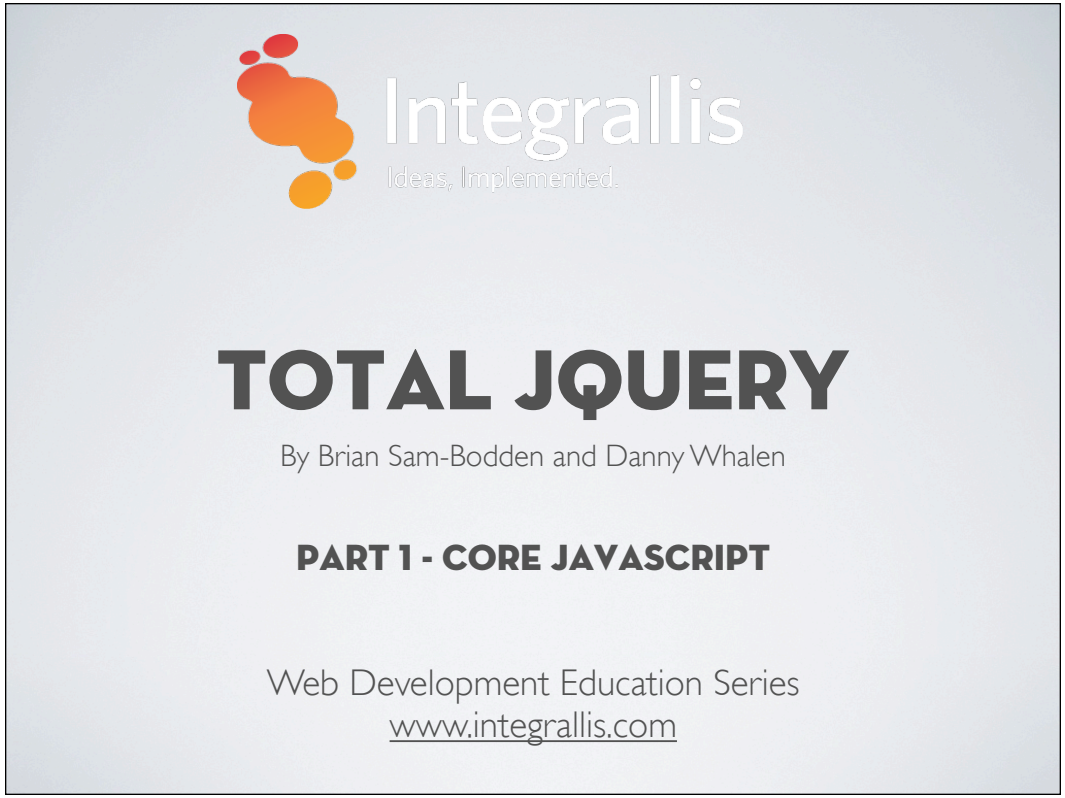


A publication of Integrallis Software, LLC.
www.integrallis.com/en/training
info@integrallis.com

OVERVIEW & OBJECTIVES

- Part `1` starts by re-introducing students to the core JavaScript language without muddying the waters with the browser DOM

- Part `1` covers:
  - Foundations
  - Functions
  - Objects
  - Methods
  - Prototypes
  - Scope
  - Closures
  - Currying
  - Code Organization

MATERIAL CONVENTIONS

- Each training day consists of a mix of lecture time, discussion time and Q & A, guided exercises and labs

- For the guided exercises you will see a green "follow along" sign on the slides

- For the labs you'll see an orange sign with the lab number

- For things that I consider good habits or good practices

PREPARATION & SETUP

CORE REQUIREMENTS

- For this course you'll need a system with:
  - Browser
    - Chrome - latest stable release (preferred)
    - Firefox or other
  - Text Editor
    - Redcar / Notepad++ / Coda / gedit

4

Students are welcome to use Internet Explorer, Google's Chrome or Apple's Safari to run the included examples

Older versions of Internet Explorer (< 8) are known for their inconsistencies and lack of adherence to the standards

JAVASCRIPT

THE LANGUAGE OF THE BROWSER

JAVASCRIPT

ONE OF THE WORLD'S MOST POPULAR PROGRAMMING LANGUAGES

- Has nothing to do with Java!

- It looks like C/Java but it is more like Lisp / Scheme
- It amazingly powerful but misused and misunderstood by most
- For a long time it became the main pain point in the "browser wars"*
- Created by **Brendan Eich** at Netscape as **Mocha** and then renamed to **LiveScript** when it shipped with the beta release of Netscape **Navigator 2.0** in **1995**

6

*Mostly due to the DOM implementation differences

JAVASCRIPT

ONE OF THE WORLD'S MOST POPULAR PROGRAMMING LANGUAGES

✓ Interpreted ✓ Imperative & Structured

✓ Constructor Functions ✓ Object-based, Class-free

✓ Prototypes Javascript ✓ Loosely Typed

✓ Closures ✓ Run-time evaluation

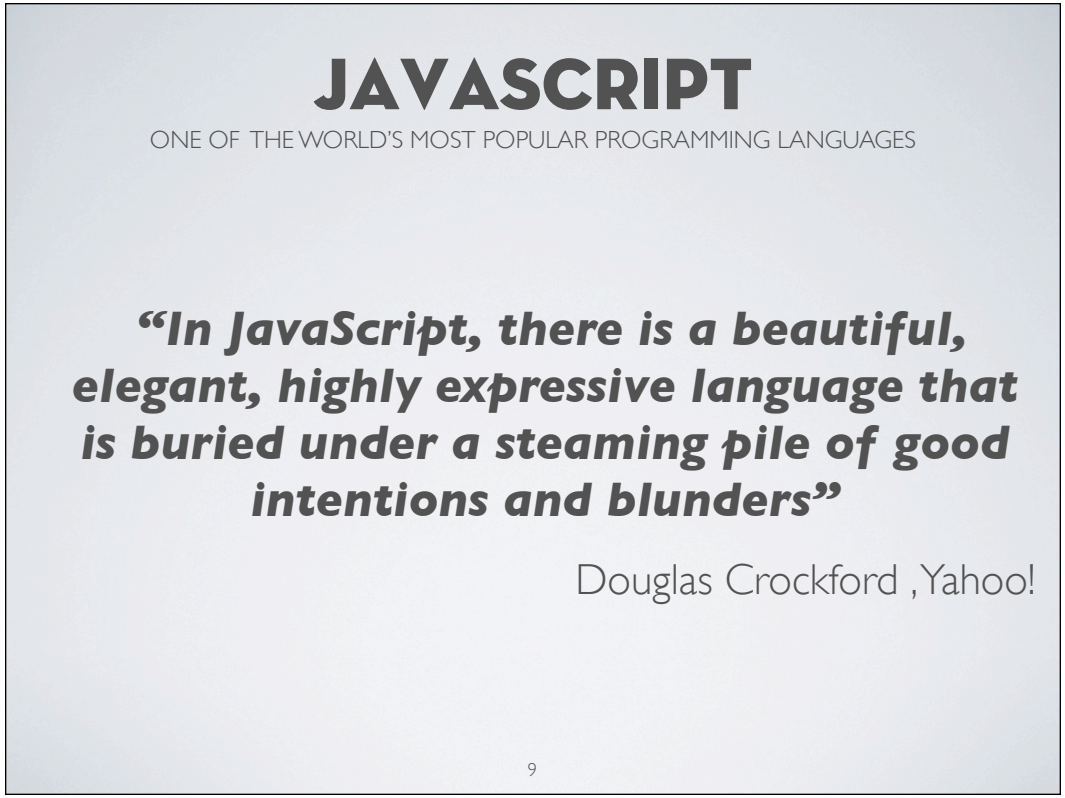
✓ Inner Functions ✓ First-class functions

JAVASCRIPT

ONE OF THE WORLD'S MOST POPULAR PROGRAMMING LANGUAGES

- JavaScript is interpreted and tagged with the label "scripting language" and therefore deemed by many as a "toy" language or a language not suited for "real" programming
- JavaScript went from inception to global adoption as fast as the browser gained popularity. It's popularity was further propelled by the fall of Java Applets and later by efforts to standardize asynchronous and dynamic behavior on the browser (AJAX)
- JavaScript is a full-featured language, a small language with many good parts and a few bad ones to be avoided

Interpreted languages are no longer considered toys, neither do dynamically typed languages



JavaScript had no time to properly mature, even its submission to the Ecma International body was rushed and seem mostly as a political move for control between Netscape and Microsoft

CORE JAVASCRIPT

GRAMMAR, SYNTAX AND BUILDING BLOCKS

JAVASCRIPT

GRAMMAR AND SYNTAX

- JavaScript has a **C-style** syntax (curly braces and semicolons)
  - **Case Sensitive** (`for != For != fOr`)*
  - Not white space sensitive
  - Semicolons are optional (but just ignore this for your own good)
  - Identifiers must start with a **letter**, **underscore** or **dollar** sign
  - Programs are written in the **Unicode** Character Set

