object.create(parent)`, whenever there's a failed property lookup on `child`, JavaScript will delegate that lookup to the `parent` object. What that means is that even though `child` doesn't have a `heritage` property, `parent` does so when you log `child.heritage` you'll get the `parent`'s heritage which was `Irish`.

Now with `Object.create` in our tool shed, how can we use it in order to simplify our `Animal` code from earlier? Well, instead of adding all the shared methods to the animal one by one like we're doing now, we can use

Object.create to delegate to the `animalMethods` object instead. To sound really smart, let's call this one `Functional Instantiation with Shared Methods and Object.create` ?

Functional Instantiation with Shared Methods and Object.create

```
const animalMethods = {
  eat(amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
  },
  sleep(length) {
    console.log(`${this.name} is sleeping.`)
    this.energy += length
  },
  play(length) {
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }
}

function Animal (name, energy) {
  let animal = Object.create(animalMethods)
  animal.name = name
  animal.energy = energy

  return animal
}

const leo = Animal('Leo', 7)
const snoop = Animal('Snoop', 10)

leo.eat(10)
snoop.play(5)
```

? So now when we call `leo.eat`, JavaScript will look for the `eat` method on the `leo` object. That lookup will fail, then, because of Object.create, it'll delegate to the `animalMethods` object which is where it'll find `eat`.

So far, so good. There are still some improvements we can make though. It seems just a tad "hacky" to have to manage a separate object (`animalMethods`) in order to share methods across instances. That seems like a common feature that you'd want to be implemented into the language itself. Turns out it is and it's the whole reason you're here - `prototype`.

So what exactly is `prototype` in JavaScript? Well, simply put, every function in JavaScript has a `prototype` property that references an object. Anticlimactic, right? Test it out for yourself.

```
function doThing () {}
console.log(doThing.prototype) // {}
```

What if instead of creating a separate object to manage our methods (like we're doing with `animalMethods`), we just put each of those methods on the `Animal` function's prototype? Then all we would have to do is instead of using Object.create to delegate to `animalMethods`, we could use it to delegate to `Animal.prototype`. We'll call this pattern `Prototypal Instantiation`.

Prototypal Instantiation

```
function Animal (name, energy) {
  let animal = Object.create(Animal.prototype)
  animal.name = name
  animal.energy = energy

  return animal
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}
```

```
Animal.prototype.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}

const leo = Animal('Leo', 7)
const snoop = Animal('Snoop', 10)

leo.eat(10)
snoop.play(5)
```

??? Hopefully you just had a big "aha" moment. Again, `prototype` is just a property that every function in JavaScript has and, as we saw above, it allows us to share methods across all instances of a function. All our functionality is still the same but now instead of having to manage a separate object for all the methods, we can just use another object that comes built into the `Animal` function itself, `Animal.prototype`.

Let's. Go. Deeper.

At this point we know three things:

1. How to create a constructor function.
2. How to add methods to the constructor function's prototype.
3. How to use Object.create to delegate failed lookups to the function's prototype.

Those three tasks seem pretty foundational to any programming language. Is JavaScript really that bad that there's no easier, "built in" way to accomplish the same thing? As you can probably guess at this point there is, and it's by using the `new` keyword.

What's nice about the slow, methodical approach we took to get here is you'll now have a deep understanding of exactly what the `new` keyword in

JavaScript is doing under the hood.

Looking back at our `Animal` constructor, the two most important parts were creating the object and returning it. Without creating the object with `Object.create`, we wouldn't be able to delegate to the function's prototype on failed lookups. Without the `return` statement, we wouldn't ever get back the created object.

```
function Animal (name, energy) {
  let animal = Object.create(Animal.prototype)
  animal.name = name
  animal.energy = energy

  return animal
}
```

Here's the cool thing about `new` - when you invoke a function using the `new` keyword, those two lines are done for you implicitly ("under the hood") and the object that is created is called `this`.

