KINDS OF JS

Javascript has no relation to Java

- Name was a marketing deal
- We all still suffer as a result

Each browser implemented, incompatible versions

- Browser Wars caused a massive lull in dev
- Evergreen browsers + following standards changed this

We will start with some basic JS that can run in modern browsers AND NodeJS

ECMASCRIPT

Standards now run by ECMA

- JS standard is "ECMAScript"

Versions were once numbered

- ES3 was around forever
- ES4 was given up on and tossed

Versions now annual, but numbers still in use

- ES6 === ECMAScript2015
- ES7 === ECMAScript2016

STRICT MODE

Javascript is remarkably backwardly compatible.

However, that means a lot of poor practices can still work.

We will use "strict mode" to prevent this.

At the very first line of your JS file, have the line:

```
"use strict";
```

The quotes are part of the line

Even if I don't show it, do it.

RUNNING JS

In browser console, type a line.

At command line, run `node`, type a line.

Known as REPL - executes each line, state is maintained.

At command line, run `node file.js`

- Not REPL

EXPRESSIONS

An "expression" evaluates to a value.

- Can be a combination of statements that result in a value

```
5
```

```
5 + 4
```

```
5 * giveMeANumber()
```

```
foo = 5
```

SEMICOLONS

- Statement expressions are followed by semicolons
- Statements are NOT

Javascript CAN "guess" where you want a semicolon

- Known as ASI (Automatic Semicolon Insertion)

Some people embrace this and encourage skipping semicolons.

If you can't say the one case where ASI will fail you...Don't skip them.

- For this class, semicolons are REQUIRED

STATEMENTS

A "statement" does stuff.

```
if( someValue ) {
  doStuff;
}
```

- You can use an expression in place of a statement
- You cannot use a statement in place of an expression

HELLO WORLD

```
"use strict";

let message = 'Hello World';

console.log(message); // This is a comment
```

DECLARING VARIABLES

A variable must be declared before use. (strict mode only)

- `var`
  `myVar;`

There are a few ways to declare a variable:

- `var`
- `let`
- `const`
- `function` statement
- implicit in function declaration signature

BUT WHAT ABOUT TYPES?

Some languages are "statically typed"

- Means the variable is bound to a certain type of data

JS is "dynamically typed"

- Means the "type" is associated with the data, and the variable can refer to whatever

VAR

The `var` declaration:

- Should not be used unless you're targeting older JS engines
- Declares a variable within the scope of the current function
- "Hoists" (meaning a declaration, but not an initialization, acts as if it at the top of the function, regardless of line)

LET

The `let` declaration:

- Is scoped to the current BLOCK
- Has special treatment in a `for` initialization
- does NOT hoist (avoid the Temporal Dead Zone)

CONST

The `const` declaration:

- Is NOT just for "constants"
- Is block-scoped
- does NOT hoist
- Prevents reassigning the variable
- Does NOT prevent mutation of an array or object
  - ...because this doesn't change the variable, just a value within it

Use `const` everywhere you can. 80-90% of your variables should be `const`

VARIABLE BEST PRACTICES

- Always declare your variables before use
- Declare where relevant/meaningful
  - Usually the top of the block
- Never use `var` unless you have to
- Always prefer `const`
  - Seriously

FUNCTIONS CAN BE EXPRESSIONS OR STATEMENTS

```
function fromAStatement( someValue ) {
     console.log(someValue);
}
```

```
const fromAnExpression = function (someValue ) {
  console.log(someValue);
}
```

- Basically Identical
- Can add an internal name after `function` in second one to make even more the same
- The first one is hoisted though
- `function` statements basically declared as `var` - that's fine

JS TYPES

- Primitives (immutable!)
  - null
  - undefined
  - Boolean
  - Number
  - String
  - Symbol
- Object
  - Array
  - Function
  - Object versions of primitives

NULL AND UNDEFINED

JS has **two** values that mean "lack of a value"

- `null` means a `deliberately no value`
- `undefined` means `a lack of value that hasn't been assigned`

These are confusing and not always consistent

Rule of Thumb: Never explicitly use `undefined` except to compare

NUMBER

Most languages have many types for numbers - integers and floats, for example.

