PRACTICAL JAVASCRIPT

Agenda

- Understand the major differences between popular static languages (C++/C#/Java) and JavaScript
- Best Practices
- Pitfalls
- ECMAScript 6
- Typescript

JavaScript is dynamic

- You don’t specify the data type of a variable when you declare it
- The same variable can point to different data types
- We use var to declare a variable
- A variable has a scope
  - Global variables should be avoided (like in any other object oriented language)

```
var answer = 42;
answer = "Meaning of life";
```

Declaring Variables

- Case sensitive
- $ and _ are valid variable names
  - And common
- Cannot use reserved keywords
- Usually, camel case convention

```
$(function () {
    var res = _.map([1, 2, 3], function (num) {
        return num * 2;
    });
});
```

- Do you like above code ?

Implicit Variable Declaration

- You can write into a variable even when this variable was not declared before
- Don't do that !
- In this case a global variable is created

```
function g() {
    global = 12;
}

g();

alert(local);
```

Strict Mode

```
"use strict"

function g(){
    x = 10;
}

g();
```

Error is thrown

Strict Mode

- A way to *opt in* to a restricted variant of JavaScript
- Strict mode makes the following changes
  - Eliminates some JavaScript silent errors
  - Fixes mistakes that make it difficult for JavaScript engines to perform optimizations
  - Prohibits some syntax likely to be defined in future versions of ECMAScript

Automatic Initialization

- Like other modern programming languages, JavaScript supports automatic initialization
- The value of uninitialized variable is undefined
  - Not the same as null value

```
var num;

console.log(num == undefined);
```

Undeclared Variable

- You cannot read a value of undeclared variable

```
try {
    if (xxx == 10) {
    }
}
catch (e) {
    console.log(e.message);
}
```

- You can ask for the typeof of an undeclared variable

```
console.log(typeof xxx);
```

→ "undefined"

Window is the Global Scope

- Every global variable is a property of a global object named window

```
var num = 10;
console.log(window.num); //prints 10

window.num = 11;
console.log(num); // prints 11
```

- Objects in JavaScript are dynamic → Global scope is dynamic ☺
  - See next slides about objects

NodeJS

- Inside NodeJS top level scope variables are not global
  - Are scoped to the current module
- Can use the global variable to create global variable
  - window does not exist
  - Is considered bad practice

Built-in types

- JavaScript supports only the following types:
  - number
  - boolean
  - string
  - function
  - object
  - undefined
- Given a variable you can use the keyword typeof to read it’s runtime type

Built-in types

```
console.log(typeof 1); // number
console.log(typeof 1.2); // number
console.log(typeof "abc"); // string
console.log(typeof "abc"[0]); // string
console.log(typeof true); // boolean
console.log(typeof function () { }); // function
console.log(typeof {}); // object
console.log(typeof null); // object
console.log(typeof new Date()); // object
console.log(typeof window); // object
console.log(typeof undefined); // undefined
console.log(typeof blabla); // undefined
```

Value vs. Reference type

- Same concept as in Java/C#
- Built-in data types are grouped into
  - Reference types (object, array and function)
  - Value types (others …)
- A reference is implemented as a pointer
  - Points to an object that resides inside the heap
  - Many references can point to the same object
- A value can only be copied
  - You cannot get the address of a value

Number

- There is no distinction between integer and double
- All type of numbers are represented as 64bit floating point values
  - 10/3 = 3.3333 not 3
- parseInt can be used to parse a string into a number. In case of failure NaN is returned

```
var str = document.getElementById("firstName").value;
if (isNaN(parseInt(str))) {
    alert("Please enter a number");
}
```

String (1)

- String contains any Unicode character
- No character type
  - str[0] is also a string !!!
- String literal can be expressed using “ or ‘

```
var str = "ABC";
var str = 'ABC';
```

- Strings are immutable
  - Allows for runtime optimization

```
var str = "ABC";
str[0] = "X";
```

→

str is still “ABC”

String (2)

- Should we use “ or ‘ when writing string literals?
  - Probably a matter of style
  - Programmers with C++\Java\C# background tend to use double quotes
  - Veteran Web Programmers tend to use single quote
- You should be aware of the following
  - JSON requires double quotes
  - HTML/XML attributes are usually expressed using double quotes
    - Therefore, when building XML fragments at runtime it is easier to use single quote for the whole string literal

