About Javascript (JS)

- No relation to Java
    - Was a marketing effort gone bad
    - Vaguely C-style syntax is only similarity
        - True of LOTS of languages
- No reference version
    - Battled out by different vendors
    - Technically most aren't JavaScript(TM)
- ECMA is the source of standards
    - JS is also ES
    - Versions numbered or by year
    - ES6 or ES2015

This is not a JavaScript course

...but we will use a lot of it

- Server
    - our webservers written in it
    - outside course: you may use other languages
- Client
    - "vanilla" JS in browser
    - React framework in browser
    - some form of JS only option for browser!

Going more in-depth this semester

- CSS/HTML was fast and shallow

New to JavaScript? No problem!

No expectation of any experience with Javascript

- Or even much experience with coding

Those with experience in other languages

- Expect to learn different ways!
- Don't be tricked by false cognates!

Ask Questions!

Very important to get any confusion clarified

- We build a lot on this foundation
- You can get further behind if you struggle silently

Hello World!

```
console.log('hello world');
```

We'll look at the parts involved soon

Running the Hello World

```
console.log('hello world');
```

- Run in the Browser console
- Run at the command line with NodeJS
    - A version of the Chrome JS engine!

Node JS and the command line

Running Node enters a "REPL":

```
node
```

Each line is Read, Evaluated, and Printed

Then it asks for a new line (Loop = REPL)

Ctrl-C to exit

Browser console is also a REPL!

Running for real

A REPL is great to spot-check code or syntax

- Not how we run most code

Create a `.js` file and run it with node:

```
// hello.js
console.log('hello world');
```

```
# at command line
node hello.js
```

Experiment!

Lots of learning through experimentation

- You can copy and change versions of files
- Non-coders wish they had this power
- (insert evil villain laugh)

```
console.log('hello world');
console.log('is this a new line?');
```

```
console.log('do spaces     collapse like HTML?');
```

```
console.log('what about
spanning lines?');
```

Variables

```
let message = 'Hello world';
console.log(message);
```

- `message` is a **variable**
- `'Hello world'` is a **value**
    - a "string" of characters
- `let` is the **declaration**
    - telling JS we want `message` to be a variable
        - and not a typo
        - allows JS do tweak performance

Strict Mode

What if we omit `let`?

```
message = 'Hello world';
console.log(message);
```

JS allows a lot of "sloppy" code

- You don't want that
- "Strict mode" disables some sloppy options

```
'use strict';
messsage = 'Hello world';
console.log(message);
```

ALWAYS use strict mode (start of file or function)

- I often skip for space

What's with the semicolons?

A **statement** is a command to JS engine

- statements are either a block in curly braces (`{}`)
    - (more on blocks shortly)
    - or end in a semicolon (`;`)
- So semicolons separate instructions

```
let message = 'hello world'; console.log(message);
```

vs

```
let message = 'hello world' console.log(message)
```

Are semicolons required?

Can't I just use new lines like Python?

- Yes and No
    - Currently in debate in the community
- JS will "guess" at EOL (end of line)
    - ~99.9% correct (not 100%)

Python (and devs) KNOW statement ends at EOL

- JS (and devs) do NOT know this

```
let daysInAYear = 365;
let secondsInAYear = 365
* 24
* 60
* 60;

console
.log(secondsInAYear);  // We do this a lot in JS
```

FOR THIS COURSE: semicolons are required!

Variables and Values

variables hold **values**

- You can change the value held

```
let name = 'Amit';
let otherName = 'Bao';
console.log(name, otherName);

otherName = name;
name = 'Charles';
console.log(name, otherName);
```

Copying sets a variable to the value

- Does not link the variable to another variable
- `otherName` above didn't become `Charles`

Some more syntax snuck in there!

Most variables are named in **camelCase**

- No spaces
- First word lowercase
- Later words have first letter capitalized

assignment (the `=`)

- Doesn't REQUIRE spaces
    - `let name='Bao';` "works"
- This course DOES **require the spaces!**
    - Code is for humans to read/skim/change

Naming things

Naming things is:

- Really important
    - Programming is communication
- Really hard!
    - Seniors spend a LOT of time changing names

Most common mistake from a new dev is poor naming

- "I'll fix it later"
    - You benefit from fixing it immediately!
    - You suffer from poor names immediately!

