Core JavaScript

Fundamentals of the Language

Peter J. Jones
pjones@devalot.com
@devalot
http://devalot.com
April 3, 2018

DEVALOT

Contents

Introduction to This Course

Source Code

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

https://github.com/devalot/webdev

Overview

This JavaScript course is delivered during a single day.

What's In Store

Before Lunch	After Lunch
JavaScript Basics	Object-Oriented Programming
Control Flow	Debugging in the Browser
Object Basics	Exception Handling
Arrays and Builtin Objects	Serialization w/ JSON
Functions and Closures	Best Practices

Course Requirements

Developer Tools

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

- Node.js LTS
- Google Chrome

Text Editor or IDE

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:

- Visual Studio Code
- Atom
- Sublime Text

Websites

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 (if necessary).

- GitHub.com

  Class-specific updates to the course source code.

- JSFiddle

  Fast prototyping and experimenting.

- Mozilla Developer Network

  Excellent documentation for HTML, CSS, and JavaScript

Chapter 1

Core JavaScript: The Language

1.1 Introduction to JavaScript

1.1.1 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.

1.1.2 A Little Bit About JavaScript

- Standardized as ECMAScript
  - 5th Edition, 2009 (widely supported)
  - 6th Edition, 2015 (not so much)
  - 7th Edition, 2016
  - 8th Edition, 2017
- 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

1.1.3 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

1.1.4 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

1.1.5 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

1.1.6 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: `;`

You might also want to a reference page on Lexical Structure and Keywords in JavaScript.

1.1.7 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.

1.1.8 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");
```

1.1.9 Simple Console 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 Values and Operators

1.2.1 Primitive Values vs. Objects

- Primitive Values:

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

- Objects (arrays, functions, etc.)

1.2.2 Variables in JavaScript

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


1.2.3 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`

1.2.4 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.)

1.2.5 `undefined` and `null`

- There are two special values: `null` and `undefined`
- Variables declared *without* a value will start with `undefined`
- Setting a variable to `null` usually indicates "no appropriate value"

1.2.6 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
```

1.2.7 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)

1.2.8 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
```

1.2.9 Value Coercion

- JavaScript is loosely typed (uni-typed)
- Implicit conversion between "types" as needed
- Usually in unexpected ways:

```
8 * null; // 0

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

1.2.10 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. */
```

1.2.11 Exercise: Using Primitive Types

1. Open the following file:

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

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

1.2.12 JavaScript Operators

Arithmetic	`+`	`-`	`*`	`/`	`%`	`**`	
Shortcut	`+=`	`-=`	`*=`	`/=`	`%=`	`**=`	
Inc/Dec	`++n`	`n++`	`--n`	`n--`			
Bitwise	`~`	`&`	`|`	`^`	`>>`	`<<`	`>>>`
Comparison	`>`	`>=`	`<`	`<=`			
Equality	`==`	`!=`	`===`	`!==`			
Logic	`!`	`&&`	`||`				
Object	`.`	`[]`					
String	`+`						

(Most operators have assignment shortcut versions.)

1.3 Equality in JavaScript

1.3.1 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
```

1.3.2 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.)

1.3.3 Same-Value Equality

Similar to "`===`" with a few small changes:

```
Object.is(NaN, NaN); // true

Object.is(+0, -0);   // false
```

(This function first appeared in ECMAScript Edition 6, 2015.)

1.4 Boolean Values and Logic Operators

1.4.1 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
```

1.4.2 Boolean Operators: && (Conjunction)

`a && b` returns either `a` or `b` and short circuits:

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

1.4.3 Boolean Operators: || (Disjunction)

`a || b` returns either `a` or `b` and short circuits:

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

1.4.4 Boolean Operators: !

Boolean negation: `!`:

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

Double negation: `!!`:

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

1.4.5 Exercise: Boolean Operators

- Experiment with `&&`:

  ```
  0 && console.log("Yep");
  1 && console.log("Yep");
  ```

- Experiment with `||`:

  ```
  0 || console.log("Yep");
  1 || console.log("Yep");
  ```


1.4.6 Conditional Statements

```
if (expression) { then_part; }

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

1.4.7 Chaining Conditionals

Shorthand:

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

Long form:

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

1.4.8 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!

