Back-End JavaScript Literacy

by Chris Minnick

Introduction

Objectives

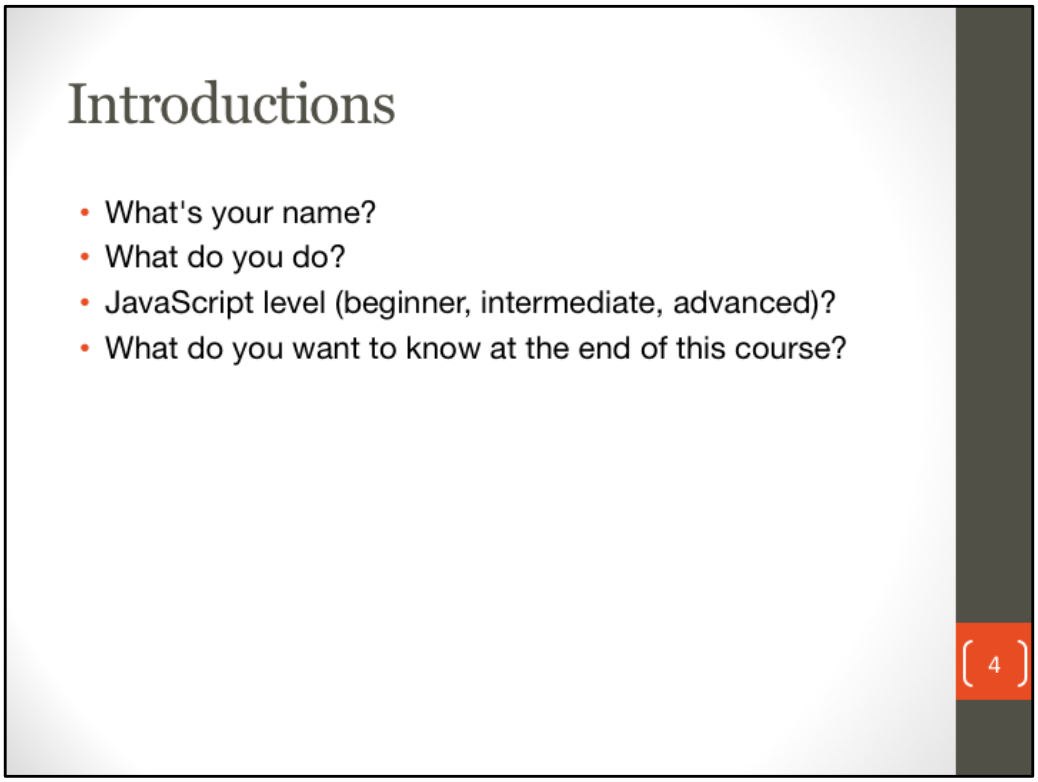
- Who am I?
- Who are you?
- Daily Schedule
- Course Schedule and Syllabus

2

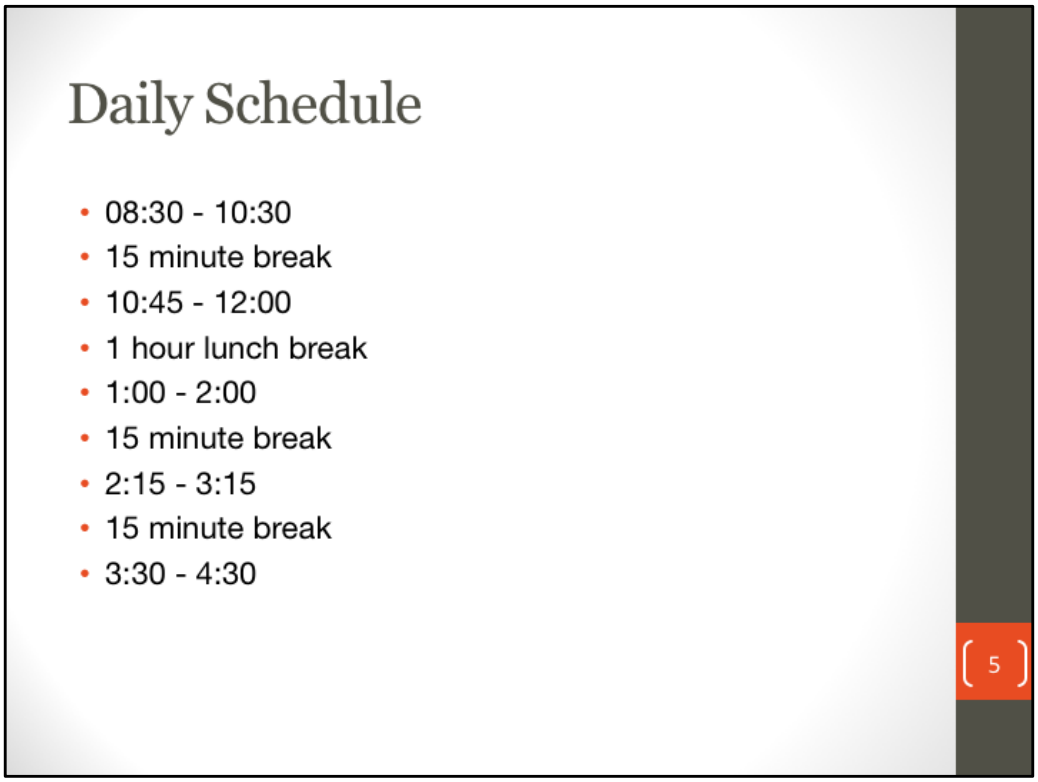
Chris Minnick is a prolific author, blogger, trainer, speaker, and web developer. His company, WatzThis? is dedicated to finding better ways to teach computer and programming skills to beginners.

Minnick has authored and co-authored over a dozen technical books for adults and kids; including XHTML, Coding with JavaScript for Dummies, JavaScript for Kids, Adventures in Coding, and Writing Computer Code.

When he's not writing about technical topics, Chris is a winemaker, a novelist, a swimmer, a cook, and a musician.



Please answer these questions!

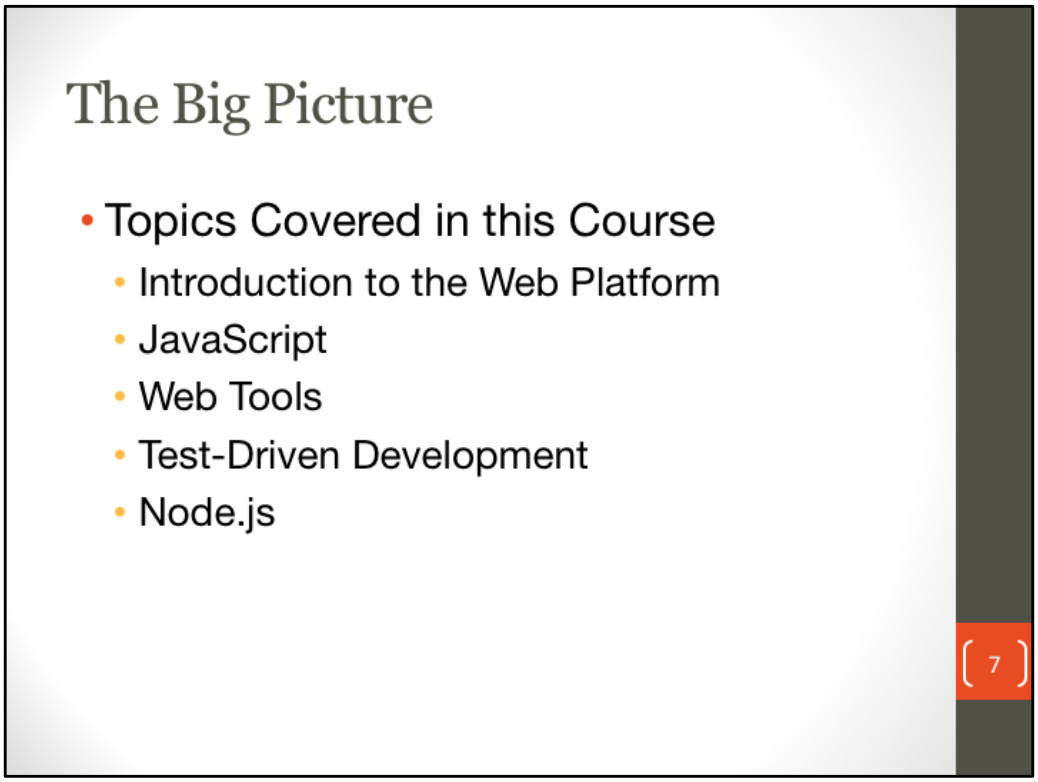


This is just a rough estimate. We may take breaks at different points, depending on where we are with the material. But, in general, we'll take a break in the morning, a break for lunch, and a couple breaks in the afternoon.

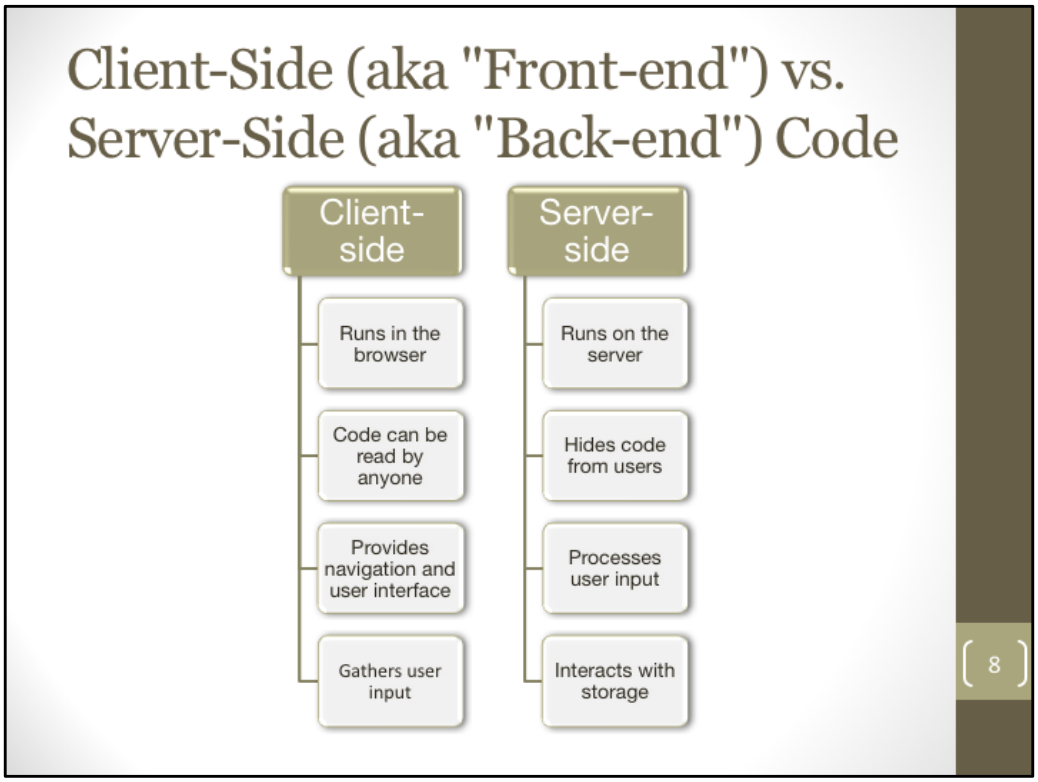
Module: Intro to the Web Platform

Objectives

- Understand Unix
- Know the Internet Protocols
- Configure a Local Server
- Learn How Browsers Work



The primary goal of this course is to give you a broad understanding of how back-end web development works. Towards that goal, we're going to cover a lot of different topics, but we won't be going in-depth into any of them. If you want to become a back-end web developer, this course will show you the main things that you'll need to learn focus on learning and practicing. If you work with programmers, this course will give you an improved ability to communicate and collaborate with them on a technical level.



Client-side code runs in your web browser. It gets downloaded from the server along with web pages. Client side code is what handles navigation, input validation, animations, and interaction with the browser and the hardware.

Server side code is what your browser interacts with when you request a web page by typing a URL into the address bar, seaching, or clicking on a link. Server side code receives data from web browsers, does something with it, and returns it to the web browser for use by client side code and for display to the user.

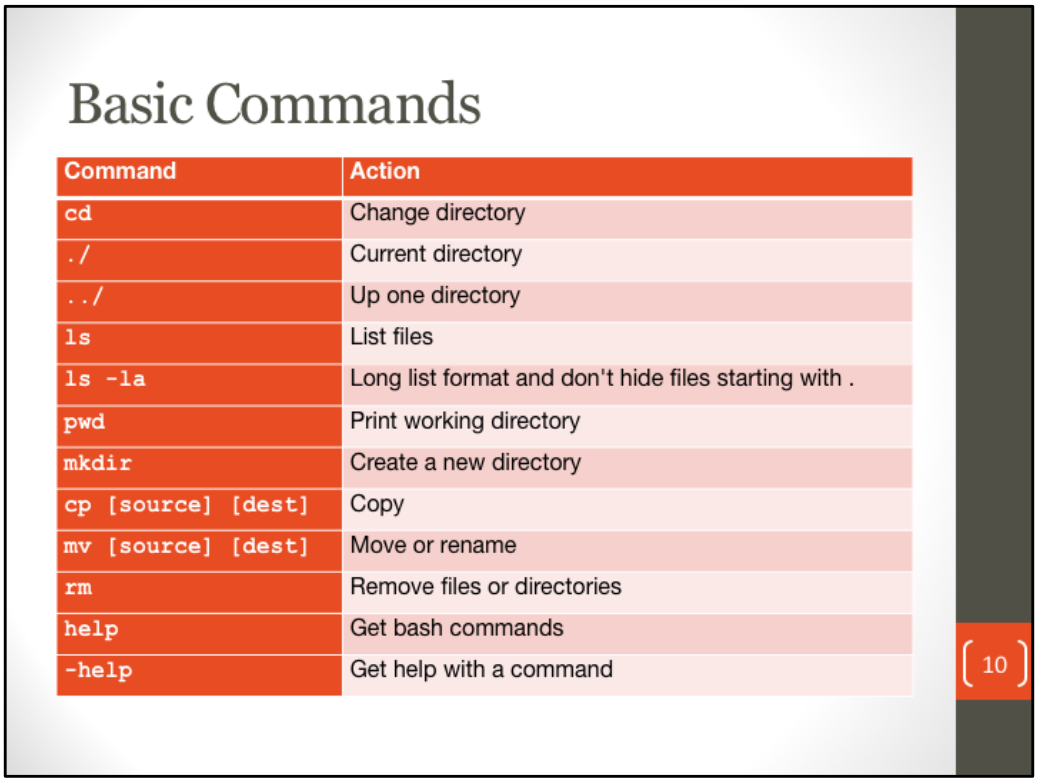
Server-side code can interact with databases, can do user authentication, and many other things that can't securely or practically be done on the client. If it involves a large amount of data, security concerns, or sharing of data between multiple users, it belongs on the server.

Exchanging data with the server is much slower than processing data on the client side; so it's usually a good idea to do all the processing of data that you can do on the client side and only interact with the server when necessary.

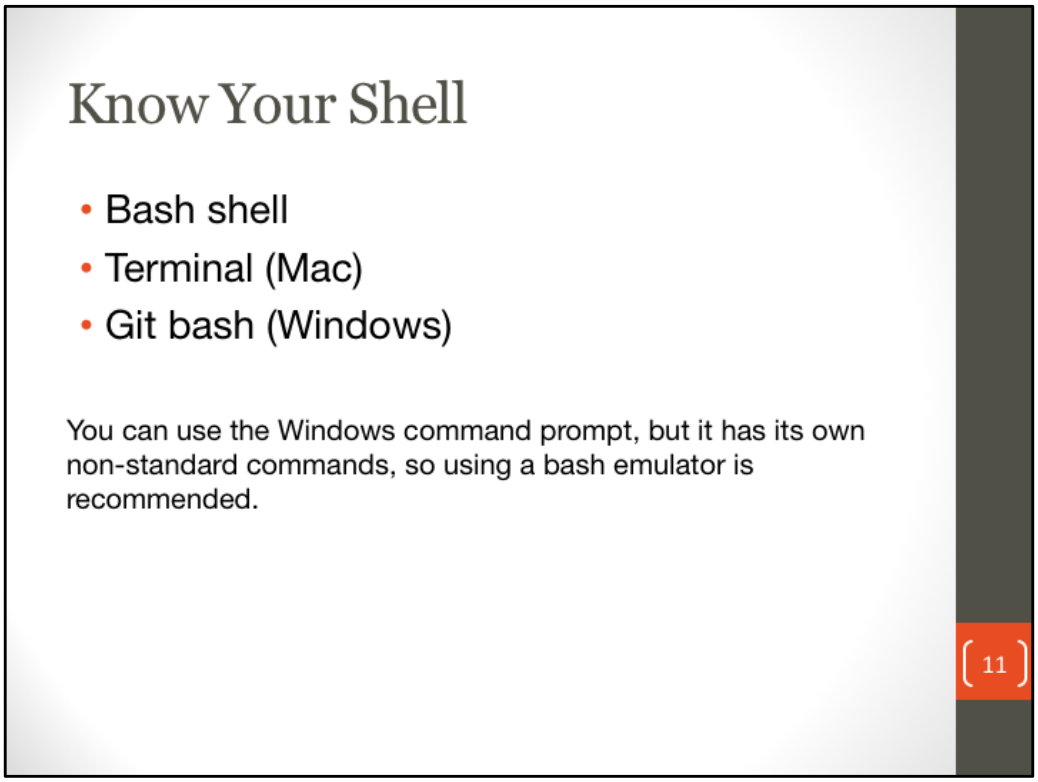
Objectives

- Use a Unix-style command prompt
- Become familiar with basic commands

The command prompt is absolutely required for front-end development, so it's important that you become comfortable with it



These are the most basic unix commands that everyone doing work on the web should know.



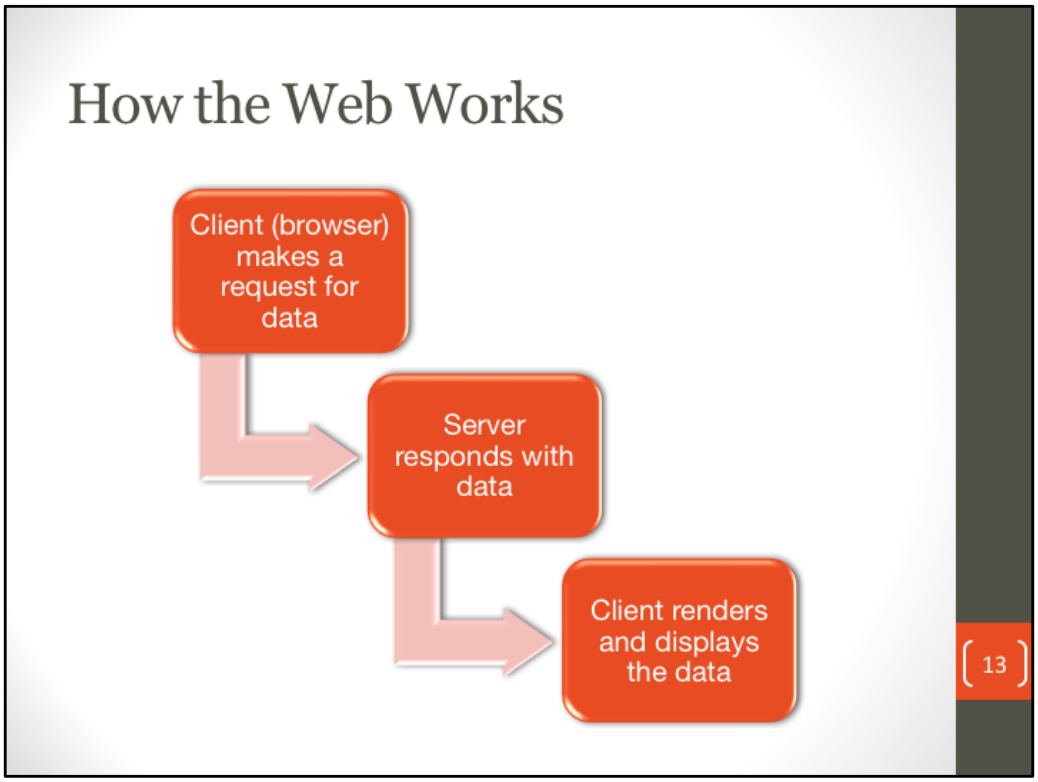
The command prompt for MacOS is called Terminal. It's located inside Applications > Utilities

On Windows, use git bash so that the Unix commands in this course will work correctly.

With 64-bit Windows 10 (build 1607+), you can install the windows subsystem for linux to get access to a real Linux environment inside windows. Follow the steps here to do this: https://msdn.microsoft.com/en-us/commandline/wsl/install_guide

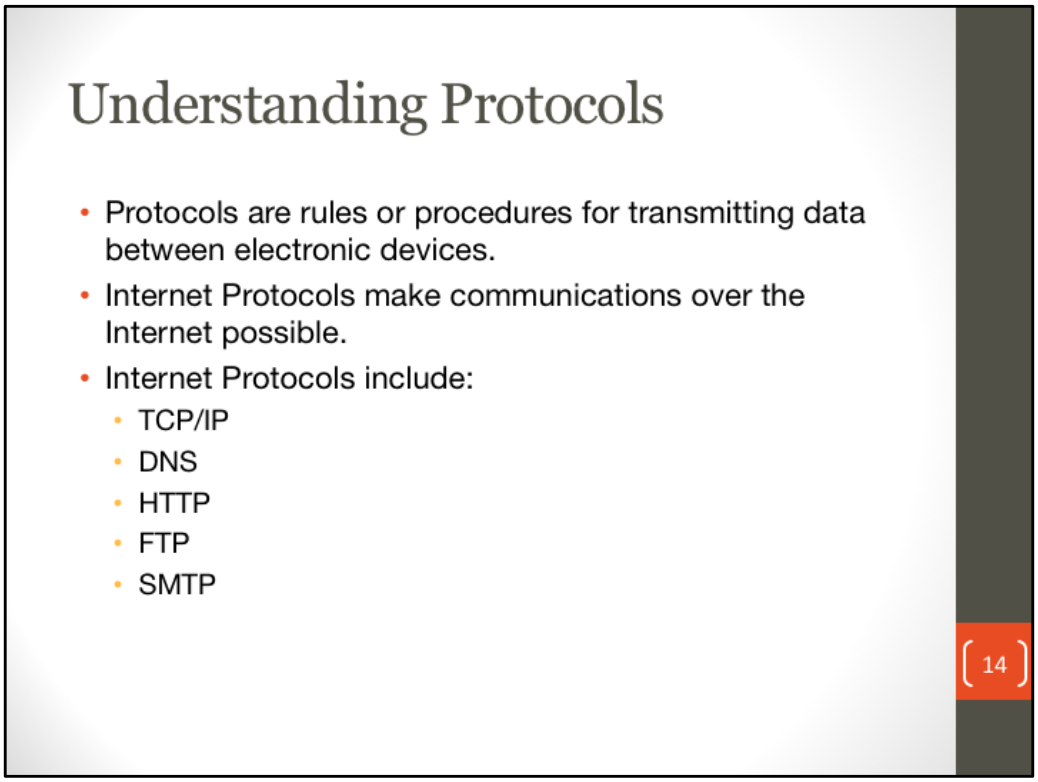
Lab 01: Working with the Unix/Linux Command Shell

- In this lab, you'll learn the most commonly used Unix commands and how to use them to find your way around a command line interface.
  - Command line
  - Basic commands
  - Files and directories

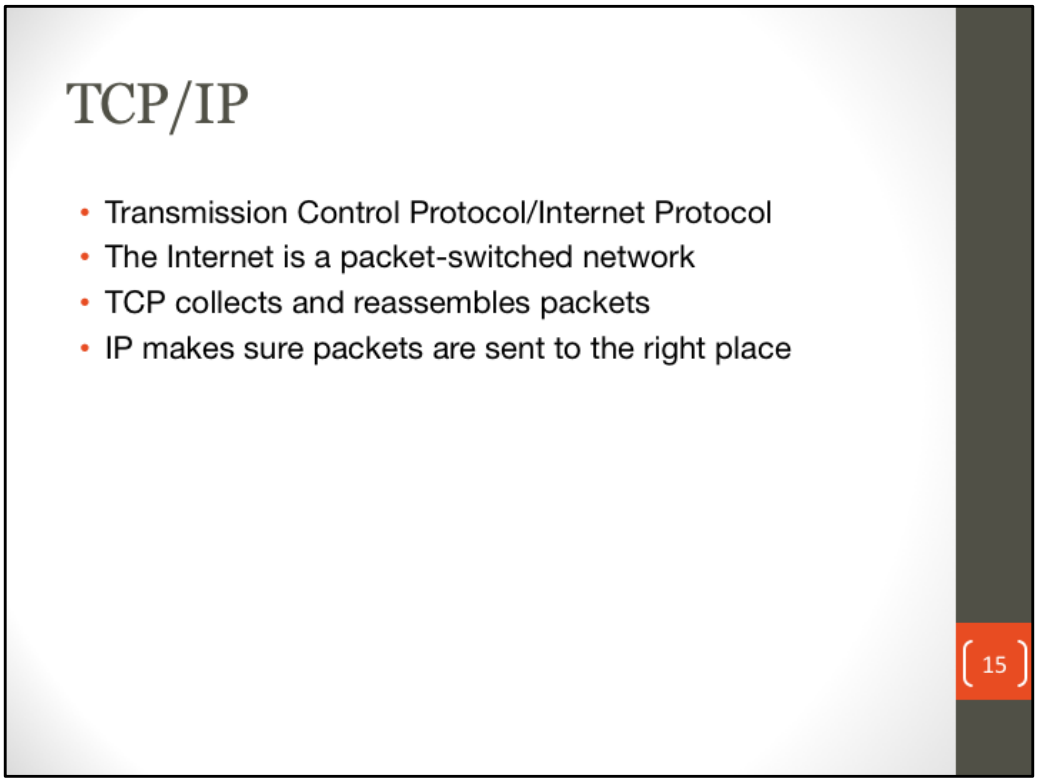


Every computer connected to the internet has an "address". When you put a URL into your browser's address bar, or click a link, this causes your browser to send a "REQUEST" to another computer on the internet somewhere. The request tells the computer (which is called a web server if it's job is to listen for requests from web browsers) to send a particular file to the web browser. The web browser then interprets and displays that file to you.

