Introduction to JavaScript

Minko Gechev

- twitter.com/mgechev
- github.com/mgechev
- blog.mgechev.com

- Data types
- Variable scope
- Inheritance
- Immediately-Invoked Function Expression (IIFE)
- Exception handling
- Object-oriented principles
- Design patterns
- Namespace pattern
- Modules
- Publish/subscribe
- Sample architecture

Data types

- Primitives - basic building blocks, they can't have methods or properties
- Objects - may have properties which could be different objects or primitives

Primitives

- Number - floating point numbers (based on IEEE 754)
- String - immutable, for each action new string is created.
  [] operator is pre-defined for them but is read-only.
- Boolean - literals true and false
- Undefined - contains a single value - undefined. All variables without value are initialized to undefined
- *Null*

typeof operator

typeof is operator, not a function (the parentheses in typeof(1) are the same as -(1)). It has single operand and its application results a string.

```
typeof 1 //number
typeof 'foo' //string
typeof true //boolean
typeof undefined //undefined
typeof null //object
```

Bad

```
function foo(bar) {
  if (typeof bar === 'object') {
    bar.baz(); //type error when null is used
  }
}
```

Better

```
function foo(bar) {
  if (bar && typeof bar.baz === 'function') {
    bar.baz();
  }
}
```

Check if variable exists

```
if (typeof variable !== 'undefined') {
  //do something
}
```

if is already defined:

```
if (variable !== undefined) //undefined is mutable in older browse
```

Number to String

```
  var num = parseInt('42');
```

...but in older versions...

```
  var num = parseInt('08');
  console.log(num); //0
```

```
  var num = parseInt('08', 10);
```

Do not forget the second parameter!

NaN

What happens when given string cannot be converted to number?

```
var number = parseInt('not a number', 10);
number == number //false
number === number //false
number !== number //true
typeof number //'number'
isNaN(number); //true
```

Numbers

Be aware of...

```
0.1 + 0.2 //0.30000000000000004
Math.pow(2, 53) === Math.pow(2, 53) + 1 //true
0 == ''
```

Booleans and boolean operators

The following values are evaluated to false:

- false
- undefined
- NaN
- null
- 0
- ''

Everything else is evaluated to true.

Usage of boolean operators

Often we want to make sure given value is boolean. Short way to do this is:

```
    var boolValue = !!notBoolValue;
```

Short way for:

```
  if (!value) {
      value = 1;
  }
```

```
  value = value || 1;
```

Objects

Objects are also used for hash maps. Something specific for them is that their keys can be only strings!

```
var obj = {},
    obj2 = { foo: 'foo' },
    obj3 = { bar: 'bar' };
obj[obj2] = obj2;
obj[obj3] = obj3;
console.log(obj[obj2]); //{ bar: 'bar' }
console.log(Object.keys(obj)); //["[object Object]"]
```

Objects

```
var obj = {};
```

If we want to add properties:

```
var obj = { foo: 'bar' };
```

...or methods...

```
var obj = {
  foo: function () {
    //body
  }
}
```

Interating over objects

```
for (var key in value) {
  //do something
}
```

Later we're going to look at **Object.key**.

Arrays

```
var arr1 = [];
var arr2 = new Array(size);
```

It is better to use the first syntax, it is faster and not that verbose.

Arrays

In ES5 there were included few very useful methods:

- forEach
- filter
- map
- every
- some
- indexOf
- lastIndexOf
- reduce
- reduceRight

Example - reduce

Sum of the array elements:

```
[0,1,2,3].reduce(function (p, c) {
  return p + c;
}/*, optional initial value */);
```

```
[0,1,2,3].reduceRight(function (p, c) {
  return p + c;
}/* optional initial value */);
```

Functions

Since functions are not primitives they can have properties. Typical example for this is $ of jQuery.

```
function bar() {
  //do some awesome stuff!
}
bar.foo = 'baz';
```

Memorization

```
function veryHeavyFunction(arg) {
  var res = veryHeavyFunction[arg];
  if (res !== undefined) return res;
  //heavy computations here...
}
```

```
  var veryHeavyFunction = (function () {
    var cache = {};
    return function (arg) {
      if (cache[arg]) return cache[arg];
      //heavy computations
    };
  }());
```

Functions

By default each function has local variable called **arguments**. It is something like an array but not exactly:

```
function foo() {
  var argsArray = Array.prototype.slice.call(arguments);
  console.log(typeof arguments);
  console.log(typeof arguments.length);
  console.log(arguments instanceof Array);
}
foo(); //'object', 0, false
```

Functions

Functions can be passed as arguments to other functions. This is very useful when we have asynchronous code (xhr, read file, socksts, events).

```
function animate(duration, callback) {
  if (!duration) {
    callback();
    return;
  }
  setTimeout(function () {
    animate(duration - TIMEOUT / 1000, callback);
  }, TIMEOUT);
}

animate(2, function () {
  alert('Alert in 2 seconds');
});
```

Comparison

```
var o1 = new Object(),
    o2 = new Object();
o1 === o2 //false
o1 == o2 //false
o1 < o2 //false
o1 > o2 //false
o1 <= o2 //true
o1 >= o2 //true
```

This happens because < and > cast their operands to strings implicity.

Comparison

```
var str = 'baz';
str instanceof String //false
typeof str === 'string' //true
var str2 = new String('baz');
str == str2 //true
str === str2 //false
str === str2.toString();
```

This is result of comparison of reference and primitive types. It is better to use primitive strings.

Switch

Switch uses "===".

```
var a = new String('a');
switch (a) {
  case 'a': alert('It\'s a!'); break;
  default: alert('Not a?');
}
```

Local variables (1)

Local variables are declared using the keyword **var**.

All variables declared in the global namespace (i.e. not declared in any function or with omitted **var**) become properties of window.

```
var foo = 'foo';
console.log(window.foo);
console.log(window['foo']);
function localScope() {
  var bar = 'bar';
  console.log(foo, window['foo']); //foo
}
console.log(bar); //undefined
```

Local variables (2)

Important fact is that the variables have functional scope - not block scope!

