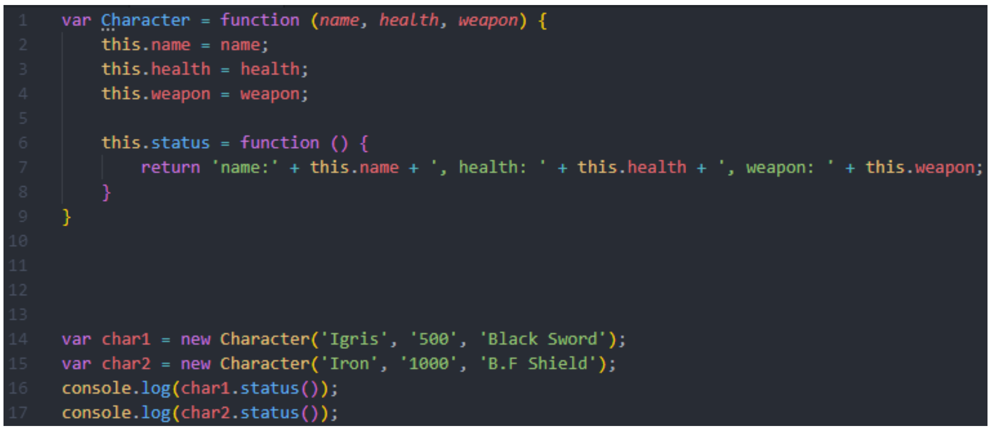
## C# vs JavaScript

In class-based languages, you define and create a class in a separate class definition. In these definitions are special methods called “constructors” used to create an instance of the class. Objects employ the *new* operator in association with the constructor method to create class instances. On the other hand, JavaScript defines a constructor function to create objects. You use the new operator with a constructor to create a new object (Guru99).

C#’s object system is based on classes, whereas JavaScript’s object system is based on prototypes. A class is a blueprint containing a description of how an object is to be created. Inheritance in JavaScript is based on prototypes also known as “prototypal inheritance” (Techsith 2016).

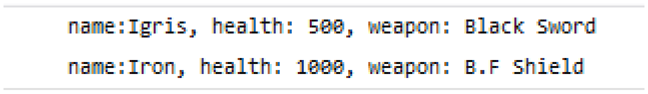
JavaScript implements inheritance by associating a prototypical object with a constructor function (MDN Web Docs 2019). Every function has a property called ‘Prototype’, which is empty by default (Cui 2011). You can add properties and methods to it. When you create objects from the Character function, it will inherit these properties and methods that define Character.

This is what it looks like creating an object without a prototype.



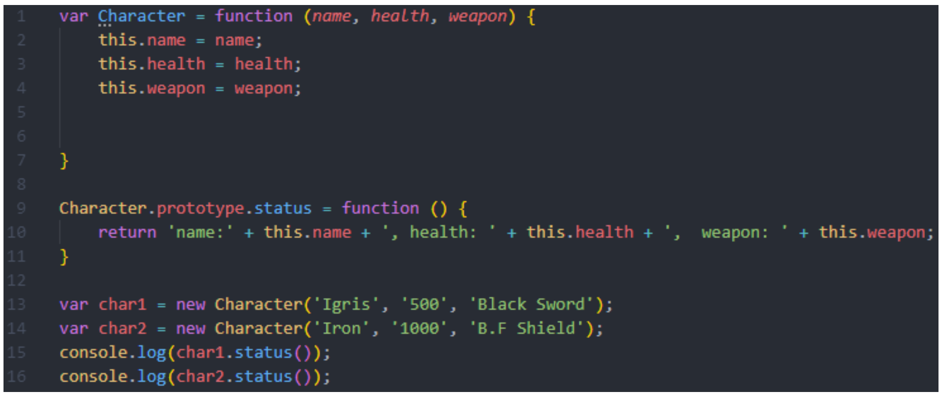
```
var Character = function (name, health, weapon) {
    this.name = name;
    this.health = health;
    this.weapon = weapon;

    this.status = function () {
        return 'name:' + this.name + ', health: ' + this.health + ', weapon: ' + this.weapon;
    }
}




var char1 = new Character('Igris', '500', 'Black Sword');
var char2 = new Character('Iron', '1000', 'B.F Shield');
console.log(char1.status());
console.log(char2.status());
```

This is the result of console.log() applied to char1 and char2, without utilising prototypes.