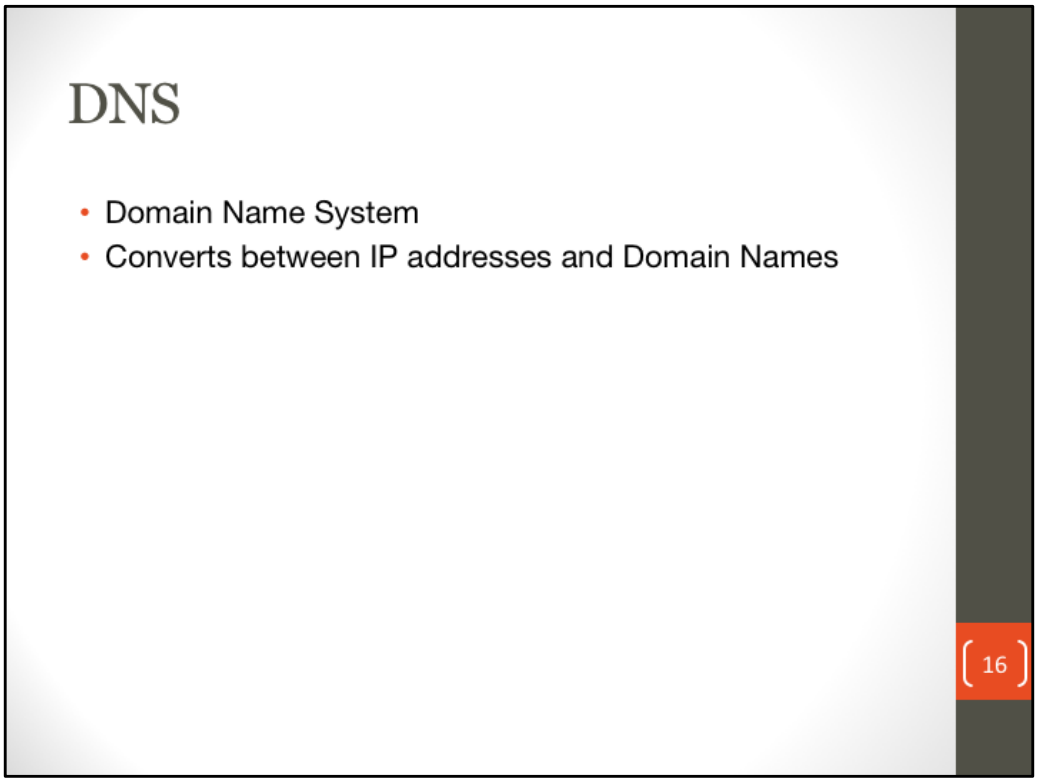
You'll see an example of this in Lab 2: Configuring an HTTP (Web) Server.



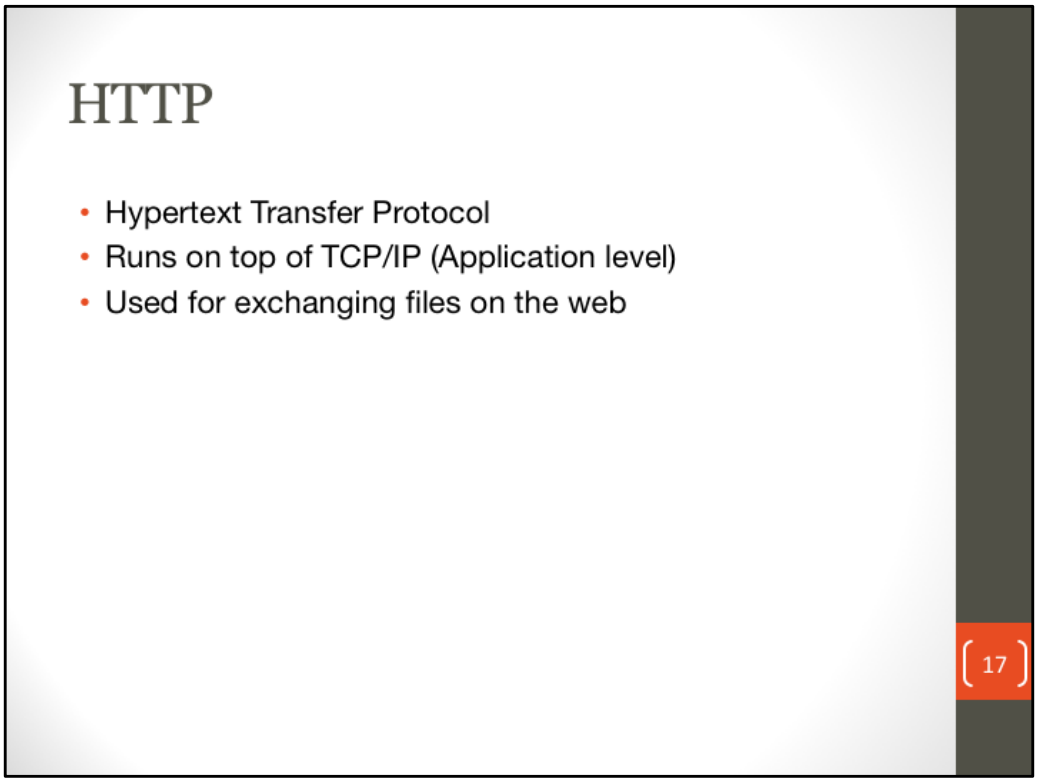
To share data, computers have to agree on how the data will be sent and received. This is the job of protocols. Protocols are created and "standardized" by a standards organization. Examples of standards organizations include ANSI, ECMA, W3C, and the IAB.



TCP/IP is the protocol that defines how data is routed throught the internet. It's used for all internet trafic. Other standards are "applications" that run on top of TCP/IP and provide additional functionality over the internet. This is similar to how your computer has an operating system (Windows or MacOS) and applications run on top of the operating system (Web browsers, photoshop, MS Word, etc).



DNS associates people friendly names, such as www.example.com, with IP addresses, which are more difficult to remember. Because of DNS, you can type www.google.com into your browser rather than having to type a long string of numbers.



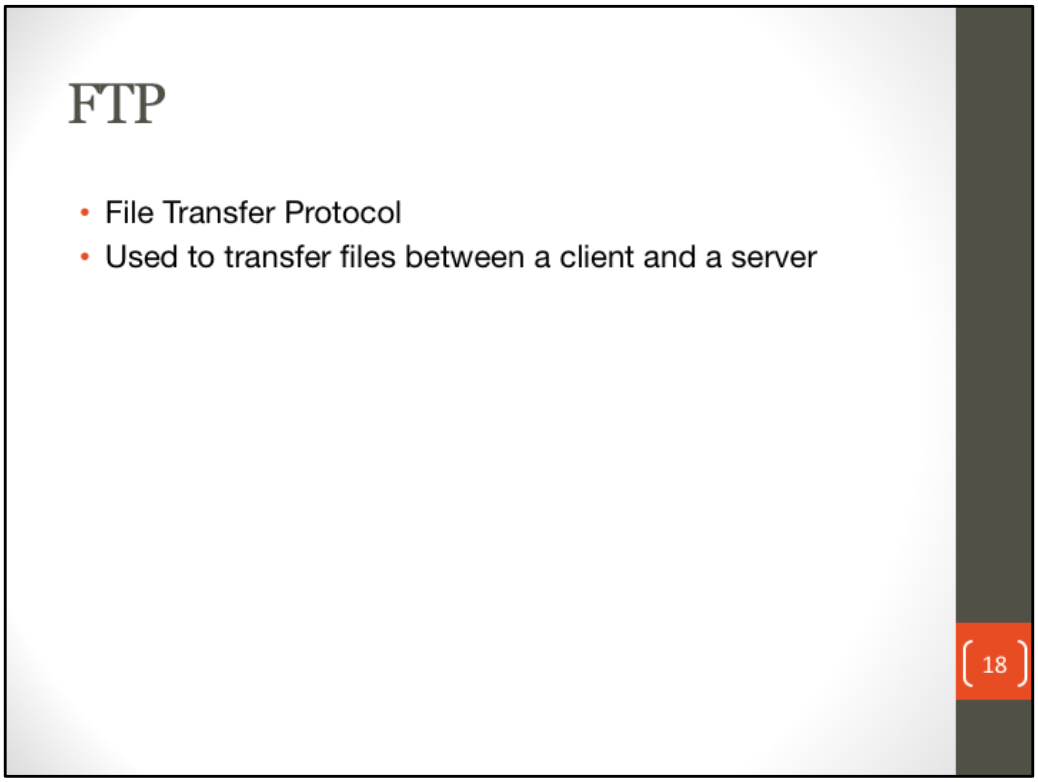
HTTP is the protocol for transmitting data over the world wide web. Development of HTTP was started by Tim Berners Lee at CERN in 1989. It's a request-response protocol. A client makes a request for a "resource" and a server responds to the request.

When a client makes a request, it specifies a Request Method (also known as a "verb") describing the action to be performed on the resource.

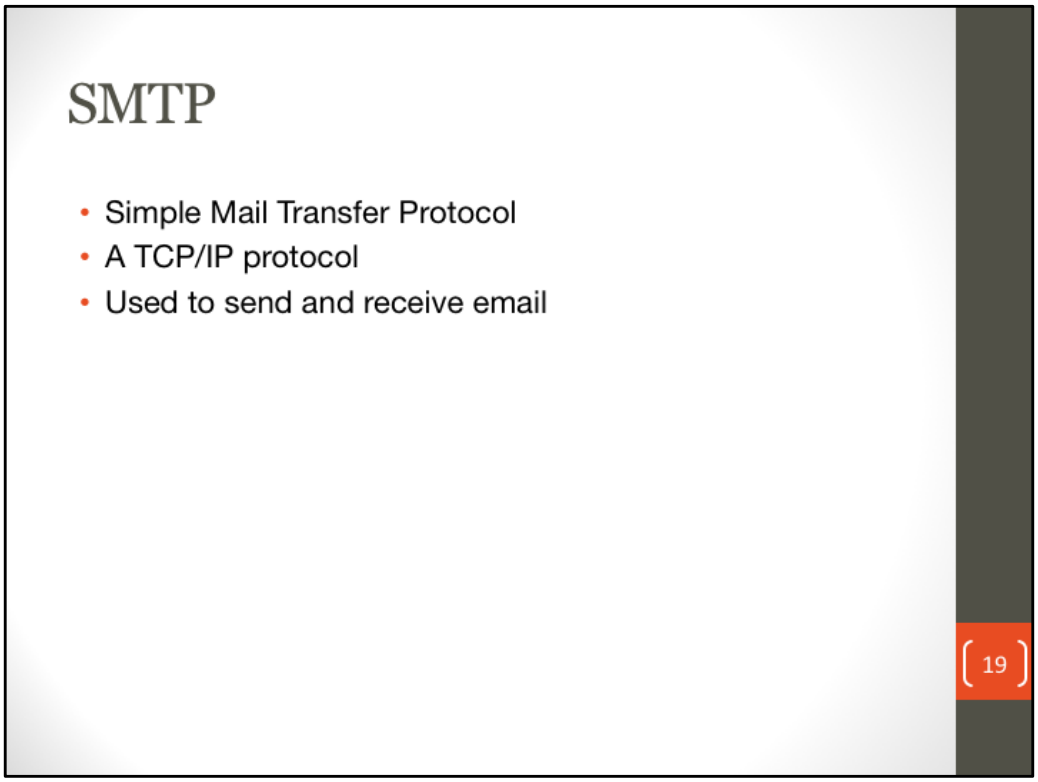
HTTP 1.1 defines 8 methods: GET, POST, HEAD, OPTIONS, PUT, DELETE, TRACE, CONNECT, and PATCH.

GET and POST are the methods that a web browser typically uses as you browse the web. GET retrieves data (for example when you access a web page). POST is generally used to create something new on the server. For example you might post a new message to a forum or a comment thread, or post form data.

PUT and DELETE are important mostly for web applications – programs running on the web (as opposed to static web pages). PUT is used to modify a resource. DELETE deletes the specified resource.



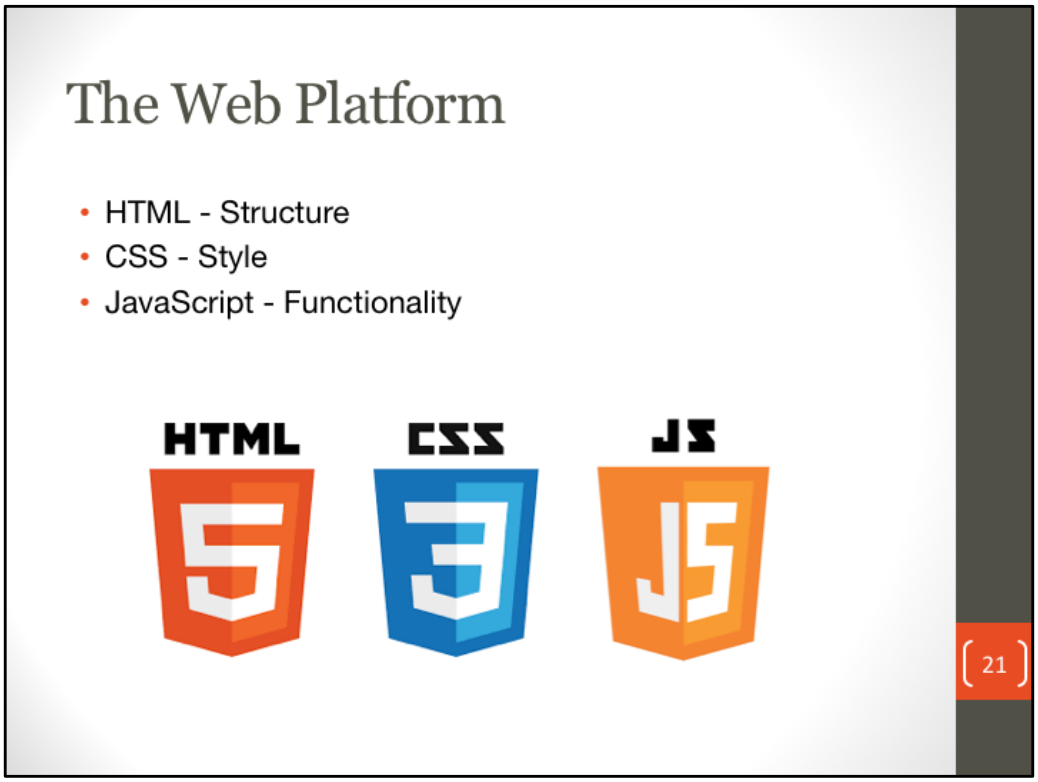
FTP, like HTTP, is a client-server protocol. It's used for transmitting files between a client (your computer) and a server (a remote computer). FTP is one way of uploading files to a remote machine to be served on the web.



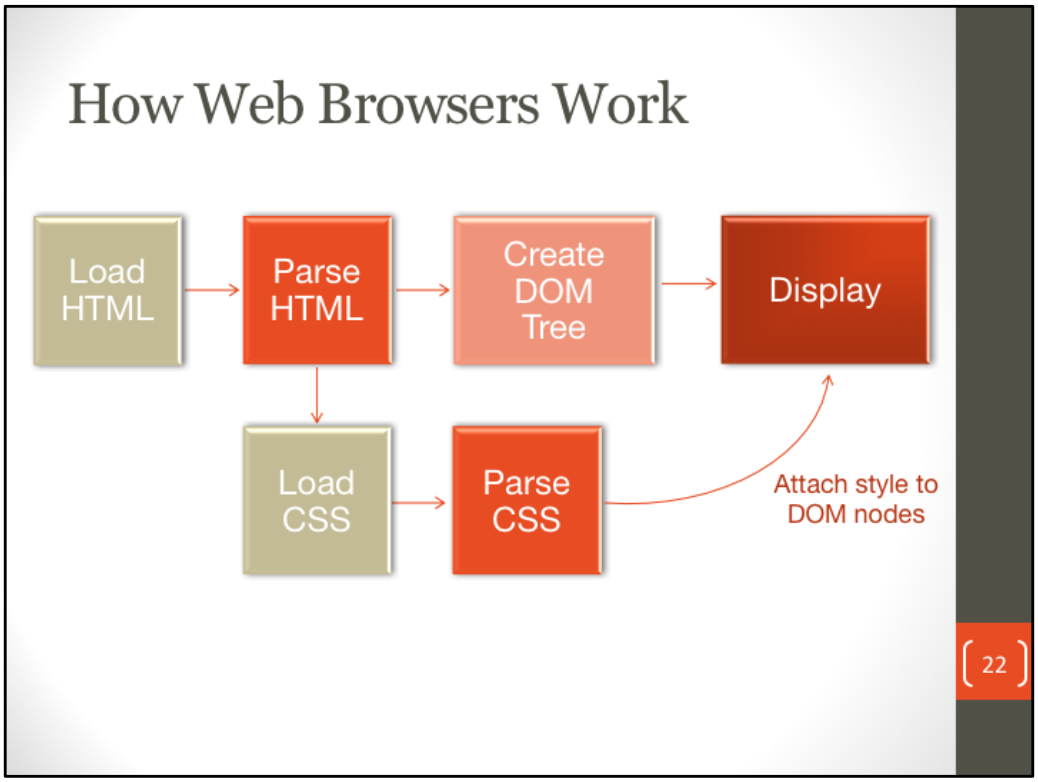
Mail servers use SMTP for sending and receiving messages. User (client) mail programs typically use it just for sending mail.

Lab 02: Configuring an HTTP (Web) Server

- In this lab, you'll configure and test a web server on your local development machine, and then view the data in a web browser. This will help you understand this process a little more clearly.



When the job of back-end code is done, the result gets sent to a web browser (client-side) by the web server. On the client-side, we often talk about the combination of languages that make up every web page as "The Web Platform". HTML5 provides the structure for web pages, CSS3 provides the style and layout, and JavaScript handles the functionality.



Once a web page has gotten from the server to your computer (using TCP/IP, DNS, and HTTP) there's still a lot to be done. The web page has to be parsed, the style information has to be loaded and parsed, any linked images and multimedia files have to be loaded, JavaScript code has to be loaded and parsed, and a complex layout process and then a "paint" process has to be completed. All of this has to happen extremely quickly in order to give you the impression of a fast web browser.

Browsers include developer tools to give web developers and programmers a view into what the browser is doing. With the latest developer tools, you can get a lot of fine-grained detail about every aspect of the browser loading and displaying process. Learning to use these tools will greatly increase your understanding of web applications and your productivity.

Lab 03: Working with Chrome Developer Tools–Element Tab

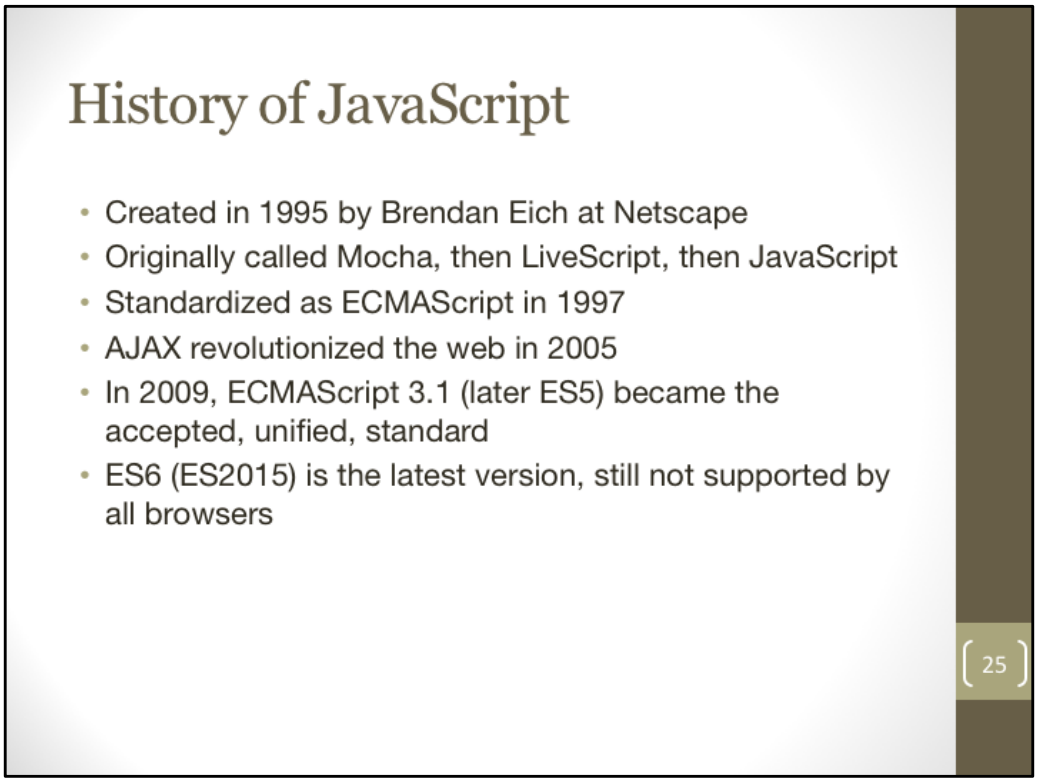
- In this lab, you'll explore the Elements pane and see how changes to the HTML affect the display of the web page in two different circumstances: 1) a page in your control and 2) a page that belongs to someone else.

Module: JavaScript

A.k.a. ECMAScript

Objectives:

- Understand the history of JavaScript
- Learn how JavaScript works



JavaScript has been around now for over 20 years. Brendan Eich created the first version of JavaScript, which was originally known as Mocha, in 1995 while at Netscape. It was intended to be a scripting language for the browser that would add interactivity to Netscape's browser and give them an edge in the brower war with Microsoft.

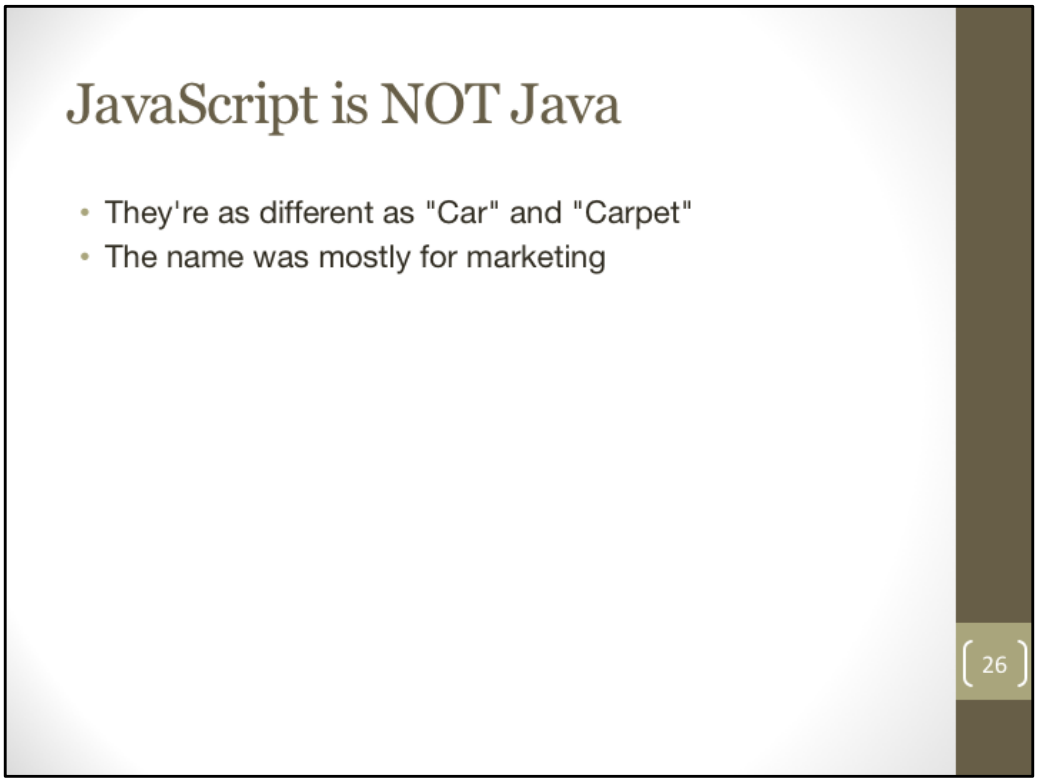
Mocha was renamed to LiveScript and then to JavaScript in its early days.

