Java Script Concepts

3 ways to define a JavaScript class

September 29th, 2006. Tagged: JavaScript

Introduction

JavaScript is a very flexible object-oriented language when it comes to syntax. In this article you can find three ways of defining and instantiating an object. Even if you have already picked your favorite way of doing it, it helps to know some alternatives in order to read other people's code.

It's important to note that there are no classes in JavaScript. Functions can be used to somewhat simulate classes, but in general JavaScript is a class-less language. Everything is an object. And when it comes to inheritance, objects inherit from objects, not classes from classes as in the "class"-ical languages.

1. Using a function

This is probably one of the most common ways. You define a normal JavaScript function and then create an object by using the `new` keyword. To define properties and methods for an object created using `function()`, you use the `this` keyword, as seen in the following example.

```
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = getAppleInfo;
}

// anti-pattern! keep reading...
function getAppleInfo() {
    return this.color + ' ' + this.type + ' apple';
}
```

To instantiate an object using the Apple *constructor function*, set some properties and call methods you can do the following:

```
var apple = new Apple('macintosh');
apple.color = "reddish";
alert(apple.getInfo());
```

1.1. Methods defined internally

In the example above you see that the method getInfo() of the Apple "class" was defined in a separate function getAppleInfo(). While this works fine, it has one drawback – you may end up defining a lot of these functions and they are all in the "global namespece". This means you may have naming conflicts if you (or another library you are using) decide to create another function with the same name. The way to prevent pollution of the global namespace, you can define your methods within the constructor function, like this:

```
function Apple (type) {
    this.type = type;
    this.color = "red";
    this.getInfo = function() {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

Using this syntax changes nothing in the way you instantiate the object and use its properties and methods.

1.2. Methods added to the prototype

A drawback of 1.1. is that the method getInfo() is recreated every time you create a new object. Sometimes that may be what you want, but it's rare. A more inexpensive way is to add getInfo() to the *prototype* of the constructor function.

```
function Apple (type) {
    this.type = type;
    this.color = "red";
}

Apple.prototype.getInfo = function() {
    return this.color + ' ' + this.type + ' apple';
};
```

Again, you can use the new objects exactly the same way as in 1. and 1.1.

2. Using object literals

Literals are shorter way to define objects and arrays in JavaScript. To create an empty object using you can do:

```
var o = {};
```

instead of the "normal" way:

```
var o = new Object();
```

For arrays you can do:

```
var a = [];
```

instead of:

```
var a = new Array();
```

So you can skip the class-like stuff and create an instance (object) immediately. Here's the same functionality as described in the previous examples, but using object literal syntax this time:

```
var apple = {
    type: "macintosh",
    color: "red",
    getInfo: function () {
        return this.color + ' ' + this.type + ' apple';
    }
}
```

In this case you don't need to (and cannot) create an instance of the class, it already exists. So you simply start using this instance.

```
apple.color = "reddish";
alert(apple.getInfo());
```

Such an object is also sometimes called *singleton*. In "classical" languages such as Java, *singleton* means that you can have only one single instance of this class at any time, you cannot create more objects of the same class. In JavaScript (no classes, remember?) this concept makes no sense anymore since all objects are singletons to begin with.

3. Singleton using a function

Again with the singleton, eh? 🙂

The third way presented in this article is a combination of the other two you already saw. You can use a function to define a singleton object. Here's the syntax:

```
var apple = new function() {
    this.type = "macintosh";
    this.color = "red";
    this.getInfo = function () {
        return this.color + ' ' + this.type + ' apple';
    };
}
```

So you see that this is very similar to 1.1. discussed above, but the way to use the object is exactly like in 2.

```
apple.color = "reddish";
alert(apple.getInfo());
```

`new function(){...}` does two things at the same time: define a function (an anonymous constructor function) and invoke it with `new`. It might look a bit confusing if you're not used to it and it's not too common, but hey, it's an option, when you really want a constructor function that you'll use only once and there's no sense of giving it a name.

Summary

You saw three (plus one) ways of creating objects in JavaScript. Remember that (despite the article's title) there's no such thing as a class in JavaScript. Looking forward to start coding using the new knowledge? Happy JavaScript-ing!

https://www.phpied.com/3-ways-to-define-a-javascript-class/

Object-Oriented programming with JavaScript

Object-Oriented programming is one of the widely used programming paradigm that uses abstraction to create model based on real world. It is a model organized around “objects” rather than “actions” and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data. Object-oriented programming (OOP) uses **“objects”** – data structures consisting of **datafields** and **methods** – and their interactions to design applications and computer programs. Each object can be seen as a tiny machine which is responsible for the set of task assign to it.

Today, many popular programming languages (such as Java, JavaScript, C#, C++, Python, PHP etc) support **object-oriented programming** (OOP).

JavaScript has strong object-oriented programming capabilities. Although there is differences in object-oriented capability of javascript compared to other languages.

Terminology

First let us see few terminologies that we use in object-oriented programming.

Class: Defines the characteristics of the Object.
Constructor: A method called at the moment of instantiation.
Object: An Instance of a Class.
Method: An Object capability as walk.
Property: An Object characteristic, such as color.
Inheritance: A Class can inherit characteristics from another Class.
Encapsulation: A Class defines only the characteristics of the Object, a method defines only how the method executes.
Abstraction: The conjunction of complex inheritance, methods, properties of an Object must be able to simulate a reality model.
Polymorphism: Different Classes might define the same method or property.

The Class in JavaScript

Unlike Java, C++ etc, JavaScript does not contains class statement. **JavaScript is a prototype-based language**. **JavaScript uses functions as classes**. Defining a class is as easy as defining a function. In the example below we define a new class called Car.

