

JavaScript "Classes"

1

STEPHEN SCHAUB

Classes

2

- JavaScript originally had no class mechanism
- Even now, when you can define classes with the class keyword, there is still no concept of a class behind the scenes
- Objects are defined using
 - Object literals or
 - Constructor functions
- Inheritance is supported via a Prototype mechanism

Object Literals with Methods

3

```
var calculator = { // an object with 3 properties
  operand1: 1,
  operand2: 1,
  compute: function() {
    this.result = this.operand1 + this.operand2;
  }
};
calculator.compute(); // What is 1+1?
print(calculator.result); // Display the result
```

The **this** Keyword

4

- **this** refers to the object on which the function was invoked
- If the function was invoked without an object, **this** refers to the global object
- The global object contains top-level variables and functions

The **this** Keyword

5

- Consider:
 - `function setName(newName) { this.name = newName; }`
- `setName` can be invoked on an object:
 - `var person = { name: "", age: 15, setName: setName };
person.setName("Johnny"); // sets person.name to "Johnny"`
- `setName` can be invoked without an object:
 - `setName("Johnny"); // defines a new global variable name
alert(name);`
- Having **this** bound to global object was not a good design decision
 - Strict mode prevents this behavior

Constructor Functions

6

- A constructor function is designed to initialize an object with properties
 - Invoke with **new** operator
 - Accesses new object using **this**

- **Example:**

```
function Rectangle(w, h) {  
  this.width = w;  
  this.height = h;  
}
```

```
var rect = new Rectangle(2, 4); // rect = { width: 2, height: 4 }
```

Constructor Function Caveat

7

- Calling a constructor function without using **new** is a big mistake
 - `rect = Rectangle(2, 4); // tromps on globals`
- Inside the constructor function, references to **this** cause variables/methods to be added to the global object
 - Or worse, existing global variables/functions are replaced
- In strict mode, calling constructor function without using *new* results in runtime error

Adding Methods to Objects

8

- We've already seen how methods can be defined in an object literal
- A constructor function can also be used to define methods for its objects:

```
function Rectangle(w, h) {  
  this.width = w;  
  this.height = h;  
  this.area = function() { return this.width * this.height; }  
}
```

```
var paper = new Rectangle(8.5, 11);  
var a = paper.area();
```


Defining Static Members

- Functions are objects, and can have properties

```
function foo() { ... }  
foo.x = 3; // create property "x"
```

- Although not useful for normal functions, this capability is helpful for constructor functions

```
function Circle(r) { this.radius = r; } // Define a "class" Circle
```

```
Circle.PI = 3.14159; // Create a "static" property
```

```
Circle.max = function(a, b) { return (a.r > b.r) ? a : b; } // Create a "static" method
```

The **prototype** Property

10

- Constructor functions have a property named **prototype**
- **prototype** specifies an object serves as a fallback source of properties for objects created by the constructor
- Add properties to a constructor function's **prototype** to define methods shared by all objects created by the constructor

```
function Circle(r) { this.radius = r; }  
Circle.prototype.area = function() {  
  return Math.Pow(this.r, 2) * Math.PI;  
}
```

```
c = new Circle(100);  
a = c.area();
```

The **prototype** Property, cont.

11

- When the object is created with **new**, it is linked to its prototype object
 - Prototype properties are **not** copied into the object
- Defining methods using the prototype approach is more efficient than defining them inside a constructor function
- Can also be used to add methods to existing "classes"
 - JavaScript libraries frequently use this capability to augment the functionality of String and Array objects

Objects and Lambda Notation

12

- Beware: lambda expressions are not appropriate for defining object methods
 - For more info, see https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Inheritance

13

- Use the **prototype** property to achieve inheritance
 - See JavaScript: The Definitive Guide for details

Summary: Defining Classes

14

- In JavaScript, constructor functions serve to define classes
 - Define instance variables using **this** inside the function
 - Assign static variables and methods as properties of the constructor function
 - Assign instance methods as properties of constructor function's **prototype** property
 - Instance methods must use **this** to access instance variables
 - ✦ **this** is not optional, as in C++ / Java

A Complete Example

15

```
function Point(x, y) {  
  this.x = x;           // create instance variables  
  this.y = y;  
  
  Point.numPoints++;  
}  
  
Point.numPoints = 0; // create "static" member  
  
Point.prototype.toString = function() {  
  return "(" + this.x + ", " + this.y + ")";  
}
```

Point Example, cont.

16

```
var pt = new Point(10, 20);
```

```
console.log( Point.numPoints ); // 1
```

```
var str = pt.toString(); // (10, 20)
```


Further Reading

17

- https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object-oriented_JS

JavaScript Modules

18

Modules

19

- A module is a named collection of variables and functions
- Contains both public variables/functions and private members
- Wraps all members in a private namespace
- Critical concept for enterprise applications
 - Avoids the danger of working in global namespace

Objects are Almost Modules

20

- ```
counter = {
 count: 0,
 increment: function() {
 return count++;
 },
 reset: function() {
 count = 0;
 }
}; // This example is broken ... can you spot the problem? ☺
```
- Gets members out of the global namespace
- Unfortunately, all members are public
  - No way to define private variables / methods

# Module Example

21

```
/* define counter "module" */
var counter = (function(){
 var count = 0;
 function doIncrement() {
 return count++;
 }
 function doReset() {
 count = 0;
 }
 return {
 increment: doIncrement,
 reset: doReset
 };
})();

/* use the module */
counter.increment ();
counter.reset ();
```

## How it works:

- An anonymous function defines the module
- Local variables and functions are private
- Module functions are exposed via an object returned from the anonymous function

# Topics

22

- Best Practices

# JavaScript Best Practices

23

- Define all variables with var
  - Makes it clear which variables are local and which are global
- Declare all local variables at the top of their function
  - Reduces confusion about variable scope
- Avoid global variables
  - Package state into objects or modules

# JavaScript Best Practices

24

- Prefer `===` and `!==` over their evil twins `==` and `!=`
  - Safer, less surprising behavior
- Before using `+` to add, ensure both operands are numbers
  - Use `parseInt()`, `parseFloat()`, or unary plus to force operands to number



# JavaScript Best Practices

25

- **Terminate statements with semicolons**
  - Reduces likelihood of errors
- **Prefer opening curly braces on the same line as the construct that starts them**
  - `if ( ... ) {`
  - Helps avoid some subtle bugs related to semicolon termination
- **Avoid the with statement**
  - Difficult to optimize
  - Function definitions and variable initializations inside a with statement lead to surprising behavior
  - Removed from strict form of language in ES 5
- **Treat eval function as toxic**
  - Can tromp on global variables

# Closing Thoughts

26

- JavaScript has good parts and bad parts
- JavaScript code quality tool: JSLint
  - Identifies poor usage patterns

# References

- Flanagan, David. JavaScript: The Definitive Guide. Highly recommended JavaScript reference.
- <https://developer.mozilla.org/en/JavaScript>  
Helpful JavaScript reference from Mozilla
- Crockford, Douglas. JavaScript: The Good Parts.
  - <http://yuiblog.com/crockford/>
- <http://www.hunlock.com/>  
Helpful Javascript Language Tutorials
- <http://addyosmani.com/resources/essentialjsdesignpatterns/book/>  
JavaScript Design Patterns (Module, Singleton, etc.)