Prologue:
JAVASCRIPT PRIMER
(a.k.a., *80-Minute JavaScript™*)

All of the projects in this class will be done in JavaScript, and we expect you to pick up the language as we go along. This document is a short introduction to get you started.

Motivation

Why would you prototype a language in JavaScript?

- So that people can try your language without having to download and install a compiler, VM, etc.
- To have easy access to the web browser, which is a rich environment with lots of opportunities for *domain-specific languages* (DSLs).
- Because JavaScript is a powerful and flexible language with decent support for OO and functional programming.

Objects

JavaScript's objects are dictionaries that map property names to values.*

* Property names are `Strings`.

You **create a new objects** by calling a constructor, e.g., `new Object()`, or by writing an object literal in JavaScript's convenient JSON syntax, e.g., `var myPoint = {x: 1, y: 2};`.

To **access properties**, you use the `.` operator, e.g., `myPoint.x` and `myPoint.y = 5`. Alternatively, you can use square brackets, e.g., `myPoint["x"]` and `myPoint["y"] = 5` — this is useful when you don't know the name of the property you want to access statically.

You can also **remove a property** using the `delete` statement, e.g., `delete myPoint.x;`.

Delegation

JavaScript is a *prototype-based* language. Every object has a (single) prototype from which it can inherit properties. E.g., the prototype of all `Strings` is `String.prototype` — this is where methods like `indexOf` and `substr` are defined (more on these below).

When people say that an object *delegates to* its prototype, they're talking about the **property look-up** operation. Here's how *prototype chain* is used for property look-up: if you look up a property `p` in `obj` and `obj` doesn't have it, JavaScript will automatically look it up in `obj`'s prototype. And if it's not there either, it will look it up in `obj`'s prototype's prototype, and so on. At the top of the prototype chain is the *object prototype* (`Object.prototype`); if the search for the value of `p` reaches `Object.prototype` and it doesn't have a value for that property, `obj.p` will evaluate to the special value `undefined`.

Properties are *copy-on-write*, which means that `obj.p = v` will always write to `obj` directly, even if before this assignment `obj` was *inheriting* the value of its `p` property from some other object up its prototype chain.

Functions

To **declare a function**, you use the `function` keyword, e.g., `function add(x, y) { return x + y; }`. JavaScript also lets you write **anonymous functions**, e.g., `function(x, y) { return x + y; }`.

Functions are *first-class*, which means you can store them in variable / properties / arrays, return them from a function / method, etc.

A function can reference variables from its enclosing environment, e.g.,

```
function makeCounter() {
  var count = 0;
  return function() { return count++; };
}
```

You can write **variadic functions**, i.e., functions that takes a variable number of arguments, using the `arguments` pseudo-variable, e.g.,

```
function sum(/* a, b, c, ... */) {
  var ans = 0;
```

```
  for (var idx = 0; idx < arguments.length; idx++) {
        ans += arguments[idx];
  }
  return ans;
}
```

(Annoyingly, `arguments` is not a real array. This is often a source of confusion, and it means that `arguments` doesn't support useful `Array` operations like `map`. If you need to, you can create a real array that holds the arguments like this: `Array.prototype.slice.call(arguments)`.

More on the `slice` and `call` methods later.

An Important Note on Scoping

JavaScript has *lexical scoping*, but it doesn't work the way you'd expect. You see, in most languages that are lexically-scoped, a *block statement* is a lexical scope. In JavaScript, the only thing that is a lexical scope is a function. So when you write code like this:

```
function f(x) {
  if (x > 5) {
    var y = x * x;
    ...
  }
  ...
}
```

what it really means is:

```
function f(x) {
  var y;
  if (x > 5) {
    y = x * x;
    ...
  }
  ...
}
```

While this is usually isn't a big deal, it's something to keep in mind. If you're used to shadowing variable declarations, get ready to spend countless hours debugging your code and cursing Brendan Eich!

Methods

A method is just a function that's stored in a property, e.g.,

```
var aPoint = {
  x: 0,
  y: 0,
  toString: function() { return '(' + this.x + ',' + this.y + ')'; }
};
```

When you call a method, e.g., `obj.m(arg1, arg2, ...)`, JavaScript will:

- evaluate `obj` (the *receiver*),
- look up the function stored in `obj`'s `m` property, and
- call that function with the arguments you supplied, and with `this` bound to the receiver.

Important: when you call a function the usual way, e.g., `f(1, 2)`, `this` is bound to the *global object* (`window`), which is the object that represents the top-level lexical scope. This often leads to bugs: e.g., if a helper function inside a method references `this`, it's not what you'd expect. Here's a common work-around for this problem:

```
var myObj = {
  ...
  someMethod: function() {
    // Store the receiver in a variable, so that it can be accessed by nested functions.
    var self = this;
    function helper() {
      ... self.someOtherMethod() ...
    }
    ... helper() ...
  }
};
```

Many ways to call a function

So far we've seen two ways to call a function,

- the "function" way, e.g., `f(1, 2)`, and

- the "method" way, e.g., `obj.m(1, 2)`

and we've seen what happens with `this` in each of these types of function call.

