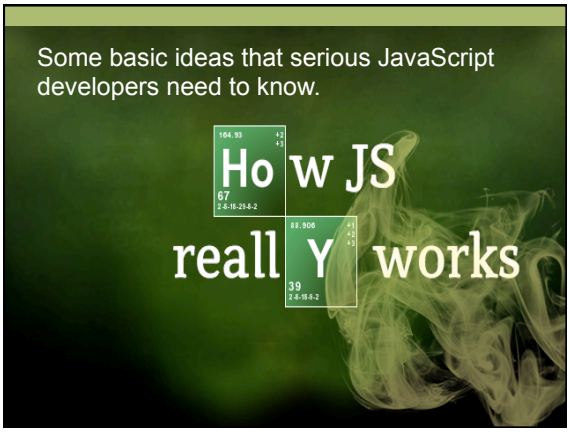
Advanced JavaScript Basics

No, that's not an oxymoron

Some basic ideas that serious JavaScript developers need to know.



"use strict";

Strict mode ...

- Protects us from doing stupid things we shouldn't be doing anyway.

- To enable, just go ...

```
"use strict";
```

- At the top of a script file or function
- The quotes are weird but help with much older browsers.

What strict mode does (1 of 4)

- Can't declare variables implicitly
- Not okay:

```
x = "foo";     // ReferenceError
```

- But this is cool:

```
var x = "foo";
```

- strict disallows the with statement altogether (Yay!)

What strict mode does (2 of 4)

- Can't use certain reserved words in ES5
- Not okay:

```
var protected = {
  class: 1,
  implements: 2,
  interface: 3
};
```

What strict mode does (3 of 4)

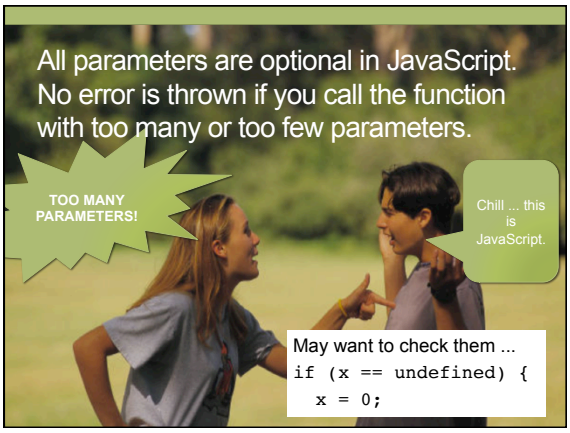
- eval() works but is restricted
- All variables in an eval are local. Prevents hackers from running certain XSS attacks.
- But eval() is ugly even without security issues, so avoid it.

What strict mode does (4 of 4)

- It won't let you alter arguments
- Which shouldn't be reassigned anyway

```
function foo(x) {
  x = "foo";  // x will change, but
              //   arguments[0] will not
  arguments = [ 1, 2, 3 ];  // Error
}
```

Functions are variadic



arguments allows access to all parameters

- Every function has an arguments object

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

Rest parameters may help with too many arguments

```
function sum(...nums) {
  let total = 0;
  nums.forEach(x => total += x);
  return total;
}
```

- arguments is array-like. nums is a true array.

Default parameters may help with too few

```
function foo(first="John",
  last="Doe", age=getVotingAge()) {
  // Do stuff with first, last, and age here
}
```

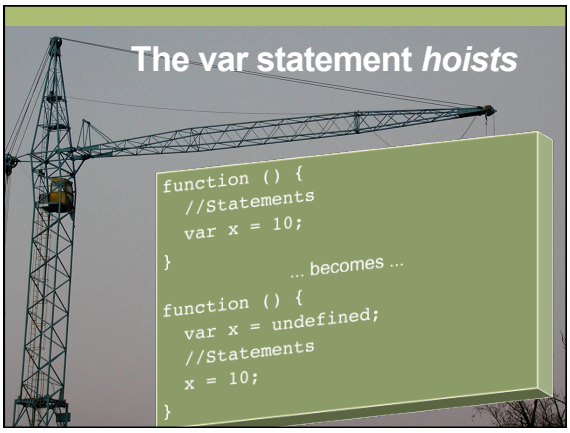
- If you supply a value it'll be used. If not, the default value is.
- Allows you to pass in null, "", 0, or false as valid values and have them used.

Hoisting

What happens to x?

```
1 var city = document.getElementById('city').value;
2 if (city === 'Albuquerque') {
3   var x = 10;
4 }
5 console.log("x is ", x);
```

What happens on line 5?
A) x is always 10
B) x may be 10 or it may be undefined
C) A ReferenceError is thrown



Remember ... let and const do not hoist

```
1 function foo() {
2   if (price < 200) {
3     var shouldSell = false;
4     const makeLess = true;

5     let capacity = oldCapacity - 10;
6   }
7 }
```

Hoisted

Stay put

Function statements vs. function expressions

Consider ...

```
var x = 5;
var x = 'a string';
var x = new Date();
var x = ['Walt','Jesse','Skyler'];
var x = {};
```

What do you call the things on the right?

Expressions!!

