JavaScript Master Seminar
Module Pattern

Sirma Gjorgievska
Johannes Fischer

Technische Universität München

September 07, 2015

Fakultät für Informatik

Agenda

- Introduction
- The Basics
- Augmentation
- Shared Private State
- Submodules
- Inheritance
- Demonstration
- Conclusion

Introduction

What is Module?

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

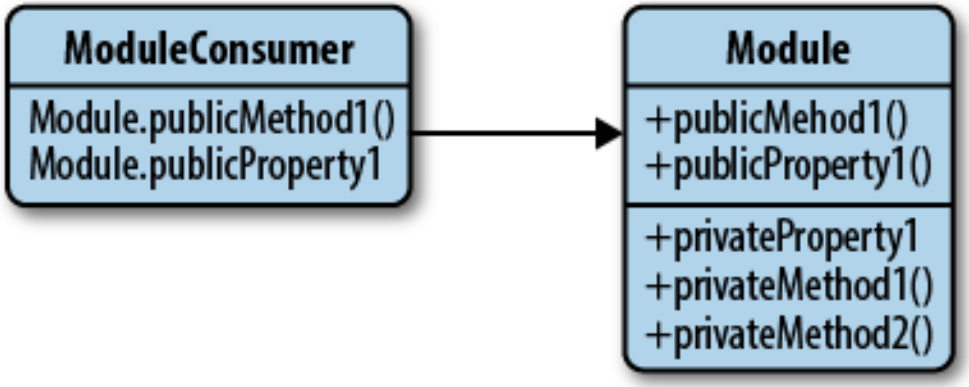
Implementation of modules

- 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



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
- Lose of extendibility
- 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

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.");
    }
    // ...
}) ();
```

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.");
    }
    // ...
}) ();
```

Implied Globals

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

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);
```

Module Export

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

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);
```

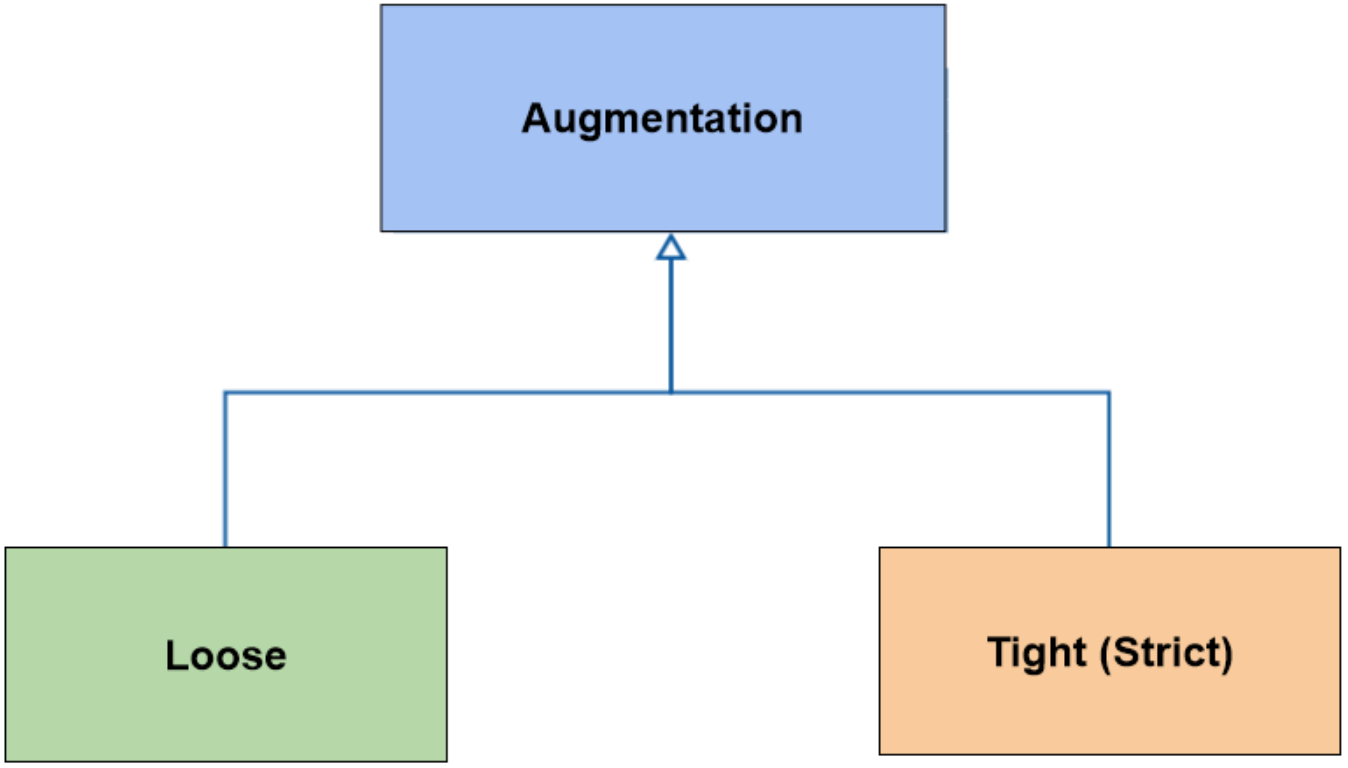
Augmentation

Problems:

- Entire module must be in one file
- Not extendable

Solution: **Augment modules**

Augmentation



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

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

Tight Augmentation **vs** **Loose Augmentation**

- Allows overrides
- Loading order fixed
- Parameter: *MODULE*

- Cannot override safely
- Loading order not fixed
- Parameter: *MODULE || {}*

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;
}());
```

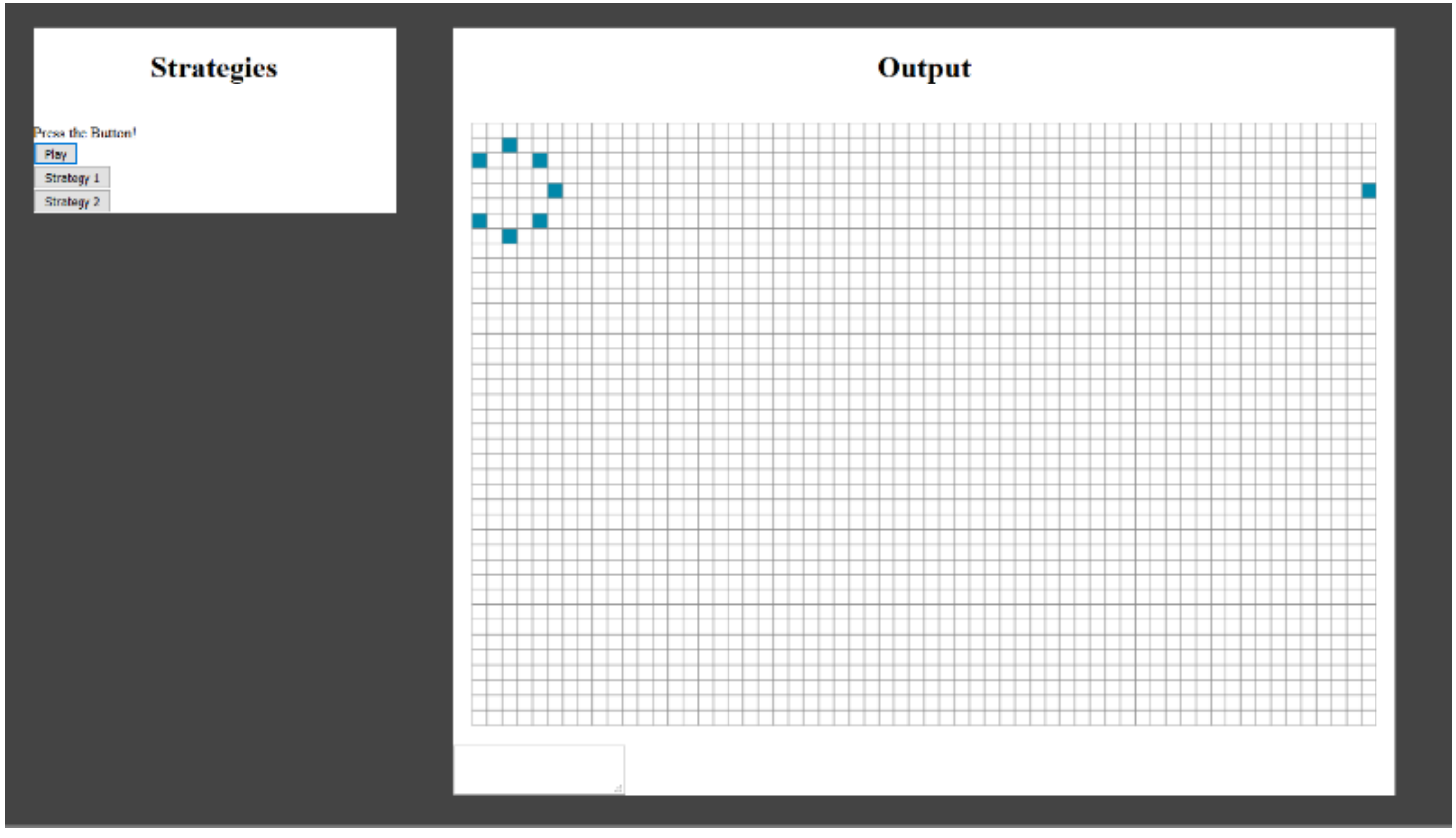
Inheritance

```
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

Live Demonstration

- JavaScript implementation of John Conway's **Game of Life**



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!