JS has "Number", covers both

```
const sum = 1 + 3;
const avg = sum/2;
console.log(sum, avg);
```

Note that all languages have issues with some floats

```
console.log(0.1 + 0.2);
```

BOOLEAN

```
const isTrue = true;
const isFalse = false;
```

STRINGS

JS has no "character" type, just strings

- Can be single-quoted
- Can be double-quoted
- Can use "backticks" (called "template literal")

Template literals

- Can be multi-line
- Allow for variable substitution

Remember strings are immutable. Changes are really just making new strings.

SYMBOLS

These are newer, not needed by newer coders

OBJECTS

Objects are a mutable datatype that can have multiple name properties that each have unrelated values.

Objects are the workhorse of JS.

Objects are a value

- can be stored in variables
- allows for nested structures.

Declared with `{}`

```
const someObj = {};
const someOtherObj = {
  someProp: 'someVal',
  otherProp: 3
};
```

ARRAYS

Arrays are Objects that have some special quirks

- Arrays are objects, but we usually talk about them separately

Arrays store a list of unnamed properties by **index**

- allows for ordered access
- properties do not have to be the same type
- properties can be any primitive or object type

Declared with `[]`;

```
const someArray = [ 1, 2, 3 ];
const mixedArray = [ 2, "even", {}, [] ];
```

FUNCTIONS ARE OBJECTS

A function is actually a callable object

As an object

- can have named properties
- can be passed or assigned as a value

Functions are "first class citizens" - equal to any other data type

OBJECT PRIMITIVES

Some of the primitives have object wrappers

```
const num = Number(100);
const bool = Boolean(true);
```

But most of the time they are just visual noise and can actually cause truthy/falsy confusion

For this class: AVOID object wrappers

COERCION

JS is dynamically typed, not statically typed.

JS is ALSO "weakly" typed, not "strongly" typed.

This means JS will "coerce" data to be the type it thinks you want.

- Only throws errors if it can't guess

```
const name = "Maru";
const age = 12;
const userId = name + age; // "Maru12"
```

TRUTHY-FALSEY

Coercion is usually risky and a bad idea

Exception: Coercing to a boolean value

```
const name = getUserName();
if( !name ) {
  console.log("I don't seem to have a name for you");
}
```

Using truthy/falsey values can make code more skimmable

WHAT IS TRUTHY

Anything not Falsy is truthy

Falsy values are:

- false -- (duh) but not "false"
- null -- but not "null"
- undefined -- but not "undefined"
- 0 -- but not '0'!
- " -- (empty string)
- NaN -- (Not a number, such as 1/"a")

COMPARISON

- = is assignment
- == is loose comparison
- === is *strict* comparison

```
let thing = '1';
if( thing = '1' ) {} // Always true!
if( thing == 1) {} // true
if( thing === 1 ) {} // false
```

Always use strict comparison unless you are using truthy/falsey

When using truthy/falsy, always consider if 0 is valid input

MORE ABOUT OBJECTS

Objects are the workhorse of JS

- Basis of all complex data structures
- Encompass functions(!) and arrays

DOT NOTATION

Object properties can be accessed by "dot-notation" using their keys

```
const obj = {
  one: 1,
  two: 2
};
console.log(obj.two); // 2
obj.three = 3;
console.log(obj.three); // 3
```

NAMED INDEX

Objects can also be accessed just like arrays, except with a named index instead of a numeric index

```
const grades = {
  amit: 84,
  bao: 97
};
const student = 'amit';
console.log( grades[student] ); // 84
```

OBJECTS AS HASHMAPS

Arrays store *ordered* data, but Objects are the best way to store *unordered* data

```
const students = {
  amit: 87,
  bao: 94
};
```

ONLY use arrays if

- the order matters
- or you will be going through all records every time

Otherwise you can use Object.keys(), Object.values(), and Object.entries() to convert an object to an array when needed.

OBJECT KEYS

Object keys are strings.

They can be unquoted when an object is declared.