JavaScript has a *function* expression

```
function (params) {
  body here
}
```

- Keyword function
- Name is optional
- Zero or more parameters
  - Bound by parentheses
  - Separated by commas
- Body
  - Bound by curly braces
  - Zero or more statements

Functions are objects! You can ...

```
// Assign to a variable
var x = function () { doSomething() };
// Pass them as arguments
doSomethingElse(x);
// Return them from other functions
function foo() {
  return function () { doThings(); };
}
// Put them in arrays
var arrayOfFunctions = [ x, y, z ];
// ... and more!!
```

The function statement

```
function foo() {
}
```

... is more like ...

```
var foo = function () { ... };
```

- But both parts are hoisted

Execution context

Where a function runs depends on how it is invoked

The *this* variable refers to the current context (environment)

Costello: My code doesn't work
Abbott: I see the problem. *this* is wrong.
Costello: I know. What's the issue?
Abbott: *this*. You're misusing *this*
Costello: I'm misusing what?
Abbott: No, you're misusing *this*.
Costello: Well are you going to tell me what's the problem or aren't you?
Abbott: I am. You're misusing *this*.
Costello: Exactly where's the issue in my code?
Abbott: No, not where ... *this*.
...

Meaning depends on context

Bad	Good
• His feelings were crushed when she left. • Is it me or was that waiter salty to us?	• He crushed it in the 4th quarter! • The bisque was delicious! So salty!

4 ways to invoke a function

1. Method form
2. Call form
3. Function form
4. Constructor form

Say you have a shopping cart ...

```
console.log(cart);
[{
  qty: 5,
  item:{id:59,desc:'jelly beans',price:2.12}
},{
  qty: 1,
  item:{id:844,desc:'rice',price:3.25}
},{
  qty: 5,
  item:{id:562,desc:'apples',price:2.75}
}]
```

... And a calcTotal() function

```
function calcTotal(prefix) {
  const sum = this.reduce((prev, curr) =>
    prev + curr.qty * curr.item.price, 0);
  return prefix + sum;
}
```

- How this runs depends on what *this* is

Method form

- The function is already part of the object on which it is working
- Example

```
cart.getTotal = calcTotal;
var tot = cart.getTotal("CAD$");
```

Call Form

- With these, the context is passed **in** to the function.
- Syntax:
- function.call(context, parm1, parm2 ...)
  - Pass in any number of parameters
- function.apply(context, [parm1, parm2 ...])
  - Pass in an array of parameters
- Examples:

```
let tot = calcTotal.call(cart, "US$");
let tot = calcTotal.apply(cart, ["€"]);
```

Function form

- Simply ... call it!
- Example:

```
calcTotal();
```

- Fails b/c window isn't a cart. What's a qty? What's an item? What's a price?

A note about pure functions

- They don't change anything outside of them.
- All values are passed in and a value is passed out
- Example ...

```
function calcCartTotal(c, prefix) {
  const sum = c.reduce((prev, curr) =>
    prev + curr.qty * curr.item.price, 0);
  return prefix + sum;
}

const total = calcCartTotal(cart, "£");
```

Constructor form

- You can "new" up a function ...
- When you use *new*,

1. An empty Object is created
   - It replaces window as the context
   - "this" refers to that new context
2. The lines in the function are run in this context
3. That new context is returned

Constructor form example

```
function Item(itemId) {
  var _item = getItemViaAjax(itemId);
  this.id = itemId;
  this.price = _item.unitPrice;
  this.desc = _item.description;
}

var cart = [
  {qty: 5, item: new Item(59)},
  {qty: 1, item: new Item(844)},
  {qty: 5, item: new Item(562)}
];
```

Summary

Form	Example	this
Method	obj1.foo();	obj1
Function	foo()	global
Constructor	newObj = new foo();	the instantiated object
Call	foo.call(obj1, p1,p2, ...); foo.apply(obj1, [p1, p2, ...]);	obj1

tl;dr

- JavaScripts hoist all variable declarations
- It also hoists function statements
- Function expressions omit the name of the function.
- Function statements include them.

Unit Testing JavaScript

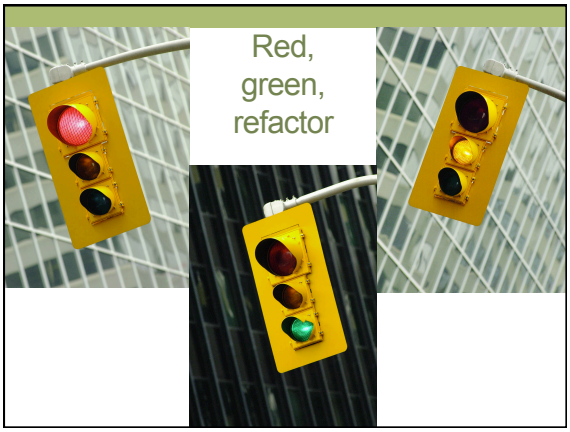
A unit test tests one unit of work

- One requirement for one method
- They are:
  - Isolated from other code
  - Isolated from other developers
  - Targeted
  - Repeatable
  - Predictable

What is NOT unit testing? ... Anything else!

