Core JavaScript

Salesforce.com

Peter J. Jones
✉ pjones@devalot.com
🐦 @contextualdev
http://devalot.com

Ryan Morris
✉ mr.morris@gmail.com
🐦 @mrmorris
https://github.com/mrmorris

April 27, 2016

DevelopIntelligence

Contents

Course Requirements

Please ensure that the following software applications are installed on the computer you'll be using for this course:

- Node.js
- Google Chrome

You will also need a text editor or IDE installed. If you don't have a preferred text editor you may be interested in one of the following:

- Atom
- Sublime Text

Finally, ensure that your network/firewall allows you to access the following web sites:

- Devalot.com

  Handouts, slides, and course source code.

- npmjs.com

  For installing Node.js packages.

- GitHub.com

  Class-specific updates to the course source code.

Chapter 1

JavaScript the Language

1.1 Introduction to This Course

The source code for this course can be found at the following URL:

https://github.com/devalot/corejs

1.2 Introduction to JavaScript

1.2.1 Approaching JavaScript

- JavaScript might be an object-oriented language with "Java" in the title, but it's not Java.
- I find that it's best to approach JavaScript as a functional (yet imperative) language with some object-oriented features.

1.2.2 A Little Bit About JavaScript

- Standardized as ECMAScript
  - 5th Edition, 2009 (widely supported)
  - 6th Edition, 2015 (not so much)
- Special-purpose language
- Dynamically typed (with weak typing)
- Interpreted and single threaded

- Prototype-base inheritance (vs. class-based)
- Nothing really to do with Java
- Weird but fun

1.2.3 Not a General Purpose Language

- JavaScript is **not** a general-purpose language
- There are no functions for reading from or writing to files
- I/O is heavily restricted

1.2.4 But, It's Not Just for the Browser

- Outside of the browser there are libraries that help make JavaScript act like a general purpose language.
- Tools such as Node.js add missing features to JS
- Weigh the pros and cons of using JS outside the browser

1.2.5 Why JavaScript?

- It's the language of the web
- Runs in the browser, options to run on server
- Easy to learn partially
- Harder to learn completely

1.2.6 JavaScript Syntax Basics

- Part of the "C" family of languages
- Whitespace is insignificant (including indentation)
- Blocks of code are wrapped with curly braces: `{ ... }`
- Expressions are terminated by a semicolon: `;`
- Lexical Structure and Keywords

1.2.7 A Note About Semicolons

- Semicolons are used to terminate expressions.
- They are optional in JavaScript.
- Due to the minification process and other subtle features of the language, you should always use semicolons.
- When in doubt, use a semicolon.

1.2.8 The Browser’s JavaScript Console

- Open your browser’s debugging console:
  - Command-Option-J on a Mac
  - F12 on Windows and Linux
- Enter the following JavaScript:

  ```
  console.log("Hello World");
  ```

1.2.9 Browser Debugging

- The browser’s “console” is a line interpreter (REPL)
- All major browsers are converging to the same API for console debugging
- Can use it to set breakpoints
- Lets you see scoped variables and context
- Can set a conditional breakpoint
- `console.log` is equivalent to `printf`

1.2.10 JavaScript Types

- Primitive Types:

  ```
  "Hello World"; // Strings
  42;            // Numbers
  true && false; // Boolean
  null;          // No value
  undefined;     // Unset
  ```

- Objects (arrays, functions, etc.)

1.2.11 Variables in JavaScript

```
var x;          // undefined
var y = "Foo";  // String
var z = 5;      // Number
```

1.2.12 Variable Naming Conventions

- Use camelCase: `userName`, `partsPerMillion`
- Allowed: letters, numbers, underscore, and `$`
- Don't use JavaScript keywords as variable names
- Always start with a lowercase letter

(All identifiers can be made up of valid Unicode characters. Don't go crazy, not all browsers support this. Stick to UTF-8 identifiers.)

1.2.13 Undefined and Null

- There are two special values: `null` and `undefined`
- Little difference between the two
- Variables declared without a value will start with `undefined`
- Can compare to `null` to see if a variable has a value:

```
null == undefined;   // true
null == 0;           // false
```

1.2.14 Numbers

- All numbers are 64bit floating point
- Integer and decimal (`9` and `9.8` use the same type)
- Keep an eye on number precision:

```
0.1 + 0.2 == 0.3; // false
```

- Special numbers: `NaN` and `Infinity`

```
NaN == NaN; // false
1 / 0;      // Infinity
```

1.2.15 How Do You Deal with Numeric Accuracy?

- Use a special data type like Big Decimal.
- Round to a fixed decimal place with `num.toFixed(2);`
- Only use integers (e.g., for money, represent as cents)

1.2.16 Strings

- Use double or single quotes (no difference between them):

  ```
  "Hello" // Same as...
  'Hello'
  ```

- Typical backslash characters works (e.g., `\n` and `\t`) in both types of strings.

- Operators:

  ```
  "Hello" + " World";  // "Hello World"
  "Lucky " + 21;       // "Lucky 21"
  "Lucky " - 21;       // NaN
  "1" - 1              // 0
  ```

1.2.17 Type Coercion

- JavaScript is loosely typed

- Implicit conversion between types as needed

- Usually in unexpected ways:

  ```
  8 * null; // 0
  "5" - 1;  // 4
  "5" + 1;  // "51"
  ```

1.2.18 Objects

- Built up from the core types

- A dynamic collection of **properties**:

  ```
  var box = {
    color: "tan",
    height: 12
  };

  box.color;             // Getter method
  box.color = "red";     // Setter method

  var x = "color";
  box[x];          // "red"
  box[x] = "blue"; // Alternative syntax
  ```

1.2.19 Object Basics

- Everything is an object (almost)
- Primitive types have object wrappers (except `null` and `undefined`)
- They remain primitive until used as objects, for performance reasons
- An object is a dynamic collection of properties
- Properties can be functions

1.2.20 The Array Object

- Arrays are objects that behave like traditional arrays
- Use arrays when order of the data should be sequential

1.2.21 Creating Arrays

```
// Array literal:
var myArray = [1, 2, 3];

// Using the constructor function:
var myArray = new Array(1, 2, 3);
```

1.2.22 Recap: Basic Data Types

- There are five primitive types:

  1. String
  2. Number
  3. Boolean
  4. null
  5. undefined

- And then there are objects
- Declare variables with `var`
- Types are automatically coerced when needed
- Everything can be represented as an object

1.2.23 JavaScript Comments

- Single-line comments:

  ```
  // Starts with two slashes, runs to end of line.
  ```

- Multiple-line comments:

  ```
  /* Begins with a slash and asterisk.

  Also a comment.

  Ends with a asterisk slash. */
  ```

1.2.24 Exercise: Using Primitive Types

1. Start the Node.js server:

   ```
   node bin/server.js
   ```

2. Open the following file:

   ```
   www/primitives/primitives.js
   ```

3. Complete the exercise.
4. Run the tests by opening http://localhost:3000/primitives/

1.2.25 Operators

- Arithmetic: `+ - * / %`
- Shortcut: `+= -= *= /= %=`
- Increment: `++x x++`
- Decrement: `--x x--`
- Bitwise: `& | ^ >> <<`
- Comparison: `> >= < <=`
- Logic: `! && ||`

1.2.26 Sloppy Equality

- The traditional equality operators in JS are sloppy

- That is, they do implicit type conversion

```
"1" == 1;   // true
[3] == "3"; // true

0 != "0";  // false
0 != "";   // false
```

1.2.27 Strict Equality

More traditional equality checking can be done with the `===` operator:

```
"1" === 1;  // false
0 === "";   // false

"1" !== 1;  // true
[0] !== ""; // true
```

(This operator first appeared in ECMAScript Edition 3, circa 1999.)

