JavaScript Programming

Mendel Rosenblum

How do you program in JavaScript?

From Wikipedia:

> …
>
> ... supporting **object-oriented**, **imperative**, and **functional programming**
>
> ...

- Originally programming conventions (i.e. patterns) rather than language features
  - ECMAScript adding language features (e.g. `class`, `=>` , etc.)

Object-oriented programming: methods

- A property of an object can be a function

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

- Method invocation:

```
obj.increment(1); // returns 1
obj.increment(3); // returns 4
```

this

- In methods `this` will be bound to the object

```
var o = { oldProp: 'this is an old property' };
o.aMethod = function () {
  this.newProp = "this is a new property";
  return Object.keys(this); // will contain 'newProp'
}
o.aMethod(); // will return ['oldProp','aMethod','newProp']
```

- In non-method functions:
  - `this` will be the global object
  - Or if `"use strict"; this` will be `undefined`

functions can have properties too

```
function plus1(value) {
  if (plus1.invocations == undefined) {
    plus1.invocations = 0;
  }
  plus1.invocations++;
  return value + 1;
}
```

- `plus1.invocations` will be the number times function is called
- Acts like static/class properties in object-oriented languages

function are objects: Have methods

```
function func(arg) { console.log(this,arg); } // this -> window object
```

- `toString()` method - return function as source string
  - `func.toString()` returns `'function func(arg) { console.log(this,arg); }'`

- `call()` method - call function specifying `this` and arguments
  - `func.call({t: 1}, 2)` prints `'{ t: 1 } 2'`
  - `apply()` like `call()` except arguments are passed as an array - `func.apply({t: 2},[2])`
  - `this` is like an extra hidden argument to a function call and is used that way sometimes

- `bind()` method - creates a new function with this and arguments bound
  - `let newFunc = func.bind({z: 2}, 3);`
  - `newFunc()` prints `'{ z: 2 } 3'`

Object-oriented programming: classes

Functions are classes in JavaScript: Name the function after the class

```
function Rectangle(width, height) {
  this.width = width;
  this.height = height;
  this.area = function() { return this.width*this.height; }
}
var r = new Rectangle(26, 14);  // {width: 26, height: 14}
```

Not correct way of adding methods

Functions used in this way are called **constructors**:

```
r.constructor.name == 'Rectangle'
console.log(r) // Rectangle { width: 26, height: 14, area:
               // [Function] }
```

Object-oriented programming: inheritance

- Javascript has the notion of a **prototype** object for each object instance
  - Prototype objects can have prototype objects forming a **prototype chain**



- On an object property read access JavaScript will search the up the prototype chain until the property is found
  - Effectively the properties of an object are its **own** property in addition to all the properties up the prototype chain. This is called prototype-based inheritance.
- Property updates are different: always create property in object if not found

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(26, 14);  // {width: 26, height: 14}
var v = r.area(); // v == 26*14
Object.keys(r) //== [ 'width', 'height' ] // own properties
```

Note: Dynamic - changing prototype will cause all instances to change

Prototype versus object instances

```
var r = new Rectangle(26, 14);
```

Understand the difference between:

```
r.newMethod = function() { console.log('New Method called'); }
```

And:

```
Rectangle.prototype.newMethod =
            function() { console.log('New Method called 1'); }

r.newMethod(); // New Method called
```

Inheritance

```
Rectangle.prototype = new Shape(...);
```

- If desired property not in `Rectangle.prototype` then JavaScript will look in `Shape.prototype` and so on.
  - Can view prototype objects as forming a **chain**. Lookups go up the prototype chain.
- Prototype-based inheritance
  - Single inheritance support
  - Can be dynamically created and modified

ECMAScript version 6 extensions

```
class Rectangle extends Shape {          // Definition and Inheritance
  constructor(height, width) {
    super(height, width);
    this.height = height;
    this.width = width;
  }
  area() {                               // Method definition
    return this.width * this.height;
  }
  static countRects() {                  // Static method
    ...
  }
}
var r = new Rectangle(10, 20);
```

React.js example class

