



中山大學

SUN YAT-SEN UNIVERSITY

# Lecture 03.

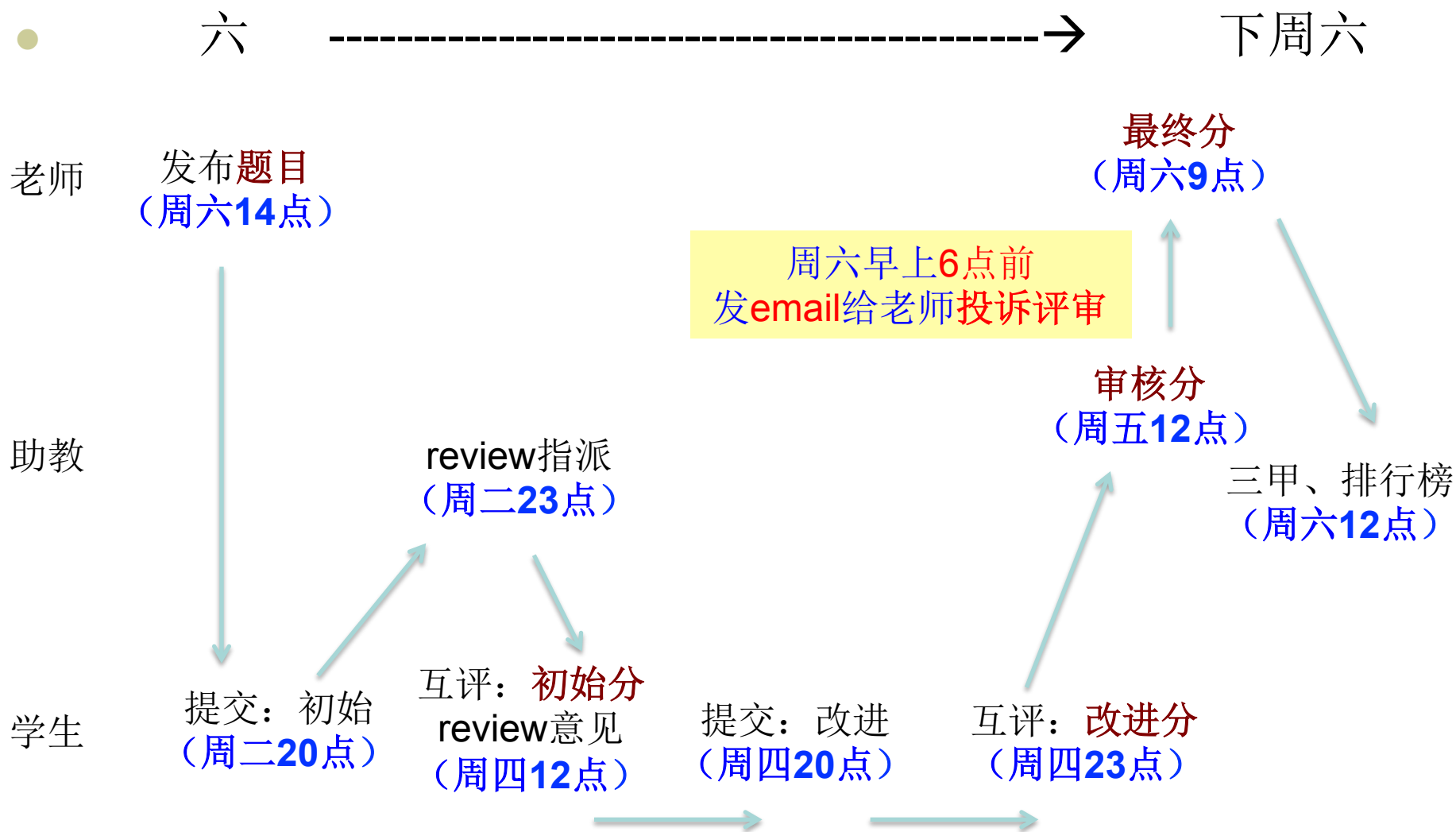
## Closure & Prototype

**Modern Web Camp**

(<http://my.ss.sysu.edu.cn/wiki/display/MWC> , supported by ReCall)

