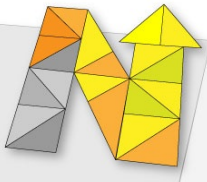


NextStep IT Training

powered by Smallrock Internet

Advanced JavaScript
for Frameworks

NS40305

Table of Contents

Copyright

ISBN:

999 Ponce de Leon Boulevard, Suite 830
Coral Gables, FL 33134
786-347-3155
http://nsitt.com

Chapter 1 - JavaScript Review

NodeJS and JavaScript
Functions, Scope, and iifes
Debugging

Objectives

- Introduce Node.js
- Review JavaScript basics and introduce some new syntax
- Debug JavaScript under NodeJS

Overview

This chapter establishes a solid foundation for the advanced JavaScript features that follow. The goal is to make sure that JavaScript may be run from NodeJS, the reader has the understanding of functions, variable scope, immediately invoked function execution syntax, and that the reader can enter and use the debugger in the web browser.

NodeJS and JavaScript

Node.js and JavaScript
Functions, Scope, and iifes
Debugging

Node.js

```
$ node helloworld.js
Hello, World!
$ node
> x = 42
42
> x
42
> console.log(x)
42
undefined
> console.log('Hello, World')
Hello, World
undefined
```

- JavaScript engine without the browser
- Launch JavaScript files at the command line
- Read-execute-print-loop

Single-Threaded

- JavaScript is single threaded
- Nothing else happens in a browser when your code executes
- Quickly respond to an event

Data Types

```
> b = true
true
> typeof b
'boolean'
> n = 42
42
> typeof n
'number'
> r = 42.5
42.5
> typeof r
'number'
> s = 'Hello, World!'
'Hello, World!'
> typeof s
'string'
```

```
> d = new Date()
2018-05-02T11:47:04.604Z
> typeof d
'object'
> d instanceof Date
true
> d instanceof String
false
>
d2 = new Date
2018-05-02T11:47:26.666Z
```

- Boolean, numbers, strings, and objects
- Built-in objects: Array, Date, Map, Math, etc.
- instanceof, typeof

Math Object

```
> Math.PI
3.141592653589793
> Math.abs(-42)
42
> Math.ceil(Math.PI)
4
> Math.floor(Math.PI)
3
> Math.max(9, 10)
10
> Math.pow(2, 3)
8
> Math.sqrt(25)
5
```

- Collection of operations and values

Operators

```
> 1 + 2
3
> 1 + 2 * 3
7
> (1 + 2) * 3
9
> 5 / 2
2.5
> '2' * '3'
6
> '2' * 3
6
> '2' + 3
'23'
```

- +, -, *, /, %
- Normalizes operands
- String concatenation

Combined Assignment Operators

```
x = 5
x *= 2 // same as x = x * 2
x *= 2 + 1 // same as x = x * (2 + 1)
```

- Combined assignment operators are a shortcut

Comparison Operators

```
let x = 5
let y = '5'

console.log(x == y) // true
console.log(x === y) // false

console.log(x != y) // false
console.log(x !== y) // true

console.log(x >= y)
console.log(x <= y)
```

- === and !== are never true if the data types are different

if Statement

```
let x = 42

if (x) {

    console.log('x === ' + x)

} else {

    console.log('not x')
}
```

```
$ node if.js
x === 42
```

- Any "zero" value is false: number, empty string, null
- Any "non-zero" value is true

- Assignments can trip up conditional logic

switch Statement

```
let x = 5

switch (x) {

    case 1:
        console.log('1')
        break

    case 2:
    case 3:
    case 4:
        console.log('2 or 3 or 4')
        break

    case '5':
        console.log("'5'")
        break

    case 5:
        console.log(5)
        break

    default:
        console.log('something else')
}
```

```
$ node switch.js
5
```

- First exact match
- Works with boolean values, numbers, and strings
- Normalization rules *not* in effect!

While

```
let a = [ 1, 2, 3 ]

console.log('while:')

let i = 0

while (i < a.length) {

    console.log('a[' + i + '] == ' + a[i])
    ++i
}

console.log('do ...while:')
```

```
i = 0

do {

    console.log('a[' + i + '] == ' + a[i])
    ++i

} while (i < a.length)
```

```
$ node while.js
while:
a[0] == 1
a[1] == 2
a[2] == 3
do ...while:
a[0] == 1
a[1] == 2
a[2] == 3
```

- Arrays are objects with numeric indexes

for

```
let a = [ 1, 2, 3 ]

for (let i = 0; i < a.length; ++i) {

    console.log('a[' + i + '] == ' + a[i])
}
```

```
$ node for.js
for loop:
a[0] == 1
a[1] == 2
a[2] == 3
for ... of:
v == 1
v == 2
v == 3
```

- For loop puts everything at the top

for ...of

```
let a = [ 1, 2, 3 ]

for (let v of a) {

    console.log('v == ' + v)
}
```

```
$ node for-of.js
for loop:
a[0] == 1
a[1] == 2
a[2] == 3
for ... of:
v == 1
v == 2
v == 3
```

- JavaScript 2015 adds a for ...of loop for iterable objects

Functions, Scope, and iifes

NodeJS and JavaScript
Functions, Scope, and iifes
Debugging

Functions

```
let x = 5
let y = square(x)

console.log('x == ' + x + ', y == ' + y)

function square(v) {

    v = v * v
    return v
}
```

```
$ node functions.js
x == 5, y == 25
```

- Functions support the DRY principle
- Accept parameters, may return a value
- Parameters are copies of a value

Scope and *var*

```
let a = 5
let x = 5
let y = square(x)

console.log('a == ' + a + ', b == ' + b + ', x == ' + x + ', y == ' + y)

function square(v) {

    a = v * v
    b = v * v
    var x = v * v

    return x
}
```

```
$ node scope.js
a == 25, b == 25, x == 5, y == 25
```

- Parameters have function scope
- Variables declared with *var* have function scope

let

```
let a = 5
let x = 5
let y = square(x)

console.log('a == ' + a + ', b == ' + b + ', x == ' + x + ', y == ' + y)

function square(v) {

    {
        a = v * v
        b = v * v
        let x = v * v
    }

    return x  // uses the global x!
}
```

```
$ node let.js
a == 25, b == 25, x == 5, y == 5
```

- JavaScript 2015 added *let* with block scope
- y is 5 because the return used global x, which is 5

const

```
const x = 5
// x = 6 // not allowed
```

- *const* is the same as *let*, except constant

Immediately Invoked Function Execution

```
let a = 42
let x = 43;

( function () {

    let a = 5
    let x = 5
    let y = square(x)

    console.log('a == ' + a + ', b == ' + b + ', x == ' + x + ', y == ' + y)

    function square(v) {
```

```
        {
            a = v * v
            b = v * v
            let x = v * v
        }

        return x  // uses the global x!
    }

} ).call()

console.log('a == ' + a + ', x == ' + x)
```

```
$ node iife.js
a == 25, b == 25, x == 5, y == 5
a == 42, x == 43
```

- *var* and *let* outside a function still have "global" scope
- Create a scope by wrapping code in a function
- Wrap the function in () and immediately call it

Global

- Nothing is truly global
- global variables and functions are properties of the *global* object
- In the browser, window is the global object

Debugging

NodeJS and JavaScript
Functions, Scope, and iifes
Debugging

Debugging

- Launch in a debugger; Visual Studio Code or in the Browser
- Explore setting breakpoints and watch expressions
- Look at variables and stack traces

Checkpoint

- What does Node.js offer?
- What scope does *var* have? Does *let* have?
- When dividing 5 / 2, what is the result?
- What does "single-threaded" mean?
- What data types does JavaScript have?
- Which loop structure is best for indexing arrays?

Chapter 2 - Strings and Template Strings

Strings and Methods
Regular Expressions
JavaScript Object Notation

Objectives

- Review String object basics
- Use JavaScript expressions in Template Strings
- Use regular expressions to check for matches in strings

Overview

Run through the basics of JavaScript strings, regular expressions, the new template strings from JavaScript 2015, and how to move objects into and out of JSON notation.

