Web Applications

CSE183

Fall 2020

JavaScript II

Baskin Engineering
UC SANTA CRUZ

Notices

- **Administration 1 & 2** due 23:59 **Thursday October 15**
- **Assignment 2** due 23:59 **Thursday October 15**

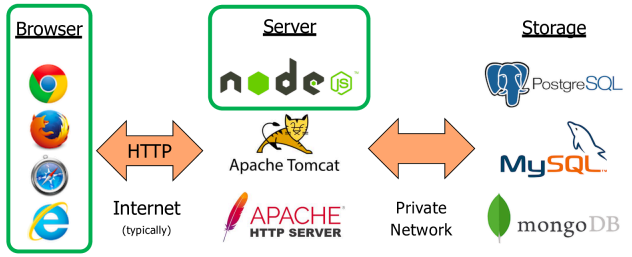
- **Quiz 1** during lecture **Friday October 16**

Today's Lecture

- JavaScript Recap
- Object Oriented Programming
- Functional Programming
- Closures
- Idioms and Quirks

- Quiz Instructions
- Questions

Full Stack Web Applications



JavaScript - Overview

- High-level
  - Heavily abstracted from hardware details
- Interpreted
  - Not compiled, executed by a platform-dependent run-time environment
- Dynamic
  - Undertakes compiler-like operations at runtime
- Untyped / Dynamically Typed
  - Any variable can hold any type of data
- Prototype-based
  - Object-oriented behaviors are re-used (inherited) from existing objects (prototypes)
- Has first-class functions
  - Functions are objects and can be manipulated as such

JavaScript - Programming

- Models
  - Object-oriented
    - Encapsulation, abstraction, inheritance, polymorphism
  - Imperative
    - Instructions executed sequentially; code is easy to understand
  - Functional
    - Declarative composition of value returning functions creating a call tree rather than manipulating a global state
- Evolution
  - Originally convention (pattern) based
  - ECMAScript keeps adding language features
    - E.g. the `class` concept

https://www.ecma-international.org/ecma-262/11.0/index.html

Object Orientation - Methods

- First class functions ⇒ an object property can be a function
  - A “method” in object oriented speak

```
var obj = {count: 0};
obj.increment = function (amount) {
 this.count += amount;
 return this.count;
}
```

- Method invocation:
  - Function is called and literal `this` is bound to the object

```
obj.increment(2); // returns 2
obj.increment(-1); // returns 1
```


Object Orientation - `this`

- In methods `this` is bound to the object

```
var obj = { prop1: 'property 1'};
obj.addProp2 = function() {
 this.prop2 = "property 2";
}
console.log(Object.keys(obj));
obj.addProp2();
console.log(Object.keys(obj));
console.log(obj);
```

```
[ 'prop1', 'addProp2' ]
[ 'prop1', 'addProp2', 'prop2' ]
{ prop1: 'property 1', addProp2: [Function], prop2: 'property 2' }
```

- In non-method functions:
  - `this` will be the global object
  - Or if `"use strict";` `this` will be undefined
    - However, `"use strict";` should *not* be used globally ¯_(ツ)_/¯

Object Orientation - Functions are Objects

- Functions can have properties

```
function increment(value) {
  if (increment.invocations == undefined) {
    increment.invocations = 0;
  }
  increment.invocations++;
  return value + 1;
}
increment(4);
increment(-1028);
// increment.invocations == 2
```

- Analogous to static/class properties in “purer” object-oriented languages like Java

Object Orientation - Functions are Objects

- Functions can have methods

```
function func(arg) { console.log(this,arg); }
```

- `toString()` returns function as source string

  `func.toString()` returns `'function func(arg) { console.log(this,arg); }'`

- `call()` calls function specifying `this` and arguments

  `func.call({t: 1}, 2)` prints `'{ t: 1 } 2'`

- `bind()` creates new function with `this` and arguments bound

```
let newFunc = func.bind({z: 2}, 3);
newFunc(); prints '{ z: 2 } 3'
```


Object Orientation - Classes

- Functions are classes

```
function Rectangle(width, height) {
  this.width = width;
  this.height = height;
  this.area = function() { return this.width * this.height; }
}
var r = new Rectangle(6, 7);
// r.area(); returns Jackie Robinson's 42
```

- Not a good way to add methods - Why?

Object Orientation - Inheritance

- JavaScript has a prototype object for each object instance
  - Prototype objects can have their own prototype objects forming a prototype chain
- When *reading* the value on an object property, JavaScript will search up the prototype chain until the property is found
  - The full set of properties of an object are its own properties plus all the properties found up the prototype chain
    - This is known as prototype-based inheritance
    - Single inheritance only, cannot inherit properties from two ‘parents’
- Property *updates* work differently:
  - JavaScript creates the property in the object instance if not found up the prototype chain

Using Prototypes

```
function Rectangle(width, height) {
  this.width = width;
  this.height = height;
}
Rectangle.prototype.area = function() {
  return this.width * this.height;
}
var r = new Rectangle(6, 7);
// r.area(); still returns Jackie Robinson's 42
```

- As JavaScript has a dynamic type system, changing the prototype causes all instances to change
  - Which is typically what we want
  - Much better way of adding a method ☺

