JavaScript Master Seminar
Module Pattern

Sirma Gjorgievska
Johannes Fischer

Technische Universität München

September 07, 2015

Agenda

- Introduction
- The Basics
- Augmentation
- Shared Private State
- Submodules
- Inheritance
- Object literal
- AMD modules
- CommonJS modules
- Harmony modules
- Demonstration
- Conclusion

Introduction

What is Module?

- Integral piece of robust application's architecture
- Keeps the units of code separated and organized

Introduction

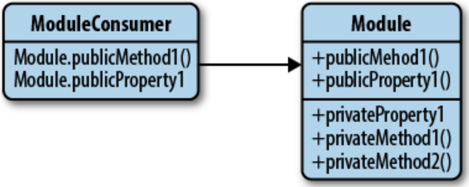
Implementation of modules

- The Module pattern
- Object literal notation
- AMD modules
- CommonJS modules
- ECMAScript Harmony modules

Introduction

What is Module pattern?

- JavaScript design pattern
- Developed in 2003
- Private and public encapsulation
- Mimic classes in software engineering



ModuleConsumer

Module.publicMethod1()
Module.publicProperty1

Module

+publicMehod1()
+publicProperty1()

+privateProperty1
+privateMethod1()
+privateMethod2()

Introduction

Advantages

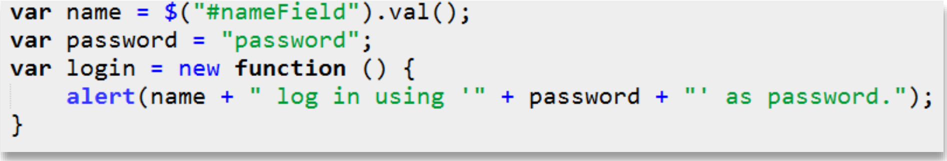
- Cleaner approach for developers
- Supports private data
- Less clutter in global namespace
- Localization of functions and variables

Introduction

Disadvantages

- Inability to create automated unit tests
- Loss of extensibility (sometimes)
- Problems when changing visibility of public/private members

The Basics



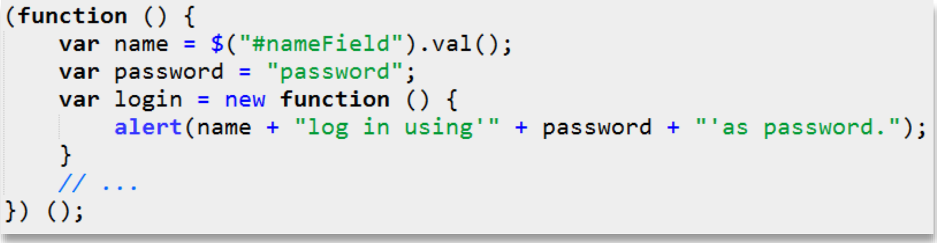
```
var name = $("#nameField").val();
var password = "password";
var login = new function () {
    alert(name + " log in using '" + password + "' as password.");
}
```

Simple code without patterns

The Basics

Anonymous Closures

- Defined function is executed immediately
- Code inside the function lives in a **closure**
- It provides **private state**
- Maintains access to all globals

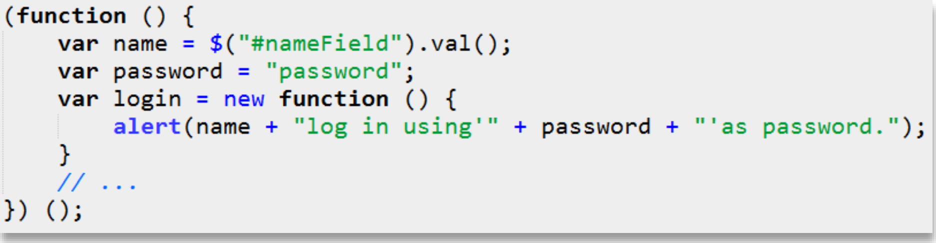


```
(function () {
    var name = $("#nameField").val();
    var password = "password";
    var login = new function () {
        alert(name + "log in using'" + password + "'as password.");
    }
    // ...
}) ();
```

The Basics

Private methods

- Methods locally declared in modules
- Inaccessible outside of the scope defined



```
(function () {
    var name = $("#nameField").val();
    var password = "password";
    var login = new function () {
        alert(name + "log in using'" + password + "'as password.");
    }
    // ...
}) ();
```