Strings and Methods

Strings and Methods
Regular Expressions
JavaScript Object Notation

String Literals & Special Characters

Escape Sequence	Meaning
\n	newline (line-feed)
\r	carriage return
\t	tab
\\	back-slash

- Double-quotes and single-quotes are equal in JavaScript; single-quotes are usually preferred in JavaScript and double-quotes in HTML
- Use \ to insert special characters and escape others

Operators and Normalization

```
let n = 5
let s = '5'

console.log(n + s) // 55
console.log(n * s) // 25
console.log(s * 1 + n) // 10
```

- A + with a string on either side performs concatenation
- Any other operator attempts to parse the string as a number

String Equality

```
console.log("abc" === "abc") // true
```

- == and === may always be used to check strings
- Strings are cached, and only one copy of each exists
- That includes strings read while the program is running

String Methods

Method	Description

!endsWith(text)|Returns true if the string ends with the text |indexOf(text)|The zero-based position of the text, or -1| |match(regex)|Returns a match object from the search| |repeat(count)|Return the string repeated the string count times| |replace(regex, value)|Replace all occurrences which match the pattern| |search(regex)|Returns the index of the text that matches the pattern, or -1| |startsWith(text)|Returns true if the string starts with the text|

- Strings are instances of the *String* class
- The length property is the number of characters
- Strings are immutable: methods returning a string return a new string

Template Strings

```
Employee e = Employees.find(123)

let message = `Employee ${e.id} is paid ${weeklySalary / 52} weekly`
```

- Template strings have expressions
- The expressions are evaluated once when the string is defined
- Template strings are defined in back-ticks (grave accents)

Separation of Concerns

```
Employee e = Employees.find(123)

let weeklySalary = e.salary / 52
let message = `Employee ${e.id} is paid ${weeklySalary} weekly`
```

- Strings are presentation
- Do not mix business logic and formatting

Template String Evaluation

```
Employee e = Employees.find(123)

e.salary = 52000

let weeklySalary = e.salary / 52
let message = `Employee ${e.id} is paid $${weeklySalary} weekly`

console.log(message) // ...$1000

let weeklySalary *= 1.10

console.log(message) // ...$1000
```

- The template string is evaluated once
- Changing a variable that affects the expression does not change the string

Template Strings and Expressions

Strings and Methods
Regular Expressions
JavaScript Object Notation

Regular Expressions

```
let p = /abc/

Console.log('Contains abc'.search(p)) // 9
```

- Strings are compared to patterns
- Patterns define what can match in each position
- The whole string does not have to match, just one segment

Flags

```
let p = /abc/igm
```

- i: ignore case
- g: find all matches
- m: search across newlines

Character Sets

```
console.log('abc'.search(/abc/)) // true
console.log('abc'.search(/a[a-z]c/)) // true
console.log('abc'.search(/a[A-Z]c/)) // flase
console.log('abc'.search(/a.c/)) // true
```

- Brackets define a set for one position
- The set may be a list of characters or a list of ranges
- . is the set of all characters

Metacharacters

Symbol	Description
\w	Word character: alhpanumeric and underscore
\W	Non-word character

\s	Whitespace
\S	Non-whitespace
\d	Digit
\D	Non-digit

Expression Modifiers

```
console.log('abc'.search(/a*/)) // true
console.log('abc'.search(/a+/)) // true
console.log('abc'.search(/x?/)) // false
```

- Modifiers influence what comes immediately in front
- * matches zero or more times, + matches one or more times, ? matches zero or one times
- *, +, and ? are shortcuts for {m, n}

{m, n} Modifiers

- {m, n} matches at least m, at most n times
- {m, } matches at least m times
- {m} matches exactly m times

Anchors

```
console.log('abc'.search(/^b/)) // false
console.log('abc'.search(/c$/)) // true
```

- ^ anchors the pattern to the beginning of the string
- $ anchors the pattern to the end of the string

Or Condition

```
console.log('abc'.search(/ABC|abc/) // true
```

- The pipe symbol | performs an or operation

Groups

```
console.log('abc'.search(/a(BC|bc))) // true
```

- Groups are useful to match a small part with an or

- Groups remember the text that matched

Matching Results

```
let match = 'abc abc'.match(/a(.)b/)

console.log(match) // [ 'abc', 'b', index: 0, input: 'abc abc' ]
```

- match[0] is the text matching the whole pattern
- match[1] is the first group, etc.

Matching with the *g* Flag

```
let match = 'abc abc'.match(/a(.)b/g)

console.log(match) // [ 'abc', 'abc' ]
```

- Returns a list of all occurances text matchng the pattern globally

Replacement (with *g*)

```
let newString = 'abc axc'.replace(/a(.)c/g, /A$1B/)

console.log(newString) // 'AbB AxB'

let swapped = 'John Smith'.replace(/(.*) (.*)/, '$2, $1')

console.log(swapped) // 'Smith, John'
```

- Replace matching text with new text
- $n resolves to the nth group in the match of the expression

Chapter 3 - Functions

Function Definitions
Callbacks and Closures
Arrow Functions

Objectives

- Review the syntax for defining functions
- Explore using functions as callbacks
- Understand the drawbacks and benefits of closures, and how arrow functions fit

Overview

Clarify the definition of functions in JavaScript, including defining and using closures and inline functions passed as parameters in another function call. JavaScript 2015 introduces another syntax for inline functions, *arrow functions.*

Function Definitions

Function Definitions
Arrow Functions

Function Definitions

```
function add(a, b) {

    return a + b
}
```

arguments

```
function f(a, b) {

    console.log(arguments.length)
    console.log(arguments)
}
```

- Not enough arguments, the remainder are undefined
- Too many arguments, the extras are ignored
- The arguments object contains all of the arguments passed

arguments and *strict*

```
"use strict";

function f(a, b) {

    arguments = [ a, b ] // fails
}

f(1, 2)
```

- If "use strict" is declared arguments cannot be changed or replaced

Rest Operator

```
function f(a, b, ...c) {

    console.log(`a: ${a}, b: ${b}, c: ${c}`)
}

f(1, 2, 3, 4)
```

```
```

- JavaScript 2015 addition: all remaining arguments are in the array "c"

Spread Operator

```
function f(a, b, c) {

    console.log(`a: ${a}, b: ${b}, c: ${c}`)
}

let args = [ 1, 2, 3 ]

f(...args)
```

- JavaScript 2015 addition: arrays can be spread out over a list of parameters

Default values

```
function f(a, b, c = 5) {

    console.log(`a: ${a}, b: ${b}, c: ${c}`)
}

f(1, 2)
```

Destructuring Arrays

```
function f() {

    return [ 1, 2, 3 ]
}

let [ a, b, c ] = f()
let [ x, , y ] = f()

console.log(a, b, c) // 1, 2, 3
console.log(x, y) // 1, 3
```

- Related to the spread operator
- Assign from array elements to variables in one statement
- Elements can be skipped

Fail-safe Destructuring

```
function f() {
```

```
    return [ 1, 2, 3 ]
}

let [ a = 97, b = 98, c = 99, d = 100, e = 101, f ] = f()

console.log(a, b, c, d, e, f) // 1, 2, 3, 100, 101, undefined
```

- Assignment variables may be assigned default fail-safe values

Destructuring Objects

```
function f() {

    return { one: 1, two: 2 }
}

let { one, two } = f()

console.log(one, two) // 1, 2
```

- Extract named properties into variables

Deep Destructuring Objects

```
function f() {

    return { one: 1, two: 2, three: { a: 'a', b: 'b', c: 'c' } }
}

let { one: o, two: t, three: { a: x, b: y, c: z } } = f()

console.log(o, t, x, y, z ) // 1, 2, 'a', 'b', 'c'
```

- Variables can be renamed using : notation

Function Objects

```
function f(a, b) {

}

console.log(f.length)
```

- Function are references to objects (more later)
- The *length* property is the number of expected arguments

"Global" Space

```
function f(a, b) {

    return a + b
}

console.log(f === window.f) // true
```

- JavaScript attaches functions to the "global" object; in the browser that is the window object
- Creating Global identifiers runs a risk of collisions
- That is worse when the collision is with existing properties of the window object

Callbacks and Closures

Function Definitions
Callbacks and Closures
Arrow Functions

Callbacks and Closures

```
function f() {

    console.log('Function f')

    function t() {

        console.log('Function t')
    }

    setTimeout(t, 1000)

    console.log('Function f: timer set')
}

f()
console.log('After function f call')
```

```
$ node closure-1.js
Function f
Function f: timer set
After function f call
Function t
```

- Closures are functions defined in another function
- Closures only exist in the scope of the enclosing function

Inline Definitions

```
function f() {

    setTimeout(function () {

        console.log('Timer expired')

    }, 1000)
}

f()
```

```
$ node closure-2.js
```

```
Timer expired
```

- A function definition is a reference to the function-object
- As such, it can be defined inline in where a function reference is needed

Variables in Scope

```
function f(message, timeout) {

    setTimeout(function () {

        console.log(message)

    }, timeout)
}

f('Timer expired', 1000)
```

```
$ node closure-3.js
Timer expired
```

- Variables in scope are bound to the closure and available when the closure runs

Variables are Shared

```
function f(message, timeout) {

    var localvariable = 10

    setTimeout(function () {

        console.log(message, `First timer localvariable ===  ${localvariable}`)
        localvariable++

    }, timeout)

    setTimeout(function () {

        console.log(message, `Second Timer localvariable === ${localvariable}`)
        localvariable++

    }, timeout + 1000)
}

f('Timer expired', 1000)
```

```
$ node closure-4.js
Timer expired First timer localvariable ===  10
Timer expired Second Timer localvariable === 11
```