Variable names

Variable names should

- Convey what the value means
- Make sense in context
- NOT be so long they are gibberish
    - Looking at you, Java!
- NOT be too brief
    - Don't use abbreviations that aren't obvious
    - Don't use abbreviations that aren't needed
    - Ex: `phone`, not `ph` or `phn`

Declaring Variables with const

`let` is not the only way to declare variables

- `const` declares a variable
    - Unlike `let`, you CANNOT reassign a `const`
    - Distinct from "constant" (different concepts)
    - We do this with ~90% of our variables
    - Using `const` passively provides information
        - But only if you use it where you can
        - Programming is communication

```
const message = 'Hello World';
console.log(message);
message = 'Suckers!'; // Throws error
```

Declaring variables with `var`

You can also declare variables with `var`

- But you shouldn't
- Only used for old JS engines (IE 10 and earlier)
- May see it on docs because not everyone cares

`var` is older and has more "cruft"

- "hoists" variables
    - See "Hoisting" on MDN
- Declares to function scope, not block scope
    - More when we reach "Scopes" today
- Pollutes global scope when not in another scope

What are Constants

- We said `const` are not all **constants**

Constants are variables that do not normally change

- Never during run of program
    - Rarely change even outside of program run
- `PI` is a constant, Company name is a constant
- Example: Conditional messages shown to user
- A variable to avoid search/replace or typos
- Use `CONSTANT_CASE` (all caps, underscores)
- `const COMPANY_NAME = "Jorts Inc";` (for humans)

Const vs Constants

- **const**
  - Variable that is never reassigned
- **constant**
  - Variable holding a value that rarely changes
- Const *values* are allowed to **mutate**
  - More when we get to mutation
- Consts are often short-lived values
  - More when we get to functions (next!)
- Most Constants are also declared as `const`

Function calls

A **function** is a collection of statements

- that can be **passed** values
- and can **return** a value

Functions are called with `()`

- any values passed are passed inside the `()`
- this is what we do with `console.log()`

Function demo

```
function sayHello() {
  console.log('hello world');
  console.log('this was fun');
}
```

But nothing happened!

- We didn't **call** the function

```
function sayHello() {
  console.log('hello world');
  console.log('this was fun');
}
sayHello();
sayHello();
```

- Notice the function definition is a **block**

Function Names

Functions are usually named with **verbs**

- Often with **verbNoun** or similar
    - Ex: `setColor`, `getAge`, `calculateWeight`
- Says what the function does
    - and to what, or why, or when
- JS Functions are `camelCase` style
    - Same as JS variables
    - Squish words together
    - Capitalized except for first

Passing values to a function

```
function greet( message, target ) {
  console.log(message + " " + target);
}

greet('hello', 'world');

const greeting = 'heyas';
greet(greeting, 'class');
```

- function definition lists **parameters**
    - during call they are considered **arguments**
- function gets the value (not the variable)
    - declares its own variables
    - no let/const/var required
- `+` used to append strings
    - more soon

Function has its own variables

```
const message = 'hello';
const target = 'world';

function greet(message, audience) {
  console.log(message + " " + audience);
  message = "heyas";
  console.log(message);
}

greet(message, target);
console.log(message);
```

- The outer `message` did NOT change value
- `audience` got the value of `target`

Scope in Javascript

Scope refers to where a variable and value are visible

- A **block** (incl function) is a scope (except for `var`)
- JS has **lexical scoping**
- If variable not defined in this scope,
    - Recursively check the enclosing scopes

```
const cat1 = 'Nyancat';
const cat2 = 'Jean';

function butterTheCat() {
  const cat1 = 'Jorts'; // const but not constant
  console.log(cat1); // Jorts, this scope
  console.log(cat2); // Jean, closest outer scope definition
}

butterTheCat();
```

Scope Best Practices

We sometimes use outer scope variables in functions

- Usually "shared" values within a context
    - When our functions only used in that context

More often we explicitly pass values to functions

- Explicit shows where they came from
- Makes it safer to change values
- Makes function easier to reuse

Values in top-most scope are **global**

- Used for built-in libraries (Ex: `Math.random()`)

Functions are values!