1.4.9 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 */ }
  ```

1.4.10 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.

1.4.11 Traditional while Loops

```
var i=0;

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

1.4.12 Flipped while Loops

```
var i=0;

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


1.4.13 Controlling a Loop

- Loops can be labeled and exited with `break`.

```
for (var i=1; i<100; ++i) {
  if (i % 2 === 0) break;
  console.log(i);
}
// prints 1
```

- Use `continue` to skip to the next iteration of the loop.

```
for (var i=1; i<100; ++i) {
  if (i % 2 === 0) continue;
  console.log(i);
}
// prints 1, 3, 5, 7, etc.
```

1.4.14 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";
```

1.4.15 Exercise: Experiment with Control Flow

1. Open the following file:

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

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

1.5 Objects

1.5.1 A Collection of Key/Value Pairs

- 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
```

1.5.2 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

1.5.3 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
  enumerable:   false, // someName is hidden
  writable:     false, // No setter for someName
  // ...
});
```

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 NaN;        // "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:

   `src/www/js/copy/copy.js`

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Hints:

- `for (var prop in someobj) { /* ... */ }`
- `someobj.hasOwnProperty(prop)`

1.5.12 The `Object.assign` Function

Copies properties from one object to another:

```
var o1 = {a: 1, b: 2, c: 3};
var o2 = { };

Object.assign(o2, o1);
console.log(o2);
```


Produces this output:

```
{ a: 1, b: 2, c: 3 }
```

(This function first appeared in ECMAScript Edition 6, 2015.)

1.6 Builtin Objects

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

- `str.length`
- `str.charAt(i);`
- `str.concat();`
- `str.indexOf(needle);`
- `str.slice(iStart, iEnd);`
- `str.substr(iStart, length);`
- `str.replace(regex|substr, newSubStr|function);`
- `str.toLowerCase();`
- `str.trim();`

1.6.3 The Number Object

- 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();`

1.6.4 The Math Object

- Constants:
  - `Math.E`
  - `Math.LOG2E`
  - `Math.PI`
  - etc.
- Generic Functions:
  - `Math.abs(n);`
  - `Math.pow(n, e);`
  - `Math.sqrt(n);`
  - etc.

1.6.5 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
```

1.6.6 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);
```

1.6.7 The Array Object

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

1.6.8 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" :(
```

1.6.9 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();`

1.6.10 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]
}
```


Chapter 2

Functions

2.1 Functions

2.1.1 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

2.1.2 Defining a Function

There are several ways of defining functions:

- Function statements (named functions)
- Function expression (anonymous functions)
- Arrow functions (new in ES2015)

2.1.3 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.

2.1.4 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

2.1.5 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`

2.1.6 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.
```

2.1.7 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
```

2.1.8 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!
};
```

2.1.9 Be Careful with Your Line Breaks

This:

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

Turns into:

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

Instead, write:

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

2.1.10 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

2.1.11 Rules for Using the `arguments` Variable

- Access all arguments, even unnamed ones
- Array-like, but not an actual array
- Only has `length` property
- Should be treated as read-only (never modify!)
- To treat like an array, convert it to one

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

or, slightly more popular (but wasteful):

```
var arr = [].slice.call(arguments);
```

or, with ES6:

```
var args = Array.from(arguments);
```

2.1.12 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)

2.1.13 Exercise: Function Arguments and Parsing

1. Open the following file:

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

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

2.1.14 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**

2.1.15 Example: Identify the Scope For Each Variable

```
var a = 5;

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

  if (d === c) {
    var e = "error: wrong number";
    console.log(e);
  }

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

- 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`, `e`, and `f` 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

2.1.16 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

2.2 Variable Hoisting

When using the `var` keyword, only functions can introduce a new variable scope. This leads to something known as hoisting.