School of Software, Sun Yat-sen University

# 作业流程：总结



# Camp Rules

---

- 简介
  - 为期2~4个月
  - 20个学位（确定之后，将安排实验室位置）
  - 总冠军得到MBP和1000¥
    - 使用权，离开实验室，需要归还
- 入营规则
  - 服从老师安排
    - 寒暑假、实习、毕业设计
  - 学位有限、公平竞争
  - 按时上课
  - 按时交作业
  - 作业未按时提交，0分，
  - 两次缺课，或两次未按时交作业，自动退营

# Camp Rules

---

- 时间保障
  - 安排学位后，每周至少**20小时**到位
  - 外出超过**3天**，必须向导师或指定负责人请假
- 加入退出机制
  - 入营向导师和**TA**提出申请，需循序渐进，层次递进各个阶段
  - 每个阶段，导师会进行**review**，不合格者会被要求退出
  - 可自行申请退出，在导师批准，并交接好工作和资源后，退出
- 保密
  - 严格保守参与项目的技术与商业秘密
  - 未经导师许可，不得在营外使用任何项目代码、文档、设计

---

*Design and  
programming  
are human activities;  
forget that and all is lost.*



*--Bjarne Stroustrup, 1991*

```

27 var count = [];
28 for (var i = 0; i < 6; i++)
29   count[i] = 1;
30 function makeAllTablesSortable() {
31   var arr = getAllTables();
32   var tthead = [],
33       tbody = [], temp1, temp3, temp4, va, temp5, v, g, temp, str = [], td = [], td1 = [], tr = [], th = [], th1;
34
35   td1 = document.getElementsByTagName("td");
36   th1 = document.getElementsByTagName("th");
37
38   for (g = 0; g < th1.length; g++)
39     th[g] = th1[g].innerHTML;
40
41   for (v = 0; v < td1.length; td++)
42     td[0] = td1[v].innerHTML;
43
44   if (i >= 0 && i <= 5) {
45     temp1 = i;
46     for (va = 0; va < 3; va++) {
47       if (temp1 > 3)
48         t = temp1 + 6;
49       else
50         t = temp1;
51       str[va] = td[t + 3 * va];
52     }
53     if (count[temp1 - 1] == 1) {
54       th1[temp1].style.backgroundImage = "url('ascend.png)";
55       count[temp1 - 1] = 0;
56       str.sort();
57     } else {
58       th1[i].style.backgroundImage = "url('descend.png)";
59       str.reverse();
60     }
61     th1[temp1].style.backgroundPosition = "bottom right";
62     th1[temp1].style.backgroundColor = "#6699FF";
63     for (temp3 = 0; temp3 < str.length; temp3++) {
64       for (temp4 = 0; temp4 < str.length; temp4++) {
65         if (temp1 > 3)
66           t1 = temp1 + 6;
67         else
68           t1 = temp1;
69         if (str[temp3] == td[3 * temp4 + t1]) {
70           for (temp5 = 0; temp5 < str.length; temp5++) {
71             var str1 = td[3 * temp4 + t1].innerHTML;
72             td1[t1 + 3 * temp1].innerHTML = td1[t1].innerHTML;
73             td1[t1].innerHTML = str1;
74
75             str1 = td[3 * temp4 + t1 + 1].innerHTML;
76             td1[t1 + 3 * temp1 + 1].innerHTML = td1[t1 + 1].innerHTML;
77             td1[t1 + 1].innerHTML = str1;
78
79             str1 = td[3 * temp4 + t1 + 2].innerHTML;
80             td1[t1 + 3 * temp1 + 2].innerHTML = td1[t1].innerHTML;
81             td1[t1 + 2].innerHTML = str1;
82
83

```

Naming ?

- Modularization ?
- Readability ?
- Correctness ?

# Code Review

```

1  define ->
2  {
3      ###
4      * convert an object to array
5      * @param _object: an array to be converted
6      * @return      : an array
7      ###
8      toArray : (_object)->
9          Array.prototype.slice.call(_object)
10
11     ###
12     * traverse an array
13     * @param _array: an array to be traversed
14     * @param _callback:
15     ###
16     each : (_array, _callback) ->
17         length = _array.length
18         while length
19             length -= 1
20             _callback.call(this, _array[length], length)
21 }

```

Naming ?

- Modularization ?
- Readability ?
- Correctness ?

Documentation comments, really good practice ?

