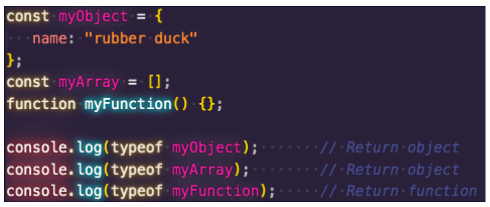
Advanced JavaScript

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.



```
const myObject = {
   name: "rubber duck"
};
const myArray = [];
function myFunction() {};

console.log(typeof myObject);       // Return object
console.log(typeof myArray);        // Return object
console.log(typeof myFunction);     // Return 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.

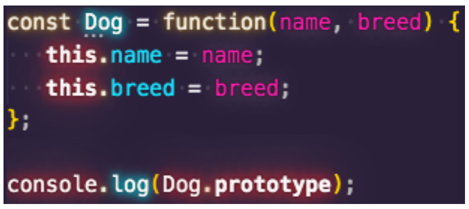
Each object has a private property which holds a link to another object called its prototype. That prototype object has a prototype of its own and so on until an object is reached with null as its prototype.

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).

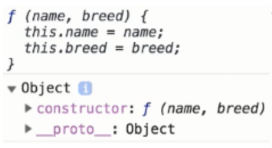
When we look at the type of object we would see in the console that “object” is returned (notice the lower case letter o). This object is an instance i.e. a spawned version of Object — if we were to look at the prototype of this Object property using console.log(Object.prototype) we would see that it 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.

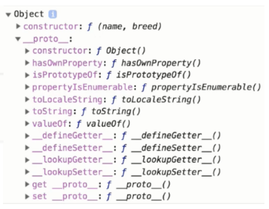
To illustrate inheritance when we instantiate an object in JavaScript we will create an object using something called a function construct (a.k.a an Object Constructor). You would usually create a Object Constructor in order to create a reusable constructor. This is demonstrated in the below example:



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

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

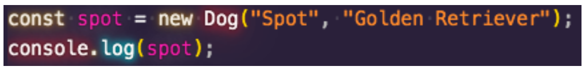


```
ƒ (name, breed) {
  this.name = name;
  this.breed = breed;
}
▼ Object
  ▶ constructor: ƒ (name, breed)
  ▶ __proto__: Object
```

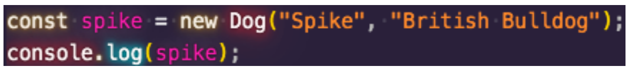


```
▼ Object
  ▶ constructor: ƒ (name, breed)
  ▼ __proto__:
    ▶ constructor: ƒ Object()
    ▶ hasOwnProperty: ƒ hasOwnProperty()
    ▶ isPrototypeOf: ƒ isPrototypeOf()
    ▶ propertyIsEnumerable: ƒ propertyIsEnumerable()
    ▶ toLocaleString: ƒ toLocaleString()
    ▶ toString: ƒ toString()
    ▶ valueOf: ƒ valueOf()
    ▶ __defineGetter__: ƒ __defineGetter__()
    ▶ __defineSetter__: ƒ __defineSetter__()
    ▶ __lookupGetter__: ƒ __lookupGetter__()
    ▶ __lookupSetter__: ƒ __lookupSetter__()
    ▶ get __proto__: ƒ __proto__()
    ▶ set __proto__: ƒ __proto__()
```

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);
```

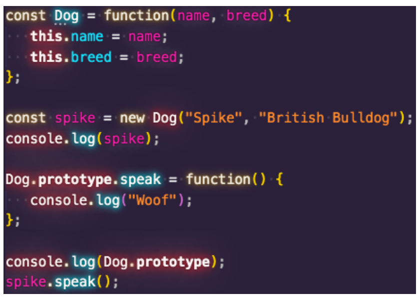


```
const spike = new Dog("Spike", "British Bulldog");
console.log(spike);
```

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:

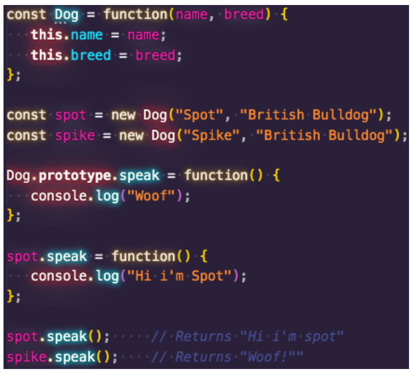
- JavaScript will first look at the object property itself (e.g. spike)
- If the answer is No, JavaScript will check the object's prototype (i.e. the Dog.prototype constructor)
- If the answer is still No, JavaScript will check the object prototype's prototype (i.e. the Object.prototype)
- This will 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();
```

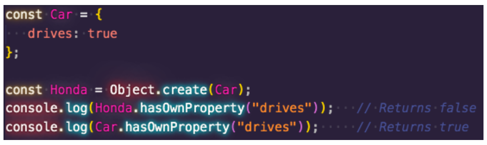
JavaScript allows you to add to the prototype object if you want to (but is not recommend) because how inheritance works in JavaScript. This is demonstrated on the example to the left. This will add the new function to all Dog objects.



```javascript
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!""
```

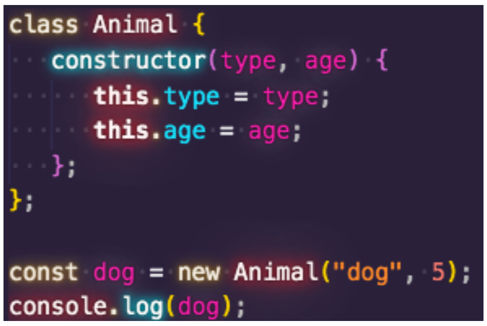
We could have added this function to only on the object itself which would only be applicable to that object. Notice in the example on the left, what is returned by the speak function are two complete different output due to how inheritance and prototypes work in JavaScript.

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.



```javascript
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.

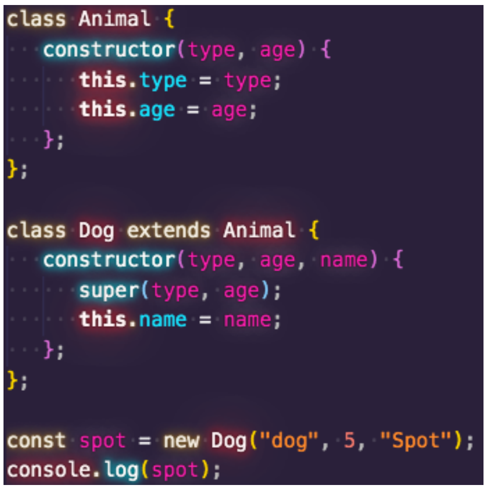


```javascript
class Animal {
  constructor(type, age) {
      this.type = type;
      this.age = age;
  };
};

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.

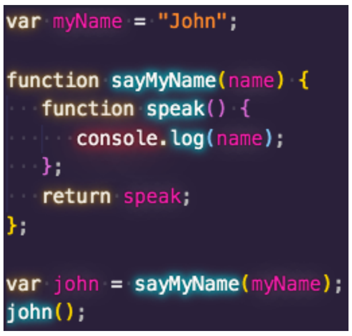


```javascript
class Animal {
  constructor(type, age) {
      this.type = type;
      this.age = age;
  };
};

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

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

1.2 Closures and Scope Chain

The scope chains is a very important topic when it comes to JavaScript. The scope chain tells us what variables/functions/objects or any given data our code has access to. The global execution context makes things globally available e.g. the DOM can be accessed anywhere inside of our code whether inside or outside a function.

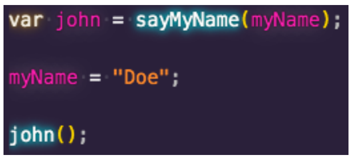
To understand scope we need to understand closures. A closure is the combination of a function and the lexical environment within which that function was declared. This environment consists of any local variables that were in-scope at the time the closure was created.



```
var myName = "John";

function sayMyName(name) {
    function speak() {
        console.log(name);
    };
    return speak;
};

var john = sayMyName(myName);
john();
```

The example on the left is an example of a very simple closure. The function sayMyName takes in name as an argument and it declares a speak function inside itself (i.e. we can have nested functions). The speak function console.log whatever was passed into the sayMyName function argument. What is interesting is that the sayMyName function will return a reference pointer to the speak function.