1.2.28 Boolean Operators: &&

`a && b` short circuit like:

```
if (a) {
  return b;
} else {
  return a;
}
```

1.2.29 Boolean Operators: ||

`a || b` short circuit like:

```
if (a) {
  return a;
} else {
  return b;
}
```

1.2.30 Boolean Operators: !

Boolean negation: `!`:

```
var x = false;
var y = !x; // y is true
```

Double negation: `!!`:

```
var n = 1;
var y = !!n; // y is true
```

1.2.31 Exercise: Boolean Operators

- Experiment with `&&`:

  ```
  false && console.log("Yep");
  true  && console.log("Yep");
  ```

- Experiment with `||`:

  ```
  false || console.log("Yep");
  true  || console.log("Yep");
  ```

1.2.32 The Ternary Conditional Operator

- JavaScript supports a ternary conditional operator:

  ```
  condition ? then : else;
  ```

- Example:

  ```
  var isWarm; // Is set to something unknown.
  var shirt = isWarm ? "t-shirt" : "sweater";
  ```

1.2.33 What Is `true` and What Is `false`?

- Things that are `false`:

  ```
  false;
  null;
  undefined;
  ""; // The empty string
  0;
  NaN;
  ```

- Everything else is `true`, including:

  ```
  "0";       // String
  "false";   // String
  [];        // Empty array
  {};        // Empty object
  Infinity;  // Yep, it's true
  ```

1.2.34 Statements and Expressions

- Expressions compute and returns values
- Statements are made up of expressions but have no value
- A program is a list of statements

1.2.35 Declaring and Initializing Variables

- Declare variables to make them local:

  ```
  var x;
  ```

- You can initialize them at the same time:

  ```
  var n = 1;

  var x, y=1, z;
  ```

- If you don't declare a variable with `var`, the first time you assign to an undefined identifier it will become a global variable.
- If you don't assign a value to a new variable it will be `undefined`

1.2.36 Constants

- Not frequently used
- Same rules as apply to variables, but keyword `const` is used instead of `var`
- They **are** scoped

```
const TIMEOUT = 5;

TIMEOUT = 10;

TIMEOUT === 5; // true
```

1.2.37 Conditional Statements

```
if (expression) { then_part; }

if (expression) {
  then_part;
} else {
  else_part;
}
```

1.2.38 Chaining Conditionals

```
if (expression) {
  then_part;
} else if (expression2) {
  second_then_part;
} else {
  else_part;
}
```

1.2.39 Switch Statements

Cleaner conditional (using strict equality checking):

```
switch (expression) {
  case val1:
    then_part;
    break;

  case val2:
    then_part;
    break;

  default:
    else_part;
    break;
}
```

Don't forget that `break;` statement!

1.2.40 The Major Looping Statements

- Traditional `for`:

```
for (var i=0; i!=n; ++i) { /* body */ }
```

- Traditional `while`:

```
while (condition) { /* body */ }
```

- Traditional `do ... while`:

```
do { /* block */ } while (condition)
```

- Object Property Version of `for`:

```
for (var prop in object) { /* body */ }
```

1.2.41 Traditional for Loops

- Just like in C:

```
for (var i=0; i<10; ++i) {
  // executes 10 times.
}
```

- Loops can be labeled and exited with `break`.
- Use `continue` to skip to the next iteration of the loop.

1.2.42 Traditional while Loops

```
var i=0;

while (i<10) {
  ++i;
}
```

1.2.43 Flipped while Loops

```
var i=0;

do {
  ++i;
} while (i<10);
```

1.2.44 Controlling a Loop

- Loops can be labeled and exited with `break`.
- Use `continue` to skip to the next iteration of the loop.

```
// Rarely used labels.  Try to avoid.
outer:

for (;;) {

  inner:

  for (;;) {
    break outer;
  }
}
```

1.2.45 Control Structures Recap

- Conditionals like `if` and `if ... else`
- `switch` statements
- Looping with `for` and `while`

1.2.46 Exercise: Experiment with Control Flow

1. Open the following file:

   ```
   www/control/control.js
   ```

2. Complete the exercise.
3. Run the tests by opening: http://localhost:3000/control/

1.3 Debugging in the Browser

1.3.1 Introduction to Debugging

- All modern browsers have built-in JavaScript debuggers
- We've been using the debugging console the entire time!

1.3.2 Browser Debugging with the Console

- The `console` object:
  - Typically on `window` (doesn't always exist)
  - Methods
    - `log`, `info`, `warn`, and `error`
    - `table(object)`
    - `group(name)` and `groupEnd()`
    - `assert(boolean, message)`

1.3.3 Accessing the Debugger

- In the browser's debugging window, choose **Scripts**

- You should be able to see JavaScript files used for the current site

1.3.4 Setting Breakpoints

- Debugger with breakpoints

- http://jsfiddle.net/mrmorris/X76Gq/

1.3.5 Stepping Through Code

- After setting breakpoints, you can reload the page

- Once the debugger stops on a breakpoint you can step through the code using the buttons in the debugger

1.4 Functions

1.4.1 Introduction to Functions

- "The best part of JavaScript"

- Functions are used to implement **many** features in JS:
  - Classes, constructors, and methods
  - Modules, namespaces, and closures
  - And a whole bunch of other stuff

1.4.2 Defining a Function

- Function statements (named functions)
- Function expression (anonymous functions)

1.4.3 Function Definition (Statement)

```
function add(a, b) {
  return a + b;
}

var result = add(1, 2); // 3
```

- This syntax is know as a **function definition statement**. It is only allowed where statements are allowed. This is when the distinction between statements and expressions becomes important.

- Most of the time you should use the expression form of function definition.

1.4.4 Function Definition (Expression)

```
var add = function(a, b) {
  return a + b;
};

var result = add(1, 2); // 3
```

- Function is callable through a variable
- Name after `function` is optional
- We'll see it used later

1.4.5 Function Invocation

- Parentheses are mandatory in JavaScript for function invocation

- Any number of arguments can be passed, regardless of the number defined

- Extra arguments won't be bound to a name

- Missing arguments will be `undefined`

1.4.6 Function Invocation (Example)

```
var add = function(a, b) {
  return a + b;
};

add(1)       // a is 1, b is undefined
add(1, 2)    // a is 1, b is 2
add(1, 2, 3) // No name for 3.
```

1.4.7 Function Invocation and Parentheses

```
var add = function(a, b) {return a + b;};

var x = add;        // x is now a function object
x(1, 2);            // Same as add(1, 2);

var y = add(1, 2); // y is 3
```

1.4.8 Functions that Return a Value

In order for a function to return a value to its caller, it must use the `return` keyword.

```
var add = function(a, b) {
  // WRONG!  Computes a sum then throws it away.
  a + b;
};
```

vs.

```
var add = function(a, b) {
  return a + b; // CORRECT!
};
```

1.4.9 Be Careful with Your Line Breaks

```
return
  x;
```

becomes:

```
return;
  x;
```

1.4.10 Special Function Variables

Functions have access to two special variables:

- `arguments`: An object that encapsulates all function arguments
- `this`: The object the function was called through

1.4.11 Rules for Using the `arguments` Variable

- Access all arguments, even unnamed ones
- Array-like, but not an actual array
- Only has `length` property
- Allows actual argument mutation
- Should be treated as read-only
- To treat like an array, convert it to one

```
var arr = Array.prototype.slice.call(arguments);
```

1.4.12 Built-in Functions (Types and Conversions)

`isNaN(num)`: Safely test if `num` is `NaN`
`isFinite(num)`: Test if `num` is **not** `NaN` or `Infinity`
`parseInt(str)`: Convert a string to a number (integer)
`parseFloat(str)`: Convert a string to a number (float)

1.4.13 Exercise: Function Arguments and Parsing

1. Open the following file:

   ```
   www/parse/parse.js
   ```

2. Complete the exercise.

3. Run the tests by opening: http://localhost:3000/parse/

1.4.14 Variable Scope

- **Scope** refers to how long a variable is alive and what code can see it
- There are basically two types of scope: **global** and **local**
- Functions are the only way to create a new local scope (with a few exceptions)
- If you don't use `var` then variables are **global**