The Basics

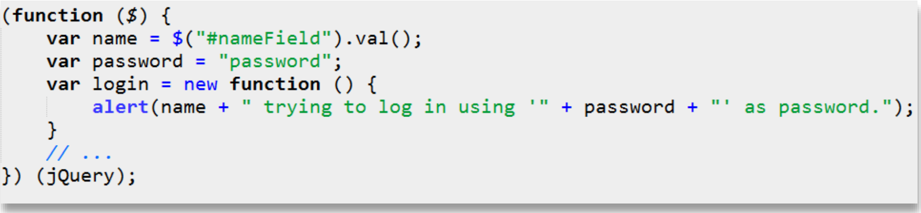
Implied Globals

- Hard-to-manage code
- Not obvious (to humans) which variables are global

The Basics

Global Import

- Better alternative
- Passing globals as parameters to anonymous function
- Better efficiency and readability



```
(function ($) {
    var name = $("#nameField").val();
    var password = "password";
    var login = new function () {
        alert(name + " trying to log in using '" + password + "' as password.");
    }
    // ...
}) (jQuery);
```

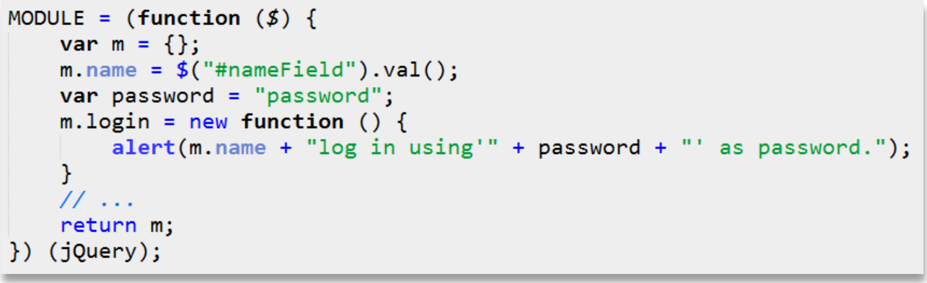
The Basics

Module Export

- Declare globals for further use
- Return value of anonymous function
- Module variables readable afterwards
- Namespacing (avoids varname conflicts)

The Basics

Module Export



```
MODULE = (function ($) {
    var m = {};
    m.name = $("#nameField").val();
    var password = "password";
    m.login = new function () {
        alert(m.name + "log in using'" + password + "' as password.");
    }
    // ...
    return m;
}) (jQuery);
```

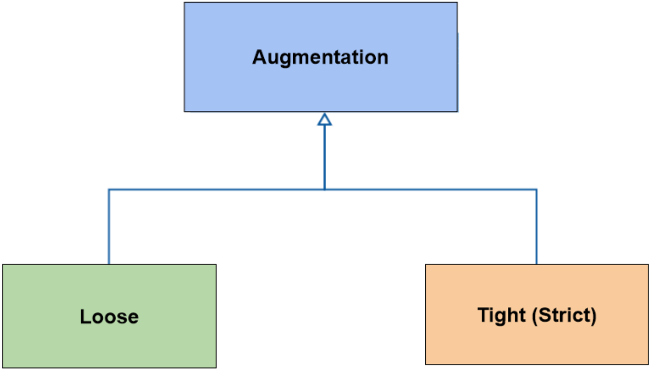
Augmentation

Problems:

- Entire module must be in one file
- Not extendable

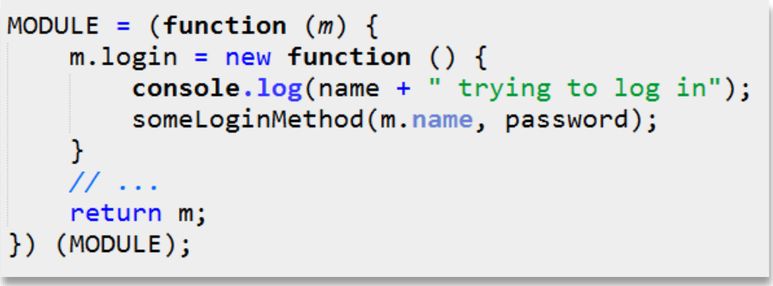
Solution: **Augment modules**

Augmentation



Augmentation

Tight (Strict) Augmentation



```
MODULE = (function (m) {
    m.login = new function () {
        console.log(name + " trying to log in");
        someLoginMethod(m.name, password);
    }
    // ...
    return m;
}) (MODULE);
```

