

Slides	Comment
2 - 8	Background on JavaScript the language and the DOM JavaScript interacts with the DOM to create interactive Web apps
9 - 13	A very quick summary/checklist of JavaScript for Java programmers listing similarities and important differences. I'll quickly go over this in the pre-quiz chat in the week 3 workshop.
15 - 36	These slides deal with important and unique JavaScript features. You should go over these slides very carefully and attempt to understand them fully. Since JavaScript is such a popular language you can find multiple explanations on the Web of almost any issue you have. People new to JavaScript often have the same issues so they have been well documented and discussed.

## FIT3083 - eBusiness Technologies

### Week 3

- Summary of Codecademy JavaScript Course
- Some More Advanced Topics

# JavaScript

JavaScript is an implementation of the ECMAScript standard. This standard has an interesting history involving infighting between various factions on the committee. There have been 8 main versions. The last three are now called ECMAScript2015 (6.0), 2016 (7.0) and 2017 (8.0)

2

- JavaScript is a high-level, dynamic, weakly typed, object-based, multi-paradigm, and interpreted programming language.
- Alongside HTML and CSS, JavaScript is one of the three core technologies of World Wide Web content production. The majority of websites employ it, and all modern web browsers support it without the need for plug-ins by means of a built-in JavaScript engine.
  - Each of the many JavaScript engines represent a different implementation of JavaScript, all based on the **ECMAScript specification**, with some engines not supporting the spec fully, and with many engines supporting additional features beyond ECMA.
- “JavaScript (JS) is a dynamic computer programming language. It is most commonly used as part of web browsers, whose implementations allow client-side scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed.”

<http://en.wikipedia.org/wiki/JavaScript> (earlier version since superceded)

# JavaScript

<http://en.wikipedia.org/wiki/JavaScript> (references removed)

3

- As a multi-paradigm language, JavaScript supports event-driven, functional, and imperative (including object-oriented and prototype-based) programming styles. It has an API for working with text, arrays, dates, regular expressions, and basic manipulation of the DOM, but does not include any I/O, such as networking, storage, or graphics facilities, relying for these upon the host environment in which it is embedded.
- Initially only implemented client-side in web browsers, JavaScript engines are now embedded in many other types of host software, including server-side in web servers and databases, and in non-web programs such as word processors and PDF software, and in runtime environments that make JavaScript available for writing mobile and desktop applications, including desktop widgets.
- Although there are strong outward similarities between JavaScript and Java, including language name, syntax, and respective standard libraries, the two languages are distinct and differ greatly in design; JavaScript was influenced by programming languages such as Self and Scheme.

# JavaScript

<http://en.wikipedia.org/wiki/JavaScript>  
(references removed, earlier version since superceeded)

4

- "JavaScript is a prototype-based scripting language with dynamic typing and has first-class functions. Its syntax was influenced by C. JavaScript copies many names and naming conventions from Java, but the two languages are otherwise unrelated and have very different semantics. The key design principles within JavaScript are taken from the Self and Scheme programming languages. It is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.
- The application of JavaScript in use outside of web pages—for example, in PDF documents, site-specific browsers, and desktop widgets—is also significant. Newer and faster JavaScript VMs and platforms built upon them (notably Node.js) have also increased the popularity of JavaScript for server-side web applications. On the client side, JavaScript was traditionally implemented as an interpreted language but just-in-time compilation is now performed by recent (post-2012) browsers.
- JavaScript was formalized in the ECMAScript language standard and is primarily used as part of a web browser (client-side JavaScript). This enables programmatic access to objects within a host environment."

# DOM (Document Object Model)

5

- The Document Object Model (**DOM**) is an API defined to “represent and interact with any HTML or XML document.”
  - In Depth <https://developer.mozilla.org/en-US/docs/Glossary/DOM>
    - "The DOM (Document Object Model) is an API that represents and interacts with any HTML or XML document. The DOM is a document model loaded in the browser and representing the document as a node tree, where each node represents part of the document (e.g. an element, text string, or comment).
    - The DOM is one of the most-used [APIs](#) on the [Web](#) because it allows code running in a browser to access and interact with every node in the document. Nodes can be created, moved and changed. Event listeners can be added to nodes and triggered on occurrence of a given event.
    - DOM was not originally specified—it came about when browsers began implementing [JavaScript](#). This legacy DOM is sometimes called DOM 0. Today, the WHATWG maintains the DOM Living Standard."
- "The principal standardization of DOM was handled by the W3C, which last developed a recommendation in 2004. WHATWG took over development of the standard, publishing it as a living document. The W3C now publishes stable snapshots of the WHATWG standard." [https://en.wikipedia.org/wiki/Document\\_Object\\_Model](https://en.wikipedia.org/wiki/Document_Object_Model)