JS functions are "first-class citizens"

- Meaning they are a value like any other value
- Can be stored in a variable
    - Already are, just hidden in syntax!
- Can be passed to functions
- Can be returned by functions

Most of that we won't do YET, but we will

- For now understand a function is a value
- when used without `()`

Functions are values stored in variables

```
function someVar() {
    console.log( "Hi" );
}

console.log( someVar );

someVar = "Hi";
console.log( someVar );
```

Function as a value

```
function sayHello(target) {
  console.log( 'hello ' + target );
}

const greet = sayHello; // Copying function value
greet('world');
```

- The `//` is a comment until EOL
  - Ignored by computer, only for humans
  - More on using comments wisely later
- `greet` and `sayHello` are variables
  - Holding a function value
  - `greet()` (or `sayHello()`) **calls** the function

Functions can return a value

- using `return` keyword

```
function grantTitle( name ) {
  return name + ' The Great';
}

const name = 'Jorts';

console.log( grantTitle(name) ); // Jorts The Great
```

return stops function

As soon as the `return` happens, function stops running

```
function grantTitle( name ) {
  return name + 'The Great';
  console.log('we cannot all be great'); // never happens
}

const name = 'Jorts';

console.log( grantTitle(name) ); // Jorts The Great
```

Types

- A function value is a value
- A string value is a value

What else can a variable hold?

- strings
- functions
- numbers
- booleans
- arrays
- objects
- null
- undefined

Numbers

JS treats all numbers as one type

- Integer OR floating-point (decimal numbers)
- Implemented internally as floating-point

```
let score = 90;
const grade = 'A-';

score = score + 1;
console.log(score); // 91
score += 4;
console.log(score); // 95
score += 0.2;
console.log(score); // 95.2
```

Caution about numbers

Computers store numbers in binary

- binary struggles with certain numbers

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

- Not a JS problem
    - a computer problem
- If precision is important (banking, rocketry)
    - use special libraries
    - In every language

JS Variables are Dynamically Typed

- Some languages "statically typed" (Java, C)
- Some languages "dynamically typed" (JS, Python)

dynamically-typed: the type comes from the VALUE

- Not the variable

```
let score = 90; // Number
console.log(score);
score = 'A-'; // string
console.log(score);
```

JS Variables are Weakly typed

- Some languages "strongly typed" (Java, Python)
- Some languages "weakly typed" (JS, C)

Weak typing means values will be **coerced** to match

```
const score = 90; // Number
console.log(score + 1);  // 91, a Number
console.log(score + '-'); // '90-', a String
```

Most of the time coercion is bad/risky!

- You should explicitly convert types
- One exception, coming up soon

When is a number not a number?

`NaN` is a special value

- A Number that represents "not a number"

```
console.log( 'Jorts' / 9 ); // NaN
```

- Doesn't throw an error (yet)
- results in `NaN`

Boolean values

"Boolean" is another kind of value

- `true` or `false`
- Not `'true'` or `'false'`, not strings
- Note for Python users: NOT `True` or `False`

`!` is the "negation" operator, it gives the inverse:

```
console.log(true);
console.log(!true);
```

Conditionals: What "If"

A **conditional** is an essential part of coding

- Make something happen IF some condition is true

```
const check = true;

if (check) {
  console.log('Check passed!');
} else {
  console.log('Check was not valid');
}
```

Or else what?

`else` is optional

```
const check = true;
if (check) {
  console.log('Check passed');
}
```

Can also chain with `else if`

```
const check = true;
const extra = false;

if (check) {
  console.log('Check passed!');
} else if (extra) {
  console.log('Check was not valid but extra was');
} else {
  console.log('Nothing was true');
}
```

`if` is a block

The `if` and `else` are each accept a **statement**

- can have a block or just one command
- but you should always have a block

```
// JS allows but you shouldn't do:
const check = true;
if (check)
  console.log('wow!');
```

Why not? See this:

```
// Not working as intended
const check = false;
if (check)
  console.log('wow!');
  console.log('check was true!'); // Not in the `if`
```

The condition is a boolean expression

An **expression** returns a value

- Conditional evaluates an expression as a boolean
- Many operators return a boolean value
  - `===` (strict comparison)
  - `>`, `<`, `<=`, `>=`, `!==`

```
const name = 'Jorts';
if ( name === 'Jorts' ) {
  console.log('Hi Jorts!');
} else {
  console.log('Who are you?');
}
```

Loose comparison

- `=` is **assignment**
- `==` is ????
- `===` is **strict comparison**

`==` is **loose comparison**

- Allows **type coercion** when comparing

```
if ( '1' === 1 ) {
  console.log('this does not happen');
} else if (1 == '1') {
  console.log('this does happen');
}
```

Coercion is bad...most of the time

Normally we want strict comparison (`===`)

- Predictable, Expected, Reliable

One big exception:

- When the coercion is to a boolean value
- This is known as **truthy/falsy**

Truthy/Falsy

These values are **falsy** (coerce to false when boolean)

- `false` (no surprise)
- `''` (empty string, no characters)
- `0` (the number 0)
- `NaN` (a number that is not a number)
- `null` and `undefined` (values that aren't values)
    - more to come on these

Anything else is **truthy** (coerce to true when boolean)

Why is Truthy/Falsy coercion okay?

