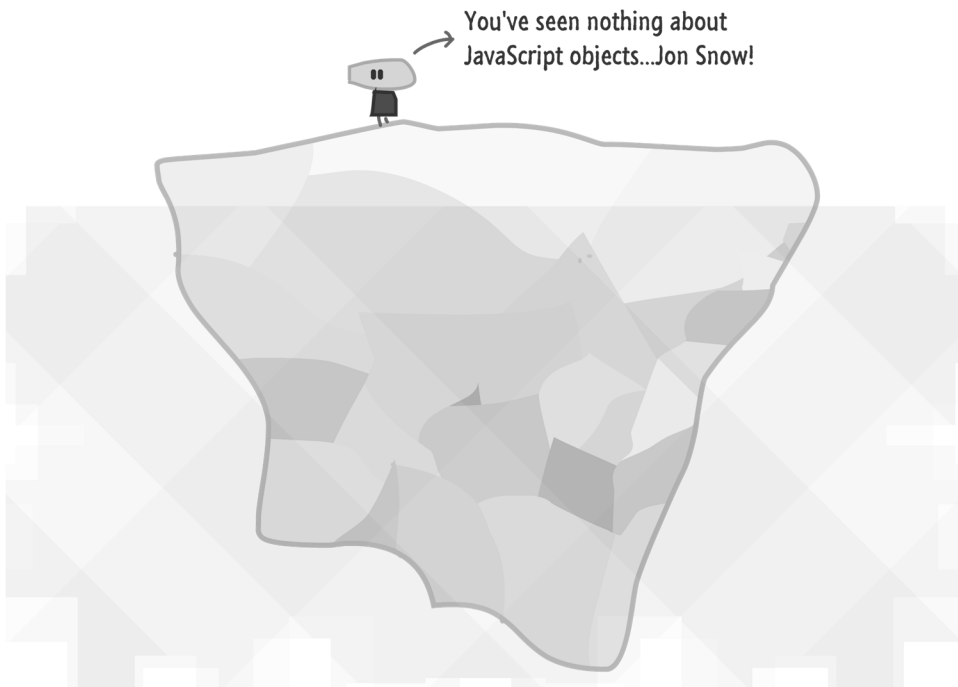


# Chapter 15: A Deeper Look at Objects

In the Introduction to Objects schtuff in **Chapter 10**, I provided a very high level overview of what objects in JavaScript are and how to think about them. That was good enough to cover the basics and some of the built-in types, but we need to go a little deeper. In this chapter, we will make that earlier chapter seem like the tip of a ginormous iceberg:



What we are going to do here is have a re-look at objects in greater detail and touch on the more advanced topics such as using the `Object` object, creating your own custom objects, inheritance, prototypes, and the `this` keyword. If all that I've listed so far makes no sense, it will after you've reached the end of this chapter...I hope.

Onwards!

## Meet the Object

At the very bottom of the food chain, you have the `Object` type that lays the groundwork for both custom objects as well as built-in types like `Function`, `Array`, and `RegExp`. Pretty much everything except `null` and `undefined` are directly related to an `Object` or can become one as needed.

As you saw from the introduction to objects forever ago, the functionality `Object` brings to the table is pretty minimal. It allows you to specify a bunch of named key and value pairs that we lovingly call **properties**. This isn't all that different from what you see in other languages with data structures like hashtables, associative arrays, and dictionaries.

Anyway, all of this is pretty boring. Let's get to some of the more exciting stuff!

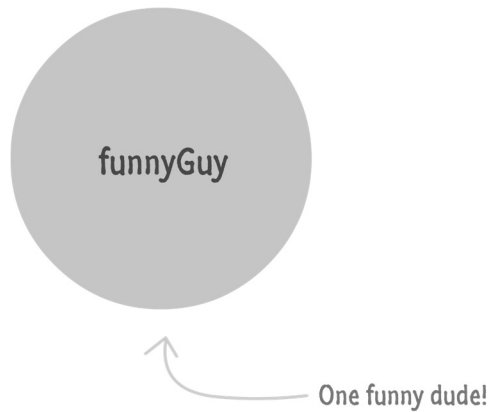
## Creating Objects

The way all the cool kids are creating objects these days is by using the funny-looking (yet compact) **object initializer** syntax:

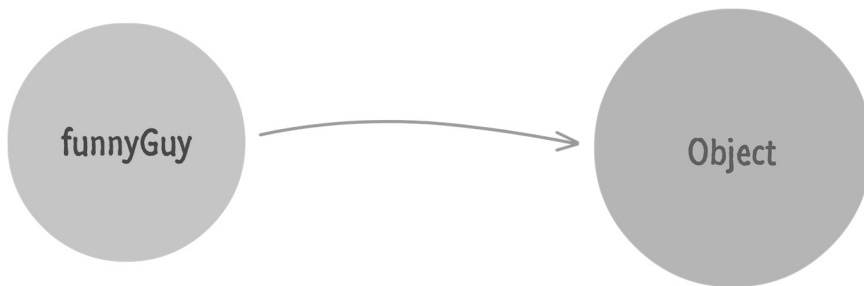
```
var funnyGuy = {};
```

That's right. Instead of typing in "`new Object()`" like you may have seen in old-timey books, you can just initialize your object by saying "`{}`". At the end of this line getting executed, you will have created an object called `funnyGuy` whose type is `Object`. If all of this makes sense so far, great!