# Going Forward

6

- To code “client-side [JavaScript] scripts to interact with the user, control the browser, communicate asynchronously, and alter the document content that is displayed” you do not need to know much about JavaScript objects and you need know nothing about prototypes or prototype chains etc.
  - You just need to be able to code basic control structures in functions and know how to trigger these event handling functions in response to events
  - Of course knowledge of the DOM object and event model is also necessary
  - i.e. you need to know how to use JavaScript to manipulate the DOM and how to make JavaScript functions execute when a DOM event occurs
- But JavaScript is becoming more than just a DOM manipulator/responder
  - Beyond the Web
  - And within the Web
- In this unit we will take the opportunity to examine this unique language in more detail and build a client side application using some of JavaScript’s more advanced features


# JavaScript – A Warning (for Java programmers)

7

- Warnings Cameron, Dane (2013-10-30). A Software Engineer Learns HTML5, JavaScript and jQuery (my emphases)
  - “JavaScript is a particularly flexible language, and does not enforce much discipline or structure on its users. In addition, JavaScript contains a number of features that can only truly be regarded as bugs. These remain in the language principally due to backwards compatibility concerns.”
  - “JavaScript also supports objects; in fact, most values in JavaScript applications will be objects. JavaScript also supports syntax for defining classes that objects can be instantiated from. This may lead you to think JavaScript is a conventional object orientated language – **this would be a mistake.**”
  - JavaScript has a **far more flexible** attitude to classes and objects [than Java], in fact classes are not essential to JavaScript programming at all.
- Important JavaScript References
  - <https://developer.mozilla.org/en/docs/Web/JavaScript>
    - Slightly formal, lots of examples and tutorials
  - <http://www.ecma-international.org/ecma-262/>
    - Strictly formal language description

# JavaScript Consoles

8

- Chrome (and Codecademy) have JavaScript consoles
  - In Chrome ctrl + shift + I (OSX: cmd + option + I) then console panel or console drawer
- Entering Single and Multiple code lines is possible
  - To prevent immediate single line execution use shift + enter
- Recovering previous entries (including multiple line entries)
  - Up arrow
- JavaScript consoles in general Output AND Echo
  - In addition to output via the console.log(...) function consoles Echo the value of the last expression they receive
    - If the expression does not evaluate to a value “undefined” is echoed
      - This is not an error just a report of an undefined value
    - Output in Chrome has no prompt character on the command line, echo has this 
- A console
  - Can be used for experimentation and checking syntax and the values of expressions etc.



# Summary

9

\* Can be used as debug tool outside of the console in real code???

\*\* == and != are valid operators. What do they mean???

\*\*\* Things can go horribly wrong for Java programmers!!!

- Summary (for Java programmers)

- Length property (not method) of String objects, arithmetic operators, line and multiline comments, confirm and prompt browser window pop up i/o dialogues
- Data types (Number, Strings and Boolean), Number, String and Boolean literals
  - But no static data typing
- `console.log(...)`\*
- Equality operators (`===` and `!==`)\*\*, relational operators
- `if ... statements, if ... else ... statements`\*\*\*
- String object substring method
- Variables
  - Declaration/initialisation syntax, assignment

Oh the horror!

```
var i = 7;  
if (i = 10)  
    console.log("true");
```

Codecademy

true

Console output

# Summary

\* This is much deeper than it looks. In JavaScript functions are **first class** which means they can be assigned to variables and therefore passed as parameters to other functions — this is the genesis of a very powerful (albeit dense and cryptic) programming paradigm called Functional Programming

10

- Summary (for Java programmers)

- Functions

- No typing of return value (because there is no static data typing)
    - Functions can be assigned to variables! Who knew!\*
    - Function definition, call, parameter passing and return values should be familiar from Java methods (but what Class do these JavaScript functions belong to?)
      - Exception: how to call an anonymous function assigned to a variable?
        - » Just use the variable name as if it were the function name i.e. varFunc()
      - Exception: At time of call the wrong number or type of actual parameters will not cause an error!!!
        - » What to do if too many actuals or too few? What to do about incorrect types
        - » The “arguments” array accessible within a function always contains all actual argument values

# Summary

11

- Summary (for Java programmers)

- Functions

```
var f = function(n1, n2){return n1 + n2};  
console.log(f(1, 2));  
  
var f2 = function add(n1, n2){return n1 + n2};  
console.log(f2(1, 2));  
//console.log(add(1, 2)); //error: add is not defined  
  
function add2(n1, n2){return n1 + n2}  
console.log(add2(1, 2));
```

RHS is an anonymous function expression i.e. it evaluates to a function which is assigned to the variable f. To execute the function use f(...)

RHS is a named function expression i.e. it evaluates to a function which is assigned to the variable f2. To execute the function use f2(...)  
Note RHS is function expression not a function definition so add(...) will cause an error.

