

44-563: Unit 03

Developing Web Applications and Services

Includes

- Schedule
- Workshops
- Motivation
- Javascript - a popular, powerful, widely-used scripting language used by all modern browsers - and more
- Git

Schedule

Wk	Topics
1	Intro, static pages, HTML5, CSS3
2	Holiday (no class M) Responsive design
3	JavaScript, A01
4	DOM, JQuery, Workshop 1, A02
5	Node.js, Exam 1 (See E01 review guide)

Exam 1 covers first 4 weeks:
HTML / CSS / Responsive design /
BootStrap / JS / DOM / JQuery

WWS: See Schedule

Whirlwind Workshops are:

- 10-15 minutes (7-10 slides + **demo**)
- Slide 1: title & team slide (with pictures & names)
- Slide 2: intro slide (list **key points**)
- Slides 3 to 5-9 content slides
- Demo/hands on activity
- Slide n-1: (list **key points** again)
- Slide n: Questions (repeat team slide w/pictures & names)

WWS More Information

Motivate and act as lead
resources for the topic.

Teams & topics (course site:
use filter to see your section)

Use **NWMSU** slides at
<http://www.nwmissouri.edu/marketing/design/templates.htm>

General NWMSU design at
<http://www.nwmissouri.edu/marketing/design/index.htm>



Today: The last of the 3 foundational web technologies



HTML for content
CSS for style
JS for event handling

**What's so
awesome about
JavaScript?**

Interactive Web Pages

- The web is much more than just displaying information
- Web pages can be **interactive**
- **Interactive web pages** respond to user actions (e.g. clicking a button)
- But how do we add logic so that we can do amazing things when events happen? How do we handle these events?

Handling events

- We can handle all kinds of events with *JavaScript!*
- Code that **handles events** is written in **JavaScript** and executed by the browser.
- This code is contained in methods called **event handlers**.
- Event handlers are methods that perform the operations needed when the user does things (like click a button or fill in forms)!

What else can JS do?

JavaScript can:

- change all HTML **elements**
- change all HTML **attributes**
- change all CSS **styles**
- **remove** HTML elements & attributes
- **add** HTML elements & attributes
- **react** to existing HTML events
- **create** HTML events

*Even the
<html>
element!*

This code

will change this element

Foreshadowing:

```
document.getElementById("greeting").innerHTML = "Hello, DOM!"  
<p id="greeting">This space for rent, reasonable rates, inquire within</p>
```

Adding JavaScript

Adding JS to HTML

- You have done this already!
- Use **<script>** ... **</script>** tags
- Use the **src** attribute when the code is in an external file.
- Order **matters** (execution starts at the top and goes down the html file):
 - Put dependencies first, before any code that uses them
 - jQuery before Bootstrap (Bootstrap uses jQuery)
 - Put **<script>** tags just before **</body>**: browsers process from top-to-bottom, so the entire page will be displayed before any potentially script processing begins

```
</div>
<!-- /container -->
<script src="https://code.jquery.com/jquery-3.1.0.min.js"></script>
<script src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.7/js/bootstrap.min.js"></script>
<script src="M03.js"></script>
</body>

</html>
```

Note: software engineering

In this course, we follow the "***separation of concerns***" design principle.

Keep HTML/CSS/JS in **different files**

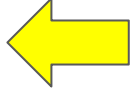
- **Separates** content, style, and event-handling code
- Easier to maintain
- Easier to apply **uniform style** to whole web site
- Required for A01, A02

JavaScript Concise Introduction

JavaScript

- JavaScript is a **programming language** implemented by browsers.
- It is NOT Java (and is not subset of Java).
- JS is a **scripting language**.
- It is read and interpreted at runtime and runs in a specific environment (in this case, a browser).
- A clear and concise re-introduction to JS:
https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

Recommended reading

- O'Reilly's JavaScript: The Good Parts book
<https://www.amazon.com/exec/obidos/ASIN/0596517742/wrrrldwideweb>
- Eloquent Javascript
 - Available at Owens Library for you to check out
 - Free online: <http://eloquentjavascript.net>  Fun!
- For more experienced programmers:
 - <https://www.codementor.io/johnnyb/tutorials/javascript-cheatsheet-fb54lz08k>
- W3Schools' [JavaScript tutorial](#)

JS: the turbo introduction*

The following slides will (hopefully) get you up and running with JavaScript. This is not an exhaustive introduction to JS; you will need to continue to explore, experiment (and, of course, break things) to truly master the language.

You can get a headstart look at the M assignment this week by going to <http://denisecase.github.io/2016/09/11/learning-js-with-qunit/> (with the [initial test results](#))

It looks more complicated than it actually is! Don't get too lost in the assignment until you know more about JavaScript

* Not to be confused with Turbo Pascal, etc..

Programming languages

JavaScript is a **Programming Language** (like Java, though they are not the same). These allow us to instruct the computer to perform a **sequence** of instructions

```
var age = window.prompt("Age?")
var ageInDogYears = age * 7
window.alert("Age in dog years " + ageInDogYears + " years old")
```

We can **repeat** commands with **loops**:

```
while(awake) {doSomething() }
```

We can **branch** based on **conditions**

```
if (rainy) {getUmbrella()} else {playCricket() }
```

We will use **variables** in JavaScript to store intermediate values

Makes sense, right? We'll do a quick overview of key features and then get into code. It'll show what's available. You can always come back to get the specifics.

Types

JavaScript Types

- **Number** (all floating points - no ints)
- **String** (no char)
- **Boolean** (true / false)
- **Function**
- **Object** (Array is a special type of object)
- **Symbol** (new in Edition 6)
- Also, **undefined** and **null**.

