



# JAVASCRIPT OOP

## Class Programming

Introduction to class  
file programming in  
JavaScript

Congratulations

# Terminology

## Namespace

A container which allows developers to bundle all functionality under a unique, application-specific name.

## Class

Defines the characteristics of the object. It is a template definition of variables and methods of an object.

## Object

An Instance of a class.

## Property

An object characteristic, such as color.

## Method

An object capability, such as walk. It is a subroutine or function associated with a class.

## Constructor

A method called at the moment of instantiation of an object. It usually has the same name as that of the class containing it.

## Inheritance

A class can inherit characteristics from another class.

## **Encapsulation**

A method of bundling the data and methods that use them together.

# Custom objects

## The class

JavaScript is a prototype-based language which contains no class statement, such as is found in C++ or Java. This is sometimes confusing for programmers accustomed to languages with a class statement. Instead, 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 Person.

```
1 function Person() { }  
2 or  
3 var Person = function(){ }
```

## The object (class instance)

To create a new instance of an object *obj* we use the statement `new obj`, assigning the result (which is of type *obj*) to a variable to access it later.

## The object (class instance)

To create a new instance of an object `obj` we use the statement `new obj`, assigning the result (which is of type `obj`) to a variable to access it later.

In the example below we define a class named `Person` and we create two instances (`person1` and `person2`).

```
1 function Person() { }  
2 var person1 = new Person();  
3 var person2 = new Person();
```

*Please also see [Object.create](#) for a new and alternative instantiation method.*

## The constructor

The constructor is called at the moment of instantiation (the moment when the object instance is created). The constructor is a method of the class. In JavaScript, the function serves as the constructor of the object therefore, there is no need to explicitly define a constructor method. Every action declared in the class gets executed at the time of instantiation.

The constructor is used to set the object's properties or to call methods to prepare the object for use. Adding class methods and their definitions occurs using a different syntax described later in this article.

In the example below, the constructor of the class `Person` logs a message when a `Person` is instantiated.

```
1 function Person() {  
2   console.log('Person instantiated');  
3 }  
4  
5 var person1 = new Person();  
6 var person2 = new Person();
```

## The property (object attribute)

Properties are variables contained in the class; every instance of the object has those properties. Properties should be set in the prototype property of the class (function) so that inheritance works correctly.

Working with properties from within the class is done using the keyword `this`, which refers to the current object. Accessing (reading or writing) a property outside of the class is done with the syntax: `InstanceName.Property`; this is the same syntax used by C++, Java, and a number of other languages. (Inside the class the syntax `this.Property` is used to get or set the property's value.)

In the example below we define the `firstName` property for the `Person` class and we define it at instantiation.

```
1 function Person(firstName) {  
2   this.firstName = firstName;  
3   console.log('Person instantiated');  
4 }  
5  
6 var person1 = new Person('Alice');  
7 var person2 = new Person('Bob');  
8  
9 // Show the firstName properties of the objects  
10 console.log('person1 is ' + person1.firstName); // logs "person1 is Alice"  
11 console.log('person2 is ' + person2.firstName); // logs "person2 is Bob"
```

## The methods

Methods follow the same logic as properties; the difference is that they are functions and they are defined as functions. Calling a method is similar to accessing a property, but you add `()` at the end of the method name, possibly with arguments. To define a method, assign a function to a named property of the class's `prototype` property; the name that the function is assigned to is the name that the method is called by on the object.

In the example below we define and use the method `sayHello()` for the `Person` class.

```
1 function Person(firstName) {  
2   this.firstName = firstName;  
3 }  
4  
5 Person.prototype.sayHello = function() {  
6   console.log("Hello, I'm " + this.firstName);  
7 };  
8  
9 var person1 = new Person("Alice");  
10 var person2 = new Person("Bob");  
11  
12 // call the Person sayHello method.  
13 person1.sayHello(); // logs "Hello, I'm Alice"  
14 person2.sayHello(); // logs "Hello, I'm Bob"
```

## Summary

---

- **private variables** are declared with the 'var' keyword inside the object, and can only be accessed by private functions and privileged methods.
- **private functions** are declared inline inside the object's constructor (or alternatively may be defined via `var functionName=function() { ... }`) and may only be called by privileged methods (including the object's constructor).
- **privileged methods** are declared with `this.methodName=function() { ... }` and may invoked by code external to the object.
- **public properties** are declared with `this.variableName` and may be read/written from outside the object.
- **public methods** are defined by `Classname.prototype.methodName = function() { ... }` and may be called from outside the object.
- **prototype properties** are defined by `Classname.prototype.propertyName = someValue`
- **static properties** are defined by `Classname.propertyName = someValue`