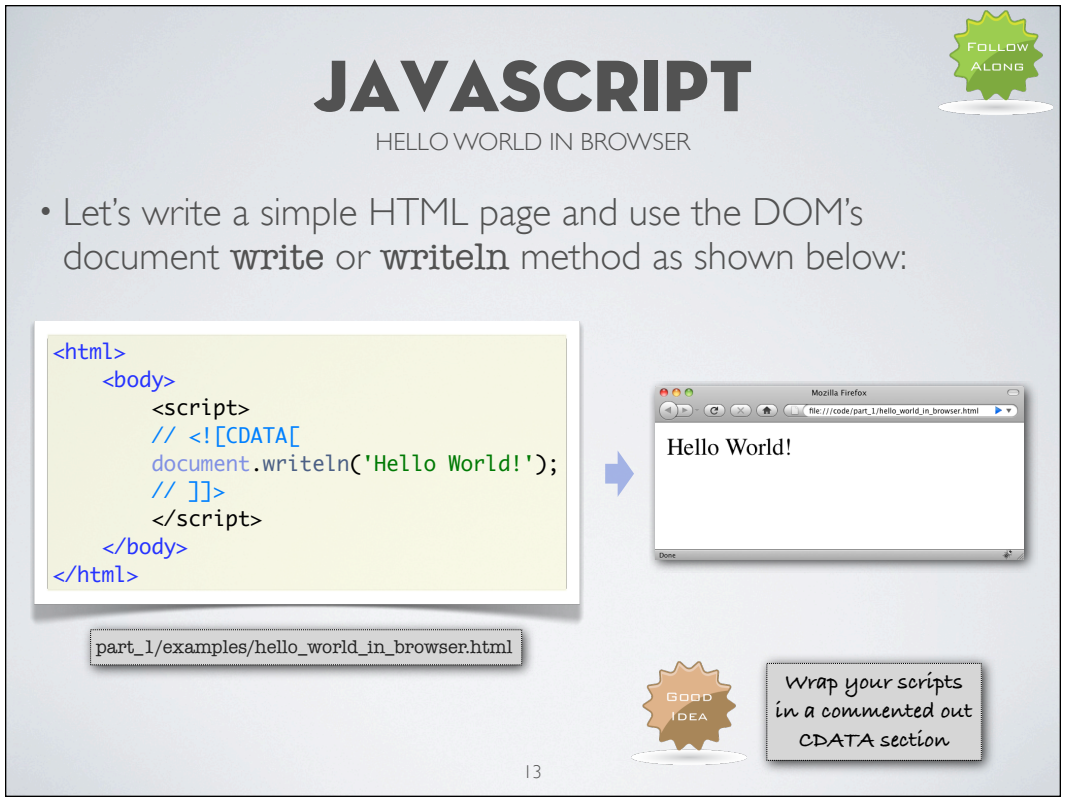
* This is specially important when dealing with HTML attributes in JavaScript

JAVASCRIPT

COMMENTS

- JavaScript Comments follow the C/C++/Java style
- Use // for single-line comments
- Use /* ... */ for multi-line comments

```
// this is a comment
/* this is also a comment */
```



Is it necessary to surround the script body in a CDATA section?

Only if you use XHTML which by default would consider the script body to be PCDATA (Parsed Character Data) and when processed by the validator will fail to validate the page

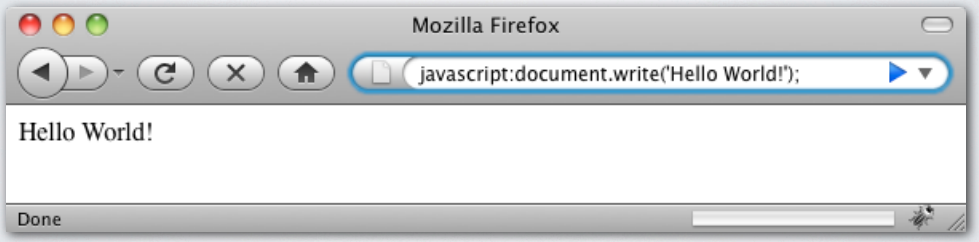
If the script is externalize (as it should) there is no need for the CDATA wrapper

In older browsers the CDATA tag might not be understood correctly so you might need to wrap it with Javascript comments as shown above

JAVASCRIPT

HELLO WORLD VIA JAVASCRIPT PSEUDO PROTOCOL

- Certain browsers (Firefox and Opera for example) support the `javascript:` URL pseudo-protocol* which allows for the inline evaluation of javascript using the browser address bar (location bar or URL bar)



The URL pseudo-protocol is not widely supported, it typically breaks in older browsers. Avoid its usage for referencing JavaScript from your markup

It is also fairly impractical to do anything with more that one line of JavaScript code this way!

JAVASCRIPT

HELLO WORLD IN FIREBUG

- We could also use `document.write` inside of the FireBug JavaScript console provided by the popular Firefox Add-on:

JAVASCRIPT

FOCUS ON THE LANGUAGE

- Since we want to focus on the core JavaScript language and not on how to work with the browser's DOM (not yet at least) we can create a simple print function to hide the details of writing to the DOM:

```
function print(text) {
     $(document).ready(function(){
          $('body').append('<p>'+text+'</p>').append('<br/>');
     });
}
```

part_1/examples/utilities/print_function.js

JAVASCRIPT

FOCUS ON THE LANGUAGE

- We can now use our custom print function by including the script in an HTML page:

```
<html>
    <head>
        <script src="http://ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.js"></script>
        <script src="utilities/print_function.js"></script>
        <script>
        // <![CDATA[
        // semicolons are 'sometimes' optional
        print("There is no semicolon")
         // ]]>
        </script>
    </head>
    <body>
    </body>
</html>
```

part_1/examples/semicolons_embedded.html

All of the included **.js** example files have a corresponding **.html** file

It might be a good idea to wrap the call to print in an event handler for the window.onload event

JAVASCRIPT

SEMI-COLONS

- Semi-colons (;) are the statement terminator/separator
- JavaScript inserts an implicit semicolon for statements separated by an end of line character

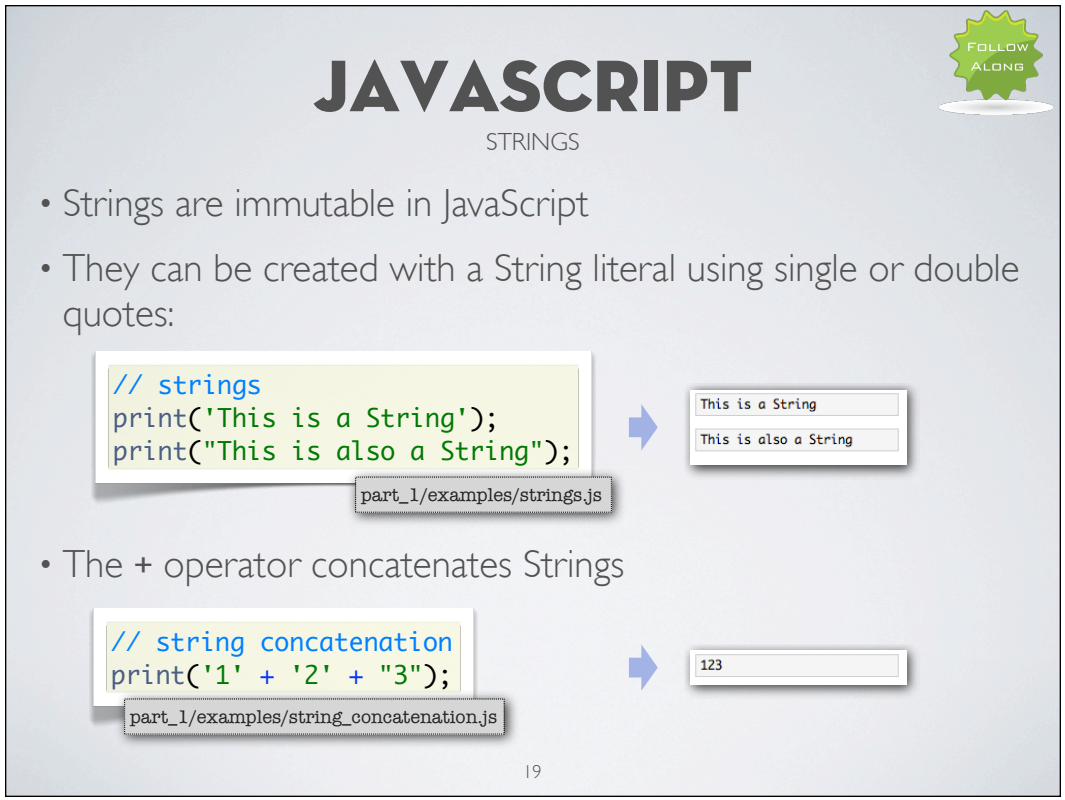
- If certain scenarios omitting a semicolon can have unintended consequences, therefore is consider good form to always use them

```
// semicolons are 'sometimes' optional
print("There is no semicolon")
```

part_1/examples/semicolons.js

```
// multiple statements per line
print("Packed in a single line"); print("Use only for readability");
```

part_1/examples/multiple_statements.js



For some welcomed additions to JavaScript string functionality and other nice thoughts see Douglas Crockford's "Remedial JavaScript" at http://javascript.crockford.com/remedial.html

JAVASCRIPT

NUMBERS

- Numbers in JavaScript are floating-point values
- All numbers, including integers* are backed by the same type (64-bit floating-point format)
- Below are some simple examples or numeric literals being manipulated:

```
// numbers
print(2 + 3 * 3);
print(1 / 0);
print(Math.sqrt(-1));
```

part_1/examples/numbers.js

```
11
Infinity
NaN
```


* IEEE 754 standard; the same format as Java's double type and most modern implementations of C and C++