Same applies for other flow control operators (if, do while, while, switch).

```
function foo() {
  for (var i = 0; i < 3; i += 1) {}
  console.log(i); //3
}
foo();
console.log(i); //undefined
```

Local variables (3)

Do not omit the var keyword, because the variable will be decalred as global.

```
function foo() {
  for (i = 0; i < 3; i += 1) {}
  console.log(i); //3
}
foo();
console.log(i); //3
```

Local variables (4)

ES6 gives us **let**. let allows us to declare local variables to given block (similar to languages like Java, C#, C++).

Currently is not supported by any of the major browsers.

```
function foo() {
  for (let i = 0; i < 3; i += 1) {
    //something here
  }
  console.log(i); //undefined
}
```

Global variables

```
foo = new Object();
window['foo'] === foo //true
window.foo === foo //true
function bar() {
  baz = 'baz';
}
bar();
window.baz //'baz'
```

Instance variables

Attached to the current context

```
    window === this //true
```

Context (1)

The context is the **this** reference, which refers to specific object.

If we call method of given nested object the context of the method will be the object just before the last ".".

Using different techniques we can change the context of the functions.

Context (2)

```
var obj = {
  bar: {
    baz: function () { return this; }
  },
  foo: function () { return this }
};

function foobar() { return this; }

foobar() === window
obj.foo() === obj
obj.bar.baz() === ?
```

Changing the context (1)

- Using new
- Using call/apply
- Using bind

Changing the context (2)

When new is called the function's context is newly created object. It will be result of the function called with new. You can attach different properties and methods to "this".

```
var temp;
function foo() {
  console.log(this === window); //false
  temp = this;
}
new foo === temp;
```

Changing the context (3)

Using the function's methods **call** and **apply** you can call a function with specific context:

```
function baz() {
  console.log(this, arguments);
}
baz.call({ foo: 'baz' }, 1,2,3);
baz.apply({ foo: 'baz' }, [1,2,3]);
```

Changing the context (4)

Using the functions' method **bind** you can change the context of given function. When bind is called with speicified context new function is returned. The context of the newly created function cannot be changed.

```
function foo() {
  console.log(this);
}
foo.bind({});
```

Schönfinkeling (Currying)

Functional transformation. It is commonly used in languages like Haskell.

It allows to call function accepting multiple arguments with just few of them. The result of the partial application is new function which accepts the rest arguments (ones which are still not filled).

```
function foo(a, b) {
  return a + b;
}
var res = foo.bind(null, 1);
res(2); //3
```

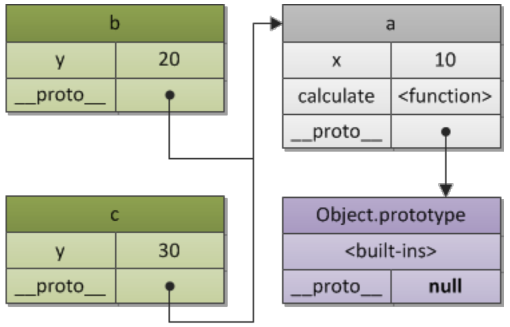
Inheritance

Two approaches:

- (Pseudo) Classical
- Prototypal

Inheritance

In both approaches we create a prototype chain:



Classical

```
function Person(name) {
    this._name = name;
}
Person.prototype.getName = function () {
    return this._name;
};

function Developer(name, languages) {
    Person.call(this, name);
    this.languages = languages;
}
Developer.prototype = new Person();

var dev = new Developer('foo', ['JavaScript', 'Perl', 'Java']);
console.log(dev.getName());     //"foo"
console.log(dev._name);         //"foo"
console.log(dev.languages[0]);  //"JavaScript"
```

Classical (+/-)

- Looks familiar

- No privacy (weak encapsulation)
- Unnecessary complexity

Functional/Closure inheritance

```
function Person(name) {
    var _name = name;
    this.getName = function () {
        return _name;
    };
}
function Developer(name, languages) {
    var _languages = languages;
    Person.call(this, name);
    this.getLanguages = function () {
        return _languages;
    };
}
Developer.prototype = new Person();
var dev = new Developer('foo', ['JavaScript', 'Perl', 'Java']);
console.log(dev._name);     //undefined
console.log(dev.getName()); //"foo"
```

Functional (+/-)

- Looks fairly familiar
- Real privacy

- Multiple copies of the methods
- Complex

Prototypal (1)

```
var person = {
    name: 'foo',
    age: 42
};
var dev = Object.create(person);
dev.languages = ['JavaScript', 'Perl', 'Java'];

console.log(dev.name);         //"foo"
console.log(dev.languages[0]); //"JavaScript"
```

Object.create accept object as first argument and create new object with prototype its first paramter.

Prototypal (+/-)

- Simplicity

- Looks strange
- No privacy (*)

Useful ES5 methods (1)

- **Object.create** - create object with prototype its first argument
- **Object.getPrototypeOf** - get the prototype of given object
- **Object.keys** - get all enumerable keys of given object
- **Object.getOwnPropertyNames** - get all enumerable or not keys of given object
- **Object.defineProperty(obj, prop, descriptor)** - defines property by setting its

Object.defineProperty example

```
Object.defineProperty(obj, "key", {
  enumerable: false,
  configurable: false,
  writable: false,
  value: 'static'
});
```

Object.defineProperty

- **configurable** - true if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object. Defaults to false.
- **enumerable** - true if and only if this property shows up during enumeration of the properties on the corresponding object. Defaults to false.

Object.defineProperty

- **value** - The value associated with the property. Can be any valid JavaScript value (number, object, function, etc) Defaults to undefined.
- **writable** - True if and only if the value associated with the property may be changed with an assignment operator. Defaults to false.

Useful ES5 methods (2)

- **Object.seal** - Prevents the object from adding, deleting properties
- **Object.freeze** - Same as seal but also prevents from changing properties' values
- **Object.preventExtensions**
- **Object.isSealed**
- **Object.isFrozen**
- **Object.isExtensible**
- **Object.getOwnPropertyDescriptor**

Strict mode

ES5 introduce "use strict";. Use strict save us from some common mistakes. According to MDN:

Eliminates errors which makes the code difficult to be optimize thus makes the code faster

Strict mode

Strict mode should be per function. It may create problems by concatenating strict and non-strict files.

