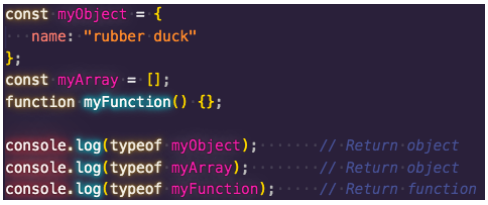
Advanced JavaScript

Section 2: Inheritance & Prototypes, Closures & Scope Chain, Call Bind Apply and IIFEs

---

1.1 Inheritance & Prototypes

Prototypes and Inheritance are key features in JavaScript and are challenging concepts especially if coming from a class based language such as Java or C++ i.e. JavaScript does things differently.



Almost everything in JavaScript is an object. We can use the typeof JavaScript command to see the type of data. Objects are basically a data structure that has has properties i.e. a function or a key:value pair. An array is also an object and can be seen as an unordered list of data. Functions are a special type of objects because we can invoke or call a function.

All the above matters in terms of how JavaScript dos inheritance. In the class based language you may essentially have one super class like animal as an example and all animals will have certain things in common e.g. a type and a breed but then you would have a sub-class such as a dog which has some properties that other animals may not have. JavaScript does not do sub-class but rather has a Prototype system.

Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own and so on until an object is reached with null as its prototype.

The prototype chain comes into play when we try to access a property of an object. Using the example above to access the myObject name property we would write myObject.name as the syntax. If the property did not exist in the object, JavaScript would look in the special prototype object to check if the property exists there. If the answer is no, JavaScript will keep going up the prototype chain until it eventually does /does not find the property. If it does not find the property the value will be null (typically JavaScript would return undefined in the console when it cannot find the property).

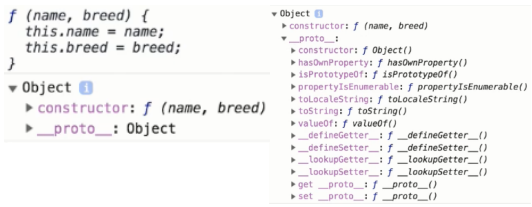
When we look at the type of object we would see in the console that “object” is returned (notice the lower case letter o). This object is an instance i.e. a spawned version of Object — if we were to look at the prototype of this Object property using console.log(Object.prototype) we would see that it has all sorts of values and properties i.e. all sort of functions e.g. .toString( ). Therefore, we can use the myObject.toString( ) function even though this function was not defined on this object but rather the myObject prototype has access to the function because it inherited everything from the Object prototype.

An array is a subtype of object and therefore arrays inherits all the prototypes properties and values from Object as well as adding its own properties and values (i.e. functions). We can inspect this in chrome developer tools by running the console.log(Array.prototype) command. This is also the reason why we can call on array functions such as .length on an Array which we cannot do on an Object because arrays have additional functions it inherits from the array prototype.

To illustrate inheritance when we instantiate an object in JavaScript we will create an object using something called a function construct (a.k.a an Object Constructor). You would usually create a Object Constructor in order to create a reusable constructor. This is demonstrated in the below example:

```
const Dog = function(name, breed) {
    this.name = name;
    this.breed = breed;
};

console.log(Dog.prototype);
```



When we see a variable with a upper case first letter it normally indicates that this variable is an object constructor. The .this property inside of an object refers to the object itself. We can use this object constructor (function construct) to create an object using the new keyword followed by the function construct name. The new keyword instantiates a object and we can create as many new objects using this object constructor.

```
const spot = new Dog("Spot", "Golden Retriever");
console.log(spot);
```

```
const spike = new Dog("Spike", "British Bulldog");
console.log(spike);
```

These are two completely different objects. This is where inheritance and prototypes come into play. If we want to access a property inside of an object the first place JavaScript will look is inside of the object itself. If we ask for a property that does not exists on the object itself for example the age property:

- JavaScript will first look at the object property itself (e.g. spike)
- If the answer is No, JavaScript will check the object's prototype (i.e. the Dog.prototype constructor)
- If the answer is still No, JavaScript will check the object prototype's prototype (i.e. the Object.prototype)
- This will continue to go up the prototype chain until it either finds what it is looking for or returns null/undefined

This is how Inheritance works in JavaScript i.e. it uses this Prototype chain whereby JavaScript will keep going up the chain until it find what it is looking for or doesn't.

```
const Dog = function(name, breed) {
    this.name = name;
    this.breed = breed;
};

const spike = new Dog("Spike", "British Bulldog");
console.log(spike);

Dog.prototype.speak = function() {
    console.log("Woof");
};

console.log(Dog.prototype);
spike.speak();
```

JavaScript allows you to add to the prototype object if you want to (but is not recommend) because how inheritance works in JavaScript. This is demonstrated on the example to the left. This will add the new function to all Dog objects.

```
const Dog = function(name, breed) {
    this.name = name;
    this.breed = breed;
};

const spot = new Dog("Spot", "British Bulldog");
const spike = new Dog("Spike", "British Bulldog");

Dog.prototype.speak = function() {
    console.log("Woof");
};

spot.speak = function() {
    console.log("Hi i'm Spot");
};

spot.speak();     // Returns "Hi i'm spot"
spike.speak();    // Returns "Woof!""
```

We could have added this function to only on the object itself which would only be applicable to that object. Notice in the example on the left, what is returned by the speak function are two complete different output due to how inheritance and prototypes work in JavaScript.

JavaScript also allows us to create new objects using the Object.create( ) method. JavaScript also has a hasOwnProperty method it uses (and we can use) to check whether to look up the chain to find whether the property exists or not on the Object. The example on the right demonstrates inheritance in action.

```
const Car = {
    drives: true
};

const Honda = Object.create(Car);
console.log(Honda.hasOwnProperty("drives"));     // Returns false
console.log(Car.hasOwnProperty("drives"));       // Returns true
```

ES6 introduces the class keyword to perform inheritance which under the hood it is the same as any other inheritance in JavaScript. Classes have the special constructor( ) keyword in JavaScript and behaves exactly the same as the function construct behind the scenes (demonstrate above with the Dog example). The class syntax is simply a new syntactical sugar syntax.

```
class Animal {
    constructor(type, age) {
        this.type = type;
        this.age = age;
    };
};

const dog = new Animal("dog", 5);
console.log(dog);
```

ES6 classes makes it easy to do inheritance using the extends keyword. Instead of using Object.create we use the extends keyword which tells JavaScript the first object to inherit from the second object.

The super function in the constructor will tell JavaScript to call on the super class i.e. the class it extends from (e.g. Animal).

Any properties we add on afterwards will belong to the class object alone (e.g. the name property belongs to the Dog class). The Dog class can also access all the properties from the super class i.e. Animal that it inherits from.

```
class Animal {
    constructor(type, age) {
        this.type = type;
        this.age = age;
    };
};

class Dog extends Animal {
    constructor(type, age, name) {
        super(type, age);
        this.name = name;
    };
};

const spot = new Dog("dog", 5, "Spot");
console.log(spot);
```