Undefined

- A special data type
- Has only one value named undefined
- The value undefined is important concept in JavaScript
- You may encounter it during several cases
  - Uninitialized variable
  - A function without a return value
  - A function parameter that was not specified by the caller
  - A non existent object property
  - A non initialized array index

Comparison Operators

- JavaScript has both strict and abstract comparisons
- A strict comparison is only true if the operands are of the same type
- Abstract comparison converts the operands to the same type before making the comparison

```
console.log(0 == false);
console.log(2 == "2");
console.log(undefined == null);
```

```
console.log(0 === false);
console.log(2 === "2");
console.log(undefined === null);
```

Data Type Conversion

- Data types are converted automatically as needed during script execution
- Operator + may convert numeric values to strings

```
var num = 10;
alert(num + "0");
```

→ 100

- Other operators may convert string values to numeric

```
var num = 10;
alert(num * "2");
```

→ 20

Conversion Tricks

- Some JavaScript programmers use operators + and * to convert data types
- Convert string to number

```
var str = document.getElementById("firstName").value;
if (isNaN(str * 1)) {
    alert("Please enter a number");
}
```

- Convert number to string

```
var num = 10;
console.log(num + "");
```

Falsy values

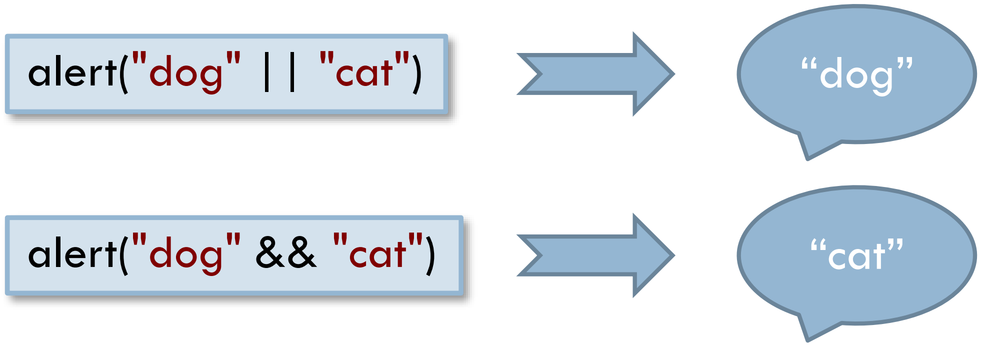
- The following values are considered false when being used inside if statement
  - false
  - null
  - undefined
  - 0
  - “”
  - NaN
- Others values are considered Truthy

Logical Operators

- Typically used with Boolean values
  - In that case, they return a Boolean value
  - Behavior is consistent with other static programming languages (C++/Java/C#)
- May be used with non Boolean values
  - In that case, they return a non-Boolean value

Logical Operators

- “dog” is considered Truthy



```
alert("dog" || "cat")
alert("dog" && "cat")
```

Array

- Array is created using the following syntax
  - []
  - new Array

Preferred

```
var arr = [];
var arr = [1,2,3];
```

Less common

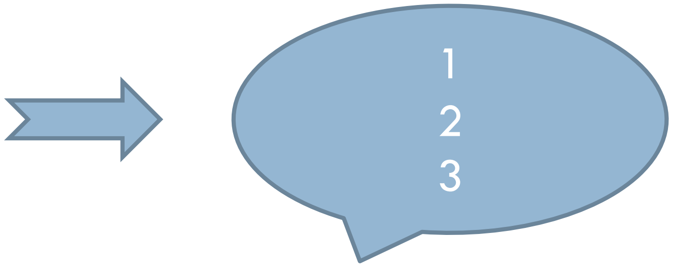
```
var arr = new Array();
var arr = new Array(10); // length is 10
var arr = new Array(10, 2); // length is 2
```

Iterating an Array

- Straight forward
- Use a running index and the length property

```
var arr = [1, 2, 3];

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}
```


```
1
2
3
```

Iterating an Array

- ES5 syntax

```
arr.forEach(function(value, index, arr){
    console.log(value, index);
});
```

There is no way to stop the loop

- ES6 syntax

```
const arr = [1,2,3];

for(const num of arr){
  console.log(num);
}
```

in syntax

- Should not be used with arrays

```
const arr = ["a","b","c"];

for(const key in arr){
    console.log(key);
}
```