Netscape submitted JavaScript to ECMA for standardization in 1997 and the standardized version of the language was named ECMAScript -- another universally disliked name -- Eich commented that it sounded like a skin disease -- but nevertheless, it is the correct name for what is popularly known as JavaScript.

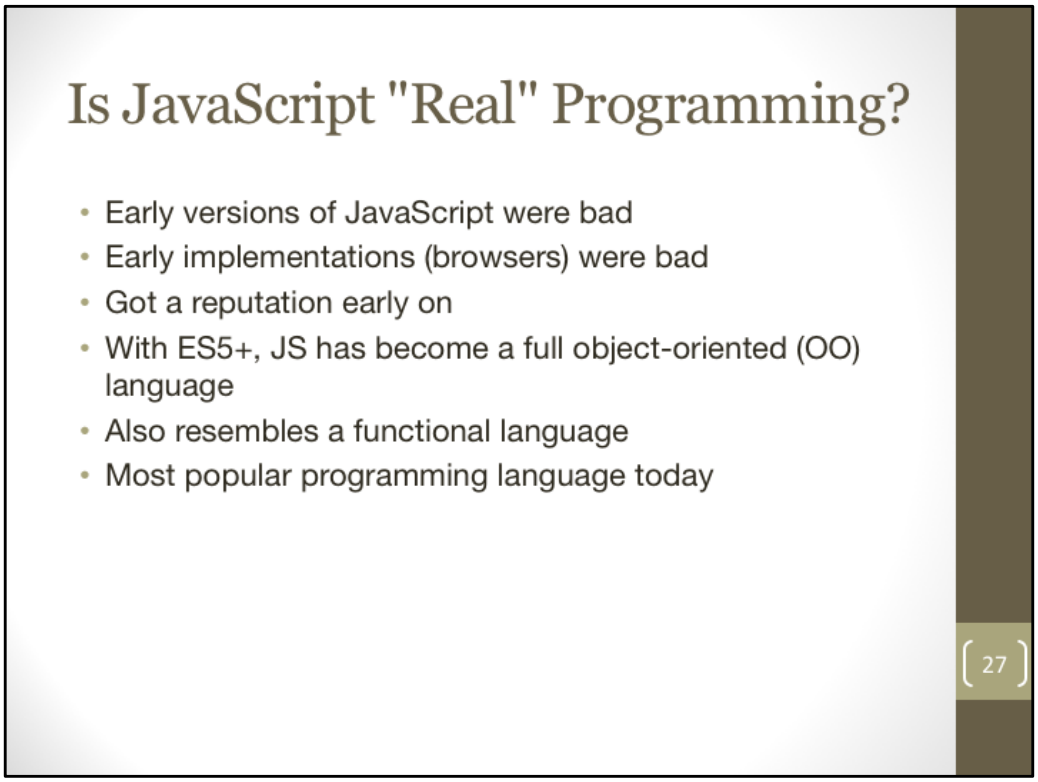
Its popularity is largely due to a revolution in how it was used starting in 2005, and the standardization browsers on of a new and unified version of JavaScript.

In 2015, ES6 (also known as ECMAScript 2015) was finalized. It adds many new features to the language and improves the syntax in many cases. However, it's not yet fully supported by browsers.

In 2016, ES7 (also known as ECMAScript 2016) was finalized.



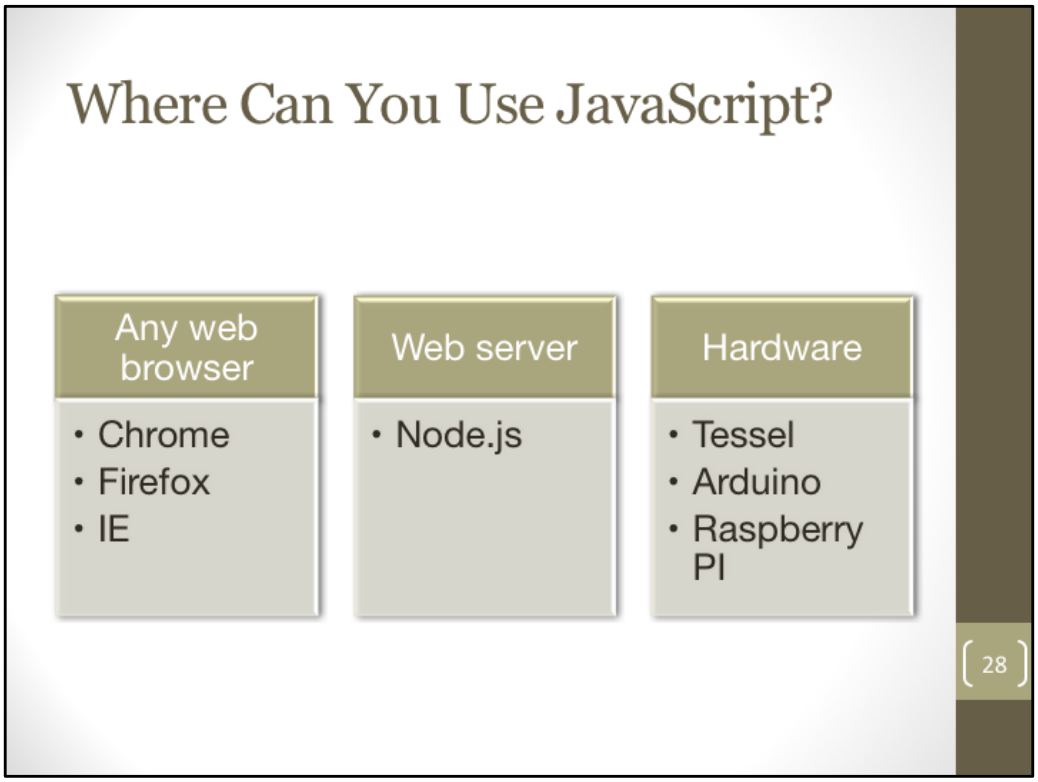
When JavaScript was created, Java was a new and popular language. It was thought that programmers would use JavaScript and Java applets together in the browser, and so the new browser scripting language (formerly known as Mocha and LiveScript) was named JavaScript for marketing reasons.



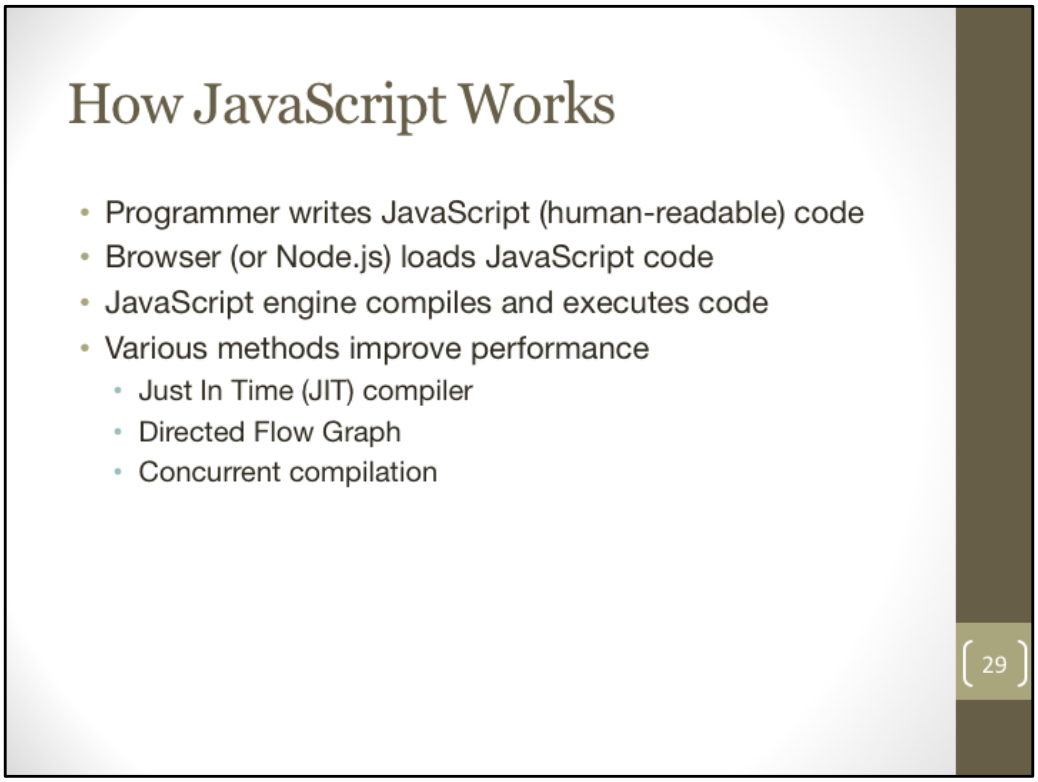
JavaScript is a mature language that has been around for over 25 years. It got a bad reputation in the early days because it was limited in what it could do, the language had (and still has) some odd quirks, and because browser implementations of the language were bad.

However, JavaScript has gone through a lot of change, and browser makers finally agreed on a standard in 2009, which is great news for those of us who code with the language.

Today, JavaScript is the most popular and widely-used programming language. Millions of people program with it, and every web browser features good support for it.



JavaScript runs everywhere. It's most commonly found running in web browsers. This is known as client-side programming. But, it's also a popular server-side programming language. Node.js, which is based on the same technology as Google's Chrome browser, allows you to create fast and powerful applications that can access databases, manipulate files, and much more. JavaScript is also a popular language for programming hardware devices.



Technically, JavaScript is a scripting language, which is a subset of programming languages. The differences between a programming language such as C or Java and JavaScript are narrowing and often debated. But, one key difference is in what form code written in the language is deployed.

In compiled programming languages, like C or Java, the code that people write is compiled into machine code prior to deployment.

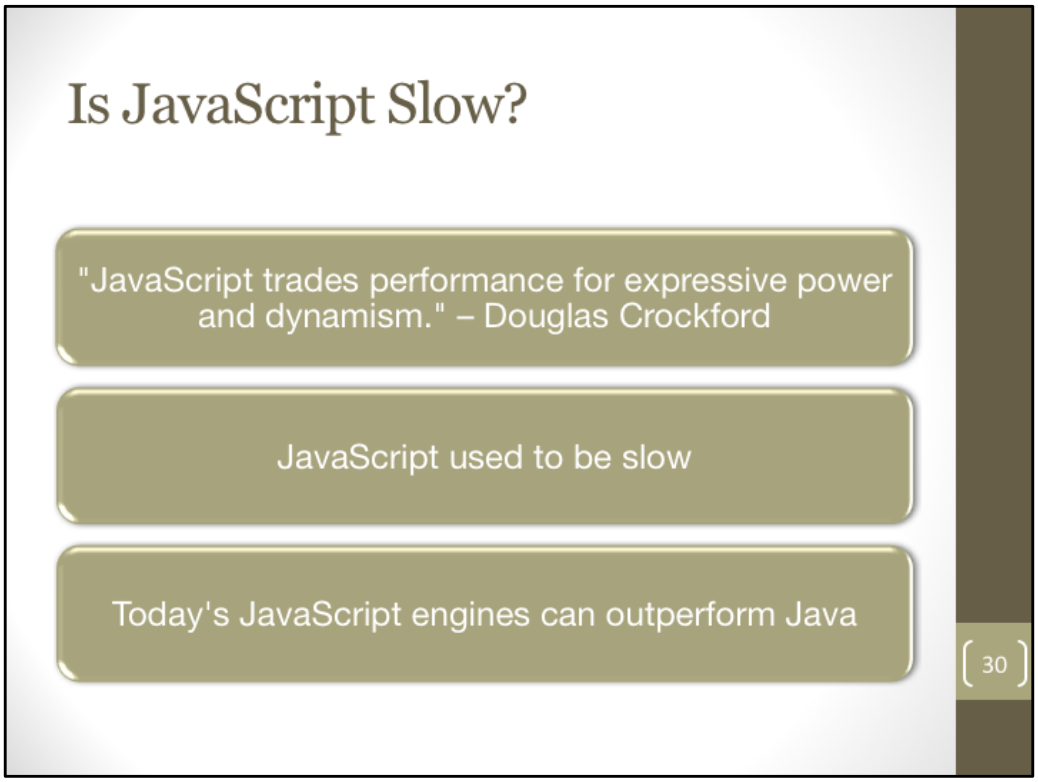
In a scripting language, such as JavaScript, perl, php, and others, programmers the code that the programmer writes is downloaded from a server and is only compiled into machine language immediately prior to running the instructions.

Another difference between scripting languages and compiled languages is that scripting languages require some sort of interpretter or other program in order to run, while compiled languages run directly inside of an operating system since they are compiled to the same language the computer understands.

This makes scripting languages easier to write with and test quickly. It also make code written in scripting languages portable. The same code written in JavaScript on a Macintosh computer can be run on Windows or Unix as long as a compliant JavaScript interpretter is installed.

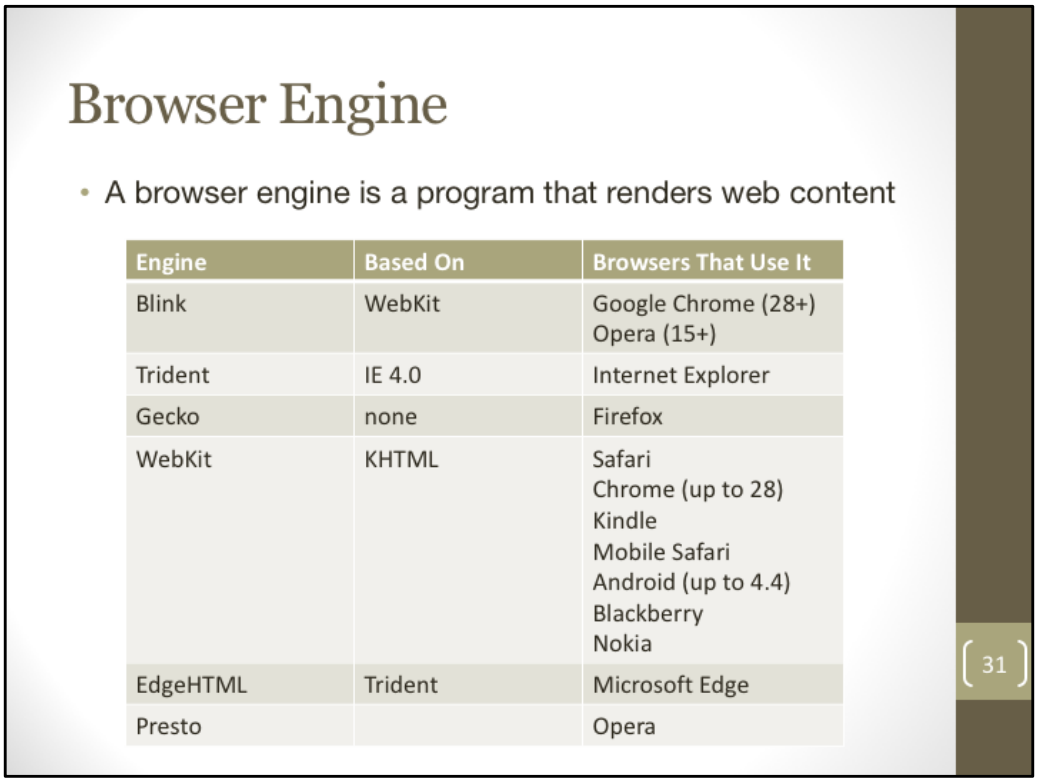
Every web browser contains a JavaScript interpretter, which is why JavaScript has been so widely adopted and successful.

The downside of scripting languages has traditionally been that the extra layer between the language and the hardware makes scripting languages slower than compiled languages.



https://strongloop.com/strongblog/node-js-is-faster-than-java/
https://rclayton.silvrback.com/speaking-intelligently-about-java-vs-node-performance

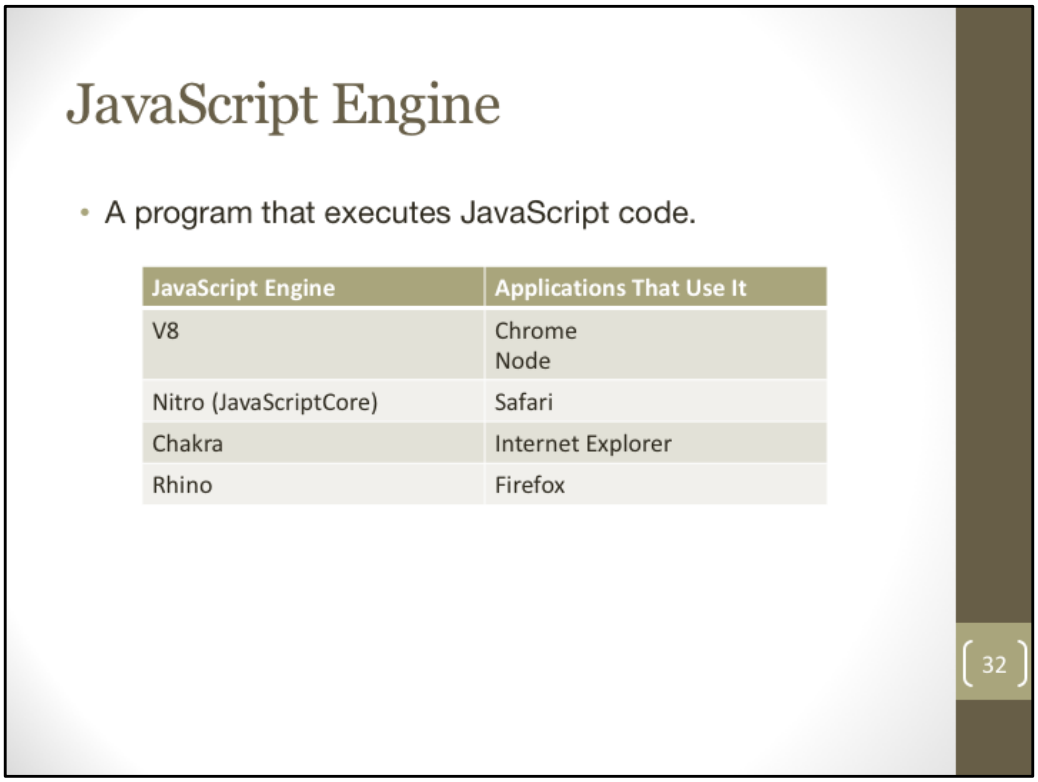
JavaScript, as Doug Crockford said, trades performance for expressive power and dynamism. However, this is not to say that JavaScript is slow. The truth is, today's hardware, combined with giant leaps in performance of JavaScript runners, makes JavaScript fast. In fact, some tests have found that JavaScript today can perform on par with compiled languages like Java.



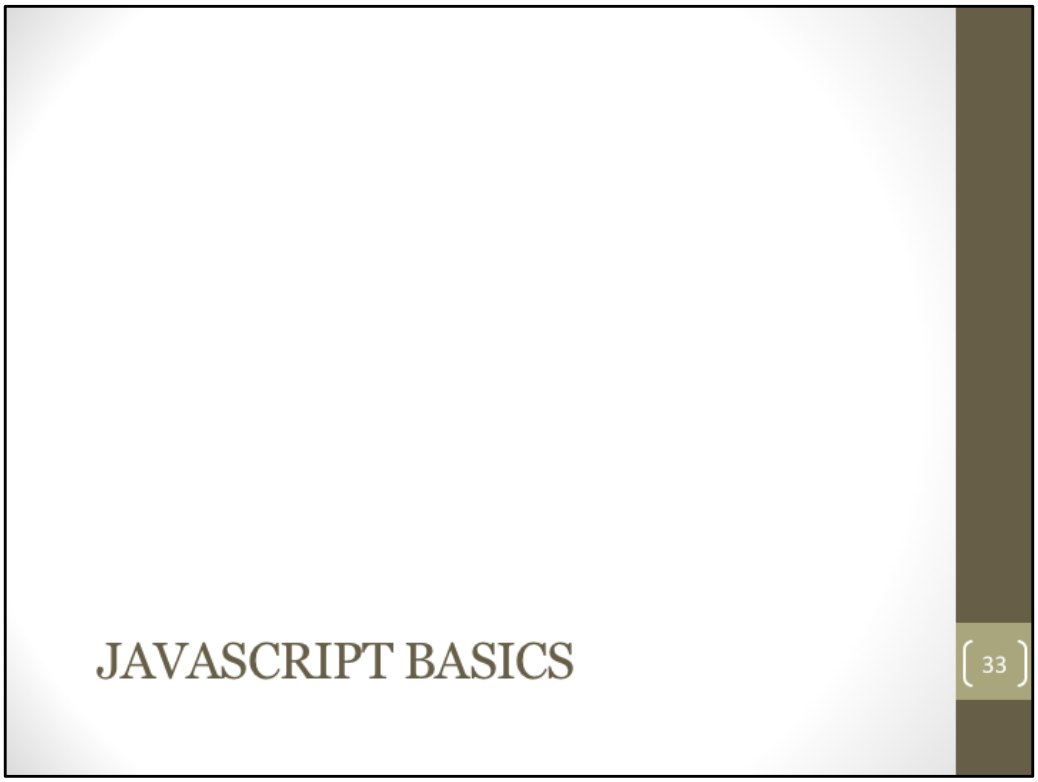
A browser engine renders web content. Web content includes HTML, CSS, JavaScript, XML, images, audio, video, and more.

Browser engines can roughly be divided into three major types:

1. webkit browsers. This includes the original webkit browser, Apple Safari, but also many other browsers that run the same engine or an engine based on webkit, such as Chrome.
2. Mozilla browsers. Gecko, which has its roots in the original Netscape browser, is the open source browser engine maintained and updated by Mozilla and used in the Firefox browser.
3. Microsoft browers. Until recently, all Microsoft browsers used the Trident engine. Today, Microsoft's edge browser uses the EdgeHTML engine.



A JavaScript engine is a program that can run JavaScript code. JavaScript engines are most commonly found as components within web browsers. Node.js uses the V8 engine to make server-side javascript possible.



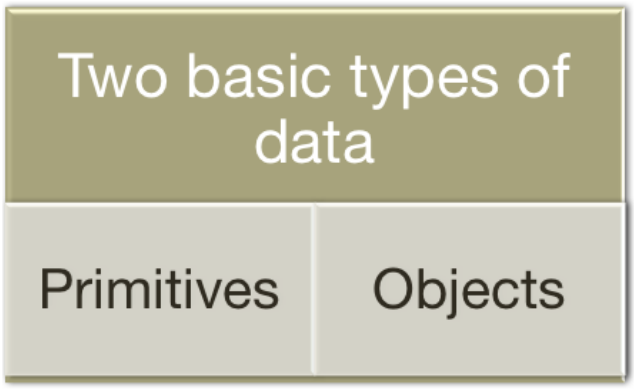
Objectives:

- Recognize JavaScript data types
- Use Objects
- Understand inheritance
- Understand the Document Object Model (DOM)

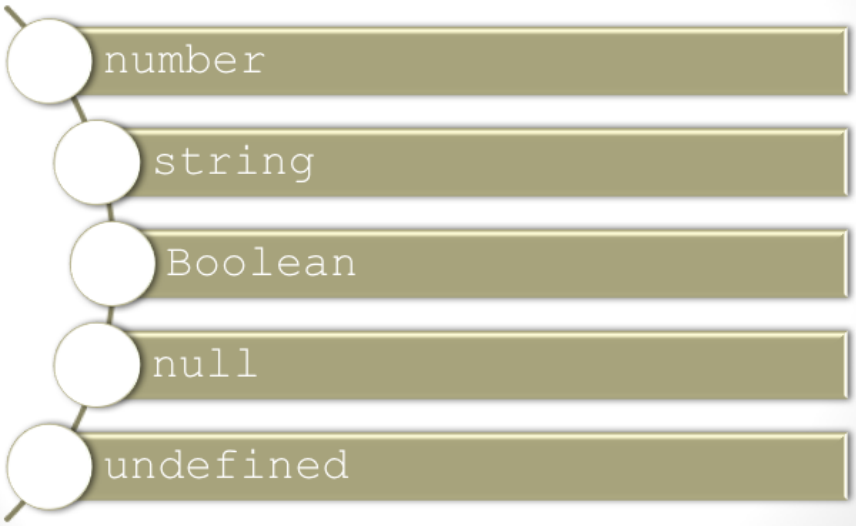
JavaScript Syntax

- Case-sensitive
- An *expression* produces a value
  - `1+1`
- A *statement* performs an action
  - `console.log(1+1);`
- Statements are separated by semicolons
- Strings can be enclosed by either single or double quotes.
- Comments can be block or line
  - `// line comment`
  - `/* block comment */`

JavaScript Data Types



JavaScript Primitives



JS Bin is an online tool for experimenting with JavaScript and web applications. In this lab, you'll learn how to use it to rapidly test JavaScript features and functions using JS Bin.