JAVASCRIPT

EQUALITY

- The regular equality operator (==) is used to check that two values are equivalent
- The strict equality operator (===) is used to check that the two operands have the same type and value

```
// equality
print(2 + 3 * 3 == '11');  // regular equality
print(2 + 3 * 3 === '11'); // strict equality
```

part_1/examples/equality.js

```
true
false
```

JAVASCRIPT

VARIABLES

- Variables are declared using the `var` keyword and are untyped until assignment time

```
// declaration
var book_title;
print(book_title);

// assignment
book_title = "JavaScript: The Good Parts";
print(book_title);

// explicit declaration & assignment
var one_more_book_title = "JavaScript: The Definitive Guide";

// implicit declaration & assignment
yet_another_book_title = "Secrets of the JavaScript Ninja";
print(one_more_book_title + ' and ' + yet_another_book_title);
```

```
undefined
JavaScript: The Good Parts
JavaScript: The Definitive Guide and Secrets of the JavaScript Ninja
```

part_1/examples/variables.js

JAVASCRIPT

VARIABLES

- Variables declared with the `var` keyword are local to their enclosing scope*
- Variables declared without the `var` keyword become global variables
- To declare multiple variables in one line you can pass a comma-separated list of identifiers
- A declared variable that has not been assigned a value has the literal value `undefined`

* we will learn more about scopes later in this section

Variables created without the var keyword belong to the global scope and are not garbage collected when the function returns creating the potential for a memory leak

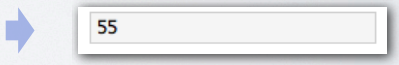
JAVASCRIPT

WHILE LOOP

- JavaScript's while loop should look familiar to C, C++ and Java developers
- It follows the syntax **while (expression) statement**
- While the **expression** evaluates to **true** the **statement** will continue to be evaluated

```
// while loop
var total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
print(total);
```

part_1/examples/while_loop.js



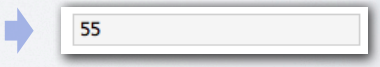
JAVASCRIPT

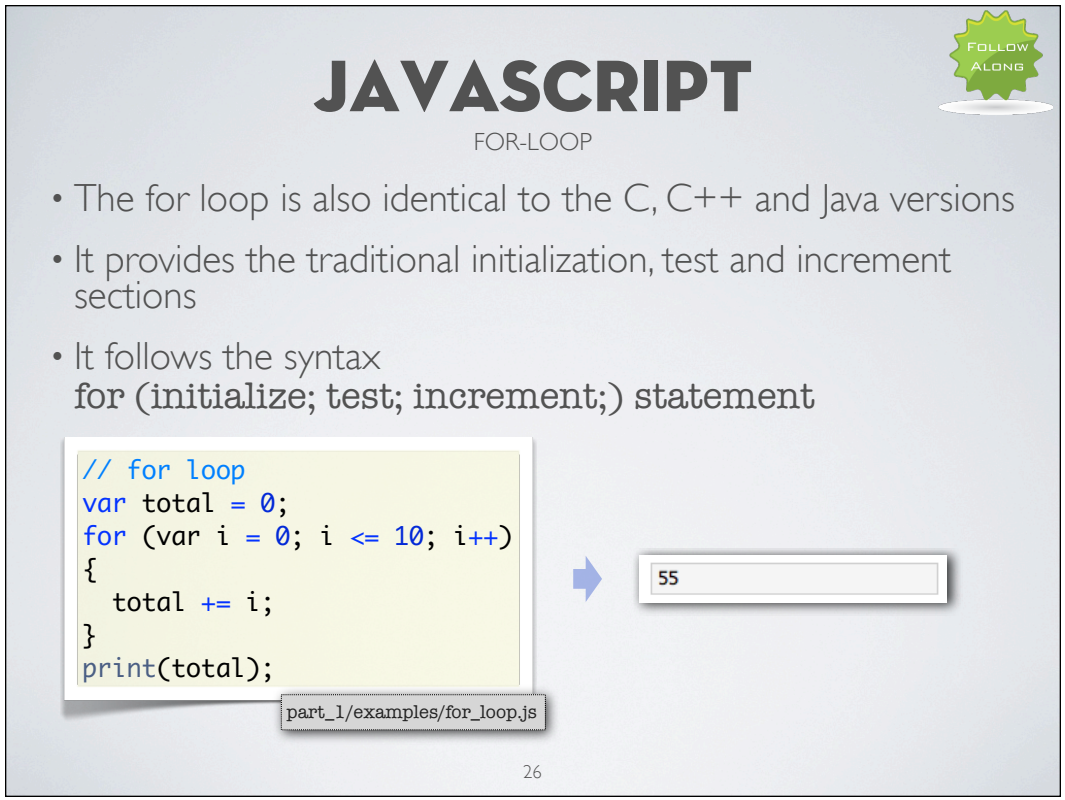
DO WHILE LOOP

- JavaScript's do-while loop evaluates the conditional expression at the bottom of the loop therefore it evaluates the statement at least once
- It follows the syntax **do statement while (expression)**

```
// do while loop
var total = 0, count = 1;
do {
  total += count;
  count += 1;
} while (count <= 10)
print(total);
```

part_1/examples/do_while_loop.js





Be sure to declare “i” with var to ensure if not defined globally.

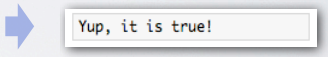
JAVASCRIPT

IF-ELSE

- The familiar if and if-else conditional statements are available in JavaScript:

```
// if-else
if (2 + 3 * 5 === 17) {
    print("Yup, it is true!");
}
else {
    print("Nope, that was a lie!");
}
```

part_1/examples/if_else.js



JAVASCRIPT

NESTED IF-ELSE

- A nested if-else-if-else statement:

```
// if-else-if-else
var today = new Date();
var hour = today.getHours();
if (hour < 12) {
    print("Good Morning");
} else if ((hour >= 12) && (hour <= 17)) {
    print("Good Afternoon");
} else if ((hour >= 18) && (hour <= 22)) {
    print("Good Evening");
} else {
    print("Good Night");
}
```

```
Good Afternoon
```

part_1/examples/if_else_if_else.js

JAVASCRIPT

SWITCH OPERATOR

- The switch statement provides a cleaner way to do multiple conditions, it follows the syntax:
  `switch(expression) { statements }`

```
The flipping weekend is here!
```

```
// switch
var today = new Date();
var day = today.getDay()
switch(day) {
    case 1:
        print("Looks like a case of the Mondays!");
        break;
    case 2: case 3: case 4:
        print("The middle of the week is for the birds!");
        break;
    case 5:
        print("Just pretend you're working until 4pm ;-)");
        break;
    default:
        print("The flipping weekend is here!");
}
```

part_1/examples/switch.js

JAVASCRIPT

SWITCH FOR RANGE CONDITIONS

- An alternate approach to deal with complex (range) conditions:

```
// alternate switch with ranging
var today = new Date();
var hour = today.getHours();
switch(true) {
    case (hour < 12):
        print("Good Morning");
        break;
    case ((hour >= 12) && (hour <= 17)):
        print("Good Afternoon")
        break;
    case ((hour >= 18) && (hour <= 22)):
        print("Good Evening");
        break;
    default:
        print("Good Night");
}
```

```
Good Afternoon
```

part_1/examples/switch_with_ranging.js

JAVASCRIPT

TRY CATCH FINALLY

- JavaScript provides a familiar try-catch-finally statement:

```
function try_it() {
    try {
        iDontExist();
    } catch (error) {
        print("stop calling non-existent functions ==> " + error.message);
    } finally {
        print("... and we're done");
    }
    return "really we are";
}

print(try_it());
```

```
stop calling non-existent functions ==> "iDontExist" is not defined.
... and we're done
really we are
```

Avoid Try-Catch block in performance critical sections of the application

part_1/examples/try_catch_finally.js

FUNCTIONS

FUNCTIONS: THE HEART OF JAVASCRIPT

FUNCTIONS

THE HEART OF JAVASCRIPT

- JavaScript is a functional language: embrace it and you'll benefit!
- JavaScript can be seen as the first functional* language to go mainstream
- A function is a reusable block of code that can take zero or more arguments and optionally return a value
- In JavaScript functions are first-class citizens; they are objects that can be assigned and passed around
- Functions are the building blocks of JavaScript and the main element used in creating reusable code libraries

* Let's call it pseudo-functional instead!

FUNCTIONS

DECLARATION

- Functions are declared with the keyword **function** and can have a name, parameters/arguments and an optional return value
- A function is evaluated/executed using the () operator

```
function foo() {
    return "bar";
}

print(foo());
```

part_1/examples/function_with_return.js

```
bar
```

FUNCTIONS

FUNCTIONS WITHOUT A RETURN VALUE

- A function without a return value returns the value `undefined`
- A function name must be a valid JavaScript identifier

```
function foo() {
    1 + 2;
}

print(foo());
```

part_1/examples/function_with_no_return.js

```
undefined
```

FUNCTIONS

ANONYMOUS FUNCTIONS

- A function without a name is called an anonymous function
- The result of creating an anonymous function with the function operator is a `Function` object

```
(function() {
    print("I've been to the desert on horse with no name");
})();
```

part_1/examples/anonymous_functions.js

```
I've been to the desert on horse with no name
```