Definition of a function called add. To call use add(...). This is more like what we are used to as Java programmers.

# Summary

\* It's possible in JS to use a variable without declaring it i.e. the left hand use of a variable auto declares it (i.e. you need to be setting its value NOT getting it obviously)  
DON'T DO THIS! Auto declared variables are Global no matter where they first appear (even within a function)

12

- Summary (for Java programmers)

- Variable scope\*

- Global (page-level): declare with var outside of a function
    - Local (function-level): declare with var inside a function

- for loops

- Similar to Java

- Arrays

- Many similarities to Java
    - Initialiser lists like Java but [..., ..., ] not {..., ..., }
    - console.log(...) outputs arrays formatted nicely
    - Heterogeneous arrays (not allowed in Java except polymorphic pointing at child class objects)
    - Array object property length = number of elements in the array (iteration limit = length - 1)
    - Array object method **push(elem)** to append elem to the end of the array

JS multidimensional arrays are, as in Java, arrays of arrays thus ragged arrays are possible in both languages. Both languages use the same cumbersome syntax e.g. a[2][3]

- Wrapping String literals across lines with “\”

- Weird!

Unlike Java arrays but like Java ArrayLists from the Java API JS arrays are dynamically resizable

# Summary

13

- Summary (for Java programmers)
  - while loops, do while loops
    - Similar to Java
  - Random number generation
    - Same as Java's Math.random()
  - Nesting if ... else ... in an outer if's else
    - Multiway selection – same as Java
  - isNaN(...) function
    - Returns true if the parameter evaluates to the value NaN else returns false if it does not
    - If the parameter is not a number its coerced to a number then tested to see if it's NaN
  - Number(...)
    - Returns parameter as a number if it can convert it (e.g. "123" can be converted) or the value NaN if not
  - switch statement
    - Same as Java
  - Logical operators (||, &&, !)
    - Same as Java

# Objects

14

- Unlike Java
  - Don't need classes to make objects
  - Data and methods (functions) are added to objects whenever we like
    - Note adding methods (functions) leverages JS functions' first class status
  - Data and methods are not protected (no encapsulation)
- Like Java
  - Data and the methods that process that data can be grouped (but not encapsulated and therefore protected, more on this later though)
- A simple and useful way of thinking about JS objects
  - key/value pairs (values can be data or functions (methods))
  - Often referred to as an associative array (associating keys and values)
  - Like a Java Map

# Objects

- Syntax to create an object

- Using **o**bject literal notation

- Using an **O**bject constructor

- new operator (similar to Java)
- Object() – Object constructor

- Do not be tempted to think this is the constructor of something like Java's topmost Class that all other classes inherit from
- Object is a function (NOT a Class) and btw all functions in JavaScript are objects
- A function becomes a constructor when it is used with the new keyword/operator
- This is tricky but important, more on it later

- Using Object.create(...)

- This method of creating an object requires a knowledge of prototype (a fundamental concept in advanced JavaScript)
- We will learn about prototypes soon
- This method of creating objects is included for completeness here

```
1 var me = {  
2     name: "Someone",  
3     age: 30  
4 };  
5  
6 var emptyObj = {};
```

```
1 var me = new Object();  
2 me.name = "Someone";  
3 me["age"] = 30;
```

# Objects

11 9/30

Codecademy

```
var list = function(obj) {  
  for(var prop in obj) {  
    console.log(prop);  
  }  
};
```

## ADVANCED CAVEATS

- Iteration order not guaranteed.
- Will chase up prototype chain
  - So not just own properties)
- Only selects enumerable properties
  - Which is pretty much what you would want
- Problems with arrays (use standard for loop)
  - See e.g. stackoverflow for details

16

- for ... in ...

- Iterating over every property name in an object

- **for (var someVar in someObject) {...}**

- someVar: refers to the current property name for this iteration
      - Use this in the loop body to access the current property name (use someVar) or property value (use someObject[someVar]) of each iteration
    - someObject: refers to the object which is having its properties iterated over

- Accessing property values

- Dot notation: **someObj.somePropertyName**

- Can only be used with property names that obey JS variable naming rules

- Bracket notation: **someObj["somePropertyName"]**

- Can be used for all property names (any JS String is valid !!!)
    - Also required if property name is a variable i.e. within a for ... in ... (no quotes of course)