- Variables in scope are shared between closures in the same scope

Arrow Functions

Function Definitions
Callbacks and Closures
Arrow Functions

Arrow Functions

```
function f(message, timeout) {

    var localvariable = 10

    setTimeout(() => {

        console.log(message, `First timer localvariable ===  ${localvariable}`)
        localvariable++

    }, timeout)

    setTimeout(() => {

        console.log(message, `Second Timer localvariable === ${localvariable}`)
        localvariable++

    }, timeout + 1000)
}

f('Timer expired', 1000)
```

```
$ node closure-4.js
Timer expired First timer localvariable ===  10
Timer expired Second Timer localvariable === 11
```

- JavaScript 2015 introduces a new function syntax: *Arrow Functions*
- Arrow functions may only be closures; they cannot be assigned to a prototype property
- The closure is defined as *parameters* => *the function body*

Arrow Function Parameters

```
p1 => {

    return p1 * 2
}

(p1, p2) => {

    return p1 * p2
}
```

- Parenthesis around the parameters are necessary only if there are no parameters or multiple parameters

Arrow Function Body

```
let f = p1 => p1 * 2

console.log(f(5)) // 10
```

- If there is only one statement without braces the result automatically becomes the closure return value

Checkpoint

- How are functions stored in JavaScript?
- What does the rest operator do?
- Must object properties be assigned into variables of the same name?
- What happens if there are more variables than elements while destructuring an array?
- What is the most significant feature of a closure?
- Why are arrow functions preferred over regular functions for closures?

Chapter 4 - Objects

Objects and Polymorphism
Object-Oriented Programming

Objectives

- Create objects and collections of objects
- Explore duck-typing to work with collections
- Transfer objects using JavaScript Object Notation

Overview

JavaScript is both a procedural and object-oriented language. All of the built-in features are objects, and JavaScript allows the program to build its own objects. To effectively use OOP, we need explore some of the features and principles that drive it.

Objects and Polymorphism

Objects and Polymorphism
Object-Oriented Programming

Literal Objects

```
var employee = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000
}

console.log(employee)
console.log(`${employee.name}'s hire date is ${employee.hiredate}}`)

employee.notes = 'Very good leadership skills'
console.log(employee)
```

```
$ node literals.js
{ name: 'John Smith',
  hiredate: 2003-07-01T00:00:00.000Z,
  salary: 52000 }
John Smith's hire date is Mon Jun 30 2003 20:00:00 GMT-0400 (EDT)}
{ name: 'John Smith',
  hiredate: 2003-07-01T00:00:00.000Z,
  salary: 52000,
  notes: 'Very good leadership skills' }
```

- JavaScript allows literal objects to be made at any time
- Use properties through the property accessor operator (.)
- New properties may be added at any time

Subscript Notation

```
var employee = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000
}

console.log(employee)
console.log(`${employee.name}'s hire date is ${johnsmith['hiredate']}}`)

employee['employee notes'] = 'Very good leadership skills'
console.log(employee)
```

```
```text
```
```

```
$ node subscripts.js
{ name: 'John Smith',
  hiredate: 2003-07-01T00:00:00.000Z,
  salary: 52000 }
John Smith's hire date is Mon Jun 30 2003 20:00:00 GMT-0400 (EDT)}
{ name: 'John Smith',
  hiredate: 2003-07-01T00:00:00.000Z,
  salary: 52000,
  'employee notes': 'Very good leadership skills' }
```

- Objects properties may be accessed using []
- Property names do not have to be legal identifiers
- Non-legal identifiers only work with []

Enumerating Object Properties

```
var employee = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000
}

for (property in employee) {

    console.log(`${property}: ${johnsmith[property]}`)
}
```

```
$ node forIn.js
name: John Smith
hiredate: Mon Jun 30 2003 20:00:00 GMT-0400 (EDT)
salary: 52000
```

- Use for ...in to iterate through object properties

Function References (Methods)

```
var employee = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000,
    calculatePay: function () { return this.salary / 52 }
}

console.log(`${employee.name}'s weekly pay is ${employee.calculatePay()}`)
```

```
$ node methods.js
John Smith's weekly pay is 1000
```

- A property may be a function reference, a "method"
- "this" references the object when a function is called
- Prototypal methods cannot be defined as arrow functions

Object-Oriented Programming

Objects and Polymorphism

Object-Oriented Programming

Encapsulation

```
var employee = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000,
    calculatePay: function () { return this.salary / 52 }
}
```

- Encapsulation is organization; putting related data and processes together
- Complex data structures may be managed as units
- Some languages, not JavaScript, include *data hiding* as part of *encapsulation*

Data Hiding

```
var employee = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000,
    calculatePay: function () { return this.salary / 52 }
}
```

- JavaScript has no concept of privacy; properties may be added, changed, or deleted at any time
- It used to be that "private" properties were named with a leading underscore
- That is generally discouraged now; there is more code stability in accepting that everything is public

Polymorphism is Key

```
var employee1 = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000,
    calculatePay: function () { return (this.salary / 52).toFixed(2) }
}

var employee2 = {
    name: 'John Rolfe',
    salary: 68000,
    calculatePay: function () { return (this.salary / 52).toFixed(2) }
}

var employees = [ employee1, employee2 ]
```

```
for (let employee of employees) {

    console.log(`Employee ${employee.name} weekly salary === $${employee.calculatePay()}`)
}
```

```
$ node polymorphism.js
Employee John Smith weekly salary === $1000.00
Employee John Rolfe weekly salary === $1384.62
```

- Client code sends messages to objects directing them to do something
- Computer programs implement this as method calls
- Similar objects may be grouped together, the client does not need to know their type

Duck Typing

```
var employee1 = {
    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000,
    calculatePay: function () { return (this.salary / 52).toFixed(2) }
}

var employee2 = {
    name: 'John Rolfe',
    salary: 68000,
    calculatePay: function () { return (this.salary / 52).toFixed(2) }
}

var employees = [ employee1, employee2 ]

for (let employee of employees) {

    console.log(`Employee ${employee.name} weekly salary === $${employee.calculatePay()}`)
}
```

- JavaScript does not compile-time check the method calls, it uses duck-typing for polymorphism
- Method calls invoke the function bound to that object with that name
- If the property exists, the operation succeeds

Substitution

```
let intuitProcessor = {
    authorize: function (amount, cardDetails) { // Authorize with Intuit Merchant Services }
}

let jpmcProcessor = {
    authorize: function (amount, cardDetails) { // Authorize with JP Morgan Chase Merchant Services }
}

let paymentProcessor = jpmcProcessor // or intuitProcessor!
```

- Similar objects may be substituted for each other; for example, lists of hourly and salary employee types
- Replacement is key: substitute a new payment processor object to use another vendor
- Adapting to changing merchant services is simple: make a new object with the same interface!

SOLID Object-Oriented Design Principles

|S|Single Responsibility| |O|Open for Extension, Closed for Modification| |L|Liskov Substitution Principle| |I|Interface Segregation| |D|Dependency Inversion|

- Single responsibility: programs, modules, objects, functions & methods, and statements should all be about one responsibility
- Open/closed: do not hack up existing code, extend an interface and substitute
- Liskov: substituted objects need to provide not just the same interface, but also the expected behavior
- Interface segregation: small, simple interfaces; apply single responsibility!
- Dependency inversion: do not decide which payment processor to use, let the client code tell you!

Checkpoint

- Does JavaScript stop you from modifying an object?
- What notations may be used to access an object property?
- What can be the value of a property?
- Should private properties be simulated?
- What is polymorphism?
- How does duck-typing work?

Chapter 5 - Classes and OOP

Class Definitions
Class Properties and Encapsulation
Class Inheritance

Objectives

- Use class definitions to create similar objects
- Understand the mechanics of class creation and use
- Implement DRY with inheritance

Overview

Most objects are created as one of a group, so using some form of a template to create them will speed up the process. Classes are a template; they define what properties a group of objects should have and every object created from the class looks and works the same.