Variables and Arrays

- Variables hold values
- Arrays are variables that can hold multiple values
- Any type of data can be stored in a variable or array
- Variable names must begin with a letter, underscore, or $
- Variable names are case-sensitive
- Certain reserved words can't be used as variable name

Creating and Using Variables

- To create a variable
  - `var myVariable; // creates an empty variable`
  - `var myVariableName = value;`
  - `var myOtherVariable = "Hello!";`
  - `var anotherVariable = 324;`
- To use a variable
  - `alert(myOtherVariable);`
  - `total = anotherVariable + 14`
- To change a variable's value
  - `myOtherVariable = "Something else";`
  - `anotherVariable = 0;`

Creating and Using Arrays

- To create an array
  - `var myArray = []; // Creates an empty array`
  - `var myArray = ['thing1','thing2','thing3'];`
- To use an array
  - `alert(myArray[0]); // alerts 'thing1'`
  - `var myOtherArray[0] = myArray[3];`
- To change an array's value
  - `myArray[4] = 'thing5';`

JavaScript Operators

Operator	Name	Description	Example
=	assignment	sets the left equal to the right	`var a =3;`
+	addition	adds left and right	`var b = a + 1;`
-	subtraction	subtracts left from right	`var c = b - a;`
*	multiplication	multiples left and right	`var d = c * b;`
/	division	divides left by right	`var e = d / b;`
%	modulo	gets remainder of dividing left by right	`var f = d%2;`
++	increment	adds one	`var g = f++`
--	decrement	subtracts one	`var h = g--`

Quick, what's the value of h?

JavaScript Operators, cont.

Operator	Name	Description	Example
+	concatenation	combines two strings	`var i = "hi, " + "Jack!";`
==	equal	returns true if both sides are equal	`var j = a == b`
!=	not equal	returns true if both sides are not equal	`var k = a != b;`
===	strict equal	returns true if both sides are equal and are the same type	`var l = a === b;`
!==	strict not equal	returns true if both sides are not equal or if they're a different type	`var m = a !== b;`

Quick, what's the value of h?

JavaScript Operators, cont.

Operator	Name	Description	Example
>	greater than	returns true if the left is greater than the right	`var n = a > b`
<	less than	returns true if the left is less than the right	`var n = a < b`
>=	greater than or equal	returns true if the left is greater than or equal to the right	`var o = a >= b`
<=	less than or equal	returns true if the left is less than or equal to the right	`var p = a <= b`

Quick, what's the value of h?

Lab 02: Using Chrome Developer Tools – JavaScript Console

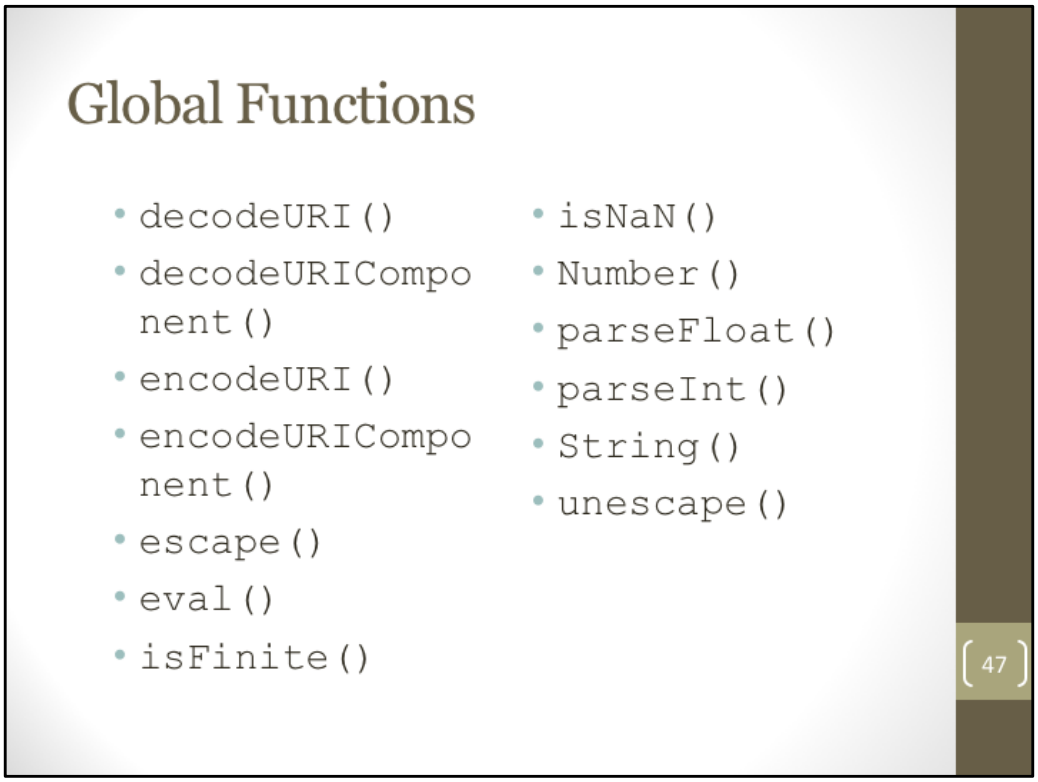
- The JavaScript console in your browser is where error and debug messages appear during the running of your program. It can also function as a place where you can test JavaScript code within your browser.

Lab 03: Using Array Methods

- JavaScript arrays have some built-in functions that make working with arrays easier and faster. In this lab, you'll use several array methods to manipulate some arrays.

Functions

- Programs within programs
- Variables declared inside a function only exist in that function
- Invoke functions using their name, followed by parentheses
- Optional comma-separated parameters may be passed between parentheses
- Examples:
  - `var a = parseInt("10");`
  - `var b = sayHello("World");`
  - `var c = String(1000);`



JavaScript built-in functions that can be run anywhere

Custom Functions

- Create functions inside your programs
- Use a `function` definition to create a function
- Use the `return` keyword to make a function return a value other than undefined

```
function myFunc(someValue) {
  var newValue = someValue + " is my
value";
  return newValue;
}
```

- To invoke `myFunc()` and assign its result to a variable

```
var result = myFunc(33);
```

JavaScript Objects

- An object is a collection of properties
- A property is made up of a name and a value
- A property may have a primitive value, an object value, or a function value
- When a property has a function value, it's called a method

Built-in Objects

- JavaScript has built-in objects
- You can use methods and properties of built-in objects by using dot notation
- Examples of built-in objects
  - Array
  - Boolean
  - Function
  - Date
  - Error
  - Object
  - RegExp

Creating Objects

- You can create custom objects in your programs
- Three ways to create objects
  1. Object literal (aka object initializer)

```
var myCar = {color: "black", doors: 4}
```

  2. Constructor Function

```
function myCar(color,doors) {
  this.doors = doors;
  this.color = color;
}
```

  3. `Object.create`
     - Allows you to specify the object the new object is based on

```
var cat = Object.create(Animal);
```

Using Objects

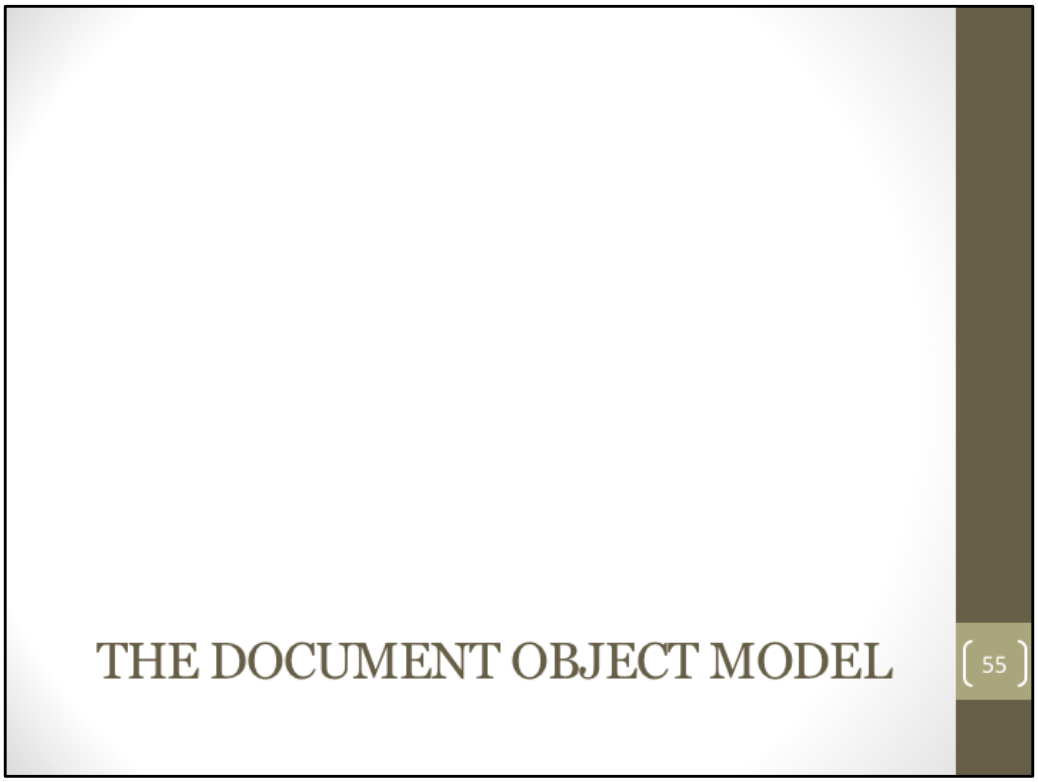
- Two ways to access and change properties
  - Dot notation
    - `var howManyDoors = myCar.doors;`
    - `alert(myCar.doors);`
  - Square brackets
    - `var howManyDoors = myCar['doors'];`
  - Methods can be set and run the same way
    - `var myCar.start = function(key,ignition){return key + ignition;}`
    - `console.log(myCar.start("click","vroom"));`

Lab 04: Using JavaScript Objects

- In this lab, you'll add properties to a painting object and retrieve and modify properties.

Prototypal Inheritance

- JavaScript objects have a link to a prototype object
- You can create new objects by copying an existing one
- When you access a property on an object, JavaScript looks for that property in the prototype (and all the way up the chain) if it doesn't find it on the referenced object.

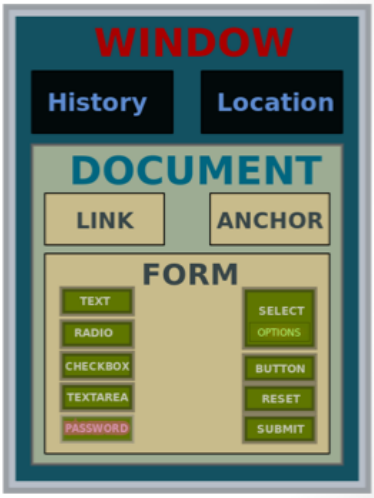


Objectives

- Understand how the DOM works
- Select DOM nodes
- Manipulate the DOM with JavaScript

What Is the DOM?

- JavaScript API for HTML documents
- Represents elements as a tree structure
- Objects in the tree can be addressed and manipulated using methods



JohnManuel [GFDL (http://www.gnu.org/copyleft/fdl.html) or CC BY-SA 3.0 (http://creativecommons.org/licenses/by-sa/3.0)], via Wikimedia Commons

Understanding Nodes

- DOM interfaces that inherit from Node
  - Document
  - Element
  - `CharacterData`
    - Text
    - Comment
  - `DocumentType`
- Nodes inherit properties from `EventTarget`

EventTarget

- An interface implemented by objects that can receive events (a.k.a., event targets)
- Examples:
  - Element
  - Document
  - Window
- Many event targets support setting event handlers

Things that happen in the DOM

Other Events

- Many interfaces implement events or inherit events
- Visit http://developer.mozilla.org/en-US/docs/Web/Events

Element

- Provides an interface for elements within a document
- Inherits properties and methods from Node and EventTarget
- Most common properties:
  - `innerHTML`
  - `attributes`
  - `classList`
  - `id`
  - `tagName`
- Most common methods
  - `getElementById`
  - `addEventListener`
  - `querySelectorAll`

Manipulating HTML with the DOM

- You can get and set properties of HTML elements with JavaScript through the DOM

```
//starting HTML and DOM Element
<p id="favoriteMovie">The Matrix</p>

<script>
getElementById("favoriteMovie")
    .innerHTML = "The Godfather";
</script>

//updates DOM, which updates the browser
<p id="favoriteMovie">The Godfather</p>
```

Manipulating HTML with the DOM

```
<ol id="favoriteSongs">
  <li class="song"></li>
  <li class="song"></li>
  <li class="song"></li>
</ol>

<script>
var mySongs=document
      .querySelectorAll("#favoriteSongs .song");
mySongs[0].innerHTML = "My New Favorite Song";
</script>
```

Manipulating HTML with JQuery

```
<ol id="favoriteSongs">
  <li class="song"></li>
  <li class="song"></li>
  <li class="song"></li>
</ol>

<script>
$("#favoriteSongs .song").first()
      .html("My New Favorite Song");
</script>
```

Manipulating HTML with React

```
<div id="favoriteSongs"></div>

<script>
var FavoriteSongs = React.createClass({
  render: function() {
    return (
    <ol>
      <li className="song">{this.props.song}</li>
      <li className="song"></li>
      <li className="song"></li>
    </ol>
    );
  }
});
ReactDOM.render(<FavoriteSongs song="My New Favorite Song" />,
        document.getElementById('favoriteSongs'));
</script>
```

Lab 05: Performing DOM Manipulation

- You're going to use JS Bin to write an exciting game, set in the distant future. You're the pilot of a spaceship armed with a powerful anti-gravity laser system. You've been sent on a mission to destroy approaching asteroids before they strike your home world and cause mass extinctions.

SURVEY OF POPULAR LIBRARIES AND FRAMEWORKS

JQuery

- "Write less, do more"
- The most widely-used JavaScript library
- Makes AJAX easier
- Simplifies DOM manipulation
- Smooths differences between browsers
- Is becoming less necessary as JavaScript adopts similar features and browser support for JavaScript improves

Angular

- Client-side framework
- Extends HTML with new attributes and elements
- Data binding updates the view when the data model changes
- Controllers separate data from the view
- Directives === reusable components

Backbone

Lightweight model-view-presenter framework

Uses one-way data flow

React

- JavaScript library for building user interfaces
- Component-based
- Keeps a Virtual DOM in memory and uses it to calculate minimal updates to the actual DOM
  - Faster updates
  - Declarative DOM updates
- Declarative
  - Describes how you want the component to look and React figures out how to make it look that way
  - Different from "imperative" updates done by jQuery and others, where you say what you want to update specifically

Lab 06: Using jQuery

- In this lab, you'll use jQuery to build a simple program for chatting with yourself or with someone sitting at the same computer as you.

Lab 07: Using AJAX with jQuery

- In this lab, you'll use AJAX to dynamically load a file and display its contents inside an HTML page.

REST

- Representational State Transfer
- Uses HTTP Methods to work with Web APIs
- HTTP Methods
  - `GET`
  - `POST`
  - `PUT`
  - `DELETE`
- Requests return JSON data

Browser vs. REST

Browser	REST

• Uses HTTP
  • `GET` fetches a website
  • `POST` submits form data
• Server response is a web page (HTML)
	• Uses HTTP
	  • `GET` fetches data
	  • `POST` posts new data
	  • `PUT` updates existing data
	  • `DELETE` deletes data
	• Server response is JSON data

Lab 08: Using JSON and REST to Work with Rooms

- In this lab, you'll use JSON Data along with REST to list rooms, create rooms, and add people to rooms in Cisco Spark.

Lab 09: Using JavaScript Functions

- In this lab, you'll write a single function to format data from the Cisco Spark Rooms API in different ways.

Module: Back-end Tools and Techniques

Objectives:

- Use Version Control
- Test Driven Development
- Understanding modularity

Objectives

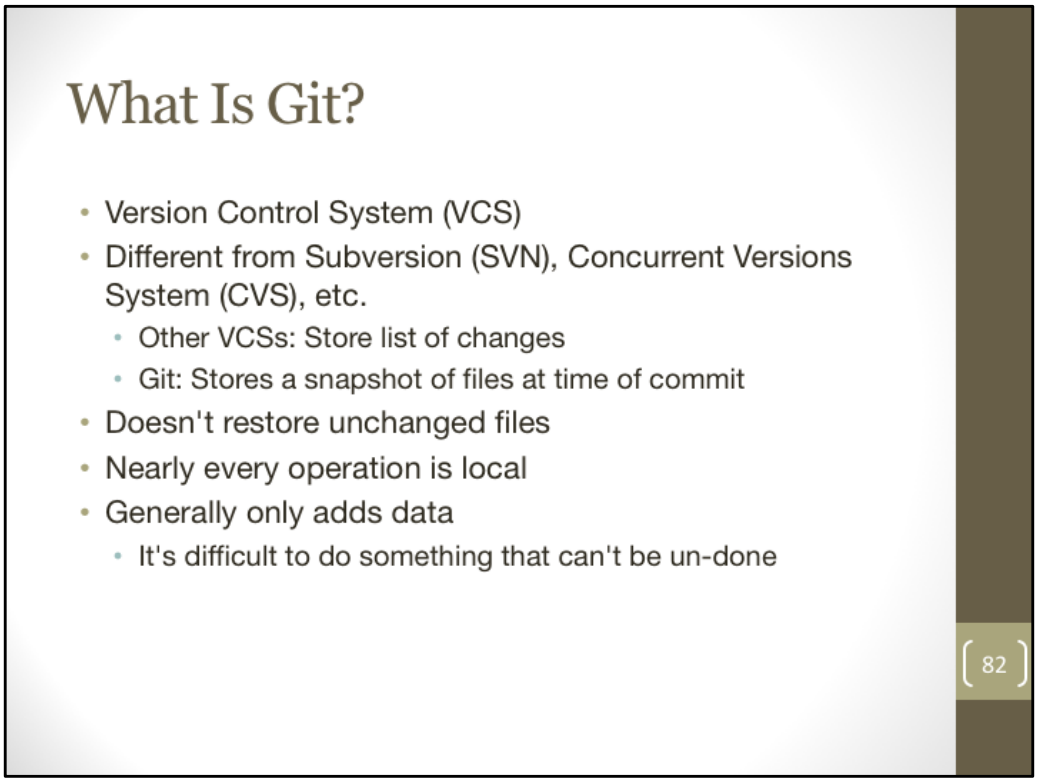
- Learn about how Git works
- Learn a typical Git workflow
- Use the most common Git commands

What Is Version Control?

- Is an essential component of any development workflow
- Records changes to files over time
- Allows recalling of specific versions
- Makes collaboration possible
- Makes software development safer

History of Git

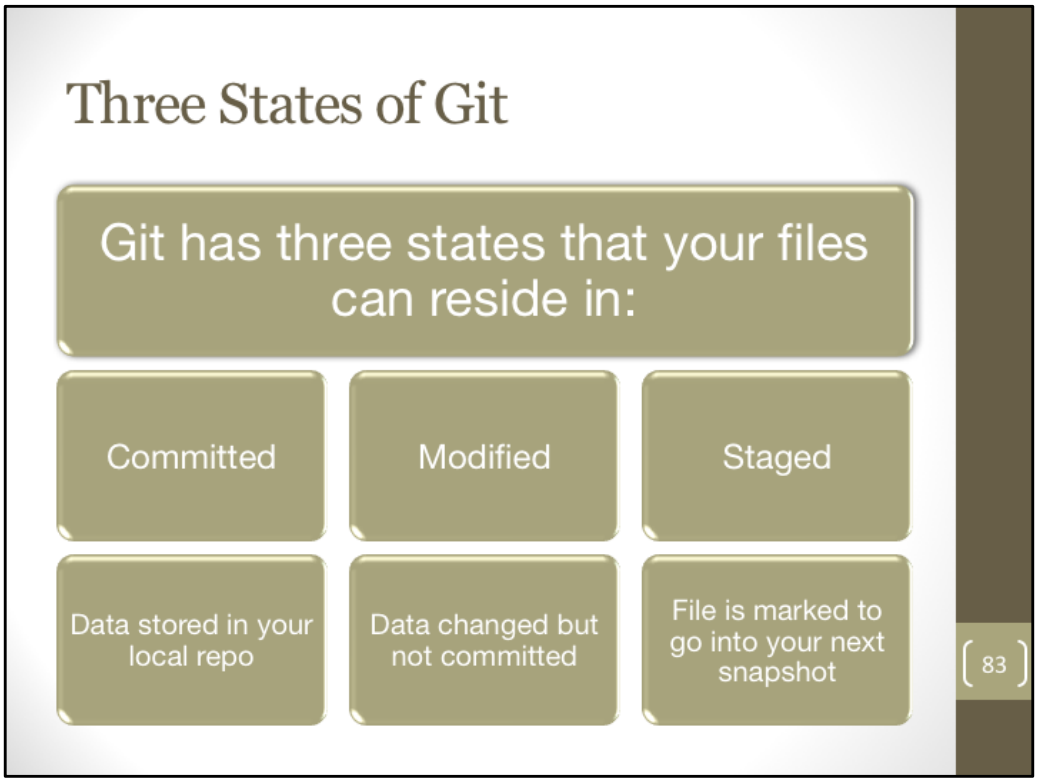
- From 2002 - 2005, the Linux Kernel used the proprietary BitKeeper Version Control System (VCS)
- In 2005, after a falling out with BitKeeper's developer, they decided to create their own VCS
- Goals
  - Speed
  - Simple design
  - Strong support for non-linear development
  - Fully distributed
  - Able to handle large projects



Git thinks about its data as a stream of snapshots. When you commit, git stores a snapshot of what the files look like. If something hasn't changed, git doesn't store the file again, just a reference to the previous identical file it has already stored.

Most operations in Git only need local files and resources to operate. You have the entire history of the project on your local disk, which makes operations much faster.

Also, there's very little you can't do if you're offline.

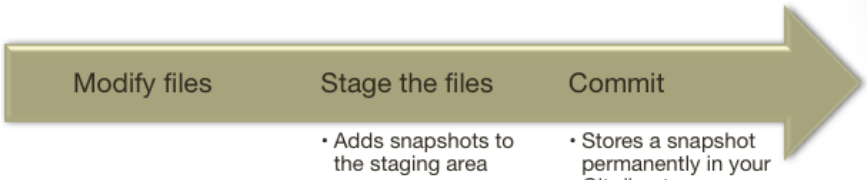


The Git directory (repository) is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

The working directory is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

The staging area is a file—generally contained in your Git directory—that stores information about what will go into your next commit. It's sometimes referred to as the "index"; it's also common to refer to it as the staging area.

Git Workflow



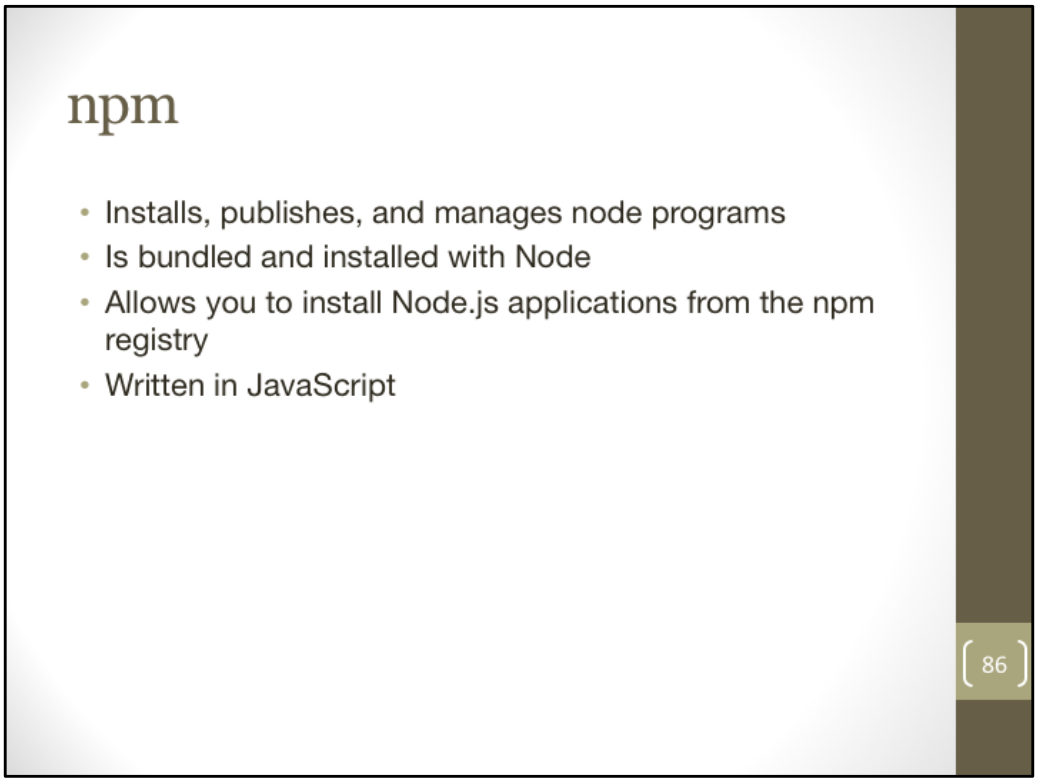
- Adds snapshots to the staging area
- Git add
- Git add is dual-purpose: Tells git to track new files and stages changes to existing files

- Stores a snapshot permanently in your Git directory
- Git commit

You can also skip the staging area by using the `-a option` with git commit. This will automatically stage all tracked files before doing the commit.