Helpful **typeof** operator:

```
typeof 37 === 'number'  
typeof undefined === 'undefined'  
typeof {a:1} === 'object'  
typeof [1, 2, 4] === 'object'  
typeof new Date() === 'object'
```

Strings and Special Types

- `null`: Deliberate non-value
- `undefined`: Uninitialized value
- All of these mean `false`
 - `0`, `null`, `undefined`, empty strings (`""`), Not A Number (`NaN`)

```
"hello".length // 5
```

```
"hello".charAt(0) // "h"
```

```
"hello, world".replace("hello", "goodbye") // "goodbye, world"
```

```
"hello".toUpperCase() // "HELLO"
```

```
"hello".endsWith("lo") // true
```

```
Boolean("") // false
```

```
Boolean(234) // true
```

For more String fun, see the [String API](#)

JS Types - Mnemonics

NSF* Bonus:

N = Number

S = String

F = Function

B = Boolean

O = Object

N = Null

U = Undefined

S = Symbol

Fun Snobs:

F = Function

U = Undefined

N = Number

S = String

N = Null

O = Object

B = Boolean

S = Symbol

**NSF = National Science Foundation*

Variables & Scope

Declaring Variables

- JavaScript uses dynamic types. It is not statically typed like Java or C#.
- There are three ways to declare “variables” in JavaScript:
 - `const`
 - `let`
 - `Var` (older)
- Using **const** declares a “variable” that cannot be reassigned

```
const PI = 3.14159
```


Understanding Scope

- **Scope** - The part of code where a variable name applies
- Not too difficult!
 - In general, if a variable is defined inside a **block** (some code enclosed in { }) it will exist from the point it is defined until the closing curly bracket (}).
 - If a variable is defined and it is *not inside a block*, it will be available in any code **below** it in the file.
 - These variables have **global scope**

Working with variables

- In JavaScript, do **not** specify the type of the variable; just use the `let` or `var` or `const` keywords

```
let name = "bob"  
var score = 214
```

- There are important differences between `let` and `var`
- The scope of a variable declared with `let` is restricted to the block in which it was defined
- Variables defined with `var` will exist in blocks *within* the block of definition

Good coding practice:

- Proper indentation is vital to making your code readable
- You should use `let` whenever possible to limit scope

let / var / const

var has been around since JavaScript was created - it is widely used

let and **const** are new as of ES2015.

A **var** variable is available in the scope of the enclosing function, or if there is none, globally.

A **let** variable is available within the scope of its enclosing { }. Its scope can be narrower than var's.

A **const** scope is the same as let; but its value cannot be changed.

If you forget to use var, let or const, a variable with global scope will be created. This is a bad idea, or in the words of MDN's JavaScript Guide, "you shouldn't use this variant".

"var x" versus "let x"

```
function varTest() {  
  var x = 1;  
  if (true) {  
    var x = 2; // same variable!  
    console.log(x); // what is x?  
  }  
  console.log(x); // what is x?  
}
```

```
function letTest() {  
  let x = 1;  
  if (true) {  
    let x = 2; // different variable  
    console.log(x); // what is x?  
  }  
  console.log(x); // what is x?  
}
```

"var x" versus "let x"

```
function varTest() {  
  var x = 1;  
  if (true) {  
    var x = 2; // same variable!  
    console.log(x); // 2  
  }  
  console.log(x); // 2  
}  
  
function letTest() {  
  let x = 1;  
  if (true) {  
    let x = 2; // different variable  
    console.log(x); // 2  
  }  
  console.log(x); // 1  
}
```

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/let>



Always favor
the more
specific "let".

Global
variables (or
broader scope)
are a widely
recognized
"code smell".

Basic Types

Numbers (& the Math object)

All numbers are floating point (no integers)

`0.1 + 0.2 = 0.30000000000000004`

```
Math.sin(3.5)
```

```
var circumference = Math.PI * (r + r)
```

```
parseInt("123", 10) // 123 base 10
```

```
parseInt("010", 10) // 10 base 10
```

```
parseInt("11", 2) // 3 binary
```

```
parseInt("2CA", 16) // 2*256 + 12*16 + 10*1 = 714 (base 16 □)
```

```
parseInt("hello", 10) // Special value "Not a Number"
```

```
isNaN(NaN) // true
```

```
1 / 0 // Infinity
```

```
-1 / 0 // -Infinity
```

```
isFinite(1/0) // false
```



Handy because user inputs are always delivered as text: we need to convert them to numbers before we can do useful things with them.

Techy Aside: parseInt()

JS's parseInt() behaves differently from Java's

```
parseInt("123")           // returns 123 (as expected)
parseInt("123Bravo")      // returns 123 (Java would revolt at this)
parseInt("Bravo123")      // returns NaN (not a Number)
```

What if you really want to know if you have a number? It's [regular expressions](#) to the rescue!!

```
function isInt(value) {
  if (/^(\-|\+)?([0-9]+|Infinity)$/.test(value))
    return Number(value);
  return NaN;
}
```

```
isInt("123")              // returns 123
isInt("123Bravo")         // returns NaN
isInt("Bravo123")         // returns NaN
```


Strings & Special Types

`null` - deliberate non-value

`undefined` - uninitialized value

`false` includes `0`, empty strings (`""`), `NaN`, `null`, `undefined`

```
"hello".length // 5
```

```
"hello".charAt(0) // "h"
```

```
"hello, world".replace("hello", "goodbye") // "goodbye, world"
```

```
"hello".toUpperCase() // "HELLO"
```

```
"hello".endsWith("lo") // true
```

```
Boolean("") // false
```

```
Boolean(234) // true
```

For more String fun, see the [String API](#)