1.4.15 Example: Three Scopes

```
var a = 5;

function foo(b) {
  var c = 10;
  d = 15;

  var bar = function(e) {
    var c = 2;
    a = 12;
    return a + c;
  };
}
```

- Three scopes exists in the above example
- Variables `a` and `d` are global
- There are two independent local variables named `c`
- Variable `bar` is a local variable containing a function.
- Variables `b` and `e` are local to their respective functions
- Each inner scope has access to the outer, but the outer scopes cannot access the inner ones
- `ReferenceError` indicates that a variable wasn't found in the current scope chain

1.4.16 Scope Tips

- Avoid using (and polluting) the global scope
- Use scoping to create namespaces (modules) your code

- You can "hide" things by wrapping them in a function
- Closures are born out of using lexical scope
- We'll see more of this later…
- No block scope

1.4.17 Exercise: Hoisting (Part 1)

What will the output be?

```
function foo () {
  a = 2;
  var a;

  console.log(a); // ?
  return a;
}
```

1.4.18 Exercise: Hoisting (Part 2)

And this one?

```
function foo () {
  console.log(b);
  var b = 2; // ?
}
```

Turns into:

```
function foo () {
  var b;
  console.log(b);
  b = 2;
}
```

1.4.19 Explanation of Hoisting

- Hoisting refers to when a variable declaration is lifted and moved to the top of its scope (only the declaration, not the assignment)
- Function statements are hoisted too, so you can use them before actual declaration

- JavaScript essentially breaks a variable declaration into two statements:

  ```
  var myVar=0, myOtherVar;

  // Is interpreted as:
  var myVar=undefined, myOtherVar=undefined;
  myVar=0;
  ```

1.4.20 Functions Recap

- Can be defined with a name or anonymously
- Are first class objects
- Create their own scope
- Declare variables at the top of the function to avoid hoisting

1.5 Objects

1.5.1 Back to Objects

- Remember: everything is an object
- Even primitives have object wrappers
- An object is a dynamic collection of properties

1.5.2 Object Literals

Create object literals with curly braces:

```
var myObjLiteral = {

  name: "Mr Object",

  age: 99,

  toString: function() {
    return this.name;
  },

};
```

1.5.3 Object Properties

There are four primary ways to work with object properties:

1. Dot notation:

   ```
   object.property = "foo";
   var x = object.property;
   ```

2. Square bracket notation:

   ```
   object["property"] = "foo";
   var x = object["property"];
   ```

3. Through the `Object.defineProperty` function

4. Using the `delete` function

1.5.4 Property Descriptors

- Object properties have descriptors that affect their behavior
- For example, you can control whether or not a property can be deleted or enumerated
- Typically, descriptors are hidden, use `defineProperty` to change them:

  ```
  var obj = {};

  Object.defineProperty(obj, "someName", {
    configurable: false // someName can't be deleted
  });
  ```

For more information on property descriptors, see this MDN article.

1.5.5 Object Reflection

Objects can be inspected with…

- the `typeof` operator:

  ```
  typeof obj;
  ```

- the `in` operator:

  ```
  "foo" in obj;
  ```

- the `hasOwnProperty` function:

  ```
  obj.hasOwnProperty("foo");
  ```

Keep in mind that objects "inherit" properties. Use the `hasOwnProperty` to see if an object actually has its own copy of a property.

1.5.6 The typeof Operator

Sometimes useful for determining the type of a variable:

```
typeof 42;        // "number"
typeof Math.abs;  // "function"
typeof [1, 2, 3]; // "object"
typeof null;      // "object"
typeof undefined; // "undefined"
```

(But not all that useful in reality.)

Instead of doing this:

```
if (typeof someVal === "undefined") {
  // ...
}
```

Just do:

```
if (someVal === undefined) {
  // ...
}
```

1.5.7 Property Enumeration

- The `for..in` loop iterates over an object's properties in an **unspecified** order.
- Use `object.hasOwnProperty(propertyName)` to test if a property is inherited or local.

```
for (var propertyName in object) {
   /*
      propertyName is a string.
```

```
      Must use this syntax:
      object[propertyName]

      Does not work:
      object.propertyName
  */
}
```

1.5.8 Object Keys

- Get an array of all “own”, enumerable properties:

  ```
  Object.keys(obj);
  ```

- Get even non-enumerable properties:

  ```
  Object.getOwnPropertyNames(obj);
  ```

1.5.9 Object References and Passing Style

- Objects can be passed to and from functions
- JavaScript is **call-by-sharing** (very similar to call-by-reference)
- Watch out for functions that modify your objects!
- Remember that `===` compares references
- Since `===` only compares references, it only returns `true` if the two operands are the same object in memory
- There’s no built in way in JS to compare objects for similar contents

1.5.10 JavaScript and Mutability

- All primitives in JavaScript are immutable
- Using an assignment operator just creates a new instance of the primitive
- You can think of primitives as using **call-by-value**
- Unless you used an object constructor for a primitive!
- Objects are mutable (and use **call-by-sharing**)
- Their values (properties) can change

1.5.11 Exercise: Create a `copy` Function

1. Open the following file:

   `www/copy/copy.js`

2. Complete the exercise.

3. Run the tests by opening http://localhost:3000/copy/

Hint: `for (var prop in obj) { /* ... */ }`

Hint: `obj.hasOwnProperty(prop)`

1.5.12 Built-in Objects

- `String`, `Number`, and `Boolean`
- `Function`
- `Array`
- `Date`
- `Math`
- `RegExp`
- `Error`

1.6 The String Object

1.6.1 The String Object

- 16 bit unicode characters (UCS-2, not quite UTF-16)
- Single or double quotes (no difference)
- Similar strings are `===` equal (checks contents)
- `>=` ES5 supports multiple line literals using a backslash

1.6.2 String Properties and Instance (Prototype) Methods

- `length`
- `charAt(i);`
- `concat();`
- `indexOf(needle);`

- `slice(iStart, iEnd);`
- `substr(iStart, length);`
- `replace(regex|substr, newSubStr|function);`
- `toLowerCase();`
- `trim();`

1.7 The Number and Math Object

1.7.1 The Number Object

- 64-bit binary floating point based on IEEE-754
- AKA `double` in Java
- `102`, `120.00`, `.0000000102`
- Be careful, decimals are approximate!

```
var a=0.1, b=0.2, c=0.3;
(a+b)+c != a+(b+c)
```

1.7.2 The Number Object (functions)

- Constants:
  - `Number.MAX_VALUE`
  - `Number.NaN`
  - `Number.POSITIVE_INFINITY`
  - etc.
- Generic Methods:
  - `Number.isInteger(n);`
  - `Number.isFinite(n);`
  - `Number.parseFloat(s);`
  - `Number.parseInt(s);`
- Prototype Methods:
  - `num.toString();`
  - `num.toFixed();`
  - `num.toExponential();`

1.7.3 The Math Object

- Constants:
  - `Math.E`

  - `Math.LOG2E`
  - `Math.PI`
  - etc.

- Generic Functions:
  - `Math.abs(n);`
  - `Math.pow(n, e);`
  - `Math.sqrt(n);`
  - etc.