## Private data members

Constructors can also have private data members within its defined scope. These include the constructor's argument parameters and the variables that are locally declared within the constructor.

```
1 // Person class with a private pass-in parameter firstName
2 function Person(firstName) {
3     this.firstName = firstName;
4
5     // this is a private variable privateVariable only
6     // accessible within the Person's scope.
7     var privateVariable = 777;
8 }
```

Why private data members? These allow classes to have its own uniqueness and provide its own manipulation of particular data sets by using private functions. For example, the class `Person` can take in a parameter `firstName` and `number`.

```
1 function Person(firstName, number) {
2     this.firstName = firstName;
3     var privateVariable = 777;
4
5     function lucky() {
6         if (number == privateVariable) {
7             console.log("You are a lucky person");
8         }
9     }
10 }
```

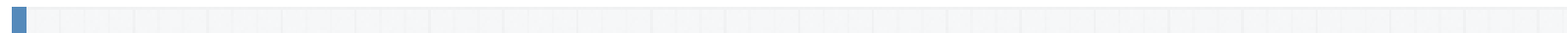


## Inheritance

Inheritance is a way to create a class as a specialized version of one or more classes (*JavaScript only supports single inheritance*). The specialized class is commonly called the *child*, and the other class is commonly called the parent. In JavaScript you do this by assigning an instance of the parent class to the child class, and then specializing it. In modern browsers you can also use `Object.create` to implement inheritance.

*JavaScript does not detect the child class `prototype.constructor` (see [Object.prototype](#)), so we must state that manually.*

In the example below, we define the class `Student` as a child class of `Person`. Then we redefine the `sayHello()` method and add the `sayGoodBye()` method.



# Person Class

```
1 // Define the Person constructor
2 function Person(firstName) {
3   this.firstName = firstName;
4 }
5
6 // Add a couple of methods to Person.prototype
7 Person.prototype.walk = function(){
8   console.log("I am walking!");
9 };
10 Person.prototype.sayHello = function(){
11   console.log("Hello, I'm " + this.firstName);
12 };
13
```

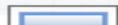
# Student Class

```
// Define the Student constructor
function Student(firstName, subject) {
  // Call the parent constructor, making sure (using Function#call)
  // that "this" is set correctly during the call
  Person.call(this, firstName);

  // Initialize our Student-specific properties
  this.subject = subject;
};
```

# Inheritance


```
3
4 // Create a Student.prototype object that inherits from Person.prototype.
5 // Note: A common error here is to use "new Person()" to create the
6 // Student.prototype. That's incorrect for several reasons, not least
7 // that we don't have anything to give Person for the "firstName"
8 // argument. The correct place to call Person is above, where we call
9 // it from Student.
10 Student.prototype = Object.create(Person.prototype); // See note below
11
12 // Set the "constructor" property to refer to Student
13 Student.prototype.constructor = Student;
14
15 // Replace the "sayHello" method
16 Student.prototype.sayHello = function(){
17     console.log("Hello, I'm " + this.firstName + ". I'm studying "
18         + this.subject + ".");
19 };
20
21 // Add a "sayGoodBye" method
22 Student.prototype.sayGoodBye = function(){
23     console.log("Goodbye!");
24 };
25
26 // Example usage:
27 var student1 = new Student("Janet", "Applied Physics");
28 student1.sayHello(); // "Hello, I'm Janet. I'm studying Applied Physics."
29 student1.walk(); // "I am walking!"
30 student1.sayGoodBye(); // "Goodbye!"
31
32 // Check that instanceof works correctly
33 console.log(student1 instanceof Person); // true
34 console.log(student1 instanceof Student); // true
```



```
1 <script>
2 function exampleClass(){
3     this.property1 = 5;
4     this.property2 = "World";
5     this.method1 = function method1(arg1){
6         return arg1+" "+this.property2;
7     }
8 }
9 var instance1 = new exampleClass();
10 var instance2 = new exampleClass();
11 var result = instance1.method1("Hello");
12 alert( result );
13 </script>
```

# Separate your class with .js file as Modular design pattern

```
1 function exampleClass(){
2     this.property1 = 5;
3     this.property2 = "World";
4     this.method1 = function method1(arg1){
5         return arg1+" "+this.property2;
6     }
7 }
```



# Your access code of Class

```
1 <script>
2   var instance1 = new exampleClass();
3   var instance2 = new exampleClass();
4   var result = instance1.method1("Hello");
5   instance1.property1 = 10;
6   alert( instance1.property1 );
7   alert( instance2.property1 );
8 </script>
```



# Real World Example


JavaScript Class OOP Tutorial Intro to Object Oriented Programming

test.html x exampleClass.js x

No syntax errors.