Comparing Values

Comparing values

- JS provides (almost) the same comparison operators as other programming languages `<` `>` `<=` `>=`
- Comparison operators return `true` or `false`

Equality operators

- JavaScript has two equality operators, `==` and `===`
 - `===` is for **strict** equality: two operands must be **same type** and have the **same value**, for `===` to return true
 - `==` is for **loose** equality: two operands are converted to a **common type** and converted values are compared.
- For example, `42 == "42"` is true, whereas `42 === "42"` is false

(almost)

Always use **triple equals** `===` or `!==`

Read & practice on your own

http://eloquentjavascript.net/01_values.html

Branching
(taking different
paths)

if-else branching

```
var name = "kittens"

//should these be == or ===?

if (name == "puppies") {
    name += "!"
} else if (name == "kittens") {
    name += "!!"
} else {
    name = "!" + name
}

// name == "kittens!!"
```



switch branching



```
switch(action) {  
  case 'draw':  
    drawIt()  
    break           // important! exits switch block  
  case 'chow down': // label if you really want to fall thru  
  case 'eat':  
    eatIt()  
    break  
  default:  
    doNothing()  
}
```


Syntax: if & if else

JavaScript allows us to change the behavior of our program based on the results of a comparison

Structure of an if statement

```
if ( EXPR ) {  
    STATEMENTS to run if expr is true  
}
```

Structure of an if-else statement

```
if ( EXPR ) {  
    STATEMENTS to run if expr is true  
}  
else {  
    STATEMENTS to run if expr is false  
}
```

Syntax: if else if else

Structure of an if-else if-else statement

```
if ( EXPR1 ){  
    STATEMENTS to run if expr1 is true  
}  
else if ( EXPR2 ){  
    STATEMENTS to run if expr1 is false and expr2 is true  
}  
else {  
    STATEMENTS to run if expr1 and expr2 are both false  
}
```

Repeating

do/do-while looping



```
while (true) {  
    // an infinite loop!  
}
```

Give me a break or listen for a kill event ...
literally: otherwise I'll never get out of here

```
while (alive) {  
    play_cricket();  
}
```

Note: this condition is not variable - **true is always true - it is not a variable whose value can be updated**; this is truly an infinite loop.

```
var input  
do {  
    input = get_input()  
} while (inputIsValid(input))
```

Syntax: do & do-while (sentinel loop)

Structure of a do-while loop

```
do {  
    STATEMENTS to run  
} while (EXPR)
```

Structure of a while loop

```
while (EXPR) {  
    STATEMENTS to run  
}
```

for looping



// how many times will these loops execute?

```
for (var i = 0; i < 5; i++) {  
    console.log(i)  
}  
  
for (var i = 0; i <= 10; i+=2){  
    console.log(i)  
}
```

Syntax: for loop (counting)

Structure of a for loop

```
for (INIT; CHECK; UPDATE) {  
    STATEMENTS to run  
}
```

When encountering the for loop, the program will:

1. Run the INIT expression (creating the LCV)
2. Run the CHECK statement
 - a. If it is true, go to step 3
 - b. If it is false, go to step 6
3. Run the STATEMENTS inside the loop body
4. Run the UPDATE statement
5. Go to step 2
6. Continue the program

Things to remember w/loops

- Two categories of loops (in JS and many programming languages):
 - Sentinel Loops
 - Counting Loops
- **Sentinel** Loops (examples: while/do while loops) run as long as some condition is true
- **Counting** Loops (example: for loop) run for a specified number of times
- When writing loops, be sure the loop control variable or condition has been initialized.
- During each iteration the control variable or condition will be evaluated and may be updated.
- Often, in game programming and intelligent agents, loops are **intended** to run indefinitely (e.g., until the character dies, or the program is terminated).

Functions

Functions

- Functions are some of the most powerful tools in a developer's toolkit
- They allow us to make code **reusable**
 - Remember: don't reinvent the wheel!
- They allow us to make code more **readable**
- They allow us to separate implementation "details" from our code
 - Often called "stubbing"
- Good programming practice: A function should do **one** thing
 - Most important word: one!
 - This makes it easier to test our code, find problems, and reuse code
 - Copy-paste BAD!
- A function may have two pieces:
 - Function definition - where you build it (in JS, sometimes you build it where you call it)
 - Function call - where the function is called (if ever)
- Functions can return values to the place where they were called!

Earlier example

http://www.tutorialspoint.com/javascript/javascript_dialog_boxes.htm

```
<html>
<head>
<script src="M02.js"></script>
</head>

<body>

<p>Try this: </p>

<form>
  <input type="button"
    value="Click Here!"
    onclick="warn()" />
</form>

</body>
</html>
```

```
function warn() {
  alert ("Danger, danger!")
}

function getConfirmation(){
  let retVal = confirm("Continue?")
  if( retVal == true){ return true}
  else{ return false}
}

function getName(){
  let retVal = prompt("Enter name: ", "Ramya")
}
```

How many functions are available? Which one do we call?

Function Definition: Examples

```
function echo(phrase, numTimes){  
  for(i=0 i < numTimes i++){  
    console.log(phrase)  
  }  
}
```

```
function cylinderVolume(height, radius){  
  return Math.PI * radius * radius * height  
}
```

```
function helloWorld(){  
  console.log("Hello world!")  
}
```

How many arguments in each function? Does it return a value?