But if the key string has formatting that would confuse the engine, they must be quoted in declaration, and cannot use dot-notation to access

```
const obj = {
  one: 1,
  'less-than-one': -1
};
console.log(obj['less-than-one']); // -1
```

WHAT IS A SCOPE?

A "scope" is a portion of code that uses the same access to the same set of variables.

JS uses "lexical scoping" and has three tiers of scope:

- Global Scope
- Function Scope
- Block Scope

Can be multiple instances of Function and Block scope

Lexical Scoping means that if a variable isn't defined in the current scope, the "containing" scopes are checked.

GLOBAL SCOPE

In a browser "global" scope is the same as "window" scope.

In NodeJS there is no "window", so it has a separate global scope.

As a general rule, you will only access existing global variables and avoid adding more.

In the browser there are a few exceptions.

FUNCTION SCOPE

A function is a block, so block-scoped variables that aren't inside other kinds of blocks act as function-scoped.

Function scoped variables are accessible to anything within that function.

```
const foo = 1;

function someFunc() {
  const foo = 2;
  console.log(foo);
}

someFunc();
console.log(foo);
```

BLOCK SCOPE

Blocks are the bodies of functions, `if`, `for`, `while`, and `try`/`catch` statements, plus the weirdness of `switch`/`case` statements and a few other places.

While you CAN use one-line versions of `if` statements, you shouldn't. Always use `{}`, even if it is one line.

This is to prevent someone from adding a second line without realizing that they need to add curlies to make that line part of the `if`.

Because of lexical scoping, blocks have access to all variables inside their containing function/block, etc.

WHAT IS A FUNCTION?

A function is a collection of callable code.

This allows for reuse and abstraction, both vital to programming.

Functions can return a value (function calls are expressions).

Function declarations can be statements OR expressions.

Functions are a type of object

- They can be passed, assigned, compared, like any other variable

DECLARING A FUNCTION

```
function fromAStatement( someValue ) {
     console.log(someValue);
}
```

```
const fromAnExpression = function (someValue ) {
  console.log(someValue);
}
```

- Recall the statement version is hoisted!

Notice there is no "typing" of a function - dynamic typing, not static

- The "type" is of the data, not the function

CALLING A FUNCTION

```
doSomething();
const result = returnsSomething();
```

```
// This is NOT called
// This is assignment
const results = returnsSomething;
```

RETURNING A VALUE

```
function min(a, b) {
  if (a < b) {
    return a;
  }
  return b;
}
```

Once you `return`, the rest of the function does not run

a simple `return;` or no return statement at all returns `undefined`.

FAT ARROW FUNCTIONS

So-called "fat arrow" (`=>`) functions are another way to declare a function (only as an expression).

```
const someFunc = (arg) => {
  console.log(arg);
};
```

There are 4 variations of it, these are detailed in the `fat-arrow-functions` doc in your reading, along with the reasons why you would use this.

WHAT IS A CALLBACK

A callback is a function passed to another function, so that the receiving function gets control over

- **How many times** (incl 0) to call the callback
- **When** to call the callback
- **What to pass** in the call to the callback

Callbacks are a hugely powerful pattern that allows for code to be written with minimal information, which reduces the complexity, which makes changes easier.

EXAMPLE CALLBACK

```
const students = {
  maru: 87,
  'grumpy cat': 65
};
```

```
const checkGrades = function( students, onStruggle ) {
  for( let name of Object.keys(students) ) {
    if( students[name] < 80 ) {
      onStruggle(name, students[name]);
    }
  }
};

const tellTeacher = function( student, grade ) {
  console.log(`${student} is getting a ${grade}`);
};

checkGrades(students, tellTeacher);
```

WHY IS THAT COOL

Notice how `checkGrades` doesn't know what you will do with the information!

And yet, `checkGrades` is in control of whether you do it!

Meanwhile, `tellTeacher` doesn't know why it is being called.

In another setup, the exact same `tellTeacher()` could be used to report star students.

In another setup, the exact same `checkGrade()` could be used to email the student a warning.

PROTOTYPES

JS is *NOT* a "classical object oriented" language

But it IS "object oriented"

Objects yes, Classes no.