Strict mode

```
"use strict";
mistypedVaraible = 17; // throws a ReferenceError
```

```
"use strict";

var obj1 = {};
Object.defineProperty(obj1, "x", { value: 42, writable: false });
obj1.x = 9; // throws a TypeError
```

```
"use strict";
delete Object.prototype; // throws a TypeError
```

Strict mode

```
"use strict";
var o = { p: 1, p: 2 }; // !!! syntax error

function sum(a, a, c){ // !!! syntax error
  "use strict";
  return a + b + c; // wrong if this code ran
}
```

```
"use strict";
var sum = 015 + // !!! syntax error
          197 +
          142;
```

Polymorphism

```
function Person(name) {
    this._name = name;
}
Person.prototype.speak = function () {
    return 'My name is ' + this._name + '.';
};
function Developer(name, languages) {
    Person.call(this, name);
    this.languages = languages || [];
}
Developer.prototype = new Person();
Developer.prototype.speak = function () {
    return 'My name is ' + this._name + ' and I know ' + (this.l
};

var person = new Person('foo');
console.log(person.speak()); //"My name is foo and."
person = new Developer('foo', ['JavaScript', 'Perl', 'Java']);
console.log(person.speak()); //"My name is foo and I know JavaSc
```

Exception handling

For excaption handling JavaScript provides:

- throw
- try
- catch
- finally

Exception handling- throw

```
    throw expression;
```

throw throws an excaption which is the value from the evaluation of the expression. Each of the following throws an exception:

```
    throw 1;
    throw new Error('Error!');
    throw true;
    throw 'Error!';
```

Exception handling - Error

Error is a constructor function for creating new errors. As first argument it accepts an error message. The following constructor functions are extending it:

- EvalError - Creates an instance representing an error that occurs regarding the global function eval()
- RangeError - Creates an instance representing an error that occurs when a numeric variable or parameter is outside of its valid range
- ReferenceError - Creates an instance representing an error that occurs when de-referencing an invalid reference

- SyntaxError - Creates an instance representing a syntax error that occurs while parsing code in eval()
- TypeError - Creates an instance representing an error that occurs when a variable or parameter is not of a valid type
- URIError - Creates an instance representing an error that occurs when encodeURI() or decodeURI() are passed invalid parameters

try..catch...finally

All exception are handled the same way:

```
try {
  //statements
} catch (e) {
  //e is reference to the thrown error
} finally {
  //block executed on success and failture
  //it is the right place for removing event listeners, releasing reso
}
```

try...catch...finally

The possible combinations are valid:

- try...catch
- try...finally
- try...catch...finally

Immediately-Invoked Function Expression (IIFE)

Problem

- Initializing some stuff
- Not polluting the global namespace
- Doing the initialization only once

Sample solution (1)

```
function addHandlers() {
    //body
}
function performLayout() {
    //body
}
addHandlers();
performLayout();
```

- Multiple globals
- Can be called multiple times

Sample solutions (2)

```
//Variant 1
function init() {
    function addHandlers() {
    }
    addHandlers();
}
init();
```

- Polluting the global namespace (1 global)
- Can be called multiple times
- Ugly

Sample solution (3)

```
//Variant 2
var init = function () {
    function addHandlers() {
    }
    addHandlers();
};
init();
delete window.init;
```

- Ugly

Solution

```
(function (w, d) {
    //initialization
}(window, document));
```

- Fixing all issues
- Hard to understand

Sample usage

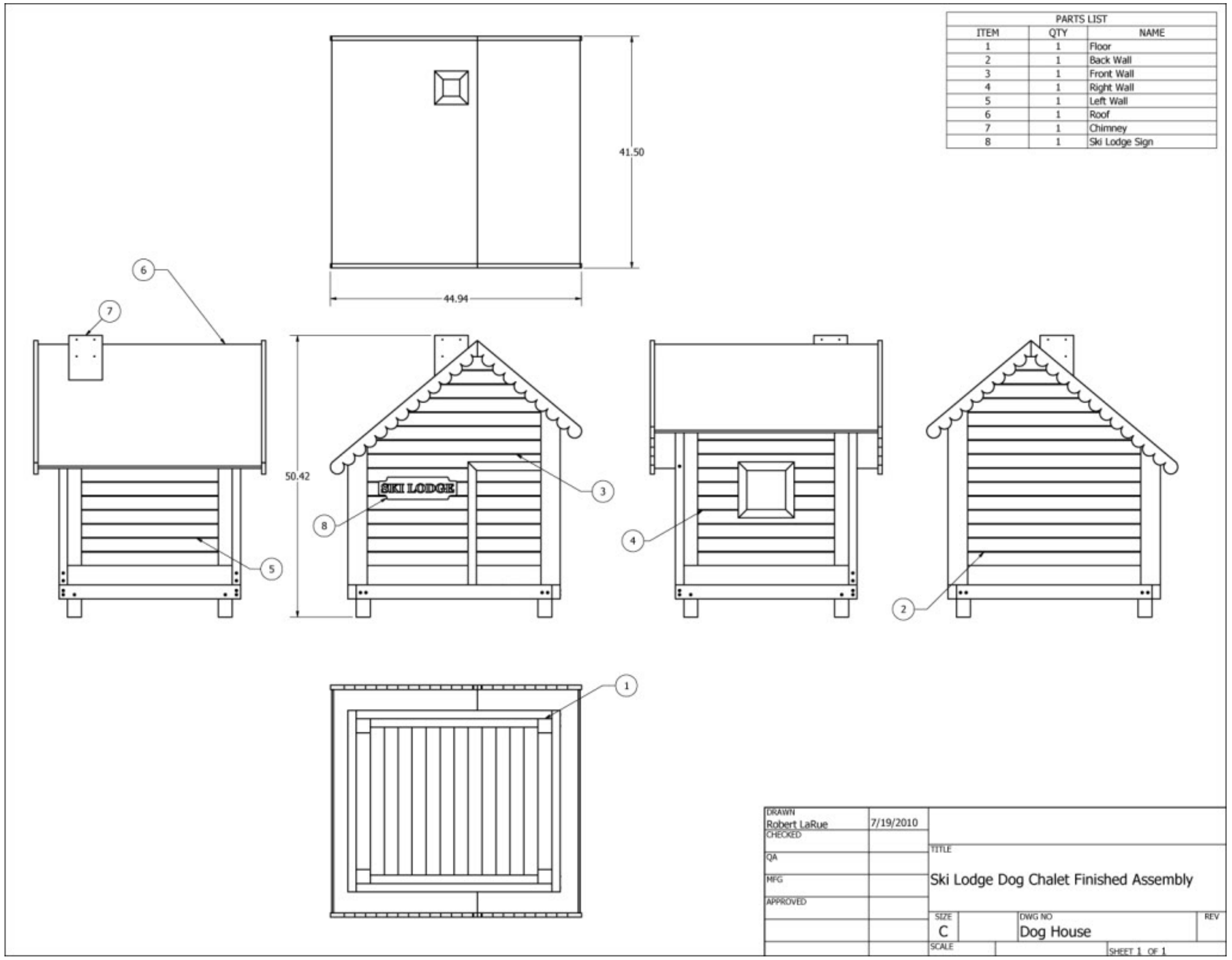
```
//Load facebook's SDK asynchronously
(function(d){
  var js, id = 'facebook-jssdk',
      refr = d.getElementsByTagName('script')[0];
  if (d.getElementById(id)) {return;}
  js = d.createElement('script'); js.id = id; js.async = true;
  js.src = "//connect.facebook.net/en_US/all.js#xfbml=1";
  refr.parentNode.insertBefore(js, refr);
}(document));
```