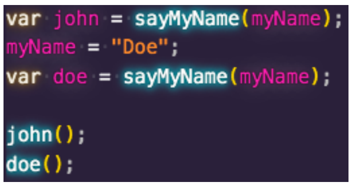
When we invoke the sayMyName function we can assign a value for the function to a variable. The var john variable is assigned to have whatever is inside of the sayMyName function's curly brackets. When we invoke the speak function, it is going to grab whatever name was passed in and console log it out. To invoke the function, we would do exactly as we would with any other function (i.e. john( ); as seen in the example).



```
var john = sayMyName(myName);

myName = "Doe";

john();
```

What we have done is closed over the value of myName, what this means is that if we changed/reassigned the value of myName after calling sayMyName this will not affect the value of the function (i.e. john( ) will print out "John" and not "Doe").

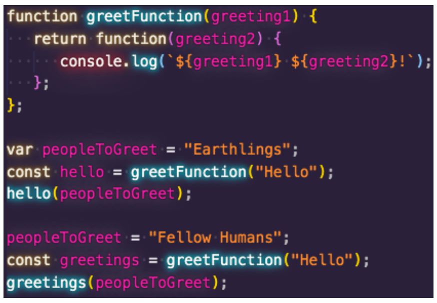
At the time the sayMyName code executed it closed over the value (i.e. it captured the original myName variable value) and it is using it inside of the speak function.



```
var john = sayMyName(myName);
myName = "Doe";
var doe = sayMyName(myName);

john();
doe();
```

Therefore, even if later on we reassigned the myName variable it does not matter because the sayMyName function has already captured what the original value was and it does not change the value of the john( ) function. This can be seen as closing over the value at the time of execution. In the example to the right john( ) will print "John" and doe( ) will print "Doe".

Closures allow us to associate data with some function that uses the data.

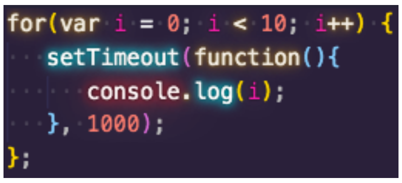


```
function greetFunction(greeting1) {
    return function(greeting2) {
        console.log(`${greeting1} ${greeting2}!`);
    };
};

var peopleToGreet = "Earthlings";
const hello = greetFunction("Hello");
hello(peopleToGreet);

peopleToGreet = "Fellow Humans";
const greetings = greetFunction("Hello");
greetings(peopleToGreet);
```

To the left is another example of a closure function. This time the greetFunction takes in an argument while the inner anonymous function takes in a different argument. This allows us to print out Hello Earthlings! This essentially demonstrates the closure of both greeting1 And greeting2 values.

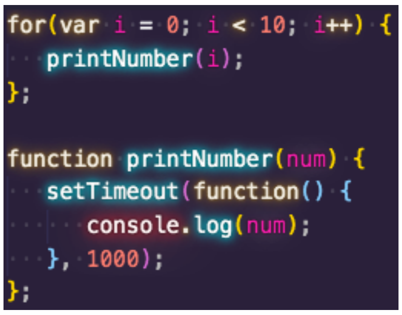
This allows us to reuse the greetFunction multiple times to create different types of greetings. Both hello and greetings have the same function body (i.e. greetFunction) but they have different lexical scope which is why they can print different greeting messages using the same function.

Another classic example when it comes to closures is a for loop. A for loop does a certain thing for a set number of times.



```
for(var i = 0; i < 10; i++) {
  setTimeout(function(){
    console.log(i);
  }, 1000);
};
```

In this example, this prints out the number 10 ten times in the console. The reason for this is because it loops very quickly it did not capture the variable i value. We are waiting a set amount of time (1 second) and printing out the value of i long after it is finished looping.



