Encapsulation in JavaScript:

What is Encapsulation?

Encapsulation is one of the main concepts in object oriented programming. It allows an object to group both private and public members under a single name. All the object oriented programming languages support this. Since JavaScript is also an object oriented programming language, it supports it too.

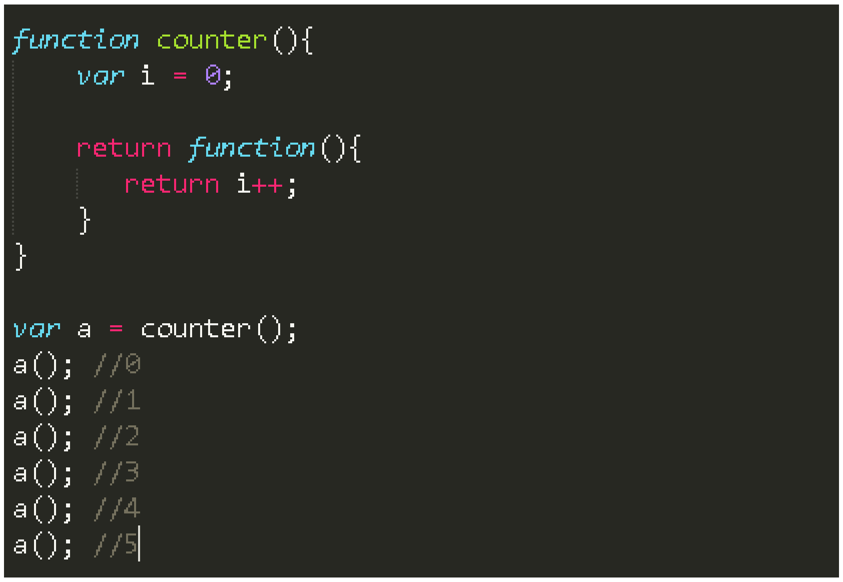
How to achieve it in JavaScript?

Encapsulation in JavaScript can be achieved by using Closures. Now the next question is what is closure?

If you have a function within a function, execution of the inner function will create a scope inside of the outer function- a nested scope. Because the inside function scope is enclosed by the outer function scope, the inner function scope is called a closure. Remember, to be a closure, you don't have to return a function; you just have to call a function inside a function.

Technically, Closure isthe ability to treat functions as values, combined with the fact that local variables are "re-created" every time a function is called.

Closure Example



```
function counter(){
    var i = 0;

    return function(){
        return i++;
    }
}

var a = counter();
a(); //0
a(); //1
a(); //2
a(); //3
a(); //4
a(); //5
```