1.8 The Date Object

1.8.1 The Date Object

- An instance of the Date object is used to represent a point in time
- Must be constructed:

```
var d = new Date(); // current date
var d = new Date("Wed, 28 Jan 2015 13:30:00 MST");
```

- Months start at 0, days start at 1
- Timestamps are unix time:

```
d.getTime(); // 1422477000000
```

1.8.2 The Date Object (functions)

- Generic Methods:
  - `Date.now();`
  - `Date.UTC();`
  - `Date.parse("March 7, 2014");`

- Prototype Methods:

```
var d = new Date();

d.getMonth();
d.getHours();
d.getMinutes();
d.getFullYear(); // Don't use d.getYear();
d.setYear(1990);
```

1.9 The Array Object

1.9.1 The Array Object

- Arrays are objects that behave like traditional arrays
- Use arrays when order of the data should be sequential

1.9.2 The Array Object (examples)

- Creating Arrays:

  ```
  // Array literal:
  var myArray = [1, 2, 3];

  // Using the constructor function:
  var myArray = new Array(1, 2, 3);
  ```

- Functions/Methods:

  ```
  var a = [1, 2, 3];
  a.length; // 3
  Array.isArray(a); // true (>= ES5)
  typeof a; // "object" :(
  ```

1.9.3 Array Cheat Sheet

- Insert: `a.unshift(x);` **or** `a.push(x);`
- Remove: `a.shift();` **or** `a.pop();`
- Combine: `var b = a.concat([4, 5]);`
- Extract: `a.slice(...);` **or** `a.splice(...);`
- Search: `a.indexOf(x);`
- Sort: `a.sort();`

1.9.4 Array Enumeration

WARNING: Use `for`, not `for...in`. The latter doesn't keep array keys in order!

```
for (var i=0; i < myArray.length; ++i) {
  // myArray[i]
}
```

1.9.5 The forEach Method

New in ES5:

```
myArray.forEach(function(val, index, arr) {
  // Do something...
});
```

1.9.6 Array Testing

- Test if a function returns `true` on all elements:

  ```
  var a = [1, 2, 3];

  a.every(function(val) {
    return val > 0;
  });
  ```

- Test if a function returns `true` at least once:

  ```
  a.some(function(val) {
    return val > 2;
  });
  ```

1.9.7 Functional Programming with Arrays

- `a.filter(f);`: New array filtered with a predicate `f`
- `a.map(f);`: New array after transforming with `f`
- `a.reduce(f);`: **Fold** an array into something else using `f`

1.9.8 Example: Using Reduce

```
var a = [1, 2, 3];

// Sum numbers in `a'.
var sum = a.reduce(function(acc, elm) {
  // 1. `acc' is the accumulator
  // 2. `elm' is the current element
  // 3. You must return a new accumulator
  return acc + elm;
}, 0);
```

1.9.9 Exercise: Arrays and Functional Programming

1. Open the following file:

   `www/array/array.js`

2. Complete the exercise.

3. Run the tests by opening http://localhost:3000/array/

Hint: Use http://devdocs.io/ or https://developer.mozilla.org/ for documentation.

Bonus Solution

1.10 Locking In the Basics

1.10.1 JavaScript Language Best Practices

1. Avoid polluting the global namespace
2. Define variables at top of your scope
3. Use `===` and `!==` (strict comparison)
4. Avoid primitive object wrappers like `Number()` and `String()`
5. `CamelCase` constructor functions
6. Use semicolons (`;`)
7. Always open and close blocks `{..}`
8. Indent your code (easier for humans)
9. Use a tool such as JSHint or ESLint

1.11 Common Patterns Involving Functions

1.11.1 Function Usage Patterns

- Anonymous Functions
- Closures
- Callbacks

1.11.2 Anonymous Functions

- A function expression without a name:

  ```
  var anon = function() {};
  ```

- Pros:
  - Powerful
  - Functions can be passed as arguments
  - Defined inline

- Cons:
  - Difficult to test in isolation
  - Discourages code re-use

1.11.3 Anonymous Functions (Tips)

- Name your anonymous functions

- It can be a good idea to name your anonymous functions

  ```
  (function myAnonFunc() {

    // body

  })();
  ```

- `myAnonFunc` is scoped to the function inner so it can iterate on itself, easier to debug; errors reference the function name

1.11.4 Closures: Basics

- One of the most important features of JavaScript

- And often one of the most misunderstood & feared features

- But, they are all around you in JavaScript

- Happens automatically when you use function expressions

1.11.5 Closures: Definitions

- When an inner function includes the scope of an outer function and the inner function maintains this scope even after the outer function has returned.
- When a function is able to remember and access its lexical scope, even when executing outside its lexical scope.
- When an inner function closes over the variables of an outer function it retains state and scope after it completes execution.

1.11.6 Lexical Scoping Example:

```
function a() {
  var name = "Grim";

  var b = function() {
    // `name' is in scope:
    console.log(name);
  };

  b();
}

a();
```

1.11.7 Closures: Example

```
function a() {
  var name = "Grim";

  var b = function() {
    console.log(name);
  };

  return b;
}

// Invoke `a' and get a function back:
var innerFunction = a();

// Sometime in the future...
innerFunction();
```

1.11.8 Closures: Practical Example

```
var module = (function() {

  var privateVar = 42;

  var getter = function() {
    return privateVar;
  };

  return {
    getPrivateVar: getter,
  };

})();

module.getPrivateVar(); // 42
```

1.11.9 Exercise: Sharing Scope

1. Open the following file:

   ```
   www/closure/closure.js
   ```

2. Complete the exercise.

3. Run the tests by opening http://localhost:3000/closure/

1.11.10 Functions as Callbacks

- When a function is provided as an argument as something to be invoked inline, or under specific circumstances (like an event):

  ```
  function runCallback(callback) {
    // does things
    return callback();
  }
  ```

1.11.11 Functions as Timers

- Establish delay for function invocation:

  ```
  // setTimeout(func, delayInMs[, arg1, argn]);
  var timer = setTimeout(func, 500);
  ```

- Use `clearTimeout(timer)` to cancel
- Establish an interval for periodic invocation

  ```
  setInterval(func, ms);
  clearInterval(timer);
  ```

- Context will always be global for the callbacks:

  http://jsfiddle.net/mrmorris/s5g2moc6/

1.11.12 Callbacks and Closures

- Be careful with function expressions in loops
- They can have scope issues:

  ```
  // What will this output?
  for (var i=0; i<3; i++) {
    setTimeout(function(){
      console.log(i);
    }, 1000*i);
  }
  console.log("Howdy!");
  ```

  Solution

1.11.13 Callbacks and Closures

- Instead, create an additional scope to maintain state for the inner function (expression)
- Closures save the day:

  http://jsfiddle.net/devalot/nudkrok8/

1.11.14 Function Patterns Recap

- Mind your scope! (Particularly in callbacks.)
- Closures create a persistent and private scope
- Functions are often passed around as callbacks

1.12 Scope and Context

1.12.1 Adding Context to a Scope

- We already discussed **scope**
  - Determines visibility of variables
  - Lexical scope (location in source code)
- There is also **context**
  - Refers to the location a function was invoked
  - Dynamic, defined at runtime
  - Context is accessible as the `this` variable

1.12.2 Context Example

The following code can be found at: http://jsfiddle.net/devalot/x56tss8v/

```
var apple = {
  name: "Apple",
  color: "red"
};

var orange = {
  name: "Orange",
  color: "orange"
};

var logColor = function() {
  console.log(this.color);
};

apple.logColor = logColor;
orange.logColor = logColor;

apple.logColor();
orange.logColor();
```

1.12.3 Context and the `this` Keyword

- The `this` keyword is a reference to “the object of invocation”
- Bound at invocation (depends on the call site)
- Allows a method to reference the “current” object

- A single function can then service multiple objects
- Central to prototypical inheritance in JavaScript

1.12.4 Constructor Functions and the `new` Operator

What's going on when you use `new`?

```
var m = new Message("pjones@devalot.com", "Hello");
m.send();
```

1.12.5 Writing a Constructor Function

```
var Message = function(sender, content) {
  this.sender  = sender;
  this.content = content;
  this.length  = content.length;
};

Message.prototype = {
  send: function() {
    if (this.length !== 0) {
      console.log(this.content);
    }
  },
};
```

1.12.6 The `new` Keyword

```
var m = new Message("pjones@devalot.com", "Hello");
m.send();
```

The `new` operator does the following:

1. Creates a new, empty object
2. Calls the function given as its operand, setting `this` to the newly created object
3. Sets up inheritance for the object and records which function constructed the object.