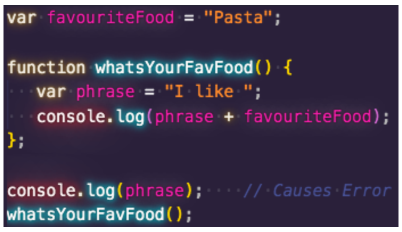
```
for(var i = 0; i < 10; i++) {
  printNumber(i);
};

function printNumber(num) {
  setTimeout(function() {
    console.log(num);
  }, 1000);
};
```

To fix the issue above is to capture the value at any given moment. By closing over the value of the i variable - at the time it invokes the printNumber function it will capture this value for every loop which will result in 0 through to 9 being printing.

This is a very good reason (and example) of why we would use closure to prevent any unintentional side effects. Closures is a very important concept in JavaScript.

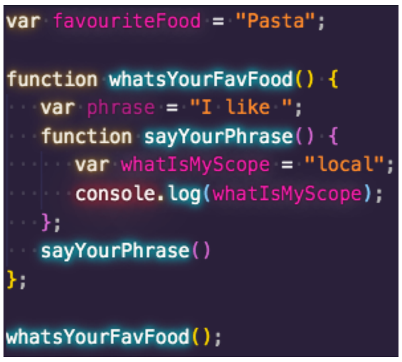
There are two types of scopes in JavaScript: local or global. When we declare variables that are not within a function or an object we are in fact declaring the variable globally. This means that we can access this variable inside of any function/object. The document object is an example of a global object which can be accessed anywhere.

Scope does not work the other way — what do we mean by this? If we declare a variable inside of a function the variable is scoped to that function and we do not have access to it outside of that function. Therefore, anything inside the function has access to the global scope as well as anything declared inside the function but not vice versa.



```
var favouriteFood = "Pasta";

function whatsYourFavFood() {
  var phrase = "I like ";
  console.log(phrase + favouriteFood);
};

console.log(phrase);    // Causes Error
whatsYourFavFood();
```

In the example on the left favouriteFood is a global variable while phrase is a local variable to whatsYourFavFood function. When trying to access the local scoped variable outside the function this causes an error that the variable has not been defined in the global scope. The local scope variables terminates when the function completes. We can think of local scopes as a reverse pyramid i.e. variables go in one direction.



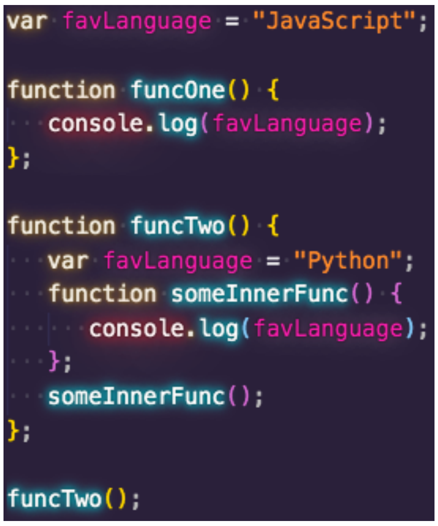
```
var favouriteFood = "Pasta";

function whatsYourFavFood() {
  var phrase = "I like ";
  function sayYourPhrase() {
    var whatIsMyScope = "local";
    console.log(whatIsMyScope);
  };
  sayYourPhrase()
};

whatsYourFavFood();
```

The second example demonstrates this whereby the sayYourPhrase has access to variables in the whatsYourFavFood as well as the variables inside of it but the whatsYourFavFood does not have access to the variables inside of sayYourPhrase. Calling the sayYourPhrase function inside of whatsYourFavFood invokes the sayYourPhrase function which has access to the variable and does not cause any errors.

This is what scope is all about at its core — variable scope is a one way stream.
It is recommended to avoid declaring variables in the global namespace because it can cause you to call global variable values in the local scope unexpectedly.

On the example to the left, funcTwo will print out Python instead of JavaScipt because the variable exists in the local scope (which also happens to have the same name as the global scoped variable.

If the favLanguage = “Python” did not exists and we were to run this code, Javascript would first search whether the favLanguage exists in the someInnerFunc local scope. If the answer is no, JavaScript will move up and look for the variable inside of the funcTwo scope. This will continue up the scope chain until it either finds the variable in local scope or it gets to the global scope. If nothing is found in either the local or global scope JavaScript would throw a ReferenceError.