Using comments to show what happens under the hood and assuming the `Animal` constructor is called with the `new` keyword, it can be re-written as this.

```
function Animal (name, energy) {
  // const this = Object.create(Animal.prototype)

  this.name = name
  this.energy = energy

  // return this
}

const leo = new Animal('Leo', 7)
const snoop = new Animal('Snoop', 10)
```

and without the "under the hood" comments

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}

Animal.prototype.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}

const leo = new Animal('Leo', 7)
const snoop = new Animal('Snoop', 10)
```

Again the reason this works and that the `this` object is created for us is because we called the constructor function with the `new` keyword. If you leave off `new` when you invoke the function, that `this` object never gets created nor does it get implicitly returned. We can see the issue with this in the example below.

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

const leo = Animal('Leo', 7)
console.log(leo) // undefined
```

The name for this pattern is `Pseudoclassical Instantiation`.

If JavaScript isn't your first programming language, you might be getting a little restless.

"WTF this dude just re-created a crappier version of a Class" - You

For those unfamiliar, a Class allows you to create a blueprint for an object. Then whenever you create an instance of that Class, you get an object with the properties and methods defined in the blueprint.

Sound familiar? That's basically what we did with our `Animal` constructor function above. However, instead of using the `class` keyword, we just used a regular old JavaScript function to re-create the same functionality. Granted, it took a little extra work as well as some knowledge about what happens "under the hood" of JavaScript but the results are the same.

Here's the good news. JavaScript isn't a dead language. It's constantly being improved and added to by the TC-39 committee. What that means is that even though the initial version of JavaScript didn't support classes, there's no reason they can't be added to the official specification. In fact, that's exactly what the TC-39 committee did. In 2015, EcmaScript (the official JavaScript specification) 6 was released with support for Classes and the `class` keyword. Let's see how our `Animal` constructor function above would look like with the new class syntax.

```
class Animal {
  constructor(name, energy) {
    this.name = name
    this.energy = energy
  }
  eat(amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
  }
  sleep(length) {
    console.log(`${this.name} is sleeping.`)
    this.energy += length
  }
  play(length) {
```

```
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }
}

const leo = new Animal('Leo', 7)
const snoop = new Animal('Snoop', 10)
```

Pretty clean, right?

So if this is the new way to create classes, why did we spend so much time going over the old way? The reason for that is because the new way (with the `class` keyword) is primarily just "syntactical sugar" over the existing way we've called the pseudoclassical pattern. In order to *fully* understand the convenience syntax of ES6 classes, you first must understand the pseudoclassical pattern.

At this point we've covered the fundamentals of JavaScript's prototype. The rest of this post will be dedicated to understanding other "good to know" topics related to it. In another post we'll look at how we can take these fundamentals and use them to understand how inheritance works in JavaScript.

Array Methods

We talked in depth above about how if you want to share methods across instances of a class, you should stick those methods on the class' (or function's) prototype. We can see this same pattern demonstrated if we look at the `Array` class. Historically you've probably created your arrays like this

```
const friends = []
```

Turns out that's just sugar over creating a `new` instance of the `Array` class.

```
const friendsWithSugar = []

const friendsWithoutSugar = new Array()
```

One thing you might have never thought about is how does every instance of an array have all of those built in methods (`splice`, `slice`, `pop`, etc)?

Well as you now know, it's because those methods live on `Array.prototype` and when you create a new instance of `Array`, you use the `new` keyword which sets up that delegation to `Array.prototype` on failed lookups.

We can see all the array's methods by simply logging `Array.prototype`.

```
console.log(Array.prototype)

/*
  concat: ƒn concat()
  constructor: ƒn Array()
  copyWithin: ƒn copyWithin()
  entries: ƒn entries()
  every: ƒn every()
  fill: ƒn fill()
  filter: ƒn filter()
  find: ƒn find()
  findIndex: ƒn findIndex()
  forEach: ƒn forEach()
  includes: ƒn includes()
  indexOf: ƒn indexOf()
  join: ƒn join()
  keys: ƒn keys()
  lastIndexOf: ƒn lastIndexOf()
  length: 0n
```

```
  map: ƒn map()
  pop: ƒn pop()
  push: ƒn push()
  reduce: ƒn reduce()
  reduceRight: ƒn reduceRight()
  reverse: ƒn reverse()
  shift: ƒn shift()
  slice: ƒn slice()
  some: ƒn some()
  sort: ƒn sort()
  splice: ƒn splice()
  toLocaleString: ƒn toLocaleString()
  toString: ƒn toString()
  unshift: ƒn unshift()
  values: ƒn values()
*/
```

The exact same logic exists for Objects as well. Alls object will delegate to `Object.prototype` on failed lookups which is why all objects have methods like `toString` and `hasOwnProperty`.