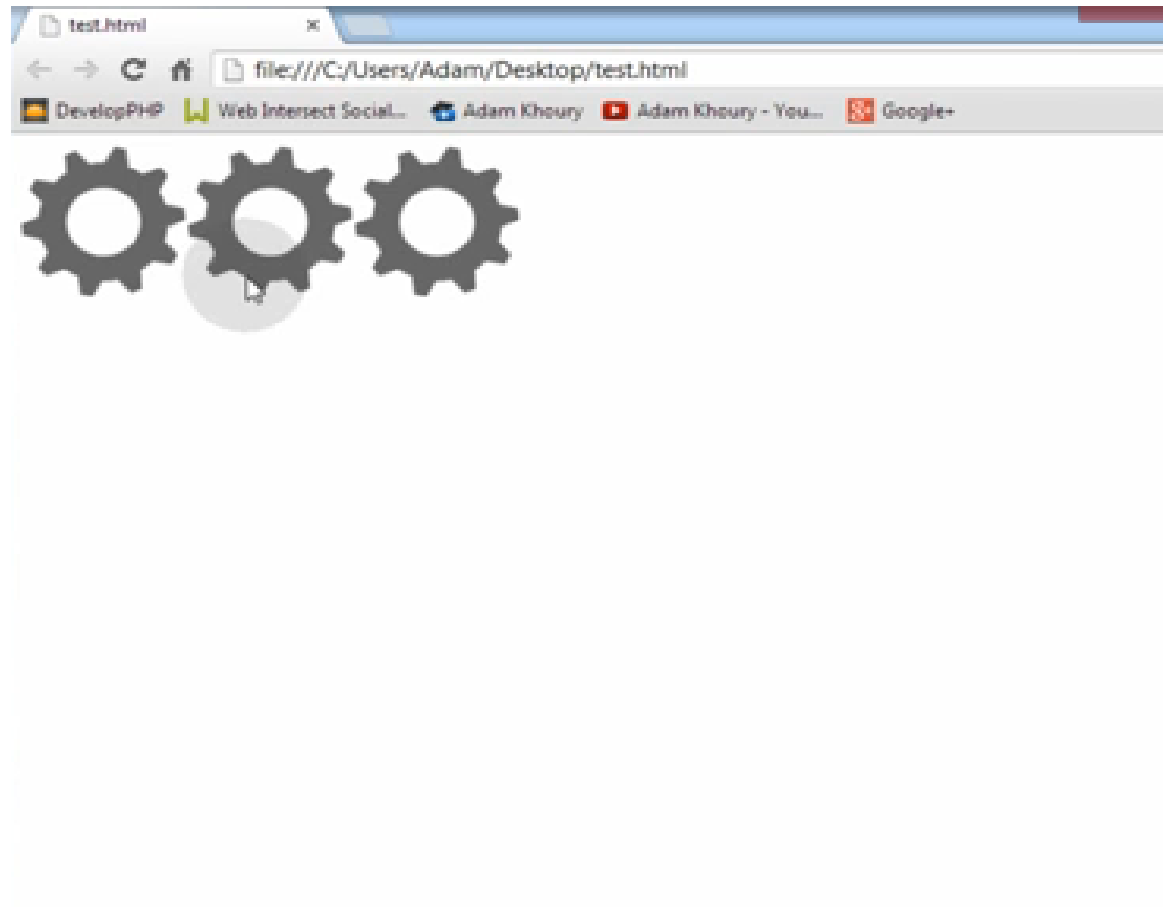
Next: prototype Property Tutorial >

Adam Khoury  
64,618 subscribers

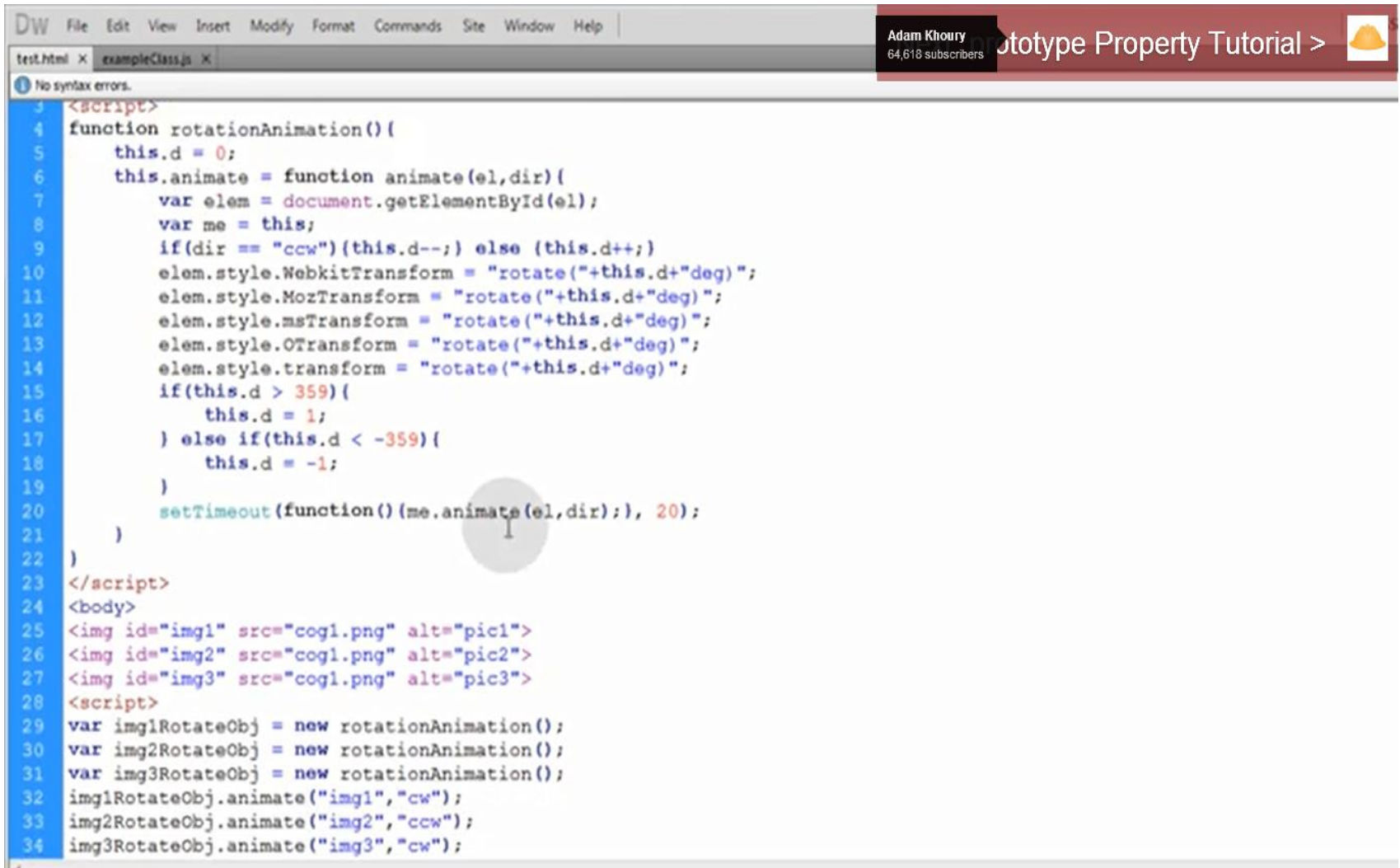


```
1 <!DOCTYPE html>
2 <html>
3 <script>
4 var d = 0;
5 function animate(el,dir){
6     var elem = document.getElementById(el);
7     if(dir == "ccw"){d--;} else {d++;}
8     elem.style.WebkitTransform = "rotate("+d+"deg)";
9     elem.style.MozTransform = "rotate("+d+"deg)";
10    elem.style.msTransform = "rotate("+d+"deg)";
11    elem.style.OTransform = "rotate("+d+"deg)";
12    elem.style.transform = "rotate("+d+"deg)";
13    if(d > 359){
14        d = 1;
15    } else if(d < -359){
16        d = -1;
17    }
18    setTimeout(function(){animate(el,dir);}, 20);
19 }
20 </script>
21 <body>
22 
23 
24 
25 <script>
26 animate("img1", "cw");
27 animate("img2", "ccw");
28 animate("img3", "cw");
29 </script>
30 </body>
31 </html>
```

