**Prototypal Inheritance (Functional)**

At a fundamental level, inheritance in JavaScript is achieved with a chain of prototypes. The approaches around creating prototype chains have evolved significantly over time, as updates to the language have brought new features and syntax.

There are many approaches and variations to creating a prototype chain in JavaScript but we will explore three common approaches:

* functional
* constructor functions
* class-syntax constructors.

For the purposes of these examples, we will be using a Wolf and Dog taxonomy, where a Wolf is a prototype of a Dog.

The functional approach to creating prototype chains is to use **Object.create**:

**const wolf = {  
  howl: function () { console.log(this.name + ': awoooooooo') }  
}**

**const dog = Object.create(wolf, {  
  woof: { value: function() { console.log(this.name + ': woof') } }  
})**

**const rufus = Object.create(dog, {  
  name: {value: 'Rufus the dog'}  
})**

**rufus.woof() // prints "Rufus the dog: woof"  
rufus.howl() // prints "Rufus the dog: awoooooooo"**

The **wolf** object is a plain JavaScript object, created with the object literal syntax (i.e. using curly braces). The prototype of plain JavaScript objects is **Object.prototype**.

The **Object.create** function can take two arguments. The first argument is the desired prototype of the object being created.

When the **dog** object is instantiated, the first argument passed to **Object.create** is the **wolf** object. So **wolf** is the prototype of **dog**. When **rufus** is instantiated, the first argument to **Object.create** is **dog**.

To describe the full prototype chain:

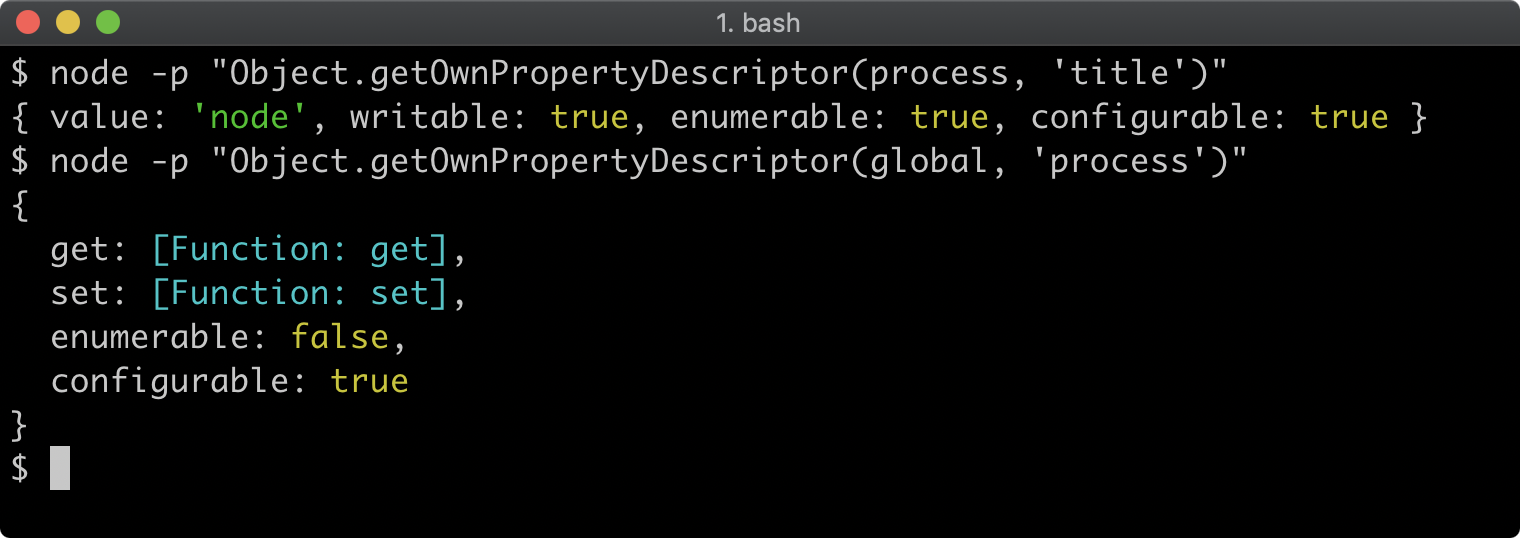
* the prototype of **rufus** is **dog**
* the prototype of **dog** is **wolf**
* the prototype of **wolf** is **Object.prototype**.

The second argument of **Object.create** is an optional Properties Descriptor object.

A Properties Descriptor object contains keys that will become the key name on the object being created. The values of these keys are Property Descriptor objects.

The Property Descriptor is a JavaScript object that describes the characteristics of the properties on another object.

The **Object.getOwnPropertyDescriptor** can be used to get a property descriptor on any object:



To describe the value of a property, the descriptor can either use **value** for a normal property value or **get** and **set** to create a property getter/setter. The other properties are associated meta-data for the property. The **writable** property determines whether the property can be reassigned, **enumerable** determines whether the property will be enumerated, in property iterator abstractions like **Object.keys** and **configurable** sets whether the property descriptor itself can be altered. All of these meta-data keys default to **false**.

In the case of **dog** and **rufus** the property descriptor only sets **value**, which adds a non-enumerable, non-writable, non-configurable property.

Property descriptors are not directly relevant to prototypal inheritance, but are part of the **Object.create** interface so understanding them is necessary. To learn more, read [*"Description"*](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty#Description) section at the MDN web docs Mozilla website.

When the **dog** prototype object is created, the property descriptor is an object with a **woof** key. The **woof** key references an object with the **value** property set to a function. This will result in the creation of an object with a **woof** method.

So when **rufus.woof()** is called, the rufus object does not have a **woof** property itself. The runtime will then check if the prototype object of **rufus** has a **woof** property. The prototype of **rufus** is **dog** and it does have a **woof** property. The **dog.woof** function contains a reference to **this**. Typically, the **this** keyword refers to the object on which the method was called. Since **woof** was called on **rufus** and **rufus** has the **name** property which is **"Rufus the dog"**, the **this.name** property in the **woof** method has the value **"Rufus the dog"** so **console.log** is passed the string: **"Rufus the dog: woof"**.

Similarly when **rufus.howl** is called the JavaScript runtime performs the following steps:

* Check if **rufus** has a **howl** property; it does not
* Check if the prototype of **rufus** (which is **dog**) has a **howl** property; it does not
* Check if the prototype of **dog** (which is **wolf**) has a **howl** property; it does
* Execute the **howl** function setting **this** to **rufus**, so **this.name** will be **"Rufus the dog"**.

To complete the functional paradigm as it applies to prototypal inheritance, the creation of an instance of a dog can be genericized with a function:

**const wolf = {  
  howl: function () { console.log(this.name + ': awoooooooo') }  
}**

**const dog = Object.create(wolf, {  
  woof: { value: function() { console.log(this.name + ': woof') } }  
})**

**function createDog (name) {  
  return Object.create(dog, {  
    name: {value: name + ' the dog'}  
  })  
}**

**const rufus = createDog('Rufus')**

**rufus.woof() // prints "Rufus the dog: woof"  
rufus.howl() // prints "Rufus the dog: awoooooooo"**

The prototype of an object can be inspected with **Object.getPrototypeOf**:

**console.log(Object.getPrototypeOf(rufus) === dog) //true  
console.log(Object.getPrototypeOf(dog) === wolf) //true**