Lab 01: Controlling Your Versions with Git

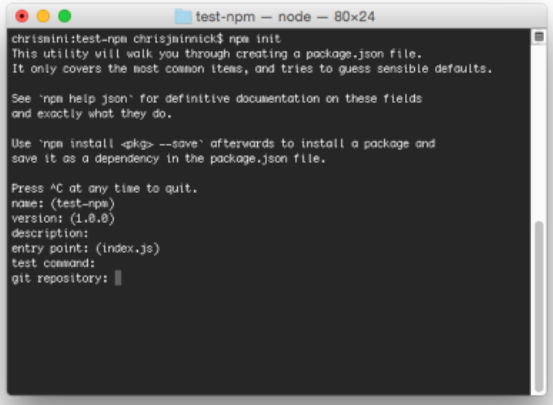
- In this lab, you will learn some basic Git commands.



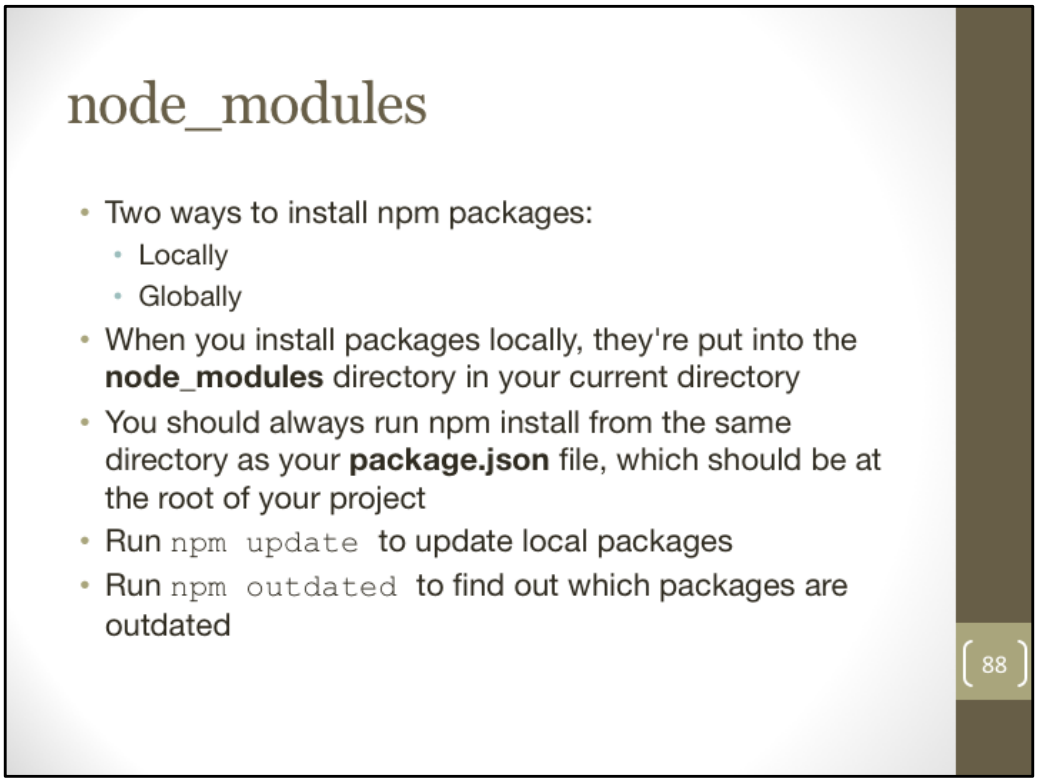
To install Gulp, and to install and manage all of the node packages we'll be using, we'll use npm.

Lab 02: Initializing npm

In this lab, you will initialize npm for your project and learn about the package.json file.



npm stores information about the packages installed in your project in package.json. Run npm init to create a package.json file.



Important note: This isn't an issue at this point, but if you're working on Windows, you're going to run into a problem sooner or later. Modules that have long chains of dependencies can create long file paths, which can break on Windows. Running npm dedupe usually fixes the problem. Other possible solutions can be found here: http://gsferreira.com/archive/2015/07/reduce-the-path-length-of-a-node-js-project/

package.json

- Manages locally installed npm packages
- Documents packages your project depends on
- Specifies the versions of each package your project can use
- Makes your build reproducible
- Package versions are specified using Semantic versioning (`semver`)
- Semver ranges:
  - `~` : patch release - 1.0.x
  - `^` : minor release - 1.x
  - `*` : major release - x

Npm Install

- Is used to download and install a package
- `-g`
  - Install globally
- `--save`
  - Package will appear in your dependencies
- `--save-dev`
  - Package will appear in your `devDependencies`
- `--save-optional`
  - Package will appear in your `optionalDependencies`
- `--save-exact`
  - Saved dependency will be configured with an exact version rather than the default range operator

Lab 03: Using npm

Objectives

- Learn about Lint tools
- Use JSHint
- Configure JSHint
- Manual testing with a local web server

Linting tools look at your code and look for errors and bad practices.

The first linting tool for JavaScript is JSLint, which was created by Douglas Crockford.

JSLint flags a lot of style issues that Crockford thinks are bad.

JSHint was created in order to give more control over which style issues are checked for.

ESLint is a new tool that is highly customizable and features plugins for many different styles, libraries, and frameworks -- including React and JSX.

Two Ways to Configure ESLint

- Configuration comments
  - Embed configuration info in JS files with comments
  - `/* eslint eqeqeq: "off", curly: "error" */`
- Configuration files
  - `.eslintrc file`

ESLint: What Can Be Configured?

Environments

- Where is the code running?
- Includes predefined global variables for each environment

Globals

- Specify additional globals your scripts use

Rules

- Enable rules at different levels

Lab 05: Linting

- In this lab, you will install ESLint and configure it
  - Install JSHint or ESLint into project
  - Test it

ESLint Rules

- Three Levels
  - `"off"` or `0`
    - Rule not applied.
  - `"warn"` or `1`
    - Warn but don't exit.
  - `"error"` or `2`
    - Error and exit
- Example rules

```
{
    "rules": {
    "eqeqeq": "off",
    "curly": "error",
    "quotes": ["error", "double"]
}
}
```

Lab 06: Using Chrome Developer Tools - Sources Tab

- In this lab, you'll learn how to debug JavaScript using the sources panel.

Objectives

- Learn the TDD Steps
- Write assertions
- Understand exception handling in JS
- Create tests with Jasmine
- Automate cross-browser testing

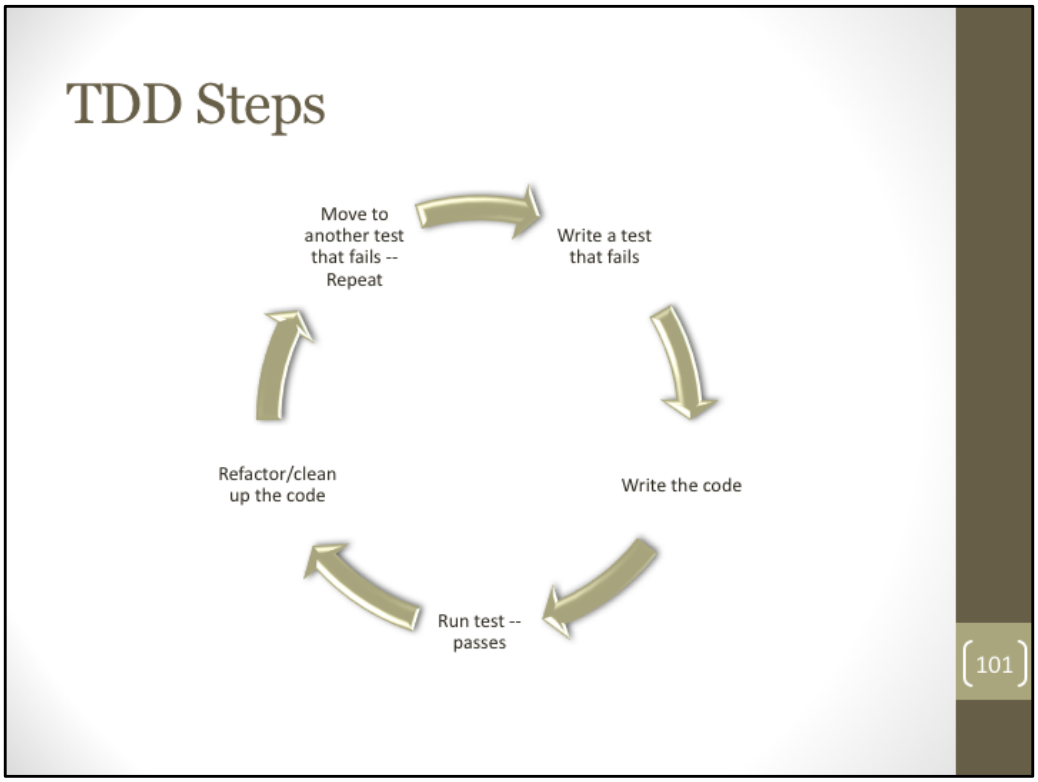
In this section, we'll start talking about test-driven development and automated testing.

The key with Test-Driven Development is to work in very small increments.

1. check that the code does was its supposed to do
2. group tests into suites
3. cross browser test automation

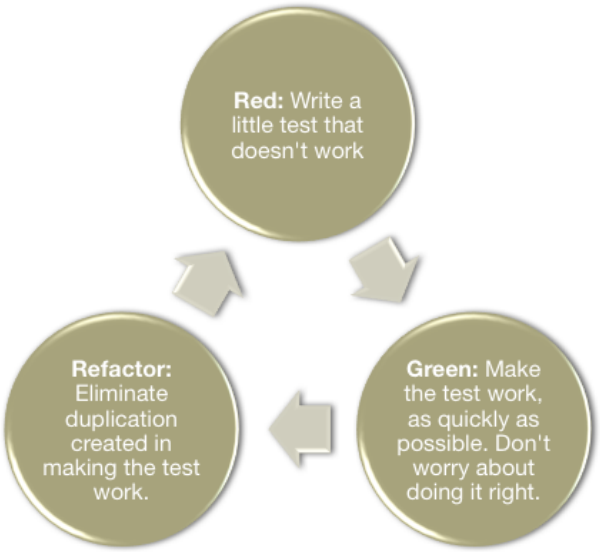
Goal of TDD


Test-driven development (TDD) helps you create clean code that works



Write a test
Test fails
Write code
Run test - passes!
Refactor
Repeat

The TDD Cycle



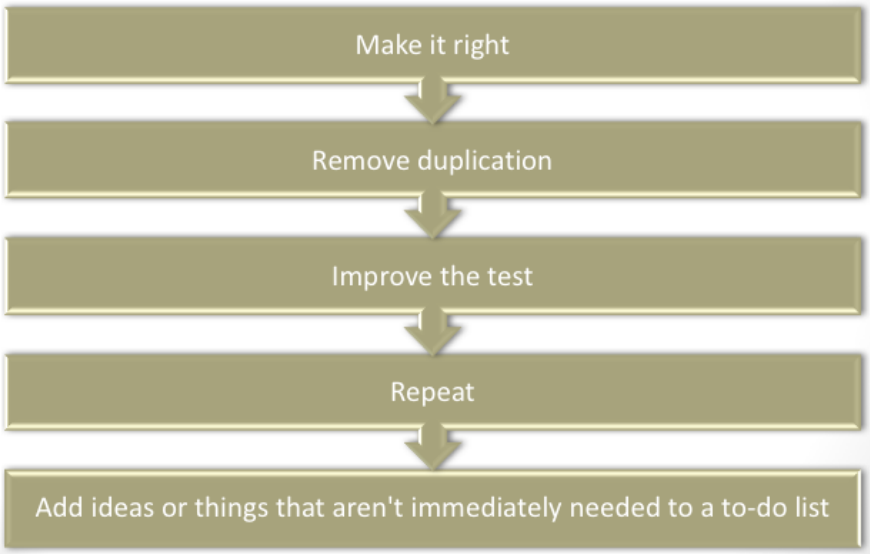
Red

- Write the story
- Invent the interface you wish you had
- Characteristics of a good tests:
  - Each test should be independent of the others
  - Any behavior should be specified in only one test
  - No unnecessary assertions
  - Test only one code unit at a time
  - Avoid unnecessary preconditions

Green

- Get the test to pass as quickly as possible
- Three strategies:
  - Fake it
    - Do something, no matter how bad, to get the test to pass
  - Use an obvious clean solution
    - But don't try too hard!
  - Triangulation
    - Only generalize code when you have two examples or more
    - When the second example demands a more general solution, then and only then, do you generalize

Refactor

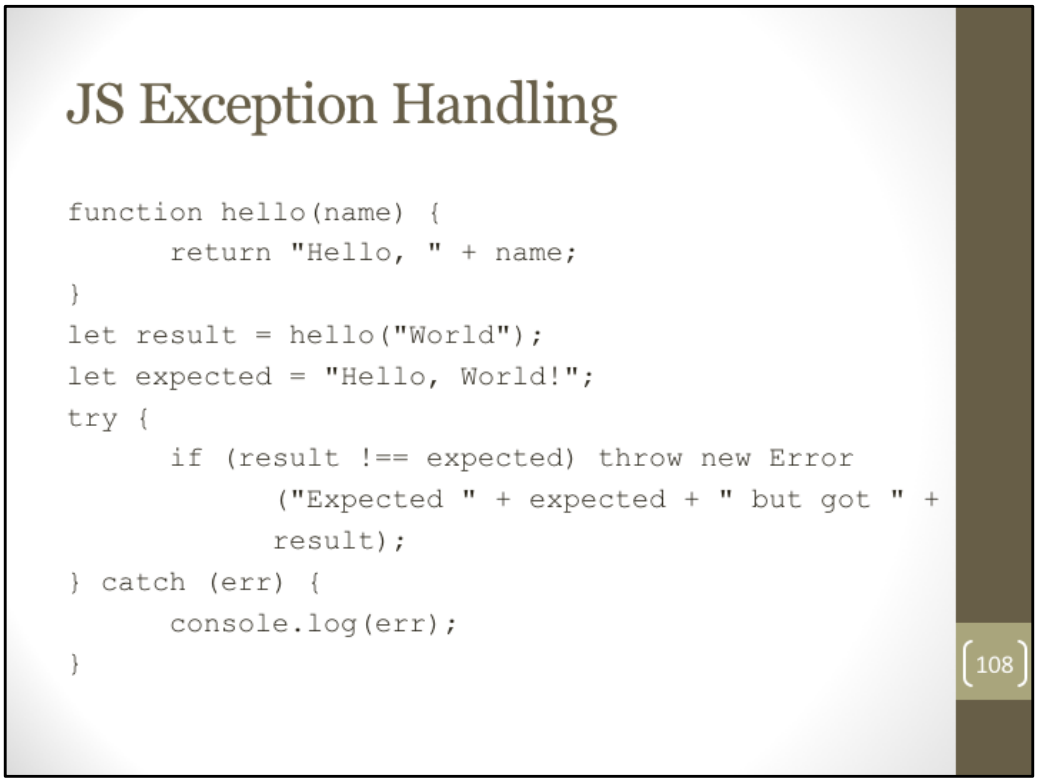

Make it right
Remove duplication
Improve the test
Repeat
Add ideas or things that aren't immediately needed to a to-do list

Assertions

- Expression that encapsulates testable logic
- Assertion Libraries
  - Chai, should.js, expect.js, better.assert
- Examples
  - `expect(buttonText).toEqual('Go!'); // jasmine`
  - `result.body.should.be.a('array'); // chai`
  - `equal($("h1").text(), "hello"); //QUnit`
  - `assert.deepEqual(obj1, obj2); //Assert`

JavaScript Testing Frameworks

- Jasmine
- Mocha
  - Doesn't include its own assertion library
- QUnit
  - From JQuery
- js-test-driver
- YUI Test
- Sinon.JS
- Jest



By using JavaScript's built-in try throw catch, you can write assertions and throw errors if the test of the assertion doesn't pass. If it doesn't, you can fix the code until the test passes. This test will fail. (node ./src/trythrowcatch.js). You could write an assertion function to simplify this, or you can use an assertion library that already exists.

Objectives

- Write test suites
- Create specs
- Set expectations
- Use matchers
- Integrate Jasmine and Gulp

Jasmine is a behavior-driven development framework that includes its own assertion library.

How Jasmine Works

- Suites describe your tests
- Specs contain expectations

```
describe("A suite is just a function", function() {
  var a;

  it("and so is a spec", function() {
    a = true;

    expect(a).toBe(true);
  });
});
```

Test Suites

- Created using the `describe` function
- Contain one or more specs
- Two params
  - Text description
  - Function

```
describe("Hello", function() {
...
}
```

Specs

- Created using the `it` functions
- Contains one or more expectations
- expectations === assertions

```
describe("Hello", function() {

    it("Concats Hello and a name", function()
{
          var expected = "Hello, World!";
          var actual = hello("World");
          expect(actual).toEqual(expected);
    });
});
```

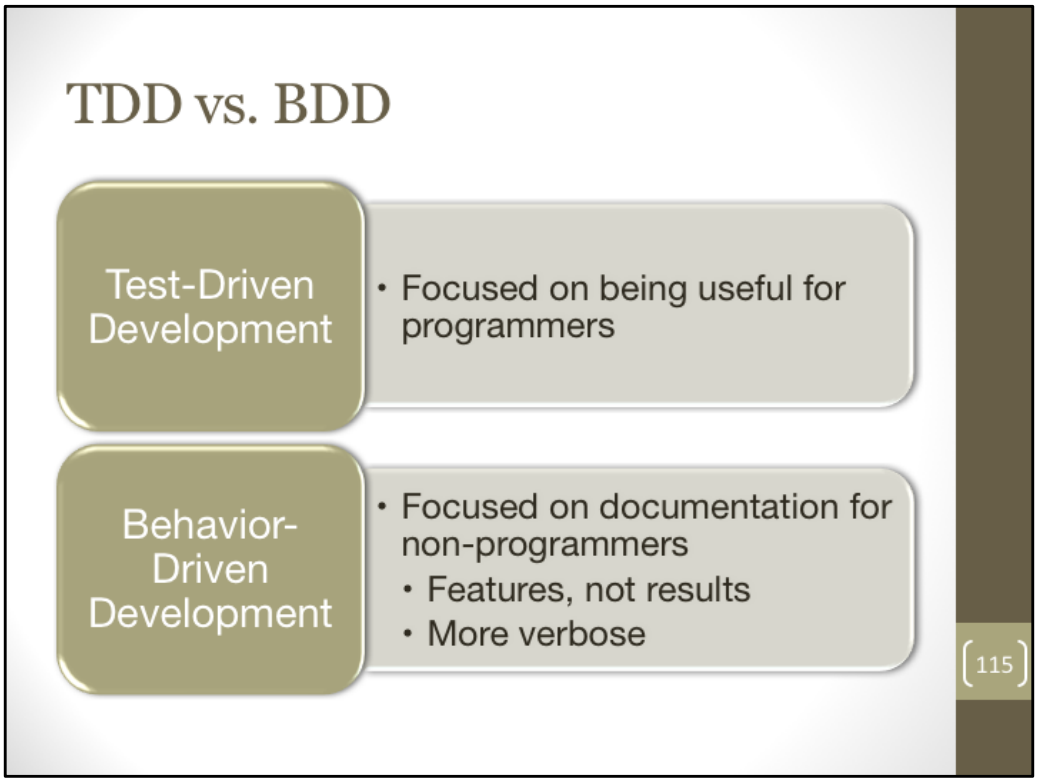
Expectations

- Made using `expect` function,
  - Takes a value
- Chained to a Matcher
  - Takes the expected value

```
expect(actual).toEqual(expected);
```

Matchers

- `expect(fn).toThrow(e);`
- `expect(instance).toBe(instance);`
- `expect(mixed).toBeDefined();`
- `expect(mixed).toBeFalsy();`
- `expect(number).toBeGreaterThan(number);`
- `expect(number).toBeLessThan(number);`
- `expect(mixed).toBeNull();`
- `expect(mixed).toBeTruthy();`
- `expect(mixed).toBeUndefined();`
- `expect(array).toContain(member);`
- `expect(string).toContain(substring);`
- `expect(mixed).toEqual(mixed);`
- `expect(mixed).toMatch(pattern);`



The difference between TDD and BDD is subtle and it's mostly in language. TDD is designed for programmers to test software, whereas BDD helps design and document it. Neither style is better than the other and the choice depends on your framework, what you're comfortable with, and other factors.

TDD Example

```
suite('Counter', function() {
  test('tick increases count to 1',

   function() {
       var counter = new Counter();
       counter.tick();
       assert.equal(counter.count, 1);
     });

});
```

BDD Example

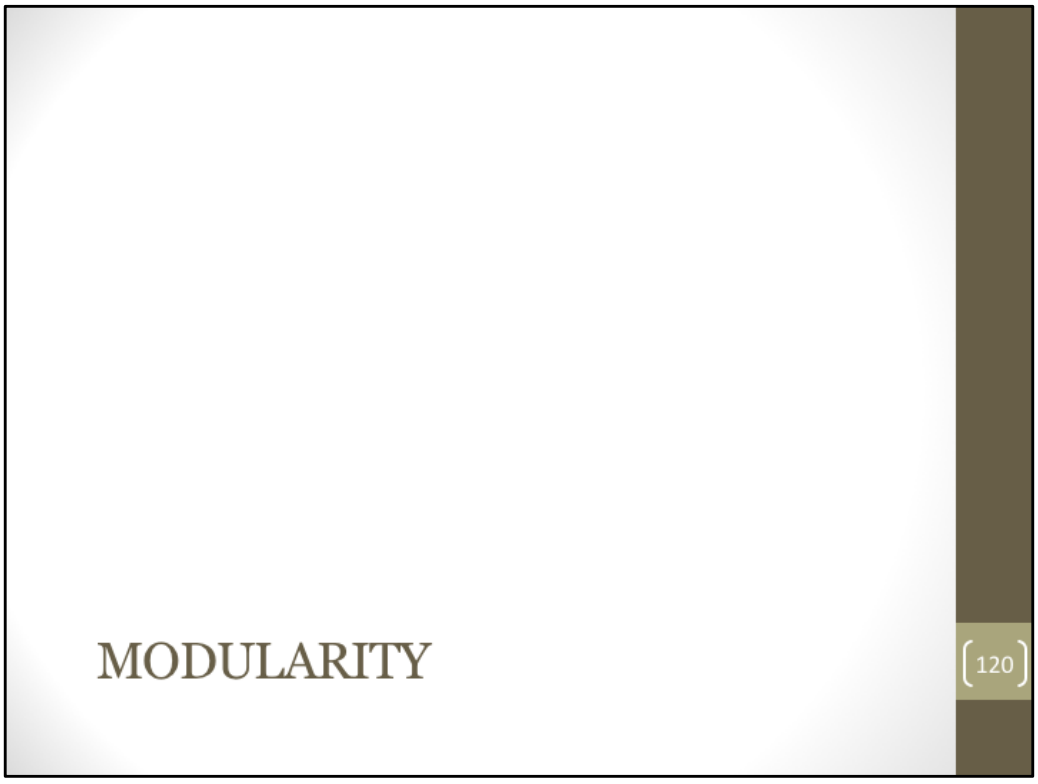
```
describe('Counter', function() {
  it('should increase count by 1 after calling
tick',
   function() {
       var counter = new Counter();
       var expectedCount = counter.count + 1;
       counter.tick();
       assert.equal(counter.count, expectedCount);
     });
});
```

Lab 07: Getting Started with Jasmine

- In this lab, you will install Jasmine and use it to create your first test suite.

Lab

- We need more features!!! Pick a feature and implement it using TDD. Break up the feature into smaller units as needed.
  - It gives an appropriate hello for the time of day
    - Good morning!
    - Good afternoon!
    - Good evening!
  - It displays a login message if no name is provided
  - It speaks German to Germans
  - It refuses to say hello after the fourth time the function is called



Objectives

- Explain modularity
- Learn different methods of using modules in JS
- Understand methods of front-end module management

Why Is Modularity Important?