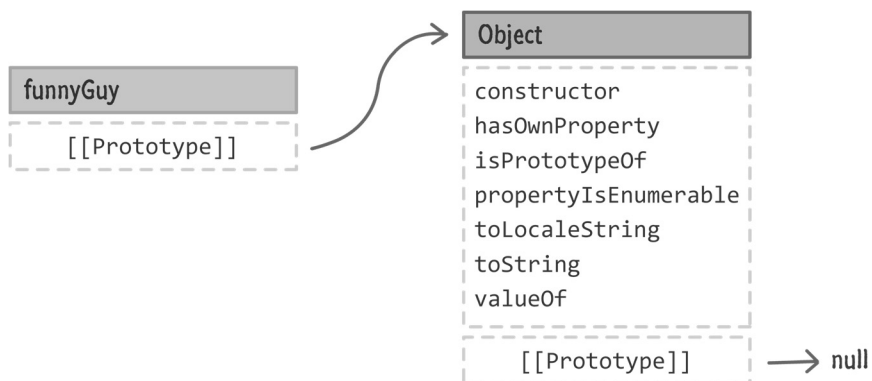
This `funnyGuy` object isn't as simple as it looks. Let's dive a little bit deeper and visualize what exactly is going on. On the surface, you just have the `funnyGuy` object:



If you back up and look more broadly at the `funnyGuy` object, you'll realize that it isn't alone here. Because it is an object, it has a connection to the main `Object` type that it derives from:



What this connection means is pretty significant. Let's add some more detail to what we have provided so far:

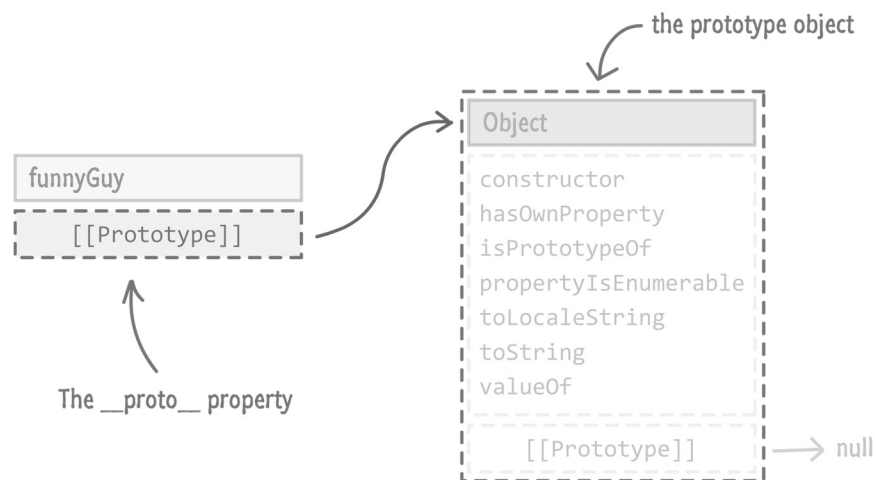


The `funnyGuy` object has no properties defined. That makes sense given what we specified in code (and can see in the diagram above):

```
var funnyGuy = {};
```

Our `funnyGuy` is simply an empty object. While there may not be any properties **we defined** on it, there is a special internal property that exists called `__proto__` and often represented as `[[Prototype]]` that points to our `Object`. If your mind just melted for a second, let me slow down and explain what is going on here.

What the `[[Prototype]]` property references is what is known as a prototype object:



A prototype object is the source that another object is based on. In our case, the `funnyGuy` object is created in the likeness of our `Object` type. What this means is best highlighted by looking at an example. We know that `funnyGuy` contains no properties of its own. Because it is "derived" from our `Object` type, you can access any properties the `Object` contains through `funnyGuy` itself.

For example, I can do something like this:

```
var funnyGuy = {};  
funnyGuy.toString(); // [object Object]
```

I am calling the `toString()` method on our `funnyGuy` object. Despite `funnyGuy` not having any property called `toString` on it, what gets returned isn't an error or `undefined`. You actually see the results of the `toString()` method having acted on our `funnyGuy` object.

Think about this in a different way. Your `funnyGuy` object is like a little kid. It doesn't have any money of its own to buy big and expensive things. What the `funnyGuy` object/kid does have is a parent with a credit card. Having access to this credit card allows the kid to go and buy things that he/she wouldn't otherwise be able to do. The `funnyGuy / Object` relationship is very much like a child / parent relationship where if the child doesn't have something, he / she can check with a parent next.