It isn't always

- Loose comparison is almost always bad
    - Surprise coercion
- Truthy/Falsy without comparison can be okay
- Hold onto this thought
    - We need just a few more concepts

When is a value not a value?

Most programming languages have a value to represent "no actual value"

- `''` represents an "empty string"
    - A string with no characters
    - But it is still a string!
- What is a non-value?

JS is twice as good, because it has TWO such values

- This is sarcasm
- `null` means "set to no value"
- `undefined` means "never had a value"

Variables are undefined if not assigned a value

```
let message;
console.log(message);
```

Functions with no explicit return will return `undefined`

```
function charlieOnTheMta() {
  console.log('this function does not return a value');
}

let returned = charlieOnTheMta();
console.log(returned);
```

In fact, that's what `console.log()` does:

```
console.log(console.log());
```

Other places to see undefined

- Functions may return undefined as "no match"
- Will show up in objects/arrays too
    - Foreshadowing!

Functions always return a value

If your function doesn't return an explicit value

- Or if it just has `return;`
- It will return `undefined`
- That's what `console.log()` does
    - Prints to screen
    - But always *returns* `undefined`

Function Parameters

Only one version of a function exists at a time

- Regardless of how many values you pass to it
- Fewer than expected and other values default
    - Normal default is `undefined`
- More than expected and extras ignored

```
function demo( one, two, three ) {
  console.log( one );
  console.log( two );
  console.log( three );
}

demo( 'one' ); // one undefined undefined
demo( 'one', 'two', 'three', 'four' ); // one two three
```

Function defaults

You can provide different defaults for parameters

```
function demo( one, two='Jorts', three ) {
  console.log( one );
  console.log( two );
  console.log( three );
}

demo( 'one' ); // one Jorts undefined
demo( 'one', 'two', 'three', 'four' ); // one two three
```

Using `undefined` and `null`

- **Never explicitly assign** `undefined`
  - Only check for it
  - Use `null` if you need to assign
    - Also rare
    - Often we use a value already `undefined`
- **Nullish coalescing operator** when falsy wrong
  - More on that later

Intro to logical && ("and") and || ("or")

Combine boolean conditions

- With `&&` ("**and**")
    - Requires both to be `true`
- With `||` ("**or**")
    - Requires at least one to be true
    - NOT exclusively
- Use `()` to group for clarity
    - **order of operations** applies

```
if( catLocation === 'closet' && cat === 'Jorts' ) {
    freeTheCat();
}
```

Conditions are important Communication!

Write your code to be easily skimmed and understood

```
if (catLocation === 'closet') {  // Worst
    if (cat === 'Jorts') {
```

```
if( catLocation === 'closet' && cat === 'Jorts' ) {  // Meh
```

```
if( (catLocation === 'closet') && (cat === 'Jorts') ) { // OK
```

```
const catInCloset = catLocation === 'closet'; // Not constant
const catIsJorts = cat === 'Jorts';

if( catInCloset && catIsJorts ) {  // Great
```

Why do we like truthy/falsy?

Compare

```
if ( name === '' || name === undefined || name === null ) {
  console.log('A name is required');
}
```

```
if ( !name ) {
  console.log('A name is required');
}
```

Programming is communication

- Which is easier to follow?

Why is this nonsensical?

```
if ( name == '' || name == undefined || name == null ) {
  console.log('name is no good');
}
```

Hint: Try this

```
const name = null;

if ( name == '' || name == undefined ) {
  console.log('That was unexpected');
}
```

- Use **strict comparison** (`===`)
- Or use **truthy/falsy** with no explicit comparison
- Basically never use `==`

More on Strings

We've seen some simple strings, but there is more

- Strings used often in web coding!
    - HTML
    - CSS classes
    - Typed input
- Three forms of quoting
    - Single quote `'a string';`
    - Double quote `"a string";`
    - Backtick:

```
`a template literal`
```

Quoting a string

Which quotes you use mostly don't matter

- Some teams prefer double quote (`"`)
- Some teams prefer single quote (`'`)
- Some teams use backticks exclusively (`` ` ``)
- Some teams use them for different purposes
    - HTML works well with `'` to have `"` inside
        - `'<a href="/cats">upgrade</a>'`
    - English works well with `"` to have `'` inside
        - `"I'd hate this with single quotes"`
- Backticks work for both English and HTML
    - But not often the default quote (yet?)