# Output



# OO java script Code



The screenshot shows a web browser window with a dark theme. The address bar shows 'test.html' and 'exampleClass.js'. The page content is a JavaScript file with a function 'rotationAnimation()' and its usage. The function is designed to rotate an element in a specified direction (clockwise or counter-clockwise) by a certain angle. It uses a 'this.d' property to track the current angle and a 'setTimeout' to animate the rotation over time. The usage part creates three rotation objects and animates three images ('img1', 'img2', 'img3') with different rotation directions and speeds.

```
3 <script>
4 function rotationAnimation(){
5     this.d = 0;
6     this.animate = function animate(el,dir){
7         var elem = document.getElementById(el);
8         var me = this;
9         if(dir == "ccw"){this.d--;} else {this.d++;}
10        elem.style.WebkitTransform = "rotate("+this.d+"deg)";
11        elem.style.MozTransform = "rotate("+this.d+"deg)";
12        elem.style.msTransform = "rotate("+this.d+"deg)";
13        elem.style.OTransform = "rotate("+this.d+"deg)";
14        elem.style.transform = "rotate("+this.d+"deg)";
15        if(this.d > 359){
16            this.d = 1;
17        } else if(this.d < -359){
18            this.d = -1;
19        }
20        setTimeout(function(){me.animate(el,dir);}, 20);
21    }
22 }
23 </script>
24 <body>
25 
26 
27 
28 <script>
29 var img1RotateObj = new rotationAnimation();
30 var img2RotateObj = new rotationAnimation();
31 var img3RotateObj = new rotationAnimation();
32 img1RotateObj.animate("img1","cw");
33 img2RotateObj.animate("img2","ccw");
34 img3RotateObj.animate("img3","cw");
```

# JAVASCRIPT OOP

## Pros:

- Makes code modular and easier to share.
- Avoids value collisions in specific cases.
- Makes specialized functions reusable.

## Cons:

- It adds more lines to code so if you do not require it, there is no good reason to use it







# JAVASCRIPT OOP

## Class Programming

### prototype property

Use to Add Object Properties  
and Methods To Your Classes

# Create Player javascript Class

```
<script>
// Establish the Player class-based object
function Player() {
    this.name;
    this.hitpoints = 100;
    this.attack = function attack(opponent) {
        opponent.hitpoints -= 10;
        alert(this.name+" just hit "+opponent.name);
    }
}

// Create two new separate Player instances
var p1 = new Player();
var p2 = new Player();
// Name the players
p1.name = "Conan";
p2.name = "Hercules";
// Make player1 attack player2
p1.attack(p2);
alert(p2.name+" has "+p2.hitpoints+" hit points left");
</script>
```

# Add method using Prototype

```
<script>
// Establish the Player class-based object
function Player(){
    this.name;
    this.hitpoints = 100;
    this.attack = function attack(opponent){
        opponent.hitpoints -= 10;
        alert(this.name+" just hit "+opponent.name);
    }
}
// Create two new separate Player instances
var p1 = new Player();
var p2 = new Player();
// Name the players
p1.name = "Conan";
p2.name = "Hercules";
// Make player1 attack player2
p1.attack(p2);
alert(p2.name+" has "+p2.hitpoints+" hit points left");
// Add a heal method to the Player object using the prototype property
Player.prototype.heal = function heal(targetPlayer){
    targetPlayer.hitpoints += 5;
};
// Make player1 heal player2
p1.heal(p2);
alert(p2.name+" has "+p2.hitpoints+" hit points left");
</script>
```

# Add Property using Prototype

```
// Create two new separate Player instances
var p1 = new Player();
var p2 = new Player();
// Name the players
p1.name = "Conan";
p2.name = "Hercules";
// Make player1 attack player2
p1.attack(p2);
alert(p2.name+" has "+p2.hitpoints+" hit points left");
// Add a heal method to the Player object using the prototype property
Player.prototype.heal = function heal(targetPlayer){
    targetPlayer.hitpoints += 5;
};
// Make player1 heal player2
p1.heal(p2);
alert(p2.name+" has "+p2.hitpoints+" hit points left");
alert(p1.name+" has "+p1.hitpoints+" hit points left");
// Add an "energy" property to the Player object using the prototype property
Player.prototype.energy = 200;
alert(p1.energy);
</script>
```





# JAVASCRIPT INHERITANCE TUTORIAL

**Object Parent / Child Relationships**

**aka:**

**Object Class / Subclass Relationships**

# Create parent and Child Class

```
1 <script>
2 // Establish a parent class
3 function Parentclass(){
4     this.parent_property1 = "Hola";
5     this.parentmethod1 = function parentmethod1(arg1){
6         return arg1+" Parent method 1 return data ...";
7     }
8 }
9 // Establish a child class
10 function Childclass(){
11     this.child_property1 = "Adios";
12     this.childmethod1 = function childmethod1(arg1){
13         return arg1+" Child method 1 return data ...";
14     }
15 }
16 // Make the Childclass inherit all of the Parent class chara
17 // by using the prototype property explained in depth here:
18 </script>
```