- Allows individual modules to be tested
- Allows distributed development
- Enables code reuse
- Reduces coupling
- Increases cohesion

CommonJS

- Modularity for JavaScript outside of the browser
- Node.js is a CommonJS module implementation
- Uses `require` to include modules

- Export an anonymous function

hello.js

```
module.exports = function () {
  console.log("hello!");
}
```

app.js

```
var hello =
require("./hello.js");
```

- Export a named function

hello.js

```
exports.hello = function () {
  console.log("hello!");
}
```

app.js

```
var hello =
require("./hello.js").hello;
```

ES6 Modules

- Compromise between AMD and CommonJS
  - Compact syntax
  - Support for asynchronous loading
- Two types
  - Named exports
    - Multiple per module
  - Default exports
    - One per module

- Named export
- lib.js

```
export function square(x) {
  return x * x;
}
```

- main.js

```
import {square} from 'lib';
```

- Default export
- myFunc.js

```
export default function() {
…
};
```

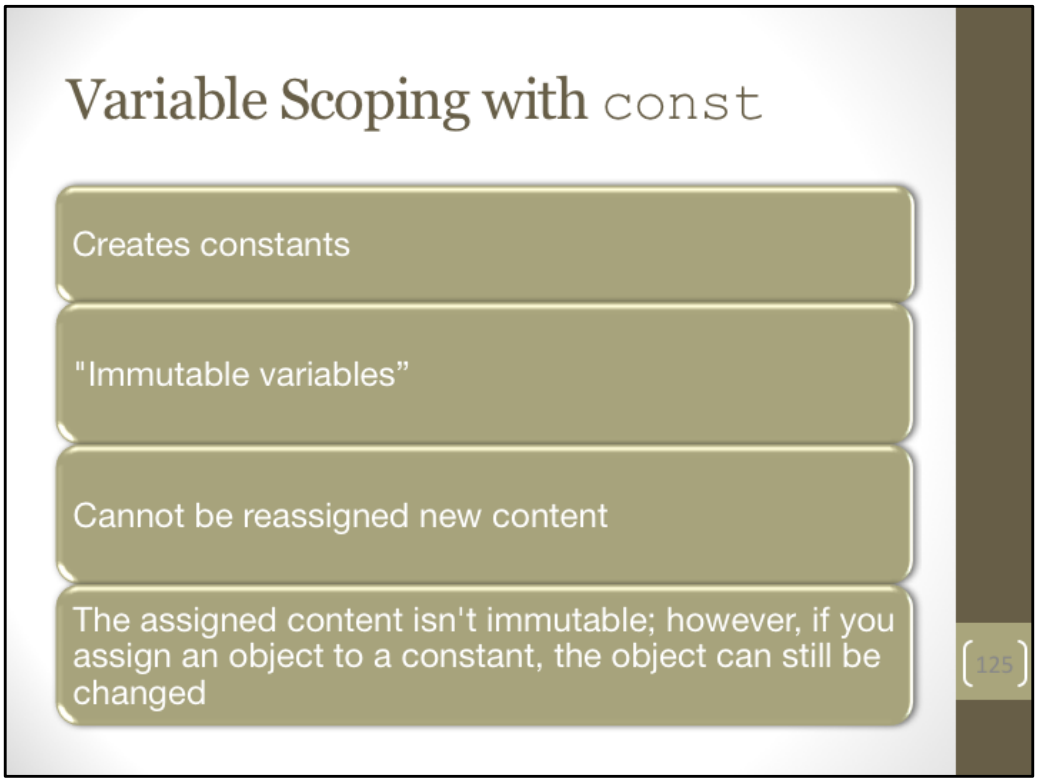
- main.js

```
import myFunc from 'myFunc';
```

Module: Advanced JavaScript

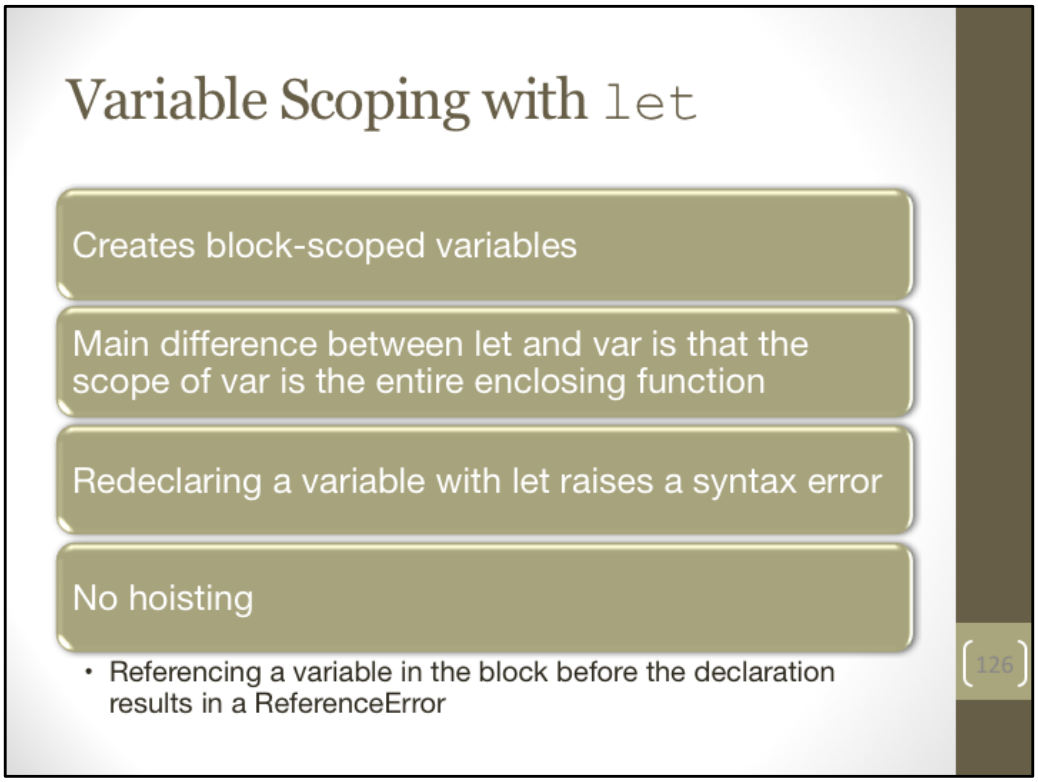
Objectives:

- Learn about scope
- Learn new ES6 syntax
- Use Babel to transpile ES6+ code



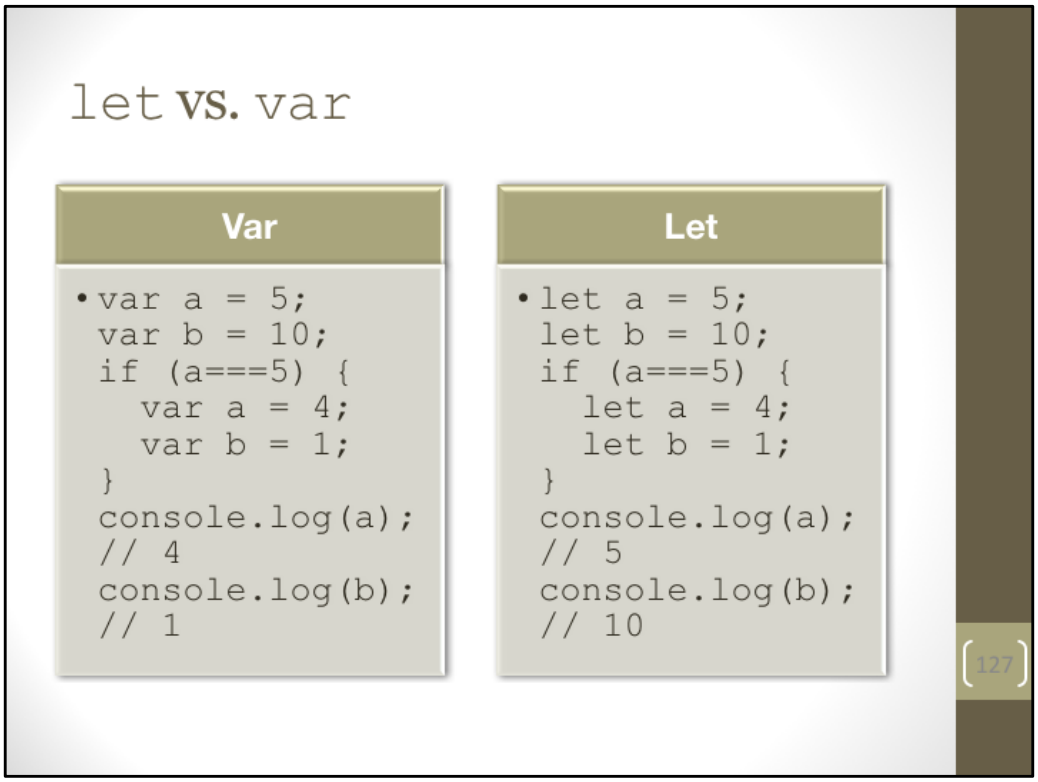
The first new things in ES6 that we'll be talking about are const and let. Unlike variables defined using the var keyword, these new es6 keywords create block-scoped variables. They have some other important differences as well.

Const creates immutable variables -- that is, once you assign a value to a const, the value can't be changed.

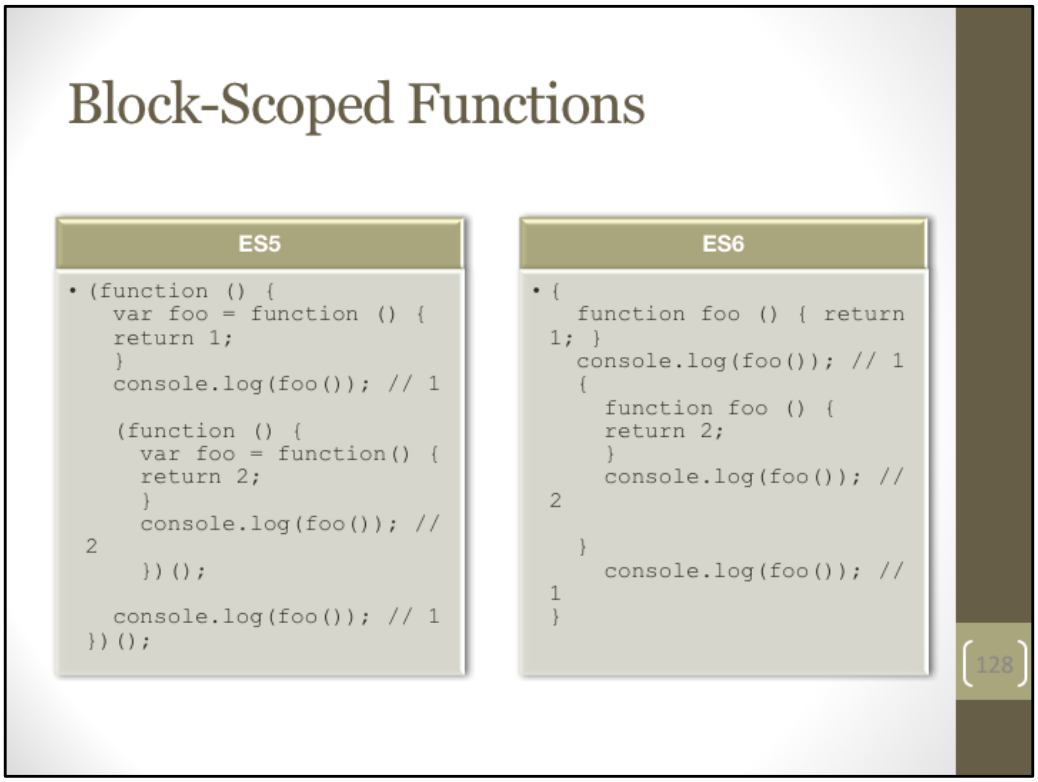


The first new things in ES6 that we'll be talking about are const and let. Unlike variables defined using the var keyword, these new es6 keywords create block-scoped variables. They have some other important differences as well.

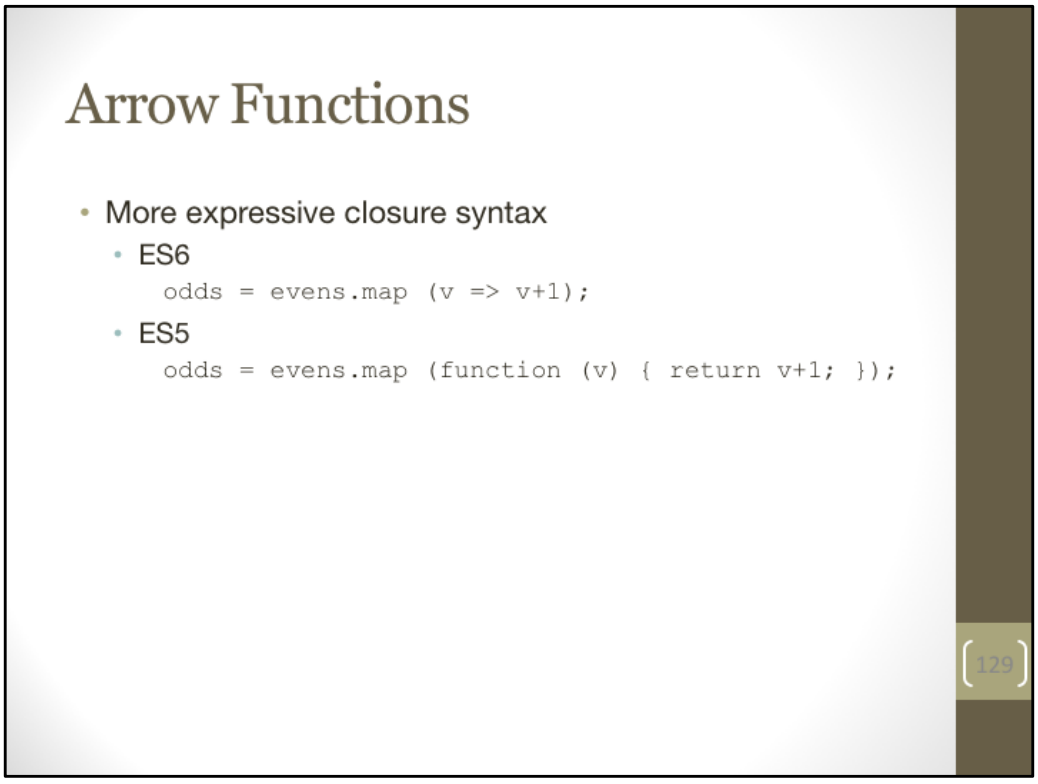
Let creates block-scoped variables (variables that are private to their enclosing curly braces), as opposed to having function scope as javascript up to this point



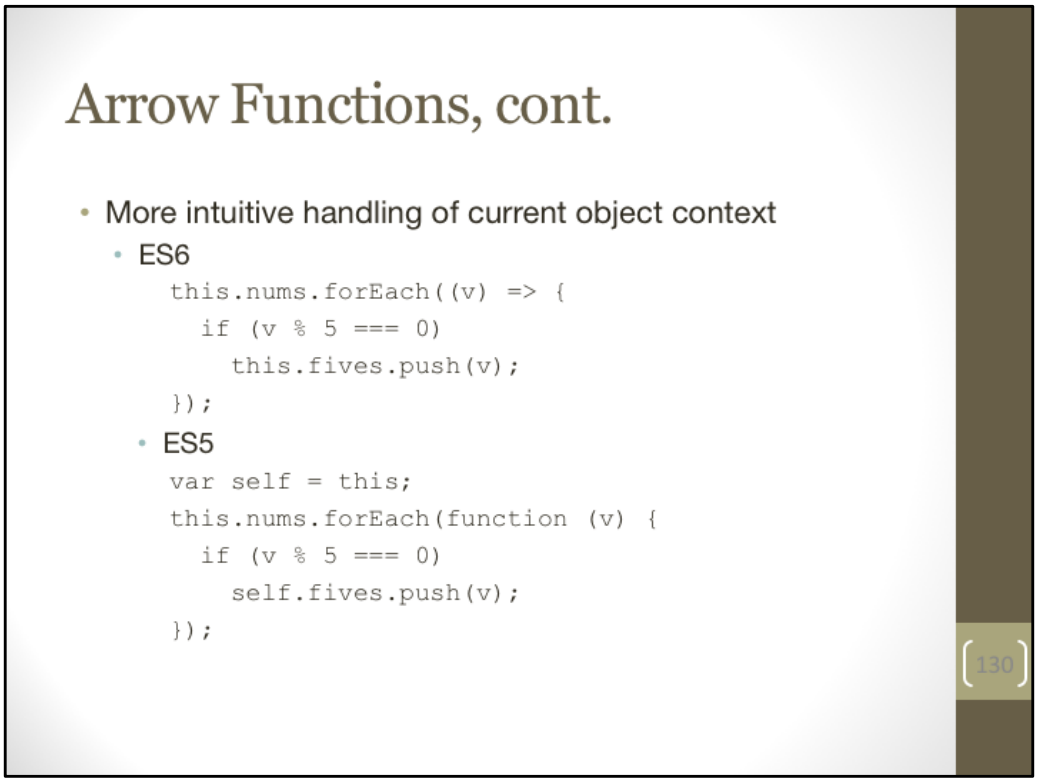
Here's an example that demonstrates the difference between scoping with let and var.



In ES5, we used iffe and function expressions to block-scope function definitions. In ES6, we can create block-scoped function definitions using brackets.



The results of both of these expressions are the same: An array of odd numbers created from an array of even numbers. But, the ES6 syntax is more concise and easier to understand.



Arrow functions are always anonymous functions, which means that the only way to assign a name to them is to assign the function to a variable. Arrow functions also handle the this keyword more intuitively, which eliminates the need for the hack shown here in order to refer to the object that a function is inside.

Default Parameter Handling

- ES6

```
function myFunc (x, y = 0, z = 13) {
  return x + y + z;
}
```

- ES5

```
function f (x, y, z) {
    if (y === undefined)
        y = 0;
    if (z === undefined)
        z = 13;
    return x + y + z;
};
```

Rest Parameter

- Aggregation of remaining arguments into single parameter of variadic functions

```
function myFunc (x, y, ...a) {
    return (x + y) * a.length;
}
console.log(myFunc(1, 2, "hello", true, 7));
```

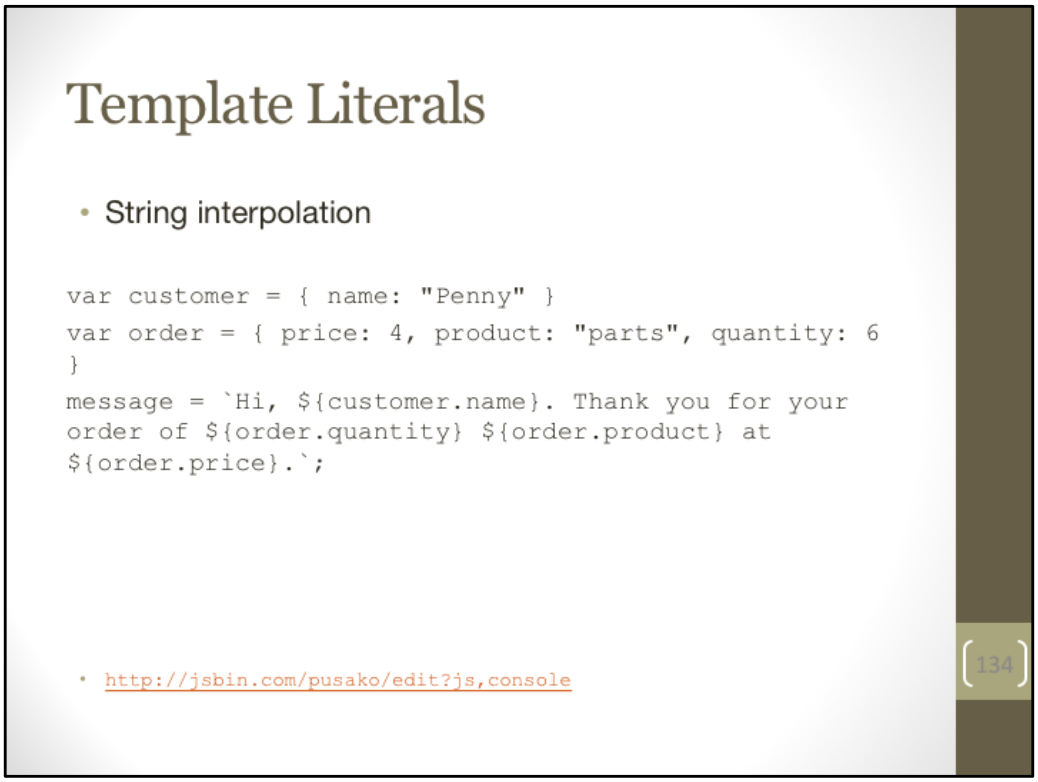
- http://jsbin.com/pisupa/edit?js,console

Spread Operator

- Spreads elements of an iterable collection (like an array or a string) into both literal elements and individual function parameters

```
var params = [ "hello", true, 7 ];
var other = [ 1, 2, ...params ];
console.log(other); // [1, 2, "hello", true, 7]

console.log(MyFunc(1, 2, ...params));

var str = "foo";
var chars = [ ...str ]; // [ "f", "o", "o" ]
```

- http://jsbin.com/guxika/edit?js,console

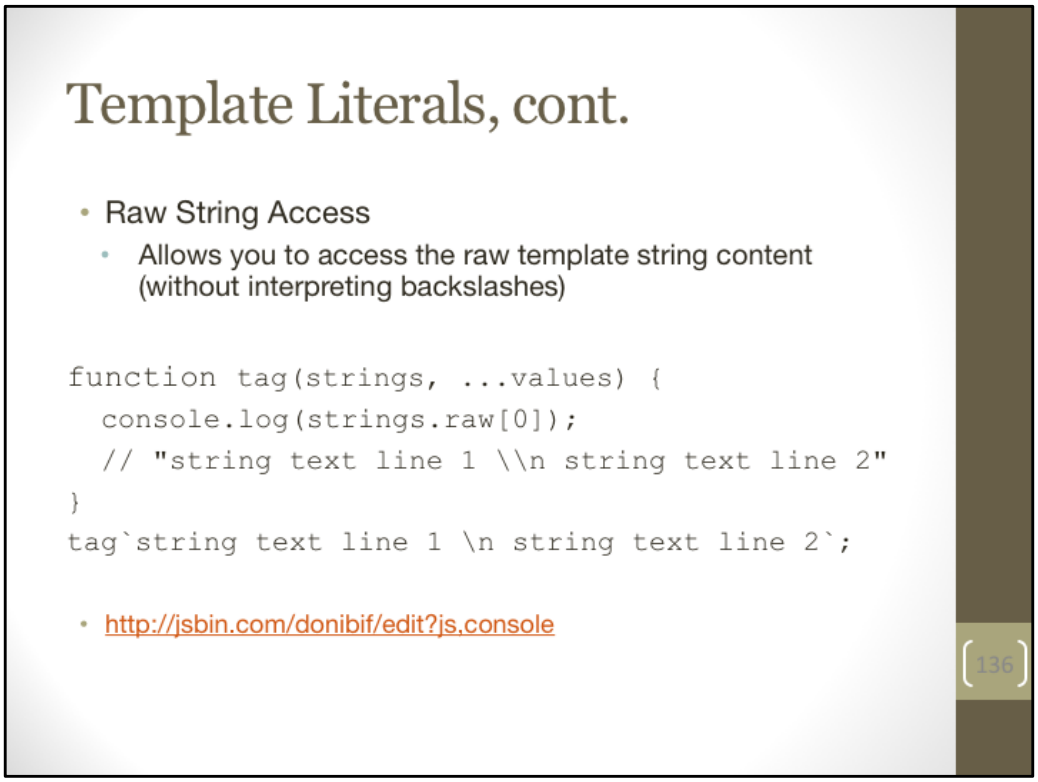


Template literals are enclosed by the back-tick (` `) (grave accent) character instead of double or single quotes.

