HTML5, CSS3 JS

Chapter 4:
Object Oriented JavaScript

Chapter Objectives

In this chapter, we will cover JavaScript technologies:

- Function techniques
- Prototypes
- Factories
- Closures
- Iterators and Generators

Chapter Concepts

- **Function Techniques**
- JavaScript Prototype
- Creating a JavaScript Factory
- JavaScript Scope
- Closures
- Iterators and Generators
- Chapter Summary

Understanding 'this'

- In JavaScript, the thing called **`this`** is the object that "owns" the executing JavaScript code
- When used in a function:
  - **`this`** is the object that "owns" the function
- When used in an object:
  - **`this`** is the object itself
- When used in an object constructor:
  - **`this`** does not have a value
  - It is a placeholder for the newly created object
- When an object constructor is used to create an object:
  - **`this`** is the newly created object

Using `call()`, `bind()`, and `apply()`

- JavaScript functions have methods
  - Including `call()`, `bind()`, and `apply()`
- The `bind()` method lets us set what object will be bound to **this**
  - When another function is called
- The following code has a problem
  - The **this** in the click handler is bound to the button element

```
var user = {
    data :[
        {name:"Gamma", age:37},
        {name:"Helm", age:43},
        {name:"Johnson", age:42},
        {name:"Vlissides", age:54}],
    clickHandler:function (event) {
        var randomNum =
          ((Math.random () * 4 | 0) + 1) - 1; // random number 0 - 3
        $("input").val(this.data[randomNum].name
            + " " + this.data[randomNum].age);
    }
}
  // Assign an eventHandler to the button's click event
$ ("button").click (user.clickHandler);
```

Using `bind()`

- The preceding slide demonstrated a common problem in JavaScript
- The `bind()` method solves this problem
  - The `user` object is now **this** in the `clickHandler` function

```
var user = {
    data :[
        {name:"Gamma", age:37},
        {name:"Helm", age:43},
        {name:"Johnsom", age:42},
        {name:"Vlissides", age:54}],
    clickHandler:function (event) {
        var randomNum =
          ((Math.random () * 4 | 0) + 1) - 1; // random number 0 - 3
        $("input").val(this.data[randomNum].name
          + " " + this.data[randomNum].age);
    }
}
  // Assign an eventHandler to the button's click event
$ ("button").click (user.clickHandler.bind (user));
```

Using `call()` and `apply()`

- The `call()` and `apply()` functions allow us to borrow functions
  - And set **this** in the function call
- Additionally, the apply function allows execution of a function with an array of parameters
  - Each parameter is passed to the function individually

```
var employeeData = {
    id: 42,
    fullName: "Not Set",
    setUserName: function (firstName, lastName)  {
    // this refers to the fullName property in this object
    this.fullName = firstName + " " + lastName;
    }}
```

```
function getUserInput (firstName, lastName, callback, callbackObj) {
  callback.apply (callbackObj, [firstName, lastName]);
}
```

```
getUserInput ("Martin", "Fowler", employeeData.setUserName, employeeData);

// the fullName property on the clientData was correctly set
console.log (employeeData.fullName);
```

Immediately Invoked Functions

- JavaScript supports Immediately Invoked Function Expressions (IIFE)
  - Pronounced “iffy”
- A function definition is the “normal” way of creating a named function
- You can assign a function expression to a variable or property
- If we want to evaluate the function right away (like immediately):
  - Just add the parentheses at the end
- Can pass arguments too

```
function normalNamedFunction() { /*...*/ }
```

```
var varFunc = function () { /* ... */ };
```

```
(function () {/* ... */})();
```

```
var foo = "bar";
(function (innerFoo) {
  console.log(innerFoo);
})(foo)
```

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Chapter Summary

Object Literal

- As you already know, an object literal is a fast way to create an object with defined values
  - *Note*: Object literals cannot be further instantiated, only use them for singletons!

```
var cell = {
        name : "LG",
        model : "Stylo",
        weight : 144.6,
        color : "silver",
        fullname : function() {
                return this.name + " " + this.model;
        }
};
```

- Values can still be changed later, but no other objects of that type can be created

```
cell.name = "Samsung";
cell.model = "Galaxy S7";
```

Function Literal (or Function Expression)

- A function literal is very similar to a constructor function, but is unnamed
  - It allows parameters and multiple instances

```
var cell = function (name, model, weight, color) {
         var name = name;    // made name private
         var model = model;  // made model private
         this.weight = weight; // these are public
         this.color = color;
         this.changeColor = function(color) {
                  this.color = color;
         }
}

var myLG = new cell("LG", "Stylo", 144.6, "silver");
myLG.changeColor("red");
```

- However, function expressions cannot be hoisted
  - They only exist when code is reached
- What is hoisting?