# MWC is only for

---

# THE BEST



# The core tech. of Web

大师JavaScript

普通JavaScript

Bigger than bigger

CSS3

HTML5

CSS

HTML



# Secrets of JavaScript

**CLOSURE**

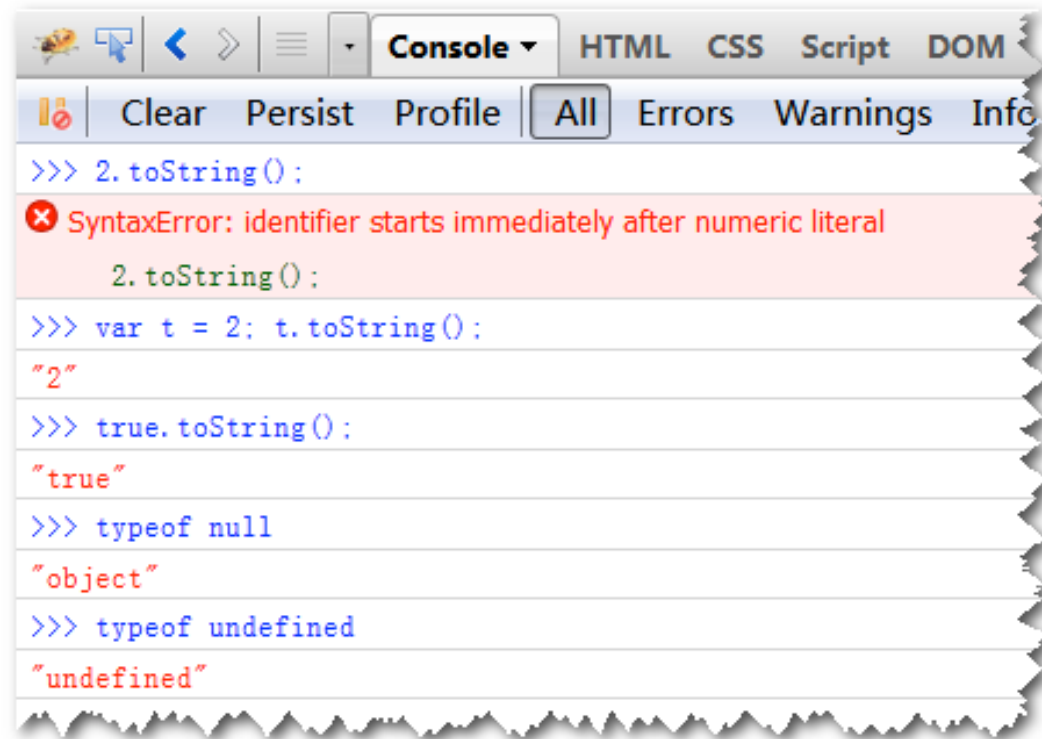
**PROTOTYPE**

**TOP SECRET**

**{JAVASCRIPT}**

# Everything in JS is an Object

- What is an object
- What is OOP



- 只有数字字面表达（number literal）和undefined不是对象

# Everything in JS is an Object

---

- JavaScript语言中对象就是一个容器，容器包含了一系列属性（**properties**）。
- 每个属性都是一组名值对，属性名可以是任意字符串，包括空字符串，属性值可以是JavaScript中除**undefined**任何外合法的值。
- 当属性值为函数时，这个属性也称为对象的方法。
- JavaScript对象的最大特点就是其动态性（动态对象，**dynamic object**），一个对象可以在创建之后，动态地增加和删除属性和方法。

# Object literal

---

## 源代码 9-1 对象字面表达（Object Literal）示例

```
var member = {  
    name: '张三',  
    age: 23,  
    'goto': 'United States',  
    say: function() {  
        return this.name + '前往' + this['goto'];  
    }  
};  
  
alert(member.age);           // 23  
alert(member.say());        // 张三前往United States
```



# Object literal

## 源代码 9-2 属性读取与更新示例

```
var member = {
    age: 23,
    'goto': 'United States',
};

alert(member.age);           // 23
alert(member['age']);        // 23
alert(member['goto']);       // United States
alert(member.goto);          // 语法错误！（某些JavaScript引擎能够执行）
```

## 源代码 9-3 安全读取和使用属性示例

```
var member = {
    age: 23,
    'goto': 'United States',
};

var name = member.name;           // undefined
var name = member.name || '未知'; // '未知'
member.address.state              // 抛出TypeError异常
member.address && member.address.state // undefined
```