```
//Define the class Car
function Car() { }
```

The Object (Class Instance) in JavaScript

For creating instances of any class i.e. objects use **new** keyword. For example, in below code snippet we created two instances of class Car.

```
//Define the class Car
function Car() { }

var car1 = new Car();
var car2 = new Car();
```

The Constructor in JavaScript

In object oriented methodology, a Constructor is a method that is used to initiate the properties of any class instance. Thus the constructor gets called when the class is instantiated.
As in JavaScript there is no class keyword and the function serves as class definition, there is no need of defining constructor explicitly. The function defined for the class acts as constructor. For example, in following code snippet we have called an alert statement when class Car is instantiated.

```
//Define the class Car
function Car() {
		alert("Class CAR Instantiated");
}

var car1 = new Car();
var car2 = new Car();
```

The Property (object attribute) in JavaScript

Properties are the variable that are member of an object and can define the state of any instance of Class. Property of a class can be accessed within the class using **this** keyword. For example, in following code snippet we have assign a property *speed* to Car.

```
//Define the class Car
function Car(speed) {
        alert("Class CAR Instantiated");
        this.speed = speed;
}

var car1 = new Car(40);
var car2 = new Car(60);

alert("Car1 Speed: " + car1.speed);
alert("Car2 Speed: " + car2.speed);
```

One thing we should note here is that for inheritance works correctly, the Properties should be set in the prototype property of the class (function). The above example becomes:

```
//Define the class Car
function Car(speed) {
        alert("Class CAR Instantiated");
        this.speed = speed;
}
Car.prototype.speedLimit=240;

var car1 = new Car(40);
var car2 = new Car(60);

alert("Car1 Speed: " + car1.speed);
alert("Car2 Speed: " + car2.speed);
alert("Car2 Speed: " + car2. speedLimit);
```

The methods in JavaScript

To define methods in a Class, all you have to do is just define a attribute and assign an function to it. For example, in below code snippet we defined a method `setSpeed()` for class `Car`.

```
//Define the class Car
function Car(speed) {
                alert("Class CAR Instantiated");
                this.speed = speed;
}
Car.prototype.speedLimit=240;
Car.prototype.setSpeed = function(speed) {
                this.speed = speed;
                alert("Car speed changed");
}

var car1 = new Car(40);
var car2 = new Car(60);

car1.setSpeed(120);
car2.setSpeed(140);
```

Inheritance in JavaScript

JavaScript supports single class inheritance. To create a child class that inherit parent class, we create a Parent class object and assign it to the Child class. In following example we created a child class Ferrari from parent class Car.

```
//Define the Car class
function Car() { }
Car.prototype.speed= 'Car Speed';
Car.prototype.setSpeed = function(speed) {
                this.speed = speed;
                alert("Car speed changed");
}


//Define the Ferrari class
function Ferrari() { }
Ferrari.prototype = new Car();


// correct the constructor pointer because it points to Car
Ferrari.prototype.constructor = Ferrari;

// replace the setSpeed method
Ferrari.prototype.setSpeed = function(speed) {
                this.speed = speed;
                alert("Ferrari speed changed");
}

var car = new Ferrari();
car.setSpeed();
```

Encapsulation in JavaScript

In JavaScript, encapsulation is achieved by the inheritance. The child class inherit all the properties and methods of parent class and needs to override the method only that needs to be changed. This encapsulation by which every class inherits the methods of its parent and only needs to define things it wishes to change.

Update: A new tutorial cover these Javascript OOPs topics in details. Please refer: **JavaScript 101: Objects and Functions**

\"In JavaScript, encapsulation is achieved by the inheritance. The child class inherit all the properties and methods of parent class\"

In JavaScript, when you create a subclass from an existing class, only the *public* and *privileged* members are passed on (public/privileged members are created using the this keyword). You can also implement encapsulation with private members using closures.

In your example, this.speed is public and accessible, you can change it after instantiating the object. To truly protect speed, you\'d want to do something like this (I hope the spacing will be preserved):

```
var Car = function(mph) {
                //private attribute
                var speed;