- > 1 requirement
- > 1 method
- > 1 project
- > 1 system
- Affects > 1 developer


It starts
with
the test


Red,
green,
refactor


The red phase

You want to write positive tests and negative tests

- Negative tests involve values that are outside acceptable ranges.
  - They should fail
  - You're testing to make sure that they do
- Positive tests are ones that should pass

The green phase

The refactoring phase

Once finished you pick up a user story and start again writing a new failing test

QUnit is a unit testing framework for JavaScript

- Built by the jQuery team

To get it, download and save two files

- qunit.js
- qunit.css

Make a testrunner

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>QUnit Example</title>
  <link rel="stylesheet"
   href="//code.jquery.com/qunit/qunit-1.22.0.css">
</head>
<body>
  <div id="qunit"></div>
  <div id="qunit-fixture"></div>
  <script
    src="//code.jquery.com/qunit/qunit-1.22.0.js">
  </script>
  <script src="tests.js"></script>
</body>
</html>
```

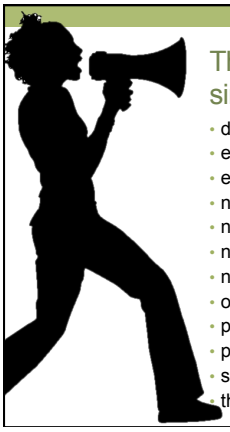
Define tests using QUnit.test()

Syntax

- QUnit.test(title, testFunction);

Example

```
QUnit.test("Lightsabre blocks blaster ray",
  function (assert) {
    var s = new Lightsabre('Quigon');
    var blaster = new Blaster('battledroid1');
    var result = s.block(blaster.fire());
    assert.equal(result, true);
});
```



The assertions are pretty simple

- deepEqual(actual, expected)
- equal(actual, expected)
- expect(number)
- notDeepEqual(act, expected)
- notPropEqual(act, expected)
- notStrictEqual(act, expected)
- notEqual(actual, expected)
- ok(bool)
- propEqual(actual, expected)
- push()
- strictEqual(actual, expected)
- throws(actual, expected)

The most basic check is ok()

- If the parameter passed is truthy, it passes
- If not, it fails

Syntax

- assert.ok(parameter)

Example

```
QUnit.test("Death Star is really big", function
(assert) {
  assert.ok(this.deathStar.getSize() > 1e25);
}
```

How to test for equality

equal(a, e)	strictEqual(a, e)	deepEqual(a, e)	propEqual(a, e)
Makes sure the actual value and the expected value are the same. Like "==".	Like equal but uses "==="	Like equal but for objects. Checks all values at all depths. Uses "==".	Like deepEqual() but checks all values using "==="
notEqual(a, e)	notStrictEqual(a, e)	notDeepEqual(a, e)	notPropEqual(a, e);

throws() makes sure an Error was thrown

Syntax

- assert.throws(function[, expectedError][, expectedMesg]);

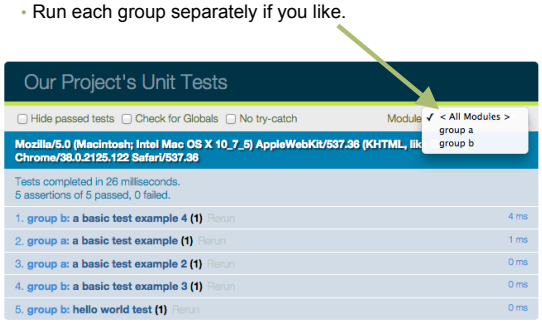
Example

```
QUnit.test("Attacking a fellow Jedi throws error",
  function (assert) {
    assert.throws(function () {
      var luke=new Jedi('Luke Skywalker');
      var mace = new Jedi('Mace Windu');
      luke.attack(mace);
    }, /traitor/);
});
```

- The highlighted function above is expected to throw an error with a message that contains the regular expression "traitor".
- If not, the unit test fails.

We can organize our runs with modules

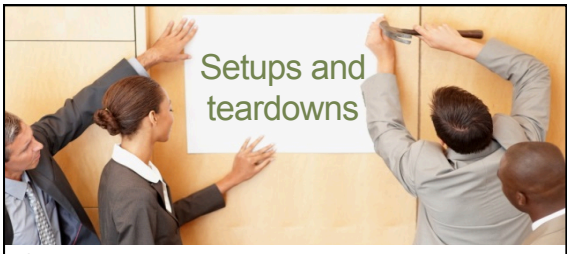
- Run each group separately if you like.



QUnit.module creates those modules

```
QUnit.module("group a");
QUnit.test( "a basic test example", function(assert) {
  assert.ok( true, "this test is fine" );
});
QUnit.test( "a basic test example 2", function(assert) {
  assert.ok( true, "this test is fine" );
});

QUnit.module("group b");
QUnit.test( "a basic test example 3", function(assert) {
  assert.ok( true, "this test is fine" );
});
QUnit.test( "a basic test example 4", function(assert) {
  assert.ok( true, "this test is fine" );
});
```



Syntax

QUnit.module(name[, lifecycleObject]);

- where lifecycle object is a JSON object:

```
{
  "beforeEach": function () { /* Do setup things here. */},
  "afterEach": function () { /* Do teardown things. */}
}
```

Shared variables

- You can also have shared variables if you define them in the lifecycleObject:

```
QUnit.module("Sith Object Tests", {
  beforeEach: function () {
    this.master= new Sith('Darth Sidius');
    this.vader = new Sith('Darth Vader');
    this.master.setApprentice(this.vader);
  }
});
```

tl;dr

- TDD is a great idea
- Unit testing is the cornerstone of TDD
- TDD = Red, green, refactor
- QUnit is a unit testing framework for JavaScript.
- It provides a testing framework with a runner
- QUnit has assertions (ok, equal, deepEqual, etc.)
- You can group your tests into modules with QUnit

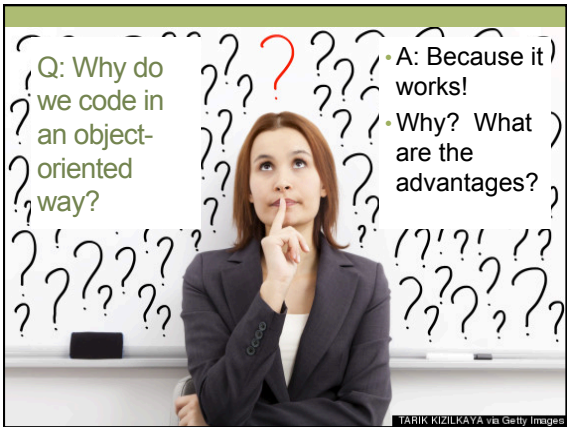
Further study

- Get QUnit from qunitjs.com
  - http://qUnitjs.com
- Great documentation on QUnit
  - http://api.QUnitjs.com

Declared Objects

Overview

- There are generally two types of objects
- Declared
  - One copy
  - Like a static object
- Created
  - Is instantiated
  - Like an instance object



Reminder: Objects and Classes

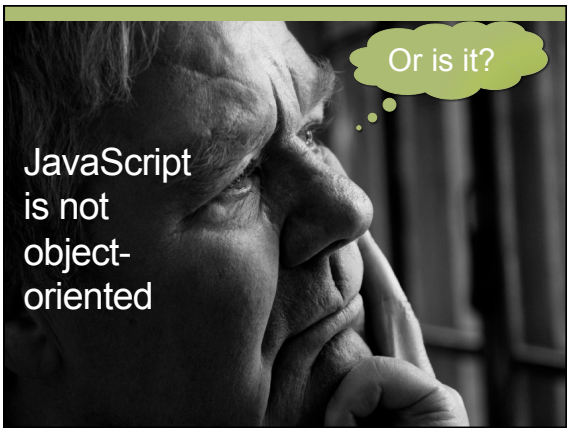
- Objects are created on the fly using _____
- Objects are really just _____-_____ pairs
- Another name for this is a _____.

To be OO, we need certain things

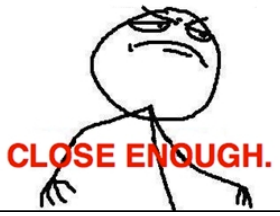
- ☐Objects
- ☐Classes
- ☐Properties
- ☐Methods
- ☐Constructors
- ☐Encapsulation
- ☐Private members
- ☐Public members
- ☐Static members
- ☐Accessor functions
- ☐Overloading
- ☐Inheritance
- ☐Overriding
- ☐Interfaces
- ☐Namespaces

JavaScript has no ...

- True classes
- Traditional inheritance
- Overloading
- Polymorphism
- Interfaces
- Namespaces



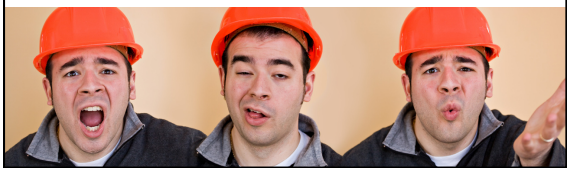
But we can simulate most of the OO behaviors



Objects

We create objects on the fly … no need for classes

```
var obj = {
  firstName: "Meg",
  lastName: "Griffin",
  pathology: "Rejection"
};
```



Two ways to create objects

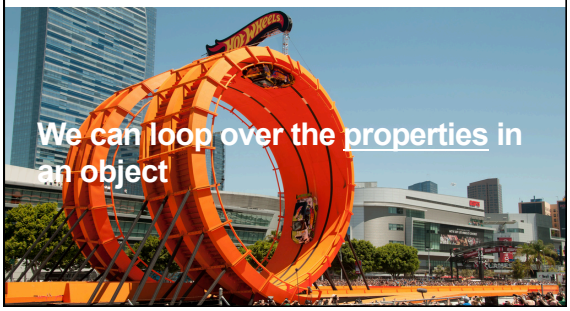
```
var o = new Object();
var o = {};
```

Properties

Properties are similarly created dynamically

```
var person1 = {
  lastName: "Griffin",
  firstName: "Stewie",
  city: "Quahog",
  state: "RI"
};
document.write("The person is " +
person1.firstName + " " + person1.lastName);
document.write("He is from " + person1["city"]);
var prop="state";
document.write("He lives in " + person1[prop]);
```

```
for (var propName in person1) {
  console.log(propName, person1[propName]);
}
```



Remember, objects can contain anything

```
var family = {
  name: "The Griffins",  // A string
  incomeInDollars: 34000,  // A number
  dog: brian,  // Another object
  parents: [ peter, lois ]  // An array
  playTheme: function () {
    return "It seems to me " +
      "that all you see is ...";
  }  // A function
}
```

Methods

Functions are objects themselves

- So they can be passed around like data

```
btn.addEventHandler('click',function () {
  // Do stuff
});
```

- They can be assigned to a variable

```
var doStuff = function () { // Do stuff });
```

- ... Even a property in a JSON object

```
var family = {
  ...
  playTheme: function () {
    return "It seems to me " +
      "that all you see is ...";
  }
}
```

So to run that last functio have to go ...