# Analyse This!

- Some Issues

- Parameters are not data typed
  - So things can go very wrong
- What object do the functions belong to?
  - Search function assumes friends is accessible
- Objects can be passed as parameters and returned as return values
- friends[prop].firstName
  - ✓ • friends[prop]["firstName"]
  - ✗ • friends.prop.firstName

prop is a variable

Codecademy

```
1  var friends = {};  
2  friends.bill = {  
3    firstName: "Bill",  
4    lastName: "Gates",  
5    number: "(206) 555-5555",  
6    address: ['One Microsoft Way', 'Redmond', 'WA', '98052']  
7  };  
8  friends.steve = {  
9    firstName: "Steve",  
10   lastName: "Jobs",  
11   number: "(408) 555-5555",  
12   address: ['1 Infinite Loop', 'Cupertino', 'CA', '95014']  
13 };  
14  
15 var list = function(obj) {  
16   for(var prop in obj) {  
17     console.log(prop);  
18   }  
19 };  
20  
21 var search = function(name) {  
22   for(var prop in friends) {  
23     if(friends[prop].firstName === name) {  
24       console.log(friends[prop]);  
25       return friends[prop];  
26     }  
27   }  
28 };  
29  
30 list(friends);  
31 search("Steve");
```

Output all property names of obj

Each friends[prop] is an object.  
friends[prop].firstName is the value  
of the property firstName of the  
object friends[prop]

This will be output to console

Not used but will be echoed to console when search called

# Methods

18

- functions

- Can be defined anonymously and assigned to a variable OR
  - This is because JS has first class functions
  - This also means we can add a function variable to an object
    - Such a function becomes a **method** of the object
- Defined with a name and not assigned to a variable

Both methods and functions may or may not return a value depending on whether or not they contain a **return returnVal** statement

- Calling (invoking) functions and methods

- Functions: **someFunc (parameterList)**
- Methods: **someObject.someMethod (parameterList)**

Where someMethod must have been added to someObject

- Keyword this

- Within a method it references the object the method is invoked on
  - Can be used to access the object's property values (**this.somePropertyName**)

# Methods - Analyse

19

Codecademy

```
var setAge = function (newAge) {  
  this.age = newAge;  
};  
  
var bob = new Object();  
bob.age = 30;  
bob.setAge = setAge;  
  
var susan = new Object();  
susan.age = 25;  
susan.setAge = setAge;  
  
bob.setAge(50);  
susan.setAge(35);  
  
console.log(bob);  
console.log(susan);
```

| 17/33

This bob AND susan setAge method takes a parameter (the new age) but returns nothing

Codecademy

```
var square = new Object();  
square.sideLength = 6;  
square.calcPerimeter = function() {  
  return this.sideLength * 4;  
};  
  
square.calcArea = function(){  
  return this.sideLength * this.sideLength;  
}  
  
var p = square.calcPerimeter();  
var a = square.calcArea();
```

| 19/33

Both these square methods take no parameters but return a value

# Constructors

The keyword `new` is what makes a function a Constructor (nothing else).

If you forget it JS doesn't mind but the `this` keyword within the function now refers to something called the global object (rather than the object being constructed) which you are now adding properties (data and methods) to. This causes difficult to find bugs and is a classic JavaScript newbie error.

20

- `new Object()`
  - Creates and Constructs an object with the minimum (common to all objects) properties and methods (e.g. `.toString()`). How?
- We can code Custom Constructors (`new MyConstructor(...)`)
  - They are just regular functions but:
    - They use the keyword `this` to assign properties (and set their initial property values) and functions to the object under construction
    - Their parameter values are generally used to initialise object properties
    - Therefore if you construct multiple objects with a custom constructor they will all have the same properties (data and methods) at least initially but different property values according to the parameter values passed to the constructor — sound familiar?
    - Constructor names should begin with an upper-case letter to stylistically distinguish them from functions not intended to be Constructors

# Constructors -Analyse

21

```
1 function Rectangle(height, width) {
2   this.height = height;
3   this.width = width;
4   this.calcArea = function() {
5     return this.height * this.width;
6   };
7   // put our perimeter function here!
8   this.calcPerimeter = function() {
9     return this.height * 2 + this.width * 2;
10  };
11 }
12
13 var rex = new Rectangle(7,3);
14 var area = rex.calcArea();
15 var perimeter = rex.calcPerimeter();
16 console.log(area);
17 console.log(perimeter);
```

Codecademy

| 24/33

Note: not all initial property values need to be supplied by constructor parameter values.

```
function Person(name,age) {
  this.name = name;
  this.age = age;
  this.species = "Homo Sapiens";
}

var sally = new Person("Sally Bowles", 39);
var holden = new Person("Holden Caulfield", 16);
console.log("sally's species is " + sally.species + " and she is " + sally.age);
console.log("holden's species is " + holden.species + " and he is " + holden.age);
```

Codecademy

| 23/33

# More

22

- Arrays of object

```
// Our person constructor
function Person (name, age) {
    this.name = name;
    this.age = age;
} // Our Person constructor

// Now we can make an array of people
var family = [];
family[0] = new Person("alice", 40);
family[1] = new Person("bob", 42);
family[2] = new Person("michelle", 8);
family[3] = new Person("timmy", 6);

// loop through our new array
for (var i = 0; i < family.length; i++){
    console.log(family[i].name);
}

console.log("done");
```