Static Methods

Up until this point we've covered the why and how of sharing methods between instances of a Class. However, what if we had a method that was important to the Class, but didn't need to be shared across instances? For example, what if we had a function that took in an array of `Animal` instances and determined which one needed to be fed next? We'll call it `nextToEat`.

```
function nextToEat (animals) {
  const sortedByLeastEnergy = animals.sort((a,b) => {
    return a.energy - b.energy
  })

  return sortedByLeastEnergy[0].name
}
```

It doesn't make sense to have `nextToEat` live on `Animal.prototype` since we don't want to share it amongst all instances. Instead, we can think of it as more of a helper method. So if `nextToEat` shouldn't live on `Animal.prototype`, where should we put it? Well the obvious answer is we could just stick `nextToEat` in the same scope as our `Animal` class then reference it when we need it as we normally would.

```
class Animal {
  constructor(name, energy) {
    this.name = name
    this.energy = energy
  }
  eat(amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
  }
  sleep(length) {
    console.log(`${this.name} is sleeping.`)
    this.energy += length
  }
  play(length) {
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }
}

function nextToEat (animals) {
  const sortedByLeastEnergy = animals.sort((a,b) => {
    return a.energy - b.energy
  })

  return sortedByLeastEnergy[0].name
}

const leo = new Animal('Leo', 7)
const snoop = new Animal('Snoop', 10)

console.log(nextToEat([leo, snoop])) // Leo
```

Now this works, but there's a better way.

Whenever you have a method that is specific to a class itself, but doesn't need to be shared across instances of that class, you can add it as a `static` property of the class.

```
class Animal {
  constructor(name, energy) {
    this.name = name
    this.energy = energy
  }
  eat(amount) {
    console.log(`${this.name} is eating.`)
    this.energy += amount
  }
  sleep(length) {
    console.log(`${this.name} is sleeping.`)
    this.energy += length
  }
  play(length) {
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }
  static nextToEat(animals) {
    const sortedByLeastEnergy = animals.sort((a,b) => {
      return a.energy - b.energy
    })

    return sortedByLeastEnergy[0].name
  }
}
```

Now, because we added `nextToEat` as a `static` property on the class, it lives on the `Animal` class itself (not its prototype) and can be accessed using `Animal.nextToEat`.

```
const leo = new Animal('Leo', 7)
const snoop = new Animal('Snoop', 10)

console.log(Animal.nextToEat([leo, snoop])) // Leo
```

Because we've followed a similar pattern throughout this post, let's take a look at how we would accomplish this same thing using ES5. In the example above we saw how using the `static` keyword would put the method directly onto the class itself. With ES5, this same pattern is as simple as just manually adding the method to the function object.

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}

Animal.prototype.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}

Animal.nextToEat = function (nextToEat) {
  const sortedByLeastEnergy = animals.sort((a,b) => {
    return a.energy - b.energy
  })

  return sortedByLeastEnergy[0].name
}

const leo = new Animal('Leo', 7)
const snoop = new Animal('Snoop', 10)

console.log(Animal.nextToEat([leo, snoop])) // Leo
```

Getting the prototype of an object

Regardless of whichever pattern you used to create an object, getting that object's prototype can be accomplished using the `Object.getPrototypeOf` method.

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}

Animal.prototype.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}

const leo = new Animal('Leo', 7)
const prototype = Object.getPrototypeOf(leo)

console.log(prototype)
// {constructor: ƒ, eat: ƒ, sleep: ƒ, play: ƒ}

prototype === Animal.prototype // true
```

There are two important takeaways from the code above.

First, you'll notice that `proto` is an object with 4 methods, `constructor`, `eat`, `sleep`, and `play`. That makes sense. We used `getPrototypeOf` passing in the instance, `leo` getting back that instances' prototype, which is where all of our methods are living. This tells us one more thing about `prototype` as well that we haven't talked about yet. By default, the `prototype` object will have a `constructor` property which points to the original function or the class that the instance was created from. What this also means is that because JavaScript puts a `constructor` property on the prototype by default, any instances will be able to access their constructor via `instance.constructor`.