```
var x = family.playTheme();
```

Overloading

JavaScript is weird about function parameters, too

- All arguments are optional by default
- If you don't supply a value, it becomes undefined
- If you supply extra arguments, they are silently ignored
- Arguments are dynamically-typed

So, we sort of get overloading for free

- If we do this:

```
function x(a, b, c) { // Do stuff with a, b, & c }
```

- These all work:

```
x(1, 2, 3);
x(1);
x();  // a, b, and c are all undefined
x(1, 2, 3, 4, 5, 6, 7, 8);  // Extras are ignored
```

- So

```
function x(a, b, c) {
  if (b) { doSomethingWithB(); }
  if (c !== undefined) { doSomethingWithC(); }
  doOtherThings();
}
```

Bonus info! Two ways to loop through indeterminate number of parameters

- Rest parameters convert any number of args into an array

```
function func(...rest) {
  for (var thing of rest) {
    console.log(thing);
  }
}
```

- The *arguments* object holds all the parameters passed in

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

To be OO, we need certain things

- ✔Objects
- ☐Classes
- ✔Properties
- ✔Methods
- ☐Constructors
- ☐Encapsulation
- ☐Private members
- ☐Public members
- ☐Static members
- ☐Accessor functions
- ✔Overloading
- ☐Inheritance
- ☐Overriding
- ☐Interfaces
- ☐Namespaces

tl;dr

- JavaScript may not have classes, but it does have objects
- Objects can be created on the fly like a hash
- The key-value pairs act like properties
- When we use functions as a value, it sure looks like a method.
- We can even create ersatz overloads when we remember that parameters are optional always

OO JavaScript – Created Objects

To be OO, we need certain things

- ✔Objects
- ☐Classes
- ✔Properties
- ✔Methods
- ☐Constructors
- ☐Encapsulation
- ☐Private members
- ☐Public members
- ☐Static members
- ☐Accessor functions
- ✔Overloading
- ☐Inheritance
- ☐Overriding
- ☐Interfaces
- ☐Namespaces

Did you notice that with JSON there's no need to write a class first?

Java

```
public class Car {
  private Engine _engine;
  public Engine getEngine()
  {
    return _engine;
  }
  public void setEngine(Engine engine)
  {
    _engine = engine;
  }
}
```

- Then later …

```
Car c = new Car();
c.setEngine(new Engine("Hemi"));
```

JavaScript

```
var c = {
    Engine: { type: "Hemi"}
}
console.log(c.Engine);
```

But I want classes!

- You can't have true classes
- JavaScript is not truly object-oriented
- It just looks like it

JavaScript's *new* operator works with functions

When you use *new*,

1. An empty Object is created
   - It replaces window as the context
   - *"this"* refers to that new context
2. The lines in the function are run in this context
3. That new context is returned
   - The object's type is its constructor function

For example:

```
function SuperHero() { ... }
var batman = new SuperHero();
```

A bad example of a function

```
var person = function (first, last, alias) {
  this.firstName = first;
  this.lastName = last;
  this.alias = alias;
  this.move = function (speed) { // Do stuff to move }
  this.attack = function (foe) {
    var d = `${this.alias} is attacking ${foe.alias}`;
    console.log(d);
    // Do other attacking stuff here
  }
}
// Run it
person("Ed", "Nigma", "Riddler");
attack(batgirl);
```

Thus, functions come in two flavors:

Some you run

- Traditional functions

```
function punch(t) {
  t.damage(10);
}
```

- Run it like this:

```
punch(riddler);
```

Some you make copies

- With the *new* operator

```
function Villain(name) {
  this.name=name;
  this.power=10;
  this.health=10;
}
```

- Run it like this

```
v = new Villain('Bane');
```

Let's look again at our example function

```
var Person = function (first, last, alias) {
  this.firstName = first;
  this.lastName = last;
  this.alias = alias;
  this.move = function (speed) { // Do stuff to move }
  this.attack = function (foe) {
    var d = `${this.alias} is attacking ${foe.alias}`;
    console.log(d);
    // Do other attacking stuff here
  }
}
var p = new Person("Oswald","Cobblepot","Penguin");
var n = new Person("Dick","Greyson","Nightwing");
p.attack(n); //Will log "Penguin is attacking Nightwing"
```

Each object knows it's constructor

```
function attack(otherPerson) {
  if (! (otherPerson instanceof Person)) {
    throw "That's not a person";
  }
// Do stuff here
}
```

Note: "of" is not camel-cased

Syntax:

- object instanceof ConstructorFunction
- Returns a bool

Encapsulation

Say you have this code ...

```
var c = new Person("James", "Gordon", "Commissioner");
var runTime = new Date();
function showInfo(person) {
  return `${person.alias} created at ${runTime}`;
}
alert(showInfo(c));
```

But we are using a library that does this

```
var runTime = programEnd – programStart;
```

- What happens to runTime?

We've polluted the global namespace

We can make it private

```
function buttonClick() {
  var c = new Person("James", "Gordon", "Commissioner");
  var runTime = new Date();
  function showInfo(person) {
    return person.alias + " created at " + runTime;
  }
  alert(showInfo(c));
}
buttonClick();
```

- Now runTime is local and a different variable than the library's copy

It is possible to have public variables and private variables in the same function

Private	Public
Put 'var' in front of the variable	Just omit the 'var'

`var x = 5;`
`var firstName = firstName;`
`var defend = function () {`
`  ...`
`};`
	`x = 5;  // Global! Bad idea`
	`this.firstName = firstName;`
	`this.defend = function () {`
	`  ...`
	`};`

Accessors

JavaScript 5 doesn't have accessors (getters/setters) either

Setters and Getters

But we can easily write them

```
var Person = function (first, last, alias) {
  this.firstName = first;
  this.lastName = last;
  this.alias = alias;
  this.move = function (speed) { // Do stuff to move }
  this.attack = function (foe) {
    var d = this.alias + " is attacking " + foe.alias;
    console.log(d);
    // Do other attacking stuff here
  }
  var hidden = 5;
  this.getHidden = function () { return hidden; }
  this.setHidden = function (newValue) {
    hidden = newValue;
  }
}
```

Static (Shared) Members

Static? What's *static*?

- "static" ==> Only need one copy of it.
- "shared" ==> Something they all share. Everyone has access to the same thing.
- An example in C#:

```
public class Person
{
  public static int NumberOfPeople { get; set; }
  public string FirstName { get; set; }
  ...
}
```

- To access:

```
int n = Person.NumberOfPeople;
```

Remember, in JavaScript ...

1. Functions are objects
2. Objects can have properties
3. Properties are added dynamically

- Therefore ...