2.2.1 Exercise: Hoisting (Part 1 of 2)

What will the output be?

```
function foo() {
  x = 42;
  var x;

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

2.2.2 Answer: Hoisting (Part 1 of 2)

This:

```
function foo() {
  x = 42;
  var x;

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

Turns into:

```
function foo() {
  var x;
  x = 42;

  console.log(x);
  return x;
}
```

2.2.3 Exercise: Hoisting (Part 2 of 2)

And this one?

```
function foo() {
  console.log(x); // ?
  var x = 42;
}
```

2.2.4 Answer: Hoisting (Part 2 of 2)

This:

```
function foo() {
  console.log(x); // ?
  var x = 42;
}
```

Turns into:

```
function foo() {
  var x;
  console.log(x);
  x = 42;
}
```

2.2.5 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 x=0, y;

// Is interpreted as:
var x=undefined, y=undefined;
x=0;
```

2.3 Functional Programming with Arrays

2.3.1 Introducing Higher-order Functions

The `forEach` function is a good example of a *higer-order* function:

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

a.forEach(function(val, index, array) {
  // Do something...
});
```

Or, less idiomatic:

```
var f = function(val) { /* ... */ };
a.forEach(f);
```

2.3.2 Array Testing

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


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

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

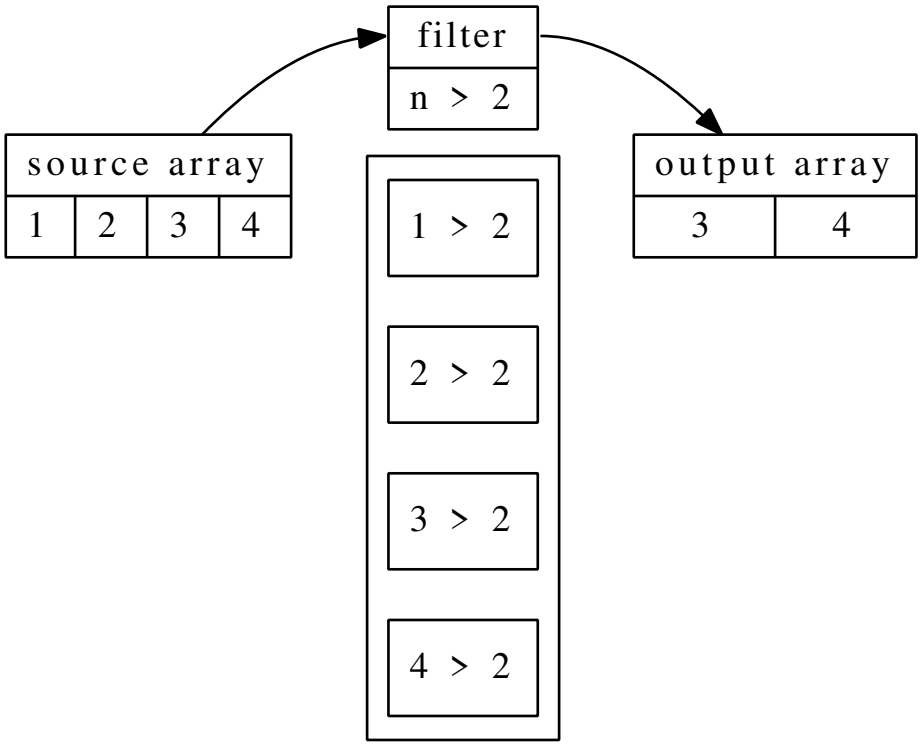
- Test if a function returns `true` at least once:

