Review

NodeJS and JavaScript
Functions, Scope, and iffes
Debugging

Objectives

- Introduce Node.js
- Review JavaScript basics
- Debug JavaScript under NodeJS

NodeJS and JavaScript

Node.js and JavaScript
Functions, Scope, and iffes
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

```
> 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

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 ]

console.log('for loop:')

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

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

console.log('for ... of:')

for (let v of a) {

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

```
$ 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
- ES6 adds a for ...of loop for iterable objects

Functions, Scope, and iffes

NodeJS and JavaScript

Functions, Scope, and iffes

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

```
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
```

- ES6 added *let* with block scope
- y is 5 because the return used global x, which is 5

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

Debugging

NodeJS and JavaScript
Functions, Scope, and iffes
Debugging

Debugging

- Launch in a debugger; Visual Studio Code or in the Browser
- Set breakpoints
- Look at variables and watch expressions

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?

Strings and Template Strings

Strings and Methods
Template Strings and Expressions

Objectives

- Review String object basics
- Use JavaScript expressions in Template Strings

Strings and Methods

Strings and Methods
Template Strings and Expressions

String Literals & Special Characters

- Double-quotes and single-quotes are equal in JavaScript
- Use \ to insert special characters and escape others

String Equality

- == and === may always be used to check strings
- Strings are cached, and only one copy of each exists

String Methods

Template Strings and Expressions

Strings and Methods
Template Strings and Expressions

Template Strings

Checkpoint

- Why

Chapter 3 - Functions

Function Definitions
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

Function Definitions

Function Definitions
Arrow Functions

Function Definition

```
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

Rest Operator

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

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

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

- ES6 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)
```

- ES6 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

Detructuring Objects

```
function f() {

    return { one: 1, two: 2 }
}

let { one, two } = f()

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

- Extract named properties into variables

Deep Detructuring 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

Closures

```
function setTimer() {}

    let message = 'Timer expired'

    setTimeout(function () {

        console.log(message)
    }, 1000)
```

```
}

setTimer()
console.log('Timer started')
```

- Closures are functions defined in the scope of another function
- Closures have access to the variables in the definition scope

Arrow Functions

Function Definitions
Callbacks and Closures
Arrow Functions

Arrow Functions

```
setTimeout( () => {

    console.log('Timer expired', 1000) )
}
```

- Arrow functions may only be closures; they cannot be assigned to a prototype property
- The closure is defined as variables => the function body

Arrow Function Syntax

- Parenthesis are necessary only if there are no parameters or multiple parameters
- If there is only one statement 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

Objects and Polymorphism

Objects and Polymorphism
Object-Oriented Programming

Literal Objects

```
var johnsmith = {

    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000
}

console.log(johnsmith)
console.log(`John Smith's hire date is ${johnsmith.hiredate}}`)

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

```
$ 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 propertys through the property accessor operator (.)
- New properties may be added at any time

Subscript Notation

```
var johnsmith = {

    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000
}

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

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

```
$ 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 johnsmith = {

    name: 'John Smith',
    hiredate: new Date('2003-07-01'),
    salary: 52000
}

for (property in johnsmith) {

    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

```
var johnsmith = {

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

console.log(`John Smith's weekly pay is ${johnsmith.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

Duck Typing

- JavaScript does not verify properties at compile time
- If the object has the property at run-time, then it works

Object-Oriented Programming

Objects and Polymorphism

Object-Oriented Programming

Polymorphism is the Key

- JavaScript uses duck-typing for polymorphism
- Calls to a method invoke the function bound to that object

Encapsulation

- Data and processes are organized together
- The method invoked belongs with the data (polymorphism)

Data Hiding

- Clients only see the object's public interface
- Clients are not dependent on the object's implementation
- JavaScript has no expectation of privacy, signal it with _

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?
- What character should start a "private" property?
- 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

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

console.log(johnsmith)
```

```
$ 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

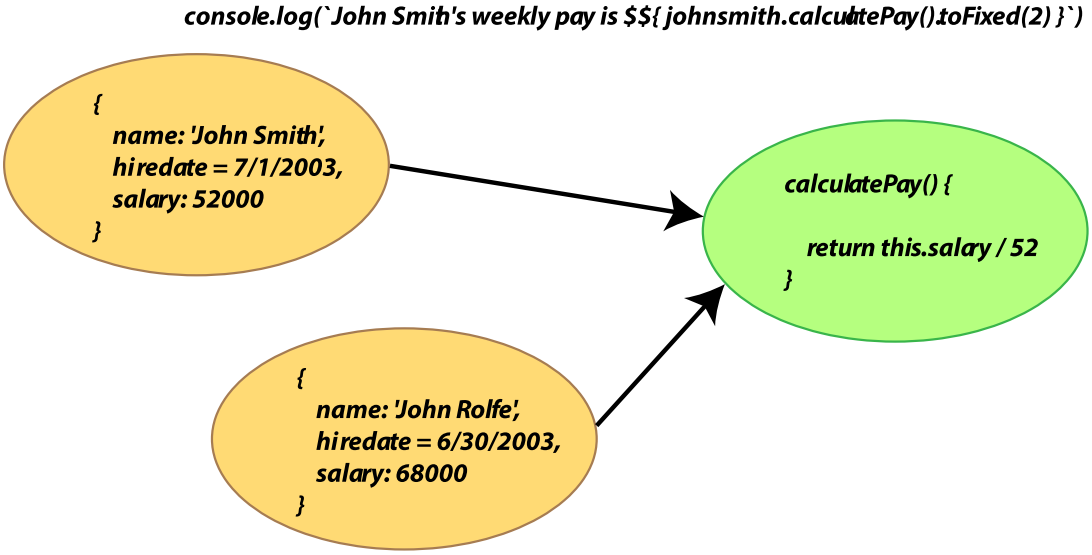
```
class Employee {

    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() }`)
```

```
$ node methods.js
John Smith's weekly pay is 0
```