```
function Person() {
  ...
}
var n = Person.numberOfPeople;
```

Interfaces

Yeah. JavaScript has no interfaces

ES2015 Classes

ES 2015

ES2015 gives us a different way to write this

```
class Person{
  attack() {
    // Do stuff in this method
  }
}
```

Note: methods don't need *this*. nor the function keyword

- Still not real classes, though!
- Just syntactic sugar on top of a constructor function

ES 2015

class is there to make OO devs feel better

ES2015 adds getter and setters

```
class Person {
  ...
  get alias() { return this._alias; }
  set alias(newValue) {
    this._alias = newValue;
  }
  move(speed) { // Do stuff to move }
  attack(foe) {
    let d = `${this._alias} is attacking ${foe.alias}`;
    // Do other attacking stuff here
  }
}
const p = new Person();
p.alias = "Joker";  // Calls set
console.log(p.alias); // Calls get
```

ES2015 adds formal constructors

```
class Person {
  constructor(first, last){
    this._first = first;
    this._last = last;
  }
  doStuff() {
    return `${this._first} ${this._last}
            doing stuff.`;
  }
}

const p = new Person("Talia", "Al-Ghoul");
console.log(p.doStuff());
```

ES2015 adds static members

```
class Person {
  constructor(first, last){
    this._first = first;
    this._last = last;
  }
  static doStuff() {
    return `Doing stuff.`;
  }
}

const p = new Person("Tim", "Drake");
console.log(Person.doStuff());
```

Classes can only have ...

```
class foo {
  constructor() { ...} // Constructor
  get x() { return this._x; } // Getters
  set x(v) { this._x = v; } // Setters
  method1() { ... } // Methods
  static doIt { ... } // Static methods
}
```

- Can **not** have "this.", "let", "var" that would be a syntax error

To be OO, we need certain things

- ✔Objects
- ✔Classes
- ✔Properties
- ✔Methods
- ✔Constructors
- ✔Encapsulation
- ✔Private members
- ✔Public members
- ✔Static members
- ✔Accessor functions
- ✔Overloading
- ☐Inheritance
- ☐Overriding
- ✖Interfaces
- ☐Namespaces

tl;dr

- Because of the way that JavaScript is being used today, we would like it to be object-oriented
- While JavaScript is not OO per-se, we can give it many of the features of a true OO language

OO JavaScript – Inheritance

To be OO, we need certain things

- ✔ Objects
- ✔ Classes
- ✔ Properties
- ✔ Methods
- ✔ Constructors
- ✔ Encapsulation
- ✔ Private members
- ✔ Public members
- ✔ Static members
- ✔ Accessor functions
- ✔ Overloading
- ☐ Inheritance
- ☐ Overriding
- ✖ Interfaces
- ☐ Namespaces

Inheritance

JavaScript does not support traditional inheritance

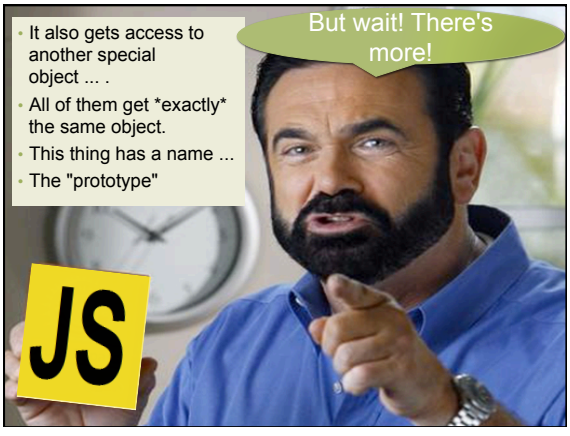
- In c#