Prints 0,1,2

Array is dynamic

- New elements can be added/deleted at runtime
  - In contrast to static languages
- The property length is automatically being updated

```
var arr = [];
arr.push(10); // add last
arr.pop(); // remove last
arr.splice(arr.length-1, 1); // remove last
arr[10] = 10; // never throws an exception
arr.length = 2; // resize
arr.shift(); // remove first
arr.unshift(111); // insert first
arr.concat([]); // clones an array
arr.slice(0, 4); // returns part of the array
```

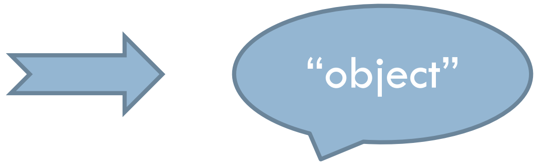
Array Extras

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

Object

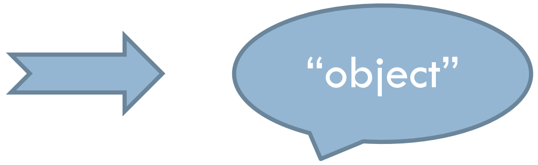
- A container of keys and values
- The key must be of type string
- Has built-in methods
- Creating empty object is easy

```
var obj = {};
alert(typeof obj);
```



Less common

```
var obj = new Object();
alert(typeof obj);
```



Initializing an Object

- An object can be initialized at declaration
- A.K.A object literal syntax (the basis for JSON)

```
var obj = {
    id: 123,
    name: "Udi",
    email: "udi@gmail.com"
};
```

Less common

```
var obj = {
    "id": 123,
    "name": "Udi",
    "email": "udi@gmail.com"
};
```

Object is dynamic

- Properties can be added/removed after creation

```
var obj = {};
obj.name = "Ori";
obj["name"] = "Ori";
```

- Removing a property

```
delete obj["name"];
delete obj.name;
```

- Accessing non existent property yields the value undefined

Performance

- Prefer defining all fields up front
- Adding new fields on demand increases object size and hurts read operation's performance

```
const obj_number = {
   id: 1,
};

const obj_number_string = {
   id: 1,
   name: "Ori Calvo"
};

const obj_number_string_boolen = {
   id: 1,
   name: "Ori Calvo",
   flag: true,
};
```

16 bytes

20 bytes

24 bytes

Object Content

- The for…in statement allows you to iterate over all object’s properties

```
var obj = {
    "id": 123,
    "name": “Roni",
    "email": "roni@gmail.com"
};

for (var key in obj) {
    var value = obj[key];
    console.log(key + " = " + value);
}
```

```
id = 123
name = roni
email = roni@gmail.com
```

Array is an Object

- You can act on an array is if it was an object (it is !)
- Not recommended
- What is the expected output?

```
var arr = [1, 2, 3];
arr.name = "Ori";

for (var i = 0; i < arr.length; i++) {
    console.log(arr[i]);
}

for (var key in arr) {
    console.log(arr[key]);
}
```

Function

- More than just a method …
  - The basic for advanced JavaScript techniques
- Declaring a function

```
function add(num1, num2) {
    return num1 + num2;
}
```

- Calling a function is also straightforward

```
var res = add(num1, num2);
```

Pass by value

- JavaScript only supports “pass by value” mechanism
- The parameter being sent to a function is copied
  - Whether it is a reference or a value

```
var str = "ABC";

function modify(str) {
    str = "XXX";
}

console.log(str);
```

What will be printed ?

- 10?
- 11?

```
var num = 11;

function doSomething() {
    console.log(num);
    var num = 10;
}

doSomething();
```

Where to declare variables ?

- A variable is accessibly inside its surrounding function
- Even before point of declaration
- Therefore many JavaScript programmers declare all variables at the beginning of the method

```
var num = 11;

function doSomething() {
    console.log(num);
    var num = 10;
}

doSomething();
```

Overloading

- JavaScript does not support Overloading
- Last method wins

```
function g(){
    console.log("abc");
}

function g(){
    console.log("123");
}

g();
```

Overloading

- You can simulate overloading

```
var ERR = "ERR";
var WRN = "WRN";
var MSG = "MSG";

function log(type, message) {
    if (message == undefined) {
        message = type;
        type = MSG;
    }

    console.log(type + " " + message);
}
```

```
log(ERR, "Internal Error");
log("Connecting to server");
```