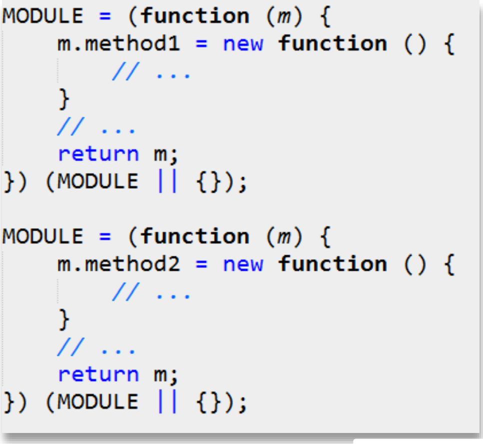
Augmentation

Tight (Strict) Augmentation

- Parameter: *MODULE*
- Loading order has to be fixed
- Properties of earlier modules usable reliably
- Properties overwritable

Augmentation

Loose Augmentation



```
MODULE = (function (m) {
    m.method1 = new function () {
        // ...
    }
    // ...
    return m;
}) (MODULE || {});

MODULE = (function (m) {
    m.method2 = new function () {
        // ...
    }
    // ...
    return m;
}) (MODULE || {});
```

Augmentation

Loose Augmentation

- Parameter: *MODULE || {}*
- Loading order irrelevant
- Module files can be loaded in parallel

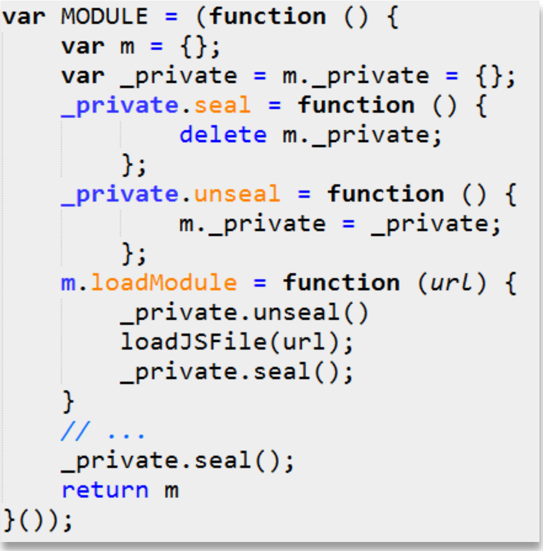
Augmentation

<table>
<tr><th>Tight Augmentation</th><th>vs</th><th>Loose Augmentation</th></tr>
<tr><td>- Allows overrides</td><td></td><td>- Cannot override safely</td></tr>
<tr><td>- Loading order fixed</td><td></td><td>- Loading order not fixed</td></tr>
<tr><td>- Parameter: <i>MODULE</i></td><td></td><td>- Parameter: <i>MODULE | | {}</i></td></tr>
</table>

Disadvantage

- Inability to share private variables between files

Shared Private State



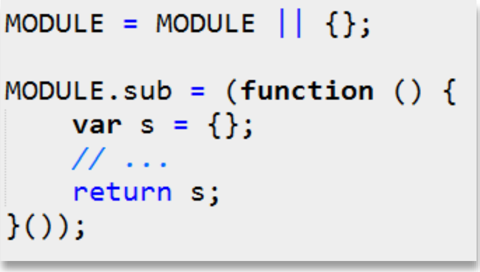
```
var MODULE = (function () {
    var m = {};
    var _private = m._private = {};
    _private.seal = function () {
            delete m._private;
        };
    _private.unseal = function () {
            m._private = _private;
        };
    m.loadModule = function (url) {
        _private.unseal()
        loadJSFile(url);
        _private.seal();
    }
    // ...
    _private.seal();
    return m
}());
```

Shared Private State

- Variable _*private* identical for all modules
- Only accessible from old module files
- Unlocks _*private* for later module file loading