Hoisting

- Hoisting means that I can call a function before it is defined in the code sequence

```
// Output: "Hello!"
functionTwo();

function functionTwo() {
  console.log("Hello!");
};
```

```
// TypeError: undefined is not a function
functionOne();

var functionOne = function() {
  console.log("Hello!");
};
```

Prototypes

- Every JavaScript object has a prototype—the prototype is also an object
- All JavaScript objects inherit their properties and methods from their prototype
- Previous example of a constructor also creates a prototype
  - However, we want to take a closer look at using the prototype property

```
function cell(name, model, weight, color) {
        this.name = name;
        this.model = model;
        this.weight = weight;
        this.color = color;
}
cell.prototype.fullname = function() {
        return this.name + " " + this.model;
};
```

- This code example adds a function to all object of the type “cell”
  - More efficient since the function is only defined once

The Prototype Pattern

- Assigns a JavaScript object literal to the prototype
- Benefits:
  - Leverage JavaScript's built-in features
  - "Modularize" code into re-useable objects
  - Variables/functions taken out of global namespace
  - Functions loaded into memory once
  - Possible to "override" functions through prototyping
- Challenges:
  - "`this`" can be tricky
  - Constructor separate from prototype definition

Step 1: Define the Constructor

- A constructor can be defined using a standard JavaScript function

Constructor

```
var Calculator = function (sum) {
  this.sum = sum;
};
```

"this" required to define instance variables

Step 2: Define the Prototype

- Define an object prototype using an object literal

```
Calculator.prototype = {
  add : function (x, y) {
    var val = x + y;
    this.sum = val;
  }
};
```

Prototype functions shared across all object instances

Reference instance variable

Prototype Pattern Structure

```
var Calculator = function (sum) {
  this.sum = sum;
  };

Calculator.prototype = {
  add : function (x, y) {
    var val = x + y;
    this.sum = val;
  }
};
```

Using Technique called: Object Literal

```
var calc = new Calculator(0.0);
console.log(calc.sum);
calc.add(40,2);
console.log(calc.sum);
```

Exercise 4.1: Using the Prototype Pattern

HANDS-ON EXERCISE

20 min

- Navigate to the Ch04\UsingthePrototypePattern folder .
- Define a Calculator class in a JavaScript file.
  a. Use the Prototype pattern to define the four basic math operations:
    - i. Add
    - ii. Subtract
    - iii. Multiply
    - iv. Divide

The Revealing Prototype Pattern

- Assigns an Immediately Invoked Function Expression (IIFE) to the prototype
- Benefits:
  - Provides encapsulation
  - Returns an object literal
  - Supports public and private members
  - Functions defined in the constructor are public
    - As are the functions in the return section of the prototype
  - Functions defined in the prototype function are private
    - Outside the returned object’s scope
- Challenges:
  - “`this`” is still tricky
  - Constructor separate from prototype definition

Working with `this`

- When working with the Revealing Prototype Patterns, the key word **this** is interesting
- In public functions:
  - **this** is used to access the variables defined in the constructor
  - No problem, since the caller is the object itself and has access to those variables
- When a public function calls a private function:
  - The context of **this** changes
  - The private function does not have access to those variables
- How to deal with this problem?
  - Pass **this** as an argument to the private functions

Step 1: Define the Constructor

- A constructor can be defined using a standard JavaScript function

Constructor

```
var Vehicle = function(wheels, maker, make) {
  this.numWheels = wheels;
  this.manufacturer = maker;
  this.make = make;
  this.startStop = function () {
  this.pressGasPedal();
  this.pressBrakePedal();
  }
}
```