Function – The Dark Side

- A function is an object

```
function f() {
    var num = 10;
}

f.num = 11;

f.hasOwnProperty(”num”)

if(f==g) {
}
```

Function – The Dark Side

- Has built-in properties and methods

```
function f(input) {
    console.log(f.name); // the name of the method
    console.log(f.length); // number of parameters
    console.log(f.toString()); //function source code
    console.log(f.arguments); // available only during execution
    console.log(f.caller.name); // available only during execution
}
```

arguments

- An array like which holds all function’s arguments
- Does not support all Array functionality
  - You may use Array.from(arguments)

```
function g() {
    for(var i=0; i<arguments.length; i++){
        console.log(arguments[i]);
    }
}

g(1,2,3);
```

Function – Indirect Invocation

- A function can be invoked using special syntax

```
function f(name) {
    console.log("Hello " + name);
 }

f.call({}, "Ori");
f.apply({}, ["Ori"]);
```

- Although not intuitive, above syntax is quite common
- Mainly, when doing Object Oriented JavaScript

Function creates a Scope

- Function creates a new scope which is isolated from outer scope
- Outer scope cannot access local variables of a function

```
var num = 20;

function f() {
    var num = 10;

    console.log(num); // yields 10
}

f();

console.log(f.num); // yields undefined
```

Closure

- Inner function may access the local variables of the outer function
  - Even after outer function completes execution
- Allows us to simulate state-full function

```
function getCounter() {
    var num = 0;
    function f() {
        ++num;
        console.log("Num is " + num);
    }
    return f;
}
```

```
var counter = getCounter();
counter();
counter();
```

Function inside an Object

- An object can contain functions

```
var obj = {
    dump: function() {
        console.log("dumping...");
    }
};

obj.dump();
```

- Feels like OOP
- The keyword this is used for accessing other properties (see next slide)

The this keyword

- Available only inside a function
- Points to the object that this function is being invoked on

```
var obj = {
    id: 123,
    dump: function() {
        console.log(this.id);
    }
};

obj.dump();
```

- Global function points to the window/global object

Apply & Call - Recap

- You can control the value of this using apply and call methods

```
var obj = {
    id: 123
};

function dump() {
    console.log(this.id);
}

dump.call(obj);
```

Self Executing Function

- A function can be declared without a name
- Since no name exist no one can invoke it
- Except the code that declared it
- A.K.A self executing function

```
(function () {
    // External code has no access to these variables
    var url = "http://www.google.com";
    var productKey = "ABC";
})();
```

Sending Parameters

- Think about the $ sign
- Usually it points to jQuery global object
- But how can we ensure that?
  - There might be a case were additional 3rd party library overrides it

```
(function ($) {
    $.ajax({
        url: "www.google.com",
        type: "GET",
    });
})(jQuery);
```

Module

- Arrange your JavaScript code into modules
- Each module is surrounded with self executing function thus hiding all local variables and functions
- Peek the ones that should be public (sparsely)

```
var server = (function () {
    var baseUrl = "http://www.google.com";

    function httpGet(relativeUrl) {
        $.ajax(...);
    }

    return {
        httpGet: httpGet,
    };
})();
```

Agenda

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

From Module to Class

- 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

```
function Point(x, 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

- 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

- 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

- 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

- 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

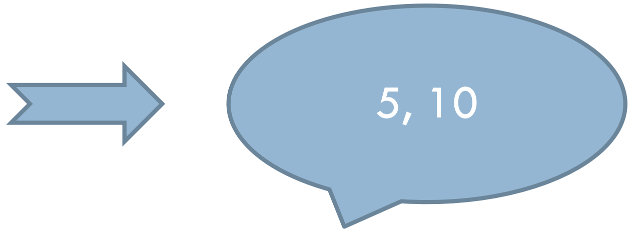
Prototype

- 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 ..)

- 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

- 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

- 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

- 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

```
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

- 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

- 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

- 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

- 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

- 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

- 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

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

    this.progLang = progLang;
}

Programmer.prototype = Object.create(Employee.prototype);

Programmer.prototype.changeLang = function (progLang) {
    this.progLang = progLang;
}

var prog = new Programmer(123, "Ori", "JavaScript");
```

Polymorphism

- 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

```
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

```
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

- 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

- 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

- 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

- 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

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?

- 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

- 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?

- 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

- 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