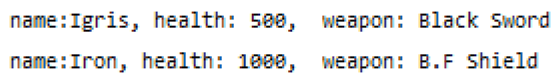
```
name:Igris, health: 500, weapon: Black Sword
name:Iron, health: 1000, weapon: B.F Shield
```

Every object that gets created using the constructor function will have its own copy of properties and methods. The problem is that it does not quite make sense for there to be two instances of the *status* function that do the exact same thing. By storing separate instances of functions for each object, a lot of memory goes to waste. There’s where the prototypes come in.



```
var Character = function (name, health, weapon) {
    this.name = name;
    this.health = health;
    this.weapon = weapon;


}

Character.prototype.status = function () {
    return 'name:' + this.name + ', health: ' + this.health + ',  weapon: ' + this.weapon;
}

var char1 = new Character('Igris', '500', 'Black Sword');
var char2 = new Character('Iron', '1000', 'B.F Shield');
console.log(char1.status());
console.log(char2.status());
```

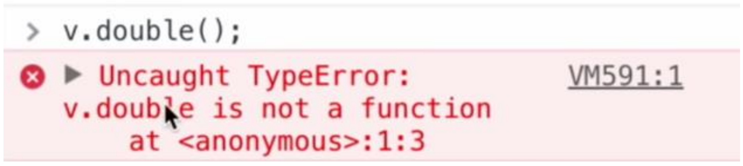
This prototype is automatically available to *char1* and *char2*, even though it is not a part of the *Character* variable. This is the result of console.log() applied to char1 and char2, utilising prototypes.



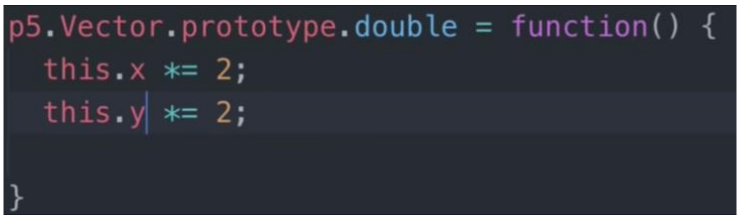
```
name:Igris, health: 500,  weapon: Black Sword
name:Iron, health: 1000,  weapon: B.F Shield
```

The only difference is that *char1* and *char2* do not have their own *status* method inside. Instead, whenever char1 and char2 get called, it will look for the method in the parent of the prototype chain. If the method exists, it will execute it from the parent, *Character* (Techsith 2016).

The prototype also allows you to add functions to libraries which is not something you can normally do in a programming language. Here is an example of adding a new *double* method that doubles the original vector with the use of prototypes. This would not have been possible to accomplish without prototypes because we can not directly edit the p5 library (The Coding Train 2017).



```
> v.double();
▶ Uncaught TypeError:                    VM591:1
v.double is not a function
    at <anonymous>:1:3
```



```
p5.Vector.prototype.double = function() {
  this.x *= 2;
  this.y *= 2;

}
```

Note: This example has nothing to do with the Characters function. I could not find an appropriate external library to use. So, I used The Coding Train's example.

In C# we start by defining the parent and child classes, creating a hierarchy of classes (Guru99 2019). Child classes inherit methods and properties of the parent class, but they can also modify how methods behaviour as well. The child class can even define its own methods if required.

In the shape task 5.3D, we had *Shape* as our parent class and our child classes were Circle, Line, and Rectangle. The Shape class knew how to be selected, save to file, load from file. This meant that Circle, Line, and Rectangle classes would also know how to do these things. However, each of these child class had additional methods such as Draw() and DrawOutline().

References:

- Cui, Y 2011, Javascript's prototypal inheritance, for a C# develop, theburningmonk , viewed 30 April, 2019, <https://theburningmonk.com/2011/01/javascripts-prototypal-inheritance-for-a-c-sharp-developer/>.
- Guru99, 2019, C# Inheritance & Polymorphism with Examples, Guru99, viewed 30/04/2019, <https://www.guru99.com/c-sharp-inheritance-polymorphism.html>.
- MDN Web Docs 2019, Details of the object model, MDN Web Docs, viewed 30 April, 2019, <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Details_of_the_Object_Model>.
- Techsith 2016, Javascript Prototype inheritance Explained ( tutorial Part1), YouTube, viewed 30 April, 2019, <https://www.youtube.com/watch?v=7oNWNlMrkpc>.
- The Coding Train 2017, 9.19: Prototypes in Javascript - p5.js Tutorial, YouTube, viewed 30 April, 2019, <https://www.youtube.com/watch?v=hS_WqkyUah8>.