                //privileged methods
                this.getSpeed = function() {
                return speed;
        }
        this.setSpeed = function(mph) {
                speed = mph;
        }

        this.setSpeed(mph);
} ;
Car.prototype = {
        //public, non-privileged methods go here
        whatever: function () {}
};
```

speed is private, you can only access it using the public getter and setter.

You can subclass Car and still access speed, but only through the privileged methods (getSpeed and setSpeed) because the privileged methods will be passed on (they are publicly accessible via the this keyword). No instance methods in the subclass will have *direct* access to speed, though – you have to go through the existing privileged methods (getSpeed and setSpeed).

\"JavaScript supports single class inheritance.\"

JavaScript also has prototypal inheritance which uses objects instead of defining a class structure and technically, you can have multiple inheritance via augmentation/mixins.

Reply

Well i just want to point that JavaScript instantiation method does not mean copy public methods from the prototype, this process is an implementation of decoration pattern. Every instance creates an empty object that decorates the constructor.prototype object, so actually instance or inheritance (actually done by instance), its a decoration process, this is the process by which JavaScript allows to have dynamic classes, and when you augment the class all previous instances of that class get the method.

i wrote an article on the same topic of Object Oriented JavaScript in Mozilla https://developer.mozilla.org/en/Introduction_to_Object-Oriented_JavaScript

http://viralpatel.net/blogs/object-oriented-programming-with-javascript/

JavaScript 101: Objects and Functions

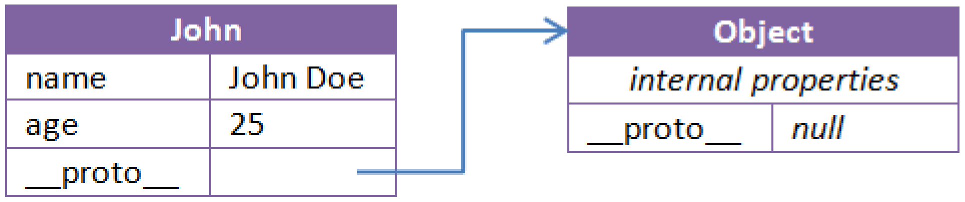
JavaScript Object and their Prototypes

Objects in JavaScript is nothing but collection of attributes (or properties to be more inline with ECMA definition). And a property is defined as a key-value pair. Object in JavaScript can be created using many ways, but for now we are gonna use a simple one.
Lets create a Person called John:

```
var John = {
         'name'  : 'John Doe',
         'age'   : '25'
};
```

This object has two property ‘name’ and ‘age’. Note that the value of a property is not at all limited to primitive types but can be any

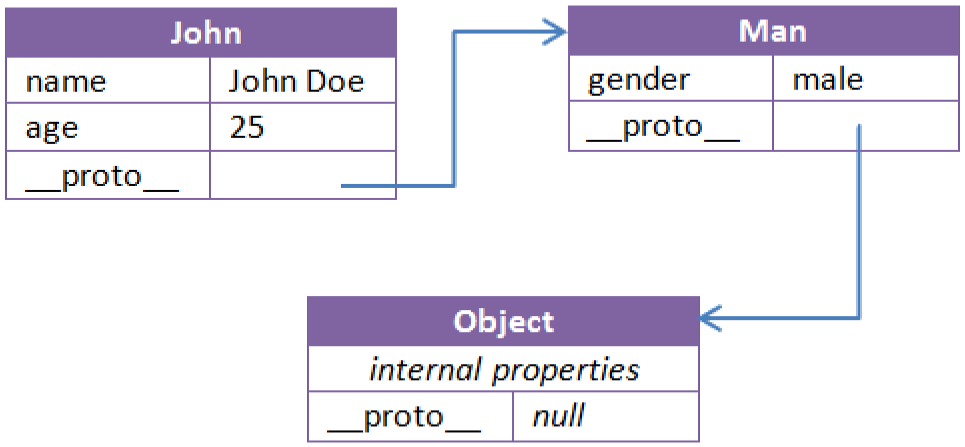
other javascript object. So for example ‘John’ may have another property ‘address’, whose value can be an Array of different address objects. Apart from the properties defined in object, every JavaScript object has another secret internal property called ‘__proto__’. This property points to the prototype of the John. But what is the prototype of John? Well for John here, its JavaScript’s parent of all objects, The Object. And what is the prototype of The Object?
Its *null*.
Below image might make this more clear:



Prototype Chaining and Inheritance

We now know that each object in JavaScript has another __proto__ property which refers to its prototype. If I am to create an object, and explicitly assign __proto__ another object… what would that mean?

```
var Man = {
  gender: 'male'
};
var John = {
  name: 'John Doe',
  age: 25,
  __proto__: Man
}
```



This would mean John *prototypically inherits* from Man. All properties of Man can be directly accessed through John, this is because whenever any property is accessed via John and if it is not found in John it will be looked in the prototype of John.