Classes are a blueprint to describe what an object can do.

In JS, Objects are not nearly so restricted.

INHERITANCE

Objects can have "inheritance" - where an object can use the properties/methods of another object.

If the code tries to access a value on the object, and the object doesn't have it defined for itself, it will check to see if it's **prototype** has it.

Because the prototype is an object, when asked for this value, if it doesn't have it, it will check to see if **it's** prototype has it.

This continues until an object doesn't have a prototype.

PROTOTYPE IS A CONCEPT

Note that we are discussing a concept.

A prototype is an object.

A prototype can be accessed.

A prototype is NOT a property named `prototype`

Just like how an Object has properties, but not a property named `properties`

USING A PROTOTYPE

Inheritance from a prototype is automatic when you try to use the property.

Many built-in functions are accessed this way.

```
const name = "amit";
name.newProperty = "someVal";
console.log(name.newProperty); // not inherited
console.log(name.toUpperCase()); // inherited
console.log(name.length); // inherited
```

ACCESSING

If you need to access the prototype object itself, there are three main ways:

- use `yourObject.__proto__` (DON'T DO THIS) - Legacy code
- use `Object.getPrototypeOf(yourObject)` - returns the prototype object
- Modify the source of the prototype - more on this later, such as with polyfills

PROTOTYPE SUMMARY

- Prototypes are *objects*, not plans
  - This means the prototype can be modified after the fact, like any object
- The prototype of an object is not the `.prototype` property of that object

THIS IS THE MOST CONFUSING TOPIC

`this` is the hardest part of JS

- Similar, but different than other languages
- Makes English hard to use to talk about "this"

ESSENTIAL TRUTH

`this` is a special variable name that will refer to a new value every time you enter a function.

The object the `this` variable refers to is called the "context" - it will usually be a relevant object, but it can get confused.

DO NOT ASSUME `this` will be what you want, you have to make it happen.

IMPLICIT BINDING

By default, `this` is bound to a value "implicitly" when you enter a function.

The value used is the value BEFORE THE DOT in the function call.

```
const obj = {
  val: 'hi',
  func: function() {
    console.log(obj.val);
  },
  implicit: function() {
    console.log(this.val);
  }
};
obj.func();  // 'hi'
obj.implicit(); // 'hi'
```

WHEN IT WORKS

Often, implicit binding works fine. It works through copies and inheritance just fine:

```
const obj = {
  val: 'hi',
  implicit: function() {
    console.log(this.val);
  }
};
const other = { val: 'hello again' };
other.implicit = obj.implicit;
obj.implicit();
other.implicit();
```

WHEN IT DOESN'T WORK

BUT implicit binding has problems.

99% of the time this is when the function with `this` is used as a callback.

```
function usesCallback( callback ) {
  callback();
}

usesCallback( obj.implicit );
```

When the function is called, what is before the dot?

EXPLICIT BINDING

The solution is, when your function will/can be used as a callback, you **explicitly bind** that function to the value of `this` that you want.

Of course, if your potential callback doesn't use `this`, you don't have to care - this is becoming increasingly common.

EXPLICIT BINDING VIA .BIND()

`.bind()` is a method on the prototype of all functions

it returns a new function, which is the function it is on, only explicitly bound.

```
usesCallback( obj.implicit.bind(obj) );
```

inside `usesCallback`, the `callback` will be the explicitly bound function.

EXPLICIT BIND VIA FAT ARROW

Unlike other functions, Fat Arrow functions do not redefine `this`

Technically this is not explicit binding, so much as not re-binding at all

AVOIDING `this` OLD-SCHOOL

Before other options, devs would bypass this problem by copying the value of `this` into another variable.

Usually called `self` or `that`; done when defining an inline function as a callback.

Unlike `this`, this variable would keep the intended value when entering a new function (lexical)

DON'T DO THIS. It's unnecessary and visually noisy.

Other methods of invoking a function, such as `.call` or `.apply` would allow you to supply the desired context object for the call. These are fairly rare.