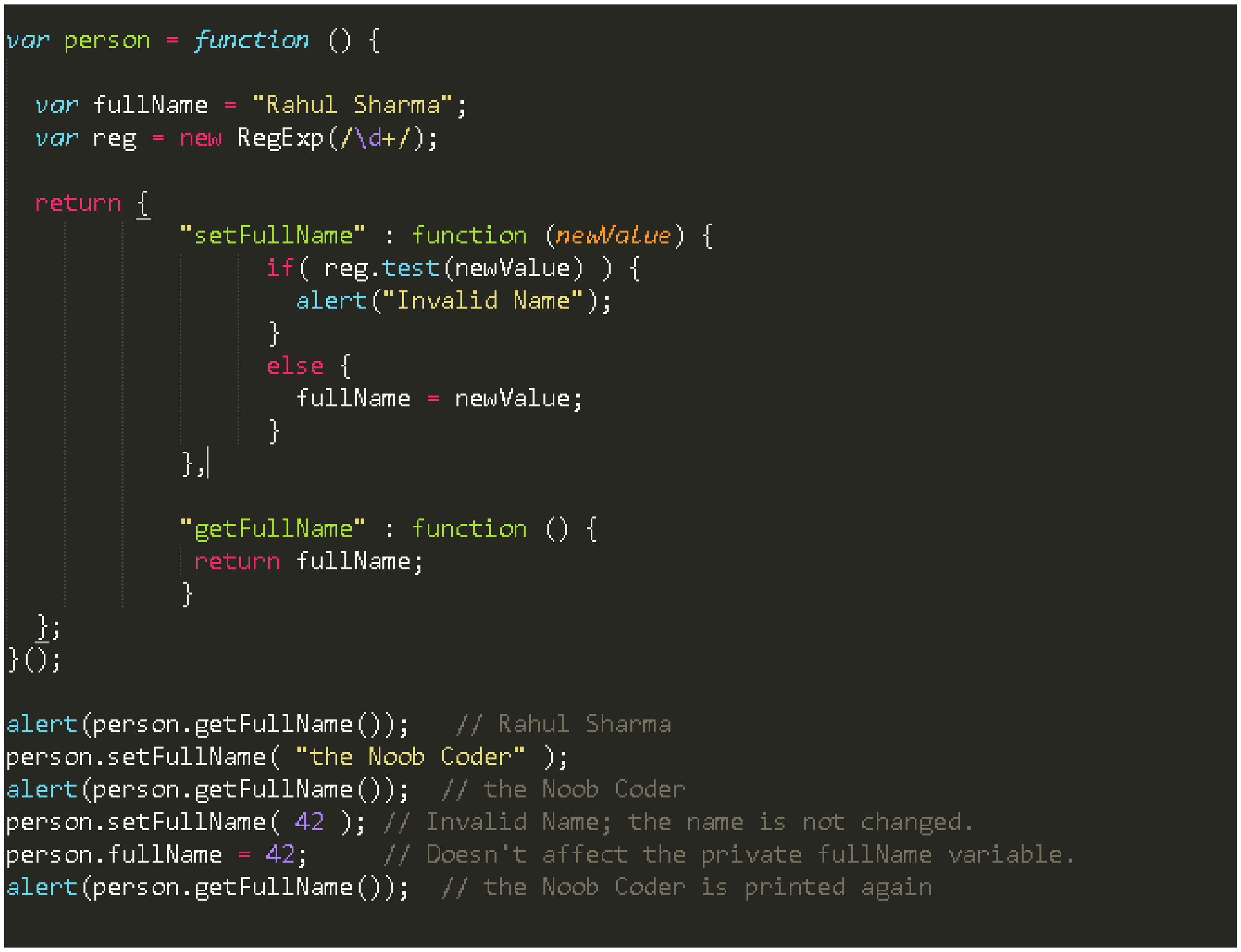
Step-1: In the above code, we execute a counter function and assign the result of the counter to a variable "a". Hence, the value of "a" will be the inner function (returning function). Just remember the inner function returns the value of "i" and also increase the value of "i". Hence, when you execute the function "a", you will get the value of "i".

Step-2: Now if you execute function "a" (execute inner function), it will return 0 (value of i) and will increase it by 1. If you execute function "a" again it will return 1 (current value of i) and will increase the value of "i" by 1. Hence, in your subsequent call you will get 2, 3, 4, 5.This means, closure holds the state of the local/private variable in the outer scope. And every time, you call it you get the value of "i" increased by 1.

This means, each closure maintains separate private variables of the outer function.

Closure saves the state of the outer function variable but does not expose those private variables. Now you can hide implementation detail while using closure. This is called encapsulation.

Creating private variables using Closure



```
var person = function () {

  var fullName = "Rahul Sharma";
  var reg = new RegExp(/\d+/);

  return {
            "setFullName" : function (newValue) {
                if( reg.test(newValue) ) {
                  alert("Invalid Name");
                }
                else {
                  fullName = newValue;
                }
            },

            "getFullName" : function () {
             return fullName;
            }
  };
}();

alert(person.getFullName());   // Rahul Sharma
person.setFullName( "the Noob Coder" );
alert(person.getFullName());  // the Noob Coder
person.setFullName( 42 ); // Invalid Name; the name is not changed.
person.fullName = 42;     // Doesn't affect the private fullName variable.
alert(person.getFullName());  // the Noob Coder is printed again
```

The trick is to remember to call the anonymous function immediately after its definition, and assign the return value (the inner object) to the outside variable (person), rather than simply assigning the anonymous function itself to the outside variable. This is done by using the invocation operator, ()**,** at the end of the function.