Class Definitions

Class Definitions
Class Properties and Encapsulation
Class Inheritance

Class Definitions

```
class Employee {

    constructor(name, hiredate, salary) {

        this.name = name
        this.hiredate = hiredate
        this.salary = salary
    }
}

var employee = new Employee('John Smith', new Date('2003-07-01'), 52000)

console.log(employee)
```

```
$ node classes.js
Employee {
  name: 'John Smith',
  hiredate: 2003-07-01T00:00:00.000Z,
  salary: 52000
}
```

- The class definition may be used with the *new* operator to consistently create new objects
- Fields may only be defined in a method
- THe constructor method is called from *new*

Methods

```
class Employee {

    constructor(source) {

        this.name = ''
        this.hiredate = null
        this.salary = 0

        if (source) {

            Object.assign(this, source)
        }
    }

    calculatePay() {
```

```
        return this.salary / 52
    }
}

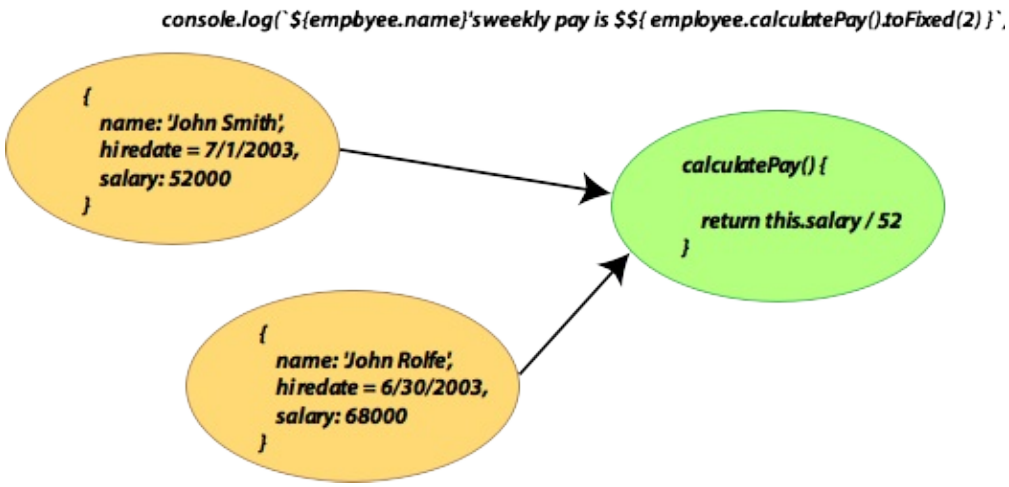
var employee = new Employee('John Smith', new Date('2003-07-01'), 52000)

console.log(`John Smith's weekly pay is $${ employee.calculatePay().toFixed(2) }`)
```

```
$ node methods.js
John Smith's weekly pay is $1000.00
```

- Method definitions are function definitions in the class
- May be referenced using the property accessor operators . and []
- Are attached to the *prototype*

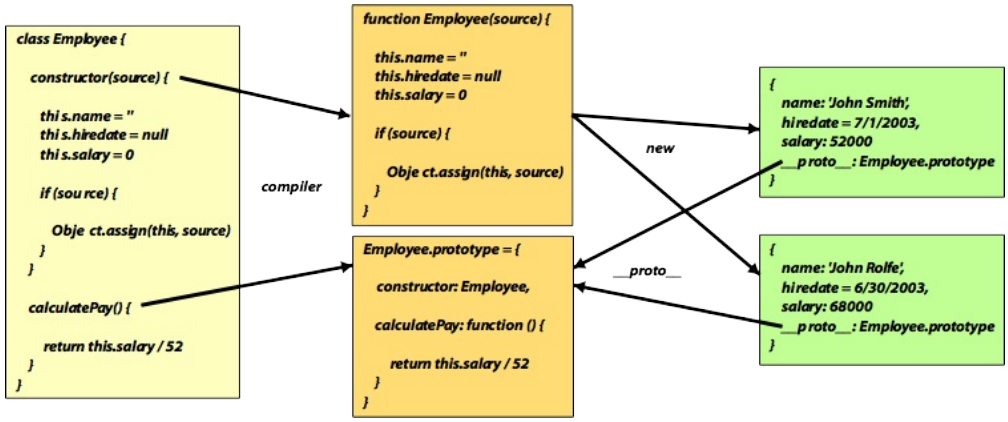
Prototypal Programming




```
John Smith's weekly pay is $1000.00
```

- Objects share methods by sharing a prototype
- When a member is not found, JavaScript looks to the prototype

Prototype Linkage



- The constructor becomes a function object, the prototype object is linked to it
- *new* creates a reference to the prototype object in the new object
- How this works is important, because there is framework syntax that uses it

Methods are Shared

```
class Employee {

    constructor(source) {

        this.name = ''
        this.hiredate = null
        this.salary = 0

        if (source) {

            Object.assign(this, source)
        }
    }

    calculatePay() {

        return this.salary / 52
    }
}

var employee1 = new Employee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })
var employee2 = new Employee({ name: 'John Rolfe', hiredate: new Date('2003-06-30'), salary: 68000 })

console.log(employee1)
console.log(`${employee1.name}'s weekly pay is $${ employee.calculatePay().toFixed(2) }`)

console.log(employee2)
console.log(`${employee2.name}'s weekly pay is $${ johnrolfe.calculatePay().toFixed(2) }`)

console.log(`employee1.calculatePay === employee2.calculatePay: ${ employee1.calculatePay === employee2.ca
lculatePay }`)
```

```
$ node shared-methods.js
Employee {
  name: 'John Smith',
  hiredate: 2003-07-01T00:00:00.000Z,
```

```
  salary: 52000 }
John Smith's weekly pay is $1000.00
Employee {
  name: 'John Rolfe',
  hiredate: 2003-06-30T00:00:00.000Z,
  salary: 68000 }
John Rolfe's weekly pay is $1307.69
employe1e.calculatePay === employee2.calculatePay: true
```

- Two objects, and the reference to the method is the same in each object

Static Members

```
class Employee {

    constructor(source) {

        if (source) {

            Object.assign(this, source)
        }
    }

    static employeeTaxRate() {

        return 0.25
    }
}

var employee = new Employee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })

console.log(`employee.employeeTaxRate: ${ employee.employeeTaxRate }`)
console.log(`Employee.employeeTaxRate: ${ Employee.employeeTaxRate }`)
```

```
$ node static.js
employee.employeeTaxRate: undefined
Employee.employeeTaxRate: employeeTaxRate() {

        return 0.25
    }
```

- Methods may be static, attached to the class instead of an instance
- Access static methods using the class name

Methods as Callbacks

```
class Employee {

    constructor(source) {

        if (source) {
```

```
            Object.assign(this, source)
        }

        this.employmentLength = this.employmentLength.bind(this)

        setInterval(this.employmentLength, 1000)
    }

    employmentLength() {

        console.log(`seconds employed: ${ Math.floor(((new Date()) - this.hiredate) / 1000) }`)
    }
}

var employee = new Employee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })
```

```
$ node callback-methods.js
seconds employed: 468645163
seconds employed: 468645164
seconds employed: 468645165
```

- Callbacks are not called through the object, so they loose the context for "this"
- "binding" the method to the instance is preferred over using arrow functions as callbacks even though a new function is created for every instance
- Better than creating a closure in a method, every time a method is called

Class Properties and Encapsulation

Class Definitions
Class Properties and Encapsulation
Class Inheritance

Properties

```
class Employee {

    constructor(source) {

        this.name = ''
        this.hiredate = null
        this.salary = 0

        if (source) {

            Object.assign(this, source)
        }
    }

    get salary() {

        return this._salary
    }

    set salary(value) {

        if (value < 0) {

            throw new RangeError('Salary must be >= 0')
        }

        this._salary = value
    }
}

var employee = new Employee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: -1 })
```

```
$ node properties.js
RangeError: Salary must be >= 0
...
```

- Properties are get and set methods, but act like fields to the client
- The methods provide opportunities to add constraints
- Properties provide encapsulation of data

Encapsulation

- Properties enhance encapsulation in a class

- Clients see the property, not what is behind it (use _ to imply privacy)
- What you can hide you can change

Properties vs Fields

- Only build properties for a reason: adding constraints
- In JavaScript, fields may always be replaced with a property of the same name

Object.assign and Properties

```
    set salary(value) {

        if (value < 0) {

            throw new RangeError('Salary must be >= 0')
        }

        this._salary = value
    }
}
```

- Object.assign utilizes properties
- The example through the exception, the setter must have been used

Reflect.setPrototypeOf

```
$ node spo-vs-assign.js
s.somethingToString = Something!
sPrime.somethingToString = Something!
Measurements between marks:
[ PerformanceEntry {
    name: 'A to B',
    entryType: 'measure',
    startTime: 74.276024,
    duration: 0.165578 } ]
[ PerformanceEntry {
    name: 'B to C',
    entryType: 'measure',
    startTime: 74.441602,
    duration: 0.061946 } ]
```

- setPrototypeOf changes the prototype of an object
- Expensive operation: A -> B is setPrototypeOf, B -> C is Object.assign