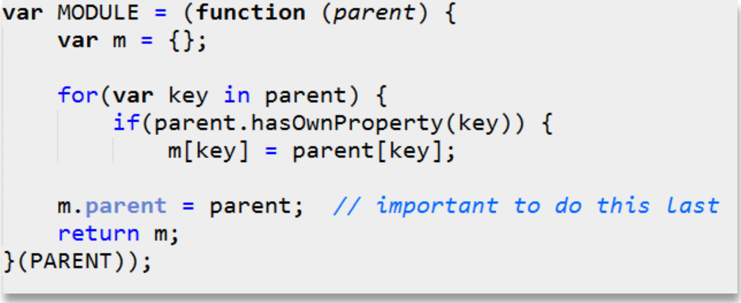
Submodules

- Creation is same like regular modules
- All the advanced capabilities of normal modules



```
MODULE = MODULE || {};

MODULE.sub = (function () {
    var s = {};
    // ...
    return s;
}());
```

Inheritance

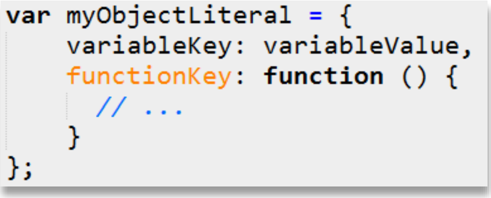


```
var MODULE = (function (parent) {
    var m = {};

    for(var key in parent) {
        if(parent.hasOwnProperty(key)) {
            m[key] = parent[key];

    m.parent = parent;  // important to do this last
    return m;
}(PARENT));
```

- Parent module has to exist
- New module now has parent and parent's properties
- *m.parent* needs to be set last

Object Literal

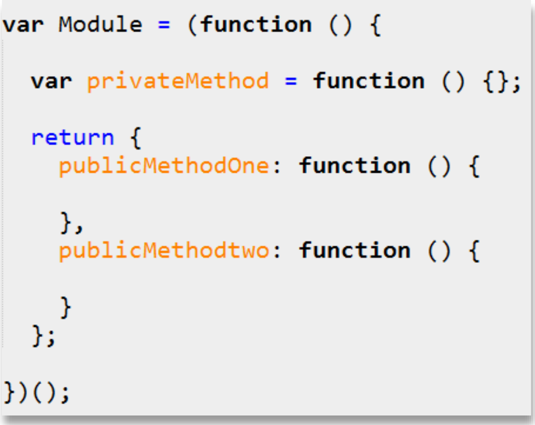
- Comma-separated list of name-value pairs wrapped in curly braces
- Encapsulate data, enclosing it in a tidy package



```
var myObjectLiteral = {
    variableKey: variableValue,
    functionKey: function () {
      // ...
    }
};
```

Object Literal

- Used in Module pattern as the return value from a scoping function

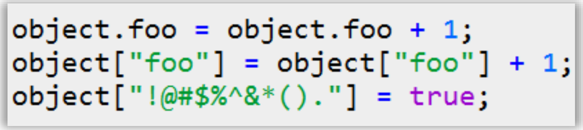


```
var Module = (function () {

  var privateMethod = function () {};

  return {
    publicMethodOne: function () {

    },
    publicMethodtwo: function () {

    }
  };

})();
```

Object Literal

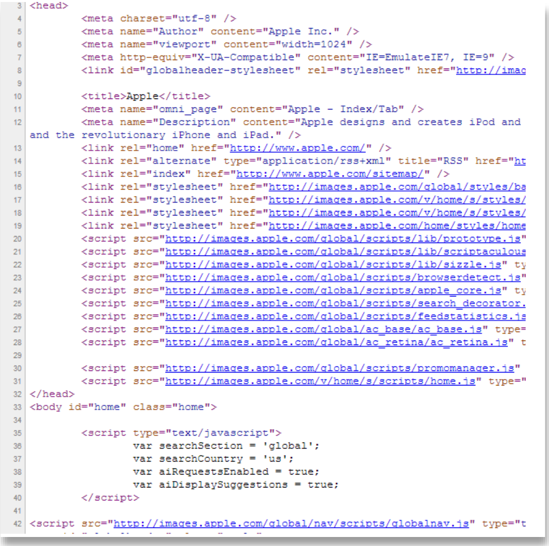
Accessing properties

- Dot notation
- Bracket notation
  - allows a variable to specify all or part of the property name
  - allows property names to contain characters forbidden in dot notation



```
object.foo = object.foo + 1;
object["foo"] = object["foo"] + 1;
object["!@#$%^&*()."] = true;
```