```
class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    ...
  }
  render() {
    return (
      <div>Hello World</div>
    );
  }
}
```

React.js example class

```
class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    this.clickHandler = this.clickHandler.bind(this);  // What does this do?
    ...
  }
  render() {
    return (
      <div onClick={this.clickHandler}>Hello World</div>
    );
  }
}
```

Functional Programming

- Imperative:

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

- Functional:

```
newArr = anArr.map(function (val, ind) {
  return val*ind;
});
```

- Can write entire program as functions with no side-effects

```
anArr.filter(filterFunc).map(mapFunc).reduce(reduceFunc);
```

Functional Programming - ECMAScript 6

- Imperative:

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

- Functional:

```
newArr = anArr.map((val, ind) => val*ind); // Arrow function
```

- Can write entire program as functions with no side-effects

```
anArr.filter(filterFunc).map(mapFunc).reduce(reduceFunc);
```

Arrow functions don't redefine `this`

Some methods of **`Array.prototype`**

`Array.prototype.filter()` creates a new array with all elements that pass the test implemented by the provided function.

`Array.prototype.map()` **creates a new array** with the results of calling a provided function on every element in the calling array.

`Array.prototype.reduce()` executes a **reducer** function (that you provide) on each element of the array, resulting in a single output value.

```
var words = ['spray', 'limit', 'elite', 'exuberant', 'destruction', 'present'];
const result = words.filter(word => word.length > 6);
console.log(result);
// expected output: Array ["exuberant", "destruction", "present"]
```

Can mostly but not totally avoid functional style

- Asynchronous events done with **callback functions**

  Browser:

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

  Server:

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

- Node.js programming: Write function for HTTP request processing
- React's JSX prefers functional style: map(), filter(), ?:

Closures

An advanced programing language concept you need to know about

```
var globalVar = 1;
function localFunc(argVar) {
  var localVar = 0;
  function embedFunc() {
    return ++localVar + argVar + globalVar;
  }
  return embedFunc;
}
var myFunc = localFunc(10); // What happens if a call myFunc()? Again?
```

- `myFunc` **closure** contains `argVar, localVar` and `globalVar`

Using Scopes and Closures

- Using scopes and closures to create modules in early JavaScript

Global

```
var i = 1;
...
function f(){
  i++;
  return i;
}
```

Versus

```
return (function () {
  var i = 1;
  ...
  function f(){
    i++;
    return i;
  }
  return f;
})();
```

Not globals

immediate function: a scope *sandbox* for some code (mostly initialization e.g., attaching event handlers, creating objects) – Immediately Invoked Function Expression

Using closures for private object properties

```
var myObj = (function () {
  var privateProp1 = 1; var privateProp2 = "test";
  var setPrivate1 = function (val1) { privateProp1 = val1; }
  var compute = function () { return privateProp1 + privateProp2; }
  return { compute: compute, setPrivate1: setPrivate1 };
})();
typeof myObj;   // 'object'
Object.keys(myObj); //  [ 'compute', 'setPrivate1' ]
```

What does myObj.compute() return?

Beware of `this` and nested functions

```
'use strict';
function readFileMethod() {
  fs.readFile(this.fileName, function (err, data) {
    if (!err) {
      console.log(this.fileName, 'has length', data.length);
    }
  });
}
var obj = { fileName: "aFile", readFile: readFileMethod };
obj.readFile();
```

- Generates error on the `console.log` state since `this` is undefined

Beware of `this` and nested functions - work around

```
'use strict';
function readFileMethod() {
  fs.readFile(this.fileName, (err, data) => {
    if (!err) {
      console.log(this.fileName, 'has length', data.length);
    }
  });
}
var obj = { fileName: "aFile", readFile: readFileMethod };
obj.readFile();
```

- Works since an arrow function doesn't smash `this`

Closures can be tricky with imperative code