"`this`" required to define instance variables

Step 2: Define the Prototype

- Define an object prototype using an immediately invoked function

```
Vehicle.prototype = function() {
  var go = function() {
    console.log(this.make + " is going");
  };
  var stop = function() {
    console.log(this.make + " is stopping");
  };
  return {
    pressBrakePedal: stop,
    pressGasPedal: go
  }
}();
```

Prototype functions shared across all object instances

Reference instance variable

Revealing Prototype Pattern Structure

```
var Vehicle = function(wheels, maker, make) {
  this.numWheels = wheels;
  this.manufacturer = maker;
  this.make = make;
  this.startStop = function () {
  this.pressGasPedal();
  this.pressBrakePedal();  }};
Vehicle.prototype = function() {
  var go = function() {
    console.log(this.make + " is going");
  };
  var stop = function() {
    console.log(this.make + " is stopping");
  };
  return {
    pressBrakePedal: stop,
    pressGasPedal: go
  }}();
```

Using Technique called:
Immediately Invoked
Function Execution

Optional Exercise 4.2: Using the Revealing Prototype Pattern

HANDS-ON EXERCISE

20 min

- Navigate to the following folder:
  - Ch04\UsingtheRevealingPrototypePattern
- Define a PartTimeMentor class in a JavaScript file.
  - Use the RevealingPrototypePattern to define the PartTimeMentor.
  - A PartTimeMentor has the following data values:
    - An id
    - A firstName
    - A lastName
    - An hourlyPay
    - An hoursWorkedPerWeek
  - A PartTimeMentor defines the following function:
    - calculateWeeklyPay

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Chapter Summary

What Is a Factory Function?

- The purpose of the object factory is to create objects
- Usually implemented in a class or a static method of a class
  - Can produce repeatedly similar objects
  - Provides a way to users of the factory to create objects without knowing the specific type (class) at compile time
- Objects created by the factory method are by design inheriting from the same parent object
  - However, there are specific subclasses implementing specialized functionality
  - Sometimes the common parent is the same class that contains the factory method

What Should the Factory Do for Us?

- We want to have a method that accepts a type given as a string at runtime and then creates and returns specific objects of that type
- We don’t want to use a constructor (“new” statement) to create these objects
  - Just a function that creates objects

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');

console.log(corolla.drive() ); // Vroom, I have 4 doors
console.log(solstice.drive() ); // Vroom, I have 2 doors
console.log(cherokee.drive() ); // Vroom, I have 17 doors
```

Let's Build Cars

- We create a common parent *CarMaker* constructor
- Then we add a static method of the *CarMaker* called *factory()*, which creates car objects
- And also add a prototype.drive function to provide that feature to all cars

```
// parent constructor
function CarMaker() {}

// a method of the parent
CarMaker.prototype.drive = function () {
    return "Vroom, I have " + this.doors + " doors";
};
```

Courtesy: JavaScript Patterns by Stoyan Stefanov

Let's Build Cars (continued)

- Now, we need the specific car models

```
CarMaker.Compact = function() {
    this.doors = 2;
};

CarMaker.Convertible = function() {
    this.doors = 4;
};

CarMaker.SUV = function() {
    this.doors = 17;
};
```

Let’s Build Cars (continued)

```
// the static factory method
CarMaker.factory = function (type) {
    var constr = type;
    var newcar;

    // error if the constructor does not exists
    if (typeof CarMaker[constr] !== "function") {
        throw {
            name: "Error",
            message: constr + " doesn't exist"
        };
    }
    // at this point the constructor is known to exist
    // let's have it inherit the parent but only once
    if (typeof CarMaker[constr].prototype.drive !== "function") {
        CarMaker[constr].prototype = new CarMaker();
    }
    // create a new instance
    newcar = new CarMaker[constr]();
    // optionally call some methods and then return.....
    return newcar;
};
```

Let's Build Cars (continued)

- We now have a method that accepts a type given as a string at runtime and then creates and returns objects of that type
- There is no constructor used with new
    - Just a function that creates objects