# Create instance of class

```
1 <script>
2 // Establish a parent class
3 function Parentclass(){
4     this.parent_property1 = "Hola";
5     this.parentmethod1 = function parentmethod1(arg1){
6         return arg1+" Parent method 1 return data ...";
7     }
8 }
9 // Establish a child class
10 function Childclass(){
11     this.child_property1 = "Adios";
12     this.childmethod1 = function childmethod1(arg1){
13         return arg1+" Child method 1 return data ...";
14     }
15 }
16 // Make the Childclass inherit all of the Parent class charac
17 // by using the prototype property, explained in depth here:
18 Childclass.prototype = new Parentclass();
19 // Create a new instance of Childclass
20 var instance1 = new Childclass();
21 // Check to see if instance1 is an instance of both objects
22 alert(instance1 instanceof Parentclass);
23 alert(instance1 instanceof Childclass);
24 </script>
```

# Comment your Alert Code

- [illegible]

# Calling both parent and child class method

```
1 <script>
2 // Establish a parent class
3 function Parentclass(){
4     this.parent_property1 = "Hola";
5     this.parentmethod1 = function parentmethod1(arg1){
6         return arg1+" Parent method 1 return data ...";
7     }
8 }
9 // Establish a child class
10 function Childclass(){
11     this.child_property1 = "Adios";
12     this.childmethod1 = function childmethod1(arg1){
13         return arg1+" Child method 1 return data ...";
14     }
15 }
16 // Make the Childclass inherit all of the Parent class charac
17 // by using the prototype property, explained in depth here:
18 Childclass.prototype = new Parentclass();
19 // Create a new instance of Childclass
20 var instance1 = new Childclass();
21 // Check to see if instance1 is an instance of both objects
22 //alert(instance1 instanceof Parentclass);
23 //alert(instance1 instanceof Childclass);
24 // Access the instance methods and properties
25 alert( instance1.parentmethod1("RESULT: ") );
26 alert( instance1.childmethod1("RESULT: ") );
27 </script>
```



```
// Make the Childclass inherit all of the Parent class chars
// by using the prototype property, explained in depth here:
Childclass.prototype = new Parentclass();
// Create a new instance of Childclass
var instance1 = new Childclass();
// Check to see if instance1 is an instance of both objects
//alert(instance1 instanceof Parentclass);
//alert(instance1 instanceof Childclass);
// Access the instance methods and properties
alert( instance1.parentmethod1("RESULT: ") );
alert( instance1.childmethod1("RESULT: ") );
alert(instance1.parent_property1);
alert(instance1.child_property1);
</script>
```

# Override parentmethod1 in the Childclass

```
16 // Make the Childclass inherit all of the Parent class characterist
17 // by using the prototype property, explained in depth here: youtu
18 Childclass.prototype = new Parentclass();
19 // Create a new instance of Childclass
20 var instancel = new Childclass();
21 // Check to see if instancel is an instance of both objects
22 //alert(instancel instanceof Parentclass);
23 //alert(instancel instanceof Childclass);
24 // Access the instance methods and properties
25 alert( instancel.parentmethod1("RESULT: ") );
26 alert( instancel.childmethod1("RESULT: ") );
27 alert(instancel.parent_property1);
28 alert(instancel.child_property1);
29 // Override parentmethod1 in the Childclass
30 Childclass.prototype.parentmethod1 = function parentmethod1(arg1){
31     return arg1+" I have overridden Parent method 1";
32 }
33 alert(instancel.parentmethod1("RESULT: "));
34 </script>
```

Public and private



# Create Class

```
1. // Constructor
2. function Kid (name) {
3.     // Empty, for now
4. }
```

Now you can create as many kid objects you want, in this fashion:

```
1. var kenny = new Kid("Kenny");
```

# Adding private Property

## Adding A Private Property

As we learned in the [JavaScript scope and closures](#) article, by declaring a variable within a function, it is only available from within there. So, if we want a private property of the Kid object, we do it like this:

```
1. // Constructor
2. function Kid (name) {
3.     // Private
4.     var idol = "Paris Hilton";
5. }
```