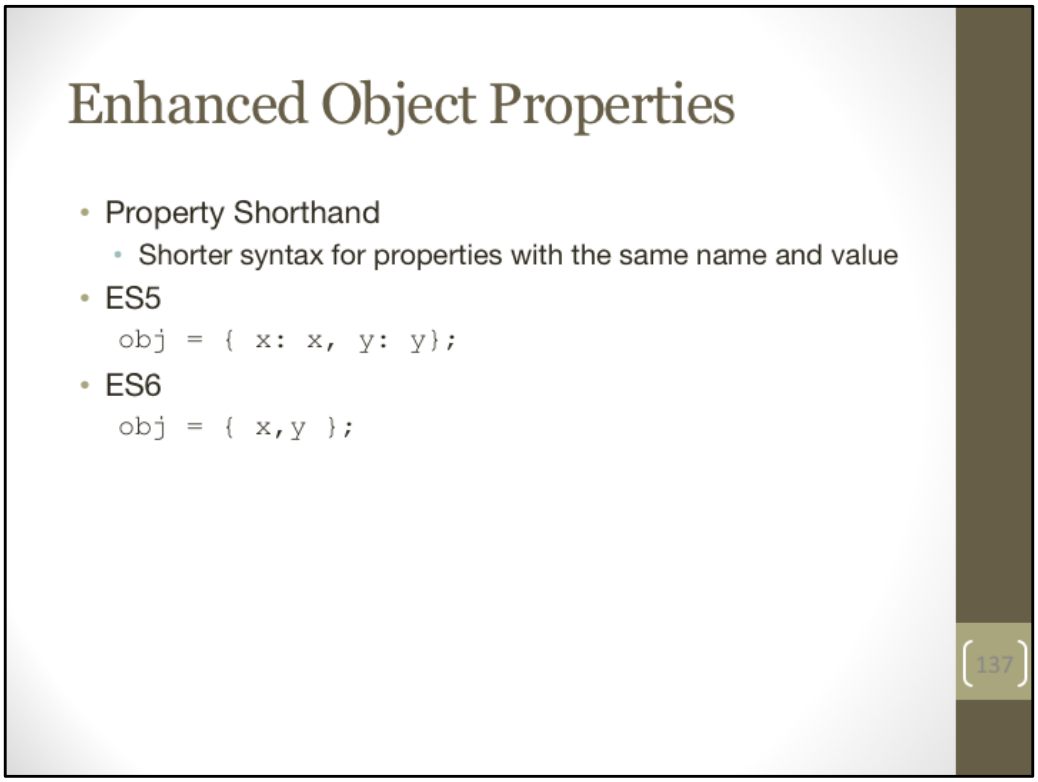
Template Literals, cont.

- Custom Interpolation
- Expression interpolation for arbitrary methods

```
get`http://example.com/cart?order=${orderId}`;
```



You can also access raw strings, without interpretted backslashes.



ES6 has a consise syntax for setting properties that have the same name and value. In this case, which part--the name or the value--do you think the x and y in the ES6 code refers to? It's actually the value. Can you think of a way to check this?

Enhanced Object Properties

- Computed names in object property definitions

```
let obj = {
  customer: "Nigel",
  [ "order" + getOrderNum() ]: 10
};
```

- http://jsbin.com/wejuqe/edit?js,console

Method Notation

ES6

```
obj = {
    foo (a,b)
{
    },
    bar (x,y)
{
    }
};
```

ES5

```
obj = {
    foo:
function(a, b)
{
    },
    bar:
function(x, y)
{
    }
};
```

Array Matching

- Intuitive and flexible destructuring of arrays into individual variables during assignment

```
var list = [ 1, 2, 3 ];
var [ a, , b ] = list; // a = 1 , b = 3
[ b, a ] = [ a, b ];
```

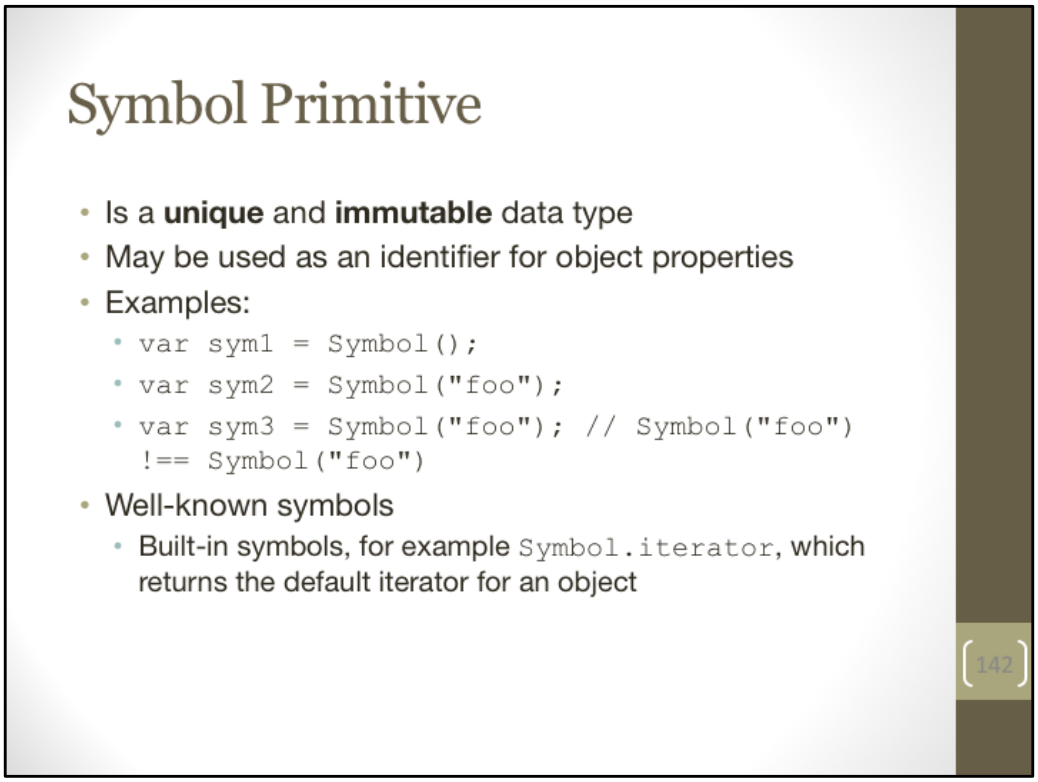
- http://jsbin.com/yafage/edit?js,console

Object Matching

- Flexible destructuring of objects into individual variables during assignment

```
var { a, b, c} = {a:1, b:2, c:3};
console.log(a); // 1
console.log(b); // 2
console.log(c); // 3
```

- http://jsbin.com/kuvizu/edit?js,console



Each call to Symbol creates a new symbol. In the second and third examples, foo is not coerced into a symbol. "foo" is just an optional description.

User-defined Iterators

- Customizable iteration behavior for objects
- In order to be iterable, an object must implement the iterator method
- The object, or one of the objects up its prototype chain, must have a property with a `Symbol.iterator` key.

```
var myIterable = {}
myIterable[Symbol.iterator] = function* () {
    yield 1;
    yield 2;
    yield 3;
};
[...myIterable] // [1, 2, 3]
```

For-Of Operator

- Convenient way to iterate over all values of an iterable object

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1
    return {
      next() {
        [ pre, cur] = [cur, pre + cur]
        return { done: false, value: cur }
        }
      }
    }
}
for (let n of fibonacci) {
    if (n > 1000)
      break;
    console.log(n);
}
```

- http://jsbin.com/nururoz/edit?js,console

144

Creating and Consuming Generator Functions

- A generator is a special type of function that works as a factory for iterators.

```
function* idMaker(){
  var index = 0;
  while(true)
    yield index++;
}
var gen = idMaker();
console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
```

- http://jsbin.com/pozite/edit?js,console

Class Definition

ES6 introduces more OOP-style classes

Can be created with Class declaration or Class expression

Class Declaration

```
class Square {
  constructor (height, width) {
    this.height = height;
    this.width = width;
    }
}
```

Class Expressions

- Can be unnamed

```
var Square = class {
  constructor(height,width) {
    this.height = height;
    this.width = width;
  }
};
```

- Or named

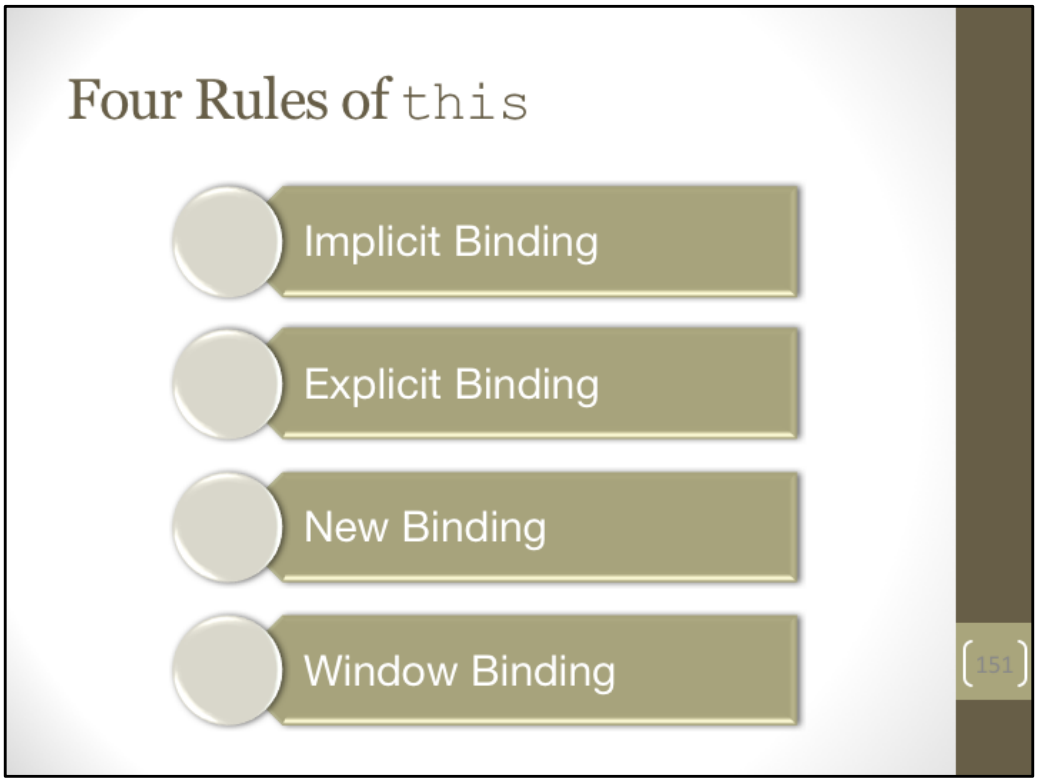
```
var Square = class Square {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Class Inheritance

```
class Rectangle extends Shape {
       constructor (id, x, y, width, height) {
         super (id, x, y);
         this.width = width;
         this.height = height;
       }
}
class Circle extends Shape {
       constructor (id, x, y, radius) {
         super (id, x, y);
         this.radius = radius;
       }
}
```

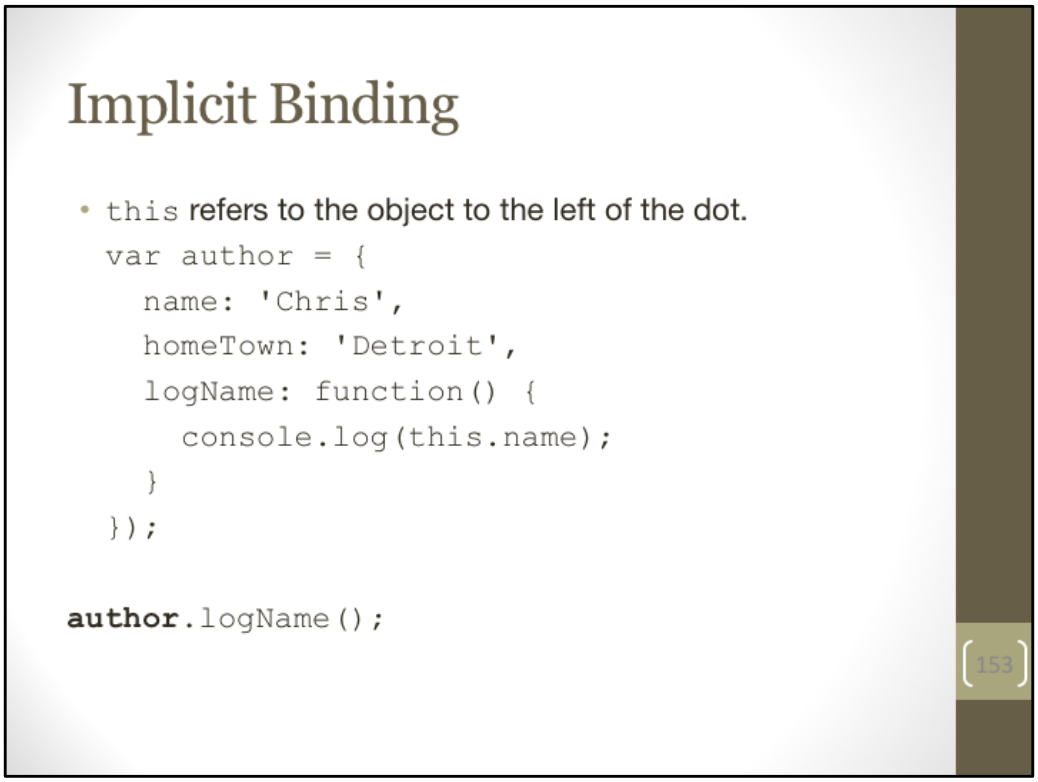
Understanding `this`

- Allows functions to be reused with different context
- Indicates which object should be focal when invoking a function

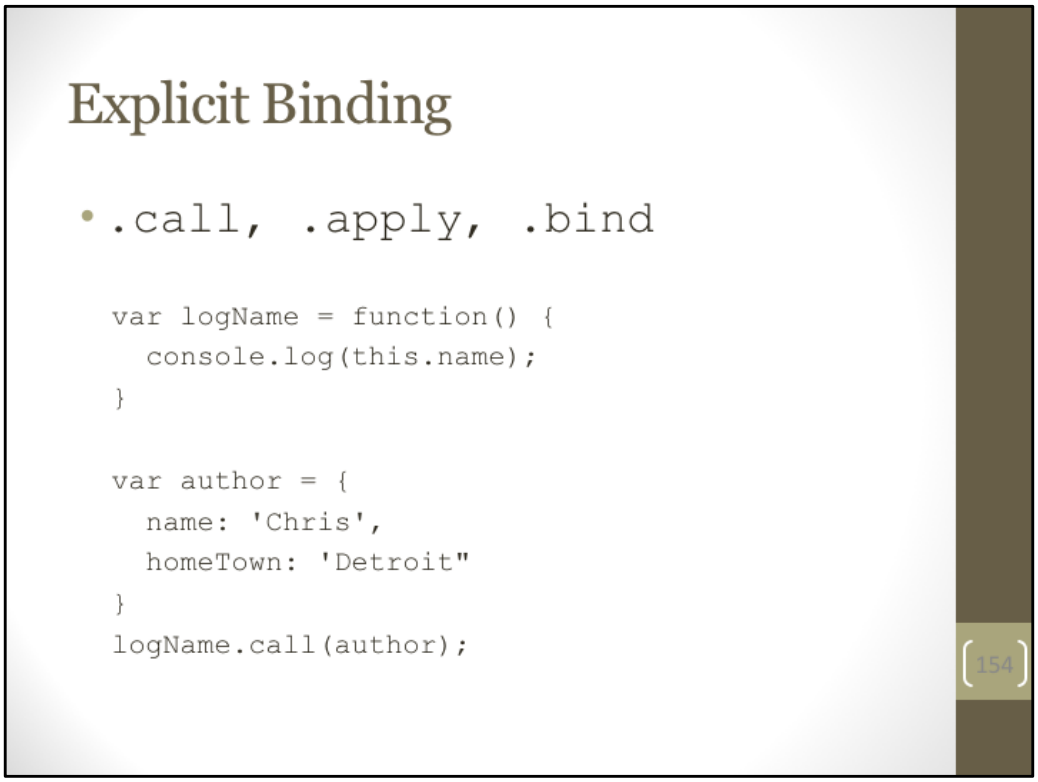

Four Rules of this
Implicit Binding
Explicit Binding
New Binding
Window Binding
151

What is `this`?

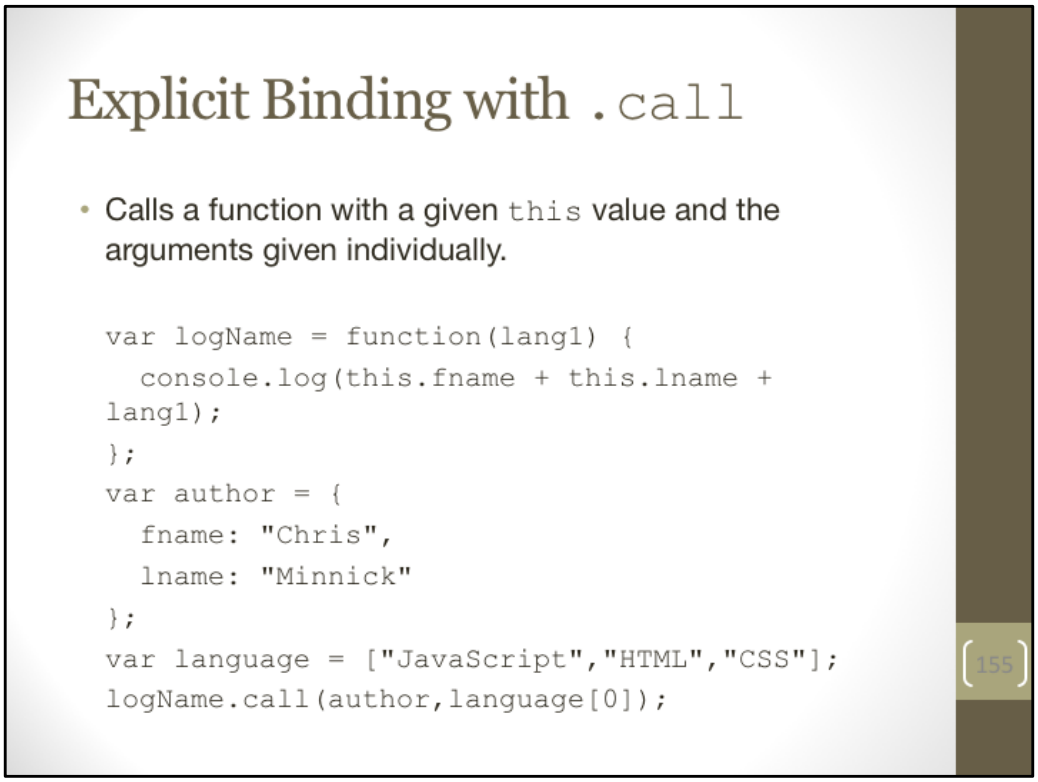
- When was function invoked?
- We don't know what `this` is until the function is invoked



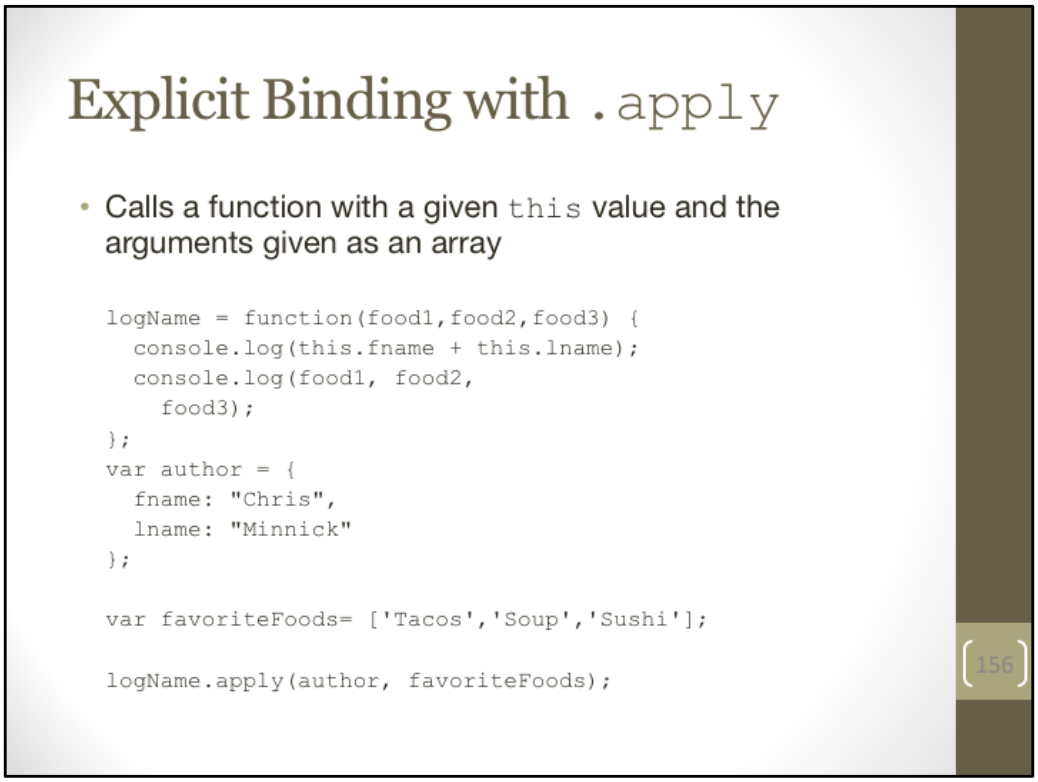
In this case, this.name is the same as author.name.



With explicit binding, you can remove the method from an object and use call to indicate the context. In this case, when logName.call is invoked, this.name refers to author.name.



The first parameter of call is the context. Here, we're saying that the logName function should be run in the context of the author object. After that, you can pass in parameters. We're passing in the first element of an array. The passed parameters are then available, as they normally would be, inside the function, and the properties of the author object are also available, by using the this keyword.



.apply allows you to bind a function to an object and pass an array to the function, as shown here. We create a function called logName that takes three parameters. We then create an object called author that has two properties and we define an array called favoriteFoods. By using .apply, we run logName in the context of the author object and we pass it the favoriteFoods array, which gets expanded for each of the three parameters of the function.

Explicit Binding with `.bind`

- Works the same as `.call`, but returns `new` function rather than immediately invoking the function

```
logName = function(food) {
  console.log(this.fname +" " + this.lname +
            "\'s Favorite Food was " + food);
};
var person = {
  fname: "George",
  lname: "Washington"
};
var logMe = logName.bind(person,"Tacos");

logMe();
```

- http://jsbin.com/xikuzog/edit?js,console

`new` Binding

- When a function is invoked with the `new` keyword, this keyword inside the object is bound to the `new` object

```
var City = function (lat,long,state,pop) {
  this.lat = lat;
  this.long = long;
  this.state = state;
  this.pop = pop;
};
var sacramento = new
City(38.58,121.49,"CA",480000);
console.log (sacramento.state);
```

`window` Binding

- What happens when no object is specified or implied
- `this` defaults to the `window` object

```
var logName = function() {
  console.log(this.name);
}
var author = {
  name: 'Chris',
  homeTown: 'Detroit"
}
logName(author); //undefined(error in 'strict'
mode)
window.author = "Harry";
logName(author); // "Harry"
```

reduce()

- `Array.reduce()`
  - Applies a function against an accumulator and each value of the array to reduce it to a single value.
  - Takes two parameters
    - A callback function (reducer)
    - An initial value (array)

```
var total = [0, 1, 2, 3].reduce(function(a, b) {
  return a + b;
});
// total == 6
```

- http://jsbin.com/regosow/edit?js,console

160

Promises

- First class representation of a value that may be made asynchronously and be available in the future

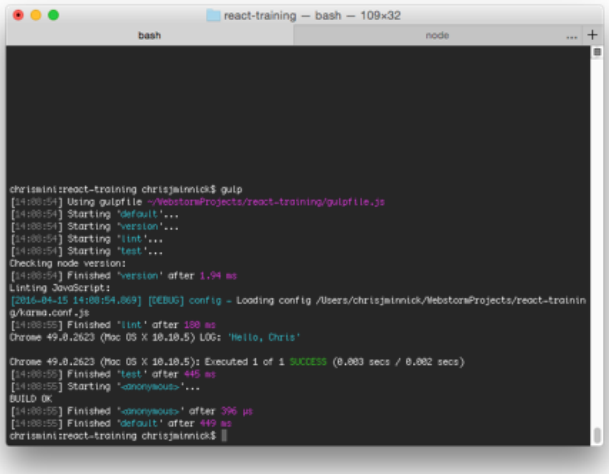
```
function msgAfterTimeout (msg, who, timeout) {
  return new Promise((resolve, reject) => {
    setTimeout(() =>
    resolve(`${msg} Hello ${who}!`), timeout)
  })
}
msgAfterTimeout("", "Foo", 100).then((msg) =>
  msgAfterTimeout(msg, "Bar", 200)
).then((msg) => {
  console.log(`done after 300ms:${msg}`)
})
```

- http://jsbin.com/dozimot/edit?js,console

Babel

- Babel converts ES6 code into its equivalent ES5 so that it will run in today's browsers

Lab 01: Transpiling with Babel



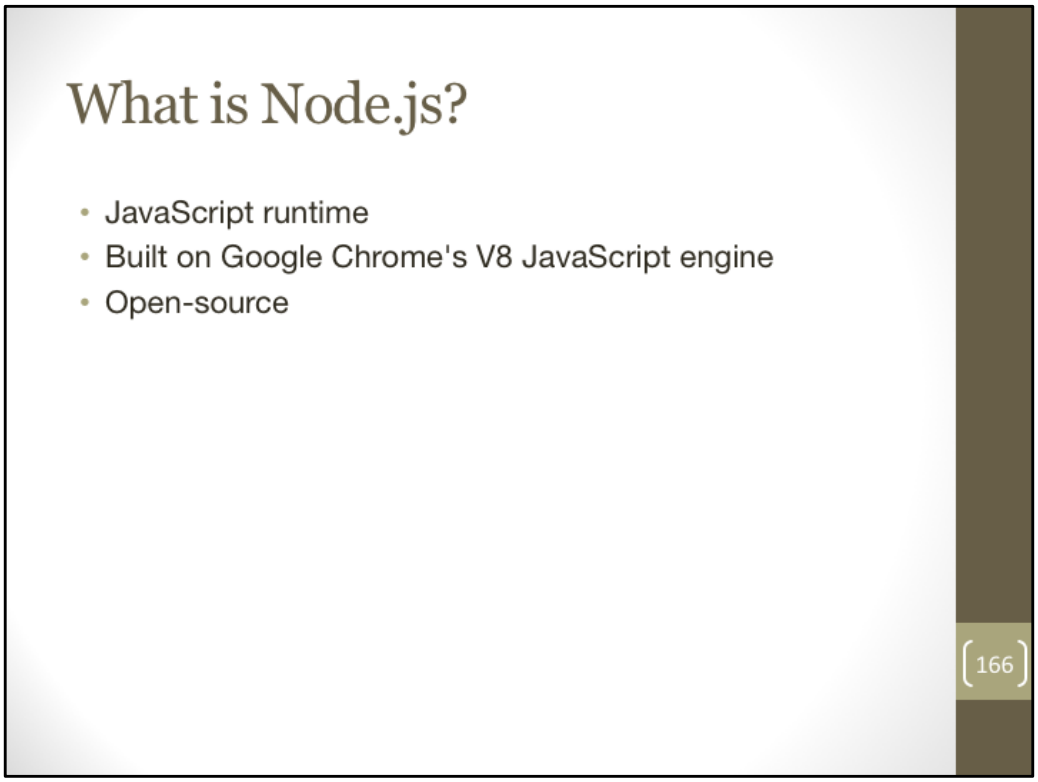
Lab 02: Converting to ES6

```
export function greet(name) {
    return "Hello, " + name;
};

import * as sayHello from './sayHello.js';
```

Intro to Node.js

Objectives:

- Understand how Node.js works
- Create and use modules
- Understand events and streams
- Creating servers and interacting with web APIs



Node started its life as a way to run JavaScript on the server. However, it's become an essential tool for managing modules, running tools, and automating front-end development.