```
// Read files './file0' and './file1' and return their length
for (var fileNo = 0; fileNo < 2; fileNo++) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
```

- Ends up printing two files to console both starting with:

  `file 2 has length XX`

  Why?

Stepping through the execution

```
for (var fileNo = 0; fileNo < 2; fileNo++) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
```

Execution starts here: `fileNo`= 0

Stepping through the execution

```
for (var fileNo = 0; fileNo < 2; fileNo++) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
```

Call the function `fs.readFile`, before we can we must evaluate the arguments: the first argument results from the string concatenation operation forming `"./file0"`, the second argument is a function which is passed as a function and its closure containing the variables accessed by the function. In this case only `fileNo` is accessed by the function so the closure contains `fileNo` (which is currently 0).

Stepping through the execution

```
for (var fileNo = 0; fileNo < 2; fileNo++) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
```

Note that `fs.readFile` returns after it has started reading the file but before it has called the callback function. The execution does the `fileNo++` and calls back to `fs.readFile` with an argument of `"./file1"` and a new closure and function. The closure has only `fileNo` (which is currently 1).

Stepping through the closure example

```
for (var fileNo = 0; fileNo < 2; fileNo++) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
```

After creating two function with closures and calling fs.readFile twice the `for` loop finishes. Some time later in the execution the file reads will finish and `fs.readFile` will call the functions we passed. Recall that `fileNo` is now 2.

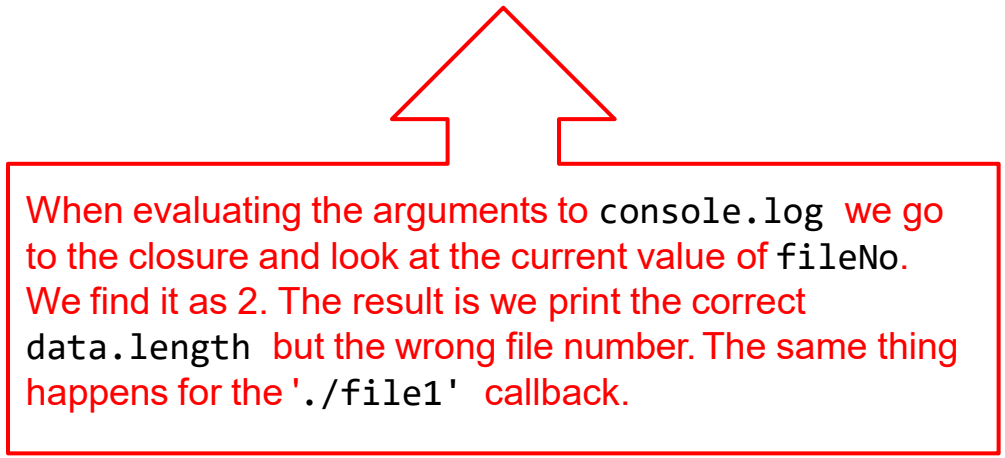
Sometime later: `file0` read finishes...

```
for (var fileNo = 0; fileNo < 2; fileNo++) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
```

`'./file0'` is read so our callback starts executing `err` is falsy so we go to the `console.log` statement.

Running callbacks....

```
for (var fileNo = 0; fileNo < 2; fileNo++) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
```



Broken fix #1 - Add a local variable

```
for (var fileNo = 0; fileNo < 2; fileNo++) {
    var localFileNo = fileNo;
    fs.readFile('./file' + localFileNo, function (err, data) {
       if (!err) {
          console.log('file', localFileNo,'has length',data.length);
       }
   });
}
```

Closure for callback now contains `localFileNo`. Unfortunately when the callback functions run `localFileNo` will be **1**. Better than before since one of the printed lines has the correct fileNo. 😀

A fix - Make fileNo an argument