# Add Privileged Method

## Adding A Privileged Method

A privileged method is a method having access to private properties, but at the same time publicly exposing itself (in JavaScript, also due to JavaScript scope and closures). You can delete or replace a privileged method, but you cannot alter its contents. I.e. this privileged method returns the value of a private property:

```
01. // Constructor
02. function Kid (name) {
03.     // Private
04.     var idol = "Paris Hilton";
05.
06.     // Privileged
07.     this.getIdol = function () {
08.         return idol;
09.     };
10. }
```

```
var Person = function(options) {  
    //private properties  
    var name = options.name  
    var birthYear = options.birthYear;  
    //private method  
    var calculateAge = function() {  
        var today = new Date();  
        return today.getFullYear() - birthYear;  
    }  
    //Privileged method  
    this.getAge = function() {  
        return calculateAge(); //calling private method  
    }  
}  
  
//new Person instance  
var p = new Person({name:'Peter', birthYear:1983});  
  
console.debug(p.getAge()); // the age  
console.debug(p.name); // undefined  
console.debug(p.birthYear); // undefined
```

# Adding Public Property and Public Method

## Adding A Public Property And A Public Method

Now that we have private and privileged members out of the way, let's look at the very basic nature of public properties and methods:

```
1. // Constructor
2. function Kid (name) {
3.     // Public
4.     this.name = name;
5.     this.getName = function () {
6.         return this.name;
7.     };
8. }
```

# The recommended way for public methods is using the prototype approach

```
1. // Constructor
2. function Kid (name) {
3.     // Public
4.     this.name = name;
5. }
6. Kid.prototype.getName = function () {
7.     return this.name;
8. };
```

# Adding A Static Property

## Adding A Static Property

A static member is shared by all instances of the class as well as the class itself (i.e. the Kid object), but it is only stored in one place. This means that its value is *not* inherited down to the object's instances:

```
1. // Constructor
2. function Kid (name) {
3.     // Constructor code
4. }
5.
6. // Static property
7. Kid.town = "South Park";
```


# How to make a singleton in javascript

- Ensure a class only has one instance, and provide a global point of access to it.
- **GoF** *Design Patterns*



## Singleton with a cached static property

```
function User() {  
    // do we have an existing instance?  
    if (typeof User.instance === 'object') {  
        return User.instance;  
    }  
  
    // proceed as normal  
    this.firstName = 'John';  
    this.lastName = 'Doe';  
  
    // cache  
    User.instance = this;  
  
    // implicit return  
    // return this;  
}
```



# Namespace

## Namespaces

In most programming languages we know the concept of **namespaces (or packages)**. Namespaces allow us to group code and help us to avoid name-collisions.

In c# for example you have this declaration

```
1 namespace MyNameSpace
2 {
3     public class MyClass
4     {
5     }
6 }
```

If you want to use *MyClass*, you need to do explicitly say in which namespace it lives:

```
1 MyNameSpace.MyClass obj;
```

# anti-pattern way

Let's first look at an **anti-pattern**, which is declaring all your functions and variables globally:

```
1  function calculateVat(prod) {  
2      return prod.price * 1.21;  
3  }  
4  
5  var product = function (price) {  
6      this.price = price;  
7      this.getPrice = function(){  
8          return this.price;  
9      };  
10     };  
11  
12  function doCalculations() {  
13      var p = new product(100);  
14      alert(calculateVat(p.getPrice()));  
15  }
```

# How to create a namespace in JavaScript

To solve this problem you can create a **single global object** for your app and make all functions and variables properties of that global object:

```
1  var MYAPPLICATION = {  
2      calculateVat: function (base) {  
3          return base * 1.21;  
4      },  
5      product: function (price) {  
6          this.price = price;  
7          this.getPrice = function(){  
8              return this.price;  
9          };  
10     },  
11     doCalculations: function () {  
12         var p = new MYAPPLICATION.product(100);  
13         alert(this.calculateVat(p.getPrice()));  
14     }  
15 }
```

Now we only have one global variable (MYAPPLICATION). Although this is not really a **namespace**, it can be used as one, since you have to go through the MYAPPLICATION object in order to get to your application code:

```
1  var p = new MYAPPLICATION.product(150);  
2  alert(p.getPrice());
```

```
4 | alert(p.getPrice());
```

## Nested namespaces

In most languages you can declare a namespace inside a namespace. This allows for even better **modularization**. We can just apply the pattern again and define an object inside the outer object:

```
1 | var MYAPPLICATION = {
2 |     MODEL: {
3 |         product: function (price) {
4 |             this.price = price;
5 |             this.getPrice = function(){
6 |                 return this.price;
7 |             };
8 |         }
9 |     },
10 |     LOGIC: {
11 |         calculateVat: function (base) {
12 |             return base * 1.21;
13 |         },
14 |         doCalculations: function () {
15 |             var p = new MYAPPLICATION.MODEL.product(100);
16 |             alert(this.calculateVat(p.getPrice()));
17 |         }
18 |     }
19 | }
```

This pattern is fairly simple and is a good way to avoid naming collisions with other libraries (or your own code for that matter).

