

OBJECT ORIENTED JAVASCRIPT



Agenda

2

- Understand how to simulate major Object Oriented concepts
- Class
- Instance and Static members
- Inheritance
- Polymorphism
- Namespace
- altJS

From Module to Class

3

- Previous chapter suggested a technique to implement a module
- A module is essentially a collection of global methods that manage some global state
- A module cannot be duplicated
 - ▣ The self executing function can only be invoked once
- However, if we use regular function we can invoke it multiple times
 - ▣ Each time a new “module” is created

Function as a Factory

4

```
function Point(x, y) {  
  var _x = x;  
  var _y = y;  
  
  function dump() {  
    console.log(_x + ", " + _y);  
  }  
  
  return {  
    dump: dump  
  };  
}
```

```
var pt1 = Point(5, 5);  
var pt2 = Point(10, 10);  
  
pt1.dump();  
pt2.dump();
```

- Note the naming convention (Pascal casing)

Pros & Cons

5

- Same syntax (almost) as module definition
- Encapsulation is supported
- Hard to support inheritance
 - ▣ State is hidden and cannot be shared with derived class
- No use of keyword **new** when instantiating objects
- **Every time **Point** is invoked a new **dump** function is created**
 - ▣ May have performance and memory impact
 - ▣ Can a method be defined once and shared between different objects?

Function as Constructor

6

- Any JavaScript function can serve as a constructor

```
function F() {  
}  
  
var f1 = new F();  
var f2 = new F();
```

- During function invocation **this** points to the newly created object

```
function Point(x, y) {  
    this.x = x;  
    this.y = y;  
}  
  
var pt1 = new Point(5, 5);
```

Function as Constructor

7

- The **new** keyword can be understood as

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
  
var pt1 = new Point(5, 5);  
  
var pt1 = {};  
Point.call(pt1, 5, 5);
```

- Does it mean that **new** is just a syntactic sugar?
 - ▣ No, look at next slide

Behind the scene

8

- An object created by a constructor is “linked” back to the constructor’s prototype

```
var pt1 = new Point(5, 5);  
  
var pt1 = {};  
pt1.__proto__ = Point.prototype;  
Point.call(pt, 5, 5);
```

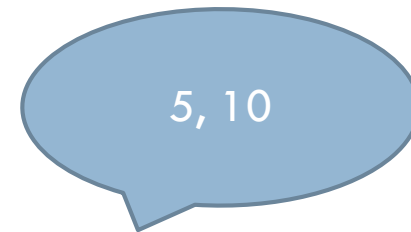
- Once created, an object is bound to its prototype for its whole lifetime
- Some browsers support the __proto__ reference
 - ▣ Chrome, Firefox, IE11

Prototype

9

- Every object is linked to its prototype
- An object “inherits” all the fields and methods specified by the prototype

```
function Point(x, y) {  
  this.x = x;  
  this.y = y;  
}  
  
Point.prototype.dump = function () {  
  console.log(this.x + ", " + this.y);  
}  
  
var pt = new Point(5, 10);  
pt.dump();
```



Prototype (more ..)

10

- When accessing an object's member the browser first looks at the object itself
- If not found, the prototype is considered
 - ▣ Continues in a recursive manner
 - ▣ Stops when `Object.prototype` is reached
- The prototype is being used only for read operations
- Write operations effect the object itself and not its prototype

Prototype Chaining

11

- Constructor's prototype is empty by default and is linked to **Object.prototype**
 - ▣ That means that custom object inherits all methods from **Object.prototype**

```
var pt = new Point(5,10);  
  
pt.dump();  
  
console.log(pt.toString());  
console.log(pt.hasOwnProperty("x"));
```

Extension Methods

12

- Every built-in type has its own prototype
 - For example, `Function.prototype`
- We can “extend” built-in data types by manipulating their prototype

```
String.prototype.format = function (arg1, arg2, arg3) {  
    ...  
}  
  
var str = "Hello {0}";  
str.format("World");
```

- Why is that considered a bad practice?

Class

13

- Using constructor and prototype we can simulate a class
- Methods go into the **prototype**
- Fields go into the **this** (during ctor invocation)
- Encapsulation is not supported
 - ▣ Since prototype's methods need access to the object state
- What about static members ?
 - ▣ They are attached to the **constructor**

Class

14