Codecademy

```
alice
bob
michelle
timmy
done
```

| 27/33



# More

- Objects as parameters
- Functions/Methods
  - 1 constructor
  - 2 functions
  - 0 methods
- Oldest
  - Note the possible return values don't even have the same type

```
// Our person constructor
function Person (name, age) {
    this.name = name;
    this.age = age;
}

// Compute the difference in ages between two people
var ageDifference = function(person1, person2) {
    return person1.age - person2.age;
}

// determine the older
var oldest = function(person1, person2) {
    var returnVal;

    if (person1.age > person2.age)
        returnVal = person1.name;
    else if (person2.age > person1.age)
        returnVal = person2.name
    else
        returnVal = "Same Age"

    return returnVal;
}

var alice = new Person("Alice", 30);
var billy = new Person("Billy", 25);

var diff = ageDifference(alice, billy);
var older = oldest(alice, billy);

console.log(diff);
console.log(older);
```

5  
Alice

23

# Analyse

24

- 2 objects
- 0 methods
- 1 array containing the 2 objects
- 2 named functions
- 2 anonymous functions referenced by variables
- 1 anonymous object
- 1 function called by 2 others

```
var bob = {
  firstName: "Bob",
  lastName: "Jones",
  phoneNumber: "(650) 777-7777",
  email: "bob.jones@example.com"
};

var mary = {
  firstName: "Mary",
  lastName: "Johnson",
  phoneNumber: "(650) 888-8888",
  email: "mary.johnson@example.com"
};

var contacts = [bob, mary];

function printPerson(person) {
  console.log(person.firstName + " " + person.lastName);
}
```

```
function list() {
  var contactsLength = contacts.length;
  for (var i = 0; i < contactsLength; i++) {
    printPerson(contacts[i]);
  }
}

var search = function(lastName){
  var contactsLength = contacts.length;
  for (var i = 0; i < contactsLength; i++) {
    if (lastName === contacts[i].lastName)
      printPerson(contacts[i]);
  }
}

var add = function(firstName, lastName, email, phoneNumber){
  contacts[contacts.length] = {
    firstName: firstName,
    lastName: lastName,
    phoneNumber: phoneNumber,
    email: email
  };
}

add("oneF", "twoF", "a.b@c.com", "4444 4444");

list();
```

for ... in ... has  
issues with arrays

could we push?

Append anonymous  
object to array

Bob Jones  
Mary Johnson  
oneF twoF

Address Book 6/6



# typeof Operator, hasOwnProperty() method

- typeof operator

- Syntax

- **typeof anything**
    - Evaluates to 'number', 'string', 'boolean', 'function', 'object', ...
    - There is some JS craziness here
      - e.g. **typeof NaN** evaluates to 'number'

- hasOwnProperty(...) method

- Inherited by all objects from Object.prototype

- Syntax

- someObj.hasOwnProperty('somePropertyName')
    - Returns true if someObj has a direct property with the name somePropertyName
      - Otherwise returns false
      - The property must be a direct property of the object NOT AN INHERITED PROPERTY
        - » So hasOwnProperty('toString') will return false for all objects except Object.prototype and objects that override this method

You can use the hasOwnProperty method on any object (how come?) to determine if it has a specified property (remember functions can be properties too).

We are about to cover prototypes and inheritance so come back here after we have done that to fully understand the hasOwnProperty method

# Warning

26

- Any talk of Class in Codecademy should be treated with care
  - Wherever you see class replace it with (depending on the context):
    - prototype object used by the constructor function OR
    - Constructor function
- JavaScript has a keyword “class”
  - It is implemented in ECMAScript 6 (JavaScript 1.8 commonly called JavaScript 8)
    - As syntactic sugar
    - The JavaScript community is very much split on the introduction of the class keyword
      - It’s trying to fit a class based inheritance syntax on a prototype based inheritance language
      - The rise of TypeScript (a Java-like language that compiles to JavaScript) is possibly a much better way of bringing classes to Web application programming
      - Either way the need for some type of code organisation that can cope with large applications is very much required as Web applications increasingly dominate application development

# Prototypes

27

- Prototypes

- Every object references a prototype which is itself an object
  - Exception: `Object.prototype` has null as its prototype
  - The reference is created at the time the object is created
    - The details depend on how the object is created (see following slides)
- Properties and methods added to an object post creation are added to the object but NOT its prototype
  - They are the objects own properties discoverable using the `hasOwnProperty` method



- Prototype Chains

- An object's prototype object will itself have a prototype etc. thus chains of prototypes can be formed
- All chains terminate at `Object.prototype` which has null as its prototype