FUNCTIONS

RECURSIVE FUNCTIONS

- As first class citizens functions are objects that can be assigned to a variable and later executed with the () operator
- Functions are accessible from within themselves enabling recursive functions
- Anonymous functions can be named but those names are only visible within the functions themselves.

```
var f = function j(x) {
    return (x < 1) ? 0 : x + j(x - 1);
};
print(f(10));
```

```
55
```

part_1/examples/named_recursion.js

FUNCTIONS

RECURSIVE FUNCTIONS

- Below is a recursive definition of the factorial function:

```
function factorial(n) {
    return (n <= 1) ? 1 : (n * factorial(n-1));
}

print(factorial(10));
```

part_1/examples/factorial.js

```
3628800
```

FUNCTIONS

ANONYMOUS RECURSIVE FUNCTIONS

- Functions have access to a local variable called **arguments** which is an array-like object corresponding to the arguments of the current invocation of the function
- The arguments object provides a **callee** property that refers to the currently executing function
- The **arguments.callee** property is the only way to create a recursive anonymous function

```
55
```

```
var f = function(x) {
    return (x < 1) ? 0 : x + arguments.callee(x - 1);
};
print(f(10));
```

part_1/examples/anonymous_recursion.js

FUNCTIONS

GLOBAL FUNCTIONS

- Functions declared in the global context do not follow the top-down declaration order; they are accessible from anywhere!

```
print("Calling a function before it is 'defined' ==> "
      + sayHello("My Little Friend"));
function sayHello(to) {
    return "Say Hello to " + to;
}
```

part_1/examples/global_functions.js

```
Calling a function before it is 'defined' ==> Say Hello to My Little Friend
```

FUNCTIONS

HIDING A FUNCTION

- Hiding a function below a function `return` does not hide it from the enclosing function but it does hide it from the global context

```
function a() {
  var result = "Neo says: " + b();
  return result;
  function b() { return "Whoa!"; }
}
print(a());
print(b());
```

part_1/examples/hiding_a_function.js

```
Neo says: Whoa!

Rhino says: uncaught JavaScript runtime exception:
ReferenceError: "b" is not defined.
```

FUNCTIONS

ORDER OF DECLARATION

- The previous example does not work with an anonymous function assigned to a variable

```
function a() {
  var result = "Neo says: " + b();
  return result;
  var b = function() { return "Whoa!"; }
}
print(a());
```

part_1/examples/cant_hide.js

Rhino says: uncaught JavaScript runtime exception:
TypeError: b is not a function, it is undefined.

FUNCTIONS

PARAMETERS

- Functions can take a named list of zero or more arguments
- A function can take any number of arguments, defined or not

```
function add_two_things(thing_one, thing_two) {
    return thing_one + thing_two;
}

print(add_two_things(1, 3));
print(add_two_things("Double ", "Rainbow"));
print(add_two_things(8, 6, 7, 5, 3, 0, 9));
```

part_1/examples/functions_fix_args.js

```
4
Double Rainbow
14
```

FUNCTIONS

PARAMETERS

- JavaScript supports variable arity functions known as variadic or var-args functions
- Using the arguments object we can access the actual arguments passed to the function

```
function many_args() {
    for (var i=0; i < arguments.length; i++) {
        print(arguments[i]);
    };

    var real_array = Array.prototype.slice.call(arguments);
    print(real_array.join(' '));
}

many_args('This', 'is', 'a', 'varargs', 'function');
```

part_1/examples/var_args.js

```
This is a varargs function
```

44

The arguments object is not a real array but an array-like object that doesn't support the join method

FUNCTIONS

PARAMETERS

- There are many different ways to accomplish something in JavaScript:

```
function many_args() {
    print(Array.prototype.slice.call(arguments).join(' '));
}

many_args('This', 'is', 'a', 'varargs', 'function');
```

part_1/examples/var_args_using_slice.js

```
This is a varargs function
```

LAB 1.0

JAVASCRIPT FUNCTIONS

- In Lab **1.0** you are asked to create a JavaScript program with one function **add_them** that will:
  - Take a variable number of numerical arguments and add them together
  - Use recursion or traditional looping techniques

```
function add_them() {
    ...
}

print(add_them(1,2,3,4,5,6,7,8,9,10));
```

should return

55

LAB 1.1

RECURSION

- In Lab `1.1` you are asked to create a JavaScript program with one recursive function `fib` that will:
  - Calculate the n^{th} Fibonacci Number
  - See http://en.wikipedia.org/wiki/Fibonacci_number

```
function fib(n) {
    ...
}

print(fib(20));
```

should return

```
6765
```

OBJECTS

OBJECTS WITHOUT CLASSES

OBJECTS

CLASS-LESS OBJECTS

- In JavaScript, anything that is not a simple type (number, string, booleans, null and undefined) is an object
- JavaScript objects are **mutable**, unordered collections of named values similar to a hash/map structure
- These values are referred to as the properties of the object and are themselves a simple type, an object or a function (method)
- An **object literal** provides a convenient way to create a new object; it consists of zero or more **name/value pairs** surrounded by **curly braces**:

```
var empty_object = {};
```

part_1/examples/empty_object.js

OBJECTS

CREATING OBJECTS WITH AN OBJECT LITERAL

- Using an object literal we can create an object with a combination of properties (string and numbers) and access those properties using dot notation:

```
10544 East Meadowhill Dr., Scottsdale AZ 85255
```

```
var address = {
    street : "10544 East Meadowhill Dr.",
    city : "Scottsdale",
    state : "AZ",
    zip : 85255
};

print(address.street + ", " +
      address.city + " " +
      address.state + " " +
      address.zip);
```

part_1/examples/object_literal.js

OBJECTS

NESTED OBJECTS

- Object properties can be objects themselves. Dot notation is used to access properties as well as subscript notation (array access notation)
- Dot notation is preferred as it is more legible*

```
var delivery = {
    to : "Anne Sam-Bodden",
    address : {
        street : "10544 East Meadowhill Dr.",
        city : "Scottsdale",
        state : "AZ",
        zip : 85255
    },
    sku : "8675309",
    quantity : 2
};

print(delivery.to);
print(delivery["to"]);
print(delivery.address.street);
print(delivery["address"]["street"]);
```

part_1/examples/nested_objects.js

```
Anne Sam-Bodden
Anne Sam-Bodden
10544 East Meadowhill Dr.
10544 East Meadowhill Dr.
```


* unless you are dealing with a multi-dimensional-array-like structure

OBJECTS

OBJECT PROPERTIES

- To use dot notation object properties must follow JavaScript identifier naming conventions
- An improperly named property causes a syntax error:

part_1/examples/bad_properties.js

```
var address = {
    street-one : "10544 East Meadowhill Dr.",
    city : "Scottsdale",
    state : "AZ",
    zip : 85255
};
```

Rhino says: on line 2 at position 8: missing : after property id
`street-one : "10544 East Meadowhill Dr.",`

Rhino says: on line 3 at position 7: syntax error
`city : "Scottsdale",`

Rhino says: on line 4 at position 8: syntax error
`state : "AZ",`

Rhino says: on line 5 at position 6: syntax error
`zip : 85255`

OBJECTS

NON-IDENTIFIER PROPERTIES

- To use a property that is not a valid JavaScript identifier we must specify the name as a String
- To access the property we can only use subscript notation

part_1/examples/string_properties.js

```
var address = {
    "street-one" : "10544 East Meadowhill Dr.",
    city : "Scottsdale",
    state : "AZ",
    zip : 85255
};

print(address["street-one"] + ", " +
      address.city + " " +
      address.state + " " +
      address.zip);
```

```
10544 East Meadowhill Dr., Scottsdale AZ 85255
```

OBJECTS

MANIPULATING PROPERTIES

- Properties can be deleted using the **delete** keyword
- Reading an undefined property returns the value **undefined**
- The **in** operator returns **true** if a property exists, false **otherwise**

part_1/examples/manipulating_properties.js

```
var person = { first : "Brian", middle : "Michael", last : "Sam-Bodden" };
print("middle" in person);
print(person.first + " " + person.middle + " " + person.last);
delete person.middle;
print("middle" in person);
print(person.first + " " + person.middle + " " + person.last);
```

```
true
Brian Michael Sam-Bodden
false
Brian undefined Sam-Bodden
```

OBJECTS

CREATING OBJECTS WITH AN OBJECT LITERAL

- Objects can also be created using special constructor functions
- The `new` operator creates an instance of a user-defined object type or of one of the built-in object types (that has a constructor function) such as `Object`
- Properties can be added (and removed) at any time!

part_1/examples/constructor_functions.js

```
var loc = new Object();
loc.latitude = 37.0625;
loc.longitude = -96.677068;

print("Latitude = " + loc.latitude + ", Longitude = " + loc.longitude);
```

```
Latitude = 37.0625, Longitude = -96.677068
```

OBJECTS

GLOBAL OBJECTS

- There are several "global objects" or objects in the global scope that posses constructor functions:

Standard Global Object	Summary	Syntax
Object	Generic Object Wrapper	`new Object(value)`
Number	Object wrapper for numerical values	`new Number(value)`
String	Global object to construct String instances	`String(literal)` or `new String(literal)`
Array	An ordered set of values associated with a single variable name	`[e1, e2, e3]` or `new Array(e1, e2, e3)`
Boolean	Object wrapper for boolean values	`new Boolean(value)`
Date	Object to work with dates and times	`new Date()`, `new Date(millis)`, `new Date(string)`, `new Date(year, month, date [, hour, minute, second, millisencond)`
RegExp	Regular Expression Matcher Object	`RegExp(pattern [, flags])` or `/pattern/flags`
Function	All Functions are Function Objects	`new Function(argumentList, functionBody)`

https://developer.mozilla.org/en/JavaScript/Reference

OBJECTS

FALSY VALUES

- The values 0, NaN, null, "" and undefined are all converted to false:

```
var truthy_primitive_boolean = true;
var falsy_primitive_boolean = false;

var truthy_object_boolean = new Boolean(true);
var falsy_object_boolean = new Boolean(false);

print(truthy_object_boolean.toString());
print(truthy_object_boolean.valueOf());
print(falsy_object_boolean.toString());
print(falsy_object_boolean.valueOf());

if (falsy_object_boolean) {
    print("Oops! I should be false!");
}

if (!falsy_object_boolean.valueOf()) {
    print("Ok, that's more like it!");
}

// primitive to object conversion
var new_boolean = Boolean(truthy_primitive_boolean);
print("The new value is " + new_boolean.valueOf());
```

```
true
true
false
false
Oops! I should be false!
Ok, that's more like it!
The new value is true
```

part_1/examples/booleans.js

Most object wrappers provide a valueOf() method to return their primitive equivalent