Getting back to our example and looking at it all one more time, what happened is this. Let's act this out:

Our JavaScript engine was like, Hey, `funnyGuy`! I'm going to call `toString()` on you.

The `funnyGuy` object replied with a, Yo...dawg. I don't know what you are talking about.

The JavaScript engine then said, Well, I'm going to check your prototype object and see if it contains a property called `toString()`.

A few milliseconds later, after finding the prototype object thanks to the `[[Prototype]]` property, our JavaScript engine says, Object, my old chum! Do you have a `toString()` property?

The Object quickly replies with a `Yep`. and calls the `toString` method.

This entire (very dramatic) interaction is part of what is known as a **prototype chain**. If an object doesn't have what you are looking for, the JavaScript engine will navigate to the next object as determined by the `[[Prototype]]` property and keep going until it reaches the very end. The very end is when you try to access the `[[Prototype]]` property on the `Object` itself. You can't go any further beyond `Object`, since that is as basic of a type as you can get. I highlight this in the diagrams by having your `Object`'s `[[Prototype]]` refer to `null`.

If you've ever heard of the term **inheritance** as it applies to programming before, what you've just seen is a simple example of it!

## Specifying Properties

I bet you didn't imagine that a single line of JavaScript would result in that much explanation, did you? Well, the nice thing is, I front loaded a lot of conceptual data on you. Hopefully that makes everything else that you see from here on out make a lot more sense.

Right now, we still just have an empty object:

```
var funnyGuy = {};
```

Let's specify some properties on it called `firstName` and `lastName`. As with all things in JavaScript, you have multiple ways of defining properties on an element. The way you've seen so far is by using the dot notation:

```
var funnyGuy = {};  
funnyGuy.firstName = "Conan";  
funnyGuy.lastName = "O'Brien";
```

Another approach involves using the square bracket syntax:

```
var funnyGuy = {};  
funnyGuy["firstName"] = "Conan";  
funnyGuy["lastName"] = "O'Brien";
```

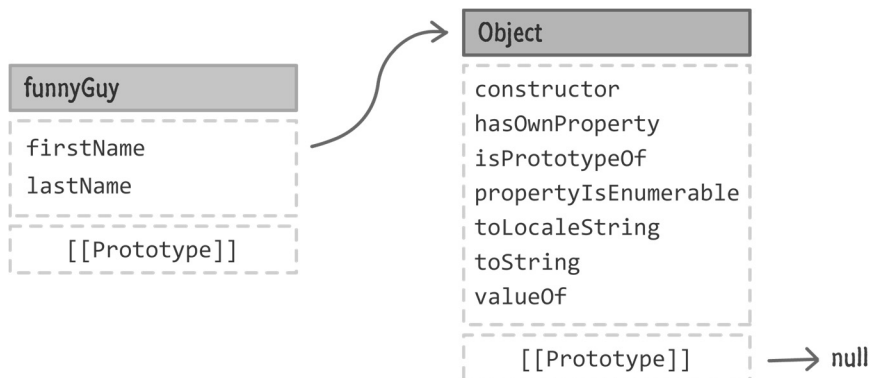
The final approach is by extending our object initializer syntax with the **literal notation** for declaring properties:

```
var funnyGuy = {  
  firstName: "Conan",  
  lastName: "O'Brien"  
};
```

There is no right or wrong approach in how you want to specify properties, so be aware of all three variants and go with one that works best for the situation you find yourself in. In general, I use...

1. The literal notation when I am specifying properties directly with a value.
2. The dot notation if I am specifying a property whose values are provided as part of an argument or expression.
3. The square bracket notation if the property name itself is something that is part of an argument or expression.

Regardless of which of the three approaches you used for specifying your properties, the end result is that your `funnyGuy` object will have these properties (and values) defined on itself:



All of this should be straightforward. Let's just do one more thing before we move on bigger and greener pastures. Let's create a method called **getName** on `funnyGuy` that will return the value of the `firstName` and `lastName` properties. I will just show what this looks like using the literal notation, for it is easy in the other approaches:

```
var funnyGuy = {
  firstName: "Conan",
  lastName: "O'Brien",

  getName: function () {
```

```
        return "Name is: " + this.firstName + " " + this.lastName;
    }
};
```

Our `getName` property's value is a function whose body simply returns a string that includes the value of our `firstName` and `lastName` properties. To call the `getName` property, ahem...method, this is all you have to do:

```
var funnyGuy = {
    firstName: "Conan",
    lastName: "O'Brien",

    getName: function () {
        return "Name is: " + this.firstName + " " + this.lastName;
    }
};

alert(funnyGuy.getName()); // displays "Name is Conan O'Brien"
```

Yep, that's all there is to declaring an object and setting properties on it. Indirectly, you learned a whole lot about what goes on behind the scenes when a simple object is created. You'll need all of this fancy learnin' in the next section when we kick everything up a few notches.