Template Literals

What is a **template literal**?

- A string
- That can span lines
- And can **interpolate** variables inside `${}`
    - Nicer than using `+`

```
const name = "Jorts";
const food = "snacks";

const greeting = `Hello ${name}, got ${food}?`;
const longForm = "Hello " + name + ", got " + food + "?";

console.log(greeting);
console.log(longForm)
console.log(` This is
  a multiline string`);
```

Defaulting values

We saw functions **parameters** can have **defaults**

```
function greet( message='Hello', target='World' ) {
  console.log(`${message} ${target}`);
}
greet(); // Hello World
greet('Heya'); // Heya World
greet('Heya', 'Class'); // Heya Class
```

We can have "default" values in other situations

Short Circuiting

We saw `||` is "or" and `&&` is "and"

- These operators **short-circuit**
- When logical result known from left argument
    - They don't evaluate the right side
- They don't actually return a boolean value
    - They return the left or right hand value!
    - We used them in a boolean context
        - Evaluated as truthy/falsy values

Demonstrating Short-Circuiting

```
function report() {
  console.log('Did Stuff');
  return 5; // a distinct truthy value for demo
}

const one = 0 && report(); // 0 is falsy
console.log(one); // 0 - short-circuited

const two = 7 && report(); // non-0 numbers are truthy
console.log(two); // Did Stuff 5 - checked both

const three = 8 || report();
console.log(three); // 8 - short circuited

const four = 0 || report();
console.log(four); // Did Stuff 5 - checked both
```

Defaulting with short circuiting

```
let name = getName(); // some random function somewhere
name = name || 'Jorts'; // Defaulting name
console.log(name);
```

- If `name` was a truthy value, it is unchanged
- If `name` was a falsy value, it is now `Jorts`
- This "defaults" `name` to `Jorts`

```
let name = getName();
name ||= 'Jorts'; // same thing, newer operator (mid-2020)
console.log(name);
```

Nullish Coalescing

Falsy values include `0`, `NaN`, `false`, and `''`

- Those are still values though
- How to "default" only values that are non-values?
    - `null` and `undefined`

Answer: The **nullish-coalescing operator** (`??`)

- Works like `||` EXCEPT
    - Returns right side if left is `null` or `undefined`
- `??=` similarly like `||=`

Demonstrating Nullish Coalescing

```
const one = '';
const two = null;

console.log( one ?? 'This does not print' );
console.log( two ?? 'This does print' );

let three = false;
let four;

three ??= 'Three';
four ??= 'Four';

console.log( three ); // false
console.log( four ); // Four
```

Conditional Operator

- `condition ? A : B`
- Also called "ternary operator"
    - Only JS operator with 3 parts

```
const test = true;

const result = test ? 'Jorts' : 'Jean';
console.log(result);
```

- Basically "If `condition`, then `A`, else `B`"
    - Expression, not a statement
    - Unlike `if`, this evalutates to a value (`A` or `B`)
- Most languages have this, JS uses a bit more often
    - Assign a value, not as flow-control

Defaulting with Conditional Operator

```
let name = getName(); // some random function somewhere
name = name ? name : 'Jorts';
console.log(name);
```

Not better than the `||=` version

- But it has additional flexibility

Conditional Operator - It Depends!

```
let coat;
if ( name === 'Jorts' ) {
  coat = 'buttered';
} else {
  coat = 'unbuttered';
}
```

- Easy to read line by line
- Harder to skim
- Draws attention to "uninteresting" parts of code

```
const coat = name === 'Jorts' ? 'buttered' : 'unbuttered';
console.log(coat);
```

- Harder to read line-by-line
- Keeps focus on important code

Arrays (Lists)

An **array** is an ordered collection of values

- The **ordered** part is important
- Accessed using numerical **index** (position)
- Index starts at 0, not 1
    - Like Western-style ages

Declaring an Array

Arrays are created using `[ ]` (square brackets)

```
const names = [ 'Jorts', 'Jean' ];
```

- Array variable names are usually **plural**
    - Programming is Communication
- Values in array do not need to be same type
    - But usually are
- Array declaration can span lines (like most JS)
- Trailing commas are okay (and even common)

```
const names = [
  'Jorts',
  'Jean',
];
```

Accessing Array Element

- An array **element** can be gotten by **index** in `[]`
    - no relation to HTML element

```
const names = [
  'Jorts',
  'Jean',
];
console.log( names[1] ); // Jean
```

You can treat an array element like any other value

- Notice we just passed one to a function
    - `console.log()`

```
const name = names[1];
const cats = [ names[1] ]; // [ 'Jorts' ]
```

Assigning to an element

An indexed element is both read AND write

```
const names = [
  'Jorts',
  'Jean',
];
names[1] = 'Nyan';

console.log( names[1] ); // Nyan
```

Wait, `names` is a `const` variable!