OBJECTS

NUMBER GLOBAL OBJECT

- JavaScript doesn't have specific integer or floating point types; all JavaScript numbers are 64bit (8 bytes) floating point numbers
- Integers are reliable up to 15 digits. Care must be taken when working with floating point numbers, specially if dealing with currencies

```
print(0.09 + 0.01);
```

```
0.09999999999999999
```

part_1/examples/precision.js

OBJECTS

NUMBER GLOBAL OBJECT

- JavaScript numbers are bound by the properties of the `Number` class `Number.POSITIVE_INFINITY` and `Number.NEGATIVE_INFINITY`

```
var value = 2;
for (var i = 1; value < Infinity; i++) {
  value = Math.pow(value, 2);
  print("i ==> " + value);
}
```

part_1/examples/infinity.js

```
i ==> 4
i ==> 16
i ==> 256
i ==> 65536
i ==> 4294967296
i ==> 18446744073709552000
i ==> 3.402823669209385e+38
i ==> 1.157920892373162e+77
i ==> 1.3407807929942597e+154
i ==> Infinity
```

OBJECTS

NUMBER GLOBAL OBJECT

- Working outside of the range of **Number** will produce results like **Infinity**, **-Infinity** or **NaN** (Not a Number)
- Notice that JavaScript does not throw an exception for division by zero like many other languages do

```
print(1/0);
print(-1/0);
print(Infinity/Infinity);
print(Infinity - Infinity);
print(Infinity / 0);
print(-Infinity / 0);
```

part_1/examples/ranges.js

```
Infinity
-Infinity
NaN
NaN
Infinity
-Infinity
```

OBJECTS

NUMBER GLOBAL OBJECT

- The **Number** class allows you to work with Octal, Hexadecimal and Binary numbers:

```
var octal = 0377;
var octal_string = "377";
var octal_as_int = parseInt(octal_string, 8);
var an_int = 255;
var int_as_octal = an_int.toString(8);

print(octal);
print(octal_as_int);
print(int_as_octal);

if (octal == octal_as_int) {
    print("octal and octal_as_int are equivalent!");
}
if (octal === octal_as_int) {
    print("but they are strictly the same");
}
```

part_1/examples/number_bases.js

```
255
255
377
octal and octal_as_int are equivalent!
but they are strickly the same
```

61

"parseInt" is a global function and not part of any core JavaScript object)it's actually part of the global object or root object)

OBJECTS

NUMBER GLOBAL OBJECT

- The `Number` class allows you to work with numbers is any base number system
- JavaScript stores all numbers as base-10. The `toString(base)` method is used to convert from base-10 to any other base

```
var decimal = 1972;
var to_hex = decimal.toString(16);
var to_binary = decimal.toString(2);
var to_septenary = decimal.toString(7);
var to_quaternary = decimal.toString(4);

print(decimal);
print(to_hex);
print(to_binary);
print(to_septenary);
print(to_quaternary);
```

part_1/examples/numbers_to_string.js

```
1972
7b4
11110110100
5515
132310
```

OBJECTS

OPERATORS

- There are several "global objects" or objects in the global scope that can handle operators:

Operator	Description
+	Addition for Numbers / Concatenation for Strings
-	Subtraction
*	Multiplication
/	Division
%	Reminder / Modulus
++	Pre/Post Increment
–	Pre/Post Decrement
+=, -=, *=, /=, %=	value = value (operator) argument

OBJECTS

ARRAYS

- JavaScript arrays are ordered collections of data values and indexed by a non-negative integer
- JavaScript arrays are zero-based
- Remember: JavaScript Objects are associative arrays indexed by names (Strings / Identifiers) while regular arrays are indexed with integers
- JavaScript does not support multidimensional arrays but you can create an array of arrays
- Arrays, like everything else in JavaScript are untyped: An array element can be a collection of different types

OBJECTS

WAYS TO CREATE AN ARRAY

- Arrays can be created using the Array class with or without the new operator or simply by using an array literal

```
var array_one = [];
var array_two = new Array(7);
var array_three = [7];
var array_four = Array(7);

print(array_one);
print(array_two);
print(array_three);
print(array_four);

array_one[4] = 'than'
array_one[1] = 'done'
array_one[6] = 'said'
array_one[3] = 'better'
array_one[0] = 'well'
array_one[2] = 'is'
array_one[5] = array_one[0]

array_two[11] = 8675309;

print(array_one);
print(array_two);
```

part_1/examples/array_creation.js

```
,,,,,,
7
,,,,,,
well,done,is,better,than,well,said
,,,,,,,,,,,8675309
```

An Array Literal is the preferred, cleaner way to create Arrays

OBJECTS

STORING THINGS IN AN ARRAY

- Arrays can hold any kind of value; primitives, objects, functions and other arrays

```
var mixed_array = [
    42,
    "Zaphod Beeblebrox",
    { title : "Hitchhiker's Guide to the Galaxy", publisher : "Megadodo" },
    function () { return this[0]; },
    ["a", "e", "i", "o", "u"]
];

for (var i=0; i < mixed_array.length; i++) {
    if ( typeof mixed_array[i] === "function") {
        print(mixed_array[i]());
    } else {
        print(mixed_array[i]);
    }
}
```

```
42
Zaphod Beeblebrox
[object Object]
42
a,e,i,o,u
```

part_1/examples/array_storage.js

OBJECTS

LOOPING OVER AN ARRAY OF OBJECTS

- A special form of the for loop; **for in** can be used for iterating through a regular array or an associative array (loop over an object properties)

```
var dogs = [{ name : "Bullseye", breed : "Miniature Bull Terrier" },
            { name : "Lassie", breed : "Collie" },
            { name : "Rin Tin Tin", breed : "German Shepherd" },
            { name : "Benji", breed : "Mutt" },
            { name : "Cujo", breed : "Saint Bernard" }];

for (var index in dogs) {
    var dog = dogs[index];
    print(dog.name + " is a " + dog.breed);
}
```

```
Bullseye is a Miniature Bull Terrier
Lassie is a Collie
Rin Tin Tin is a German Shepperd
Benji is a Mutt
Cujo is a Saint Bernard
```

part_1/examples/array_looping.js

OBJECTS

ARRAY OF ARRAYS

- Multidimensional arrays can be faked using an array of arrays

```
var oranges = ['Valencia', 'Tangerines', 'Mandarines']
var apples = ['Macintosh', 'Granny Smith', 'Gravestein

var oranges_and_apples = []

for (var i=0; i < 3; i++) {
    oranges_and_apples[i] = [oranges[i], apples[i]];
}

for (var index in oranges_and_apples) {
    print("You say " + oranges_and_apples[index].join(" and I say "));
}
```

```
You say Valencia and I say Macintosh
You say Tangerines and I say Granny Smith
You say Mandarines and I say Gravestein
```

part_1/examples/array_of_array.js

OBJECTS

ARRAY OPERATIONS


```
var an_array = [1, "Two", 3, "Cuatro", 5, "Six", 7];
var another_array = [8, 9, 10];
var elementIsANumber = function(n) { return (typeof n == "number"); }
var print_squared = function(n) { print(n + " squared is " + (n * n)); }
var cubed = function(n) { return Math.pow(n, 3); }

print(an_array);
print("Length ==> " + an_array.length);
print("Concatenating ==> " + an_array.concat(another_array));
print("All Numbers? ==> " + an_array.every(elementIsANumber));
print("All Numbers? ==> " + another_array.every(elementIsANumber));
print("Some Numbers? ==> " + an_array.some(elementIsANumber));
print("Just the numbers ==> " + an_array.filter(elementIsANumber));
another_array.forEach(print_squared);
print(an_array.join(" and "));
print("The index of 'Two' is ==> " + an_array.indexOf('Two'));
print("The index of 'Three' is ==> " + an_array.indexOf('Three'));
print("All cubed ==> " + another_array.map(cubed));
print("Popped ==> " + another_array.pop() + ", array ==> " + another_array);
another_array.push(11, 12);
print("Pushed 11, 12 ==>" + another_array);
print("Reversed ==> " + an_array.reverse());
print("Shifted ==> " + another_array.shift() + ", array ==> " + another_array);
another_array.unshift(44);
print("UnShifted 44 ==> " + another_array);
print("Sorting ==> " + an_array.sort());
```

```
1,Two,3,Cuatro,5,Six,7
Length ==> 7
Concatenating ==> 1,Two,3,Cuatro,5,Six,7,8,9,10
All Numbers? ==> false
All Numbers? ==> true
Some Numbers? ==> true
Just the numbers ==> 1,3,5,7
8 squared is 64
9 squared is 81
10 squared is 100
1 and Two and 3 and Cuatro and 5 and Six and 7
The index of 'Two' is ==> 1
The index of 'Three' is ==> -1
All cubed ==> 512,729,1000
Poppped ==> 10, array ==> 8,9
Pushed 11, 12 ==>8,9,11,12
Reversed ==> 7,Six,5,Cuatro,3,Two,1
Shifted ==> 8, array ==> 9,11,12
UnShifted 44 ==> 44,9,11,12
Sorting ==> 1,3,5,7,Cuatro,Six,Two
```

part_1/examples/array_operations.js

METHODS

ATTACHING FUNCTIONS TO AN OBJECT

METHODS

ATTACHING FUNCTIONS TO AN OBJECT

- Previously we learned that functions are first-class citizens that can be assigned to variables and passed around
- When a function is assigned to a property of an object we call it a method
- The object that owns the method becomes the context for the underlying function
- The owner object can be access from within the body of the method using the `this` keyword