Inheritance

```
function Shape(name) {
  this.name = name;
}
function Rectangle(width, height) {
  this.width = width;
  this.height = height;
}
Rectangle.prototype = new Shape('Rectangle');
Rectangle.prototype.area = function() {
  return this.width * this.height;
}
var r = new Rectangle(6, 7);
let s = new Shape('Generic Shape');

// r.Name == 'Rectangle'
s.area(); // syntax error
```


ECMAScript Syntax

```
class Shape {                           // Class definition
  constructor(name) {
    this.name = name;
  }
}
class Rectangle extends Shape {         // Class definition and Inheritance
  constructor(height, width) {
    super('Rectangle');
    this.height = height;
    this.width = width;
  }
  area() {                              // Instance method definition
    return this.width * this.height;
  }
  static foo() {                        // Static method definition
  }
}
var r = new Rectangle(10,20);
console.log(r);
```

```
Rectangle { name: 'Rectangle', height: 10, width: 20 }
```


Functional Programming

```
let before = [1,2,3,4,5,6];
let after = [];
```

- Imperative

```
for (var i = 0; i < before.length; i++) {
  after[i] = before[i]*i;
}
```

- Functional

```
after = before.map(function (value, index) {
  return value*index;
});
```

- ECMAScript "Arrow" Functional

```
after = before.map((value, index) => value*index);
```

- In all cases

```
console.log(after);
```

```
[ 0, 2, 6, 12, 20, 30 ]
```

Functional Programming

- You can program almost entirely in an imperative style, but..
- Functional programming is required in asynchronous code
  - Browser:

```
function callback() {
  console.log("timeout");
}
setTimeout(callback, 3*1000);
```

  - Server:

```
function callback(err, data) {
  console.log(String(data));
}
fs.readFile('/etc/passwd', callback);
```


Closures

- Given:

```
var glob = 1;
function local(arg) {
  var loc = 0;
  function embedded() {return ++loc + arg + glob;}
  return embedded;
}
```

- What's the difference between:

```
let func = local(2);
console.log(func());
console.log(func());
```

```
4
5
```

and

```
console.log(local(2)());
console.log(local(2)());
```

```
4
4
```

- The closure of `func` includes `glob`, `loc`, and `arg`
- The other version has two distinct instances of `local`

Closures for Privacy

```
let obj = (function() {
 let count = 1;
 let text = "test";
 let setCount = function(value) { count = value; }
 let compute = function() { return count + ' ' + text; }
 return {compute: compute, setCount: setCount};
})();
// typeof obj ==  'object'
// Object.keys(obj) returns [ 'compute', 'setCount' ]
obj.setCount(128);
```

- What does `obj.compute()` return?
  `'128 test'`
- What is the value of `obj.count`?
  `undefined`

`count` and `text` are enclosed as private attributes of `obj`

Nested functions and `self`

```
function readFileFunc() {
  fs.readFile(this.fileName, function (err, data) {
    if (!err) {
      console.log(this.fileName, 'length', data.length);
    }
  });
}
({fileName: "/etc/passwd", readFile: readFileFunc}).readFile();
```

```
undefined length 6946
```

```
function readFileSelf() {
  let self = this;
  fs.readFile(this.fileName, function (err, data) {
    if (!err) {
      console.log(self.fileName, 'length', data.length);
    }
  });
}
({fileName: "/etc/passwd", readFile: readFileSelf}).readFile();
```

```
/etc/passwd length 6946
```

```
function readFileArrow() {
  fs.readFile(this.fileName, (err, data) => {
    if (!err) {
      console.log(this.fileName, 'length', data.length);
    }
  });
}
({fileName: "/etc/passwd", readFile: readFileArrow}).readFile();
```

```
/etc/passwd length 6946
```

JavaScript Object Notation (JSON)

```
let obj = { s: 'str', n: 1, a: [1,'two',,4], o: { n: 1}};
console.log(obj);
```

```
{ s: 'str', n: 1, a: [ 1, 'two', <1 empty item>, 4 ], o: { n: 1 } }
```

```
let json = JSON.stringify(obj);
console.log(json);
```

```
{"s":"str","n":1,"a":[1,"two",null,4],"o":{"n":1}}
```

```
let nobj = JSON.parse(json);
console.log(nobj);
```

```
{ s: 'str', n: 1, a: [ 1, 'two', null, 4 ], o: { n: 1 } }
```


A Useful JavaScript Idiom

- Assign default values

```
class Rectangle {
  constructor(height, width) {
    this.height = height || 0;
    this.width = width || 0;
  }
  area() {
    return this.width * this.height;
  }
}
```


CSE183 Quizzes

- Zoom Proctoring
  - Have your camera on pointing at you
  - Turn microphone off
- Canvas Quiz
  - Randomized question order
  - Must be answered in order presented
  - Some multiple choice
  - Some text-entry
- 25 Minutes
  - Starts at 09:25
  - DRC Accommodations have time multipliers
  - Submits automatically
- Practice quiz available later today

Upcoming Lectures

- Friday: Quiz 1 & Assignment 3
- Monday: JavaScript III & Document Object Model

Tasks

- **Administration 1 & 2** due 23:59 **Thursday October 15**
- **Assignment 2** due 23:59 **Thursday October 15**