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

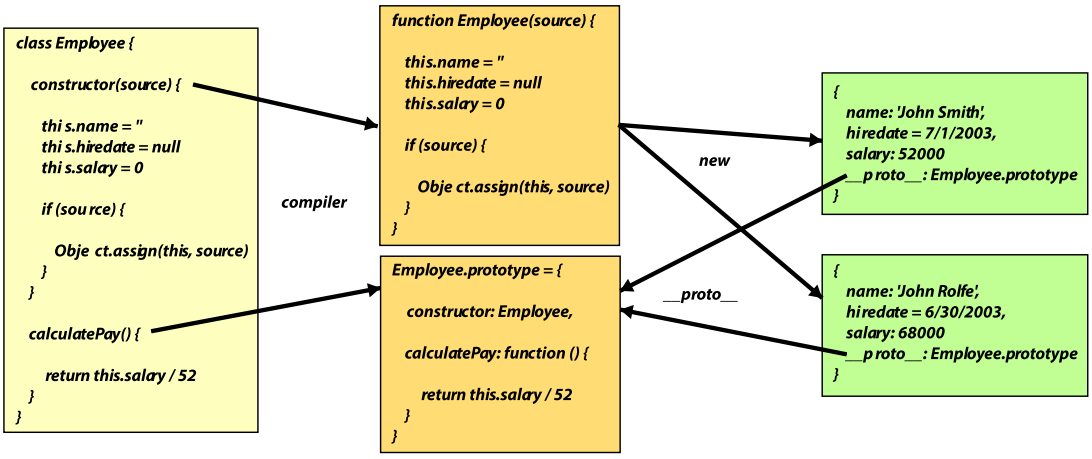
Prototypal Programming




```
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

```
// NS20305 shared-methods.js
// Copyright (c) 2018 NextStep IT Training. All rights reserved.
//

class Employee {

    constructor(source) {

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

        if (source) {

            Object.assign(this, source)
        }
    }

    calculatePay() {

        return this.salary / 52
    }
}

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

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

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

console.log(`johnsmith.calculatePay === johnrobles.calculatePay: ${
johnsmith.calculatePay === johnrolfe.calculatePay }`)
```

```
$ 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
johnsmith.calculatePay === johnrobles.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 johnsmith = new Employee({ name: 'John Smith', hiredate: new
Date('2003-07-01'), salary: 52000 })
```

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

```
$ node static.js
johnsmith.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 johnsmith = 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 Objecct.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 johnsmith = new Employee({ name: 'John Smith', hiredate: new
Date('2003-07-01'), salary: 52000 })

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

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

Class Inheritance

Class Definitions Class Properties and Encapsulation 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 super-class
- An object of the extended class is an instance of the super-class

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 super-class method through super

Super-Class 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.salaray / 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 super-class constructor call
- But, the super-class assignment could have initialized the properties...

Inheritance is 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 johnsmith = new SalaryEmployee({ name: 'John Smith', hiredate: new
Date('2003-07-01'), salary: 52000 })

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

```
$ node static-inheritance.js
johnsmith.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 super-class 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 ES6 modules
- Export and import multiple objects
- Discuss how modules should be organized

Modules

Modules
Module Content

Modules

<< graphic of modularity >>

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

console.log(johnsmith)
```

- 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

ES6 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 johnsmith = new Employee({ name: 'John Smith', hiredate: new
Date('2003-07-01'), salary: 52000 })

console.log(johnsmith)
```

- ES6 defines a module syntax
- NodeJS requires ES6 modules have .mjs extensions
- 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 johnsmith = new Worker({ name: 'John Smith', hiredate: new Date('2003-07-01'), salary: 52000 })

console.log(johnsmith)
```

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

Aliases

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

console.log(TAX_RATE)
```

- All imported names may be aliased
- Use to avoid collisions between modules
- Avoid to 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

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
- NodeJS uses the same rules for CommonJS and ES6 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
export VESTED_AT
export default Employee
```

- Keep the module cohesive, exports related to each other
- Disributed 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

<< graphic showing decomposition and delegation >>

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

Re-Exporting Imports

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

class SalaryEmployee extends Employee {

}

export TAX_RATE
export 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

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 vs Closures

```
var timeoutMessage = 'Timer expired'

function timerExpired() {

    console.log(timeoutMessage)
```

```
}

function task() {

    var message = 'Closure timer expired'

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

task()
task()
```

```
Timer expired
Closure timer expired
Timer expired
Closure timer expired
```

- Named function definitions support DRY, but exist in global space
- Encourage sharing global variables
- 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
- Confusing to follow

Promises

Events and Callbacks
Promises

Promises

<< graphic with bnumbered steps: 1) client call, 2) Promise returned, 3) client registers handlers, 4) Promise resolves, 5) Handler executes >>

- 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 recieving 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
```

- 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

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

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:2020/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: "Capuccino", 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:2020/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*

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:2020/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: "Capuccino", 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:2020/products')

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

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

await

```
async function getProducts() {

    var results = await fetch('http://localhost:2020/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 before the await completes and the return statement provides the resolution of the promise

await Chains

```
async function getProducts() {

    var results = await fetch('http://localhost:2020/products')

    return results
}

async function loadData() {

    var products = await getProducts()

    console.log(products)
}
```

-

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 - 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

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 super-set: 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

Local vs Global Installation

- Some applications require global installation
- Unfortunately TypeScript versions are not backwards-compatible

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