METHODS

ATTACHING FUNCTIONS TO AN OBJECT

- We can create an object literal in which one of the properties is an anonymous function; we call this a *method*
- The method invocation is performed using (.) dot notation and the () operator
- The method invocation can also be accomplished using subscript notation [] and the () operator

```
var address = {
    street : "10544 East Meadowhill Dr.",
    city : "Scottsdale",
    state : "AZ",
    zip : 85255,
    print_it : function () {
        print(this.street + ", " + this.city + " " + this.state + " " + this.zip);
    }
};

address.print_it();
address["print_it"]();
```

```
10544 East Meadowhill Dr., Scottsdale AZ 85255
10544 East Meadowhill Dr., Scottsdale AZ 85255
```

part_1/examples/functions_as_properties.js

LAB 1.2

OBJECTS AND ARRAYS

- In Lab `1.2` you are asked to create a JavaScript program that:
  - Will create an object called `text_info` containing a String
  - The object should have a method called `count_chars` that will populate a data structure containing a character count for each character in the source String
  - Add a function `count_for(character)` to retrieve an individual character's count

LAB 1.2

OBJECTS AND ARRAYS

- Below is the expected result for Lab 1.2:

```
var text_info = {
    ...
};

print(text_info.content);
text_info.count_characters();
for (c in text_info.characters) {
    print('('+ c +')'+' appears '+
text_info.characters[c] + ' times.');
}
```

a man, a plan, a canal panama

(a) appears 10 times.

( ) appears 6 times.

(m) appears 2 times.

(n) appears 4 times.

(,) appears 2 times.

(p) appears 2 times.

(l) appears 2 times.

(c) appears 1 times.

PROTOTYPES

OBJECTS WITHOUT CLASSES

PROTOTYPES

CLASS-LESS OBJECTS

- JavaScript is an object-based language without classes
- JavaScript is an example of a prototype-based language
- Behavior reuse (which inheritance is a form of) is accomplished by cloning existing objects that serve as prototypes
- In a prototype-based language objects have a prototype chain of objects that contribute their properties and behaviors via delegation
- The prototype property of all JS objects is an object whose properties and methods are made available to objects that are part of its prototype chain

PROTOTYPES

UNDERSTANDING PROTOTYPES

- Let's revisit the creation of an object using the **new** operator against the **Object** built-in data type:

```
var loc = new Object();
loc.latitude = 37.0625;
loc.longitude = -96.677068;
```

part_1/examples/prototype_object.js

- The location object can be depicted as the table below. The additional property **constructor** is an internal JavaScript function that was "inherited" from its prototype (the **Object** prototype)

location	
latitude	37.0625
longitude	96.677068
Object.prototype	
constructor	Object

```
print(loc.constructor);
```

```
function Object() { [native code for Object.Object, arity=1] }
```

PROTOTYPES

UNDERSTANDING PROTOTYPES

- Every object has a prototype*, objects created from an object literal or with `new Object()` have `Object.prototype` as their prototype
- When we use `new` with a function an empty object is created, the constructor function is invoked as a method of that object and the prototype property is set
- All properties and methods of the prototype are then "available" to the new object
- When a property or a method is invoked on an object, JavaScript first checks the object directly, if not found there, it continues to check up the prototype chain

78

* Except for the object at the top of the prototype chain

PROTOTYPES

UNDERSTANDING PROTOTYPES

- The loc object prototype can be accessed via the constructor function prototype property or via the special property __proto__ of the object itself
- As we can see the prototype for loc is Object.prototype

```
print(loc.constructor.prototype);

print(loc.__proto__);

print(Object.prototype);
```

part_1/examples/prototype_object.js

```
[object Object]
[object Object]
[object Object]
```


A new “class” method Object.getPrototypeOf(object) is available int the ECMAScript 3.1 (JavaScript 1.8.1) specification and currently available in Firefox 3.5+ and Chrome 5+

PROTOTYPES

CONSTRUCTOR FUNCTIONS

- The constructor function below creates a new **Rectangle** object
- It expects 2 parameters; **base** and **height**
- The third property **area** is set to an anonymous function that calculates the area

```
var Rectangle = function(base, height) {
    this.base = base;
    this.height = height;
    this.area =  function() {
      return this.base * this.height;
    }
}

var rectangleOne = new Rectangle(5, 6);
var rectangleTwo = new Rectangle(2, 4);

print(rectangleOne.area());
print(rectangleTwo.area());
```

part_1/examples/rectangle_1.js

rectangleOne	
base	5
height	6
Object.prototype	
constructor	Object

rectangleOne	
base	2
height	4
Object.prototype	
constructor	Object

```
30
8
```


Constructor functions are like just regular functions (except that they typically do not explicitly return a value). What makes them construct anything is when we pair them with the new operator. Inside of the functions the object this refers to the object that will be returned by the constructor function.
By convention we name constructor functions starting with upper case

PROTOTYPES

UNDERSTANDING PROTOTYPES

- We can use the `hasOwnProperty` method to see which properties "live" in the object and which ones "live" somewhere in the prototype chain
- As we can see all 3 properties reside in the `rectangleOne` object

```
print(rectangleOne.hasOwnProperty("base"));
print(rectangleOne.hasOwnProperty("height"));
print(rectangleOne.hasOwnProperty("area"));
```

part_1/examples/rectangle_1.js

```
true
true
true
```

PROTOTYPES

UNDERSTANDING PROTOTYPES

- If we assign the **area** function to the **prototype** object of **Rectangle** we accomplish the same localized result; that is every **Rectangle** can calculate its own area:

```
var Rectangle = function(base, height) {
    this.base = base;
    this.height = height;
}

Rectangle.prototype.area = function() {
    return this.base * this.height;
}

var rectangleOne = new Rectangle(5, 6);
var rectangleTwo = new Rectangle(2, 4);

print(rectangleOne.area());
print(rectangleTwo.area());
```

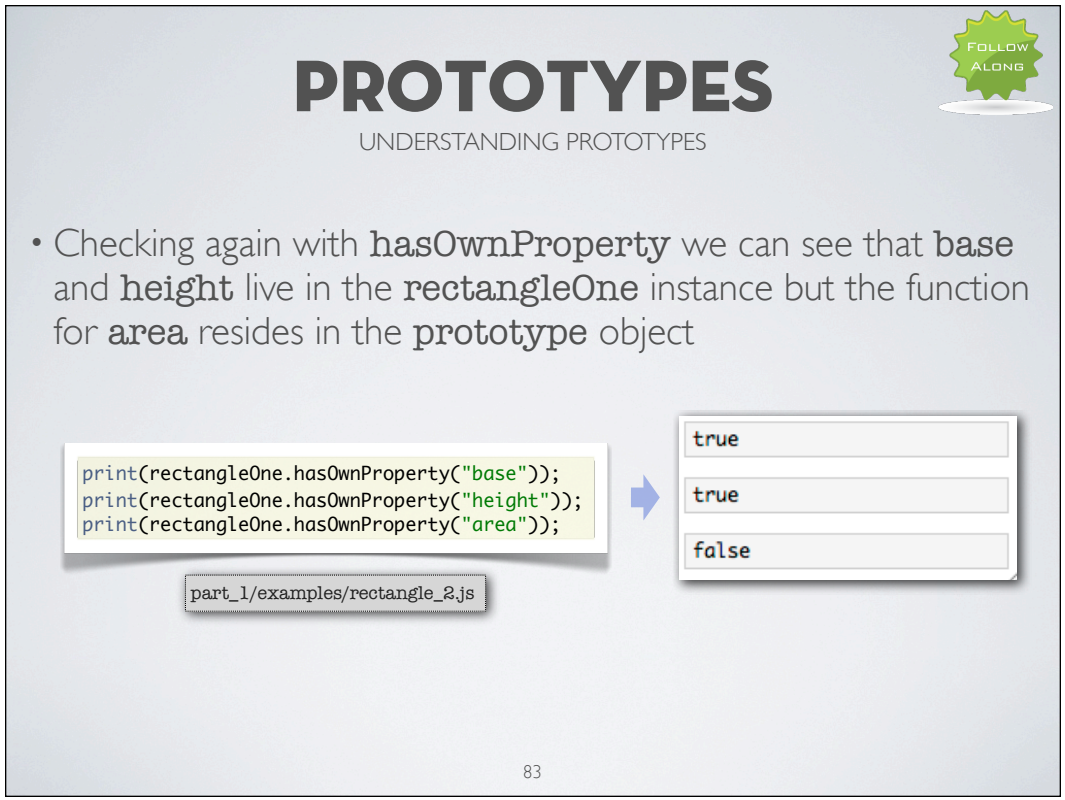
part_1/examples/rectangle_2.js

rectangleOne	
base	5
height	6

rectangleTwo	
base	2
height	4

```
30

8
```



The important distinction here is that there is only one instance of the area method which now lives in the prototype

PROTOTYPES

UNDERSTANDING PROTOTYPES

- Let's create a **Box** constructor function that takes 3 parameters; **base**, **height** and **depth**
- The function then calls the constructor function of **Rectangle** in the context of **this** (the object being created)
- We set the **prototype** of **Box** to be a **new Rectangle**
- Finally we add a **volume** method to the **prototype** of **Box**

```
var Box = function(base, height, depth) {
    Rectangle.call( this, base, height);
    this.depth = depth;
}
Box.prototype = new Rectangle();

Box.prototype.volume = function() {
  return this.area() * this.depth;
}

var myBox = new Box(5, 6, 3);

print("myBox area  ==> " + myBox.area());
print("myBox volume ==> " + myBox.volume());
```

```
myBox area  ==> 30

myBox volume ==> 90
```

part_1/examples/rectangle_and_box.js

PROTOTYPES

UNDERSTANDING PROTOTYPES

- But even though we can fake classical inheritance it doesn't mean that we need it
- Deep or broad class hierarchies have proven impractical
- Successful developers treat JavaScript as a functional programming language; you should too!