```
public class Employee : Person
{
  public decimal Salary { get; set; }
  public overrides string ToString() {
    return string.Format("{0}, {1}",
      this.Last, this.First);
  }
}
```

- But it does have these things called *prototypes*

Reminder: constructor functions

- When you instantiate a function via the *new* operator, a context is created (*this*) and is passed back.
- This new object gets a copy of all the properties of the constructor function.



- It also gets access to another special object ... .
- All of them get *exactly* the same object.
- This thing has a name ...
- The "prototype"

prototype

/ˈprōdəˌtīp/

noun

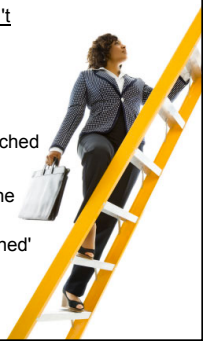
1. the original or model on which something is based or formed.
2. someone or something that serves to illustrate the typical qualities of a class; model; exemplar. ie: She is the prototype of a student activist.
3. something analogous to another thing of a later period. ie: a Renaissance prototype of our modern public housing.
4. Biology. an archetype; a primitive form regarded as the basis of a group.

From Random House Dictionary via Dictionary.com

The prototype serves as a 'backup' of sorts

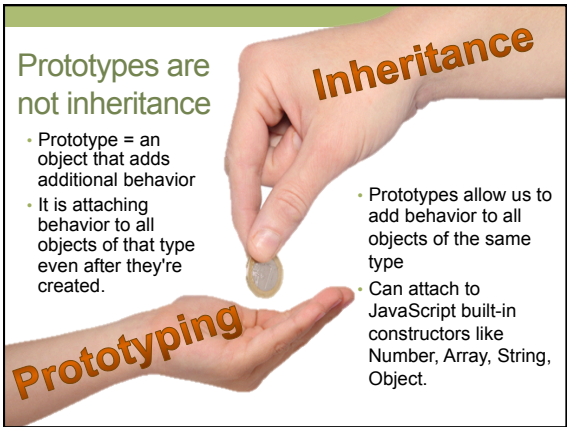
If we try to access a property which doesn't exist on the object itself,

1. JavaScript looks in its constructor's prototype for that property.
2. If it is present, that property is reported
3. If not, the prototype's prototype is searched
4. ... and so on and so on and so on
5. ... until JavaScript reaches the top of the chain
6. Then it reports that property as 'undefined'



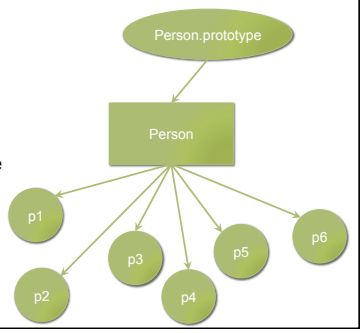
Prototypes are not inheritance

- Prototype = an object that adds additional behavior
- It is attaching behavior to all objects of that type even after they're created.



- Prototypes allow us to add behavior to all objects of the same type
- Can attach to JavaScript built-in constructors like Number, Array, String, Object.

- The object's prototype is never checked.
- Regular objects don't have a prototype, only Functions do.
- The object's constructor's prototype is checked.
- Thus you can add prototypes after instantiation and it is there.



The prototype on instances is read-only

```
var p = new Person();
console.log(p.firstname);
console.log(p.prototype.firstname);
p.firstname = "Joe";
p.prototype.firstname = "Joe";
p.__proto__.firstname = "Joe";
```

Person does not have a firstname, but its prototype does.

Works – pulls from the prototype

Does not work b/c objects don't have prototypes; Functions do

Works. Adds a firstname to p

Does not work. Can't set property 'firstname' of 'undefined'

Works, but really bad idea. Sets firstname for all instances. Breaks encapsulation

So you can see why this isn't exactly like traditional inheritance

Overriding

To override, simply set a value (property) or a function (method) in either the constructor or on the object itself after instantiation

Overridding is easy ...

```
function SuperHero(){ }
SuperHero.prototype.fly = function() {
  // Most heroes can fly
};
function HumanSuperHero() { ... }
HumanSuperHero.prototype = new SuperHero();
var batman = new HumanSuperHero();
batman.fly();
// Human superheroes can't fly! Let's override
HumanSuperHero.prototype.fly = function () {
  console.warn('This hero cannot fly');
};
batman.fly();
```

Most SuperHeroes can fly.

A HumanSuperHero is a SuperHero

Batman can fly?!? That's not right.

Better. Batman can't fly now.

ES2015 Classes

ES2015 Added some new keywords

- extends - Assigns a prototype
- super - Refers to the prototype

- Don't be tricked!
- These don't add any new capabilities. They're only more direct ways of working with prototypes.

extends registers a prototype

```
class Employee extends Person {
  constructor(first, last, salary) {
    super(first, last);
    this._salary = salary;
  }
}
const e = new Employee("Chris","Lee",75000);
console.log(e.doStuff()); // Chris Lee doing stuff
```

To be OO, we need certain things

- ✔Objects
- ✔Classes
- ✔Properties
- ✔Methods
- ✔Constructors
- ✔Encapsulation
- ✔Private members
- ✔Public members
- ✔Static members
- ✔Accessor functions
- ✔Overloading
- ? Inheritance
- ✔Overriding
- ✖Interfaces
- □Namespaces

tl;dr

- JavaScript doesn't handle traditional inheritance but it does have prototypal inheritance and that can be used to mimic it.
- Overriding can be done simply by setting the value in the object itself; you get it for free.

OO JavaScript – Design Patterns

A design pattern is …

- … a battle-tested solution to a known design problem.
- Allows for fast, reliable solutions.
- Allows quicker, more precise _____.
- Allows a higher level of _____.

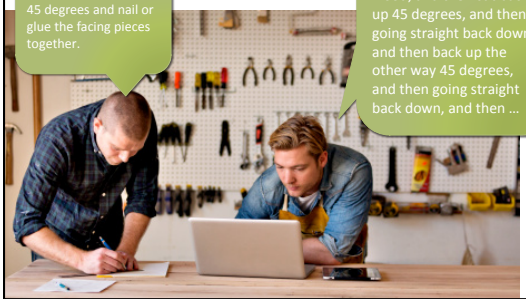
An example: Two carpenters making a drawer

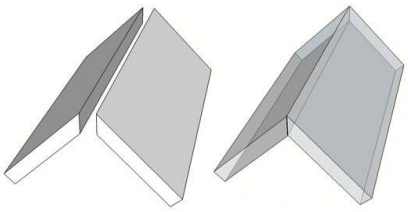
How should they join the drawer sides?

Two solutions

We could cut each at 45 degrees and nail or glue the facing pieces together.

Maybe we could make the joint by cutting straight down into the wood, and then cut back up 45 degrees, and then going straight back down, and then back up the other way 45 degrees, and then going straight back down, and then …



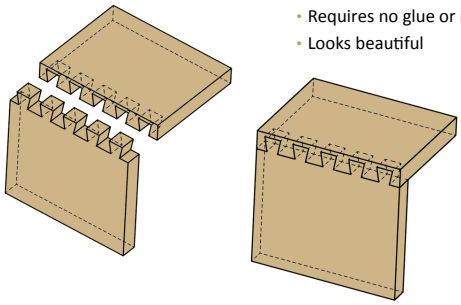


- Easy
- Quick
- Cheap
- Invisible

Miter Joint

Dovetail joint

- Very complicated!
- Time consuming
- Extremely strong
- Requires no glue or nails
- Looks beautiful

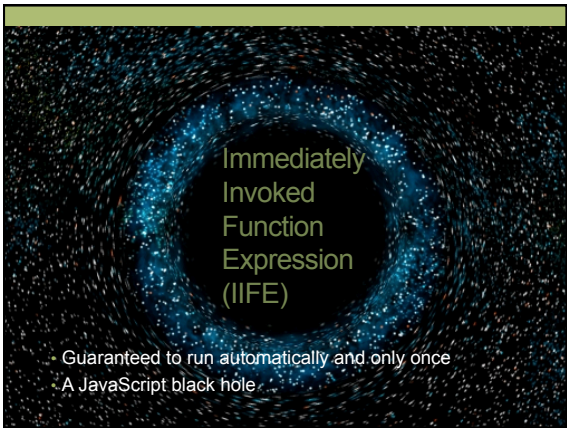


JavaScript Design Patterns

- We'll cover several well-used JavaScript design patterns
- Most are unique to the JavaScript community
- Since JavaScript makes it so easy to pollute the global namespace and since libraries are so popular, the biggest deal in all of these patterns is ...

Encapsulation

Immediately Invoked Function Expression



Quiz: What do these do?