Inheritance

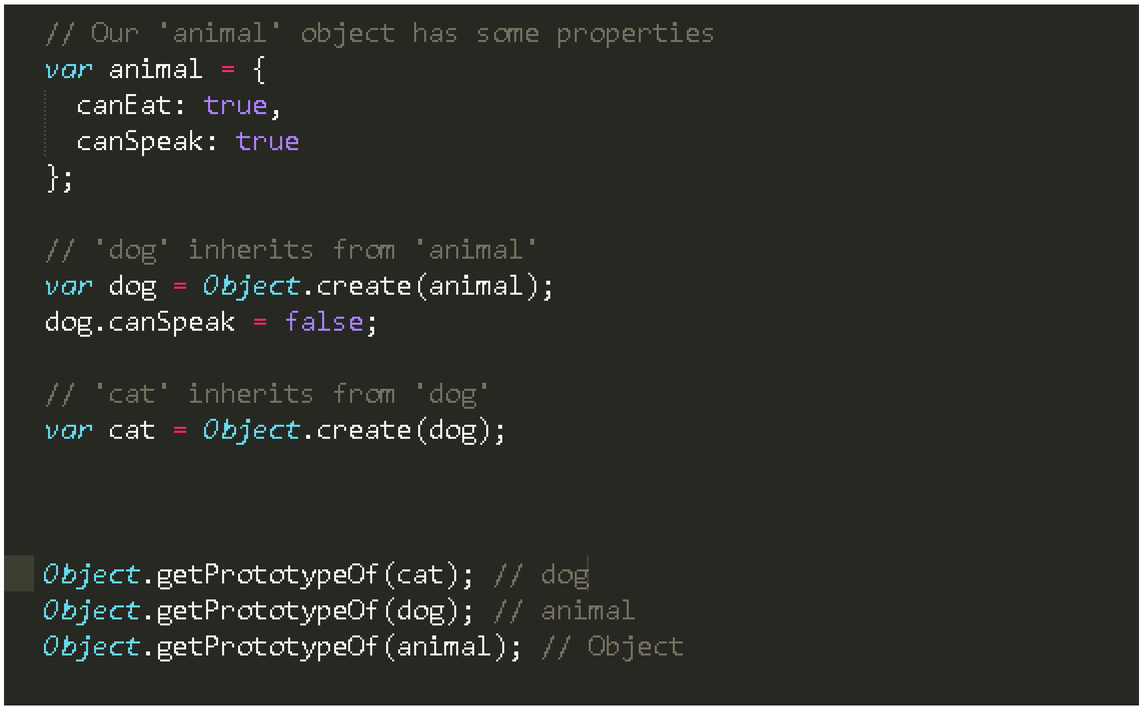
In classical languages, like Java, instances are created from classes. JavaScript differs in that it has prototypal inheritance, in which there are no classes.Instead, objects inherit from other objects.

To have a better understanding of the concepts behind prototypal inheritance, you must unlearn what you have learned.In the latest version of JavaScript available in all modern browsers (Chrome, Firefox, Safari, Opera, and IE9+), we have new syntax for creating object-to-object inheritance:

var childObject = Object.create(parentObject);

Inheritance in Classless JavaScript

A very simple example :



```
// Our 'animal' object has some properties
var animal = {
  canEat: true,
  canSpeak: true
};

// 'dog' inherits from 'animal'
var dog = Object.create(animal);
dog.canSpeak = false;

// 'cat' inherits from 'dog'
var cat = Object.create(dog);



Object.getPrototypeOf(cat); // dog
Object.getPrototypeOf(dog); // animal
Object.getPrototypeOf(animal); // Object
```

In this simple example, we’ve been able to set up a prototype chain without using a single constructor or “new” keyword.

So how does a prototype chain work? When we try to read a property from our new cat object, it checks the object itself and, if it didn’t find the property, it traverses up the prototype chain, checking each of its prototype objects in order until it finds the first occurrence of the property.

All this happens when we ask for the value of a property from an object.

Cat.canEat;

In order to properly evaluate the value of 'canEat' on 'cat', the JS engine does the following:

1. Checks 'cat' for the 'canEat' property, but find nothing.
2. Checks 'dog', but doesn't find the 'canEat' property.
3. Checks 'animal' and finds the 'canEat' property, so returns its value, which is 'true'.