JSON and Properties

```
class Employee {

    constructor(source) {
```

```
        this.name = ''
        this.hiredate = null
        this.salary = 0

        if (source) {

            Object.assign(this, source)
        }
    }

    get salary() {

        return this._salary
    }

    set salary(value) {

        if (value < 0) {

            throw new RangeError('Salary must be >= 0')
        }

        this._salary = value
    }
}
```

- Object.assign works better
- setPrototypeOf requires JSON would need to provide the private fields for properties

Properties may be static

```
class Employee {

    constructor(source) {

        if (source) {

            Object.assign(this, source)
        }
    }

    static get employeeTaxRate() {

        return 0.25
    }
}

var employee = new Employee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })

console.log(`employee.employeeTaxRate: ${ employee.employeeTaxRate }`)
console.log(`Employee.employeeTaxRate: ${ Employee.employeeTaxRate }`)
```

```
$ node static-properties.js
employee.employeeTaxRate: undefined
Employee.employeeTaxRate: 0.25
```

Class Inheritance

Extending a class

```
class Base {

    toString() { return 'class Base' }
}

class A extends Base {
}

class B extends Base {
}

var a = new A()
var b = new B()

console.log(`a.toString(): ${ a.toString() }`)
console.log(`a instanceof A: ${ a instanceof A }`)
console.log(`a instanceof Base: ${ a instanceof Base }`)
console.log(`a instanceof B: ${ a instanceof B }`)

console.log(`a.toString(): ${ a.toString() }`)
console.log(`b instanceof B: ${ b instanceof B }`)
console.log(`b instanceof Base: ${ b instanceof Base }`)
console.log(`b instanceof A: ${ b instanceof A }`)
```

```
$ node extend.js
a.toString(): class Base
a instanceof A: true
a instanceof Base: true
a instanceof B: false
b.toString(): class Base
b instanceof B: true
b instanceof Base: true
b instanceof A: false
```

- An extended class inherits all the members of the superclass
- An object of the extended class is an instance of the superclass

Overriding Methods

```
class Base {

    toString() { return 'class Base' }
```

```
}

class A extends Base {

    toString() { return super.toString() + '; class A' }
}

class B extends Base {

    toString() { return 'class B; ' + super.toString() }
}

var a = new A()
var b = new B()

console.log(`a.toString(): ${ a.toString() }`)
console.log(`a instanceof A: ${ a instanceof A }`)
console.log(`a instanceof Base: ${ a instanceof Base }`)
console.log(`a instanceof B: ${ a instanceof B }`)

console.log(`b.toString(): ${ b.toString() }`)
console.log(`b instanceof B: ${ b instanceof B }`)
console.log(`b instanceof Base: ${ b instanceof Base }`)
console.log(`b instanceof A: ${ b instanceof A }`)
```

```
$ node override-methods.js
a.toString(): class Base; class A
a instanceof A: true
a instanceof Base: true
a instanceof B: false
b.toString(): class B; class Base
b instanceof B: true
b instanceof Base: true
b instanceof A: false
```

- Create a new method with the same name to override a method
- Call the superclass method through super

SuperClass Constructor

```
class Employee {

    constructor() {

        if (source) {

            Object.assign(this, source)
        }

        this.name = this.name ? this.name : ''
        this.hiredate = this.hiredate ? this.hiredate : null
    }
}

class SalaryEmployee extends Employee {

    constructor(source) {
```

```
        super(source)
        this.salary = this.salary ? this.salary : 0
    }

    calculatePay() {

        return this.salary / 52
    }
}

class HourlyEmployee extends Employee {

    constructor(source) {

        super(source)
        this.hourlyRate = this.hourlyRate ? this.hourlyRate : 0
        this.hoursPerWeek = this.hoursPerWeek ? this.hoursPerWeek : 0
    }

    calculatePay() {

        return this.hourlyRate * this.hoursPerWeek
    }
}
```

- The constructor must be called first
- Members must be initialized after the superclass constructor call
- But, the superclass assignment could have initialized the properties...

Inheritance is about DRY

- Some languages use inheritance to get to polymorphism
- JavaScript uses duck typing
- Inheritance is strictly about the DRY principle

Static Methods are Inherited

```
class Employee {

    constructor(source) {

        if (source) {

            Object.assign(this, source)
        }
    }

    static employeeTaxRate() {

        return 0.25
    }
}

class SalaryEmployee extends Employee {

}
```

```
var employee = new SalaryEmployee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })

console.log(`employee.employeeTaxRate: ${ employee.employeeTaxRate }`)
console.log(`SalaryEmployee.employeeTaxRate: ${ SalaryEmployee.employeeTaxRate }`)
console.log(`Employee.employeeTaxRate: ${ Employee.employeeTaxRate }`)
```

```
$ node static-inheritance.js
employee.employeeTaxRate: undefined
SalaryEmployee.employeeTaxRate: employeeTaxRate() {

        return 0.25
    }
Employee.employeeTaxRate: employeeTaxRate() {

        return 0.25
    }
```

- Static methods *are* inherited, an unusual language feature
- Behind the scenes the subclass function-object gets a copy of the superclass member referencing the static method

Checkpoint

- Why object-oriented programming?
- What makes OOP different in JavaScript?
- Explain how prototypes work
- Why use get and set methods for properties?
- What is the problem for using methods as callbacks?
- What is the DRY principle? How does inheritance support it?

Chapter 6 - Modules

Modules
Module Content

Objectives

- Explore the syntax of JavaScript 2015 modules
- Export and import multiple objects
- Discuss how modules should be organized

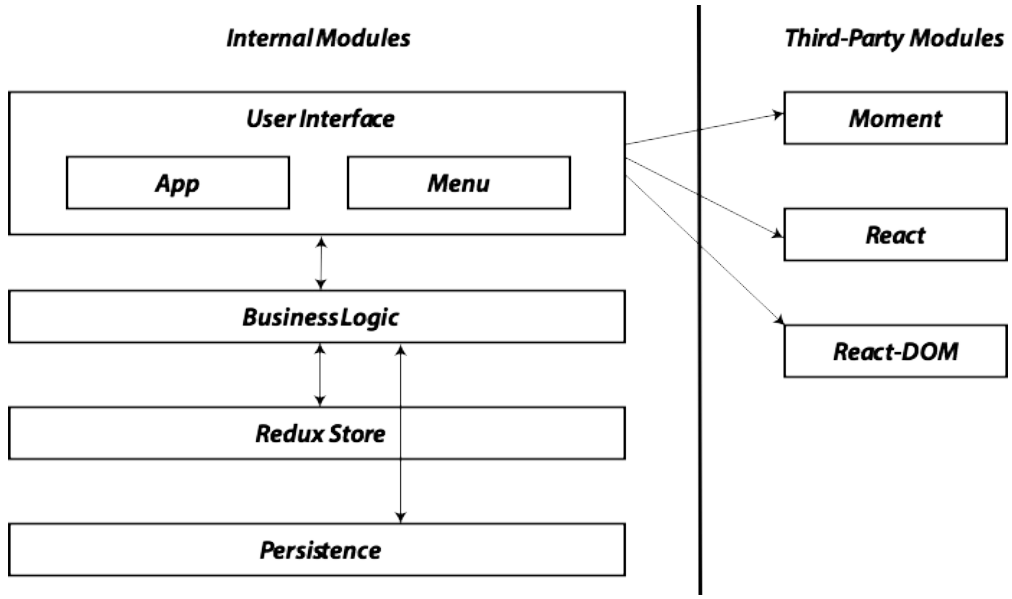
Overview

Robert Martin's *Single Responsibility* principle is about dividing up an application so that every part has a single responsibility. This works at many levels, the application itself has a single responsibility when viewed from a high level, and classes define objects that have their own individual responsibilities. Modules exist in-between: one module, made up of several classes, may have the responsibility of persisting data, another of managing the user interface. Modules offer both the means to group related things together as one unit, and a mechanism to share that unit across projects.