```
console.log(John.name); // John Doe
console.log(John.age); // 25
console.log(John.gender); // male
```

Further, this prototypical inheritance is not limited to two levels and can be extended to any level. This sort of chaining is called **Prototype Chaining**.

The rule to resolve any property is: If a property is not found in an object, it is looked into the prototype chain of the object all the way to Object unless found earlier. If property is not found, we get a property undefined error.

Function

A function is a special type of Object which may be invoked as a routine. A function may have properties, and some lines of code which determines the behavior of the function. For example, we can have a function named isEven() which takes a number as parameter and tell us whether a number is even or not.

```
function isEven(no){
  return no % 2 === 0;
}
```

Further functions can be used to create Objects acting as a Constructor for the class. All that is required is to use `new` operator when calling that function. For example, I can write following Person function to create different Person objects:

```
function Person(name, age){
  this.name = name;
  this.age = age;
  this.shoutYourName = function(){
                          alert('My name is ' + name +'!');
                    };
}
//create Objects
var John = new Person('John'25);
var Donn = new Person('Donn'29);

console.log(John.name);  //John
console.log(John.age);  //25
John.shoutYourName();   //alerts 'My name is John!'
```

Now the question is what gets create and stored in memory above functions are created? What's the prototype of it? And how is it that same functions can act as Constructors to create new objects?

To answer these question,
A function that is called using *new* operator for creation of new objects is called a Constructor Function.
Creating a function creates an object of type `Function`. That means, any new function will have its __proto__ point to prototype of 'Function'.
As a side note: *Function is a global internal Object, it doesn't have any properties of its own but inherits them from Function.prototype.*
Further, objects of type Function have another property called 'prototype' pointing to the prototype of the objects which will be created using this function.

In case of above example:
1. Person is an object of type Function. And this also means Person has its __proto__ property pointed to Function.prototype
2. Person has a property ‘prototype’ pointing to an object which will be assigned to __proto__ property of objects created by Person, in our case, John and Donn. Lets call this prototype object Person.prototype. So this would mean any objects created using Person will be prototypically inherited from Person.

```
console.log(John.__proto__ === Person.prototype)    //true
console.log(Donn.__proto__ === Person.prototype)    //true
console.log(Person.__proto__ === Function.prototype) //true
```

Structure of Person.prototype

As already mentioned above, all objects created using Person will have their __proto__ pointing to Person.prototype. This means, these objects will Prototypically inherit from Person.prototype

1. __proto__ of Person.prototype points to Object.
2. And extra property added to Person.prototype will be prototypically available to objects created by calling new Person(). And using `this` reference inside properties of Person.prototype will refer to properties of Person. So, after creating the objects John and Donn, if I am to add below code:

```
Person.prototype.getAllCapsName = function(){
return this.name.toUpperCase();
};
```

this property `getAllCapsName` will be immediately available in already created objects ‘John’ and ‘Donn’. And that’s because, saying it again, __proto__ of John and Donn is pointing to Person.prototype object,i.e, John and Donn prototypically-inherits from Person.

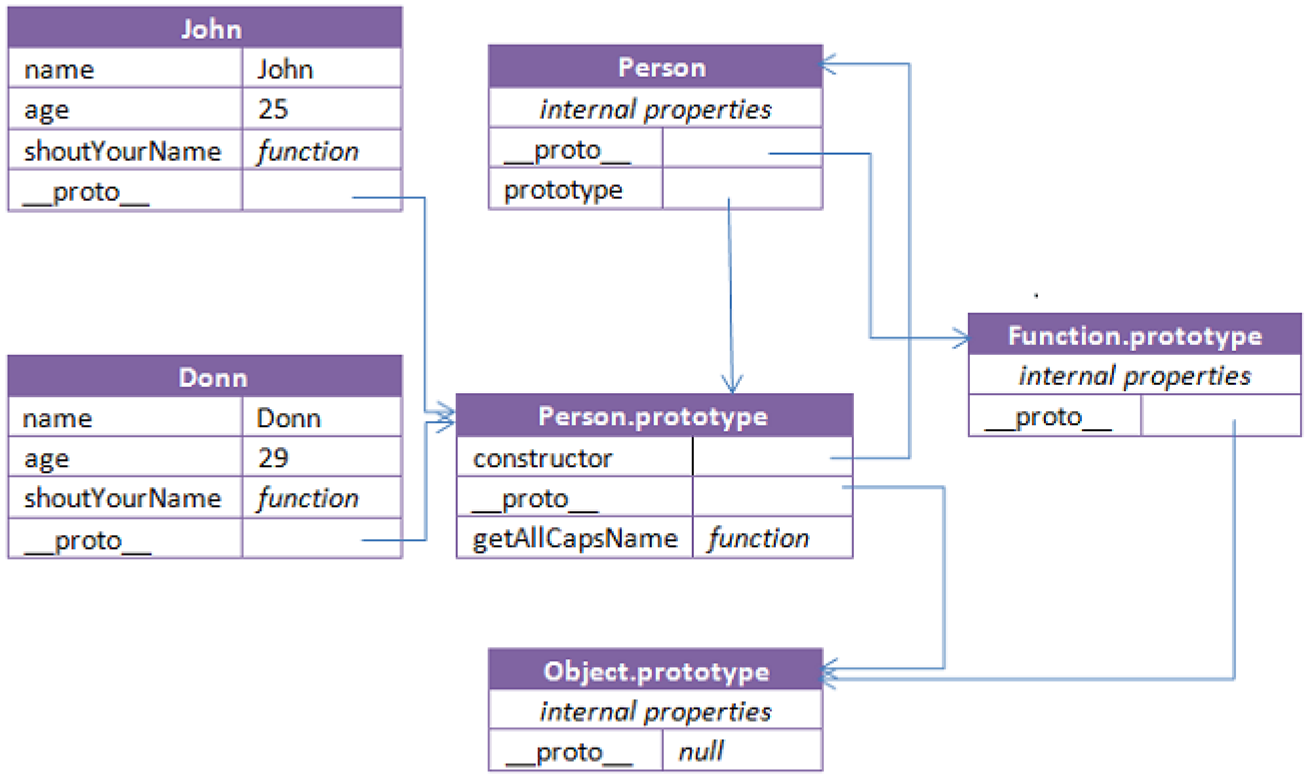
```
console.log(John.getAllCapsName()) //JOHN
console.log(John.__proto__.getAllCapsName()) //Can't access property of John using
this.
```