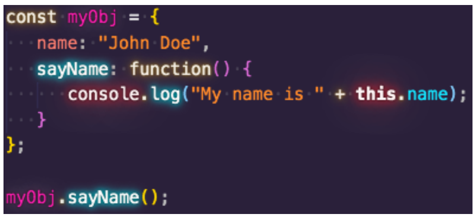


```
var favLanguage = "JavaScript";

function funcOne() {
  console.log(favLanguage);
};

function funcTwo() {
  var favLanguage = "Python";
  function someInnerFunc() {
    console.log(favLanguage);
  };
  someInnerFunc();
};

funcTwo();
```

1.3 Call, Apply, Bind

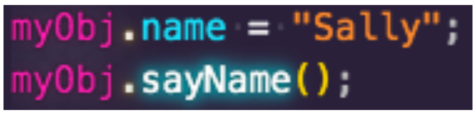
The Call, Apply and Bind are different methods that we can call on functions i.e. they exists on the function prototype object i.e. any functions can use these methods. These methods are designed to set context i.e. what the “this” value refers to. Depending on the use case we are going to use either methods to determine the context of the this keyword.

The .call( ) method calls a function with a given “this” value and arguments provided individually.



```
const myObj = {
  name: "John Doe",
  sayName: function() {
    console.log("My name is " + this.name);
  }
};

myObj.sayName();
```

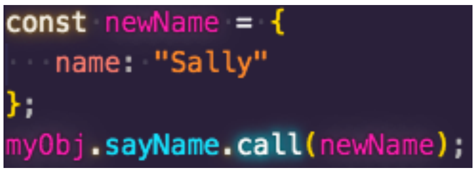
In the example to the left the the function “this” keyword is pointing to the object itself which has a property called name.



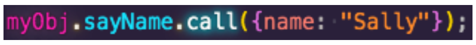
```
myObj.name = "Sally";
myObj.sayName();
```

What if we wanted to say a different name? We could easily alter the myObj.name value and this would work but we have now altered the object which is usually something you should avoid doing (especially with a large JavaScript file where we are referencing this object in multiple places).

This is what call is all about. Instead of immediately invoking a function we can use the .call( ) method which takes in a couple of different parameters. First we need to provide what context the “this” keyword to be.



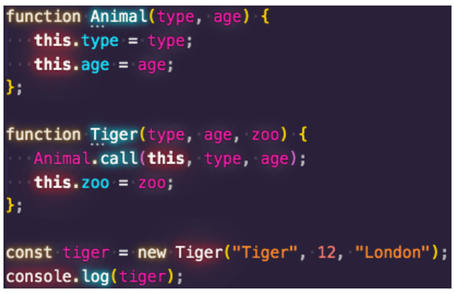
```
const newName = {
  name: "Sally"
};
myObj.sayName.call(newName);
```



```
myObj.sayName.call({name: "Sally"});
```

We could give the call method a new object for the “this” keyword context i.e. pass in a new “this” value directly as the first and only parameter/argument to the call method. It would now look for this.name in the new object. We could also instantiate an object on the fly as the parameter to the call method in JavaScript rather than assigning an object to a variable.

Whenever we use the call method it invokes the call method immediately.



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

function Tiger(type, age, zoo) {
  Animal.call(this, type, age);
  this.zoo = zoo;
};