"I now see my early attempts to support the classical [inheritance] model in JavaScript as a mistake."

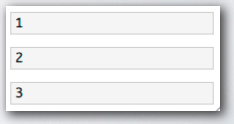
Douglas Crockford ,Yahoo!

LAB 1.3

OPEN OBJECT AND PROTOTYPES

- In Lab `1.3` you are asked to enhance the core JavaScript `Array` to:
  - Implement an Ruby-style iterator with a method called `each`
  - The `each` method will take a function as a parameter and invoke it once for each element in the array, passing that element as a parameter

```
[1, 2, 3].each(function() { print(this); });
```



SCOPE

OBJECT VISIBILITY AND LIFESPAN

SCOPE

OBJECT VISIBILITY AND LIFESPAN

- Scope in JavaScript is not as straight forward as other C-syntax like languages
- In JavaScript we have blocks but we **do not have block scope**, that is, values defined inside of a `{}` are not necessarily inaccessible from outside the block
- Instead JavaScript has **"function" scope**; parameters and variables defined in a function are not visible outside of the function
- In JavaScript every variable is global unless it is defined inside of a function declaration using the **var** keyword

SCOPE

OBJECT VISIBILITY AND LIFESPAN

- The **"function" scope** is also intimately tied to the **"execution" scope** (`this`). Before we said that all methods are functions, it also turns out that all functions are methods in JavaScript
- When the interpreter starts up it creates a global object. In a browser it is the `Window` object.

```
print(this);
```

```
[object Window]
```

part_1/examples/global_object.js

The global object (regardless of the environment) is initialized by the interpreter with a number of objects (properties and methods)

SCOPE

OBJECT VISIBILITY AND LIFESPAN

- Within the body of a JavaScript function a temporary "call object" is created that has two purposes

  1. **Provide access to the Arguments Object:** The arguments property is exposed via the Call Object as a property

  2. **Administer the Scope Chain:** The Call Object exists temporarily until the function has finished execution. The Call Objects places local variables, named parameters and the Arguments Object on to the scope chain

SCOPE

OBJECT VISIBILITY AND LIFESPAN

- The Call Object sits in front of the Global Object and places local function variables ahead of global variables in the scope chain
- Each function invocation has it own Call Object guaranteeing that nested functions will maintain 'variable' integrity
- As call Objects pop off the scope chain, the variables of lower objects rise to the top and are accessible from the appropriate scope

SCOPE

OBJECT VISIBILITY AND LIFESPAN

- Let's review some of the implications of scope:

```
var global_1 = "Hello";
global_2 = "Dolly!"
global_3 = 1;

print("global_1 ==> " + global_1);
print("global_2 ==> " + global_2);
print("this.global_1 ==> " + this.global_1);
print("this.global_2 ==> " + this.global_2);
print("this.global_function(global_1, global_2) ==> "
      + this.global_function(global_1, global_2, global_3));
print("global_1 ==> " + global_1);
print("global_3 ==> " + global_3);
print("global_4 ==> " + global_4);

function global_function(value_1, value_2) {
    var local_to_the_function = value_1 + " " + value_2;
    global_4 = "Oh no I shouldn't be out here!";
    var global_3 = "I not really global_3";
    value_1 = "Hola";
    return local_to_the_function;
}
```

```
global_1 ==> Hello
global_2 ==> Dolly!
this.global_1 ==> Hello
this.global_2 ==> Dolly!
this.global_function(global_1, global_2) ==> Hello Dolly!
global_1 ==> Hello
global_3 ==> 1
global_4 ==> Oh no I shouldn't be out here!
```

part_1/examples/scope.js

SCOPE

OBJECT VISIBILITY AND LIFESPAN

- In a method the **this** variable points to the owning object:

```
var an_object = {
    a_method: function() {
        print(this);
    }
}
an_object.a_method();
```

part_1/examples/this_in_a_method.js

```
[object Object]
```

SCOPE

CALL AND APPLY

- The value of `this` can be overridden during a function call by using the `call` and `apply` methods
- The `call` method calls a function with a given `this` and a list of arguments
- The `apply` method calls a function with a given `this` and arguments passed as an array
- The `call` and `apply` methods are heavily used by JavaScript frameworks to override the value of `this`

SCOPE

CALL AND APPLY

- The **call** and **apply** methods can be used to apply a "detached" function to a particular object:

```
On 09/13/2010 at 4:13 PM the Ford Pinto says vroom!
On 09/14/2010 at 5:15 PM the Ford Pinto says vroom!
```

```
function start(date, time) {
    print(["On", date, "at", time,
           "the", this.make, this.model,
           "says", "vroom!"].join(" "));
}

var pinto = {
    make : "Ford",
    model : "Pinto",
};

start.apply( pinto, [ "09/13/2010", "4:13 PM" ] );
start.call( pinto,  "09/14/2010", "5:15 PM" );
```

part_1/examples/call_and_apply.js

CLOSURES

FUNCTIONAL PROGRAMMING

CLOSURES

FUNCTIONAL PROGRAMMING

"Without understanding functional programming, you can't invent MapReduce, the algorithm that makes Google so massively scalable"

Joel Spolsky, Joel on Software

CLOSURES

FUNCTIONAL PROGRAMMING

- A closure is a first-class function with free variables that are bound in the lexical environment. The function is said to be "closed over" the free variables.
- A JavaScript closure is a function that is created within the execution context of another function and it effectively closes over any variables declared in the "parent" function.
- In summary, a closure is a nested function in which the inner function has access to the enclosing function variables. Think of a closure as an object with a single method (the state is the bound variables and the method is the function itself).

CLOSURES

FUNCTIONAL PROGRAMMING

- A closure factory; showing how variables local to the function are bound with the closure:

```
function closure_maker(closure_maker_argument) {
    var local_to_the_function = "and";
    return function(closure_arg) {
        return ([closure_maker_argument,
                 local_to_the_function,
                 closure_arg]).join(' ');
    }
}

var closure_1 = closure_maker("Benny");
var closure_2 = closure_maker("Johnny");

print(closure_1("Jets"));
print(closure_2("Chachi"));
print(closure_1("Joon"));
print(closure_2("June"));
```

part_1/examples/closure_maker.js

```
Benny and Jets
Johnny and Chachi
Benny and Joon
Johnny and June
```

CLOSURES

FUNCTIONAL PROGRAMMING

- Closures can be used to:
  - Create callback functions
  - Associate functions with object instance methods
  - Encapsulate related functionality
  - Emulate private methods

LAB 1.4

CLOSURES

- In Lab 1.4 you are asked to fix the application below to so that an invocation of the **describe** method returns the correct index for an object in the array:

```
objects = [{}, {}, {}];
for (var index in objects) {
    objects[index].describe = function() {
        print('This object has an index of ' + index);
    }
}

objects[0].describe();
objects[1].describe();
objects[2].describe();
```

part_1/examples/broken_scope.js

```
This object has an index of 2
This object has an index of 2
This object has an index of 2
```

CURRYING

FUNCTIONAL PROGRAMMING

- Currying is the ability to partially apply a function by passing only some of its parameters. The function becomes the holder of state
- The function is transformed into a chain of functions each with a single argument
- What can you do with Currying?
  - Functional composition: Create a base function that can be use to implement a family of functions
  - Cleaner more efficient callbacks by refining a higher level function

CURRYING

FUNCTIONAL PROGRAMMING

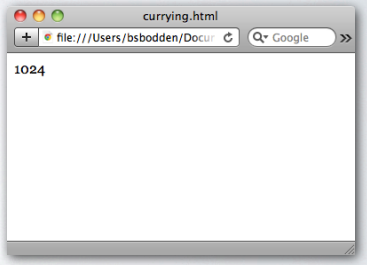
- A simple example of currying; partial application is used to create a new function:

```
function multiply(a) {
    return function(b) {
        return a * b;
    }
}

// create a new function by using
partial application
var doubleIt = multiply(2);

// use the new function
print(doubleIt(512));
```

part_1/examples/currying.js



JSON

JAVASCRIPT OBJECT NOTATION

JSON

JAVASCRIPT OBJECT NOTATION

- The JavaScript Object Notation or JSON is a simple lightweight hierarchical text data-interchange format
- It's easy for humans to read and write and it is easy for machines to parse and generate (move over XML)
- It's programming language independent!
- It uses pure JavaScript object literals as a data envelope; effectively a collection of name/value pairs (JSON is a subset of the object literal notation of JavaScript)
- JSON became a standard feature of JavaScript in the Fifth Edition of ECMAScript Standard in 2009