1.12.7 Implementing Our Own new Operator

```
var fakeNew = function(func) {
  var newObject = Object.create(func.prototype);

  newObject.constructor = func;
  func.call(newObject);

  return newObject;
};
```

1.12.8 Factory Functions (Hand-made Constructors)

```
var Message = function(sender, content) {
  var m = Object.create(Message.prototype);

  m.sender  = sender;
  m.content = content;
  m.length  = content.length;

  return m;
};

Message.prototype = { /* ... */ };

var message = Message("pjones@devalot.com", "Hello");
```

1.12.9 How JavaScript Sets the this Variable

- Resides in the global binding
- Implicit (inner function does not capture this)
- The this object can be set manually!

Chapter 2

Exception Handling

2.1 Errors in JavaScript

Handling errors in JavaScript is done through exceptions. Programmers familiar with Java or C++ will feel (mostly) comfortable with JavaScript's exception system.

2.1.1 Exception Basics

- Errors in JavaScript propagate as exceptions
- Dealing with errors therefore requires an exception handler
- Keywords for exception handling:
  - `try`: Run code that might throw exceptions
  - `catch`: Capture a propagating exception
  - `throw`: Start exception processing
  - `finally`: Resource clean-up handler

2.1.2 Example: Throwing an Exception

When a major error occurs, use the `throw` keyword:

```
if (someBadCondition) {
  throw "Well, this is unexpected!";
}
```

2.1.3 Exception Objects

While you can throw exceptions with primitive types such as numbers and strings, it's more idiomatic to throw exception objects.

2.1.4 Built-in Exception Objects

- `Error`: Generic run-time exception
- `EvalError`: Errors coming from the `eval` function
- `RangeError`: Number outside expected range
- `ReferenceError`: Variable used without being declared
- `SyntaxError`: Error while parsing code
- `TypeError`: Variable not the expected type
- `URIError`: Errors from `encodeURI` and `decodeURI`

2.1.5 Creating Your Own Exception Object

This looks more traditional, but it's missing valuable information.

```
function ShoppingCartError(message) {
  this.message = message;
  this.name    = "ShoppingCartError";
}

// Steal from the `Error' object.
ShoppingCartError.prototype = Error.prototype;

// To throw the exception:
throw new ShoppingCartError("WTF!");
```

2.1.6 Custom Exceptions: The Better Way

If you start with an `Error` object, you retain a stack trace and error source information (e.g., file name and line number).

```
var error = new Error("WTF!");
error.name = "ShoppingCartError";
error.extraInfo = 42;
throw error;
```

2.2 Catching Exceptions

If you can handle an error condition thrown from code inside a `try` block then you can use a `catch` block to do so. In JavaScript you can only use a *single* `catch` statement. That means you have to catch an exception and then inspect it to see if it's the one you can handle.

2.2.1 Example: Catching Errors

```
var beSafe = function() {
  try {
    // Some code that might fail.
  }
  catch (e) {
    // Errors show up here.  All of them.
  }
};
```

2.2.2 Example: Catching Exceptions by Type

Most of the time you only want to deal with specific exceptions:

```
var beSafe = function() {
  try { /* Code that might fail. */ }
  catch (e) {
    if (e instanceof TypeError) {

      // If you're here then the error is a TypeError.

    } else {
      throw e; // Re-throw the exception.
    }
  }
};
```

Chapter 3

Regular Expressions

3.1 Introduction to Regular Expressions

3.1.1 Regular Expressions

- Patterns used to match character combinations in strings
- Very tough to understand but extremely powerful
- Useful for data validation
- JavaScript supports literals for the `RegExp` object:

```
var re = /^\d+$/;
re.test("1234"); // true
```

3.1.2 Expression Language Primer

Token	Meaning
`.`	Match any single character
`\w`	Match a word character
`\d`	Match a digit
`\s`	Match a space character
`\b`	Word boundary

Repeater	Meaning
`?`	Match zero or one preceding token

Repeater	Meaning
`*`	Match zero or more preceding tokens
`+`	Match one or more preceding tokens

3.2 Using Regular Expressions

3.2.1 `String` Methods That Take Regular Expressions

`str.match(re);` If the expression matches, returns an array describing what matched.

`str.replace(re);` Replace parts of a string matched by an expression.

`str.search(re);` Tests to see if the expression matches. Faster than `match` because it stops after the first match and returns `1`.

`str.split(re);` Split a string at locations matched by the expression and return an array.

3.2.2 Exercise: String Manipulation

1. Open the following file:

   `www/string/string.js`

2. Complete the exercise.
3. Run the tests by opening http://localhost:3000/string/

Hint: Use http://devdocs.io/ or https://developer.mozilla.org/ for documentation.

Solution

3.3 Additional Resources on Regular Expressions

- Interactive Tool
- Cheat Sheet

Chapter 4

JavaScript and the Web Browser

4.1 Where JavaScript Fits In

4.1.1 JavaScript and the Browser

How JavaScript fits in:

- HTML for content and user interface
- CSS for presentation (styling)
- JavaScript for behavior (and business logic)

4.1.2 Question Time: Can You...

- Write an HTML form from scratch?
- Style a form (or full page) from scratch?
- Manipulate elements in the page with just the DOM?
- Set up an event handler for form submissions? Clicks?
- Know what events are and why they are important?

4.2 Brief Overview of HTML

4.2.1 What is HTML?

- Hyper Text Markup Language

- HTML is very error tolerant (browsers are very forgiving)
- That said, you should strive to write good HTML
- Structure of the UI and the content of the **view data**
- Parsed as a tree of nodes (elements)
- HTML5
  - Rich feature set
  - Semantic (focus on content and not style)
  - Cross-device compatibility
  - Easier!

4.2.2 Anatomy of an HTML Document/Page

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello Again!</h1>
    <p>This is a paragraph.</p>
  </body>
</html>
```

4.2.3 Anatomy of an HTML Element

- Also known as: nodes, elements, and tags:

  ```
  <element key="value">
    Content of element
  </element>
  ```

- Block vs. inline:

  ```
  <p>Paragraph</p>  <!-- Creates a new visual block -->
  <span>Text</span> <!-- Only affects inline text -->

  <p>Hey, this is a <span>paragraph</span></p>
  ```

- Self-closing elements:

  ```
  <input type="password" name="pin"/>
  ```

4.2.4 HTML Element Refresher: Structure Elements

- `div`, and `span`
- `table`, `tr`, `td`, `thead`, `tbody`, etc.
- `form`, `fieldset`, `label`, `input`, etc.
- And new HTML5 semantic elements

4.2.5 HTML Element Refresher: Content Elements

- `h1` through `h6`
- `p`
- `ol` or `ul` along with `li`
- Text modifies such as `em` and `strong`

4.2.6 HTML Element Refresher: Reference

- https://developer.mozilla.org/en-US/docs/Web/HTML/Element

4.2.7 HTML5 Semantic Elements

- Designed to degrade gracefully on non-HTML5 browsers
- Defines an outline and semantic hints for a document
  - `header`, `footer`, `nav`, `main`
  - `section`, `article`, `aside`, `figure`, `figcaption`
  - `time`, `mark`, `details`, `summary`
- http://jsfiddle.net/mrmorris/cb47mzpq/

4.2.8 HTML5 Forms

- New input types:
  - `number`, `range`, `url`, `email`
  - `tel`, `color`, `search`
- New element: `datalist`
- New input attributes:
  - `required`, `autofocus`, `placeholder`, `list`
- Built-in validation
- http://jsfiddle.net/mrmorris/zh18vn4x/

4.3 Brief Overview of CSS

4.3.1 What is CSS?

- Cascading Style Sheets
- Rule-based language for describing the look and formatting
- Separates presentation from content
- Can be a separate file or inline in the HTML
- Prefer using a separate file

4.3.2 What Does CSS Look Like?