Why does this work?

`const` is not a constant

```
const names = [ 'Jorts' ];
names[0] = 'Jean';
```

- Recall `const` means **variable won't be reassigned**
    - `names` still refers to SAME array
- An array is a container for ordered elements
    - Container contents are not the container
- Changing collection contents known as **mutation**

Mutation in passing

Consider this:

```
let name = 'Jorts';
let color = 'orange';
let names = [ name ];

function change( name, color, names ) {
  name = 'Jean';
  color = 'tabby';
  names[0] = 'Nyan';
  names = ['Jean'];
}

change( name, color, names );
console.log( name, color, names );
```

Why did only the array mutate?

JS functions are passed VALUES

- But arrays are **references** to collections
    - Changing an element mutates the collection
    - Replacing local scope variable values doesn't alter outer scope variables

This will come up a lot

- Make sure it makes sense before moving forward
    - Or you will struggle with weird bugs later

Nested Arrays

Array elements can be any JS value

- Including other arrays

```
const toys = [
  [ 'mousie', 'bed' ],
  [ 'mousie', 'laser pointer' ],
];

console.log( toys[0][1] ); // bed
```

Objects

JS Objects are a huge deal in JS

- But very different from many other languages!

Objects are NOT

- Instances of a class (usually)
- Defined/limited by a class

Objects ARE

- Collections of elements (like array)
- Indexed by a string "key"
- Often used as "dictionaries" or "hashmaps"

Declaring an Object

Objects are declared with **curly braces** (`{ }`)

- Values of keys (**properties**) are any JS value
  - Including arrays, objects, and *functions*
  - Keys separated from values by a colon (`:`)
  - Keys must be unique (they will overwrite)
- Different key/value pairs are comma-separated
  - Trailing commas allowed and common

```
const cat = {
  name: 'Jorts',
  age: 3,
  toys: ['mousie', 'laser pointer'],
};
```

More about Object Declaration

- `{ }` like a block, but it is **not a block**
- Keys are strings
    - Do not require quotes unless invalid variables
        - Special chars, start with numbers, etc
- Keys should usually be **camelCase**
    - Just like variable names
    - Except when key is data (such as NEU ID)

```
const cat = {
  name: 'Jorts',
  age: 3,
  toys: ['mousie', 'laser pointer'],
};
```

Object Shorthand

Objects often built from other variables

```
const name = 'Jorts';
const age = 3;
const toys = ['mousie', 'laser pointer'];

const cat = {
  name: name,
  age: age,
  items: toys,
};
```

Shorthand notation highlights when name is different

```
const cat = {  // Same result as above
  name,
  age,
  items: toys,
};
```

Accessing an Object Element

A value in an object is called a **property**

- If the value is a function, then also a **method**

Values are accessed by using **dot notation**

```
const cat = {
  name: 'Jorts',
  age: 3,
  toys: [ 'mousie', 'laser pointer'],
};

console.log( cat.name );
```

Bracket Notation

- When a property name isn't a valid variable name
    - Special characters, etc
- Or is coming from a variable

You can use **bracket notation** to get the value

```
const cat = {
  name: 'Jorts',
  age: 3,
  'fav toys': ['mousie', 'laser pointer'],
};

console.log( cat['fav toys'] ); // cat.fav toys won't work

const property = 'name';
console.log( cat[property] ); // cat.property won't work
```

Computed Property Name

Assign to a property named by variable using brackets

```
const cat = {};
const someVariable = 'name';
cat[someVariable] = 'Jorts';

console.log(cat.name); // Jorts
```

You can also create the object in similar way:

```
const someVariable = 'name';
const cat = {
  [someVariable]: 'Jorts',  // Rare, but it does happen
};

console.log(cat.name); // Jorts
```

Object properties default to `undefined`

```
const cat = {
  name: 'Jorts',
  age: 3,
};

console.log( cat.color ); // undefined
```

Does NOT throw an error

`undefined` is NOT an object though:

```
const cat = { age: 3 };
console.log( cat.name.first );
```

```
Uncaught TypeError: Cannot read properties of undefined
(reading 'first')
```

- Problem is `cat.name` not `first`!
- Error Message is accurate, read closely!