## Creating Custom Objects

Working with the generic `Object` and putting properties on it serves a useful purpose, but its awesomeness fades away really quickly when you are creating many objects that are basically the same thing:

```
var funnyGuy = {
    firstName: "Conan",
    lastName: "O'Brien",
```



```
    getName: function () {
        return "Name is: " + this.firstName + " " + this.lastName;
    }
};

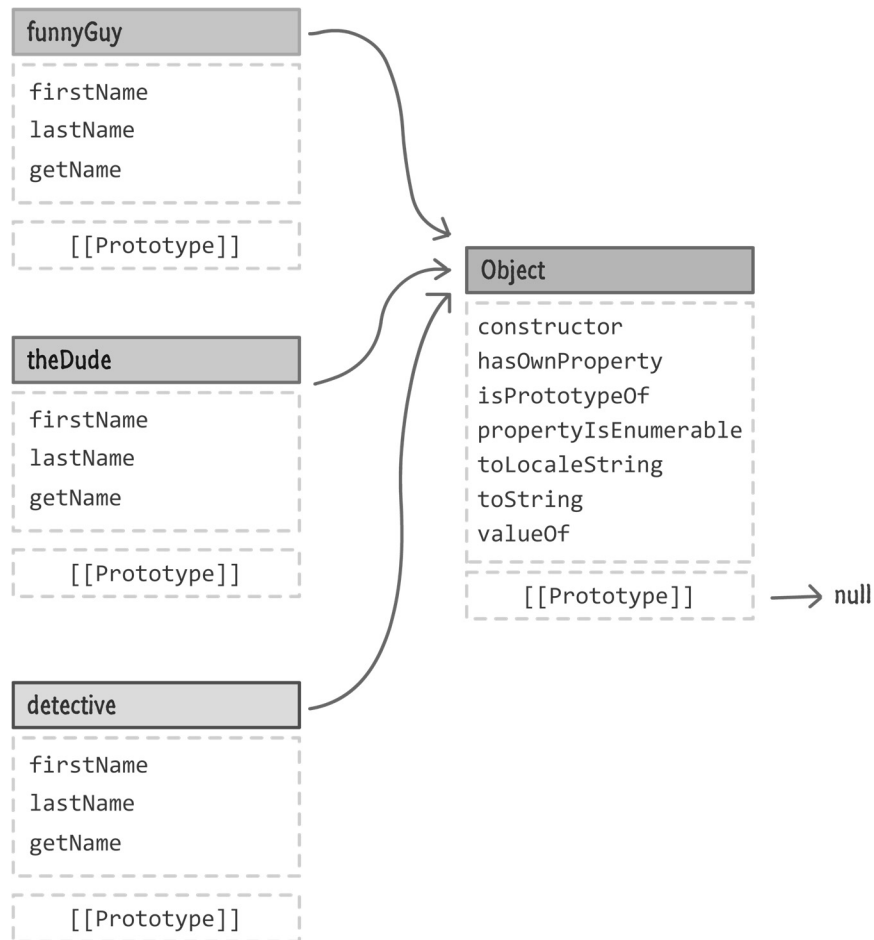
var theDude = {
    firstName: "Jeffrey",
    lastName: "Lebowski",

    getName: function () {
        return "Name is: " + this.firstName + " " + this.lastName;
    }
};

var detective = {
    firstName: "Adrian",
    lastName: "Monk",

    getName: function () {
        return "Name is: " + this.firstName + " " + this.lastName;
    }
};
```

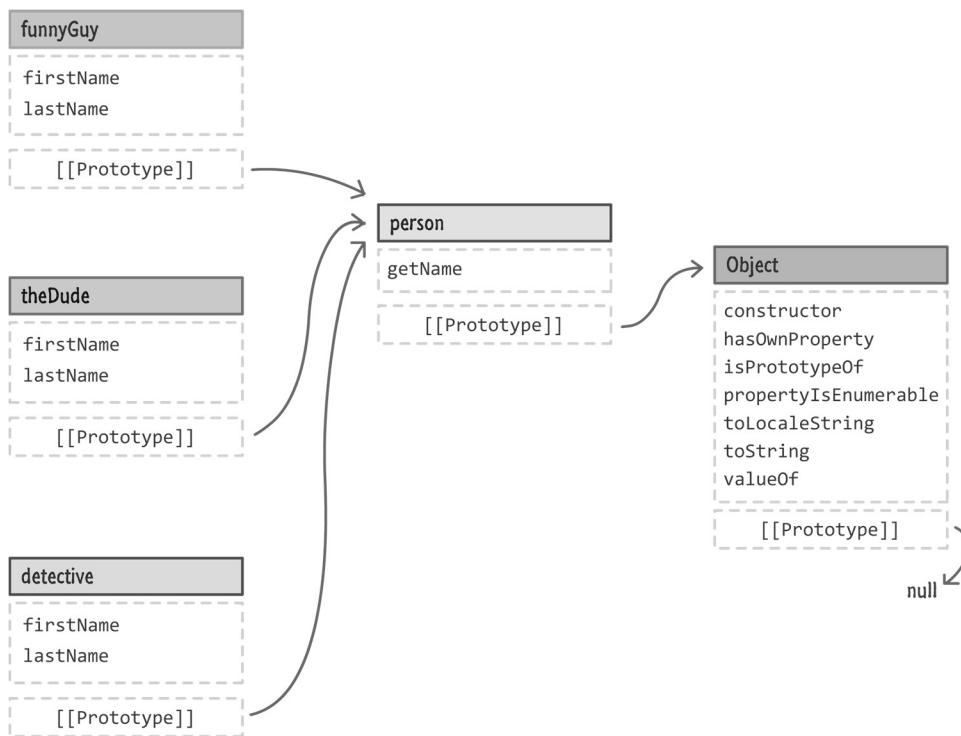
Currently, if we had to visualize what he have right now, this is what you will see:



There is a lot of duplicated stuff here that is, frankly, unnecessary. Let's fix that using what we've learned about inheritance and the prototype chain.

What we want to do is create an intermediate **parent** object that contains the properties that are more generic and not necessary to be on the **child** object itself. From what we have here, the `firstName` and `lastName` properties are going to be unique for each object we create. Because of that, these two properties still belong on the **funnyGuy**, **theDude**, and **detective** objects.

Our `getName` property, though, does not have to be duplicated for each object. This is something we can parcel off into a parent object that the rest of the objects can inherit from. Let's call this object **person**:



Visually, this makes sense. How do we end up creating something like this?

Well, thinking out loud, we need to create our **funnyGuy**, **theDude**, and **detective** objects and ensure the `firstName` and `lastName` properties are defined on them. That's easy. Of course, if this is all we did, this wouldn't be adequate. The prototype for these objects will be **Object**, and we don't want that. We want the **person** object with the `getName` property to be a part of our prototype chain as the immediate parent. The way we do that is by ensuring the `[[Prototype]]` property on **funnyGuy**, **theDude**, and **detective** references **person**.

In order to do this, we use the extremely awesome **Object.create** method. Let me quickly explain what it does before we see it in action. The **Object.create** method, as its name implies, creates a new object. As part of creating the object, it allows you to specify what your newly created object's prototype will be. Strange how what we are wanting to do and what **Object.create** provides are identical! :P

Let's use `Object.create` and the rest of the code that brings the diagram and the explanation you've seen to life:

```
var person = {
  getName: function () {
    return "The name is " + this.firstName + " " + this.lastName;
  }
};

var funnyGuy = Object.create(person);
funnyGuy.firstName = "Conan";
funnyGuy.lastName = "O'Brien";

var theDude = Object.create(person,
  {
    firstName:
      {
        value: "Jeffrey"
      },
    lastName:
      {
        value: "Lebowski"
      }
  });

var detective = Object.create(person,
  {
    firstName: { value: "Adrian" },
    lastName: { value: "Monk" }
  });
```

Let's look at all of this code in greater detail. First, we have our `person` object:

```
var person = {
```

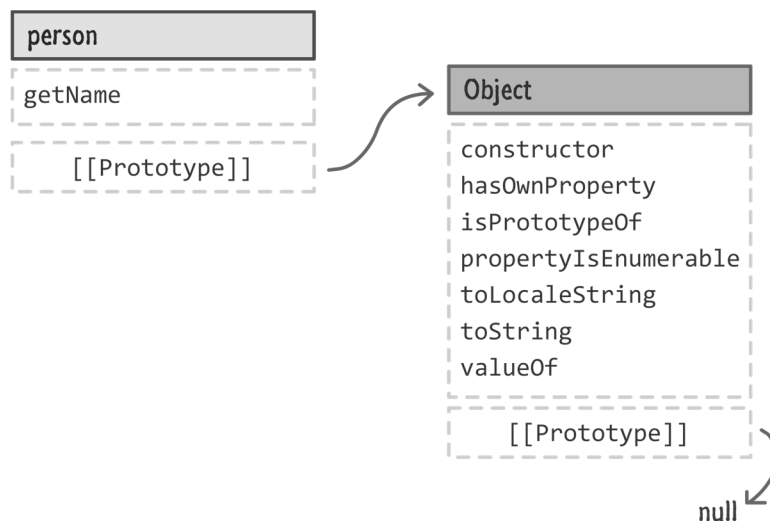
```

    getName: function () {
        return "The name is " + this.firstName + " " + this.lastName;
    }
};

```

There is nothing special going on here. We create a new `person` object whose type is `Object`. It's `[[Prototype]]` property will point you to the `Object` type. It contains a method called `getName` that returns some string involving `this.firstName` and `this.lastName`. We'll come back to the `this` keyword shortly and how this works, so keep that one under your hat for now.