const tiger = new Tiger("Tiger", 12, "London");
console.log(tiger);
```

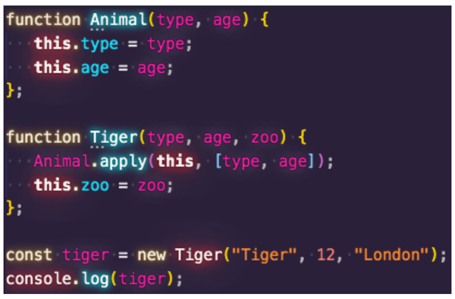
In the example to the right we have an object constructor example that can use the call method. We can instantiate the Animal and Tiger using the new keyword.

If we want to instantiate an Animal but from within Tiger — in Tiger function we can use the Animal function (even though we are using this function to construct an object) and use the .call( ) method to change the value of this keyword. The first argument is the "this" keyword which refers to the current context i.e. the Tiger context. It is then followed by a list of arguments that we ant to change in Animal i.e. what do we want to pass from Tiger to Animal. Animal needs the type and age values because the this.type and this.age values are set based on the type and age we pass in. Therefore, the "this" in the call method is going to refer to the current context it is in i.e. Tiger. The list must be in the same order of the Animal function parameters.
The zoo value is not passed into Animal.call( ) parameter list because Animal does not have a zoo property as this property only belongs to Tiger.

Important Note: The first argument to the call method does not strictly have to be "this" keyword but is common to see as a used pattern. We could pass in an object instead as seen in the previous example (i.e. whatever the context of the "this" keyword needs to be when using the call method).

We can now create a new Tiger object based off of the Animal constructor and assign/store the value into a variable. This can be useful if we want to have a generic function that we can use anywhere. This comes back to the inheritance pattern when it comes to JavaScript.

Important Note: We could have created a ES6 class and used the super keyword instead but this is something new and may not be supported across all browsers.



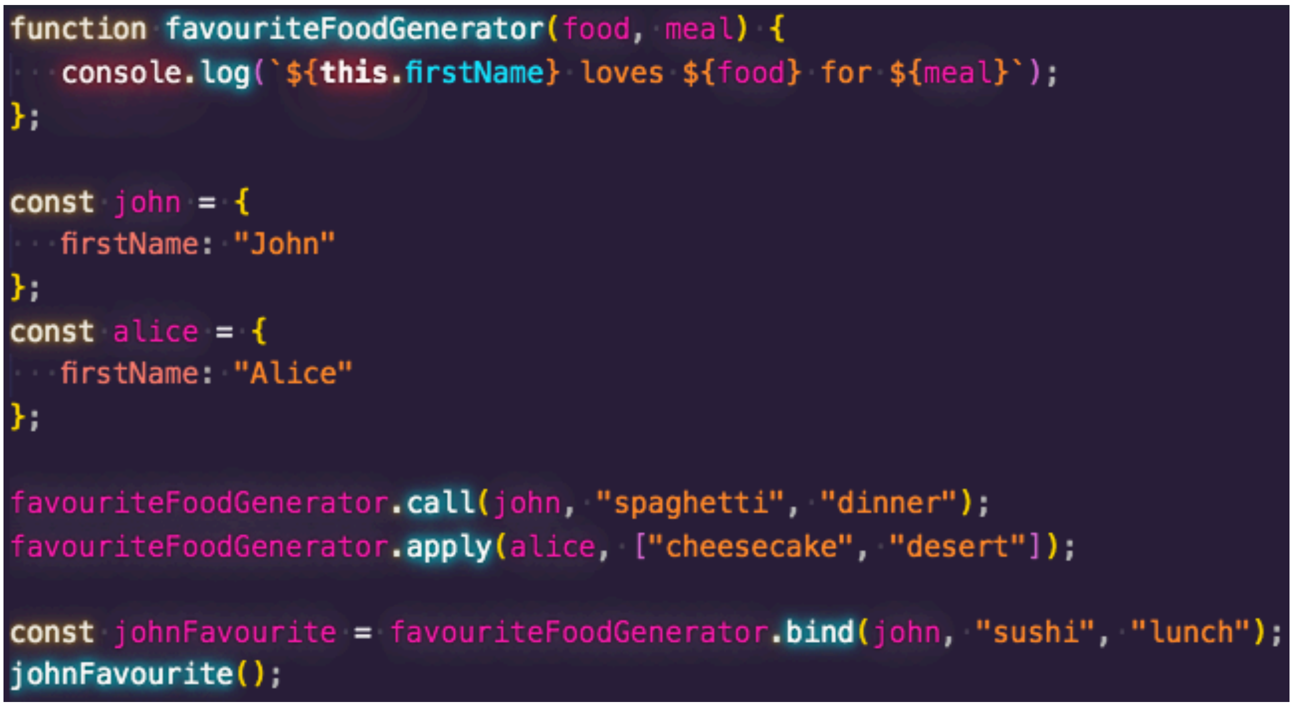
```
function Animal(type, age) {
  this.type = type;
  this.age = age;
};

function Tiger(type, age, zoo) {
  Animal.apply(this, [type, age]);
  this.zoo = zoo;
};