Modules

Modules
Module Content

Modules and Layers



- Modules are containers for objects, functions, and classes.
- Modules have scope, items must be explicitly exported
- Promote sharing and reuse

CommonJS Modules

```
( () => {

    class Employee {

        constructor(source) {

            if (source) {

                Object.assign(this, source)
            }

            this.name = this.name ? this.name : null
            this.hiredate = this.hiredate ? this.hiredate : null
        }
    }

    module.exports = Employee

} ).call()
```

```
var Employee = require('./Employee')

var employee = new Employee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })

console.log(employee)
```

- Native module format for Node.js
- The properties of "exports" is the public interface
- *iife* not necessary in NodeJS, but necessary in a browser to ensure scope

JavaScript 2015 Modules

```
export default class Employee {

    constructor(source) {

        if (source) {

            Object.assign(this, source)
        }

        this.name = this.name ? this.name : null
        this.hiredate = this.hiredate ? this.hiredate : null
    }
}
```

```
import Employee from './Employee'

var employee = new Employee({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })

console.log(employee)
```

- JavaScript 2015 defines a module syntax
- NodeJS requires JavaScript 2015 modules have .mjs extensions (and the --experimental-modules flag)
- Modules always scoped, no *iife* required

Implied "use strict"

- ES modules are automatically strict
- Variables must be declared, *eval* and *arguments* may not be changed
- Syntactically incorrect structures allowed by some engines are forbidden

Exports

```
export const TAX_RATE = 0.23
export const VESTED_AT = 6

export default class Employee {
```

```
    constructor(source) {

        if (source) {

            Object.assign(this, source)
        }

        this.name = this.name ? this.name : null
        this.hiredate = this.hiredate ? this.hiredate : null
    }
}
```

```
import Employee, { TAX_RATE, VESTED_AT } from './Employee'

console.log(TAX_RATE)
```

- One default export
- Multiple named exports
- Import named values in a list

Default Export

```
import Worker from './Employee'

var employee = new Worker({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })

console.log(employee)
```

- The name can be changed on import
- Change the name to prevent conflicts
- Changing the name adds confusion

Aliases

```
import E, { TAX_RATE as TS, VESTED_AT as V } from './Employee'

console.log(TAX_RATE)
```

- Any imported name may be aliased (or not)
- Use to avoid collisions between modules
- Avoid aliasing, help keep code readable

Exports

```
const TAX_RATE = 0.23
```

```
const VESTED_AT = 6

class Employee {

    constructor(source) {

        if (source) {

            Object.assign(this, source)
        }

        this.name = this.name ? this.name : null
        this.hiredate = this.hiredate ? this.hiredate : null
    }
}

export TAX_RATE
export VESTED_AT
export default Employee
```

- Combine exports to be readable

Singletons

```
const Type = {
    SALARY: 0,
    HOURLY: 1
}

class EmployeeFactory {

   CreateEmployee(type, source) {

       let employee

       switch (type) {

       Type.SALARY:
           employee = new SalaryEmployee(source)
           break

       Type.HOURLY:
           employee = new HourlyEmployee(source)
           break

       default:
           throw new RangeError('Unknown employee type')
       }

       return employee
   }
}

export Type
export const employeeFactory = new EmployeeFactory()
```

- Hide the class definition and export a.single instance

- Clients are guaranteed to share the one instance

Modules are Read Once

- Singleton or not, when a module is first imported it is read and processed by the module loader
- The module loader caches whatever is exported
- Subsequent imports (or requires) from other modules get the cached results

Simulating Enumerated Types

```
const Type = {
    SALARY: 0,
    HOURLY: 1
}
```

- JavaScript is dynamic and loosely typed, impossible to catch a bad value at compile time
- Using properties does allow an undefined property to be caught at run-time

Module Search Path

- Module search paths are defined by the loader being used
- Note: NodeJS requires the **--experimental-modules** flag, and JavaScript 2015 modules must be have the *.mjs* (Michael Jackson) extension
- NodeJS uses the same rules for CommonJS and JavaScript 2015 modules

Module Content

Modules
Module Content

Module Content

```
const TAX_RATE = 0.23
const VESTED_AT = 6

class Employee {

    constructor(source) {

        if (source) {

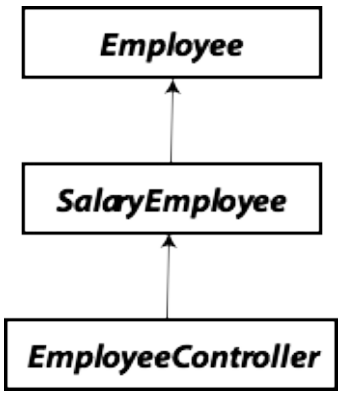
            Object.assign(this, source)
        }

        this.name = this.name ? this.name : null
        this.hiredate = this.hiredate ? this.hiredate : null
    }
}

export { TAX_RATE, VESTED_AT }
export default Employee
```

- Keep the module cohesive, exports related to each other
- Distributed modules may encapsulate complex functionality with a simple interface
- A local module may export just a class, maybe related constants

Modules All The Way Down



- Modules may import other modules
- Complexity delegated to other modules
- Client imports one module with a simple interface

How Many Classes?

- One class per module? Multiple classes per module?
- Both! If classes are shared between modules maybe one-per
- If classes belong to a module, keep them in that module

Re-Exporting Imports

```
import Employee, { TAX_RATE, VESTED_AT } from 'Employee'

class SalaryEmployee extends Employee {

}

export { TAX_RATE, VESTED_AT }
export default SalaryEmployee
```

- A module may include and re-export functionality from other modules
- Expands the interface, consider making the client import everything necessary

Checkpoint

- What are two advantages of using modules?
- How many exports may be made?
- How many default exports are there?
- Do imports need to retain the exported names?
- When is a module too big?

Chapter 7 - Promises

Events and Callbacks
Promises

Objectives

- Review asynchronous programming and callbacks
- Use Promises for multiple callbacks and callback chains
- Handle rejection at the Promise and chain levels

Overview

Callbacks are restrictive and suffer from several problems: often only one callback may be provided for an event, a callback often needs to set up another operation with its own callback, and in some circumstances a callback could be registered after the event has already happened. Promises fix all of these problems!

Events and Callbacks

Events and Callbacks
Promises

Callbacks

```
setTimeout( () => console.log('Timer expired'), 1000)
console.log('Start')
```

```
Start
Timer expired
```

- Common, traditional way to respond to an event

Event Driven

```
window.addEventListener('load', (event) => {

    console.log('The page has loaded!')
})
```

- Browser applications are event driven

Named Functions

```
var timeoutMessage = 'Timer expired'

function timerExpired() {

    console.log(timeoutMessage)
}

function task() {

    var message = 'Function timer expired'

    setTimout(timerExpired, 1000)
    setTimout( () => console.log(message), 1000)
}

task()
task()
```

```
Timer expired
Function timer expired
Timer expired
Function timer expired
```

- Named function definitions support DRY, but exist in global space
- They (wrongly) encourage sharing global variables

Closures

```
function task() {

    var timeoutMessage = 'Closure timer expired'

    function timerExpired() {

        console.log(timeoutMessage)
    }

    setTimout(timerExpired, 1000)
    setTimout( () => console.log(timeoutMessage), 1000)
}

task()
task()
```

```
Timer expired
Function timer expired
Timer expired
Function timer expired
```

- Closures may be named functions, inline functions, or arrow functions
- Closures bind local variables in scope, but are created at every pass; two closure copies created here

Nested Callbacks

```
function task() {

    setTimout( () => {

        console.log('First timer expired')

        setTimeout( () => {

            console.log('Second timer expired')

            setTimeout( () => {

                console.log('Third timer expired')

            }, 1000)
```

```
        }, 1000)
    }, 1000)
}

task()
```

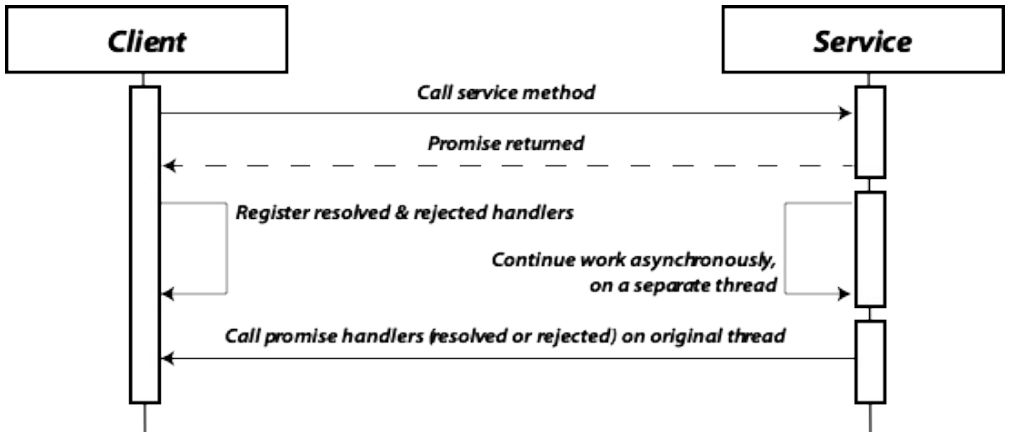
```
First timer expired
Second timer expired
Third timer expired
```

- When a sequence of actions is made for a sequence of events
- Nested callbacks are very confusing to follow

Promises

Events and Callbacks
Promises

Promises



- Client gets Promise immediately, registers handlers
- More than one handler may be registered
- A handler runs even if registered after the Promise is resolved

Creating a Promise

```
function timer(wait) {

  return new Promise( (resolve, reject) => {

      setTimeout( () => resolve(true), wait)

      console.log('End of the Promise callback')
  })
}
```