Modifying the objects on runtime

The interesting thing in JavaScript is it can be modified on runtime. For Example, in order to tell if 'dog' or 'cat' is dangerous or not, we could do the following:

Dog.isDangerous = false;

Cat.isDangerous; //false, checks in dog

Where is Super?

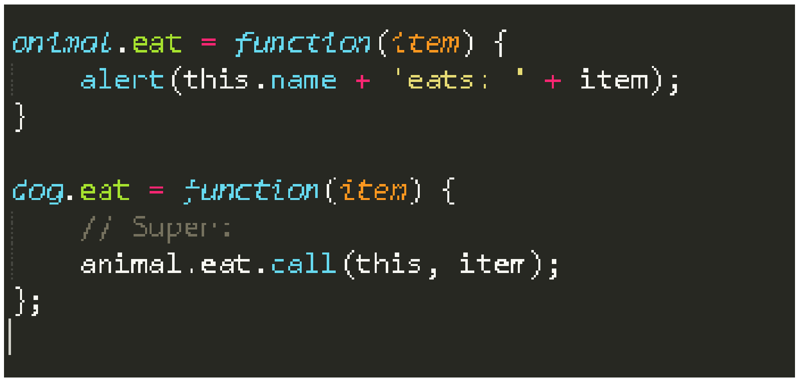
In classical languages, when overriding methods, you can run the method from the parent class in the current context.

In JavaScript we have even more options: we can run any function in any context.

How do we achieve this? Using JavaScript's 'call' and 'apply' methods which are available on all functions. Appropriately enough, these are available because they exist on the 'Function' prototype.

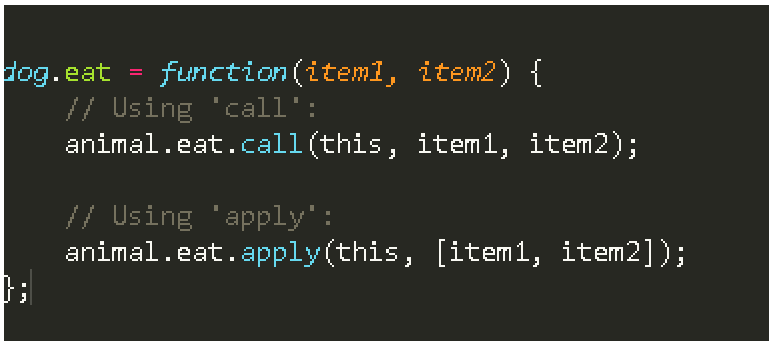
They both allow us to run a function in a specific context which we provide as the first parameter, along with any arguments we wish to pass to the function.

Okay, let's achieve this with our 'dog' example:



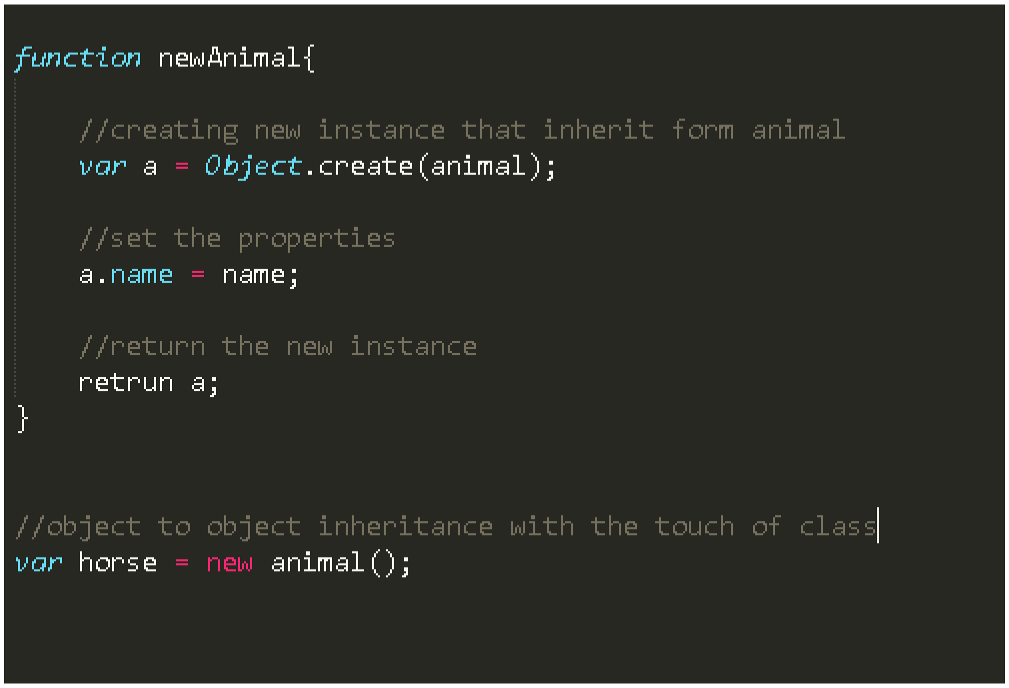
```
animal.eat = function(item) {
    alert(this.name + 'eats: ' + item);
}

dog.eat = function(item) {
    // Super:
    animal.eat.call(this, item);
};
```