# Prototype

28

- Accessing an object's prototype

- `someObject.__proto__`  Not recommended (most browsers understand it)
  - Is a reference to `someObject`'s prototype
  - But it's not part of ECMAScript 5 (formalised in ECMAScript 6 for backward compatibility)
- `Object.getPrototypeOf(someObject)`  Recommended
  - Returns a reference to `someObject`'s prototype
- Once you have got a reference to a prototype you can add properties including methods as required just like any other object (except the consequences usually are more far reaching as they involve all objects on all connected chains below it)

- `someFunctionObject.prototype`

- Does not give access to the function's prototype but the far more useful prototype for objects the function constructs (if it is a constructor i.e. if it used with `new`)
  - i.e. objects created using `new` and `someFunctionObject` will have their prototype set to `someFunctionObject.prototype`

# Prototype Chains and Inheritance

29

- When an attempt to access a property including invoking a method (a function property) on an object is made:
  - The property is first searched for in the object then its prototype then the prototype's prototype etc.
  - If null (Object.prototype's prototype) is reached without finding the property a Reference Error occurs
- This enables inheritance in JavaScript
  - e.g. if you create an object using literal object notation the automatically assigned prototype is Object.prototype which includes several fundamental properties/methods that are available to the newly created object via inheritance (via its prototype chain)

- Object is actually a function (functions are objects in JavaScript).
- Like all functions when combined with "new" Object becomes a constructor.
- Like all function objects Object has a property called prototype (not \_\_proto\_\_) which references the object that will be set as the prototype for all objects instantiated when it is used as a constructor.
- In this case that object is Object.prototype.
- Object.prototype's prototype is Null (i.e. the end of all prototype chains)

# Prototype Depends on Creation Mechanism

30

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance\\_and\\_the\\_prototype\\_chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain)

- Creation with syntax constructs

- Literal object notation (**var o = { ... };**)
  - $o \rightarrow \text{Object.prototype} \rightarrow \text{null}$
- Array objects (**var a = [elem1, elem2, ...];**)
  - $a \rightarrow \text{Array.prototype} \rightarrow \text{Object.prototype} \rightarrow \text{null}$
- Function objects (**function f(){ ... };**)
  - $f \rightarrow \text{Function.prototype} \rightarrow \text{Object.prototype} \rightarrow \text{null}$

→ Prototype reference

- Creation with **Object.create(...)**

- “ECMAScript 5 introduced a new method: **Object.create(...)**. Calling this method creates a new object. The prototype of this object is the first argument of the function”
  - The second optional argument (which has special syntax) allows additional properties/methods to be added to the created object (but not its prototype)
- **var a = {...};**  
**var b = Object.create(a);**
  - $b \rightarrow a \rightarrow \text{Object.prototype} \rightarrow \text{null}$

# Prototype Depends on Creation Mechanism

31

- Creation with a Constructor

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance\\_and\\_the\\_prototype\\_chain](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Inheritance_and_the_prototype_chain)

- ‘A "constructor" in JavaScript is "just" a function that happens to be called with the new operator.’ e.g. `new Object()`, `new MyConstructor(...)`
- The function **prototype** property <http://sporto.github.io/blog/2013/02/22/a-plain-english-guide-to-javascript-prototypes/>
  - “Every function in JavaScript has a special property called ‘**prototype**’.
  - As confusing as it may sound, this ‘**prototype**’ property is not the real prototype [...] of the function [which is `Function.prototype`].
  - This of course generates a lot of confusion as people use the term ‘**prototype**’ to refer to different things. I think that a good clarification is to always refer to the special ‘**prototype**’ property of functions as ‘the function **prototype**’, never just ‘**prototype**’.
  - The ‘**prototype**’ property points to the object that will be assigned as the prototype of instances created with that function when using ‘**new**’” i.e. when it’s used as a constructor
    - The default value of the prototype property is an object with 2 properties
      - » `__proto__` set to `Object` → `Object.prototype` → `null` (i.e. end of chain)
      - » `constructor` set back to reference the constructor

Important Convention: begin Constructor names with an upper case letter so we will not forget to use them with “new”



