Everything You Do Not Know About JavaScript

COMPSCI 220

University of Massachusetts Amherst
Adapted from the syllabus:

- First-class functions
- Design patterns that use higher-order functions and objects (together and in isolation)
- An accurate mental model of programs that use mutable state, objects, higher-order functions, and garbage collection
- Testing strategies: unit testing, property-based testing, and randomized testing

None of this material is JavaScript-specific. We could have taught this class using a wide variety of languages, from popular languages such as Python and Ruby to arcane languages such as Haskell and Scheme.

JavaScript-specific concepts covered in this class:

- The `this` keyword in JavaScript, which is unlike the `this` keyword in other languages.
- Anything else?
Inheritance and Classes in JavaScript
Experts agree that *implementation inheritance* is often a bad idea:

- Several newer programming languages deliberately do not support implementation inheritance: Rust, Haskell, Julia.
- In languages that do support inheritance, you should “favor composition over inheritance”. (Lots of blog posts and books use this phrase to explain why.)
This example shows that the dog object inherits from the animal object. Notice that there are no classes involved.

```javascript
let animal = { length: 13, width: 7 };  
let dog = { __proto__: animal, barks: true };  
console.log(dog.barks); // prints barks  
console.log(dog.width); // prints 7
```

The __proto__ field is a special field name, that contains the prototype of an object. If an object does not specify __proto__, it is set to a builtin default that has common methods such as toString.

An object can explicitly set __proto__ to null, in which case toString is not inherited.

```javascript
let x = { };  
let y = { __proto__: null };  
console.log(x.toString()); // prints '[object Object]'  
console.log(y.toString()); // raises an error
```
Prototype Inheritance Pitfalls

- An object’s prototype can change, which changes what it inherits:

```javascript
let dog = { __proto__: animal };  
dog.__proto__ = null;
```

- Prototype inheritance does not affect updates:

```javascript
let A = { x: 100 };  
let B = { __proto__: A };  
let C = { __proto__: A };  
B.x = 200;  
console.log(C.x);  // prints 100
```
Classes in JavaScript

JavaScript does not have classes. It fakes classes using prototypes, but you can see through the fake.

class A {
    constructor() { }
    method1() {
        console.log("In method1");
    }
}

class B extends A {
    constructor() { }
}

This is really prototype inheritance under the hood:

let o = new B();
o.__proto__ = null;
o.method1(); // throws an exception
Ordinary functions can be used as constructors:

```javascript
function A() {
  this.x = 100;
}
let o = new A();
console.log(o.x); // prints 100
```

But, `A` is also an ordinary function:

```javascript
A(); // no errors
console.log(x); // new global variable x!
```
Functions in JavaScript
The *arity* of a function is the number of arguments it takes.

An *arity mismatch error* occurs when a function receives the wrong number of arguments. In statically typed languages (e.g., Java), arity mismatch errors occur before the program is run. In dynamically typed languages, arity mismatch errors occur while the program is running.

JavaScript does not have arity mismatch errors.

```javascript
function F(x) {
    console.log(x);
}
F(1, 2); // prints 1. The 2 is ignored.
F(); // prints undefined.
```

**Note:** Ocelot adds arity mismatch errors to JavaScript.
All the arguments of a function are available in the `arguments` object.

```javascript
function F(x) {
    console.log(x);
    for (let i = 0; i < arguments.length; i++) {
        console.log(arguments[i]);
    }
}

F(1,2,3,4); // prints 1 1 2 3 4
```

Notice that `arguments` contains all arguments and not just the extra arguments.

**Question:** What does this code print?

```javascript
function F(x) {
    arguments[0] = 200;
    console.log(x);
}

F(100); // prints 200 or 100?
```

**Answer:** 200 in *non-strict mode* (the default) and 100 in *strict mode* (added to JS in 2009).
Implicit Conversions in JavaScript
The binary operators of JavaScript implicitly coerce their arguments in a variety of ways.

For example, `true` is coerced to `1` and `false` is coerced to `0` by arithmetic operators:

- `false + true` // 1
- `true + true` // 2
- `true << 10` // 1024

Ocelot disables most of these coercions.
What is Wrong with JavaScript?
JavaScript 1.0: built in May 1995 in 10 days.

Today, JavaScript has several powerful, well-designed new features and is rapidly evolving. The JavaScript standard (known as ECMAScript) is designed by an open standards body called TC-39 (https://github.com/tc39).

Newer versions of JavaScript cannot abandon old (bad) features, because old web sites may depend on them. Don’t break the web is a guiding principle of the language design.
All Programming Languages Have Badly Designed Features

A few references:


- Ross Tate and Nada Amin. Java and Scala’s Type Systems are Unsound. Language. *ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2016. (http://io.livecode.ch/learn/namin/unsound)

How to Avoid JavaScript’s Badly Designed Features

- **Linters**: such as ESLint (https://eslint.org/). Only catches trivial errors. Does not catch most of the crazy behavior we have shown.

- **Ocelot**: changes the semantics of JavaScript to eliminate its worst features.

  Code written in Ocelot will work outside class, with the exception of lib220 (obviously). However, the lib220 functions are easy to replace.

- **Overlay type systems**: Microsoft TypeScript and Facebook Flow add statically-typed languages that look like JavaScript, compile to JavaScript, but are not JavaScript.

  Facebook is built with Flow (and TypeScript). Office Online is built in TypeScript. Ocelot is built in TypeScript.

- **Do not use JavaScript** (but sometimes you have to).
This is an example of TypeScript:

```typescript
function map[A,B](f: (x: A) => B, arr: A[]): B[] {
    let result: B[] = [];
    for (let i = 0; i < arr.length; i++) {
        result.push(f(a[i]));
    }
    return result;
}
```

Notice that the type annotations are identical to the notation we use in COMPSCI220.

The TypeScript compiler type-checks the program and then erases the annotations to get an ordinary JavaScript program.

The TypeScript type system is very sophisticated and can type-check code that would not type-check in Java (or most other type-checkers).

```typescript
function f(x: number | undefined): number {
    if (typeof x !== 'number') {
        x = 0;
    }
    return x + 10; // TypeScript knows that x: number
}
```
Sometimes you have to use JavaScript (e.g., web programming). You can still avoid it by using a language that compiles to JavaScript.

- **Elm** a statically typed and takes inspiration from Haskell (https://elm-lang.org).
- **Reason** a statically typed and takes inspiration from OCaml (https://reasonml.github.io/). *Most of Facebook Messenger is built with Reason.*
- Almost any programming language that you can think of has a compiler to JavaScript. The general problem of compiling to JavaScript is challenging.


- **WebAssembly**: A new assembly language” for the web.
Conclusion
Where to Go From Here

- Web programming with JavaScript (COMPSCI 326, Spring 2020)
- Programming language design and implementation (COMPSCI 497P, Fall 2019)
- Robotics, Planning (COMPSCI 403, Fall 2019)