- A new Promise provides references to *resolve* and *reject* functions to a callback that is immediately invoked
- The Promise is returned when the callback finishes; setTimeout is asynchronous so the operation will complete after the callback finishes
- When the operation is complete, call *resolve* with a value

Consuming Promises

```
function timer(wait) {

  return new Promise( (resolve, reject) => {
```

```
      setTimeout( () => resolve(true), wait)

      console.log('End of the Promise callback')
  })
}

console.log('Before timer set')

var p = timer(1000)

console.log('After timer set')

p.then( (result) => console.log('Timer expired') )

console.log('After handler registered')
```

```
Before timer set
End of the Promise callback
After timer set
After handler registered
Timer expired
```

- A function returns a Promise instead of receiving a callback
- A handler is registered with the Promise
- When the work is done, the handler is run

Multiple Registrations

```
var p = timer(1000)

p.then( (result) => console.log('Timer expired') )
p.then( (result) => console.log('Timer expired here too') )
```

```
Timer expired
Timer expired here too
```

- The first value of Promises: multiple handlers may be registered
- They are all run when the Promise resolves

Register Anytime

```
function timer(wait) {

  return new Promise( (resolve, reject) => {

      resolve(true)
      console.log('Promise resolved')
  })
}
```

```
var p = timer(1000)

p.then( (result) => console.log('Timer expired') )
```

```
Promise resolved
Timer expired
```

- A non-asynchronous Promise will resolve before a handler is registered
- That is OK, handlers are always run

Rejected

```
function timer(wait) {

  return new Promise( (resolve, reject) => {

      if (wait <= 1000) {

          resolve(true)
          console.log('Promise resolved')

      } else {

          reject('Sorry')
          console.err('Promise rejected')
      }
  })
}

var p = timer(1001)

p.then( (result) => console.log('Timer expired'),
        (reason) => console.log(reason))
```

```
Promise rejected
Sorry
```

- The second handler is for rejection

Chaining Promises

```
function timer(wait, msg) {

  return new Promise( (resolve, reject) => {

      if (wait <= 1000) {

          resolve(msg)
          console.log('Promise resolved')
```

```
      } else {

          reject('Sorry')
          console.err('Promise rejected')
      }
  })
}

var p = timer(1000, 'First timer')

p.then( (result) => {

   console.log(result)

   return timer(1000, 'Second timer')

}).then( (result) => {

   console.log(msg)

   return 'Done'

}).then( (result) => {

   console.log(msg)
})
```

```
Promise tesolved
First timer
Promise resolved
Second timer
Done
```

- The second value of Promises: chaining beats nesting
- Return a promise, it is the next Promise in the chain
- Return a value: a new Promise is returned, resolved to the value

Catch (Rejected)

```
function timer(wait, msg) {

  return new Promise( (resolve, reject) => {

      if (wait <= 1000) {

          resolve(msg)
          console.log('Promise resolved')

      } else {

          reject('Sorry')
          console.log('Promise rejected')
      }
  })
}
```

```
var p = timer(1001, 'First timer')

p.then( (result) => {

   console.log(result)

   return timer(1000, 'Second timer')

}).then( (result) => {

   console.log(msg)

   return 'Done'

}).then( (result) => {

   console.log(msg)

}).catch( (reason) => {

   console.err(reason)
})
```

```
Promise rejected
Sorry
```

- Second handler in *then* handles a rejection for only that promise
- Catch at the end catches any rejection in the chain
- Both forms are useful, and not mutually exclusive

Fetch

```
class product {

    constructor(source) {

       Object.assign(this, source)
    }

    get name() { return this._name }
    set name(value) { this._name = value }

    get price() { return this._price }
    set price(value) { this._price = value }
}

function getProducts() {

    var p = fetch('http://localhost:3001/products')

    return p.then( (result) => result.json() )
}

function loadData() {
```

```
    var p = getProducts()

    p.then( (source) => source.map( obj => new Product(obj) )
        .catch( (reason) => console.err(reason) )
}

loadData()
console.log('After loadData')
```

```
After loadData
[
    { name: "Cappuccino", price: 4.65 },
    { name: "Carmel Mocha", price: 3.75 }
]
```

- fetch returns AJAX results with a Promise
- The client *loadData* expects a Promise, the function *getProducts* returns the last Promise in the chain
- Requires the service in Resources/Service to be started, otherwise the rejection from the fetch in *getProducts* will be caught by the catch in *loadData* (try it!)

Class Methods

```
class DataSource {

    get products() {

        var p = fetch('http://localhost:3001/products')

        return p.then( (results) => results.json() )
    }
}
```

- Class methods, except for the constructor, may return promises
- A property (get) may evaluate to a Promise

Promisify

```
// NodeJS readFile expects a callback accepting (err, data)
// util.promisify wraps it with a Promise

var util = require('util')
var fs = require("fs")

var readFile = util.promisify(fs.readFile);

var p = readFile("myfile.js", "utf8")

p.then( (contents) => console.log(contents) )
    .catch( (e) => console.err("Error reading file", e) )
```

- *Promisify* libraries wrap asynchronous functions accepting callbacks with a function returning a promise
- Works similar to these examples wrapped setTimeout

Checkpoint

- What are the two advantages of Promises?
- How is a Promise resolved?
- How is a Promise rejected?
- What happens if a handler is registered after resolution or rejection?
- What is the most important feature of closures as callbacks and handlers?
- What is the advantage of the *catch*?

Chapter 8 - *async* and *await*

async and *await*

Objectives

- Simplify program syntax using *async* and *await*
- Apply *async* to functions, arrow functions, and class methods
- Explore using *async* in situations where it is ignored

Overview

Async and await are an alternative syntax to support promises. What they offer is simply a more synchronous appearance of the code structure when asynchronous operations are performed.

async and *await*

async* and *await

async

```
async function square(value) {

    return value * value
}

var p = square(5)

p.then( (result) => console.log(result) )

console.log('After handler registration')
```

```
25
After handler registration
```

- *async* wraps a function with a promise
- A client sees a Promise, whatever the function returns is the resolution of the Promise

Promise Chain

```
function getProducts() {

    var p = fetch('http://localhost:3001/products')

    return p.then( (results) => results.json() )
}

function loadData() {

    var p = getProducts()

    p.then( (products) => console.log(products) )
        .catch( (reason) => console.err(reason) )
}

loadData()
console.log('After loadData')
```

```
After loadData
[
    { name: "Cappuccino", price: 4.65 },
    { name: "Carmel Mocha", price: 3.75 }
]
```

- fetch returns AJAX results with a Promise
- The client *loadData* expects a Promise, the function *getProducts* returns the last Promise in the chain
- Requires the service in Resources/Service to be started, otherwise the rejection from the fetch in *getProducts* will be caught by the catch in *loadData* (try it!)

async and Promises

```
async function getProducts() {

    var p = fetch('http://localhost:3001/products')

    return p.then( (results) => results.json() )
}
```

- It is OK to wrap a function that already returns a Promise with *async*
- It declares to clients the function returns a Promise

await

```
async function getProducts() {

    var results = await fetch('http://localhost:3001/products')

    return results
}
```

- Only in an *async* function
- Captures the resolution of a promise and allows a synchronous looking statement
- The function is asynchronous; it returns immediately and the return statement is the resolution of the promise

await Chains

```
async function getProducts() {

    var results = await fetch('http://localhost:3001/products')

    return results
}

async function initializeProducts(products) {

    var results = await products.map( (product) => new Product(product) )

    return results
}

async function initialize() {

    var data = await getProducts()
    var products = await initializeProducts(data)
```

```
    return products
}
```

- This is the full benefit: the code in initialize looks synchronous but operates asynchronously using promises
- *initializeProducts* is actually synchronous, but still returns results through a promise

async and *await* in Classes

- methods, static methods, properties, and static properties may be asynchronous
- constructors may not be asynchronous

Checkpoint

- Why use *async*/*await* instead of Promises?
- Where can await be used?
- What does await return if it is not used to call an async/Promise?

Chapter 9 - Unit Testing & Test-Driven Development

Unit Testing & Test-Driven Development

Objectives

- Identify what needs to be tested
- Explore how to test it
- Build unit tests using a standard framework

Overview

Everybody tests their code. Most programmers write a little, test a little. Unit testing provides a standardized framework, which makes it easy to maintain the tests. Test-driven development reverses the work; write the tests first, then write the code to satisfy the tests.