# Use of properties

## 源代码 9-4 更新和移除属性示例

```
var member = {  
    age: 23,  
    'goto': 'United States',  
};  
  
member.age = 34;  
alert(member.age);           // 34  
member.name = '李四';  
alert(member.name);          // '李四'  
delete member.name;  
alert(member.name);          // undefined
```



**JavaScript 动态对象 (dynamic objects):** 传统静态类型、强类型语言，如：C++、Java 等等，对象的类型固定，创建之后就不能改变增加或者删除其属性和方法。JavaScript 和很多动态类型、弱类型语言，如：ruby，python 等等，对象的类型不固定，都可以在对象创建后，灵活地增加和删除属性与方法。

# Array literal

---

- Array is a special kind of object

## 源代码 9-6 数组字面表达示例

```
var a = ['a', 'b', 'c', 0, {name: '张三'}];  
alert(typeof a); // object  
alert(a.length); // 5  
alert(a[4].name); // '张三'
```

JavaScript 的对象在使用中传递的都是引用，没有传值，即复制传递的方式。

## 源代码 9-5 更新和移除属性示例

```
var peter = {nickname: 'rabbit'};  
var littlePeter = peter;  
peter.nickname = 'bear';  
alert(littlePeter.nickname); // 'bear'
```



# Function is also an Object

---

## 源代码 9-7 函数对象示例

```
var func = function(otherFunc) {  
    alert('func');  
    otherFunc();  
    return otherFunc;  
};  
  
var func2 = function() {  
    alert('func2');  
};  
  
func.method = function() {  
    alert('method of func');  
};  
  
var obj = {  
    myFunc : func  
};  
  
var arr = [func, func2];  
  
func.method();           // method of func  
obj.myFunc(func2);       // func func2  
arr[0](arr[1])();        // func func2 func2
```

# Scope

---

## 源代码 9-9 JavaScript 嵌套作用域示例

```
function outter() {  
    var outterName = 'outter name';  
    var inner = function() {  
        alert(outterName);    // outter name  
        var innerName = 'inner name';  
    };  
    inner();  
    alert(innerName); // ReferenceError: innerName is not defined  
}  
  
outter();
```

- var 局部；无 var 全局
- 局部为function block scope

# Duration

---

## 源代码 9-11 JavaScript 词法作用域（lexical scope）示例

```
function outter() {  
    var secret = 'secret';  
    inner = function() {  
        alert(secret);  
    };  
    inner();  
}  
outter(); // secret  
inner(); // secret
```

词法作用域是 JavaScript 构造闭包（closure）的基础，参考 f) 闭包。

# arguments, caller, callee

---

源代码 9-12 arguments 参数示例

```
var sum = function () {  
    var i, sum = 0;  
    for (i = 0; i < arguments.length; i += 1) {  
        sum += arguments[i];  
    }  
    return sum;  
};  
  
alert(sum(4, 8, 15, 16, 23, 42)); // 108
```

# Constructor

---

## 源代码 9-13 函数构造子示例

```
var Foo = function() {  
    this.name = 'foo';  
}  
  
var result = Foo();           // 普通函数  
alert(result);               // undefined  
alert(name);                 // foo  
var result = new Foo();      // 构造子  
alert(result);               // Object  
alert(result.name);          // foo
```

# 4 kinds of function invoking

---

- 普通函数
- 构造子
- 方法(method)调用
- 应用(apply、call)调用

# this problem

---

## 源代码 9-15 this 丢失问题示例

```
var peter = {name: 'peter'};
var name = 'global';
var sayHello = function() {
    var helper = function() {
        alert(this.name + ' says hello');
    };
    helper();
};
peter.greeting = sayHello;

peter.greeting(); // global says hello (应该是peter says hello)
```

# that for this

---

## 源代码 9-16 使用 that 模式修复 this 丢失问题示例

---

```
var peter = {name: 'peter'};
var name = 'global';
var sayHello = function() {
    var that = this;
    var helper = function() {
        alert(that.name + ' says hello');
    };
    helper();
};
peter.greeting = sayHello;

peter.greeting(); // peter says hello
```

---



# call , apply

---

## 源代码 9-17 函数应用（apply、call）模式调用示例

```
var sayHello = function(message, to){  
    alert(this.name + ' says ' + message + ' to ' + to);  
};  
var peter = {name: 'peter'};  
var name = 'global';  
sayHello.apply(this, ['hello', 'Marry']); //global says hello to Marry  
sayHello.apply(peter, ['hello', 'Marry']); // peter says hello to Marry  
sayHello.call(this, 'hello', 'Marry'); // global says hello to Marry  
sayHello.call(peter, 'hello', 'Marry'); // peter says hello to Marry
```