Function Fundamentals

A **function** is a named set of statements that is invoked by writing its name. It consists of:

- the keyword **function**
- the function name (any legitimate JS identifier)
- a parameter list (comma-delimited identifiers)
- a body, consisting of JS statements in { }

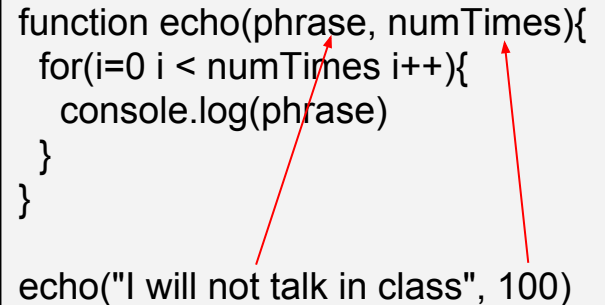
In the `echo()` example, the 2 arguments get passed into the parameters `phrase` and `numTimes`, where they are used.

In the `cylinderVolume()` example, the function **returns** a value that can be assigned or used elsewhere.

```
function cylinderVolume(height, radius){  
  return Math.PI * radius * radius * height  
}
```

```
var height = window.prompt("Cylinder height?")  
var radius = window.prompt("Cylinder radius?")  
var cylVol = cylinderVolume(height, radius)  
window.alert("Your cylinder has a volume of " + cylVol)
```

```
function echo(phrase, numTimes){  
  for(i=0; i < numTimes; i++){  
    console.log(phrase)  
  }  
}  
  
echo("I will not talk in class", 100)
```



Q: Does it matter that `height` and `radius` are used in two places (as parameters and variables)?

A: No. The *parameter* `height`, and the *variable* `height` (for example: the same applies to `radius`), have different **scope** -- they are distinct containers. It is perfectly *safe* and *sound* coding practice to reuse the names

More Function Examples

```
function doSomething(){  
    window.alert("Are you ready to rumble?? OK, how about add a range of numbers?")  
    let low = parseInt(prompt("Enter starting number"))  
    let high = parseInt(prompt("Enter an ending number"))  
    window.alert("The sum from " + low + " to " + high + " is " + sum(low,high))  
}
```

```
// returns the sum from + (from+1) + ... + to  
// from: first number in sum  
// to: last number in the sum
```

```
function sum(from, to){  
    let sum = 0  
    for (var i = from; i <= to; i++) {  
        sum = sum + i  
    }  
    return sum  
}
```

Functions in JS

Functions in JS are first class citizens: that means they can be passed into functions, returned from functions, and stored in variables

Functions can be **named** (as we just saw) or **anonymous**. With an anonymous function, you omit the name, and store the expression in a variable (or a parameter if you pass it into another function). To invoke the function, merely write the variable/parameter name, followed by arguments in ()



```
// The named approach
```

```
function sayHello(name) {  
    console.log("Hello, " + name)  
}
```

```
sayHello("Priyanka") // Hello, Priyanka
```

```
// An anonymous function - note the absence of a name  
// between the keyword function and the parameter list
```

```
var sayHowdy = function(name){  
    console.log("Howdy, " + name)  
}
```

```
sayHowdy("Mounika") // Howdy, Mounika
```

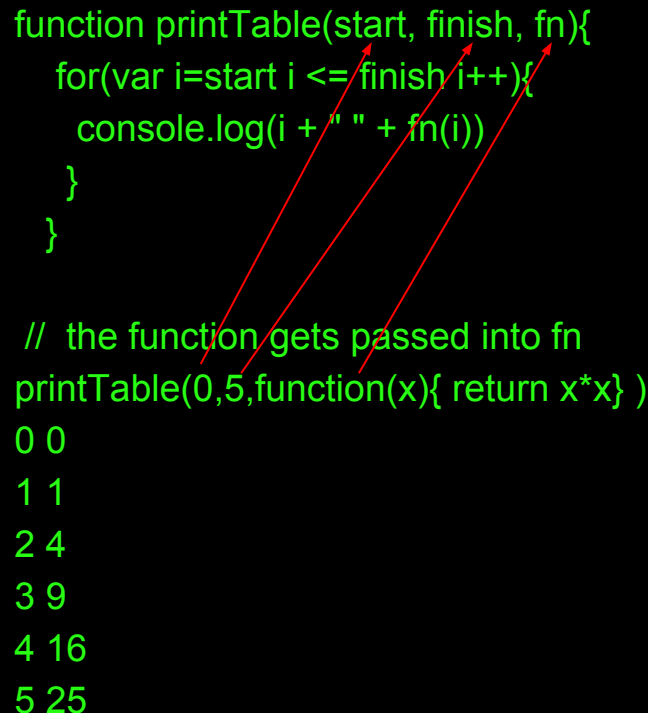
Functions in JS

Normally you would *not* assign an unnamed function to a variable. Rather, you would pass it into another function as a parameter, and invoke it internally.

```
function printTable(start, finish, fn){
  for(var i=start; i <= finish; i++){
    console.log(i + " " + fn(i))
  }
}

// the function gets passed into fn
printTable(0,5,function(x){ return x*x } )

0 0
1 1
2 4
3 9
4 16
5 25
```

A diagram with three red arrows originates from the function call `printTable(0,5,function(x){ return x*x })`. One arrow points from the first argument `0` to the parameter `start` in the function definition. A second arrow points from the second argument `5` to the parameter `finish`. A third arrow points from the third argument `function(x){ return x*x }` to the parameter `fn`.

```
setTimeout(function(){ console.log("Blastoff!") }, 5000)
// 5 seconds later ...
Blastoff!
```

Since `fn` contains a 1-parameter, anonymous function, we can write `fn(i)` in order to invoke that function.

Note that we now have the ability to tailor the behavior of `printTable()` simply by passing in a different function.

e.g.,
`printTable(1,10,function(x){return Math.sin(x)})`
`printTable(1,10,function(x){return Math.random(x)})`

Without this ability, we would have to write a different version of `printTable()` for each different function that we'd like to generate a table for -- yuck!