Object-oriented principles

Complexity

- Large projects
- Large complexity
- Strong coupling

Small project



PARTS LIST		
ITEM	QTY	NAME
1	1	Floor
2	1	Back Wall
3	1	Front Wall
4	1	Right Wall
5	1	Left Wall
6	1	Roof
7	1	Chimney
8	1	Ski Lodge Sign

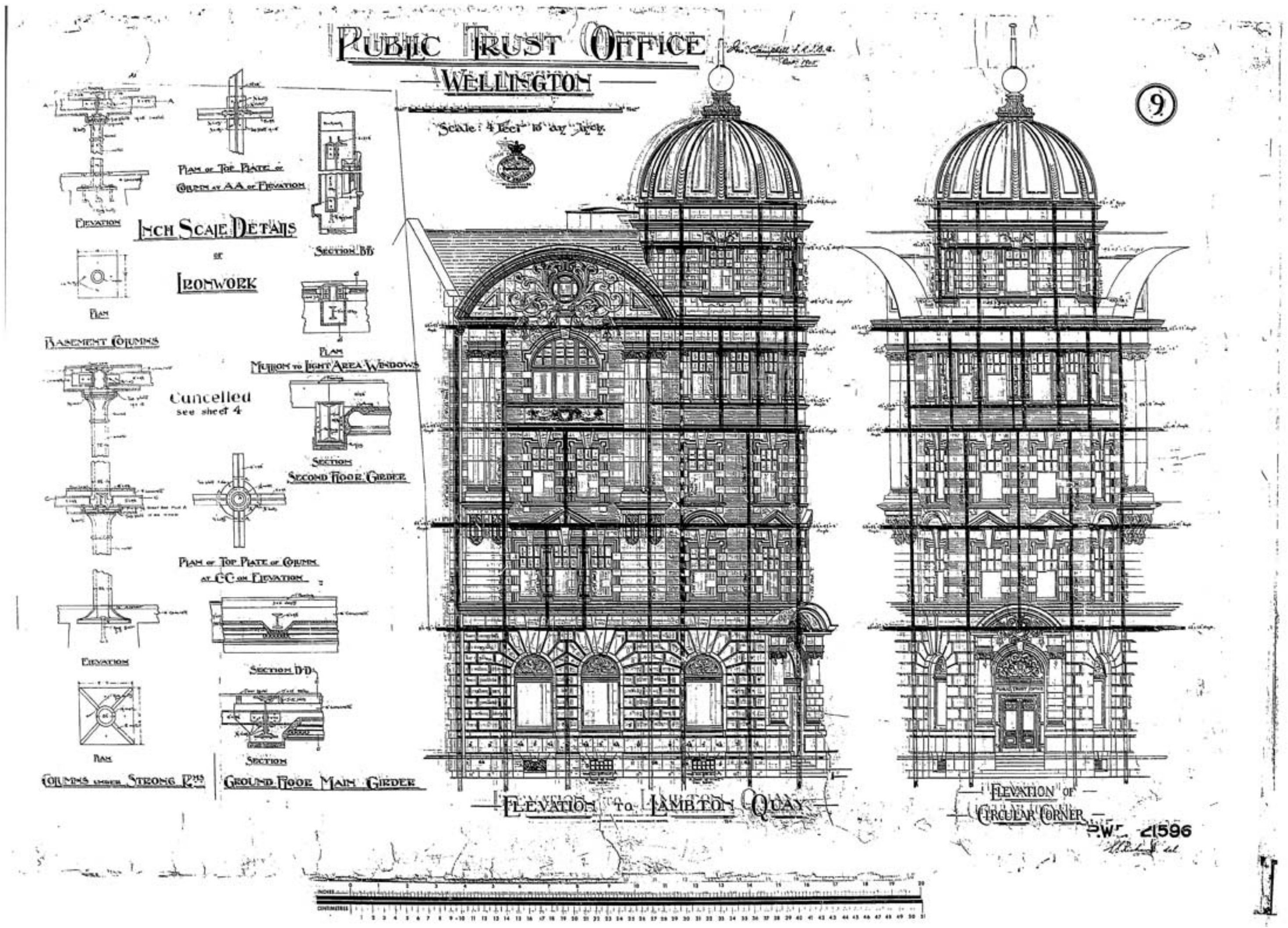
DRAWN Robert LaRue	7/19/2010
CHECKED	
QA	
MFG	
APPROVED	