```
var corolla = CarMaker.factory('Compact');
var solstice = CarMaker.factory('Convertible');
var cherokee = CarMaker.factory('SUV');

console.log(corolla.drive() ); // Vroom, I have 4 doors
console.log(solstice.drive() ); // Vroom, I have 2 doors
console.log(cherokee.drive() ); // Vroom, I have 17 doors
```

Exercise 4.3: Factories

30 min

- The CarMaker example in the Course Notes defines static factory method that creates JavaScript objects based on the type argument that is passed to that method. The factory method creates a new CarMaker object and returns it.

- It would be better to have the CarMaker actually create a Car object. In this exercise, you will do exactly that.

1. Work in the Ch04\carfactory directory.
2. Define a Car class that contains the following data:
    1. The number of seats
    2. A description of the car
    3. The number of doors
3. The Car class should also define the following functions, each of which should print a message to the console:
    1. Start
    2. Drive
    3. Stop
4. Define some specific car models such as the following:
    1. VW Bug
    2. Jeep Cherokee
    3. Tesla Model S
5. Define a CarFactory class that can create a Car of any model type.

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Chapter Summary

Understanding Scope

- In JavaScript, scope is the set of variables that the code currently has access to
  - The set of variables, objects, and functions that can be accessed
- JavaScript has lexical scoping
  - With function scope
  - Even though it looks like it should have block scope (`{ … }`)
- A new scope is created only when a new function is created
- Nested functions
  - The inner function has access to the outer function scope
  - Known as “lexical scope”
  - Aka “closure”

Nested Functions

- Nested functions have access to outer function scope

```
var amigo = new GangOfFour();
amigo.scope1().scope2().scope3();
```

```
var GangOfFour = function () {
  var amigo1 = "Grady";
  this.scope1 = function () {
    console.log("The first amigo:" + amigo1);
    this.scope2 = function () {
      var amigo2 = "Ivar";
      console.log("Two amigos: " + amigo1 + " " + amigo2);
      this.scope3 = function () {
        var amigo3 = "James";
        console.log("Three egos: " + amigo1 + " " + amigo2 + " " + amigo3);
      };
      return this;
    };
    return this;
  };
};
```

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Chapter Summary

Closures Step by Step

- A closure is a function having access to the parent scope
  - Even after the parent function has closed
- Let’s go step by step to understand why we need closures

The Scope Problem

- We already know private variables and global variables

```
function myFunction() {
    var a = 4;
    return a * a;
}
```

```
var a = 4;
function myFunction() {
    return a * a;
}
```

- However, the lifetime of these variables is very different due to their nature
  - Private variables live within the function they are declared in; whenever the function is called, the variable is created
  - Global variables live as long as your window/web page does
    - A variable that is created **without** the **var** key word is ALWAYS GLOBAL

The Counter Problem

- No privacy!

```
var counter = 0;

function add() {
    counter += 1;
}

add();
add();
add();

// the counter is now equal
to 3
```

- Correct result, but everyone can overwrite counter, without `add()`

- Too private

```
function add() {
    var counter = 0;
    counter += 1;
}

add();
add();
add();

// want the counter to be 3
// but it does not work
```

- Every time we call `add()`, the counter is newly created and set to 1

Nested Functions

```
function add() {
    var counter = 0;
    function plus() {counter += 1;}
    plus();
    return counter;
}
```

- This could work, if we could reach the plus function
  - So far, we can only reach that function if we create an object from add
  - But that is NOT what we are looking for
- If only we could find a way to execute **var counter = 0** only once

Closures

- Remember self-invoking functions and what they do?
- What is happening here?
- The first `add` (declared with `var`) invokes the entire function
  - `add` receives the inner function as return value
- When you call `add()` you actually only invoke the inner function without recreating the counter variable
- This is a **closure** which allows functions to have private variables

```
var add = (function () {
  var counter = 0;
  return function () {
    return counter += 1;
  }
})();