const tiger = new Tiger("Tiger", 12, "London");
console.log(tiger);
```

The .apply( ) method calls a function with a given "this" value and arguments provided as an array (or an array-like object). The .apply( ) method also executes immediately.
This is very similar to the .call( ) method with the array list being the only difference between the two methods. The .call( ) and .apply( ) methods can be used interchangeably.

The .bind( ) method creates a new function that, when called, has its "this" keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

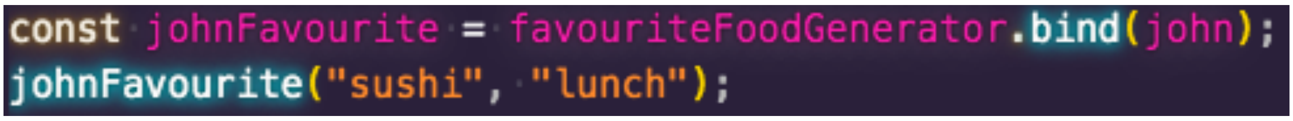
Therefore, the .bind( ) method wraps around the function and does not execute immediately i.e. it creates a new function behind the scenes which wraps around the original function which we can then call at a later time.

Below is an example of a function using all three different methods to view the difference in approach and the context of the “this” keyword:



```
function favouriteFoodGenerator(food, meal) {
  console.log(`${this.firstName} loves ${food} for ${meal}`);
};

const john = {
  firstName: "John"
};
const alice = {
  firstName: "Alice"
};

favouriteFoodGenerator.call(john, "spaghetti", "dinner");
favouriteFoodGenerator.apply(alice, ["cheesecake", "desert"]);

const johnFavourite = favouriteFoodGenerator.bind(john, "sushi", "lunch");
johnFavourite();
```

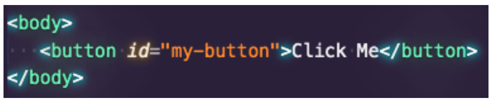
We can also use the .bind( ) method differently — the example below is called a partial function or a currying function which means the function completes when we pass in the arguments.



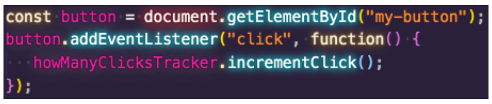
```
const johnFavourite = favouriteFoodGenerator.bind(john);
johnFavourite("sushi", "lunch");
```

Important Note: If we forget to pass in the variables it will show up as undefined.

Below is a more practical example using a HTML button that increments:



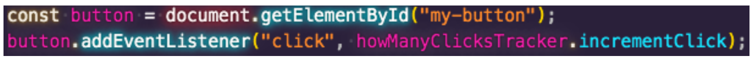
```
<body>
  <button id="my-button">Click Me</button>
</body>
```



```
const button = document.getElementById("my-button");
button.addEventListener("click", function() {
  howManyClicksTracker.incrementClick();
});
```

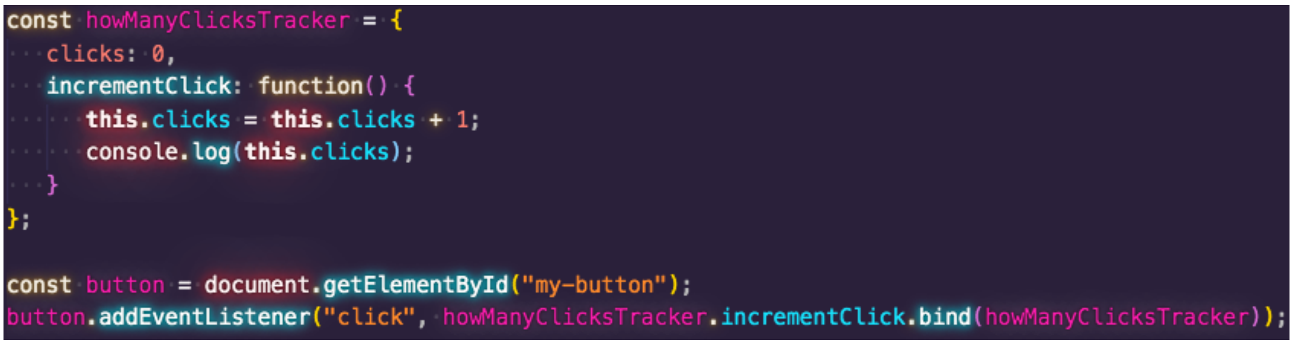
The example on above provides an example of incrementing using an anonymous function; however, the “this” keyword context changes from an “object Object” when inside of the incrementClick function to the “object HTMLButtonElement” when inside the anonymous function (*this can be demonstrated by adding console.log(this) in both the anonymous function and the incrementClick function*). This demonstrates that the “this” keyword in JavaScript can be daunting. By using the the .bind( ) method we do not have to worry about the “this” keyword.

The below will not work because the “this” refers to the HTMLButtonElement which does not have the count property needed. Therefore, the HTMLButtonElement is irrelevant at this moment which is why we need to use bind instead.