Errors in JS

Confused JS **throws an error**

- Unlike HTML/CSS, JS quits and stops
    - Unless error is **caught**
- Uncaught errors only report to console
- Most users don't use console
    - They see no messages, no errors
    - They just sit and wait

```
const cat = { age: 3 };
console.log( cat.name.first );
console.log('will not print');
```

Catching Errors

A `try...catch` can **catch errors**

- Allows code to keep running
- Often only to better report error
- `catch` block skipped if no error

```
try {
  const cat = { age: 3 };
  console.log( cat.name.first );
} catch (error) {
  console.log('only prints if error');
  console.log(`saw an error (${error})`);
  console.log(`but continuing anyway`);
}
console.log('will always print');
```

Each of `try` and `catch` are separate blocks

- Variables declared in blocks are in that scope

```
try {
  const cat = { age: 3 };
  console.log( cat.name.first );
} catch (error) {
  console.log(`saw an error (${error})`);
  console.log(`bad data is in:`, cat);
}
console.log('This line does not print');
```

- Tells us `cat is not defined`
- Crashed because error happened in `catch`, not `try`

This change works, but do you understand why?

```
const cat = { age: 3 };

try {
  console.log( cat.name.first );
} catch (error) {
  console.log(`saw an error (${error})`);
  console.log(`bad data is in:`, cat);
}
console.log('This line prints');
```

The `finally` block

Optional `finally` block runs after both `try` and `catch`

- Regardless of an error happening
- Even if `catch` throws an error

```
try {
  const cat = { age: 3 };
  console.log( cat.name.first );
} catch (error) {
  console.log(`saw an error (${error})`);
  console.log(`bad data is in:`, cat);
} finally {
  console.log('this line always prints');
}
```

Handling Errors

When you catch an error, what do you do with it?

- It Depends
- Two kinds of errors
    - Recoverable (non-essential)
    - Unrecoverable (it broke)
- Keep going or tell user nicely
    - Future dev (you?) needs to know what broke!
- Frontend JS:
    - **Do not use** `alert()`, `prompt()`, `confirm()`
    - Ugly and "block" interactions

Not Handling Errors

JS often written with limited/no error handling

- This isn't part of JS, but where/how JS is used
- Frontend JS involves no dev-readable log
- End users are often very disconnected from devs
- Web dev is often fast-paced over reliable
    - "Move fast and break stuff" - Facebook
- Focus often on giving a good UX to failure

You should not under-use OR over-use error handling

- It depends
- Lessons will often involve minimal examples

Methods

An object property that is a function is a **method**

- In JS a function is a value like any other value

```
const cat = {
  name: 'Jorts',
  play: function() { // function name is always optional!
    console.log('Checks to see if gravity still works');
  },
};

cat.play();
```

A JS function is a value that is callable

- Can be assigned to variables
- When declared, function name creates a variable
    - Function name is always optional!
        - You need some way to call the function
        - Can call a method via object key
    - A function declared as a value
        - only creates variable in own scope

```
const sayHello = function() {
  console.log('hello world');
};

sayHello();
```

Objects vs Arrays

Many new JS devs overuse Arrays and underuse Objects

- Arrays only make sense if:
    - The specific order matters
    - AND you access elements by order most of the time

Quite often you have items not accessed by order

- Student records
- Recipe names
- Inventory selections

What to use as an object key?

Use some identifying value

- common to repeat value in object

```
const students = {
  '1234': {
    neuId: '1234',
    name: 'Amit',
    grade: 89,
  },
  '2345': {
    neuId: '2345',
    name: 'Bao',
    grade: 94,
  }
};
```

Changing an element

Like arrays, an element is both read AND write

```
const cat = {
  name: 'Jorts',
  age: 3,
};

cat.age = 4;
console.log( cat.age );
```

Notice we mutated the `const` cat!