```
                                //this refers to Person.prototype object
console.log(Person.prorotype.getAllCapsName()) //Can't access property using this.
```

6. Person.prototype has another special property called `constructor`, which points to the original function object Person. And due to above point:

```
7.    console.log(Person.prototype.constructor===Person) //true
8.     console.log(John.constructor)// print Person function
9.     console.log(Person.prototype.constructor)// print Person function
10.    console.log(John.constructor===Person) //true
11.    console.log(John.__proto__.constructor===Person.prototype.constructor) //true
12.    console.log(John.__proto__.constructor===Person.prototype.constructor) //true
```

In case things are still hazy, image below might help:



Now since behavior of all objects created using one Constructor Function will be same, it makes more sense to add the behavior in the prototype of Constructor Function than in the Constructor Function. I.e. to add functions properties inside Person.prototype than in Person. This is because any function-property in Person (say shoutYourName()) will be copied in John and Donn. But, any function-property in Person.prototype will be available to John and

Donn through __proto__ reference.
So if I am to change the definition of shoutYourName for John, behavior of Donn will not be affected. But it’s not the case with getAllCapsName().

Let see this working:

```
var Person = function(name, age){
    this.name = name;
    this.age = age;
    this.shoutYourName = function(){
        return 'SHOUTING '+ this.name;
    };
};

var John = new Person('John',25);
var Donn = new Person('Donn',25);

console.log(John.shoutYourName()) // SHOUTING John
console.log(Donn.shoutYourName()) // SHOUTING Donn

Person.prototype.shhhYourName = function(){
    return 'shhh ' + this.name;
};

console.log(John.shhhYourName()) // shhh John
console.log(Donn.shhhYourName()) // shhh Donn

//changing definition of only John
John.shoutYourName = function(){
    return 'SHOUTING ' + this.name + '!!!!!';
};

//changing definition in prototype
Person.prototype.shhhYourName = function(){
    return 'shhh ' + this.name + '!!!!!';
};

console.log(John.shoutYourName()) // SHOUTING John!!!!!
console.log(Donn.shoutYourName()) // SHOUTING Donn

console.log(John.shhhYourName()) // shhh John!!!!!
console.log(Donn.shhhYourName()) // shhh Donn!!!!!
```

Hope it helped to have a better understanding of Objects and Functions is JavaScript.
and The End. :)

http://viralpatel.net/blogs/javascript-objects-functions/

Closures are functions that refer to independent (free) variables (variables that are used locally, but defined in an enclosing scope). In other words, these functions 'remember' the environment in which they were created.

Lexical scopingEDIT

Consider the following:

```
function init() {
  var name = 'Mozilla'; // name is a local variable created by init
  function displayName() { // displayName() is the inner function, a closure
    alert(name); // use variable declared in the parent function
  }
  displayName();
}
init();
```

init() creates a local variable called name and a function called displayName(). The displayName() function is an inner function that is defined inside init() and is only available within the body of the init() function. The displayName() function has no local variables of its own. However, because inner functions have access to the variables of outer functions, displayName() can access the variable name declared in the parent function, init().

Run the code and notice that the alert() statement within the displayName() function successfully displays the value of the name variable, which is declared in its parent function. This is an example of *lexical scoping*, which describes how a parser resolves variable names when functions are nested. The word "lexical" refers to the fact that lexical scoping uses the location where a variable is declared within the source code to determine where that variable is available. Nested functions have access to variables declared in their outer scope.

ClosureEDIT

Now consider the following example:

```
function makeFunc() {
  var name = 'Mozilla';
  function displayName() {
    alert(name);
  }
  return displayName;
}

var myFunc = makeFunc();
myFunc();
```

Running this code has exactly the same effect as the previous example of the init() function above: this time, the string "Mozilla" will be displayed in a JavaScript alert box. What's different — and interesting — is that the displayName() inner function is returned from the outer function before being executed.

At first glance, it may seem unintuitive that this code still works. In some programming languages, the local variables within a function exist only for the duration of that function's execution. Once makeFunc() has finished executing, you might expect that the name variable would no longer be accessible. However, because the code still works as expected, this is obviously not the case in JavaScript.

The reason is that functions in JavaScript form 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 that the closure was created. In this case, myFunc is a reference to the instance of the function displayName created when makeFuncis run. The instance of displayName maintains a reference to its lexical environment, within which the variable name exists. For this reason, when myFunc is invoked, the variable nameremains available for use and "Mozilla" is passed to alert.

Here's a slightly more interesting example — a makeAdder function:

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}

var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

In this example, we have defined a function makeAdder(x), which takes a single argument, x, and returns a new function. The function it returns takes a single argument, y, and returns the sum of x and y.

In essence, makeAdder is a function factory — it creates functions which can add a specific value to their argument. In the above example we use our function factory to create two new functions — one that adds 5 to its argument, and one that adds 10.

add5 and add10 are both closures. They share the same function body definition, but store different lexical environments. In add5's lexical environment, x is 5, while in the lexical environment for add10, x is 10.

Practical closuresEDIT

Closures are useful because they let you associate some data (the lexical environment) with a function that operates on that data. This has obvious parallels to object oriented programming, where objects allow us to associate some data (the object's properties) with one or more methods.

Consequently, you can use a closure anywhere that you might normally use an object with only a single method.

Situations where you might want to do this are particularly common on the web. Much of the code we write in front-end JavaScript is event-based — we define some behavior, then attach it to an event that is triggered by the user (such as a click or a keypress). Our code is generally attached as a callback: a single function which is executed in response to the event.

For instance, suppose we wish to add some buttons to a page that adjust the text size. One way of doing this is to specify the font-size of the body element in pixels, then set the size of the other elements on the page (such as headers) using the relative em unit:

```
body {
  font-family: Helvetica, Arial, sans-serif;
  font-size: 12px;
}

h1 {
  font-size: 1.5em;
}

h2 {
  font-size: 1.2em;
}
```

Our interactive text size buttons can change the font-size property of the body element, and the adjustments will be picked up by other elements on the page thanks to the relative units.

Here's the JavaScript:

```
function makeSizer(size) {
  return function() {
    document.body.style.fontSize = size + 'px';
  };
}

var size12 = makeSizer(12);
var size14 = makeSizer(14);
var size16 = makeSizer(16);
```

size12, size14, and size16 are now functions which will resize the body text to 12, 14, and 16 pixels, respectively. We can attach them to buttons (in this case links) as follows:

```
document.getElementById('size-12').onclick = size12;
document.getElementById('size-14').onclick = size14;
document.getElementById('size-16').onclick = size16;
<a href="#" id="size-12">12</a>
<a href="#" id="size-14">14</a>
<a href="#" id="size-16">16</a>
```

Emulating private methods with closuresEDIT

Languages such as Java provide the ability to declare methods private, meaning that they can only be called by other methods in the same class.

JavaScript does not provide a native way of doing this, but it is possible to emulate private methods using closures. Private methods aren't just useful for restricting access to code: they also provide a powerful way of managing your global namespace, keeping non-essential methods from cluttering up the public interface to your code.

The following code illustrates how to use closures to define public functions that can access private functions and variables. Using closures in this way is known as the module pattern:

```
var counter = (function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  };
})();

console.log(counter.value()); // logs 0
counter.increment();
counter.increment();
console.log(counter.value()); // logs 2
counter.decrement();
console.log(counter.value()); // logs 1
```

In previous examples, each closure has had its own lexical environment. Here, though, we create a single lexical environment that is shared by three functions: counter.increment, counter.decrement, and counter.value.

The shared lexical environment is created in the body of an anonymous function, which is executed as soon as it has been defined. The lexical environment contains two private items: a variable called privateCounter and a function called changeBy. Neither of these private items can be accessed directly from outside the anonymous function. Instead, they must be accessed by the three public functions that are returned from the anonymous wrapper.

Those three public functions are closures that share the same environment. Thanks to JavaScript's lexical scoping, they each have access to the privateCounter variable and changeByfunction.

You'll notice we're defining an anonymous function that creates a counter, and then we call it immediately and assign the result to the counter variable. We could store this function in a separate variable makeCounter and use it to create several counters.