The second important takeaway from above is that `Object.getPrototypeOf(leo) === Animal.prototype`. That makes sense as well. The `Animal` constructor function has a prototype property where we can share methods across all instances and `getPrototypeOf` allows us to see the prototype of the instance itself.

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

const leo = new Animal('Leo', 7)
console.log(leo.constructor) // Logs the constructor function
```

To tie in what we talked about earlier with `Object.create`, the reason this works is because any instances of `Animal` are going to delegate to `Animal.prototype` on failed lookups. So when you try to access `leo.constructor`, `leo` doesn't have a `constructor` property so it will delegate that lookup to `Animal.prototype` which indeed does have a `constructor` property. If this paragraph didn't make sense, go back and read about `Object.create` above.

You may have seen **proto** used before to get an instances' prototype. That's a relic of the past. Instead, use **Object.getPrototypeOf(instance)** as we saw above.

Determining if a property lives on the prototype

There are certain cases where you need to know if a property lives on the instance itself or if it lives on the prototype the object delegates to. We can see this in action by looping over our `leo` object we've been creating. Let's say the goal was the loop over `leo` and log all of its keys and values. Using a `for in` loop, that would probably look like this.

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}

Animal.prototype.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}

const leo = new Animal('Leo', 7)

for(let key in leo) {
  console.log(`Key: ${key}. Value: ${leo[key]}`)
}
```

What would you expect to see? Most likely, it was something like this -

```
Key: name. Value: Leo
Key: energy. Value: 7
```

However, what you saw if you ran the code was this -

```
Key: name. Value: Leo
Key: energy. Value: 7
Key: eat. Value: function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}
Key: sleep. Value: function (length) {
  console.log(`${this.name} is sleeping.`)
```

```
    this.energy += length
  }
  Key: play. Value: function (length) {
    console.log(`${this.name} is playing.`)
    this.energy -= length
  }
```

Why is that? Well a `for in` loop is going to loop over all of the **enumerable properties** on both the object itself as well as the prototype it delegates to. Because by default any property you add to the function's prototype is enumerable, we see not only `name` and `energy`, but we also see all the methods on the prototype - `eat`, `sleep`, and `play`. To fix this, we either need to specify that all of the prototype methods are non-enumerable **or** we need a way to only console.log if the property is on the `leo` object itself and not the prototype that `leo` delegates to on failed lookups. This is where `hasOwnProperty` can help us out.

`hasOwnProperty` is a property on every object that returns a boolean indicating whether the object has the specified property as its own property rather than on the prototype the object delegates to. That's exactly what we need. Now with this new knowledge we can modify our code to take advantage of `hasOwnProperty` inside of our `for in` loop.

```
...

const leo = new Animal('Leo', 7)

for(let key in leo) {
  if (leo.hasOwnProperty(key)) {
    console.log(`Key: ${key}. Value: ${leo[key]}`)
  }
}
```

And now what we see are only the properties that are on the `leo` object itself rather than on the prototype `leo` delegates to as well.

```
Key: name. Value: Leo
Key: energy. Value: 7
```

If you're still a tad confused about `hasOwnProperty`, here is some code that may clear it up.

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

Animal.prototype.eat = function (amount) {
  console.log(`${this.name} is eating.`)
  this.energy += amount
}

Animal.prototype.sleep = function (length) {
  console.log(`${this.name} is sleeping.`)
  this.energy += length
}

Animal.prototype.play = function (length) {
  console.log(`${this.name} is playing.`)
  this.energy -= length
}

const leo = new Animal('Leo', 7)

leo.hasOwnProperty('name') // true
leo.hasOwnProperty('energy') // true
leo.hasOwnProperty('eat') // false
leo.hasOwnProperty('sleep') // false
leo.hasOwnProperty('play') // false
```

Check if an object is an instance of a Class

Sometimes you want to know whether an object is an instance of a specific class. To do this, you can use the `instanceof` operator. The use case is pretty straight forward but the actual syntax is a bit weird if you've never seen it before. It works like this

```
object instanceof Class
```

The statement above will return true if `object` is an instance of `Class` and false if it isn't. Going back to our `Animal` example we'd have something like this.

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}

function User () {}

const leo = new Animal('Leo', 7)

leo instanceof Animal // true
leo instanceof User // false
```

The way that `instanceof` works is it checks for the presence of `constructor.prototype` in the object's prototype chain. In the example above, `leo instanceof Animal` is `true` because `Object.getPrototypeOf(leo) === Animal.prototype`. In addition, `leo instanceof User` is `false` because `Object.getPrototypeOf(leo) !== User.prototype`.