If it takes multiple arguments, then call and apply will look like this:



```
dog.eat = function(item1, item2) {
    // Using 'call':
    animal.eat.call(this, item1, item2);

    // Using 'apply':
    animal.eat.apply(this, [item1, item2]);
};
```

Using Constructor

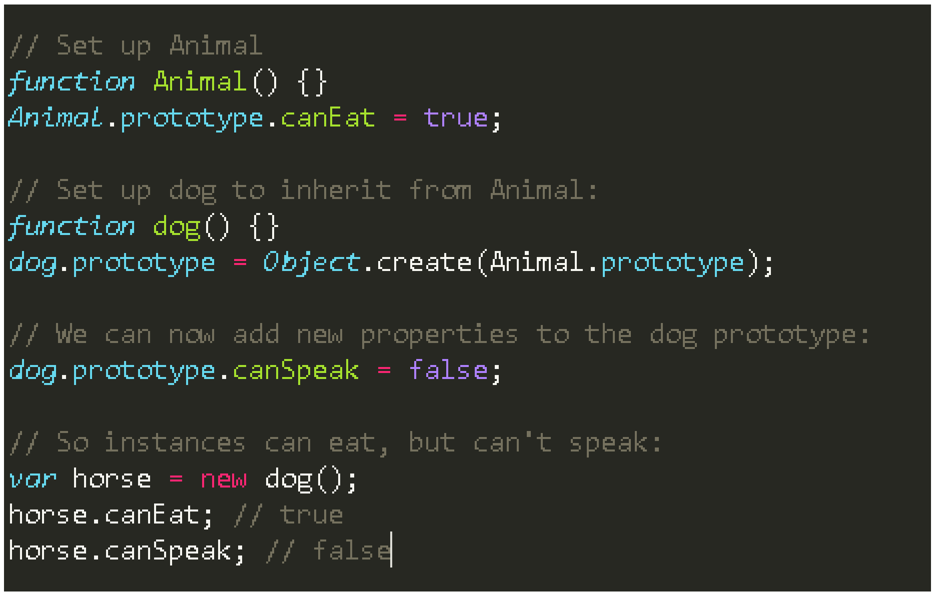


```
function newAnimal{

    //creating new instance that inherit form animal
    var a = Object.create(animal);

    //set the properties
    a.name = name;

    //return the new instance
    retrun a;
}


//object to object inheritance with the touch of class
var horse = new animal();
```

Here, animal is not obliviously not a class but a function.

Animal is used as a constructor because it has been invoked by a ‘new’ keyword.

The funny thing about JavaScript is that any function can be used as a constructor, so there’s nothing special about our ‘animal’function that makes it different from any other function.



```javascript
// Set up Animal
function Animal() {}
Animal.prototype.canEat = true;

// Set up dog to inherit from Animal:
function dog() {}
dog.prototype = Object.create(Animal.prototype);

// We can now add new properties to the dog prototype:
dog.prototype.canSpeak = false;

// So instances can eat, but can't speak:
var horse = new dog();
horse.canEat; // true
horse.canSpeak; // false
```

So, what is happening?

1. Creates a new object **horse** which inherits from **Animal. Prototype**.
2. Run the **Animal** function against horse and returns **horse**.