```
function printFileLength(fileNo) {
  fs.readFile('./file' + fileNo, function (err, data) {
    if (!err) {
      console.log('file', fileNo, 'has length', data.length);
    }
  });
}
for (var fileNo = 0; fileNo < 2; fileNo++) {
  printFileLength(fileNo);
}
```

Note: This works but sometimes it prints the `file0` line first and sometimes it prints the `file1` line first.

JavaScript Object Notation (JSON)

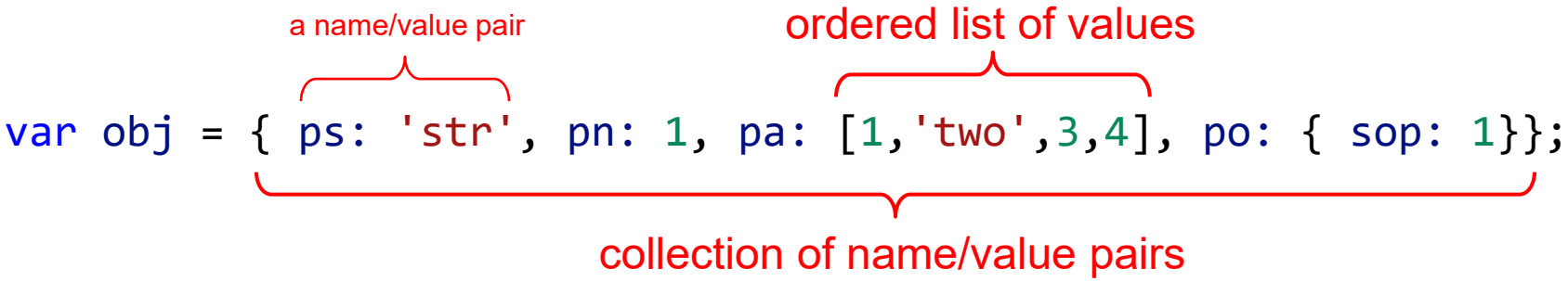
a lightweight data-interchange format

- easy for humans to read and write
- easy for machines to parse and generate
- subset of the JavaScript Programming Language

built on two structures

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

JavaScript Object Notation (JSON)




```
var s = JSON.stringify(obj) // to convert to string
     // '{"ps":"str","pn":1,"pa":[1,"two",3,4],"po":{"sop":1}}'
typeof s == 'string'

JSON.parse(s) // returns object with same properties
```

- JSON is the standard format for sending data to/from a browser

JavaScript: The Bad Parts

Declaring variables on use - Workaround: Force declarations

```
var myVar = 2*typeoVar + 1;
```

Automatic semicolon insertion - Workaround: Enforce semicolons with checkers

```
return
    "This is a long string so I put it on its own line";
```

Type coercing equals: `==` - Workaround: Always use `===`, `!==` instead

`("" == "0")` is `false` but `(0 == "")` is `true`, so is `(0 == '0')`

`(false == '0')` is `true` as is `(null == undefined)`

`with`, `eval` - Workaround: Don't use

Some JavaScript idioms

- Assign a default value

```
hostname = hostname || "localhost";
port = port || 80;
```

- Access a possibly undefined object property

```
var prop = obj && obj.propname;
```

- Handling multiple `this`:

```
fs.readFile(this.fileName + fileNo, function (err, data) {
  console.log(this.fileName, fileNo); // Wrong!
});
```

Some JavaScript idioms

- Assign a default value

```
hostname = hostname || "localhost";
port = port || 80;
```

- Access a possibly undefined object property

```
var prop = obj && obj.propname;
```

- Handling multiple `this`: `self`

```
var self = this;

fs.readFile(self.fileName + fileNo, function (err, data) {
  console.log(self.fileName, fileNo);
});
```

Some JavaScript idioms

- Assign a default value

```
hostname = hostname || "localhost";
port = port || 80;
```

- Access a possibly undefined object property

```
var prop = obj && obj.propname;
```

- Handling multiple `this`:

```
fs.readFile(this.fileName + fileNo, (err, data) =>
  console.log(this.fileName, fileNo)
);
```