Script hell

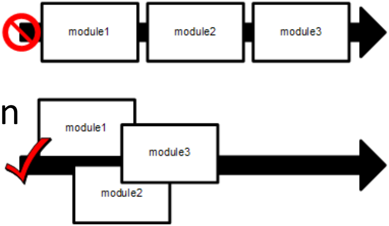


```
<head>
        <meta charset="utf-8" />
        <meta name="Author" content="Apple Inc." />
        <meta name="viewport" content="width=1024" />
        <meta http-equiv="X-UA-Compatible" content="IE=EmulateIE7, IE=9" />
        <link id="globalheader-stylesheet" rel="stylesheet" href="http://imag

        <title>Apple</title>
        <meta name="omni_page" content="Apple - Index/Tab" />
        <meta name="Description" content="Apple designs and creates iPod and
and the revolutionary iPhone and iPad." />
        <link rel="home" href="http://www.apple.com/" />
        <link rel="alternate" type="application/rss+xml" title="RSS" href="ht
        <link rel="index" href="http://www.apple.com/sitemap/" />
        <link rel="stylesheet" href="http://images.apple.com/global/styles/ba
        <link rel="stylesheet" href="http://images.apple.com/v/home/s/styles/
        <link rel="stylesheet" href="http://images.apple.com/v/home/s/styles/
        <link rel="stylesheet" href="http://images.apple.com/home/styles/home
        <script src="http://images.apple.com/global/scripts/lib/prototype.js"
        <script src="http://images.apple.com/global/scripts/lib/scriptaculous
        <script src="http://images.apple.com/global/scripts/lib/sizzle.js" ty
        <script src="http://images.apple.com/global/scripts/browserdetect.js"
        <script src="http://images.apple.com/global/scripts/apple_core.js" ty
        <script src="http://images.apple.com/global/scripts/search_decorator.
        <script src="http://images.apple.com/global/scripts/feedstatistics.js
        <script src="http://images.apple.com/global/ac_base/ac_base.js" type=
        <script src="http://images.apple.com/global/ac_retina/ac_retina.js" t

        <script src="http://images.apple.com/global/scripts/promomanager.js"
        <script src="http://images.apple.com/v/home/s/scripts/home.js" type="
</head>
<body id="home" class="home">

        <script type="text/javascript">
                var searchSection = 'global';
                var searchCountry = 'us';
                var aiRequestsEnabled = true;
                var aiDisplaySuggestions = true;
        </script>

<script src="http://images.apple.com/global/nav/scripts/globalnav.js" type="t
```

AMD modules

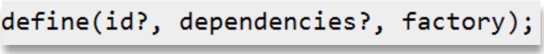
Asynchronous Module Definition

- Specifies a mechanism for defining modules
- Module and dependencies can be asynchronously loaded
- Many advantages:
  - Dependencies are easy to identify
  - Avoids global variables



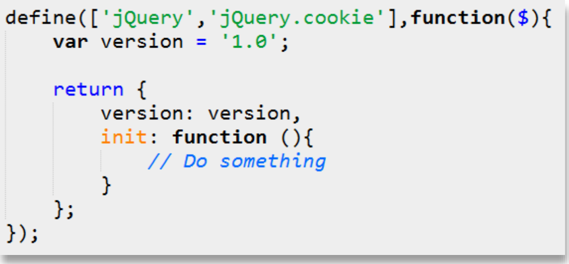
AMD modules

- API consists of a single function:



```
define(id?, dependencies?, factory);
```

- Sample module that has dependencies on jQuery and jQuery cookie plugin:



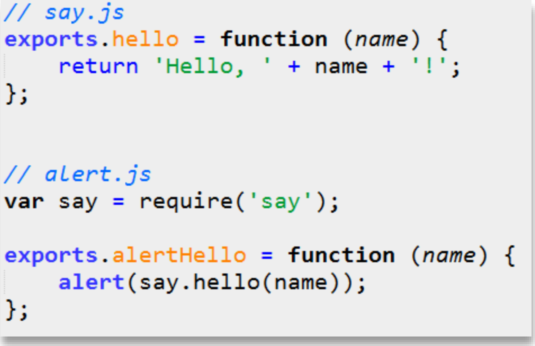
```
define(['jQuery','jQuery.cookie'],function($){
    var version = '1.0';

    return {
        version: version,
        init: function (){
            // Do something
        }
    };
});
```