# 4 kinds of function invoking

表 9-1 4 种调用方式的对比

调用模式	this	无 <b>return</b> 时的返回值
函数模式	顶层对象（在浏览器中执行时为 <b>window</b> ）	<code>undefined</code>
方法模式	当前对象（方法从属的对象，即成员操作符 “.” 的左侧）	<code>undefined</code>
构造子模式	正在构造的对象	<code>this</code> （构造好的对象）
应用模式	第一个参数	<code>undefined</code>

# Closure

---

- 闭包(closure)指函数和函数所能访问的函数体外部局部变量构成的组合
- 闭包中的函数称为闭包函数,闭包函数能够访问的函数体外部局部变量称为闭包变量
- JavaScript 的scope和duration使得闭包, 自然、容易、强大

## 源代码 9-18 闭包 (closure) 示例

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

```
alert(counter()); // 0  
alert(counter()); // 1  
alert(counter()); // 2
```

```
.....  
<ol>  
    <li>第一项</li>  
    <li>第二项</li>  
    <li>第三项</li>  
    <li>第四项</li>  
</ol>  
.....
```

#### ----- 错误的JavaScript -----

```
window.onload = function() {  
    var lis = document.getElementsByTagName('li');  
    for (var i = 0; i < lis.length; i++) {  
        lis[i].onclick = function() {  
            alert(i);  
        }  
    }  
}
```

#### ----- 正确的JavaScript(使用闭包) -----

```
window.onload = function() {  
    var lis = document.getElementsByTagName('li');  
    for (var i = 0; i < lis.length; i++) {  
        lis[i].onclick = function(i) {  
            return function() {  
                alert(i);  
            };  
        }(i);  
    }  
}
```

# Functional Programming

---

## 源代码 9-22 Memoization 示例

----- 原始fibonacci -----

```
var fibonacci = function(n) {  
  return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);  
};  
  
for (var i = 0; i <= 10; i += 1) {  
  document.writeln('// ' + i + ': ' + fibonacci(i));  
} // fibonacci被调用452次
```

# Functional Programming

---

```
----- Memoization fibonacci -----  
var fibonacci = function() {  
  var memo = [0, 1];  
  var fib = function(n) {  
    var result = memo[n];  
    if (typeof result !== 'number') {  
      result = fib(n - 1) + fib(n - 2);  
      memo[n] = result;  
    }  
    return result;  
  };  
  return fib;  
}();  
  
for (var i = 0; i <= 10; i += 1) {  
  document.writeln('// ' + i + ': ' + fibonacci(i));  
} // fibonacci被调用29次
```