After creating our `person` object, this is what our world looks like right now:



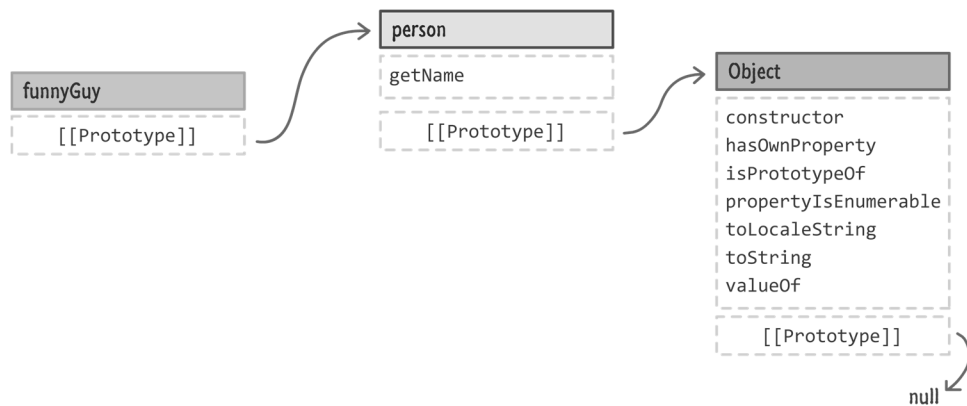
In the next line, we declare our `funnyGuy` variable and initialize it to the object that gets returned by `Object.create`:

```

var funnyGuy = Object.create(person);

```

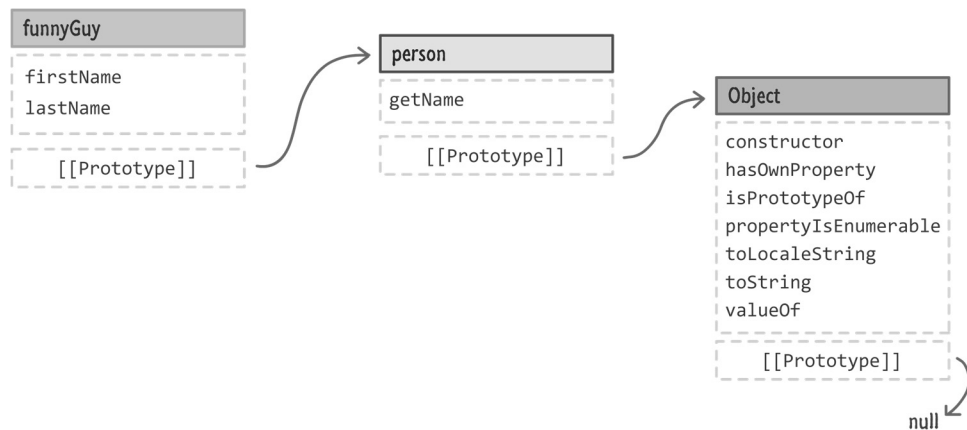
Notice that I pass in the `person` object as an argument to `Object.create`. Like I mentioned earlier, what this means is you create a new object with the `[[Prototype]]` value pointing to our `person` object. This is how things look now:



We created our `funnyGuy` object with the `person` object set as its prototype object. In the next two lines in our code, I define the `firstName` and `lastName` properties on the object:

```
var funnyGuy = Object.create(person);
funnyGuy.firstName = "Conan";
funnyGuy.lastName = "O'Brien";
```

This is your standard, run-of-the-mill property declaration on an object using a name and value. What happens should be of no surprise to you:



We just created our `funnyGuy` object and set the `firstName` and `lastName` properties on it. We just have our `theDude` and `detective` objects left. For these two remaining objects, I use the

`Object.create` method's second argument to pass in default values to set on our object. Let's take a look.

Our `theDude` object is defined by the following chunk of code:

```
var theDude = Object.create(person,
  {
    firstName:
    {
      value: "Jeffrey"
    },
    lastName:
    {
      value: "Lebowski"
    }
  });
```

Just like with `funnyGuy`, I declare a variable and initialize to `Object.create`. The first argument is our `person` object that will act as this newly created object's prototype. The second argument is me specifying the `firstName` and `lastName` properties and their values.

You would think that you could just use the object literal notation that you saw earlier to specify the properties:

```
var theDude = Object.create(person,
  {
    firstName: "Jeffrey",
    lastName: "Lebowski"
  });
```

Unfortunately, you can't be quite that brief. This will throw an error, for every simple property defined via `Object.create` has to be an object with a `value` property containing the value you care about.