Creating new agnostic constructor functions

Can you spot the error in the code below?

```
function Animal (name, energy) {
  this.name = name
  this.energy = energy
}
```

```
const leo = Animal('Leo', 7)
```

Even seasoned JavaScript developers will sometimes get tripped up on the example above. Because we're using the `pseudoclassical pattern` that we learned about earlier, when the `Animal` constructor function is invoked, we need to make sure we invoke it with the `new` keyword. If we don't, then the `this` keyword won't be created and it also won't be implicitly returned.

As a refresher, the commented out lines are what happens behind the scenes when you use the `new` keyword on a function.

```
function Animal (name, energy) {
  // const this = Object.create(Animal.prototype)

  this.name = name
  this.energy = energy

  // return this
}
```

This seems like too important of a detail to leave up to other developers to remember. Assuming we're working on a team with other developers, is there a way we could ensure that our `Animal` constructor is always invoked with the `new` keyword? Turns out there is and it's by using the `instanceof` operator we learned about previously.

If the constructor was called with the `new` keyword, then `this` inside of the body of the constructor will be an `instanceof` the constructor function itself. That was a lot of big words. Here's some code.

```
function Animal (name, energy) {
  if (this instanceof Animal === false) {
    console.warn('Forgot to call Animal with the new keyword')
  }
```

```
  this.name = name
  this.energy = energy
}
```

Now instead of just logging a warning to the consumer of the function, what if we re-invoke the function, but with the `new` keyword this time?

```
function Animal (name, energy) {
  if (this instanceof Animal === false) {
    return new Animal(name, energy)
  }

  this.name = name
  this.energy = energy
}
```

Now regardless of if `Animal` is invoked with the `new` keyword, it'll still work properly.

Re-creating Object.create

Throughout this post we've relied heavily upon `Object.create` in order to create objects which delegate to the constructor function's prototype. At this point, you should know how to use `Object.create` inside of your code but one thing that you might not have thought of is how `Object.create` actually works under the hood. In order for you to **really** understand how `Object.create` works, we're going to re-create it ourselves. First, what do we know about how `Object.create` works?

1. It takes in an argument that is an object.
2. It creates an object that delegates to the argument object on failed lookups.
3. It returns the new created object.

Let's start off with #1.

```
Object.create = function (objToDelegateTo) {

}
```

Simple enough.

Now #2 - we need to create an object that will delegate to the argument object on failed lookups. This one is a little more tricky. To do this, we'll use our knowledge of how the `new` keyword and prototypes work in JavaScript. First, inside the body of our `Object.create` implementation, we'll create an empty function. Then, we'll set the prototype of that empty function equal to the argument object. Then, in order to create a new object, we'll invoke our empty function using the `new` keyword. If we return that newly created object, that'll finish #3 as well.

```
Object.create = function (objToDelegateTo) {
  function Fn(){}
  Fn.prototype = objToDelegateTo
  return new Fn()
}
```

Wild. Let's walk through it.

When we create a new function, `Fn` in the code above, it comes with a `prototype` property. When we invoke it with the `new` keyword, we know what we'll get back is an object that will delegate to the function's prototype on failed lookups. If we override the function's prototype, then we can decide which object to delegate to on failed lookups. So in our example above, we override `Fn`'s prototype with the object that was passed in when `Object.create` was invoked which we call `objToDelegateTo`.

Note that we're only supporting a single argument to Object.create. The official implementation also supports a second, optional argument which allow you to add more properties to the created object.

Arrow Functions

Arrow functions don't have their own `this` keyword. As a result, arrow functions can't be constructor functions and if you try to invoke an arrow function with the `new` keyword, it'll throw an error.

```
const Animal = () => {}

const leo = new Animal() // Error: Animal is not a constructor
```

Also, because we demonstrated above that the pseudoclassical pattern can't be used with arrow functions, arrow functions also don't have a `prototype` property.

```
const Animal = () => {}
console.log(Animal.prototype) // undefined
```