**Prototypal Inheritance (Constructor Functions)**

Creating an object with a specific prototype object can also be achieved by calling a function with the **new** keyword. In legacy code bases this is a very common pattern, so it's worth understanding.

All functions have a **prototype** property. The Constructor approach to creating a prototype chain is to define properties on a function's prototype object and then call that function with **new**:

**function Wolf (name) {  
  this.name = name  
}**

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

**function Dog (name) {  
  Wolf.call(this, name + ' the dog')  
}**

**function inherit (proto) {  
  function ChainLink(){}  
  ChainLink.prototype = proto  
  return new ChainLink()  
}**

**Dog.prototype = inherit(Wolf.prototype)**

**Dog.prototype.woof = function () {  
  console.log(this.name + ': woof')  
}**

**const rufus = new Dog('Rufus')**

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

This will setup the same prototype chain as in the functional Prototypal Inheritance example:

**console.log(Object.getPrototypeOf(rufus) === Dog.prototype) //true  
console.log(Object.getPrototypeOf(Dog.prototype) === Wolf.prototype) //true**

The **Wolf** and **Dog** functions have capitalized first letters. Using PascaleCase for functions that are intended to be called with **new** is convention and recommended.

Note that a **howl** method was added to **Wolf.prototype** without ever instantiating an object and assigning it to **Wolf.prototype**. This is because it already existed, as every function always has a preexisting **prototype** object. However **Dog.prototype** was explicitly assigned, overwriting the previous **Dog.prototype** object.

To describe the full prototype chain:

* the prototype of **rufus** is **Dog.prototype**
* the prototype of **Dog.prototype** is **Wolf.prototype**
* the prototype of **Wolf.prototype** is **Object.prototype**.

When **new Dog('Rufus')** is called a new object is created **(rufus)**. That new object is also the **this** object within the **Dog** constructor function. The **Dog** constructor function passes **this** to **Wolf.call**.

Using the **call** method on a function allows the **this** object of the function being called to be set via the first argument passed to **call**. So when **this** is passed to **Wolf.call**, the newly created object (which is ultimately assigned to **rufus**) is also referenced via the **this** object inside the **Wolf** constructor function. All subsequent arguments passed to **call** become the function arguments, so the **name** argument passed to **Wolf** is **"Rufus the dog"**. The **Wolf** constructor sets **this.name** to **"Rufus the dog"**, which means that ultimately **rufus.name** is set to **"Rufus the dog"**.

In legacy code bases, creating a prototype chain between **Dog** and **Wolf** for the purposes of inheritance may be performed many different ways. There was no standard or native approach to this before EcmaScript 5.

In the example code an **inherit** utility function is created, which uses an empty constructor function to create a new object with a prototype pointing, in this case, to **Wolf.prototype**.

In JavaScript runtimes that support EcmaScript 5+ the **Object.create** function could be used to the same effect:

**function Dog (name) {  
  Wolf.call(this, name + ' the dog')  
}**

**Dog.prototype = Object.create(Wolf.prototype)**

**Dog.prototype.woof = function () {  
  console.log(this.name + ': woof')  
}**

Node.js has a utility function: **util.inherits** that is often used in code bases using constructor functions:

**const util = require('util')**

**function Dog (name) {  
  Wolf.call(this, name + ' the dog')  
}**

**Dog.prototype.woof = function () {  
  console.log(this.name + ': woof')  
}**

**util.inherits(Dog.prototype, Wolf.prototype)**

In contemporary Node.js, **util.inherits** uses the EcmaScript 2015 (ES6) method **Object.setPrototypeOf** under the hood. It's essentially executing the following:

**Object.setPrototypeOf(Dog.prototype, Wolf.prototype)**

This explicitly sets the prototype of **Dog.prototype** to **Wolf.prototype**, discarding whatever previous prototype it had.