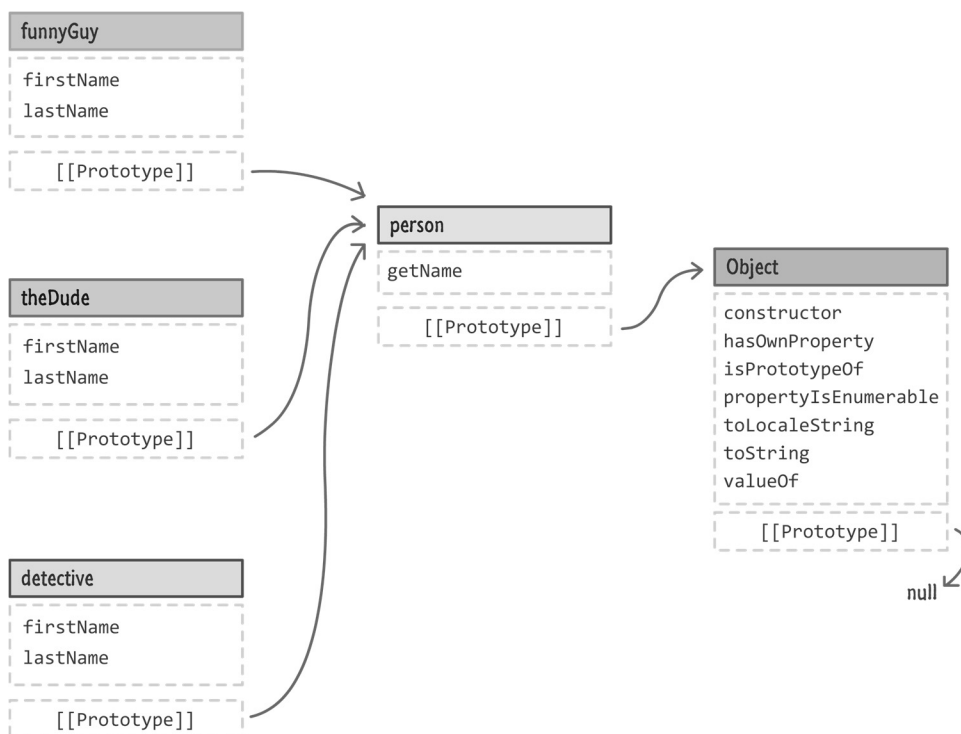
Once you've figured that out and specified the properties to set once this object has been created, the end result is identical to what you saw earlier when creating the `funnyDude` object. You will have created a `theDude` object whose prototype object is `person` with the `firstName` and `lastName` properties storing the appropriate value.

One more object remaining! For our last `detective` object, just to try something similar...yet different, I try to create a more compact way of defining the initial properties as part of creating the object via `Object.create`:

```
var detective = Object.create(person,
  {
    firstName: { value: "Adrian" },
    lastName: { value: "Monk" }
  });
```

By now, you know exactly what is going on. I won't repeat myself here. At the end of all this, you will end up with your child objects created, chained to the parent `person` object, which is in turn chained to the `Object`:





At this point, if you've been following along and understand what is going on, you should be quite impressed with yourself. Many people who work with JavaScript for a very long time have difficulty grasping how inheritance and prototypes tie in to object creation. Wrapping your head around all of this is quite an accomplishment.

Before you pop the champagne bottle and start celebrating, we are not done yet. There is one last thing we need to look at before we call it a day.

## The `this` Keyword

Let's go back to our `person` object and, more specifically, the `getName` property:

```

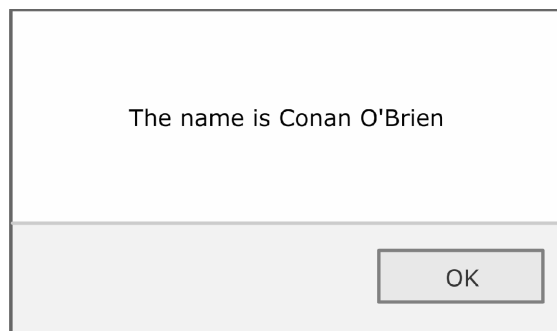
var person = {
  getName: function () {
    return "The name is " + this.firstName + " " + this.lastName;
  }
}
  
```

```
};
```

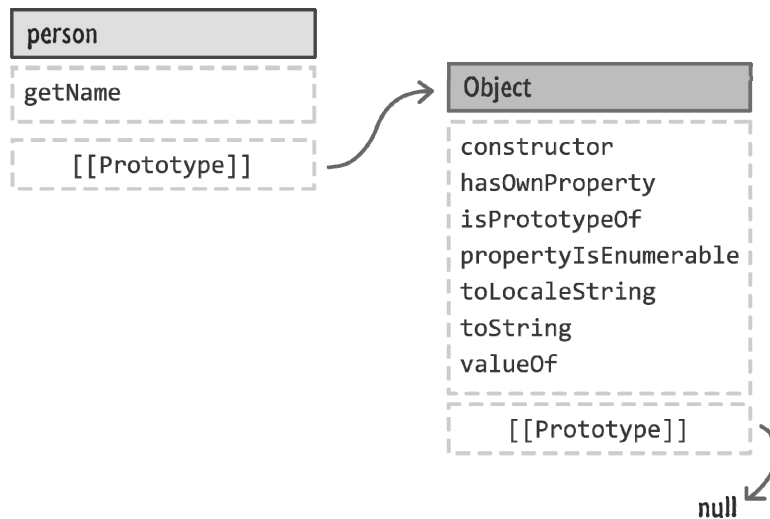
When you call `getName`, depending on which object you called it from, you'll see the appropriate name returned. For example, let's say you do the following:

```
var funnyGuy = Object.create(person);  
funnyGuy.firstName = "Conan";  
funnyGuy.lastName = "O'Brien";  
  
alert(funnyGuy.getName());
```