Unit Testing and & Test-Driven Development

Unit Testing & Test-Driven Development

Unit Tests

- Simply tests to ensure the code works correctly
- Programmers have always tested their code, unit tests just formalize it
- Test a unit of code: usually a class

What to Test

```
class Calculator {

    add(a, b) {

        this.checkConstraints(a, b)

        let result = a + b

        return result
    }

    checkConstraints(a, b) {

        if (a <= 0 || b <= 0 || a > 100 || b > 100) {

            throw new Error('parameters must be between 1 and 100')
        }
    }
}
```

- Given a calculator with add, subtract, multiply and divide methods all expecting *a* and *b*
- *a* and *b* must be greater than zero and less than or equal to 100
- The calculator will throw an Error object if the constraints are not met

Test Suites

- Plenty of test suites for JavaScript: Jasmine, Mocha, Jest, etc.
- Many share fundamental syntax with Jasmine at https://jasmine.github.io

Starting with Jasmine

```
$ npm install jasmine jasmine-console-reporter --save-dev
$ npx jasmine init
$ npx jasmine
```

- To test from inside NodeJS, the console reporter is required
- *jasmine init* will create the *spec* folder and a basic configuration
- Create spec files in the *spec* folder, and then run Jasmine

How to Test

```
require("jasmine")

const Calculator = require("../Calculator")

describe('Calculator Tests', () => {

    // Addition.

    it('adds 1 and 1', () => {

        const calculator = new Calculator()

        expect(calculator.add(1, 1)).toBe(2)
    })
```

- *describe* a suite of tests
- Each *spec* is defined as a callback to *it*
- This example uses Node with CommonJS

Assertions

```
expect(calculator.add(1, 1)).toBe(2)
```

- *expect* is passed the result of doing something
- Matchers are: toEqual, toBe, toBeCloseTo, toBeNaN, etc.
- Jasmine matchers are defined at https://jasmine.github.io/api/edge/matchers.html

beforeEach

```
require("jasmine")

const Calculator = require("../Calculator")

describe('Calculator Tests', () => {

    let calculator = null

    beforeEach( () => {

        calculator = new Calculator()
    })

    // Addition.
```

```
    it('adds 1 and 1', () => {

        expect(calculator.add(1, 1)).toBe(2)
    })
```

- DRY: *beforeEach* is run before every test, *afterEach* is run after every spec
- Use them to establish a common state before and after each spec is run

Edge/Boundary Conditions

- Testing random values is OK, but probably not very useful
- Test at the edges: 100 & 101, 0 & 1
- Zero is funny, programmers make mistakes, so always test -1, 0, and 1

False Positives

- Watch out for false positives
- Testing calculator.add(2, 2) will succeed, even if the method actually does multiplication

Simple Tests

```
it('adds numbers between 1 and 100', () => {

    expect(calculator.add(1, 1)).toBe(2)
    expect(calculator.add(100, 100)).toBe(200)
})
```

- Think single responsibility
- Multiple assertions are not always checked, Jasmine stops when the first one fails
- Testing the four methods of the calculator requires at least 32 simple specs

Testing for Exceptions

```
it('refuses to add -1 and 1', () => {

    expect( () => calculator.add(-1, 1) ).toThrow()
})
```

- To check an exception do not to pass the result of calling the method (it failed), but the method itself
- To pass the method, wrap it in a callback and Jasmine will realize that it needs to be run
- *toThrow* accepts a parameter, in which case what was thrown must be an exact match

Test Doubles

```
class EmployeeLoader {
```

```
    constructor(dataLoader) {

        this.loader = dataLoader
    }

    getEmployeeCount() {

        let employees = this.loader.getEmployees()

        return employees.length
    }
}
```

```
var dataLoader
var employeeLoader

beforeAll( () => {

    dataLoader = {

        getEmployees() {

            return [ { id: 1, name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 } ]
        }
    }
})

beforeEach( () => {

    employeeLoader = new EmployeeLoader(dataLoader)
})

it('Should get the correct employee count', () => {

    expect(employeeLoader.getEmployeeCount()).toBe(1)
})
```

- Test doubles are code that replaces production code for the purpose of a test
- Used when the code must produce consistent results, and production code will not
- Tied to the Open/Closed and Dependency Inversion principles: substitution with another object of the same interface

Types of Test Doubles

Type	Description
Stub	Code that is standing in for something not yet available
Spy	Captures results for later verification
Mock	Provides consistent results and can be inspected
Fake	Provides consistent results but is simple and cannot be inspected
Dummy	An object standing in, but does not provide functionality

- The previous example used a *fake*

- A *mock*, like a *spy* could have told us things, like the *getEmployees* method was actually called
- The problem with spies is that the test is looking at the internal workings, and why test that?

Using Test Doubles

- Always test a single unit, not what it depends on
- One school of thought says that means you should mock every dependency
- Another says only mock dependencies that do not provide consistent results

Testing a Promise

```
class EmployeeLoader {

    constructor(dataLoader) {

        this.loader = dataLoader
    }

    async getEmployeeCount() {

        let employees = async this.loader.getEmployees()

        return employees.length
    }
}
```

```
var dataLoader
var employeeLoader

beforeAll( () => {

    dataLoader = {

        async getEmployees() {

            return new Promise( (resolve, reject) => {

                resolve( [ {
                                id: 1,
                                name: 'John Smith',
                                hiredate: new Date('2003-07-01'),
                                salary: 52000
                            } ] )
            }
        }
    }
})

beforeEach( () => {

    employeeLoader = new EmployeeLoader(dataLoader)
})

it('Should get the correct employee count', async (done) => {

    count = await employeeLoader.getEmployeeCount()
```

```
    expect(count).toBe(1)
    done()
})
```

- Same example, but the fake now returns a promise and the methods in between are async
- The spec accepts the parameter *done*, and uses it to tell Jasmine when the test is done

Test-Driven Development

- Write tests, then write the code that satisfies the test
- To write the test, the requirements need to be understood
- Tests get written, no extra features are added to the code, everything goes faster

Checkpoint

- What are the benefits of TDD?
- Why are multiple assertions in one spec generally a bad idea?
- What should you check if there are constraints?
- Why is zero funny, and what should you do?
- To enforce DRY, where should common setup for each spec be placed?
- What is different about checking exceptions?

Appendix A - TypeScript

TypeScript
TypeScript Features

Objectives

- Understand the relationship between JavaScript and TypeScript
- Explore using functions as callbacks
- Understand the drawbacks and benefits of closures, and how arrow functions fit

Overview

TypeScript is the next step past JavaScript, a strongly-typed version of JavaScript where variable and parameter types are checked by a compiler and type mismatches are eliminated at compile-time. TypeScript also has other features, such as decorators that add metadata to classes and class members at compile time. These features are absolutely necessary for some frameworks, like Angular, but the strongly typed features of the language are aslo popular with frameworks like React.

TypeScript

TypeScript
TypeScript Features

TypeScript

- JavaScript with strong type checking
- JavaScript syntax still works, parameters and variables do not need be strongly typed
- TypeScript is a superset: JavaScript + strongly typed features

TypeScript Compiler

```
$ npm install typescript -g
$ ts code.ts
$ node code.js
```

- Node does not support TypeScript directly
- TypeScript cross-compiles into JavaScript
- Some applications require this global installation

Local vs Global Installation

```
$ npm install typescript --save-dev
$ npx ts code.ts
$ node code.js
```

- Unfortunately TypeScript versions are not backwards-compatible
- Usually better to install the required version in the project
- Run the project version with *npx*

TypeScript Features

TypeScript
TypeScript Features

Typed Parameters and Variables

```
function f( n: Number, b: String ) {

    return n + b * 1
}
```

- TypeScript supports types using a : notation
- TypeScript does not require typed parameters and variables

Enumerated Types

```
enum Compass { North, South, East, West }

let direction: Compass = Compass.North
```

- direction can only be assigned one of the Compass types

Interfaces

```
Interface SalariedEmployee {

    weeklySalary(): Number
}
```

Classes

```
class Employee {

    name: String
    salary: Number

    get weeklySalary(): Number {

        return this.salary / 52
    }
}
```

- TypeScript adds member variable declarations

Type Inference

```
let e: Employee = new Employee()
let s: SalariedEmployee = e // e is-a SalariedEmployee as well
```

- An object is considered to be of a type if it provides the interface
- It does not need to be explicitly declared to implement the interface

Generics

```
class ction pay<T>(employee: T) {

}
```

Decorators

```
@Component({
  selector: 'tabs',
  template: `
    <ul>
      <li>Tab 1</li>
      <li>Tab 2</li>
    </ul>
  `
})
export class Tabs {

}
```

- TypeScript supports dectorators; some frameworks, e.g. Angular, depend heavily on this feature
- Decorators are TypeScript function that adds metadata to TypeScript
- Decorators apply to classes, class fields, and methods

Checkpoint

- Why use the TypeScript language?
- What features does TypeScript offer for strong type checking?
- Can TypeScript and JavaScript co-exist?
- How do classes in TypeScript differ from JavaScript?
- What are two type features that TypeScript adds over JavaScript?
- What do decorators accomplish?