JSON

JAVASCRIPT OBJECT NOTATION

- JSON requires that properties are quoted.
  Like `{ "answer" : 42 }`
- JSON is implicitly typed by supporting JavaScripts literal values for strings, numbers, array and boolean

```
var data = {
    "aList" : [1, 3, 5, 7, 11],
    "anotherObject" : { "a": 1, "b": 2, "c": 3}
};

print(data.anotherObject.b);
print(data.aList[2]);
```

```
2
5
```

part_1/examples/json_1.js

JSON

JAVASCRIPT OBJECT NOTATION

- Most modern browsers posses a JSON parser that can be used to reliably parse JSON data:

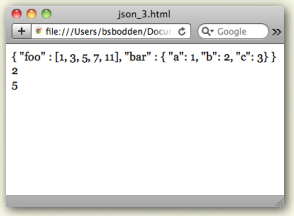
part_1/examples/json_3.html

```
<html>
    <head>
        <script src="print_function.js"></script>
        <script>
        // <![CDATA[
        var rawData = '{ "foo" : [1, 3, 5, 7, 11], "bar" : { "a": 1, "b": 2, "c": 3} }';

        print(rawData);

        var data = JSON.parse(rawData);

        print(data.bar.b);
        print(data.foo[2]);
        // ]]>
        </script>
    </head>
    <body>
    </body>
</html>
```



For older browsers, check for the existence of the JSON object and if not available load a custom parser such as http://ajax.cdnjs.com/ajax/libs/json2/20110223/json2.js

CODE ORGANIZATION

STRUCTURING YOUR JAVASCRIPT CODE

CODE ORGANIZATION

STRUCTURING YOUR JAVASCRIPT CODE

- There are many techniques that can help you keep a project using HTML, JavaScript and CSS organized, in this section we will cover some of the ones that I've successfully used in the past:
  - **Code Structure:** Organizing your JS classes and functions
  - **File Organization:** Directory structures and file naming
  - **Page Structure:** How to structure your HTML pages

CODE STRUCTURE

MODULE PATTERN

- The Module Pattern, first introduced by Douglas Crockford is a way to namespace and encapsulate your JavaScript classes and functions
- Why do we need the Module Pattern?
  - JavaScript doesn't posses the scoping abilities of other languages; it only has function scope
  - Everything in JS is public and cluttering the global space can lead to very hard to maintain code

CODE STRUCTURE

MODULE PATTERN

- The Module Pattern uses closures to create private functions
- It makes use the self-executing anonymous function:

```
(function() {
  // everything here is 'private'
  // but it has access to any globals
})();
```

- Remember that the result of creating an anonymous function with the function operator is a `Function` object and the code inside the outer function lives in a closure

CODE STRUCTURE

MODULE PATTERN

- The simplest application of the Module Pattern allows us to wrap a "class" in a name space
- Let say we wanted to implement a library of data structures, we could use the module pattern as follows:

```
var INTEGRALLIS = {};
INTEGRALLIS.structures = (function() {
    var Stack = function() {
       // ...
    }

    return { Stack : Stack }
})();
```

CODE STRUCTURE

MODULE PATTERN

```
var INTEGRALLIS = {};
INTEGRALLIS.structures = (function() {
    var Stack = function(options) {
        this.push = function(element) {
            elements.push(element);
        }

        this.pop = function() {
            return elements.pop();
        }

        this.size = function() {
            return elements.length;
        }

        this.clear = function() {
            elements = [];
        }

        this.toString = function() {
            return '[ ' + elements.join(',') + ' ]';
        }

        // initializes the stack
        var elements = [];
    }

    return { Stack : Stack }
})();
```

module_pattern/integrallis.structures.js

- The global variable **INTEGRALLIS** is a empty object
- We define a property called **structures** which is an anonymous self-executing function
- Inside we create the variable **Stack**, a constructor function
- Inside we add behavior (methods) and 'private' state (properties) for the stack
- The return value is an object with named properties after the 'classes' made available in our module

This particular implementation of the module pattern is sometimes called the "revealing module pattern"

CODE STRUCTURE

MODULE PATTERN

```
<html>
    <head>
        <script src="../print_function.js"></script>
        <script src="integrallis.structures.js"></script>
    </head>
    <body>
        <h1>Stack</h1>
        <script>
        print("<b>Creating the stack =></b>");
        var myStack = new INTEGRALLIS.structures.Stack();

        print("<br/><b>Pushing 1..5</b>");
        var i = 0;
        for (i = 0; i <= 5; i++) {
            myStack.push(i);
        }

        print(myStack.toString());

        print("<br/><b>Popping one off the stack</b>")
        print(myStack.pop());
        print(myStack.toString());

        print("<br/><b>Clearing the stack</b>")
        myStack.clear();
        print(myStack.toString());
        </script>
    </body>
</html>
```

module_pattern/integrallis_stack.html

- To use the module we include the script file integrallis.structures.js
- I tend to name the file following the "namespaces" separated by periods
- To create a new Stack object we use the new operator and the full namespace (INTEGRALLIS.structures.Stack)

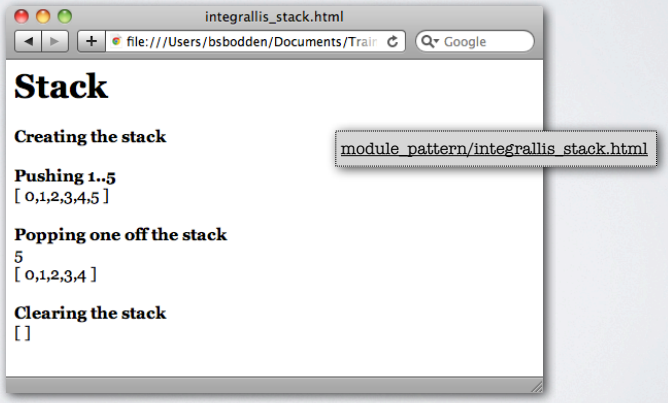
later in the course we'll revisit the module pattern in the context of jQuery

For some extra reading on the module pattern see:
- http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth
- http://yuiblog.com/blog/2007/06/12/module-pattern/

CODE STRUCTURE

MODULE PATTERN

- The example shows the stack being loaded with a few elements using the push method, popping an element, clearing and printing its contents



CODE STRUCTURE

MODULE PATTERN

- Some frameworks provide implementations of the module pattern such as YUI library*
- The module pattern is not without it drawbacks:
  - It makes it hard to modify a class indirectly (patch)
  - The hard encapsulation provided by it, its not always necessary nor desirable
  - Is sometimes hard to determine the value of the this keyword
  - Creating multiple instances of the class consumes more memory

*See http://developer.yahoo.com/yui/3/yui/

CODE STRUCTURE

PROTOTYPE PATTERN

- To create the namespace we create empty objects and nest them
- The Prototype pattern is nothing more than attaching methods to the **prototype** object of a **function**

```
var INTEGRALLIS = {}
INTEGRALLIS.structures = {}

INTEGRALLIS.structures.Stack = function() {
    // initialize state here!
};

INTEGRALLIS.structures.Stack.prototype.aMethod = function() {
    //...
}
```

CODE STRUCTURE

PROTOTYPE PATTERN

```
var INTEGRALLIS = {}
INTEGRALLIS.structures = {}

INTEGRALLIS.structures.Stack = function() {
    this.elements = [];
};

INTEGRALLIS.structures.Stack.prototype.push = function(element) {
    this.elements.push(element);
}

INTEGRALLIS.structures.Stack.prototype.pop = function() {
    return this.elements.pop();
}

INTEGRALLIS.structures.Stack.prototype.size = function() {
    return this.elements.length;
}

INTEGRALLIS.structures.Stack.prototype.clear = function() {
    this.elements = [];
}

INTEGRALLIS.structures.Stack.prototype.toString = function() {
    return '[ ' + this.elements.join(',') + ' ]';
}
```

prototype_pattern/integrallis.structures.js

FILE ORGANIZATION

DIRECTORY STRUCTURE AND FILE NAMING

- There are no industry-wide standards on how to organize your project files in an HTML/CSS/JavaScript project but many frameworks tend to have their own set of guidelines, below are some of the ones that I like:
  - Keep all JS files separate from the rest of the application in a **'scripts'**, **'javascripts'** or **'js'** directory (name is not as important)
  - If a script is specific to a page name the page and the script the same (e.g. **mypage.html** and **mypage.js**)
  - Use directories to separate visual versus non-visual scripts, plugins. If using a framework separate plugins, extensions, etc.

PAGE STRUCTURE

ORGANIZING YOUR HTML PAGE STRUCTURE

- The ideal HTML page structure:

```
<html>
    <head>
        <!-- A Single Combined StyleSheet -->
        <link rel="stylesheet" href="styles/style.css">
    </head>
    <body>
        <div id="content">

        </div>

        <!-- A Single External JS Resource -->
        <script src="javascripts/scripts.js"></script>

        <!-- Page Expecific Embedded JS if necessary -->
        <script language="javascript">
            // ...
        </script>
    </body>
</html>
```

ideal.html

- As a general rule is a good idea to coalesce all stylesheets and JavaScript resources under a single file
- Stylesheets should be loaded at the top of the page in the HEAD section
- JavaScript files should be loaded in the body of the page right before the closing of the body tag

120

- A great deal of information is available at Steve Souders website http://stevesouders.com
- His books “High Performance Web Sites” and “Even Faster Web Sites” are the definitive guides for structuring your web pages and optimizing their loading
- Steve’s tool Cuzillion (http://stevesouders.com/cuzillion/) can help you build and test a page
- Use YSlow (http://developer.yahoo.com/yslow/) a FireFox Add-On to analyze web pages. FireFox Add-On site https://addons.mozilla.org/en-US/firefox/addon/yslow/
- Read about the Best Practices for Speeding Up Your Website at http://developer.yahoo.com/performance/rules.html