Let's get started (M03)

Find a neighbor or two. Only one keyboard. Open a browser and go to:

<http://denisecase.github.io/2016/09/11/learning-js-with-qunit/>



<http://www.codecademy.com/DynamicImage/Service/75cc9816-2d7e-4df0-a05c-43d03a00aeb>

Public M03 repository

<https://bitbucket.org/professorcase/m03>

The starting code for the assignment is available here.

The screenshot shows a web browser displaying the Bitbucket repository page for 'Denise Case / M03'. The browser's address bar shows the URL 'https://bitbucket.org/professorcase/m03'. The Bitbucket interface includes a navigation bar with 'Teams', 'Projects', 'Repositories', and 'Snippets'. The repository page has a yellow 'JS' icon and an 'Overview' tab. A table displays repository statistics: 'Last updated' (2016-09-11), 'Language' (JavaScript), 'Access level' (Admin), '1 Branch', and '0 Tags'. A sidebar on the right offers an 'Invite users to this repo' button and a 'Send invitation' button. The browser's taskbar at the bottom shows various open applications and a search bar.

Last updated	2016-09-11	1	0
Language	JavaScript	Branch	Tags
Access level	Admin	2	1

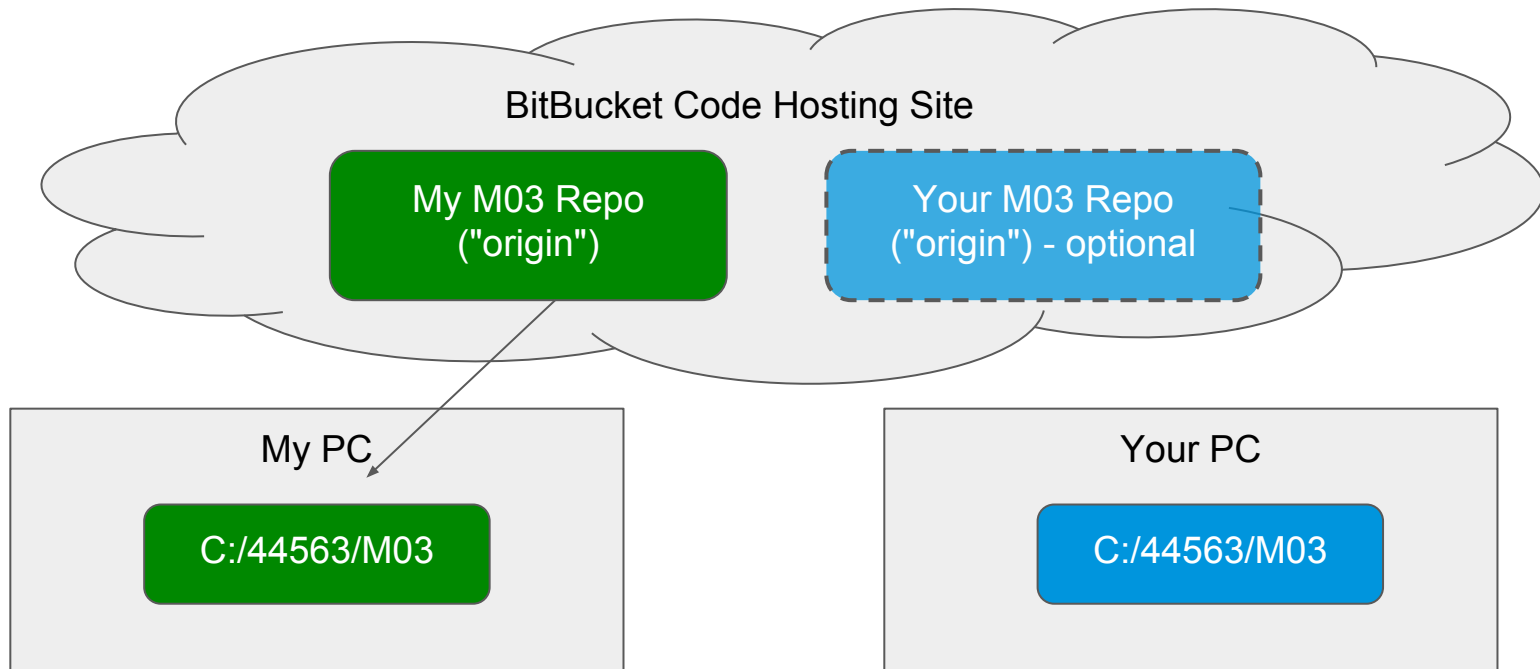
M03

Have **Git for Windows** installed.

Have **TortoiseGit** installed.