CommonJS modules

- Module proposal that specifies a simple API for declaring modules
- Better suited for server-side environments than for browsers
- Most popular implementation is in Node.js
- Performance suffers when browsers must load a lot of modules

CommonJS modules

- Standardize the following scoped variables:
  - ***require***
  - ***exports***
  - ***module***



```
// say.js
exports.hello = function (name) {
    return 'Hello, ' + name + '!';
};


// alert.js
var say = require('say');

exports.alertHello = function (name) {
    alert(say.hello(name));
};
```

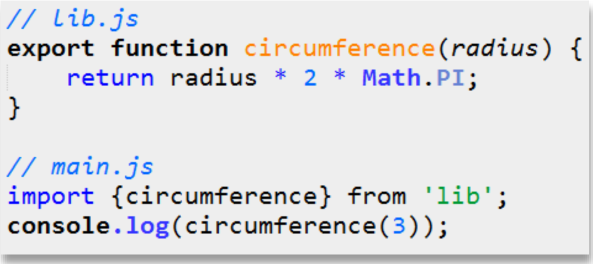
ES6 Harmony modules

- Format good for users of CommonJS and AMD
- Modules have a compact syntax
- Modules have direct support for asynchronous loading

ES6 Harmony modules

- Two kinds of exports:
  - Named exports (several per module)
  - Default exports (one per module)

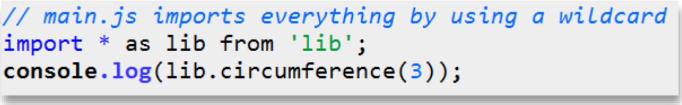
- **Named export**



```
// lib.js
export function circumference(radius) {
    return radius * 2 * Math.PI;
}

// main.js
import {circumference} from 'lib';
console.log(circumference(3));
```

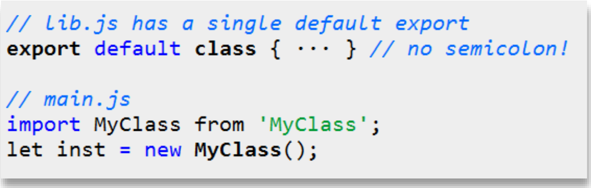
ES6 Harmony modules

- Another alternative is to import the complete module



```
// main.js imports everything by using a wildcard
import * as lib from 'lib';
console.log(lib.circumference(3));
```

- **Default export**

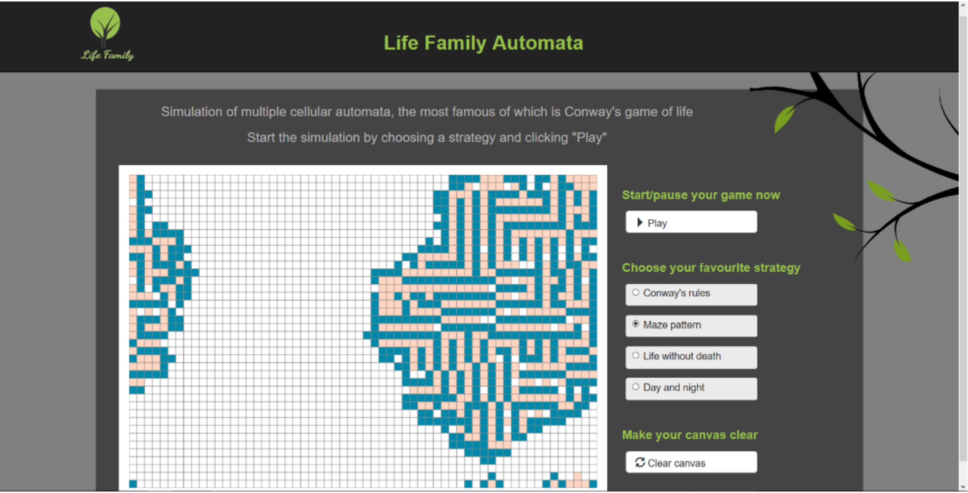


```
// lib.js has a single default export
export default class { ··· } // no semicolon!

// main.js
import MyClass from 'MyClass';
let inst = new MyClass();
```