```
const button = document.getElementById("my-button");
button.addEventListener("click", howManyClicksTracker.incrementClick);
```

Using .bind( ) on the incrementClick to refer the “this” keyword to the howManyClicksTracker works because this contains a clicks property i.e. the howManyClicks object. This will also work because howManyClicksTracker is in the global scope. The clicks counter will now increment when we click on the HTML button.



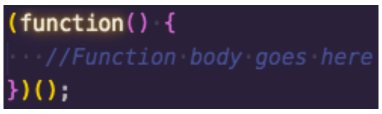
```
const howManyClicksTracker = {
  clicks: 0,
  incrementClick: function() {
    this.clicks = this.clicks + 1;
    console.log(this.clicks);
  }
};

const button = document.getElementById("my-button");
button.addEventListener("click", howManyClicksTracker.incrementClick.bind(howManyClicksTracker));
```

This is a very common use case of the .bind( ) method. It is similar to the .call( ) and the .apply ( ) methods but different in that you can can use it something like an eventListener because it does not get invoked immediately.

1.4 IIFEs

Immediately-Invoked Function Expression (IIFE) is a function that is executed right after it is created i.e. immediately. IIFE is commonly used when trying to avoid cluttering up the global namespace (i.e. cuts down on the use of global variables). Global variables are frowned upon not only in JavaScript but in programming in general.

The IIFE pattern is to to declare a function inside of round brackets and then invoke the anonymous function at the end using open and closing round brackets. The function does not have to be anonymous but this is the most common IIFE design pattern.

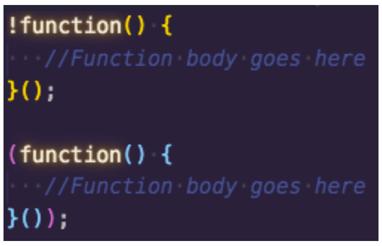


```
(function() {
  //Function body goes here
})();
```

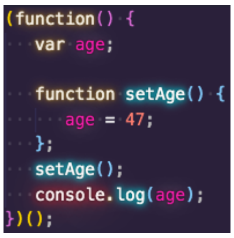
As soon as JavaScript sees an IIFE code it immediately executes the function. This is used for scoping reasons. Variables declared inside the IIFE is accessible inside the local scope of the function like any other functions. This helps to keep the name space clean.

There are other ways to write IIFEs that you should be familiar with. This is demonstrated with examples to the right.

IIFE’s help to keep functions private rather than in the global name space. This will prevent variables from leaking out and no other developer will accidentally change your variables down the line somewhere.



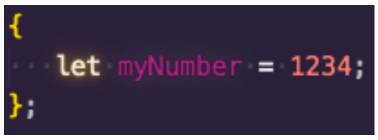
```
!function() {
  //Function body goes here
}();

(function() {
  //Function body goes here
}());
```



```
(function() {
  var age;

  function setAge() {
    age = 47;
  };
  setAge();
  console.log(age);
})();
```

We can also have functions within functions in IIFE and both functions/ variables will not be in the global scope.

ES6 introduces a much simpler way to write IIFEs whereby the variable or functions are wrapped inside of curly brackets. The var variable does not work with this new ES6 IIFE and we must use the new ES6 const/let variables.



```
{
  let myNumber = 1234;
};
```

The var variable is function scope and therefore if the var is not inside a function then it may leak out while both const and let variables are not function scoped and therefore does not leak outside its scope.

END