JavaScript Programming
Mendel Rosenblum
How do you program in JavaScript?

From Wikipedia:

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

... 

- Mostly 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

```javascript
var o = {count: 0};
o.increment = function (amount) {
    if (amount == undefined) {
        amount = 1;
    }
    this.count += amount;
    return this.count;
}
```

- Method invocation:

```javascript
o.increment();   // returns 1
o.increment(3);  // returns 4
```
**this**

- In methods this will be bound to the object

```javascript
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

```javascript
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
Object-oriented programming: classes

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

```javascript
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}
```

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

```javascript
r.constructor.name == 'Rectangle'
```

```javascript
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
  ● Can lead to fun in AngularJS
Using prototypes

```javascript
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'); }
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
```javascript
class Rectangle extends Shape { // Definition and Inheritance
    constructor(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);
```
Functional Programming

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

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

- Can write entire program as functions with no side-effects
  ```javascript
  anArr.filter(filterFunc).map(mapFunc).reduce(reduceFunc);
  ```
Can mostly but not totally avoid functional style

- Asynchronous events done with callback functions

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

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

An advanced programming language concept you need to know about

```javascript
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

- Consider effect on the scopes of:

```javascript
var i = 1;

(function () {
    var i = 1;
})();
```

Versus

```javascript
var i = 1;

(function () {
    var i = 1;
})();
```
Using closures for private object properties

```javascript
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?
```
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

Why?
Stepping through the execution

```javascript
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
for (var fileNo = 0; fileNo < 2; 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++) {
    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

```javascript
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...

```javascript
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.

```javascript
```
Running callbacks....

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

When evaluating the arguments to `console.log` we go to the closure and look at the current value of `fileNo`. We find it as 2. The result is we print the correct `data.length` but the wrong file number. The same thing happens for the './fileNo1' callback.
Broken fix #1 - Add a local variable

```javascript
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

```javascript
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)

var obj = { ps: 'str', pn: 1, pa: [1,'two',3,4], po: { sop: 1}};

var s = JSON.stringify(obj) =
   '{"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 and from a browser
JavaScript: The Bad Parts

Declaring variables on use - Workaround: Force declarations

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

Automatic semicolon insertion - Workaround: Enforce semicolons with checkers

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

- Access a possible undefined object property
  ```javascript```
  var prop = obj && obj.propname;
  ```javascript```

- Handling multiple `this`:
  ```javascript```
  fs.readFile(this.fileName + fileNo, function (err, data) {
    console.log(this.fileName, fileNo);  // Wrong!
  });
  ```javascript```
Some JavaScript idioms

- Assign a default value
  
  ```javascript
  hostname = hostname || "localhost";
  port = port || 80;
  ```

- Access a possible undefined object property
  
  ```javascript
  var prop = obj && obj.propname;
  ```

- Handling multiple `this`: self
  
  ```javascript
  var self = this;
  fs.readFile(self.fileName + fileNo, function (err, data) {
    console.log(self.fileName, fileNo);
  });
  ```