When you run this, you'll see something that looks as follows:

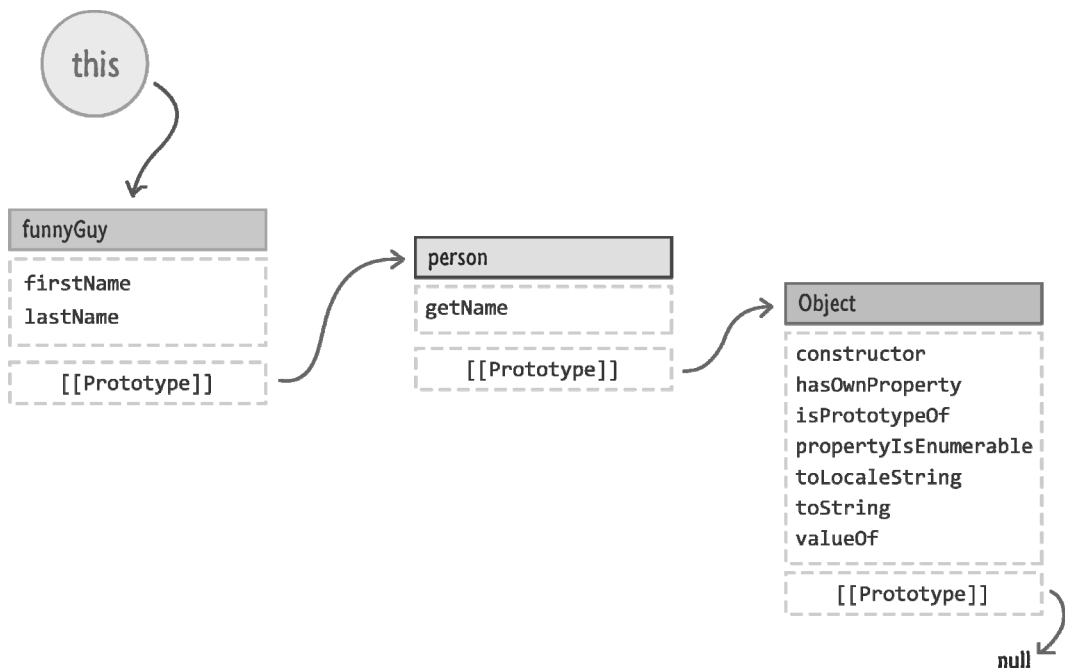


If you look at the `getName` property again, there is absolutely no existence of the `firstName` or `lastName` properties on the `person` object. When a property doesn't exist, I mentioned that we walk down the prototype chain from parent to parent. In this case, that would be `Object`:



There is no existence of the `firstName` or `lastName` properties on `Object` either. How is it that this `getName` method happens to work and return the right values?

The answer has to do with the `this` keyword that precedes `firstName` and `lastName`. The `this` keyword refers to the object that our `getName` method is bound to. That object is, in this case, `funnyGuy`:



At the point where the `getName` method is evaluated and the `firstName` and `lastName` properties have to be resolved, the lookup starts at whatever the `this` keyword is pointing to. In our case, the `this` keyword is pointing to the `funnyGuy` object - an object that contains the `firstName` and `lastName` properties!

Knowing what the `this` keyword refers to is something we'll devote more time to later,, but what you've seen now will get you pretty far.

## Conclusion

Because so much fuss is made about JavaScript's object orientedness, it is only natural that a topic that covers it would be as wide and deep as what you've seen here. A bulk of what you saw here dealt with inheritance directly or indirectly where objects are derived and based on other objects. Unlike other, more class-ical languages that use classes as templates for objects, JavaScript has no such concept of a class. JavaScript uses what is known as a **prototypical inheritance model**. You don't instantiate objects from a template. Instead, you create objects either from scratch or, more commonly, by copying / cloning another object.

In the bazillion pages here, I tried to reinforce JavaScript's new functionality for working with objects and extending them for your own needs. There is still more to cover, so take a break and we'll touch upon some more interesting topics in the near future that extend what you've seen in more powerful, expressive, and awesome ways.

Some additional resources and examples:

- **Understanding “Prototypes” in JS:** <http://bit.ly/kirupaJSPrototypes>
- **A Plain English Guide to JS Prototypes:** <http://bit.ly/kirupaPrototypesGuide>
- **How does JavaScript “prototype” work?:** <http://bit.ly/kirupaPrototypeWork>