ECS 162
WEB PROGRAMMING

4/12

Javascript is evolving

- Javascript started out in 1995 as a small scripting language for the Netscape browser (the ancestor of Firefox).
- It is now the most-used programming language in the world. Why?

Javascript is evolving

- We are now on ES8, the eighth standardized version (ECMAScript8).
- There was a big step up from ES5 to ES6, so we casually refer to "modern Javascript" as ES6.
- Browser support always lags releases, and there are slight differences, but we will not worry about it.

Javascript is evolving

- ES6 has a lot of interesting features.
- We will concentrate on features we need for communication over the internet.
- Outside of those, in lecture and labs we will stick to basic features found in almost all languages.
- You are welcome to use features outside our subset in the homework, and discuss them on Piazza!

Everyone has a subset

- Javascript gives you a lot of freedom.
- Also it has a huge collection of legacy and new features.
- This book is a classic on picking and choosing Javascript features.

Strict mode

- The original Javascript was intended to be super-easy to write.
- Like the rest of the browser code, the interpreter is tolerant of mistakes.
- Example: semi-colons are required, but your code runs (often but not always correctly…) without them.
- In "strict mode", some of the things that caused frequent errors are not allowed (in particular relaxed rules about variable declarations that led to accidental global variables).

Getting into strict mode

- This line is the first thing in your Javascript file:
  "strict mode";
- Why isn't strict mode the default? Why do we have to request it?

Getting into strict mode

- This line is the first thing in your Javascript file:
  "strict mode";
- Why isn't strict mode the default? Why do we have to request it?
  - Because browsers have to run a lot of legacy code!

Function statement

```
function f () {
    let r = 1;
    return r;
}
let a = f();
let b = f;
let c = b();
```

- What is in a? b? c?

Function assignment

```
let a = f();
let b = f;
let c = b();
```

- a contains the number one.
- b contains the function f.
- c contains the number one.

Function expression

- Functions (like everything else) are objects in Javascript.
- So you can put them into variables, pass them as parameters to other functions, etc.

```
let a = function f () {
    let r = 1;
    return r;
}
```

Function expressions are an alternative way to define a function.

Function expression

```
let a = function () {
    let r = 1;
    return r;
}
```

- Here "function" is an expression that returns a function, which gets put into variable a.
- On the right-hand side, it does not have a name yet. It is anonymous.

Anonymous functions

```
let a = function(f) { return f(3); }
a( function (b) {return b+1} )
```

- What does this do? And what is going on?

Anonymous functions

```
let a = function(f) { return f(3); }
a( function (b) {return b+1} )
```

- What does this do? And what is going on?

a is a function that takes another function as input.
We call a on an anonymous function, which adds one to its input.
So the value returned by the second line is 4.

Arrow functions

- There is a third function declaration syntax in ES6, called arrow functions:

```
let times = (x,y) => { return x * y;}
let square = x => { return x * x;}
```

- parameters => function body block
- Shorter than the function keyword
- Mostly used as shorthand. I will mostly not use it.
- Know it when you see it.

Variables

- Three kinds of variable declaration:

```
let a = 1;  // "the usual"
var a = 1;  // visible throughout function
const a = 1;  // cannot be changed; usual scope
```

- Can also declare without initializing

```
let a;  // a now contains "undefined"
```

Scope Example

```
const f = function () {
        let id = "Ralph";
        if (true) {
                let id = "Molly";
                console.log(id);
        }
        console.log(id);}
f();
```

- Prints "Molly", then "Ralph"
- It would be much better to give these two variables different names.

Scope Example

```
const f = function () {
        let id = "Ralph";
        if (true) {
                let id = "Molly";
                console.log(id);
        }
        console.log(id);}
f();
```

- Two blocks. The scope of a variable is the block within which it is visible (recognized as declared).

Scope Example

```
const f = function () {
        // let id = "Ralph";
        if (true) {
                var id = "Molly";
                console.log(id);
        }
        console.log(id);}
f();
```

- Prints "Molly", then "Molly".

Scope Example

```
const f = function () {
        let id;
        if (true) {
                id = "Molly"
                console.log(id);
        }
        console.log(id);}
f();
```

- Exactly the same behavior as previous slide.

When to use var?

- Almost never.
- It used to be the only choice. It was for the convenience of the interpreter, not the programmer. Good riddance!
- Much of the code you see on Stack Exchange, etc, uses "var". Not changing it to "let" is a good indication that you are plagiarizing instead of learning.

Function hoisting

- Function variables declared using the function expression:

  `let f = function () {}`

  have the usual scope rules.
- Function variables declared using the function statement:

  `function f () {}`

  Are silently "hoisted" to the top of the scope they are in.

Example

```
f();
function f () {
        let id = "Ralph";
        console.log(id);
}
```

- Works fine.
- Changing the function declaration to a function expression causes an error.
- Should you use this function statements?

Pros and Cons

- Should you use function statements?

- Pro:

  Lets your code flow more nicely. Eg. "main" can go at the top and serve as an outline.

- Con:

  Similar to var, allowing things to be used before they are defined might lead to bugs. Also, function expressions look ugly.