# Functional Programming

## 源代码 9-23 Memoization 示例

```

----- memoizer -----
var memoizer = function(memo, fundamental) {
  var shell = function(n) {
    var result = memo[n];
    if (typeof result !== 'number') {
      result = fundamental(shell, n);
      memo[n] = result;
    }
    return result;
  };
  return shell;
};

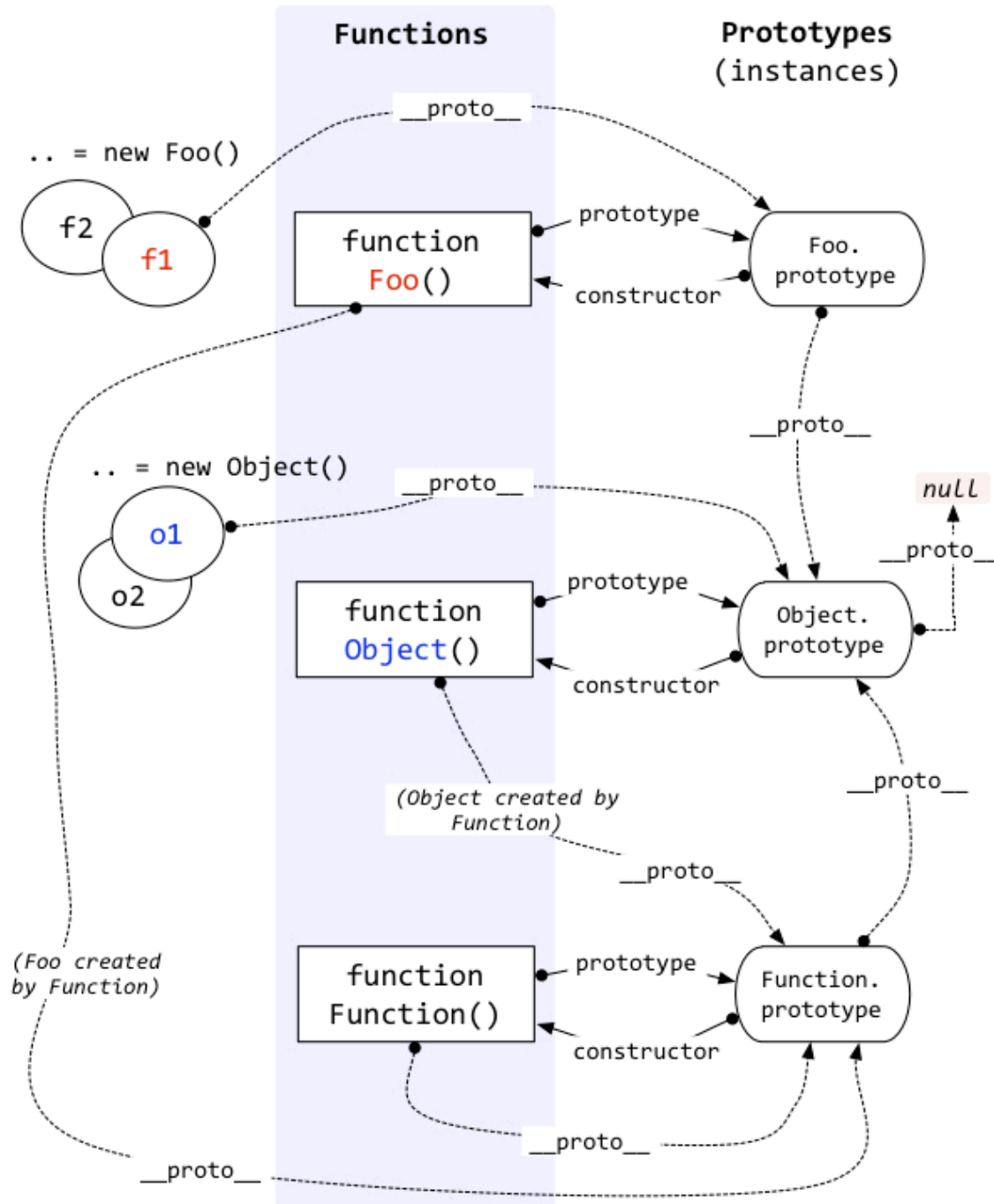
----- memoizer fibonacci函数 -----
var fibonacci = memoizer([0, 1], function(shell, n) {
  return shell(n - 1) + shell(n - 2);
});

----- memoizer阶乘函数 -----
var factorial = memoizer([1, 1], function(shell, n) {
  return n * shell(n - 1);
});

```

# Prototype

- `__proto__`, 缺属性去那儿找
- `__proto__` 指向构造子的prototype
- 构造子的prototype用constructor反指构造子
- 构造子的prototype也是object, 也有自己的构造子
- 形成了`__proto__`链条





# Singletons

```
var singleton = (function () {  
    var privateVariable;  
    function privateFunction(x) {  
        ...privateVariable...  
    }  
    return {  
        firstMethod: function (a, b) {  
            ...privateVariable...  
        },  
        secondMethod: function (c) {  
            ...privateFunction()...  
        }  
    };  
})();
```

# A Module Pattern

```
var singleton = (function () {  
    var privateVariable;  
    function privateFunction(x) {  
        ...privateVariable...  
    }  
    return {  
        firstMethod: function (a, b) {  
            ...privateVariable...  
        },  
        secondMethod: function (c) {  
            ...privateFunction()...  
        }  
    };  
})();
```

# Applications are Singletons

```
MYAPP.MyApplication = (function () {  
    var privateVariable;  
    function privateFunction(x) {  
        ...privateVariable...  
    }  
    return {  
        firstMethod: function (a, b) {  
            ...privateVariable...  
        },  
        secondMethod: function (c) {  
            ...privateFunction()...  
        }  
    };  
})();
```

# Privileged Method

- A Privileged Method is a function that has access to secret information.
- A Privileged Method has access to private variables and private methods.
- A Privileged Method obtains its secret information through closure.