`this` SUMMARY

- `this` is a variable name that gets redefined when entering a function call
- Implicit binding is to "what is before the dot"
  - `this` can be a problem if the function is used as a callback
- Explicit binding is possible via `.bind()`
- A fat-arrow function can avoid the redefinition
- If you use a non-OOP programming style you can avoid `this` entirely
- Don't use work-arounds like `that` or `self`

WHY INHERITANCE

Don't over do Inheritance

- Modern best practices, even for OOP, favor Composition over Inheritance

Inheritance can be great to provide common functionality.

Inheritance can be a problem if you have many instances but then need to change half of them.

HOW TO CREATE INHERITANCE

JS has 4 ways to create inheritance, which is really 4 ways to create a prototype.

- Constructor Function
- Object.create
- ES6 classes
- Brute Force Prototype Assignment

CONSTRUCTOR FUNCTION

The `new` keyword can be used on a function call. If so, the `prototype` property is assigned as the prototype of the new object.

Such functions are MixedCase, not camelCase.

```
const Cat = function(name) {
  this.name = name;
};
Cat.prototype.beNice = function() {
  console.log(`${this.name} says 'No'`);
};

const maru = new Cat('Maru');
maru.beNice();
```

Notice `maru.beNice` IS inherited, but `maru.name` is NOT inherited

OBJECT.CREATE

`Object.create()` gives you a new object, with the new object's prototype set to passed object (default is `Object`). No initialization code runs (no constructor).

```
const cat = {
  beNice: function() {
    console.log(`${this.name} says 'No'`);
  }
};
const maru = Object.create(cat);
maru.name = 'Maru';
maru.beNice();
```

ES6 CLASSES

- Hotly debated
- More comfortable for those used to other languages
- Can mislead, as they are still objects, not blueprints

```
class Cat {
  constructor(name) {
    this.name = name;
  }
  beNice() {
    console.log(`${this.name} says 'No'`);
  }
}

const maru = new Cat('Maru');
maru.beNice();
}
```

BRUTE FORCE

You can set the prototype directly

```
const cat = {
  beNice: function() {
    console.log(`${this.name} says 'No'`);
  }
};
const maru = { name: 'Maru' };
Object.setPrototypeOf(maru, cat);
maru.beNice();
```

- DO NOT DO THIS - Prefer `Object.create()` or one of the other methods

SOME JS LANGUAGE CONSTRUCTS

The *ternary operator* is a short `if`/`else`

- Use for generating a value only, not to avoid typing "if() {} else {}"
- If in doubt, don't use it - you want clarity, not confusion

```
function min(a,b) {
  return a < b ? a : b;
}
```

FOR OF

It is very, VERY common for people to loop over a counter to get every element in an array

THIS IS USUALLY A WASTE

Unless you care about the index itself (the number), just iterate over the items.

```
const array = [4,8,15,16,23,42];
for(let number of array) {
  console.log(number);
}
```

CLOSURES

Because JS uses lexical scoping and because functions are objects, you can return functions that have access to variables that are otherwise out of scope.

```
const makeCounter = function() {
  let count = 1;
  return function() {
    console.log(count++);
  };
};
const one = makeCounter();
const two = makeCounter();
one(); // 1
one(); // 2
two(); // 1
one(); // 3
```

INSTANCE OF

Compared to statically typed languages, JS does NOT make a lot of use of `typeof` to check types, and even LESS use of `instanceof`

That's because the types can be misleading, and instances are not indicative of what an object can do.

DECONSTRUCTION

You can pull individual values out of an object:

```
const obj = { foo: 'a', bar: 'b', baz: 'c' };
const { foo } = obj;
console.log(foo);

const { bar, baz } = obj;
console.log(bar, baz);
```

FUNCTION ARGUMENTS

You can use deconstruction to create "named" function arguments:

```
function praiseStudent({ name, class, grade }) {
  console.log(`Great job in ${class}, ${name}!  Your grade was ${grade}`);
}

praiseStudent({ class: 'info6250', name: 'Alex', grade: 'A' });
```

- Parameter order doesn't matter
- Particularly nice for boolean values

I recommend you do this for every function with 3+ arguments