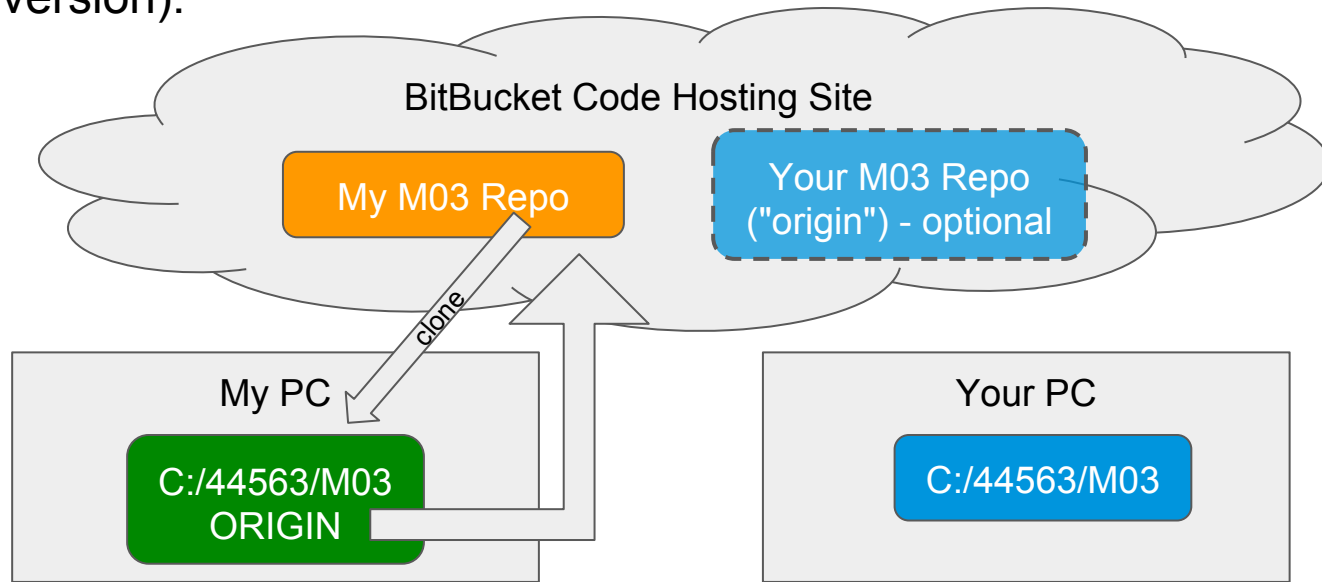
You may install SourceTree (optional).

Get a copy of the code on your computer - you will not have to push it back up to the cloud.



Techy Aside: Forks vs. Clones

- **Forks** versus **clones** (which sounds like a great WWF match, but it isn't)
- The original repo is in orange.
- The green shows a **clone**. Cloning a repo, creates a local copy with a pointer (ORIGIN) back to the origin(al) repo. Pushing would therefore send it yonder (to the origin).
- The blue shows a **fork**. When you fork a repo, you get a copy of the code, which will exist in your BitBucket account (and it has no relationship to the green version).



M03

Once you have a local working copy of M03:

1. Uncomment last two tests in **M03 / test / M03test.js**
2. Implement two new functions in **M03 / M03.js**.
3. Check your local tests to make sure all 9 tests pass.

M03 Hints

- The **initial** M03 commit contained a working solution.
- Since all git repositories contain the **full history of the code**, you should be able to explore the history of each file and discover a working solution.
- If you have troubles cloning (or creating your own BitBucket repo and importing M03), just click the source icon on the right-hand side. Click each file to view it. Click "Raw" to copy & paste it into your own code files.

The screenshot shows the Bitbucket web interface for a repository named 'Denise Case / M03'. The page title is 'Source'. The file being viewed is 'M03 / M03.css'. The interface includes a sidebar with icons for JS, a file explorer, and a commit history icon (highlighted with a red box). The main content area shows the source code for 'M03 / M03.css' with a commit hash of '34e3ced' from '2 days ago'. The code is as follows:

```
1 body {  
2   padding-top: 60px;  
3   padding-bottom: 10px;  
4 }
```

At the top right of the code area, there are buttons for 'Blame', 'Embed', 'Raw', and 'Edit'. A red arrow points to the 'Raw' button, indicating where to click to view the raw source code.

More about JavaScript objects

Objects: not what you think

JS object is a collection of **name:value** pairs

Widely-used data structure, similar to:

- Dictionaries in Python and Swift.
- Hashes in Perl and Ruby.
- Hash tables in C and C++.
- HashMaps in Java.
- Associative arrays in PHP

*To allow dot notation, use **names without spaces (always)**.*

```
var obj = new Object()
var obj = {}
var obj = {
  name: "Carrot",
  "for": "Max",
  details: {
    color: "orange",
    size: 12
  },
  gm:function(x,y){return x*x+y*y}}
obj.details.color // orange - chaining attributes
obj["details"]["size"] // 12
obj.gm(3,4) // 25
```

Fun fact! new is not required to make an object ... that's different from Java. Also note: an anonymous function has crept into the code again...

Techy Aside: The Great Debate -- Dot or Bracket Notation?

Either is fine most of the time. Bracket notation:

- takes 3 extra characters and is harder to read
- allows for spaces in member names (but we won't use those, so that's a moot argument)

To programmatically select a member, or to iterate through all members, requires bracket notation

```
var person = {"name":"Tammy", age:18, gender:"female",
              ableToVote:function(legalAge){return this.age >= legalAge}}
console.log(person.name)
console.log(person["name"])

if(person.ableToVote(18)) {
  console.log("Go, make a difference")
}

for (key in person) {
  console.log(key + ": " + person[key])
}
```

```
Tammy
Tammy
Go, make a difference
name: Tammy
age: 18
gender: female
ableToVote: function (legalAge){return this.age >= legalAge}
```

Object prototypes

```
function Person(name, age) { //object prototype for Person objects
    // it's a constructor function
    this.name = name
    this.age = age
}

// Define an instance of that object prototype with "new"

var you = new Person("You", 24)

// Create a new person named "You"

// (that was the first parameter, and the age..)
```

Aside: Fun with Prototypes

Every object has its own set of properties. It also has another property, a **prototype**, that acts as another source of properties. So when an object is asked for its property, if it doesn't have it, it looks in its prototype for it.