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

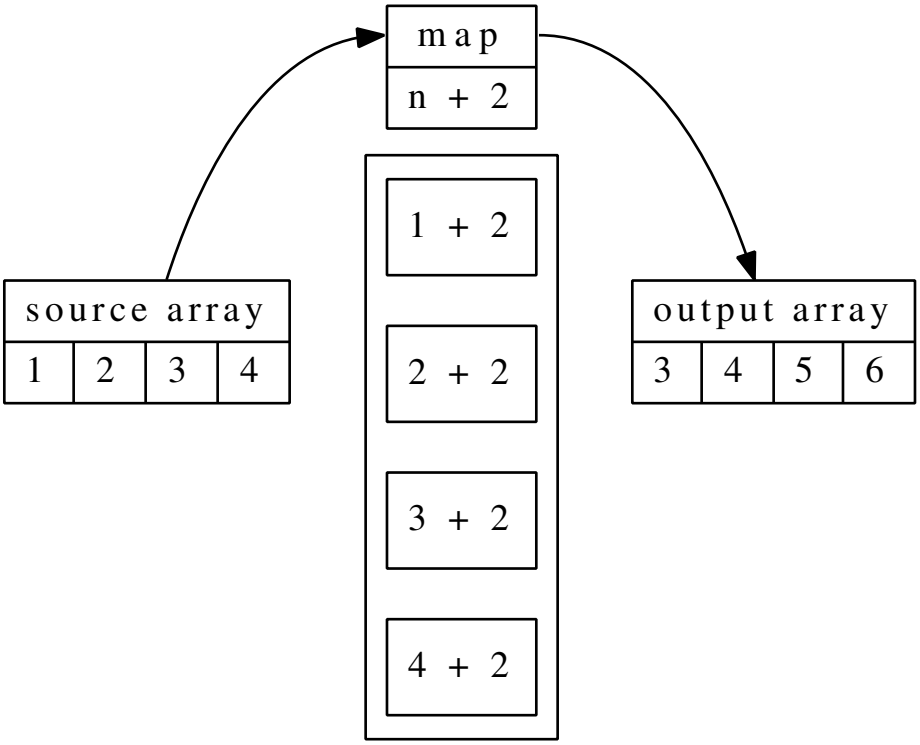
2.3.3 Higher-order Array Functions

- `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`

2.3.4 Filtering an Array with a Predicate Function



2.3.5 Mapping a Function Over an Array



2.3.6 Example: Folding an Array with `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);
```

2.3.7 Exercise: Arrays and Functional Programming

1. Open the following file:

   `src/www/js/array/array.js`

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Hint: Use https://developer.mozilla.org/ for documentation.

2.4 Common Patterns Involving Functions

2.4.1 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

2.4.2 Anonymous Functions (Tips)

- Name your anonymous functions for debugging

```
numbers.forEach(function foo(e) {
  console.log(e);
});
```

- Name is scoped to the inside of the anonymous function so it can refer to itself, easier to debug; errors reference the function name

2.4.3 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();
}
```


- Functions that take functions as arguments are called *higher-order* functions.

2.4.4 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);
```

2.5 Function Closures

2.5.1 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

2.5.2 Closures: Definitions

- Bound variable: local variables created with `var` or `let` are said to be *bound*.
- Free variable: Any variable that isn't bound and isn't a global variable is called a *free* variable.
- A function that uses free variables *closes around* them, capturing them in a *closure*.
- A closure is a new scope for free variables.

2.5.3 Closures: Example

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



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

  return inner;
}

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

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

See: `src/examples/js/closure.html`

2.5.4 Closures: Practical Example

```
var Foo = function() {

  var privateVar = 42;

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

  return {
    getPrivateVar: getter,
  };
};

var x = Foo();
x.getPrivateVar(); // 42
```

2.5.5 Exercise: Sharing Scope

1. Open the following file:

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

2. Complete the exercise.

3. Run the tests by opening the `index.html` file in your browser.

2.5.6 Loops 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!");
```


Chapter 3

Object-Oriented Programming

3.1 Scope and Context

3.1.1 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

3.1.2 Context Example

```
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();
```

3.1.3 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

3.1.4 How JavaScript Sets the `this` Variable

- Resides in the global binding
- Inner functions do not capture parent's `this` (there are several workarounds such as `var self = this;`, `bind`, and ES6 arrow functions)
- The `this` object can be set manually! (Take a look at the `call`, `apply`, and `bind` functions.)