There are two more ways to call a function in JavaScript that are worth mentioning. The first is via the function's `call` method, which is useful because it lets you specify the object that should be bound to `this` when the function's body is evaluated, e.g., `f.call(objToUseAsTheReceiver, 1, 2)`.

The other way is via the function's `apply` method, which is like `call` except that you pass the arguments as an array, e.g., `f.apply(objToUseAsTheReceiver, [1, 2])`. This is useful when the number of arguments that you want to pass to the method is not known statically.

"Classes"

You can get much of the functionality of classes in JavaScript using functions. Here's how that works:

- First, you declare a function that does what the constructor of your class would do, e.g., `function Point(x, y) { this.x = x; this.y = y; }`.
- When you use that function as a constructor, e.g., `new Point(1, 2)`, JavaScript creates a new object that delegates to `Point.prototype` and binds it to `this` for the execution of the function's body. By default, all functions have a `prototype` property that is initialized with an empty object.
- This means that you can store whatever methods and/or default values you would like `Point` instances to have in `Point.prototype`, e.g.,

```
Point.prototype.toString = function() {
  return "(" + this.x + "," + this.y + ")";
};
```

Here's how you do inheritance:

```
function Person() { ... }
function Student() { ... }
Student.prototype = new Person();
```

Get it? And here's how you do a super-send:

```
Student.prototype.toString() {
  Person.prototype.toString.call(this);  // like super.toString(), in Java.
  ...
};
```

For Reference

Functional Programming

- `[1,2,3].map(function(x) { return x + 1; })` evaluates to `[2,3,4]`.
- `[1,2,3].reduce(function(x, y) { return x + y; }, 0)` evaluates to 6.
- `[1,2,3].filter(function(x) { return x > 1; })` evaluates to `[2,3]`.
- I said this before, but it bears repeating: **be careful about `this`!** If you call any of these methods inside a method, the function that you give as an argument will be called with `this` bound to the global object, which is pretty much never what you want. The way around this problem is to store `this` in a local variable (people usually call it `self`) and reference the receiver through that variable. (I've been bit by this bug *hundreds* of times. Literally.)

Meta Stuff

- In JavaScript, you can dynamically compile and evaluate a program using the `eval` function. Calling `eval` is generally frowned upon (it's a huge security hole!) but it can be very useful when you're prototyping a language via *source-to-source translation*. We'll talk about source-to-source translation later in the course.
- To test if `obj` has a property `p` that's not inherited from its prototype: `obj.hasOwnProperty("p")`.
- To get an array containing all of `obj`'s "own" property names: `Object.keys(obj)`.
- To create an object that delegates to another object: `var newObject = Object.create(anotherObject)`. You can also pass an additional argument that is an object with the properties that the new object should have, e.g., `Object.create(anotherObject, {foo: 5, bar: 6})`.
- To declare a method that can be accessed like a property:

```
Object.defineProperty(obj, "p", {
  get: function() { ... },
  set: function(value) { ... }
```

```
});
```

Numbers

- All numbers in JavaScript are double-precision floating point numbers.
- To test if n is a number: `typeof n === "number"`.
- Note that the test above will evaluate to `true` even for values like `Number.POSITIVE_INFINITY` and `Number.NaN`.
  - To test if a number n is finite: `Number.isFinite(n)`.
  - To test if a "number" n is not a number: `Number.isNaN(n)`.
- To convert a `String` to a number, you can use JavaScript's `parseInt` and `parseFloat` functions. Both of these functions takes a `String` as an argument, and return corresponding number, or `NaN` if the argument can't be parsed.

Arrays

- To test if `arr` is an `Array`: `Array.isArray(arr)`.
- Number of elements in `arr`: `arr.length`.
- You can truncate or grow the array but assigning into `length`.
- Access the `i`th element: `arr[i]`.
  - `undefined` if `i` is out of bounds.
- Set the `i`th element: `arr[i] = value`.
  - If `i` ⩾ `arr.length`, `arr.length` is updated automatically.
- Add a `newElement` to the end: `arr.push(newElement)`.
- Remove the last element: `arr.pop()`
- Add `newElement` to the beginning: `arr.unshift(newElement)`
- Remove the first element: `arr.shift()`.
- Insert `newElement` at `idx`: `arr.splice(idx, 0, newElement)`.
- Iterate over the array: `arr.forEach(function(x) { ... })`.

Strings

- To test if an object is a `String`: `typeof obj === "'string"`.
- JavaScript `Strings` are immutable.
- You can write string literals with single- or double-quotes, e.g., `"hello world"` and `'foo'` are both valid string literals.
- The `length` property of a string tells you how many characters are in it.
- `String` indices are `0`-based.
- To access a character, you either use square brackets (`s[5]`) or the `charAt` method `` `s.charAt(5)``).
  - Note that the value of these expressions isn't really a character (there's no such thing in JavaScript) but rather a `String` of length 1.
- Other useful methods:
  - `s.indexOf(anotherString)` returns the index of the first occurrence of `anotherString` in `s`, or `-1` if it's not found.
  - `s.substr(startIdx, length)` returns a substring of `s`.
  - ...