TITLE
Ski Lodge Dog Chalet Finished Assembly

SIZE		DWG NO	REV
C		Dog House	

SCALE SHEET 1 OF 1

Real project



PUBLIC TRUST OFFICE
WELLINGTON
Scale: 4 Feet to an Inch.
INCH SCALE DETAILS of IRONWORK
Plan of Top Plate of Column at AA of Elevation
Elevation
Section BB
Plan
Basement Columns
Plan
Mullion to Light Area Windows
Cancelled see sheet 4
Section
Second Floor Girder
Plan of Top Plate of Column at CC on Elevation
Elevation
Section DD
Plan
Section
Columns under Strong Rms
Ground Floor Main Girder
ELEVATION to LAMBTON QUAY
ELEVATION of CIRCULAR CORNER
9
P.W.D. 21596

4 principles

- Inheritance
- Polymorphism
- Abstraction
- Encapsulation (Data-hiding)

Additional principles

- *GRASP*
- *SOLID*

Inheritance

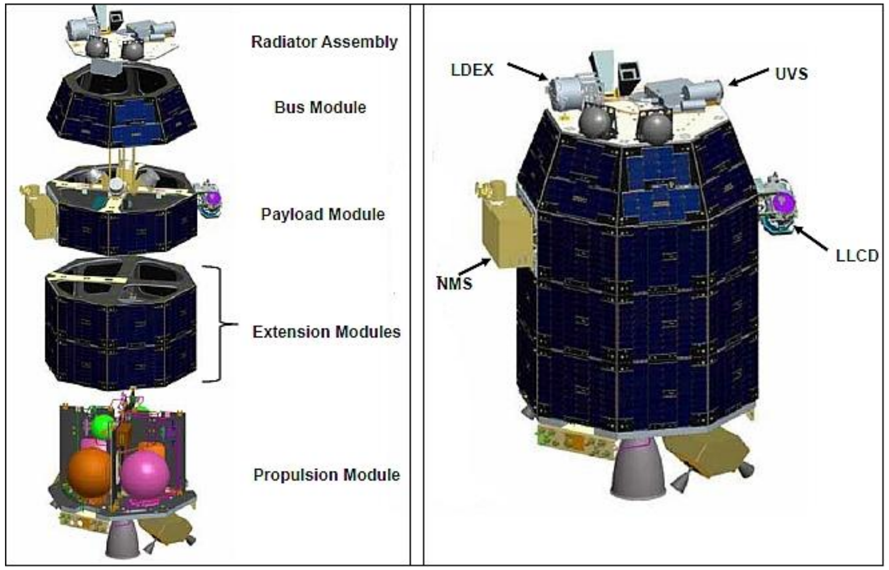
- Code reuse

Polymorphism

- Overloading

"Now Speak!"
"Quack"
"Woof"
"Meow"

Abstraction



Encapsulation

" Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behavior of an object... encapsulation focuses upon the implementation that gives rise to this behavior... "
Grady Booch

Design patterns

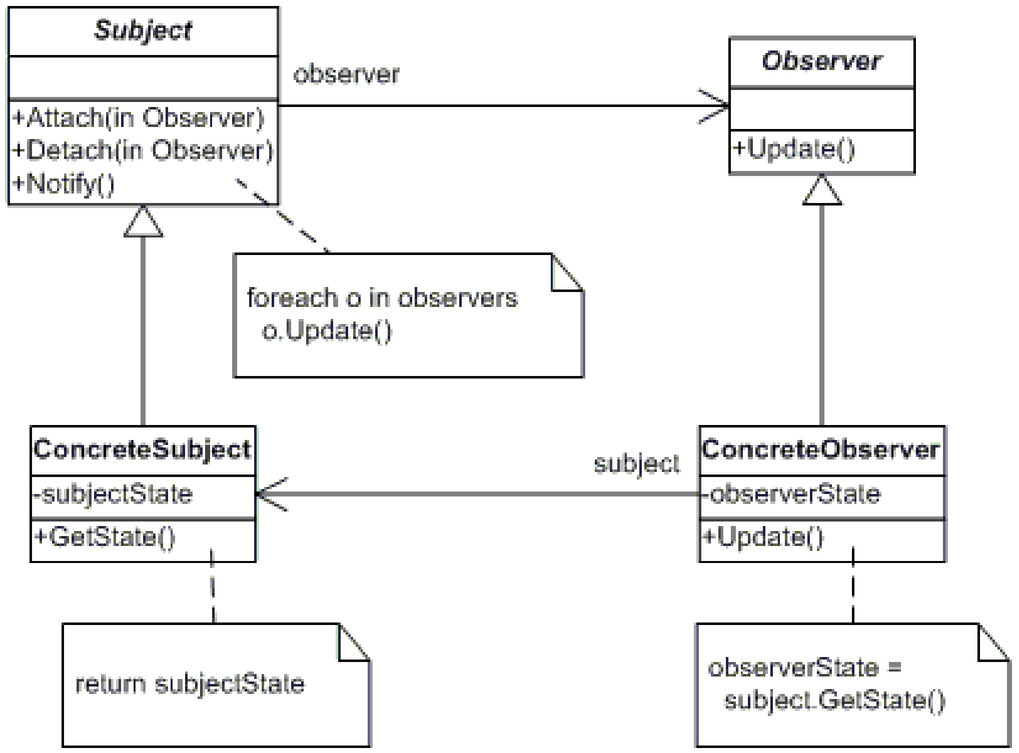
- Common problems
- Good approaches
- Balance

Observer

Problem

- Managing state-change notifications
- Minimum coupling
- Managing event data (pull/push)