```
function Account(name, email) {  
  this.id = Account.generateId();  
  this.name = name;  
  this.email = email;  
}  
  
Account.prototype.dump = function () {  
  console.log(this.id + ": " + this.name);  
}  
  
Account.nextId = 1000;  
  
Account.generateId = function () {  
  return Account.nextId++;  
}
```

```
var acc = new Account("Ori", "ori@g.com");  
acc.dump();
```

Inheritance

15

- Inheritance is a bit tricky
- Object level
 - ▣ Derived object should contain both base and derived fields
 - ▣ Achievable by calling the base ctor from the derived ctor
- Prototype level
 - ▣ Base class methods should be accessible through derived objects
 - ▣ Achievable by chaining the prototype of the derived class to the prototype of the base class

Inheritance – Object Level

16

- Derived ctor should invoke base ctor and let it manipulate the object being created
- Assuming **Programmer** derives from **Employee** what is wrong with below implementations?

```
function Employee(name) {  
    this.name = name;  
}
```

```
function Programmer(name, progLang) {  
    Employee(name);  
  
    this.progLang = progLang;  
}
```

```
function Programmer(name, progLang) {  
    new Employee(name);  
  
    this.progLang = progLang;  
}
```


Inheritance – Calling base ctor

17

- We need to explicitly send the `this` pointer when invoking the base ctor
- **Function.call** and **Function.apply** can do that

```
function Employee(name) {  
    this.name = name;  
}  
  
function Programmer(name, progLang) {  
    Employee.call(this, name);  
  
    this.progLang = progLang;  
}
```

Inheritance – Class Level

18

- A derived object inherits all methods defined in its own prototype
 - ▣ But what about methods from the base prototype?
- By default a prototype object is linked to `Object.prototype`
 - ▣ Remember that once an object is created you cannot change its prototype
- Need to create a new prototype object
 - ▣ Which is linked to base class prototype
 - ▣ Any idea?

Inheritance – Class Level

19

- Create a new base class object
- Use it as the prototype for derived class
 - ▣ Quite strange (from OOP perspective)
 - ▣ But it works (at least from Prototyping perspective)

```
function Programmer(name, progLang) {  
    Employee.call(this, name);  
    this.progLang = progLang;  
}  
  
Programmer.prototype = new Employee();  
  
var prog = new Programmer(123, "Ori", "JavaScript");
```

Inheritance – Prototype Chaining

20

- Previous technique works most of the time
- But still it feels wrong
 - ▣ Why do we need to create a new base class object just to fix prototype chaining
 - ▣ What parameters should we send to the base class ctor?
- It would be better to create empty object that does nothing but is still linked to the base class prototype

Inheritance – The Right Way

21

```
function Programmer(name, progLang) {  
    Employee.call(this, name);  
  
    this.progLang = progLang;  
}  
  
function Dummy() { }  
Dummy.prototype = Employee.prototype;  
Programmer.prototype = new Dummy();  
  
Programmer.prototype.changeLang = function (progLang) {  
    this.progLang = progLang;  
}  
  
var prog = new Programmer(123, "Ori", "JavaScript");
```

Inheritance - Reuse

22

- The Dummy trick can be encapsulated by **inherit** function

```
function inherit(derived, base) {  
  function Dummy() { }  
  Dummy.prototype = base.prototype;  
  
  derived.prototype = new Dummy();  
}
```

```
function Programmer(name, progLang) {  
  Employee.call(this, name);  
  this.progLang = progLang;  
}  
  
inherit(Programmer, Employee);
```

Polymorphism

23

- How can a derived class override methods from the base class?
 - ▣ Just add the function to the derived prototype
 - ▣ Prototype chaining ensures that derived prototype has higher precedence than base prototype
- Actually, you can override the method in the object itself
 - ▣ No equivalent concept from static OO languages
 - ▣ Although possible, not so common in JavaScript

Polymorphism – Full Sample

24

```
function Shape(x, y) {...}

Shape.prototype.draw = function () {
  console.log("shape");
}

function Rect(x, y, width, height) {
  Shape.call(this, x, y);
  this.width = width;
  this.height = height;
}

inherit(Rect, Shape);