3.2 Constructor Functions

3.2.1 Constructor Functions and the `new` Operator

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

```
var m = new Message("pjones@devalot.com", "Hello");
m.send(); // calls `Message.prototype.send'
          // with `this' set to `m'
```

3.2.2 Writing a Constructor Function

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



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

3.2.3 The `new` Keyword

```
var m = new Message("pjones@devalot.com", "Hello");
m.send(); // calls `Message.prototype.send'
          // with `this' set to `m'
```

The `new` operator does the following:

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

3.2.4 Implementing a Fake `new` Operator

```
var fakeNew = function(func) {
  // Step 1. Create an object with proper inheritance:
  var newObject = Object.create(func.prototype);

  // Step 2. Invoke the constructor:
  func.call(newObject);

  // Step 3. Return the new object:
  return newObject;
};
```

3.2.5 Exercise: Constructor Functions

1. Open the following file:

   `src/www/js/constructors/constructors.js`

2. Complete the exercise.
3. Run the tests by opening the `index.html` file in your browser.

Chapter 4

Debugging

4.1 Debugging in the Browser

4.1.1 Introduction to Debugging

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

4.1.2 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)`

4.1.3 Accessing the Debugger

- In the browser's debugging window, choose **Sources**
- You should be able to see JavaScript files used for the current site

4.1.4 Setting Breakpoints

There are a few ways to create breakpoints:

- Open the source file in the browser and click a line number
- Right-click the line number to create conditional breakpoints
- Use the `debugger;` statement in your code

4.1.5 Stepping Through Code

- After setting breakpoints, you can reload the page (or trigger a function)
- Once the debugger stops on a breakpoint you can step through the code using the buttons in the debugger
  - Step In: Jump into the current function call and debug it
  - Step Over: Jump over the current function call
  - Step Out: Jump out of the current function

4.1.6 Console Tricks

- `$_` the value of the last evaluation
- `$0`—`$4` last inspected elements in historical order
- `$("selector")` returns first matching node (CSS selector)
- `$$("selector")` returns all matching nodes
- `debug(function)` sets a breakpoint in `function`
- `monitor(function)` trace calls to `function`

See the Chrome Command Line Reference for more details.

Chapter 5

Exceptions

5.1 Exception Handling

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

5.1.1 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

5.2 Throwing Exceptions

5.2.1 Example: Throwing an Exception

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

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

5.3 Exception Objects

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

5.3.1 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`

5.3.2 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!");
```

5.3.3 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;
```


5.4 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.

5.4.1 Example: Catching Errors

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

5.4.2 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.
    }
  }
};
```

5.4.3 Exercise: Exceptions

1. Open the following file:

   `src/www/js/exceptions/exceptions.js`

2. Complete the exercise.

3. Run the tests by opening the `index.html` file in your browser.

Chapter 6

Wrapping Up

6.1 Best Practices

6.1.1 JavaScript Language Best Practices

1. Use a linting tool such as [JSHint][] or [ESLint][]
2. Avoid polluting the global namespace
3. Always use semicolons (`;`)
4. Prefer `===` and `!==` (strict comparison)
5. `CamelCase` the names of constructor functions
6. Don't use the `eval` function
7. Never modify the `arguments` object
8. Don't use `for...in` loops with arrays
9. Avoid monkey-patching standard objects

JavaScript Resources

JavaScript Documentation

- Mozilla Developer Network

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

Training Videos from Pluralsight

Beginner to Intermediate

- Basics of Programming with JavaScript
- JavaScript Fundamentals
- Building a JavaScript Development Environment
- JavaScript: From Fundamentals to Functional JS

Intermediate to Advanced

- Object-oriented Programming in JavaScript
- Reasoning About Asynchronous JavaScript
- Advanced JavaScript
- TypeScript Fundamentals
- Angular 2: Getting Started

Libraries

- Testing: [Jasmine][], [JSPec][], [Sinon][], and [Chai][]

Compatibility Tables

- ES6 Status By kangax