Pull Observer

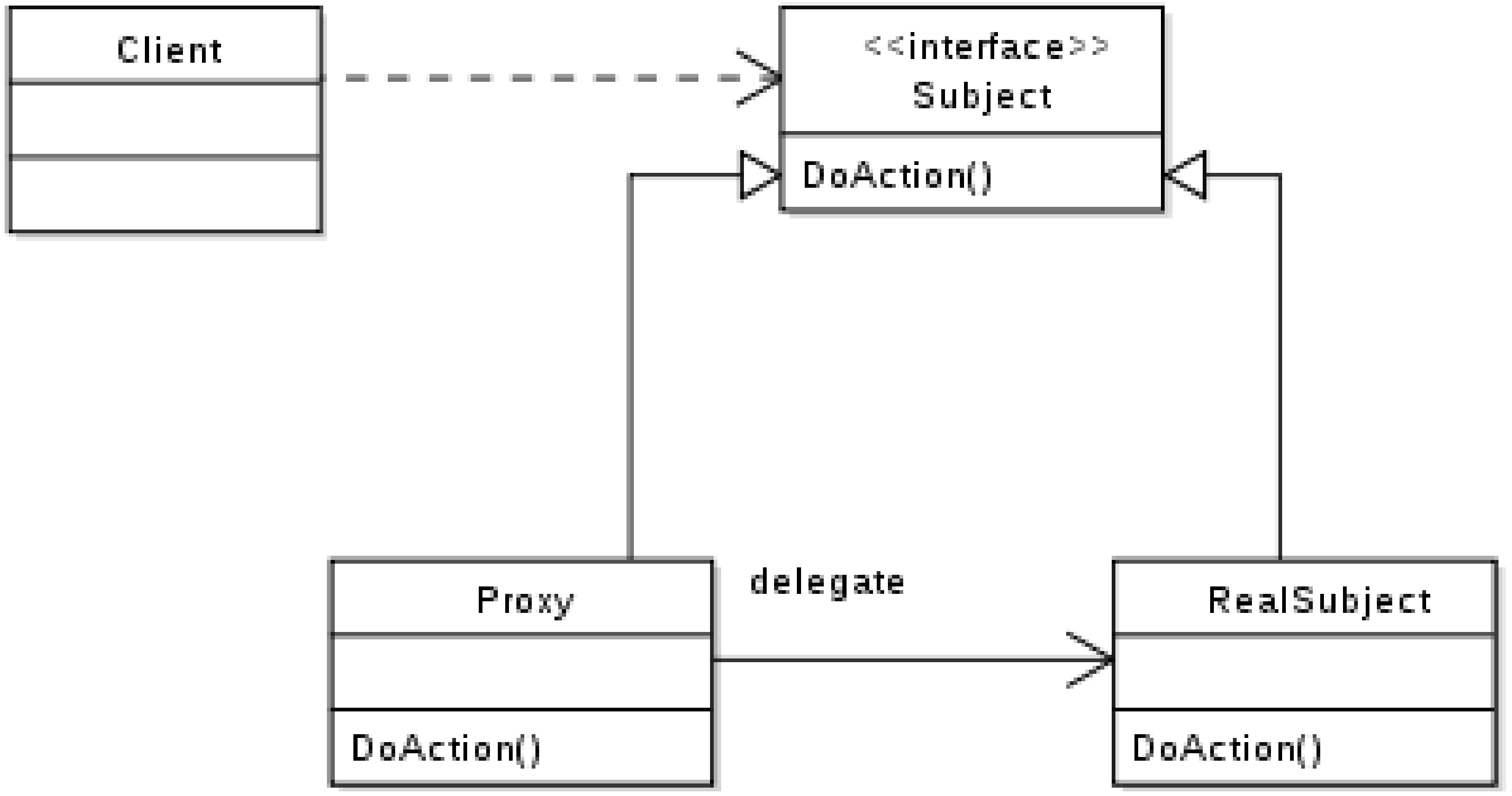


Proxy

Problem

- Remote proxy (cross-domain requests)
- Restriction proxy
- Etc...