# Creating a multi-purpose Namespace function

What we'd like to do is simply call a function that creates a namespace safely and then lets us define function and variables in that namespace. Here's where JavaScript's dynamic nature really shines. Let's start with an example of what we want to achieve:

```
1  var MAYAPPLICATION = MYAPPLICATION || {};  
2  
3  var ns = MYAPPLICATION.createNS("MYAPPLICATION.MODEL.PRODUCTS");  
4  
5  ns.product = function(price){  
6      this.price = price;  
7      this.getPrice = function(){  
8          return this.price;  
9      }  
10 };
```

We still need to check our main namespace (you have to start somewhere), but it will allow us to create a hierarchy of namespaces with a single line and have it all figured out.

# Safely creating namespaces

## Safely creating namespaces

Since we still have one global object, there's still a possibility that we are overwriting another global object with the same name. Therefore, we need to build in some safety:

```
1 // not safe, if there's another object with this name we will overwrite it
2 var MYAPPLICATION = {};
3
4 // We need to do a check before we create the namespace
5 if (typeof MYAPPLICATION === "undefined") {
6     var MYAPPLICATION = {};
7 }
8
9 // or a shorter version
10 var MAYAPPLICATION = MYAPPLICATION || {};
```





**Dynamic Namespacing** We could also call this section *namespace injection*. The namespace is represented by a proxy which is directly referenced *inside* the function wrapper – which means we no longer have to bundle up a return value to assign to the namespace.

`.apply()`

```
1  //library code
2  var protoQueryMooJo = function() {
3      //everything
4  }
5
6  //user code
7  var thirdParty = {};
8  protoQueryMooJo.apply(thirdParty);
```

## 5. Use *this* as a Namespace Proxy

A recent posting by [James Edwards](#) piqued my interest. [My Favorite JavaScript Design Pattern](#) was apparently misunderstood by many commentators, who thought he might as well resort to the module pattern. The article peddles multiple techniques (which probably contributed to readers' confusion) but at its heart is a little bit of genius which I've revamped and presented a namespacing tool.

The beauty of the pattern is that it simply uses the language as designed – nothing more, nothing less, no tricks, no sugar. Moreover because the namespace is injected via the *this* keyword (which is static within a given execution context) it cannot be accidentally modified.

```
1  var myApp = {};  
2  (function() {  
3      var id = 0;  
4  
5      this.next = function() {  
6          return id++;  
7      };  
8  
9      this.reset = function() {  
10         id = 0;  
11     }  
12 }).apply(myApp);  
13  
14 window.console && console.log(  
15     myApp.next(),  
16     myApp.next(),  
17     myApp.reset(),  
18     myApp.next()  
19 ); //0, 1, undefined, 0
```

# Regular expression

## RegExp Object

A **regular expression** is an object that describes a pattern of characters.

Regular expressions are used to perform pattern-matching and "search-and-replace" functions on text.

## Syntax

```
var patt=new RegExp(pattern,modifiers);
```

or more simply:

```
var patt=/pattern/modifiers;
```

- pattern specifies the pattern of an expression
- modifiers specify if a search should be global, case-sensitive, etc.

# Modifiers

Modifiers are used to perform case-insensitive and global searches:

Modifier	Description
<u>i</u>	Perform case-insensitive matching
<u>g</u>	Perform a global match (find all matches rather than stopping after the first match)
<u>m</u>	Perform multiline matching

# Brackets

Brackets are used to find a range of characters:

Expression	Description
<u>[abc]</u>	Find any character between the brackets
<u>[^abc]</u>	Find any character not between the brackets
<u>[0-9]</u>	Find any digit between the brackets
<u>[^0-9]</u>	Find any digit not between the brackets
<u>(x y)</u>	Find any of the alternatives specified

# Metacharacters

Metacharacters are characters with a special meaning:

Metacharacter	Description
<code>.</code>	Find a single character, except newline or line terminator
<code>\w</code>	Find a word character
<code>\W</code>	Find a non-word character
<code>\d</code>	Find a digit
<code>\D</code>	Find a non-digit character
<code>\s</code>	Find a whitespace character
<code>\S</code>	Find a non-whitespace character
<code>\b</code>	Find a match at the beginning/end of a word
<code>\B</code>	Find a match not at the beginning/end of a word
<code>\0</code>	Find a NUL character
<code>\n</code>	Find a new line character

Thanks