```
p {
  background-color: white;
  color: blue;
  padding: 5px;
}

.spoiler {
  display: none;
}

p.spoiler {
  display: block;
  font-weight: bold;
}
```

4.3.3 Anatomy of a CSS Declaration

- Rules (called selectors) choose which elements you want to style. In the body of the rule you set properties:

  ```
  selector {
    property-x: value;
    property-y: val1 val2;
  }
  ```

- For example:

  ```
  h1 {
    color: #444;
    border: 1px solid #000;
  }
  ```

4.3.4 The Various Kinds of Selectors

- Using the element's type (name):
  - HTML: `<h1>Hello</h1>`
  - CSS: `h1 {...}`
- Using the ID attribute:
  - HTML: `<div id="header"></div>`
  - CSS: `#header {...}`
- Using the class attribute:
  - HTML: `<div class="main"></div>`
  - CSS: `.main {...}`
- Using any attribute:
  - HTML: `<div name="user"></div>`
  - CSS: `div[name="user"] {...}`
- Using location or relationships:
  - HTML: `<ul><li><p>One</p></li><li>Two</li></ul>`
  - CSS: `ul li p {...}`

4.3.5 The Cascade

What happens when properties conflict?

- HTML:

  ```
  <div id="main" class="fancy">
    What color will this text be?
  </div>
  ```

- CSS:

  ```
  #main {color: red;}

  #main.fancy {color: blue;}

  div.fancy {color: green;}
  ```

4.3.6 Specificity Chart

Selector	Points
Universal selector	0
Type selectors	1
Pseudo elements	1
Classes	10
Pseudo classes	10
Attribute selectors	10
ID selectors	100

- Inline styles add 1,000 points.
- Tie breaker: last defined style wins.
- Force highest specificity with `!important`.

4.3.7 New Selectors and Classes

- Attribute selectors:

  http://jsfiddle.net/mrmorris/tp6t6skt

- Sibling selectors:

  http://jsfiddle.net/mrmorris/98jg21y3/

- Pseudo-classes, form inputs:

  http://jsfiddle.net/mrmorris/nqsbj80o/

- Pseudo-classes, structural (location):

  http://jsfiddle.net/mrmorris/ghddq4eu/

4.4 Getting JavaScript into the Browser

4.4.1 How the Browser Processes JavaScript

- HTML parser continues to process HTML while downloading JS
- Once downloaded, JS is executed and blocks the browser
- Include the JS at the bottom of the page to prevent blocking

4.4.2 Getting JavaScript into a Web Page

- Preferred option:

  ```
  <script src="somefilename.js"></script>
  ```

- Inline in the HTML (yuck):

  ```
  <script>
    var x = "Hey, I'm JavaScript!";
    console.log(x);
  </script>
  ```

- Inline on an element (double yuck):

  ```
  <button onclick="console.log('Hey there');"/>
  ```

4.4.3 How JavaScript Affects Page Load Performance (Take Two)

- The browser blocks when executing JS files
- JS file will be downloaded then executed before browser continues
- Put scripts in file and load them at the bottom of the page

Chapter 5

The Document Object Model

5.0.1 What is the DOM?

- What most people hate when they say they hate JavaScript
- The DOM is the browser's API for the document
- Through it you can manipulate the document
- Browser parses HTML and builds a tree structure
- It's a live data structure

5.0.2 The Document Structure

- The `document` object provides access to the document
- It's a tree-like structure
- Each node in the tree represents one of:
  - Element
  - Content of an element
- Relationships between nodes allow traversal

5.0.3 Looking at the Parsed HTML Tree (Part 1)

The browser will parse the following HTML:

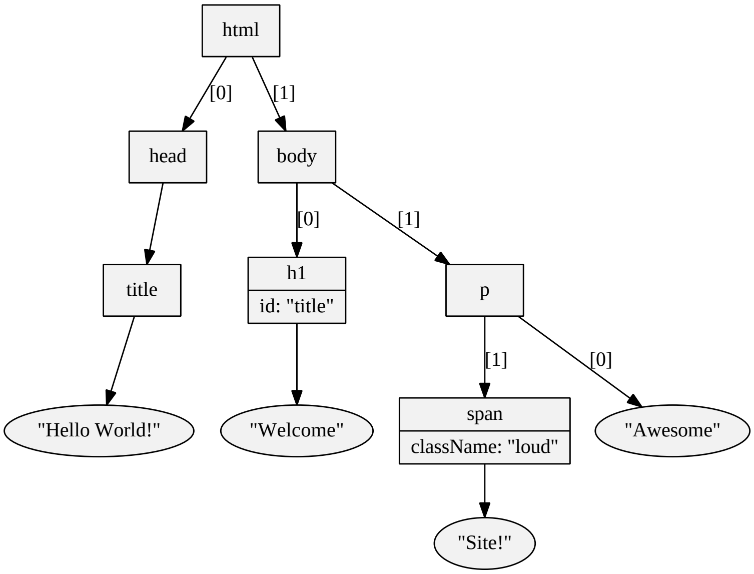
```html
<html>
  <head>
    <title>Hello World!</title>
  </head>

  <body>
    <h1 id="title">Welcome</h1>

    <p>
      Awesome <span class="loud">Site!</span>
    </p>
  </body>
</html>
```

5.0.4 Looking at the Parsed HTML Tree (Part 2)

And produce this tree structure:



5.0.5 Element Nodes

- The HTML:

  ```
  <p id="name" class="hi">My <span>text</span></p>
  ```

- Maps to:

  ```
  var node = {
    tagName:    "P",
  ```

```
  childNodes: NodeList,
  className:  "hi",
  innerHTML:  "My <span>text</span>",
  id:         "name",
  // ...
};
```

  - Attributes may **very loosely** to object properties

5.0.6 Working with the Document Object Model

- Accessing elements:
  - Select a single element
  - Select many elements
  - Traverse elements
- Working with elements
  - Text nodes
  - Raw HTML
  - Element attributes

5.0.7 Performance Considerations

- Dealing with the DOM brings up a lot of performance issues
- Accessing a node has a cost (especially in IE)
- Styling has a bigger cost (it cascades)
  - Inserting nodes
- Layout changes
  - Accessing CSS margins
  - Reflow
  - Repaint
- Accessing a `NodeList` has a cost

5.1 Getting References to Elements

Starting with the `document` global variable, you can access specific elements in the DOM using the following functions. Once you have a specific element you can use these functions again (with the element as the receiver) to search the DOM, which starts the search in the element's decedents.

5.1.1 Accessing Individual Elements

Starting on the `document` object or a previously selected element:

`getElementById("main");` Returns the element with the given ID (e.g., `<div id="main">`).

`querySelector("p span");` Returns the *first* element that matches the given CSS selector. The search is done using depth-first pre-order traversal.

5.1.2 Accessing a List of Elements

Starting on the `document` object or a previously selected element:

`getElementsByTagName("a");` Returns a `NodeList` containing *all* `<a>` elements.

`getElementsByClassName("foo");` Returns a `NodeList` containing *all* elements that have a `class` attribute set to `foo` (e.g., `<div class="foo">`).

`querySelectorAll("p span");` Returns a `NodeList` containing *all* elements that match the given CSS selector.

5.2 Traversing the DOM

Once you have a single element in the DOM you can traverse from that point to somewhere else in the tree using the following read-only **properties**:

5.2.1 Traversal Functions

`parentNode` The parent of the specified element.

`previousSibling` The element immediately preceding the specified element.

`nextSibling` The element immediately following the specified element.

`firstChild` The first child element of the specified element.

`lastChild:` The last child element of the specified element.

`childNodes` A `NodeList` containing the direct decedents (children) of the specified element.

5.2.2 Traversal Example

Note: Remember that when you traverse the DOM you will encounter text nodes and comments in addition to child elements!

(1) Example: Examining the children of a node:

```
var main = document.getElementById("main");

if (main) {
  console.log("#main child count: ", main.childNodes.length);
  console.log("first child is: ", main.firstChild);
}
```

5.3 Node Types

While traversing the DOM it's helpful to know which type of nodes you are working with. The `nodeType` property is an integer that precisely identifies the node.

5.3.1 The `nodeType` Property

Interesting values for the `element.nodeType` property:

Value	Description
1	Element node
3	Text node
8	Comment node
9	Document node

5.4 Manipulating the DOM Tree

5.4.1 Creating New Nodes

`document.createElement("a");` Creates and returns a new node without inserting it into the DOM. In this example, a new `<a>` element is created.

`document.createTextNode("hello");` Creates and returns a new text node with the given content.

5.4.2 Adding Nodes to the Tree

```
var element = document.getElementById("foo"),
    child   = element.firstChild,
    other   = document.createElement("a");
```

`element.appendChild(other);` Appends `other` to the end of `element.childNodes`.

`element.removeChild(child);` Removes `child` from `element.childNodes`.

`element.insertBefore(other, child);` Inserts `other` in `element.childNodes` just before the existing child node `child`.

`element.replaceChild(other, child);` Removes `child` from `element.childNodes` and inserts `other` in its place.

5.5 Node Attributes

5.5.1 Getting and Setting Node Attributes

```
var element = document.getElementById("foo"),
    name    = "bar";
```

`element.getAttribute(name);` Returns the value of the given attribute.

`element.setAttribute(name, value);` Changes the value of the given attribute name to `value`.

`element.hasAttribute(name);` Returns `true` if `element` has an attribute with the given name.

`element.removeAttribute(name);` Removes the named attribute from `element`.

5.6 The Class Attribute

5.6.1 Class Attribute API

```
var element = document.getElementById("foo"),
    name    = "bar";
```

`element.classList.add(name);` Add `name` to the list of classes in the class attribute.

`element.classList.remove(name);` Remove `name` from the list of classes in the class attribute.

`element.classList.toggle(name);` If `name` is present in the class list, remove it. Otherwise add it to the class list.

`element.classList.contains(name);` Check to see if the class list contains `name`.

5.7 Node Content

5.7.1 HTML and Text Content

```
var element = document.getElementById("foo"),
    name    = "bar";
```

`element.innerHTML` Get or set the element's decedents as HTML.

`element.textContent:` Get or set *all* of the text nodes (including decedents) as a single string.

`element.nodeValue` If `element` is a text node, comment, or attribute node, returns the content of the node.

`element.value` If `element` is a form input, returns its value.

5.8 DOM Nodes: Exercises

5.8.1 Exercise: DOM Manipulation

1. Open the following files:

   - `www/flags/flags.js`
   - `www/flags/index.html`

2. Open http://localhost:3000/flags/

3. Complete the exercise.

5.9 Event Handling and Callbacks

5.9.1 Events Overview

- Single-threaded, but asynchronous event model
- Events fire and trigger registered handler functions
- Events can be click, page ready, focus, submit (form), etc.

5.9.2 So Many Events!

- UI: load, unload, error, resize, scroll
- Keyboard: keydown, keyup, keypress
- Mouse: click, dblclick, mousedown, mouseup, mousemove
- Touch: touchstart, touchend, touchcancel, touchleave, touchmove
- Focus: focus, blur
- Form: input, change, submit, reset, select, cut, copy, paste

5.9.3 Using Events (the Basics)

1. Select the element you want to monitor
2. Register to receive the events you are interested in
3. Define a function that will be called when events are fired

5.9.4 Event Registration

Use the `addEventListener` function to register a function to be called when an event is triggered:

Example: Registering a click handler:

```
var main = document.getElementById("main");

main.addEventListener("click", function(event) {
  console.log("event triggered on: ", event.target);
});
```

Note: Don’t use older event handler APIs such as `onClick`!

See this reference for a list of all event types.

5.9.5 Event Handler Call Context

- Functions are called in the context of the DOM element
- I.e., `this === eventElement`
- Use `bind` or the `var self = this;` trick

5.9.6 Event Propagation

Some additional details about events, propagation, and the browser's default action.

- By default, events propagate from the target node upwards until the root node is reached (bubbling).
- Event handlers can stop propagation using the `event.stopPropagation` function.
- Event handlers can also stop the browser from performing the default action for an event by calling the `event.preventDefault` function

Example: Typical Event Handler

```
main.addEventListener("click", function(event) {
  event.stopPropagation();
  event.preventDefault();

  // ...
}, false);
```

5.9.7 Event Delegation

- Parent receives event instead of child (via bubbling)
- Children can change without messing with event registration
- Fewer handlers registered, fewer callbacks
- Relies on some event object properties:
  - `event.target`: The element the event triggered for
  - `event.currentTarget`: Registered element (parent)

5.9.8 Event Handler: Elements and Actions

```
element.addEventListener("click", function(event) {

  // Add a CSS class:
  event.target.classList.add("was-clicked");

  // You can stop default browser behavior:
  event.preventDefault();

  // Or you can stop the event from bubbling:
  event.stopPropagation();
});
```

5.10 Event Handling Exercises

5.10.1 Exercise: Simple User Interaction

1. Open the following files:

   - `www/events/events.js`
   - `www/events/index.html`

2. Open http://localhost:3000/events/

3. Complete the exercise.

5.11 Event Handling Tips and Techniques

5.11.1 Event Loop Warnings

- Avoid blocking functions (e.g., `alert`, `confirm`)

- For long tasks use eteration or web workers

- Eteration: Break work up using `setTimeout(0)`

5.11.2 Event "Debouncing"

- Respond to events in intervals instead of in real-time
- Reuse a timeout object to process events in the future

```
var input   = document.getElementById("search"),
    output  = document.getElementById("output"),
    timeout = null;

var updateSearchResults = function() {
  output.textContent = input.value;
};

input.addEventListener("keydown", function(e) {
  if (timeout) clearTimeout(timeout);
  timeout = setTimeout(updateSearchResults, 100);
});
```

Chapter 6

Asynchronous JavaScript and XML

6.1 Introduction

6.1.1 Ajax Basics

- Asynchronous JavaScript and XML
- API for making HTTP requests
- Handled by the `XMLHttpRequest` object
- Introduced by Microsoft in the late 1990s
- Why use it? Non-blocking server interaction!
- Limited by the same-origin policy

6.1.2 Ajax: Step by Step

1. JavaScript asks for an HTTP connection
2. Browser makes a request in the background
3. Server responds in XML/JSON/HTML
4. Browser parses and processes response
5. Browser invokes JavaScript callback

6.2 The XHR API

6.2.1 Sending a Request, Basic Overview

```
var req = new XMLHttpRequest();

// Attach event listener...

req.open("GET", "/example/foo.json");
req.send(null);
```

6.2.2 Knowing When the Request Is Complete

```
var req = new XMLHttpRequest();

req.addEventListener("load", function(e) {
  if (req.status == 200) {
    console.log(req.responseText);
  }
});
```

Full example: http://jsfiddle.net/devalot/pz2kf3jj/5/

6.3 Payload Formats

6.3.1 Popular Data Formats for Ajax

- HTML: Easiest to deal with
- XML: Pure data, but verbose
- JSON: Pure data, very popular

6.3.2 Ajax with HTML

- Easiest way to go
- Just directly insert the response into the DOM
- Scripts will **not** run

6.3.3 Ajax with XML

- More work to turn the XML into HTML
- http://jsfiddle.net/devalot/axpj7zv7/

6.3.4 What is JavaScript Object Notation (JSON)?

- Used as a data storage and communications format
- Very similar to object literals, with a few restrictions
  - Property names must be in double quotes
  - No function definitions, function calls, or variables
- Built-in methods:
  - `JSON.stringify(object);`
  - `JSON.parse(string);`
- Example:

```
{
  "messages": [
    {"text": "Hello", "priority": 1},
    {"text": "Bye",   "priority": 2}
  ],
  "sender": "Lazy automated system"
}
```

6.3.5 Ajax with JSON

- Sent and received as a string
- Needs to be serialized and de-serialized:

```
req.send(JSON.stringify(object));

// ...

var data = JSON.parse(req.responseText);
```

Full example: http://jsfiddle.net/devalot/z5k2udk0/

6.4 Tips and Tricks

6.4.1 Should You Use the XHR API?

- It is best to use an abstraction for `XMLHttpRequest`

- They usually come with better:
  - `status` and `statusCode` handling
  - Error handling
  - Callback registration
  - Variations in browser implementations
  - Additional event handling (progress, load, error, etc.)
- So, use a library like jQuery

6.5 Putting It All Together

6.5.1 Exercise: Making Ajax Requests

1. Open the following files:
   - `www/ajax/ajax.js`
   - `www/ajax/index.html`
2. Open http://localhost:3000/ajax/
3. Complete the exercise.

6.6 Restrictions and Getting Around Them

6.6.1 Same-origin Policy and Cross-origin Requests

- By default, Ajax requests must be made on the same domain
- Getting around the same-origin policy
  - A proxy on the server
  - JSONP: JSON with Padding
  - Cross-origin Resource Sharing (CORS) (>= IE10)

6.6.2 Introducing JSONP

- Browser doesn't enforce the same-origin policy for resources (images, CSS files, and JavaScript files)
- You can emulate an Ajax call to another domain that returns JSON by doing the following:

1. Write a function that will receive the JSON as an argument
2. Create a `<script>` element and set the `src` attribute to a remote domain, include the name of the function above in the query string.
3. The remote server will return JavaScript (not JSON)
4. The JavaScript will simply be a function call to the function you defined in step 1, with the requested JSON data as its only argument.

6.6.3 Example: JSONP

1. Define your function:

   ```
   function myCallback (someObject) { /* ... */ }
   ```

2. Create the script tag:

   ```
   <script src="http://server/api?jsonp=myCallback">
   </script>
   ```

3. The browser fetches the URL, which contains:

   ```
   myCallback({answer: "Windmill"});
   ```

4. Your function is called with the requested data

Chapter 7

Exercises and Solutions

7.1 Scope Sharing Exercise

One possible solution:

```
function outer(value) {
  var shared = value;

  var inner1 = function() {
    console.log("from inner1: " + shared);
  };

  var inner2 = function() {
    console.log("from inner2: " + shared);
  };

  return [inner1, inner2];
}

funcs = outer(15);
funcs[0]();
funcs[1]();
```

And here’s another version, this time using objects:

```
var outer = function(value) {
  return {
    first: function() {
      console.log("First:", value);
      },
      second: function() {
        console.log("Second:", value);
      }
    };
  };

  funcs = outer(15);
  funcs.first();
  funcs.second();
```

7.2 Arrays: Reverse an Array

1. Reverse and array:

```
function reverse (array) {
  var result = [];

  for (var j=0, i=array.length - 1; i >= 0; --i, ++j) {
    result[j] = array[i];
  }

  return result;
}

console.log(reverse(["A", "B", "C", "D"]).toString());

function reverse2 (array) {
  var result = [];

  for (var i=0; i<array.length; ++i) {
    result.unshift(array[i]);
  }

  return result;
}

console.log(reverse2(["A", "B", "C", "D"]).toString());
```

Throw an exception:

```
  function safeReverse (toReverse) {
    var result = [];
    //  Solution 1:
    if (!Array.isArray(toReverse)) {
      throw new TypeError("safeReverse expects an array");
    }

    // Solution 2:
    if (!(toReverse instanceof Array)) {
      throw new TypeError("safeReverse expects an array");
    }

    // Now reverse the array...
  }
```

Inline version (bonus):

```
  function inlineReverse (array) {
    for (var x, i=0, j=array.length - 1; i < j; ++i, --j) {
      x = array[i];
      array[i] = array[j];
      array[j] = x;

    }

    return array;
  }

  console.log(inlineReverse(["A", "B", "C", "D"]).toString());
  console.log(inlineReverse(["A", "B", "C"]).toString());
```

7.3 Strings: Replacing Words

```
processString = function(input) {
  var today = (new Date()).toDateString(),
      count = 0;

  var result = input.replace(/\b\w+\b/g, function(word) {
    count += (word.match(/x/gi) || []).length;

    switch (word.toLowerCase()) {
    case "today":
      return today;

    case "pi":
      return "3.14";

    default:
      return word;
    }
  });

  return result + " " + count;
};
```

7.4 Printing an Array of Objects

```
  function gridify (list) {

    if (list.length === 0) {
      return;
    }

    // Log the header row:
    var headers = Object.keys(list[0]).sort();
    console.log(headers.map(String.capitalize).join("\t"));

    // Log each of the objects:
    list.forEach(function(e) {
      var values = headers.map(function(h) {
        return e[h];
      });

      console.log(values.join("\t"));
    });
  }

  gridify([
    {name: "Ryan", value: 913},
    {name: "Jimmy", value: 20003},
    {name: "Donna", value: 923}

  ]);
```

7.5 Callbacks and Closures

- All of them print `3`
- Original Exercise

```
for (var i=0; i<3; i++) {
  (function(index){
    setTimeout(function(){
      console.log(index);
    }, 1000*index);
  })(i);
}

// Rewrite:

for (var i=0; i<3; i++) {
  var outer = function(index) {
    var inner = function() {
      console.log(index);
    };

    setTimeout(inner, 1000*index);
  };

  outer(i);
}
```

7.6 Create an Object Literal

- Create an object that represents yourself
- Example properties:
  - Name
  - Age
  - Height
  - etc.

7.7 Add a Method to Your Object

- Create a function property on your object
- Call the function `speak`
- It should accept a string argument and log a message
- Example:

  ```
  var me = {
    name: "Peter",
  ```

```
  height: 67,
  speak: function(message) {
    // log: "Peter says {message}"
  }
};

me.speak("hello there!");
```

One Possible Solution:

```
var me = {
  name: "Peter",
  height: 67,
  speak: function(message) {
    console.log(this.name + " says " + message);
  }
};
```

7.8 Flags, Buckets, and Events

- http://jsfiddle.net/devalot/fgvpdLd8/

```
var Bucket = function(bucket_id) {
  var bucket = document.getElementById(bucket_id);

  var move = function(element) {
    bucket.appendChild(element);
  };

  var moveOnClick = function(selector) {
    var element = document.querySelector(selector);

    if (!element) {
      console.error("No matching element: " + selector);
      return;
    }

    element.addEventListener("click", function(e) {
      move(e.target);
    });
  };

  return {
    move: move,
    moveOnClick: moveOnClick
  };
};

var bucket = Bucket("bucket");
bucket.moveOnClick(".main li:nth-child(2)");
bucket.moveOnClick("#articles .flag");
bucket.moveOnClick(".footer div div div div");
```

7.9 Tabbed UIs, JSON, and Ajax

1. Make a tabbed user interface work:
   http://jsfiddle.net/mrmorris/osq6fed3/
2. Turn JSON data into an HTML table:
   http://jsfiddle.net/mrmorris/mnyn3y0t/
3. Bonus points: use Ajax to load the data

JavaScript Resources

7.10 JavaScript Documentation

- Mozilla Developer Network

7.11 Books on JavaScript

- JavaScript: The Good Parts
  - By: Douglas Crockford
  - Great (re-)introduction to the language and common pitfalls
- "You Don't Know JS" (book series)
  - By: Kyle Simpson
  - Look at JavaScript in a new light
  - https://github.com/getify/You-Dont-Know-JS
- Learning JavaScript Design Patterns
  - By: Addy Osmani
  - Through book about design patters in JavaScript
  - Exercises and Answers