## Prototypes - An Example

```
// a constructor function
function Foo(y) {
  // which may create objects
  // by specified pattern: they have after
  // creation own "y" property
  this.y = y; 1
}

// also "Foo.prototype" stores reference
// to the prototype of newly created objects,
// so we may use it to define shared/inherited
// properties or methods, so the same as in
// previous example we have:

// inherited property "x"
Foo.prototype.x = 10; 2

// and inherited method "calculate"
Foo.prototype.calculate = function (z) {
  return this.x + this.y + z;
}; 2

// now create our "b" and "c"
// objects using "pattern" Foo
var b = new Foo(20);
var c = new Foo(30);

// call the inherited method
b.calculate(30); // 60
c.calculate(40); // 80

// let's show that we reference
// properties we expect

console.log(

  b.__proto__ === Foo.prototype, // true
  c.__proto__ === Foo.prototype, // true

  // also "Foo.prototype" automatically creates
  // a special property "constructor", which is
  // reference to the constructor function itself
  // instances "b" and "c" may find it via
  // delegation and use to check their constructor

  b.constructor === Foo, // true
  c.constructor === Foo, // true
  Foo.prototype.constructor === Foo, // true

  b.calculate === b.__proto__.calculate, // true
  b.__proto__.calculate === Foo.prototype.calculate // true
);
```

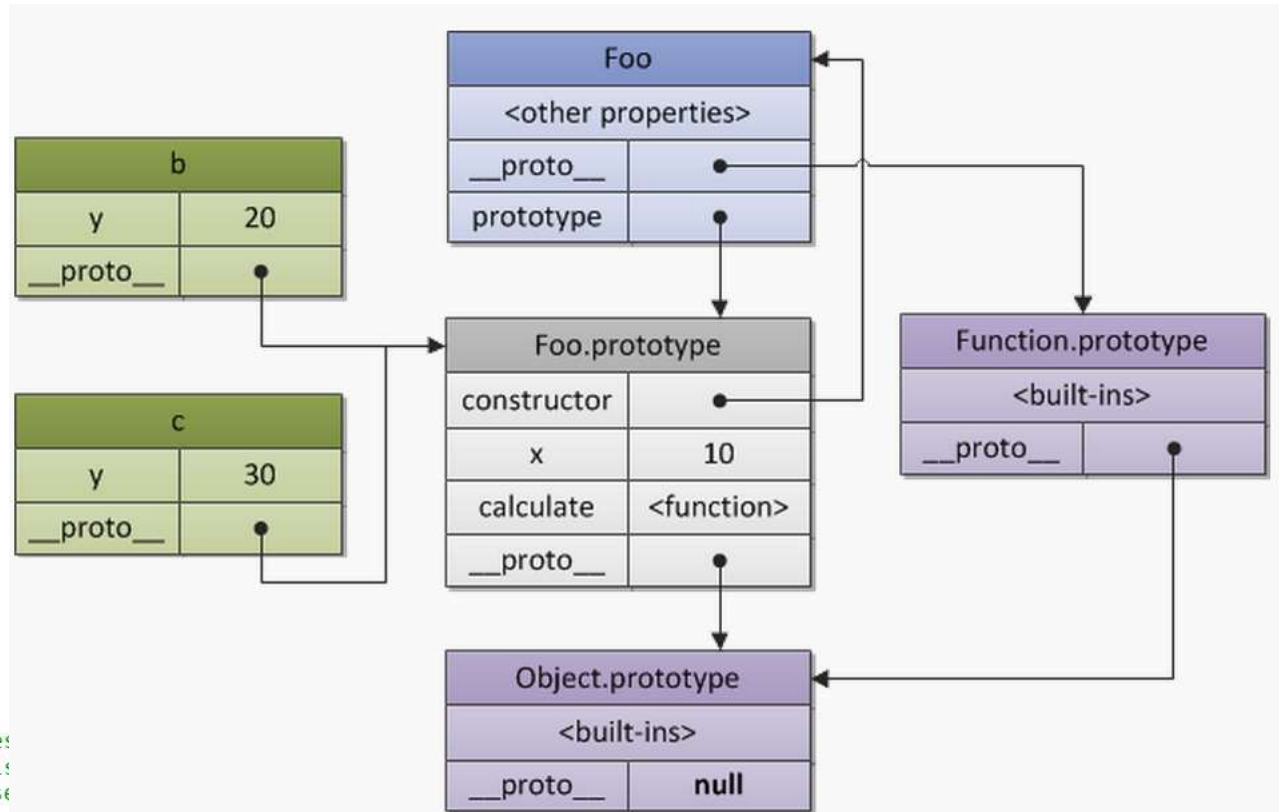


Figure 3. A constructor and objects relationship.

Every object constructed with the `Foo` constructor function will have

1. An own property `y`
2. Inherited properties `x` (data) and `calculate` (method)



# Prototypes - More

33

- Dynamic Inheritance

- If you add a property (data or method) to an object
  - You only modify that object
  - If you add a property with the same name as one in the object's prototype chain it will shadow (override) the property in the prototype chain
    - It's actually **replacement** rather than shadowing so you cannot call super as you would in Java unless you save a reference to the parent property before replacing it
- If you add a property (data or method) to a prototype
  - The property is **immediately** available to all objects that reference that prototype somewhere in their prototype chain
    - e.g. through a construction prototype chain
    - e.g. through `Object.create(...)`

Immediately NOT from now on if they are created and have the prototype in their prototype chain!!!