One way to create objects with the same prototype is to use an object constructor function.

```
function Person(name, age) { // object prototype for
                              // Person objects
    this.name = name
    this.age = age
}
```

```
var suman = new Person("Suman", 24) // both have the same prototype
var sarah = new Person("Sarah", 48)
```

```
console.log(suman.prototype === sarah.prototype) // true
```

Aside: Fun with Prototypes

- To add properties or methods to an existing object, assign it to that instance.
- To add properties or methods to a prototype (so that all instances of that object can access those properties/methods), use **prototype**.

```
var suman = new Person("Suman", 24)
```

```
var sarah = new Person("Sarah", 48)
```

```
suman.height = 180 // only assigned to suman
```

```
sarah.height // undefined
```

```
Person.prototype.nationality = " " // every person gets it
```

```
sarah.nationality // ' '
```

```
suman.nationality // ' '
```

```
suman.nationality = "India" // now Suman's nationality is India
```

```
sarah.nationality // still ' '
```

Array Objects

Array Objects

Much like regular JS objects - just with a magic property: **length**.

```
var a = new Array()
```

```
a[0] = "dog"
```

```
a[1] = "cat"
```

```
a[2] = "hen"
```

```
a.length // 3
```

```
var a = ["dog", "cat", "hen"]
```

```
a.length // 3
```

```
var a = ["dog", "cat", "hen"]
```

```
a[100] = "fox"
```

```
a.length // 101
```



Tech Notes: Fun Array Facts

- **.push()** // adds element at the end of the array
- **.pop()** // removes element from the end of the array
- **.concat()** // concatenates a second array and returns a new array
- **.join()** // munges the array elements into a String
- **.length** is assignable -- you can change the size of an array (set it to 0 to clear out the array set it to -1 to destroy the universe, but do so quickly, before Trump does :-))
- **.reverse(), .sort()** // does the obvious
- **.indexOf()** // finds the location of the argument
- **.forEach()** // executes a callback on each element of the array
// the callback should have a single parameter

Probably not on the exam, but the array is handy JS data structure.

T.A.: Fun Array Facts Demo

```
> var people = [] // now people is [ ]  
> people[0] = "Barry Allen" // now people is [ 'Barry Allen' ]  
> people[3] = "Iris West" // [ 'Barry Allen', , , 'Iris West' ] // note null entries  
> people.reverse() // [ 'Iris West', , , 'Barry Allen' ]  
> people.push("Joe West") // [ 'Iris West', , , 'Barry Allen', 'Joe West' ]  
> var topCop = people.pop() // [ 'Iris West', , , 'Barry Allen' ] , topCop is 'Joe West'  
> people.indexOf('Barry Allen') // returns 3  
> people.forEach(function(role){console.log(role)})  
// Output:  
Iris West  
Barry Allen
```


Scope: Major Difference

var is not limited to block scope (curly braces)

Any variable declared with **var** is available to the whole function (bad practice)

To limit scope, always use the new **let** and **const**.

```
const firstname = "Bhanu"    // no rebinding allowed
let age = 24                 // can be reassigned
```

```
const foo = {}
foo.bar = 42
console.log(foo.bar)  // → 42
```

```
const bar = 27
bar = 42 // throws TypeError exception
```

Techy Aside: Printing Objects

When printing an object by itself, `console.log()` works fine. But if you try to label that object you will encounter an issue: **JSON.stringify()** will come to your rescue.

```
var speedster = {"name":"Barry Allen", "age":25}
console.log(speedster) // output: { name: 'Barry Allen', age: 25 } -- perfect

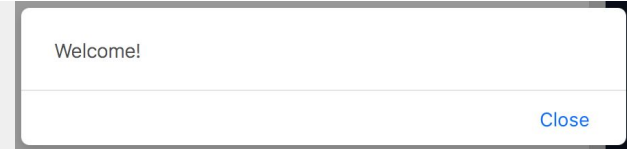
console.log("Fastest man alive: " + speedster)
// output: Fastest man alive: [object Object] // Ruh, roh!

console.log("Fastest man alive: " + JSON.stringify(speedster))
// output: Fastest man alive: {"name":"Barry Allen","age":25} // Much better!
```

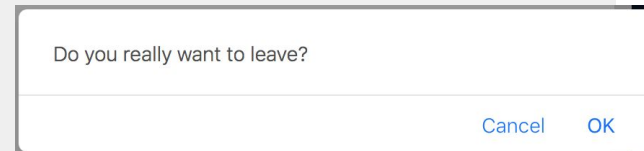
Important JavaScript objects

JS Window object: alert / confirm / prompt

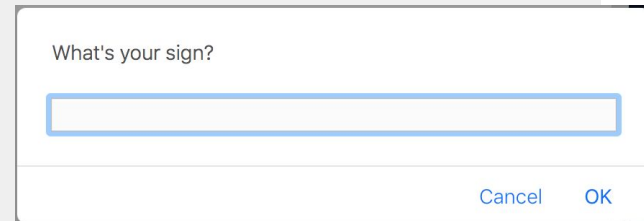
```
window.alert("Welcome!")
```



```
if (window.confirm("Do you really want to leave?")) {  
    window.open("exit.html", "Thanks for Visiting!")  
}
```



```
var sign = prompt("What's your sign?") //window is optional  
if (sign.toLowerCase() == "scorpio") {  
    alert("Wow! I'm a Scorpio too!")  
}
```



These are very basic and useful user interactions... they are worth memorizing. :)

JS Navigator (browser)

The window object has a **navigator** object that refers to the browser.

window.navigator.appName	// e.g., Netscape
window.navigator.appCodeName	// e.g., Mozilla
window.navigator.appVersion	// lots...
window.navigator.platform	// e.g., Win32

Try it: Example of how to use JavaScript to change a web page:

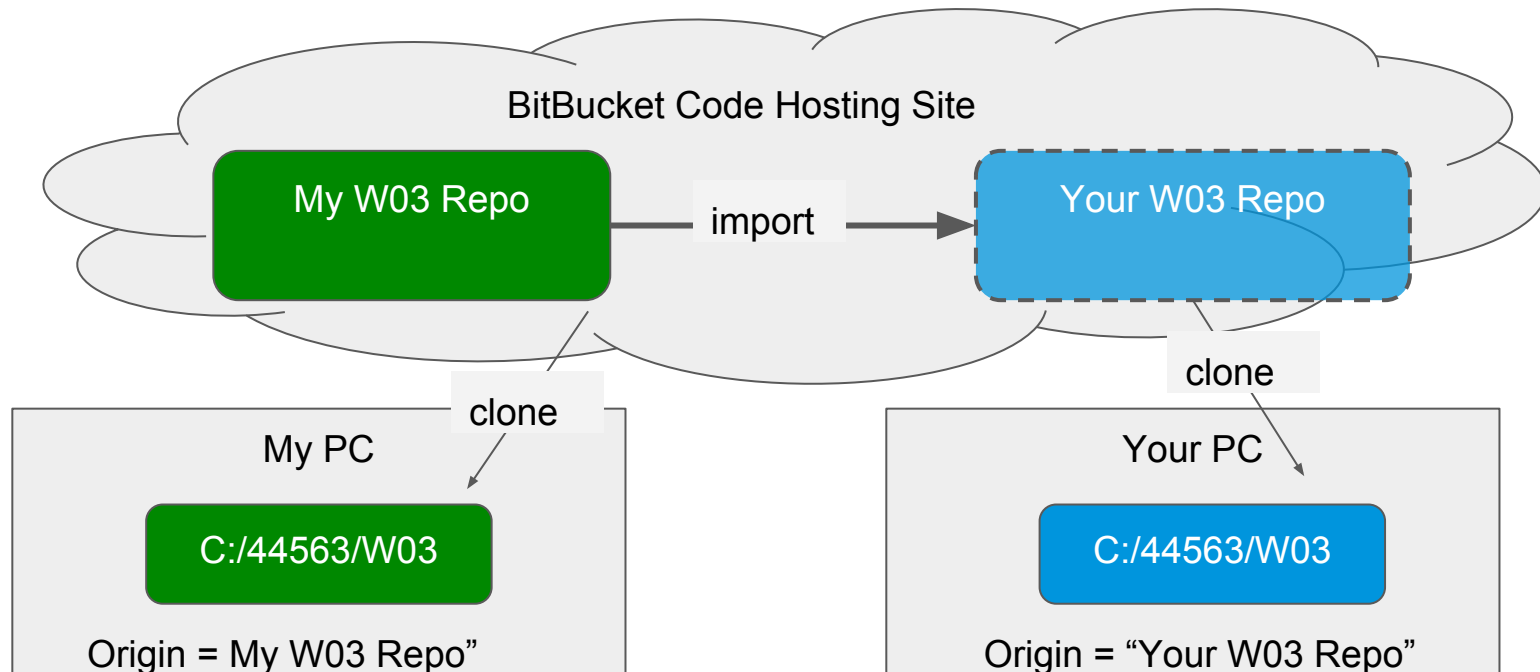
http://www.w3schools.com/js/tryit.asp?filename=tryjs_nav_cookieenabled

W03 Coding Practice

First, we'll talk about it. After this introduction, we'll get in teams. One member will go to their **BitBucket** account. Click **Repositories / Import repository**.

URL = <https://bitbucket.org/professorcase/w03>

Enter a name for your repo (e.g. w03). Verify **private** is checked and **import**.



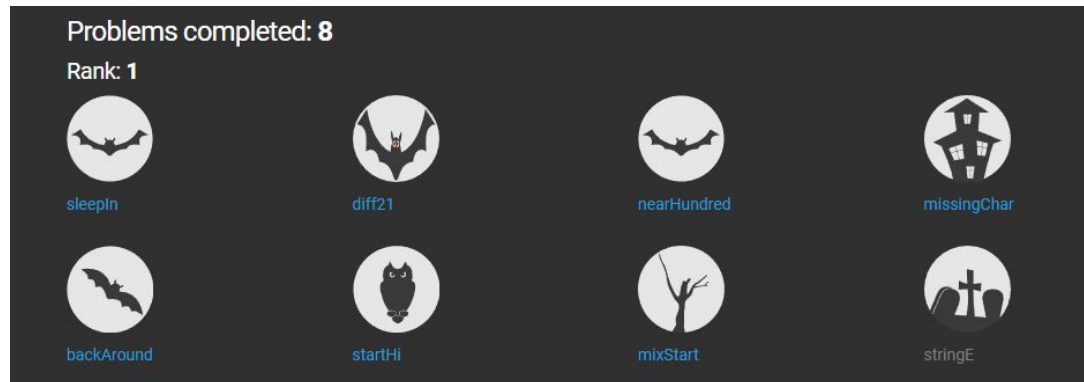
- ❑ M03 - Starting JavaScript
- ❑ W03 - Quizzes & App version
- ❑ A01 - Your Site: Responsive
- ❑ Q03 - Everything up to now
- ❑ A02 - Your Site: Event-driven



NodingBat

Go to <http://nodingbat.com/>

Test drive your JavaScript coding superpowers by completing several coding exercises.



18 puzzles; 18 points

See the course site for recommended schedule to complete the puzzles.

References and resources

- [Mozilla Tutorial](#) is a nice friendly introduction to OOJS (Object Oriented JavaScript).
- [JSBin](#) and [Thimble](#) are online JS interpreters
- A [visualization of prototypes](#)
- A [discussion of inheritance](#) in JS
- For students just starting programming, this could be a fun way to practice:
<https://codehs.gitbooks.io/introcs/content/Programming-with-Karel/index.html>