Module pattern is easily transformed into a powerful constructor pattern.

# Power Constructors

1. Make an object.
  - Object literal
  - **new**
  - **Object.create**
  - call another power constructor

# Power Constructors

1. Make an object.
  - Object literal, `new`, `Object.create`, call another power constructor
2. Define some variables and functions.
  - These become private members.

# Power Constructors

1. Make an object.
  - Object literal, `new`, `Object.create`, call another power constructor
2. Define some variables and functions.
  - These become private members.
3. Augment the object with privileged methods.



# Power Constructors

1. Make an object.
  - Object literal, `new`, `Object.create`, call another power constructor
2. Define some variables and functions.
  - These become private members.
3. Augment the object with privileged methods.
4. Return the object.

# Step One

```
function myPowerConstructor(x) {  
    var that = otherMaker(x);  
}
```

# Step Two

```
function myPowerConstructor(x) {  
    var that = otherMaker(x);  
    var secret = f(x);  
}
```

# Step Three

```
function myPowerConstructor(x) {  
    var that = otherMaker(x);  
    var secret = f(x);  
    that.priv = function () {  
        ... secret x that ...  
    };  
}
```

# Step Four

```
function myPowerConstructor(x) {  
    var that = otherMaker(x);  
    var secret = f(x);  
    that.priv = function () {  
        ... secret x that ...  
    };  
    return that;  
}
```

# Power Constructor

- Public methods (from the prototype)

```
var that = Object.create(my_base) ;
```

- Private variables (**var**)
- Private methods (inner functions)
- Privileged methods (**that...**)
- No need to use **new**

```
myObject = power_constructor() ;
```

# Functional Inheritance

- A power constructor calls another constructor, takes the result, augments it, and returns it as though it did all the work.

```
function symbol(s, p) {  
    return {  
        id: s,  
        lbp: p,  
        value: s  
    };  
}
```

```
function delim(s) {  
    return symbol(s, 0);  
}
```

```
function stmt(s, f) {  
    var x = delim(s);  
    x.identifier = true;  
    x.reserved = true;  
    x.fud = f;  
    return x;  
}
```

```
function blockstmt(s, f) {  
    var x = stmt(s, f);  
    x.block = true;  
    return x;  
}
```



# Pseudoclassical Inheritance

```
function Gizmo(id) {  
    this.id = id;  
}  
Gizmo.prototype.toString = function () {  
    return "gizmo " + this.id;  
};
```

```
function Hoozit(id) {  
    this.id = id;  
}  
Hoozit.prototype = new Gizmo();  
Hoozit.prototype.test = function (id) {  
    return this.id === id;  
};
```

# Functional Inheritance

```
function gizmo(id) {  
  return {  
    id: id,  
    toString: function () {  
      return "gizmo " + this.id;  
    }  
  };  
}
```

```
function hoozit(id) {  
  var that = gizmo(id);  
  that.test = function (testid) {  
    return testid === this.id;  
  };  
  return that;  
}
```

# Secrets

```
function gizmo(id) {  
    return {  
        toString: function () {  
            return "gizmo " + id;  
        }  
    };  
}
```

```
function hoozit(id) {  
    var that = gizmo(id);  
    that.test = function (testid) {  
        return testid === id;  
    };  
    return that;  
}
```

# Shared Secrets

```
function gizmo(id, secret) {  
    secret = secret || {};  
    secret.id = id;  
    return {  
        toString: function () {  
            return "gizmo " + secret.id;  
        }  
    };  
}
```

```
function hoozit(id) {  
    var secret = {};    /*final*/  
    var that = gizmo(id, secret);  
    that.test = function (testid) {  
        return testid === secret.id;  
    };  
    return that;  
}
```

# Super Methods

```
function hoozit(id) {  
    var secret = {};  
    var that = gizmo(id, secret);  
    var super_toString = that.toString;  
    that.test = function (testid) {  
        return testid === secret.id;  
    };  
    that.toString = function () {  
        return super_toString.apply(that);  
    };  
    return that;  
}
```

# Inheritance Patterns

- Prototypal Inheritance works really well with **public** methods.
- Functional Inheritance works really well with **privileged** and **private** and **public** methods.
- Pseudoclassical Inheritance for elderly programmers who are old and set in their ways.

# Thank you!