# Analyse

Codecademy

```
function Dog (breed) {  
  this.breed = breed;  
}  
  
// here we make buddy and teach him how to bark  
var buddy = new Dog("Golden Retriever");  
buddy.bark = function() {  
  console.log("Woof");  
};  
buddy.bark();  
  
// here we make snoopy  
var snoopy = new Dog("Beagle");  
  
// we need you to teach snoopy how to bark here  
//snoopy.bark = function() {  
//  console.log("Woof");  
//};  
  
// this causes an error, because snoopy doesn't know how to bark!  
snoopy.bark();
```

Woof  
TypeError: undefined is not a function

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Object/proto](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/proto)

The reference above strongly warns against resetting an object's prototype to another object due to very slow performance of the actual change and the far reaching (probably bad) effects of such a change on any objects downstream in the object's prototype upside down tree of prototype chains.

On this slide we update a prototype but do not reset it.

```
function Dog (breed) {  
  this.breed = breed;  
};  
  
// here we make buddy  
var buddy = new Dog("golden Retriever");  
// here we teach all dogs to bark  
Dog.prototype.bark = function() {  
  console.log("Woof");  
};  
buddy.bark();  
  
// here we make snoopy  
var snoopy = new Dog("Beagle");  
/// this time it works!  
snoopy.bark();
```

Woof  
Woof

This is not the prototype of the Dog function object (which is `Function.prototype`) but the prototype set for new objects created using the Dog constructor (i.e. when using it with `new`).

See previous slides.

# Analyse – Prototype Chains & Inheritance

35

Codecademy

actually  
replacement  
not overriding

overriding?

overriding?

overriding?

where does name come from?

where does numLegs come from?

```
// original classes
function Animal(name, numLegs) {
  this.name = name; //try this.name = name + 'x'
  this.numLegs = numLegs;
  this.isAlive = true;
}

function Penguin(name) {
  this.name = name; //try this.name = name + 'x'
  this.numLegs = 2;
}

function Emperor(name) {
  this.name = name;
  this.saying = "Waddle waddle";
}

// set up the prototype chain
Penguin.prototype = new Animal();
Emperor.prototype = new Penguin();

var myEmperor = new Emperor("Jules");

console.log(myEmperor.name); //should print "Jules"
console.log(myEmperor.saying); //should print "Waddle waddle"
console.log(myEmperor.numLegs ); //should print 2
console.log(myEmperor.isAlive); //should print true
```

Jules  
Waddle waddle  
2  
true

Note: we are not resetting any object's prototype here.

II 21/30

We are resetting a function object's prototype property which is not the same thing. See previous slides if you don't get this

# Analyse – Locals are Private

36

**Note: Simulating Java Classes including data/method encapsulation and data hiding**

For now remember using **this** makes a property (data or method) public (obvious from what we know about JavaScript object properties).

Using **var** makes a property (data or method) private (this has to do with a surprising but fundamental JavaScript feature called closure).

We will agonise over closure later in the unit

public  
public  
public  
private  
public

```
function Person(first,last,age) {  
  this.firstname = first;  
  this.lastname = last;  
  this.age = age;  
  var bankBalance = 7500;  
  this.setBankBalance = function(newBalance){  
    bankBalance = newBalance;  MUTATOR  
  };  
  this.getBankBalance = function(){  
    return bankBalance;  ACCESSOR  
  };  
}  
  
// create your Person  
var john = new Person("John", "Smith", 21);  
console.log(john.bankBalance); undefined  
console.log(john.getBankBalance()); 7500  
  
var john2 = new Person("John2", "Smith2", 22);  
john2.setBankBalance(-100);  
console.log(john2.bankBalance); undefined  
console.log(john2.getBankBalance()); -100
```

Codecademy

public  
public  
public  
private  
  
private

returning a  
function/  
method  
reference

11 23/30

```
function Person(first,last,age) {  
  this.firstname = first;  
  this.lastname = last;  
  this.age = age;  
  var bankBalance = 7500;  
  
  var returnBalance = function() {  
    return bankBalance;  
  };  PRIVATE ACCESSOR (helper)  
  
  // create the new function here  
  this.askTeller = function(){  
    return returnBalance;  
  }  PUBLIC ACCESSOR (calls helper)  
}  
  
var john = new Person('John','Smith',30);  
  
console.log(john.returnBalance); undefined  
  
var myBalanceMethod = john.askTeller;  
var myBalance = myBalanceMethod();  
console.log(myBalance); 7500
```

Codecademy

public

11 25/30

These last three lines are just a reminder that JavaScript functions are first class.  
We could have just coded: `console.log(john.askTeller());`