Rect.prototype.draw = function () {
  console.log("rect");
}
```

```
var shapes = [
  new Shape(5, 10),
  new Rect(5, 10, 100, 200),
];

for (var i = 0; i < shapes.length; i++) {
  var shape = shapes[i];
  shape.draw();
}
```


Calling base method

25

```
function Shape(x, y) {...}

Shape.prototype.dump = function () {
  console.log("x = " + this.x);
  console.log("y = " + this.y);
}

function Rect(x, y, width, height) {...}

inherit(Rect, Shape);

Rect.prototype.dump = function () {
  Shape.prototype.dump.call(this);

  console.log("width = " + this.width);
  console.log("height = " + this.height);
}
```

instanceof

26

- JavaScript offers a keyword named **instanceof**
- Allows you to query an object regarding its runtime type
- **instanceof** returns true if the specified object is linked to specified constructor (directly or indirectly)

```
var r = new Rect();  
console.log(r instanceof Rect); // true  
console.log(r instanceof Shape); // true  
console.log(r instanceof Object); // true  
console.log(r instanceof String); // false
```

Namespace

27

- Declaring constructors at the global scope might create name conflicts with other programmers/libraries
- We can reduce the chances for conflicts by declaring global variable and attach to it all constructors
- As long as the global variable has non conflicting name we are safe
 - ▣ Usually your product name will do the work

Namespace

28

□ Declaring the namespace

```
var MyProduct = {};
```

□ Attach the constructor to the namespace variable

```
MyProduct.Shape = (function () {  
    function Shape(x, y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    Shape.prototype.dump = function () {  
        ...  
    }  
  
    return Shape;  
})();
```

```
var s = new MyProduct.Shape(5, 10);  
s.dump();
```

Namespace Cross Multiple Files

29

- Previous technique is problematic if repeated cross multiple JavaScript files
 - ▣ Each file overwrites the namespace variable
- You can move the namespace variable declaration into a single file and include it first inside the HTML
- Better solution

```
var MyProduct = MyProduct || {};
```
- This line of code can be repeated multiple times

Complete Sample

30

Shape.js

```
var PaintApp = PaintApp || {};

PaintApp.Shape = (function () {
  function Shape(x, y) {
    this.x = x;
    this.y = y;
  }

  Shape.prototype.dump = function () {
    console.log("x = " + this.x);
    console.log("y = " + this.y);
  }

  return Shape;
})();
```

Rect.js

```
var PaintApp = PaintApp || {};

PaintApp.Rect = (function () {
  var Shape = PaintApp.Shape;

  function Rect(x, y, width, height) {
    Shape.call(this, x, y);

    this.width = width;
    this.height = height;
  }

  inherit(Rect, Shape);

  Rect.prototype.dump = function () {
    Shape.prototype.dump.call(this);

    console.log("width = " + this.width);
    console.log("height = " + this.height);
  }

  return Rect;
})();
```

Common.js

```
function inherit(derived, base) {
  function Dummy() {}
  Dummy.prototype = base.prototype;
  derived.prototype = new Dummy();
}
```

App.js

```
var s = new PaintApp.Rect(5, 10, 20, 20);
s.dump();
```

Too much details?

31

- At first glance you might be thinking that we are trying too much
- After all, JavaScript is not a real object oriented programming language
- Good news
 - ▣ You are not alone
 - ▣ It takes time to get used to it
 - ▣ Many programmers think that is quite fun
 - ▣ **Other prefer “Compile to JavaScript” languages**

altJS Languages

32

- There are many
 - ▣ CoffeeScript
 - ▣ Dart
 - ▣ Typescript
 - ▣ GWT
 - ▣ SharpKit
- Others
 - ▣ <https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS>

altJS – How to choose?

33

- Probably a matter of style
- Need to think about
 - ▣ Whether significant ramp up is required
 - ▣ Integrating with JavaScript libraries
 - ▣ Tooling support
 - ▣ Debugging
 - ▣ Future ECMAScript standard
 - ▣ Native browser support
 - ▣ Extensive class library

Summary

34

- Many say that JavaScript is a prototype based language
- It has object oriented capabilities
- But requires the programmers to understand major JavaScript concepts like
 - ▣ Closure
 - ▣ Constructor
 - ▣ Prototype