Proxy

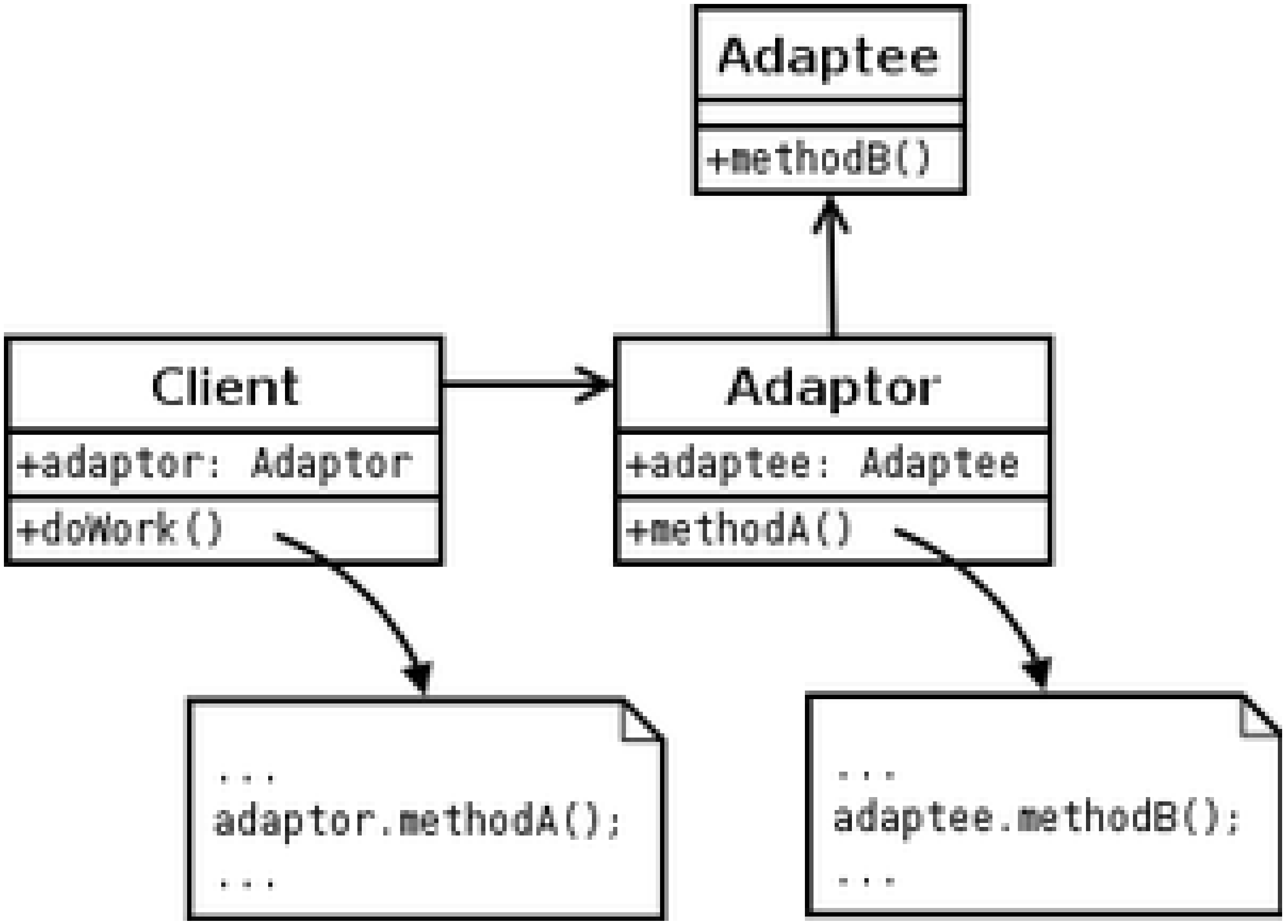

Client
<<interface>>
Subject
DoAction()
Proxy
DoAction()
delegate
RealSubject
DoAction()

Adapter

Problem

- JavaScript library (jQuery, MooTools, Dojo, ExtJS...)
- Independent application
- Dynamically changing the library

Adapter


Adaptee
+methodB()
Client
+adaptor: Adaptor
+doWork()
Adaptor
+adaptee: Adaptee
+methodA()
...
adaptor.methodA();
...
...
adaptee.methodB();
...

Variable scope

- Local variables
- Global variables
- Instance variables

Namespace pattern

Problem

- Organizing the source code
- Holding logical groups of unique identifiers or symbols
- Dealing with name collisions

- Java - packages
- C++, C# - namespaces
- JavaScript - ?!

JavaScript objects

They have

- Properties - objects
- Methods - objects

The object literal

```
var person = {
 talk: 'Foo',
 action: function () {
     console.log('I\'m ' + this.name);
 }
};
```

- Less code
- Almost 2x faster (than using new)

- Harder to understand

Nesting objects

```
var foo = {
    bar: {
        foobar: { prop: 'I\'m inside foobar!' }
    }
};
```

Namespace function

```
function namespace(namespaceString) {
  var parts = namespaceString.split('.'),
      parent = window,
      currentPart = '';

  for(var i = 0, length = parts.length; i < length; i++) {
      currentPart = parts[i];
      parent[currentPart] = parent[currentPart] || {};
      parent = parent[currentPart];
  }

  return parent;
}
```

Sample usage

```
(function () {
  var stream = namespace('com.appblast.stream');

  stream.xhr = function () {
      //implementation
  };

  stream.websocket = function () {
      //implementation
  };
}());
```

Module pattern

Problem

- Data hiding
- Encapsulation
- Abstraction
- Complexity

Creating "classes"

```
function Person(name) {
  this.name = name;
  this.talk = function () {
      return this.name + " is talking!";
  }
}
var person = new Person('Foo');
console.log(person.name); // "Foo"
console.log(person.talk()); // "Foo is talking!"
```

- A basic encapsulation
- Abstraction

- No data hiding

Module pattern

```
var module = (function () {
 function privateFoo() {
     console.log('The private foo.');
 }
 return {
     publicFoo: function () {
         privateFoo();
         console.log('The public foo.');
     }
 };
}());
module.publicFoo(); //'The private foo.'
                 //'The public foo.'
module.privateFoo(); //TypeError: Object #<Object>
                      // has no method 'privateFoo'
```

Example

```
var website = (function () {
    var title = 'Default title';
    return {
        setTitle: function (t) {
            title = t;
            document.title = title;
        },
        getTitle: function () {
            return title;
        }
    };
}());
website.setTitle('Sample title');
console.log(website.getTitle()); //'Sample title'
```

Module pattern - Variation (1)

```
var module = (function () {
    function private() {
        console.log('Private method');
    }
    function public() {
        console.log('Public method');
    }
    return {
        public: public
    };
}());
```

```
var module = (function () {
    function public() {
        private();
    }
    function anotherPublic() {
        console.log('anotherPublic');
    }
    function private() {
        anotherPublic();
    }
    return {
        public: public,
        anotherPublic: anotherPublic
    };
}());

module.public(); // 'anotherPublic'
module.anotherPublic = function () {
    console.log('Brand new public');
};
module.public(); // 'anotherPublic'
```

AMD and CommonJS

CommonJS

Development of independent pieces of JavaScript which can be easily reused.

Globals:

- exports
- require

CommonJS exports

By adding properties to the **exports** object we specify what we want to make available for other modules.

```
var lib = require('./utils/lib'), //relative
    fooLib = require('fooLib');

lib.doAwesomeThing();
fooLib();

function bar() {
  //body
}

exports.bar = bar;
```

CommonJS exports

```
//Human.js
function Human(name) {
  this.name = name;
}
exports.Human = Human;

//app.js
var Human = require('./Human.js').Human;

var p = new Human('foo');
console.log(p.name); //'foo'
```

CommonJS

Typical usage of CommonJS is in Node.js. The modules are loaded synchronously so we don't need additional callbacks or promises.