Node.js allows you to run JavaScript from the command line.

What is it good for?

- Web servers
- Networking tools
- Client-side tools

History of Node.js

- 2009: Created by Ryan Dahl @ Joyent and introduced at JSConf
- 2010: Node's package manager, npm, was introduced
- 2012: Dahl stepped aside as project lead
- 2014: Node.js was forked to create io.js as an open governance alternative to Node.js
- 2015: Node.js Foundation was created and Node.js and io.js were merged back together.

Who Uses Node?

- Netflix
- NYTimes
- Paypal
- Uber
- Medium
- LinkedIn
- Walmart
- CBS
- eBay
- Pinterest
- HP
- Groupon
- Wall Street Journal
- Lowe's
- Capital One
- Microsoft
- Intel
- Amazon
- Github
- Google
- YouTube
- IBM
- etc etc etc

How Does Node.js Work?

- Event-driven
  - Listens for events and does things in response.
- Doesn't wait
  - If there's no event, Node is "sleeping."
  - If there's nothing left to do, the Node process exits
- Asynchronous code
  - "When you're done doing that, do this."

How is Node.js Different?

- Node Applications are Single-threaded
  - Can handle thousands of concurrent connections with minimal overhead
  - Node itself is written in C++, and is multi-threaded
- No buffering
  - Node applications never buffer data.
  - Data is output in chunks.
- Non-blocking
  - Node doesn't wait for data to be returned from APIs.
  - While it's possible to write blocking code in Node.js, it's discouraged.

What is Node.js Made Of?

1. libuv
   - evented I/O library
2. V8
   - Google's JavaScript engine
3. Custom JS & C++ code

Blocking vs. Non-blocking

Blocking

- In blocking code, the next statement doesn't execute until the previous one finishes.

```
var contents =
fs.readFileSync('file.txt','
utf8');
/* this line is reached when
the file is completely read
*/
console.log(contents);
```

Non-blocking

- In non-blocking code, execution doesn't wait.

```
fs.readFile('file.txt',
'utf8', function(err, data)
{
  /* this line is reached
     later when the results
     are in */
  console.log(data)
});
/* readFile returns
immediately and this line
is reached right away */
```

Blocking code

Non-Blocking code

Your First Node Program

Create a new file with a .js extension

- `vi my-first-program.js`

Write a statement in the new file to write to the console

- `console.log("Hello Node");`

Save and run the program from the command prompt

- `node my-first-program.js`

Running a Node.js program

- Type node followed by the program name.

```
node program.js
```

Lab 01: Intro to Node

- In this lab, you'll install Node.js, learn to use the interactive shell (aka REPL), write a node application, and learn about using npm for package management.

Lab 02: First Look at Async Code

```
console.log('starting...');

process.nextTick(function(){
    for (var i=1; i<11; i++){
        console.log(i);
    }
});

console.log('Done!');
```

NODE MODULES

Modules Overview

- Node uses CommonJS for loading modules
- Original name was ServerJS
- Provides a way to import dependencies in JavaScript when used outside the browser (such as server side or in desktop applications)
- `module.exports`
  - Encapsulates code into a module
- `require`
  - Imports a module into JavaScript code

CommonJS Example

foo.js

```
var circle = require('./circle.js');
console.log('Area of circle: ' + circle.area(4));
```

circle.js

```
exports.area = function(r){
  return Math.PI * r * r;
}
```

Using Modules

- Modules explicitly make their functionality available by exporting their functionality.
- Use `require()` to load a module and assign it to a variable.
  - `var foo = require("foo");`
- Modules can export variables, functions, or objects.
- If a module is a constructor function or class, import the module to a variable starting with a capital letter.
  - `var Bar = require('bar');`
  - `var myBar = new Bar();`
- If a module exports variables and functions, assign it to a camelCased variable.
  - `var foo = require('foo');`

Sources of Modules

- Node's core library
- Your project's files
- npm

Node's Core Modules

- Certain modules are built into node and installed at the same time as Node.js.
- These modules can automatically be used in any application.
- Examples:
  - assert – used for writing util tests
  - crypto – provides cryptographic functionality
  - fs – used for performing file system operations
  - http – http server and client functionality
  - os – operating system-related utility methods

The http Module

- `var http = require('http');`

- Core module for working with HTTP protocol
- `http.request()`
  - method for performing HTTP requests
- `http.get()`
  - shortcut for GET requests
- `http.Server`
  - class used for creating an HTTP server
- `http.createServer()`
  - method for creating an instance of `http.Server`

Lab 03: A Simple Node.js Server

- In this lab, you'll use Node.js and the http module to create a simple web server that listens for connections and responds with a simple Hello World message.

The `fs` module

- `var fs = require('fs');`
  - Performs filesystem operations
  - Contains blocking (a.k.a. synchronous) as well as non-blocking (asynchronous) methods
- Async example:

```
fs.readFile('hello.txt', function(err){
  if (err) throw err;
  console.log('success');
}
```

- Sync example:

```
fs.readFileSync('hello.txt');
console.log('success');
```

Buffer Objects

- Makes it possible to read and manipulate streams of binary data.
- Converts Buffer objects to strings like this:
  - `var str = buf.toString();`
- `fs` module returns a buffer, unless you pass 'utf8' as the 2nd argument

Modularizing Your Code

- Create a separate `.js` file
- Default export
  - Uses `module.exports = function(){...}` to export a single function or variable.
- Named export
  - Used for exporting multiple values from a file.
    - `module.exports.pi = 3.14;`
    - `module.exports.c = 299792458;`

Returning Values from Modules

- To return a value from a module, take a callback function as an argument and call that function with `err` or return data

```
module.exports =
function(firstNum,secondNum,callback){
  var sum = firstNum + secondNum;
  if (isNaN(sum)){
    callback("error: sum not a number");
  }
  callback(null,sum);
});
```

Using a Local Module

- Use local modules the same way as you use core modules
- Local modules must include the path to the file.
- `.js` is implied and can be left off

```
var sumModule = require('./sumModule');
sumModule(1,3,function(err,data){
  if(err) throw err;
  console.log("The sum is: " + data);
}
```

Lab 04: Creating Modules

- In this lab, you'll write your first node module and then use that module in a program.

EVENTS AND STREAMS

Non-blocking with Events

- Events can be used to write non-blocking code.
- Publish, Subscribe Design Pattern
  - An object can publish events
  - Other objects can subscribe to events and be notified when they happen.

Events

- Many objects emit events
- All objects that emit events are instances of process.EventEmitter

EventEmitter

- `EventEmitter` is a class that lets you listen for events and assign actions when those events occur
- Very loose way of coupling different parts of your application

```
var EventEmitter = require("events").EventEmitter;

var ee = new EventEmitter();
ee.on("someEvent", function () {
    console.log("event has occured");
});

ee.emit("someEvent");
```

EventEmitter Patterns – Pattern 1

- Extend EventEmitter
  - Create a module that inherits EventEmitter
  - Emit events from the module
  - Listen for events

Code Walkthrough: Extending EventEmitter

```
var util = require('util');
var EventEmitter = require('events').EventEmitter;

function Resource (c) {

    var maxEvents = c;
    var self = this;

    process.nextTick(function() {
        var count = 0;
        self.emit('start');
        var t = setInterval(function() {
            self.emit('data', ++count);
            if (count === maxEvents) {
                self.emit('end', count);
                clearInterval(t);
            }
        },10);
    });
};

util.inherits(Resource,EventEmitter);

module.exports = Resource;
```

Code Walkthrough: Extending EventEmitter (cont)

- var Resource = require('./resource');

```
var r = new Resource(7);

r.on('start', function() {
    console.log("I've started!");
});

r.on('data', function(d) {
    console.log(`Got data: ${d}`);
});

r.on('end', function(t) {
    console.log(`Finished. Got ${t} events.`);
});
```

EventEmitter Patterns - Pattern 2

- Return events from a function
  - Run a function that emits events
  - Listen for events the function can emit

Code Walkthrough: Emit events from a function

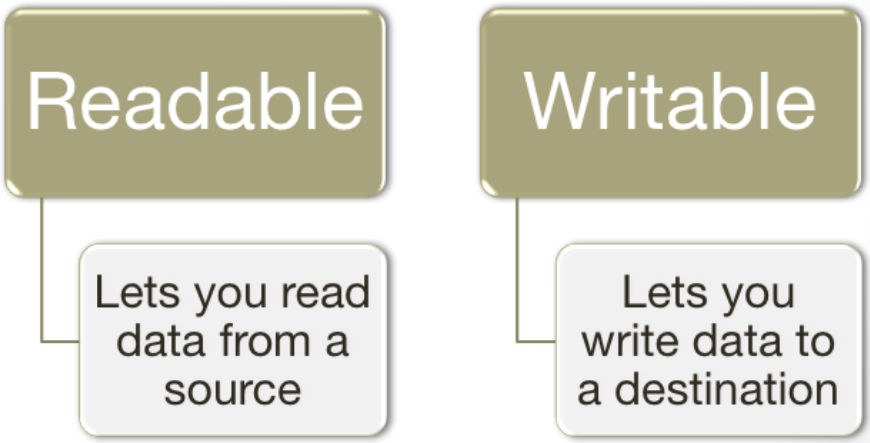
```
var EventEmitter = require('events').EventEmitter;

var getResource = function(c) {
    var e = new EventEmitter();
    process.nextTick(function() {
        var count = 0;
        e.emit('start');
        var t = setInterval(function() {
            e.emit('data', ++count);
            if (count === c) {
                e.emit('end', count);
                clearInterval(t);
            }
        },10);
    });
    return(e);
};

var r = getResource(5);

r.on('start', function() {
    console.log("I've started!");
});

r.on('data', function(d) {
    console.log(`Got data: ${d}`);
});

r.on('end', function(t) {
    console.log(`Finished. Got ${t} events.`);
});
```

Node Stream Objects

- Unix Pipes for moving data around
- Objects that emit events
- Stream events
  - data
    - Emitted when a chunk of data is available
    - Chunk size depends on the data source
  - error
    - Emitted on error
  - end
    - Emitted when there's no more data to be consumed

```
response.on("data"), function(data){
  /* do something with data */
}
```

Types of Streams



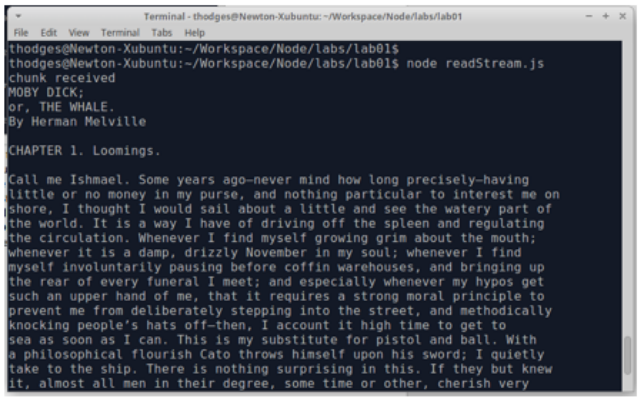
Using Readable Streams

```
var fs = require('fs');
var readableStream =
fs.createReadStream('file.txt');
var data = '';

readableStream.on('data', function(chunk) {
    data+=chunk;
});

readableStream.on('end', function() {
    console.log(data);
});
```

Using Writable Streams

```
var fs = require('fs');
var readableStream =
fs.createReadStream('file1.txt');
var writableStream =
fs.createWriteStream('file2.txt');

readableStream.setEncoding('utf8');

readableStream.on('data', function(chunk) {
    writableStream.write(chunk);
});
```

Lab 05: Working with Streams



Lab 06: Piping Between Streams

The `process` Object

- The process object is a global that provides information about and control over the current Node.js process

Lab 07: The process object

Command Line Arguments

- You can pass command-line arguments to Node when you run a program
  - `node my-first-program.js 1 2 3`
- Command line arguments can be accessed using the process object
- Arguments are stored in the `process.argv` array
- First element of process.argv is 'node'
- Second element of proess.argv is the path to your program
- Your arguments come after that
- Example:
  - `[ 'node', '/path/to/my-first-program.js', '1', '2', '3' ]`

Command Line Arguments (cont)

- All elements of `process.argv` are strings
- "Number strings" must be converted to number data type prior to using for math
  - Two ways
    - `Number(process.argv[1])`
    - `+process.argv[1]`
- To loop through arguments in `process.argv`

```
for (i=2; i<process.argv.length; i++) {
  console.log('argument #'+ i + ': ' +
process.argv[i]);
}
```

PROMISES

What Are Promises?

- An abstraction for asynchronous programming
- Alternative to callbacks
- A promise represents the result of an async operation
- Is in one of three states
  - pending – the initial state of a promise
  - fulfilled – represents a successful operation
  - rejected – represents a failed operation

Promises vs. Event Listeners

- Event listeners are useful for things that can happen multiple times to a single object.
- A promise can only succeed or fail once.
- If a promise has succeeded or failed, you can react to it at any time.
  - `readJSON(filename).then(success,failure);`

Why Use Promises?

- Chain them together to transform values or run additional async actions
- Cleaner code
  - Avoid problems associated with multiple callbacks
    - Callback Hell
    - Christmas Tree
    - Tower of Babel
    - Etc

Demo: Callback vs. Promise

- Callback
  - ```
    fs.readFile('text.txt', function(err, file){
        if (err){
            //handle error
        } else {
            console.log(file.toString());
        }
    });
    ```
- Promise
  - ```
    readText('text.txt')
        .then(function(data) {
                console.log(data.toString());
            }, console.error
        );
    ```

Using Promises

```
const fs = require('fs');

function readFileAsync (file, encoding) {
    return new Promise(function (resolve, reject) {
        fs.readFile(file, encoding, function (err, data) {
            if (err) return reject(err);
            resolve(data);
        })
    })
}

readFileAsync('myfile.txt')
       .then(console.log, console.error);
```

Promises with Bluebird

```
var Promise = require('bluebird');
var fs = Promise.promisifyAll(require('fs'));

fs.readFileAsync("name", "utf8").then(function(data) {
    console.log(data.toString());
});
```

Lab 08: Promises

NODE ON THE WEB

Making an HTTP Request

- `http.get()`
  - First argument is the URL you want to GET
  - Second argument is a callback with the following signature:

```
function callback(response){ /* ... */ }
```

- The `response` object is a Node Stream

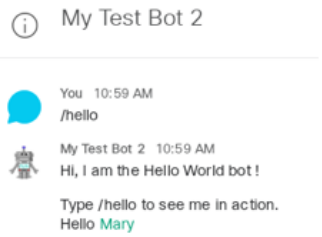
```
http.get('http://example.com/users/',function(response){
  response.on("data", function(data) { /* ... */ }
}
```

Lab 09: Getting Data with HTTP

- In this module, you'll use the http module to do an HTTP get request to a server. You'll then use the response stream to log each chunk of data from the server to the console.

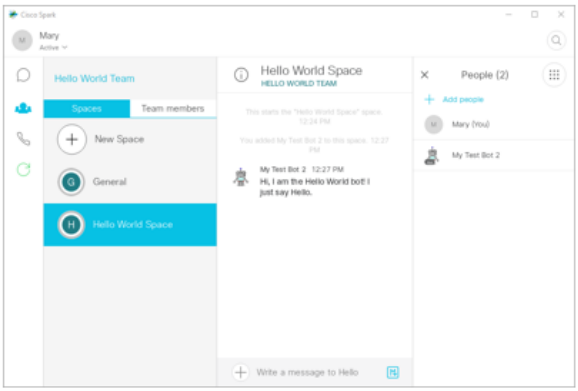
Lab 10: Making a Bot

- In this lab, you'll create a bot for Cisco Spark using sample code from CiscoDevNet. You'll use ngrok to allow the bot to communicate with Cisco Spark, and you'll configure Webhooks to notify your bot of events that occur on Spark.



Lab 11: Making a Hello World Bot

- In this lab, you'll program your first bot, which will just say hello when it's started up.



TESTING NODE.JS

The assert module

- Contains assertions
  - equality
  - throws an exception
  - test for truthiness
  - test whether error parameter was passed to callback
  - more
- Types of equality
  - assert.equal(): shallow (==)
  - assert.strictEqual(); strict (===)
  - Date equality
  - Other object equality: equal if they have the same number of owned properties, equivalent values for each key, and the same prototype.

Using assert

```
var sumModule = require('./sumModule');
var assert = require('assert');

sumModule(1,2,function(err,data){
    assert.equal(data,3);
});

sumModule(1,'egg',function(err,data){
    assert.ifError(err);
}
```

Lab 12: Testing with assert

- Run the tests in sumModuleTests.js
- Assert that the module will return an error if both arguments passed to it aren't positive.
- Run the test to confirm that it fails.
- Update sumModule to make the test pass.

Mocha and should.js

- Install Mocha globally
  - npm install –g mocha
- Install should locally
  - npm install --save should

Testing with Mocha and should.js

- Testing Async Code

```
var should = require('should');
var sumModule = require('./sumModule');

describe('sumModule', function() {
  it('should add numbers together', function(done){
    sumModule.sum(2,2,function(err,result) {
      result.should.equal(4);
      done();
    });
  });
}
```

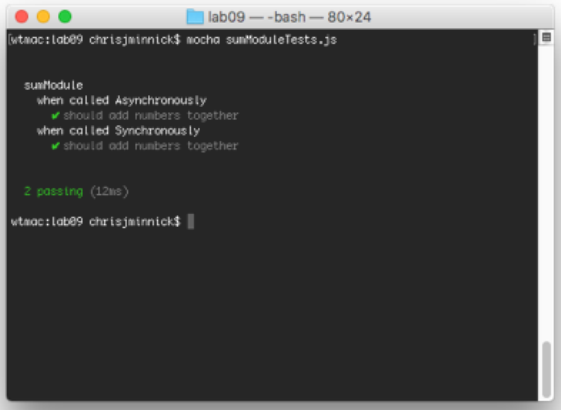
Testing with Mocha and should.js

- Testing Sync Code

```
var should = require('should');
var sumModule = require('./sumModule');

describe('sumModule', function() {
  it('should add numbers together', function(done){
    sumModule.sumSync(2,2).should.equal(4);
    });
  });
}
```

Running Tests with Mocha

```
mocha sumModuleTest.js
```



USING EXPRESS

What is Express?

- Web application framework for Node.js.

Getting Started with Express

1. Install Express
   - npm install express
2. Include Express in a file
3. Create an app object
   - var app = express();
4. Use app object methods
   - Routing
   - Configure middleware
   - Render HTML views
   - Register a template engine

Express Hello World Server

```
#app.js
const express = require('express');
const app = express();

app.get('/', function(req, res) {
  res.send('Hello World!');
});

app.listen(3000, function() {
  console.log('Listening on Port 3000');
});
```

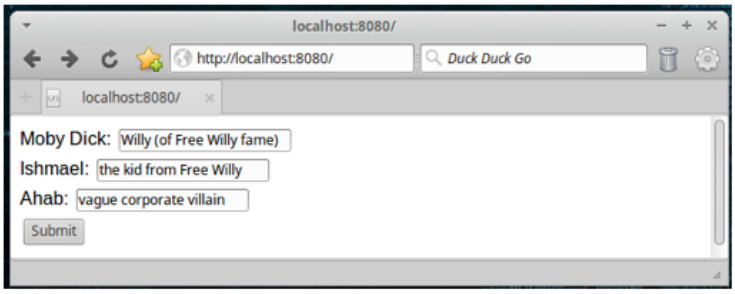
Routing with Express

- Routing determines how an application responds to a client request to an endpoint.
- Endpoint
  - URI (path)
  - HTTP Request method
- Route definition structure
  - app.METHOD(PATH, HANDLER);
- METHOD
  - get, post, put, delete
- PATH
  - a path on the sever
- HANDLER
  - a function to execute when the route is matched.

Serve Static Content with Express

- express.static
  - Built-in middleware function
  - Can be used to serve images, CSS, JavaScript, HTML
- app.use(express.static('public'))
  - Allows loading of files in the 'public' directory.
- Virtual path prefix
  - app.use('/static', express.static('public'))
    - Allows loading of assets from public directory using /static path
    - http://localhost:3000/static/images/tacos.gif

Lab 13: Make an Express Server



Database Access with Node.js

- Node can be used to access many different types of databases.
- The most popular DB to use with Node is MongoDB

Install MongoDB

- Download mongoDB from mongodb.org
- Install the correct version for your operating system.
- Create the data directory:
  - C:\data\db (Windows)
  - \data\db (Mac)
- Make sure that the system can write to the datadb directory.

mongod and mongo

- mongod is the server
- mongo is the command line

Test Mongo Command Line

- In your terminal or command prompt, go to where you installed Mongo (or add the mongo directory to your path) and type `mongo` to start the command line interface.
- `>show dbs`
  - Will list your databases. You probably don't have any right now.

Use MongoDB in Node

- Install the MongoDB driver
  - npm install --save mongodb
- Review the driver documentation
  - http://npmjs.com/package/mongodb

Insert Data

```
//app.js
var express = require('express');
var mongodb = require('mongodb').MongoClient;
var app = express();
var books = [{title:'East of Eden'},
             {title:'War and Peace'}]; //add more
app.get('/addData', function(req,res){
  var url = 'mongodb://localhost:27017/booksApp';
  mongodb.connect(url,function(err,db){
       //add code here (see next slide)
  });
});
app.listen(5000, function(err){
  console.log('running server');
});
```

Adding Data (continued)

```
mongodb.connect(url, function(err,db) {
  var collection = db.collection('books');
  collection.insertMany(books,
    function(err,results){
      res.send(results);
      db.close();
    });
});
```

Test it out

- Start the mongo server
  - `mongod`
- Run app.js
  - `node app.js`
- Go to http://localhost:5000/addData
  - You should see the inserted data
- Open the command line in a new console window
  - `mongo booksApp`
  - `> show collections`
  - `> db.books.find()`

Retrieve from MongoDB

Create a new route in app.js

```
app.get('/viewAll', function(req,res){
  var url = 'mongodb://localhost:27017/booksApp';
  mongodb.connect(url,function(err,db){
    var collection = db.collection("books");
    collection.find({}).toArray(
      function(err, results) {
        res.send(results);
      }
  );
});
});
```

Testing the `/viewAll` route

- Stop and start app.js
- Go to http://localhost:5000/viewAll
- See the array!

Challenges

- Select just one book from the database.
- Display the book information in an HTML template
- Create a form to insert new records
- Make a route that displays a list of books, with a link next to each to a separate route that displays just one book.