console.log("counter = " + add());
console.log("counter = " + add());
console.log("counter = " + add());
```

Objects vs. Closures

Objects	Closures
Can add functions later; flexibility	Cannot add function; safety
Can use function from other source; Reuse of code!	Has to create function always from scratch; More memory used; Possible redundancy
Need to be carful when a method gets detached from an object, ***this*** will get a different meaning; Complexity!	You don't need to keep track of ***this***
WHEN TO USE	
Less concern with privacy, many instances of an object required	High privacy requirements and few “objects” of that type are needed

Exercise 4.4: Exploring Closures

HANDS-ON EXERCISE

20 min

1. Work in the following folder Ch04\closures.
2. Define Counter class that contains the following data:
    1. A privateCounter that will store a numeric value.
3. The Counter class should define a private function:
    1. A function named modify that has one argument.
    2. The argument is added to the privateCounter.
4. The Counter class should return the following public methods:
    1. Increment
        1. Calls modify(1)
    2. Decrement
        1. Calls modify(-1)
    3. Value
        1. Returns the current value of privateCounter

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Chapter Summary

Processing Collections

- Processing a collection of items is very common
- JavaScript supports several ways of iterating over a collection
  - Simple for loops
  - Iterators and generators
    - Provide a mechanism for customizing the behavior of `for…of` loops

Iterators

- An iterator knows how to access the items in a collection
  - One at a time
  - Keeps track of its current position in the collection
- JavaScript iterators
  - Provide a `next()` function
    - Returns the next item in the collection
    - The object returned has two properties
      - `done`
      - `value`

```
function makeAnIterator(array) {
    var nextIndex = 0;

    return {
        next: function() {
            return nextIndex < array.length ?
                {value: array[nextIndex++], done: false} :
                {done: true};
        }
    };
}
```

```
var it = makeAnIterator(['Grady', 'Ivar', 'James']);
console.log(it.next().value); // 'Grady'
console.log(it.next().value); // 'Ivar'
console.log(it.next().done);  // false
```

Iterables

- An iterable in ES6 is an object that defines its iterator
- The `for…of` loop can loop over any iterable
- You can create your own iterables
  - Define a function on the object names `@@iterator`
  - Or use `Symbol.iterator` as the function name
- Since JavaScript does not have interfaces:
  - Iterable is a convention
- JavaScript does provide some built-in iterables
  - String
  - Array
  - Map
  - Set

```
let iterableUser = {
    name: 'Grady',
    lastName: 'Booch',
    [Symbol.iterator]: function*(){
        yield this.name;
        yield this.lastName;
    }
}

// logs 'Grady' and 'Booch'
for(let item of iterableUser){
    console.log(item);
}
```

Generators

- A generator is a special function
  - Allows you to write an algorithm that maintains its own state
  - And can be paused and resumed
- A generator is a factory for iterators
- A generator function is marked with an *
  - And contains at least one yield statement

```
function* generateRandomNumbers(){
  let start = 1;
  let end = 42;
  while(true)
    yield Math.floor((Math.random() * end) + start);
}
```

```
//no execution here
//just getting a generator
let sequence = generateRandomNumbers();

for(let i=0;i<5;i++){
    console.log(sequence.next());
}
```

Advanced Generators

- Generators compute their yielded values on demand
  - Efficiently represent a sequence of values that are expensive to compute
  - Or even an infinite sequence!
- The `next()` function accepts an input argument
  - Can be used to modify the internal state of the generator
  - Will be used as the result of the last yield expression of the generator
- *Note:* generators do NOT like recursion!

```
function* fibonacci() {
  var fn1 = 1;
  var fn2 = 1;
  while (true) {
    var current = fn1;
    fn1 = fn2;
    fn2 = current + fn1;
    var reset = yield current;
    if (reset) {
        fn1 = 1;
        fn2 = 1;
    }}}
```

```
var sequence = fibonacci();
console.log(sequence.next().value);     // 1
console.log(sequence.next().value);     // 1
console.log(sequence.next().value);     // 2
console.log(sequence.next().value);     // 3
console.log(sequence.next(true).value); // 1
console.log(sequence.next().value);     // 1
console.log(sequence.next().value);     // 2
console.log(sequence.next().value);     // 3
```

Optional Exercise 4.5: Iterators and Generators

HANDS-ON EXERCISE

20 min

1. Work in the following folder Ch04\generators.
2. Create a new JavaScript file that defines the Fibonacci sequence generator that is defined in the Course Notes.

Chapter Concepts

Function Techniques

JavaScript Prototype

Creating a JavaScript Factory

JavaScript Scope

Closures

Iterators and Generators

Chapter Summary

Chapter Summary

In this chapter, we covered JavaScript technologies:

- Function techniques
- Prototypes
- Factories
- Closures
- Iterators and Generators