Asynchronous Module Definition

- Definition of JavaScript modules for the Web (browser first approach)
- Dojo - XHR + eval
- Allows asynchronous loading of the modules
- Without coupling between the source code and module represenation
- Mailing list

Module definition

```
define(
    module_id /* optional */,
    [dependencies] /* optional */,
    definition function /* function for instantiating the module or ob
);
```

Special dependencies

- require
- exports
- module

The default arguments which the factory function accepts

Dependencies

- Can be specified with relative indentifiers
- Their resolution starts depending on their order in the dependencies array
- The order in the dependencies array shows the order in which they will be passed to the factory

Example

```
define(["alpha"], function (alpha) {
    return {
      verb: function(){
        return alpha.verb() + 2;
      }
    };
});
```

Example

```
define("alpha", ["require", "exports", "../beta"], function (require,
    exports.verb = function() {
        return beta.verb();
    }
});
```

Example

```
define({
  add: function(x, y){
    return x + y;
  }
});
```

Example (CommonJS in AMD)

```
define(function (require, exports, module) {
  var a = require('a'),
      b = require('b');

  exports.action = function () {};
});
```

Publish/subscribe

Problem

- Loose coupling
- Easy reuse
- Maintainability

Solution

```
var pubsub = {};
(function(q) {
  var topics = {};
  q.publish = function (topic, args) {
    if (!topics[topic]) {
      return false;
    }
    var subscribers = topics[topic],
        len = subscribers ? subscribers.length : 0;
    while (len--) {
      subscribers[len].call(null, topic, args);
    }
    return this;
  };
  q.subscribe = function (topic, func) {
    if (!topics[topic]) {
      topics[topic] = [];
    }
    topics[topic].push(func);
    return this;
  };
}(pubsub));
```

Sample architecture

Problem

- Dealing with complexity
- Easy replacement of the components
- Reusability

The Core

Adapter

- Standardized interface for DOM, AJAX..whatever library.
- Providing this interface to the "Sandbox"

```
App.core.dom = {
    setWidth: function (elem, width) {
        $(elem).width(width);
    },
    getWidth: function (elem) {
        return $(elem).width();
    },
    getOffset: function (elem) {
        return $(elem).offset();
    }
};
App.core.ajax = {
    get: function (url, data) {
        $.ajax({
            type: 'get',
            url: url,
            data: data
        });
    }
};
```

The Sandbox

Pub-sub/Proxy

- Providing basic API for the modules
- Modules communicate via the Sandbox
- Can provide some form of restriction if necessary

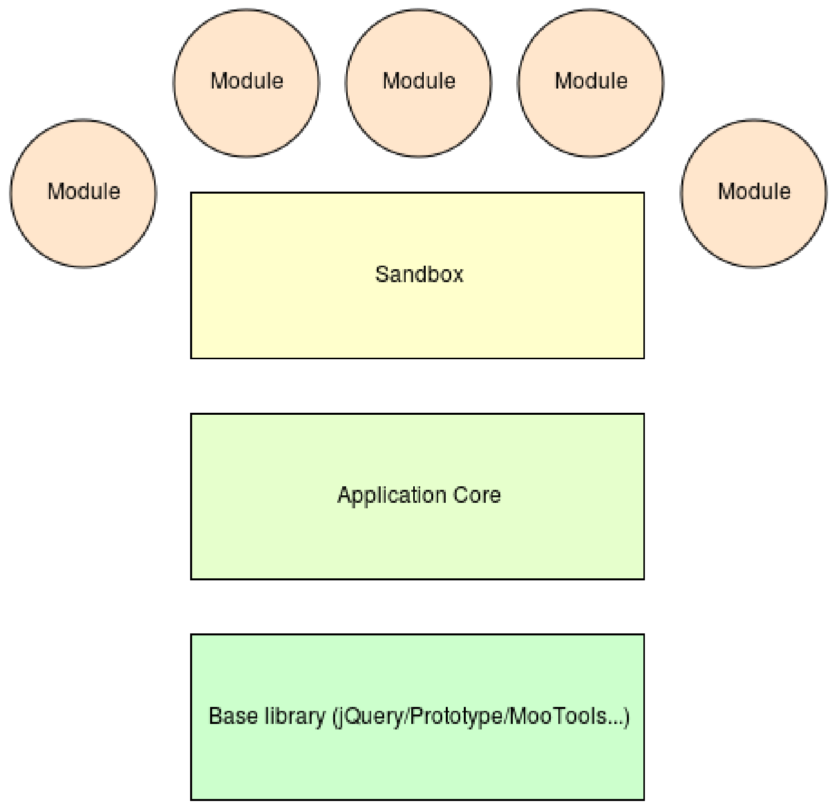
```
App.sandbox = (function (core) {
    var modules = {},,
        topics = {},    //publish/subscribe
        //...
        moduleInterface = {
            dom: core.dom,
            ajax: core.ajax,
            publish: publish,
            subscribe: subscribe
        };
        function start(moduleId) {
            var module = modules[moduleId];
            if (module !== undefined && typeof module.init === 'functi
                modules.init.call(null, moduleInterface);
            }
        }
    return {
        register: register, //register module
        start: start,
        stop: stop
    };
}(App.core));
```

Modules

The Module pattern

- Our building blocks
- Do not access any globals!
- Do not know for the other modules, just playing in the Sandbox

```
App.sandbox.register('news', (function () {
    function init(sandbox) {
        addHandlers();
        //...
    }
    function addHandlers() {
        //...
    }
    return {
        init: init
    };
}());
```


Module
Module
Module
Module
Module
Sandbox
Application Core
Base library (jQuery/Prototype/MooTools...)

Sample architecture (+/-)

- Simple to build
- Encapsulation
- Low coupling
- Reusability

- HARD for unit testing

Implementations

- Aura.js
- ScaleApp
- Kernel.js
- Terrifically
- Hydra.js
- Yahoo projects

Best practices

- Always use === instead of ==
- Do not omit var
- Use a single var keyword for declaring multiple variables
- Do not pollute the global namespace
- Do not define functions in a loops
- Do not use eval
- Do not pass strings to setInterval or setTimeout

Best practices (2)

- Use [] and {} instead of new Array, new Object
- Use the radix parameter of parseInt
- Always use semicolons
- Use IIFE
- Use JSLint or JSHint

Useful resources

- Object-Oriented JavaScript
- JavaScript: The Good Parts
- JavaScript Patterns
- Learning JavaScript Design Patterns
- JSLint
- JSHint
- ES5 compatibility table

Thank you!