Live Demonstration

- Simulation of cellular automata
- Implemented using JavaScript and Module Pattern
- A **cellular automaton** is a discrete model
- Consists of a regular grid of *cells*
- The most popular is **Conway's Game of Life**

Live Demonstration



Life Family Automata

Simulation of multiple cellular automata, the most famous of which is Conway's game of life

Start the simulation by choosing a strategy and clicking "Play"

Start/pause your game now

Choose your favourite strategy

○ Conway's rules

● Maze pattern

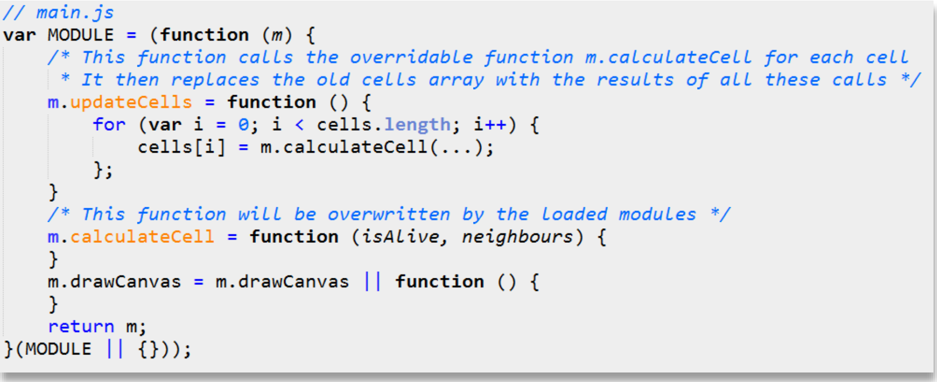
○ Life without death

○ Day and night

Make your canvas clear

Live Demonstration

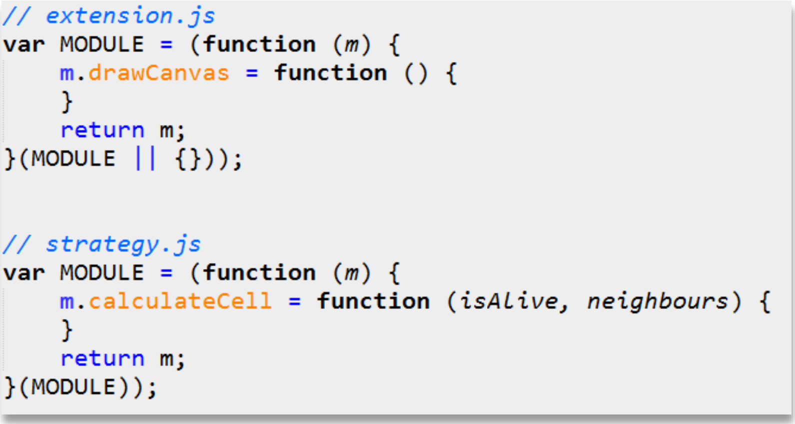
- How we used the module pattern in our application?



```
// main.js
var MODULE = (function (m) {
    /* This function calls the overridable function m.calculateCell for each cell
     * It then replaces the old cells array with the results of all these calls */
    m.updateCells = function () {
        for (var i = 0; i < cells.length; i++) {
            cells[i] = m.calculateCell(...);
        };
    }
    /* This function will be overwritten by the loaded modules */
    m.calculateCell = function (isAlive, neighbours) {
    }
    m.drawCanvas = m.drawCanvas || function () {
    }
    return m;
}(MODULE || {}));
```

Live Demonstration

- How we used the module pattern in our application?



```
// extension.js
var MODULE = (function (m) {
    m.drawCanvas = function () {
    }
    return m;
}(MODULE || {}));


// strategy.js
var MODULE = (function (m) {
    m.calculateCell = function (isAlive, neighbours) {
    }
    return m;
}(MODULE));
```

Conclusion

- Module pattern is very common JavaScript pattern
- Faster download (parallel)
- No performance penalty
- Shorter reference chains might increase performance
- **Loose augmentation** allows easy non-blocking parallel download

Thank you!