```
var x = foo;
```

- Sets x equal to foo – even if foo is a function

```
foo();
```

- Runs the foo function
- So if you take a function and put parens after it, you're telling JavaScript to run that function

Quiz: What is this?

```
function () {
  // Do stuff here
}
```

- An anonymous function, of course
- How about this?

```
(function () {
  // Do stuff here
})
```

- Same thing

The payoff! The iife

- If you combine the anonymous function with parentheses, you have your iife:

```
(function () {
  // do stuff here
})()
```

- This says to define this function and then run it.

You can encapsulate some parts and expose others

```
(function () {
  Car = function (make) {
    this.make=make;
    this.go = function () {
      // Do stuff to make it 'go'
    }
  };
})();
var c = new Car('chevy');
c.go();
var p = new Car('porsche');
```

The Prototype Pattern

The prototype pattern

- Don't get too excited. This is simply saving memory and controlling behavior by sharing properties via the prototype.

Two forms ...

1. Object literal
2. Constructed

Object literal format

- If Person is the constructor function ...

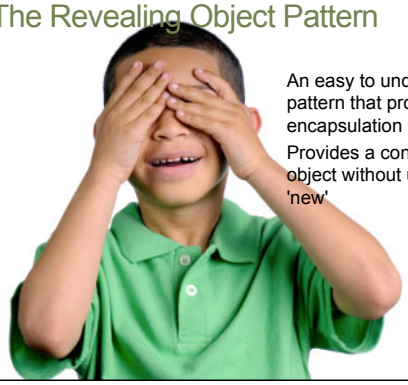
```
Person.prototype = {
    doSomething: function () {
        console.log('do something');
    },
    doSomethingElse: function () {
        console.log('do something else');
    }
};
```

Constructed format

```
// build our blueprint object
var MyBluePrint = function MyBluePrintObject() {
  this.someFunction = function someFunction() {
    alert( 'some function' );
  };
  this.someOtherFunction = function someOtherFunction() {
    alert( 'some other function' );
  };
  this.showMyName = function showMyName() {
    alert( this.name );
  };
};
function MyObject() {
  this.name = 'testing';
}
MyObject.prototype = new MyBluePrint();
```

The Revealing Object Pattern

The Revealing Object Pattern



An easy to understand pattern that provides encapsulation

Provides a controlled object without using 'new'

The revealing object pattern

```
var createWorker = function () {
  var workCount=0; //A private variable
  var t1 = function () {
    workCount++;
    console.log("task1 " + workCount);
  };
  var t2 = function () { ... } //Same as t1
  return {
    doWork1: t1,
    doWork2: t2
  };
};
var worker = createWorker();
worker.doWork1();
worker.doWork2();
```

The Self-attaching Object Pattern

The Self-attaching Object Pattern

- Whenever you have an object that needs to be attached to a namespace
- Allows you to keep the attaching completely modular and complete encapsulated.

Pass in the namespace

```
(function (ns) {
 var bar = {
    p1: someValue,
    f1: function() { ... }
  }
  ns.foo = bar;
})(duff.products.ales);
```

... then later ...

```
var x = duff.products.ales.foo.p1;
duff.products.ales.foo.f1();
```

tl;dr

- We looked at some very common patterns unique to JavaScript
- They help us to overcome the limitations of this non-object-oriented language

Further Study

- Large listing of more JavaScript design patterns
  - http://bit.ly/SimplifiedJSPatterns/
- Free e-Book from O'Reilly
  - http://bit.ly/JavaScriptDesignPatternsEBook
- Understanding GOF patterns in JavaScript
  - http://bit.ly/UnderstandingDesignPatternsInJavaScript