```
var makeCounter = function() {
  var privateCounter = 0;
  function changeBy(val) {
    privateCounter += val;
  }
  return {
    increment: function() {
      changeBy(1);
    },
    decrement: function() {
      changeBy(-1);
    },
    value: function() {
      return privateCounter;
    }
  }
};

var counter1 = makeCounter();
var counter2 = makeCounter();
alert(counter1.value()); /* Alerts 0 */
counter1.increment();
counter1.increment();
alert(counter1.value()); /* Alerts 2 */
counter1.decrement();
alert(counter1.value()); /* Alerts 1 */
alert(counter2.value()); /* Alerts 0 */
```

Notice how each of the two counters, counter1 and counter2, maintains its independence from the other. Each closure references a different version of the privateCounter variable through its own closure. Each time one of the counters is called, its lexical environment changes by changing the value of this variable; however changes to the variable value in one closure do not affect the value in the other closure.

Using closures in this way provides a number of benefits that are normally associated with object oriented programming -- in particular, data hiding and encapsulation.

Creating closures in loops: A common mistakeEDIT

Prior to the introduction of the let keyword in ECMAScript 2015, a common problem with closures occurred when they were created inside a loop. Consider the following example:

```
<p id="help">Helpful notes will appear here</p>
<p>E-mail: <input type="text" id="email" name="email"></p>
<p>Name: <input type="text" id="name" name="name"></p>
<p>Age: <input type="text" id="age" name="age"></p>
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
      showHelp(item.help);
    }
  }
}

setupHelp();
```

The helpText array defines three helpful hints, each associated with the ID of an input field in the document. The loop cycles through these definitions, hooking up an onfocus event to each one that shows the associated help method.

If you try this code out, you'll see that it doesn't work as expected. No matter what field you focus on, the message about your age will be displayed.

The reason for this is that the functions assigned to onfocus are closures; they consist of the function definition and the captured environment from the setupHelp function's scope. Three closures have been created by the loop, but each one shares the same single lexical environment, which has a variable with changing values (item.help). The value of item.help is determined when the onfocus callbacks are executed. Because the loop has already run its course by that time, the item variable object (shared by all three closures) has been left pointing to the last entry in the helpText list.

One solution in this case is to use more closures: in particular, to use a function factory as described earlier:

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function makeHelpCallback(help) {
  return function() {
    showHelp(help);
  };
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    var item = helpText[i];
    document.getElementById(item.id).onfocus = makeHelpCallback(item.help);
  }
}

setupHelp();
```

This works as expected. Rather than the callbacks all sharing a single lexical environment, the makeHelpCallback function creates *a new lexical environment* for each callback, in which helprefers to the corresponding string from the helpText array.

One other way to write the above using anonymous closures is:

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    (function() {
       var item = helpText[i];
       document.getElementById(item.id).onfocus = function() {
         showHelp(item.help);
       }
    })(); // Immediate event listener attachment with the current value of item (preserved
until iteration).
  }
}

setupHelp();
```

If you don't want to use more closures, you can use the let keyword introduced in ES2015 :

```
function showHelp(help) {
  document.getElementById('help').innerHTML = help;
}

function setupHelp() {
  var helpText = [
      {'id': 'email', 'help': 'Your e-mail address'},
      {'id': 'name', 'help': 'Your full name'},
      {'id': 'age', 'help': 'Your age (you must be over 16)'}
    ];

  for (var i = 0; i < helpText.length; i++) {
    let item = helpText[i];
    document.getElementById(item.id).onfocus = function() {
      showHelp(item.help);
    }
  }
}

setupHelp();
```

This example uses let instead of var, so every closure binds the block-scoped variable, meaning that no additional closures are required.

Performance considerationsEDIT

It is unwise to unnecessarily create functions within other functions if closures are not needed for a particular task, as it will negatively affect script performance both in terms of processing speed and memory consumption.

For instance, when creating a new object/class, methods should normally be associated to the object's prototype rather than defined into the object constructor. The reason is that whenever the constructor is called, the methods would get reassigned (that is, for every object creation).

Consider the following case:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
  this.getName = function() {
    return this.name;
  };

  this.getMessage = function() {
    return this.message;
  };
}
```

Because the previous code does not take advantage of the benefits of closures, we could instead rewrite it as follows:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype = {
  getName: function() {
    return this.name;
  },
  getMessage: function() {
    return this.message;
  }
};
```

However, redefining the prototype is not recommended. The following example instead appends to the existing prototype:

```
function MyObject(name, message) {
  this.name = name.toString();
  this.message = message.toString();
}
MyObject.prototype.getName = function() {
  return this.name;
};
MyObject.prototype.getMessage = function() {
  return this.message;
};
```

The above code can also be written in a cleaner way with the same result:

```
function MyObject(name, message) {
    this.name = name.toString();
    this.message = message.toString();
}
(function() {
    this.getName = function() {
        return this.name;
    };
    this.getMessage = function() {
        return this.message;
    };
}).call(MyObject.prototype);
```

In the three previous examples, the inherited prototype can be shared by all objects and the method definitions need not occur at every object creation. See Details of the Object Model for more.

https://developer.mozilla.org/en/docs/Web/JavaScript/Closures

JavaScript Closure - examples

August 16, 2013

Earlier, I talked about the basics of JavaScript Closure. In this post, lets continue to explore Closure with the help of some practical examples.

Before we begin, just to recap,

Closure encloses function and the set of variables that were in scope of the function when it was declared. The variables inside the closure kept alive as long as the function alive.

With that in mind, let’s make some closures.

1. Maintain State between function calls

Let’s say you have function `add()` and you’d like it to add all the values passed to it in several calls and return the sum. For example,

```
add(5);  // returns 5
add(20);  // returns 25 (5+20)
add(3);  // returns 28 (25 + 3)
```

Of course, you can use a global variable in order to hold the total. But keep in mind that this dude will eat you alive if you (ab)use globals.

For scenarios like this, Closure is the best candidate for maintaining state between function calls without using globals. Let’s see how.

```
// Using IIFE, to not to pollute global namespace.
(function(){

   var addFn = function addFn(){
      // local to closure and hold the value inbetween multiple calls.
      var total = 0;
      return function(val){
         total += val;
         return total;
      }

   };

   var add = addFn();
```

```
    console.log(add(5));  // 5
    console.log(add(20));  // 25
    console.log(add(3));  // 28

}());
```

Run this example on JSFiddle.

2. Partial application, a.k.a Currying

Suppose you have a function that takes several arguments and you only know values for some of the arguments in the beginning. For this scenario, you can make use of Currying technique to pre-fill the values for known arguments and supply values for the rest of the arguments later.

Here's an example, illustrating Currying using Closure.

Assume you have a `showMessage()` function that shows given message on screen with the given type and position. It takes three arguments. So, every time you want to call this function, you need to supply these three values.

```
(function(){
    function  showMessage(type,  position,  message){
        // displays a message at position and sets it type (for CSS styling)
    }

    showMessage('error',  'top',  'Not good.');
    showMessage('info',  'top',  'You better know this.');

}());
```

What if you want to make this function call, simpler? What if you create two other methods, namely, `showError()` and `showInfo()` that prefill the message type and position and supply the actual message in a later point in time? Let's Curry them.

The Curry function is taken from John Resig's post (which btw is a good read about Currying).

```
(function(){

    // Lets add Curry method to Function so that we can call it on any function we want.
    Function.prototype.curry  =  function(){
        var  fn  =  this,  args  =  Array.prototype.slice.call(arguments);
        return  function(){
```

```
        return  fn.apply(this,  args.concat(Array.prototype.slice.call(arguments)));
      };
   };

   // Core method.
   function  showMessage(type,  position,  message){
      console.log('showing ['  +  message  +  '] of type ['  +  type  +  '] at ['  +  position  +  '].'
);
   }

   // Create special versions of Core method using Currying.
   var  showError  =  showMessage.curry('error',  'top');
   var  showInfo  =  showMessage.curry('info',  'bottom');

   // Call our special methods.
   showError('Not good.');
   showInfo('You better know this.');

}());
```

Run this example of JSFiddle

3. Private methods in JavaScript?

Yes, we can emulate private methods in JavaScript using Closure. Let’e see how.

```
(function(){

   var  makeCar  =  function(){

      // private variable
      var  fuel  =  0;

      // private method
      function  burnFuel(){
         fuel-=10;
         console.log('Burned fuel [10]');
      }

      return  {
         accelerate  :  function(){
            if(fuel  >  0){
               burnFuel();
            }else{
               console.log('Out of gas. Fill now.');
            }
         },

         fillGas  :  function(gas){
            if(fuel  <=  100){
               fuel  +=  gas;
            }else{
               console.log('Reached capacity. Stop spilling.');
            }
         }
      }

   };

   var car = makeCar();
   car.accelerate();  // Out of gas. Fill now.
   car.fillGas(75);
   car.accelerate();  // Burned fuel [10]
   car.accelerate();  // Burned fuel [10]

}());
```

Run this example on JSFiddle

As you can see, the `makeCar()` function returns an object with two methods: `accelerate` and `fillGas`. These two methods has access to the private method `burnFuel` and private variable `fuel`. But the outer world can not directlty access these two.

So, with the help of closure you can simulate object oriented programming in JavaScript.

With that, I am concluding this post of Closure examples. Of course, these are not the *only* examples of Closures. There are lot many out there. Btw, If you have written closure for an interesting use case, feel free to share it in the comments section.

https://veerasundar.com/blog/2013/08/javascript-closure-examples/