- An object is a reference to a collection of values
    - Like arrays

Mutation in passing (object version)

This works just like with arrays:

```
let name = 'Jorts';
let color = 'orange';
let names = [ name ];
let cat = { name, color };
// above same as { name: name, color: color }

function change( name, color, names, cat ) {
  name = 'Jean';
  color = 'tabby';
  names[0] = 'Nyan';
  names = ['Jean'];
  cat.name = 'Maru';
  cat = { name, color };
}

change( name, color, names, cat );
console.log( name, color, names, cat );
```

Adding a property/method

Define a new property by assigning

- even if the property never existed before

```
const cat = {
  name: 'Jorts',
};

cat.age = 3;

console.log( cat ); // { name: 'Jorts', age: 3 }
```

Notice: Mutating a `const` collection

Deleting a property

Setting a property to `null` (or `undefined`) does NOT delete it

- it is still an existing, enumerable property

Instead, use `delete` keyword

```
const cat = {
  name: 'Jean',
  age: 5,
};

delete cat.age;

console.log( cat ); // { name: 'Jean' }
```

Notice: Mutating a `const` collection

Weird Secret of JS objects

- Every value in JS is a **primitive** or an **object**
    - Primitives: string, number, boolean, undefined, null
        - Primitives can't mutate (**immutable**)
        - Can only be replaced by new values
    - BUT all primitives have object versions
        - "autoboxing"
        - allows you to call methods on that type

```
const num = 1/3; // 0.3333333
console.log( num.toFixed(2) ); // 0.33
console.log( (1/3).toFixed(2) ); // Parens so dot is clear
```

All primitives have autoboxed properties/methods

```
const name = 'Jorts';
console.log( name.toUpperCase() ); // JORTS
console.log( name.length ); // 5
```

No primitive method mutates the value

- has to return a new value
- example: `.toUpperCase()` above

Non-primitives are all objects

- arrays and functions(!) are technically objects
    - "an object" usually means "plain" objects
    - But they are actually objects
        - have properties
        - and methods
        - and can mutated

Chaining

The **dot operator** lets you access properties on the value to the left

```
const cat = {
  name: 'Jorts',
  toy: {
    name: 'mousie',
    texture: 'fuzzy',
    hasCatnip: true,
  },
};

console.log( cat.name );
console.log( cat.toy.texture );
```

chaining is when the value to the left came from another evaluation

Chaining can break

```
const cat = {
  name: 'Jorts',
  toy: {
    name: 'mousie',
    texture: 'fuzzy',
    hasCatnip: true,
  },
};

console.log( cat.age ); // undefined, but no error
console.log( cat.color.markings ); // Throws error
```

Optional Chaining can help

The **optional chaining** operator can help

- If the value chained from is **nullish**

```
const cat = {
  name: 'Jorts',
  toy: {
    name: 'mousie',
    texture: 'fuzzy',
    hasCatnip: true,
  },
};

console.log( cat?.age ); // undefined
console.log( cat.color?.markings ); // also undefined
console.log( cat?.color?.markings ); // also undefined
console.log( cat.meow?.() ); // works with functions too
```

Don't overuse optional chaining!

Optional chaining is great to avoid a lot of

```
if( cat && cat.color && cat.color.markings ) {
  //...
}
```

- But don't use it to silence errors
    - The error is still there!
    - Now just harder find the source
- Use optional chaining only when a nullish value
    - an expected and valid option

Common Array methods

- Arrays methods can mutate array
- Adding an element:
    - `.unshift( item );` adds `item` to start of array
    - `.push( item );` adds `item` to end of array
- Removing an element:
    - `.shift();` returns item removed from start
    - `.pop();` returns items removed from end
    - `splice(...);` returns items removed
        - `...` here means "there's stuff in there"
- See MDN for more on these and other methods

Loops

Loops are a thing in coding

- Same instruction
- Done multiple times
- Often done a number of times based on data
- Or done for each piece of data in a collection

`for` loops

`for` loops are a very common loop style

- but we have a few ways to use them

C-Style for loops

C-style `for` loops - not that common!

```
for ( let index = 0; index < 10; index++ ) { // ++ is += 1
  console.log(`running with index ${index}`);
}
```

- `for` is a statement, **should be a block**
    - just like `if`
- LOOKS like a function, but isn't
    - 3 statements inside `()`, between `;`
        - initializer, runs at start
        - condition, checks before an iteration
        - accumulator, runs after an iteration

Using C-style for loop

New JS devs may mimic other languages

- use c-style for loops to iterate over array

```
const names = ['Jorts', 'Jean', 'Nyan'];

for( let index = 0; index < names.length; index++ ) {
  console.log( names[index] );
}
```

But this isn't the best way

- We rarely care about `index` here
- We just use it to get the element

`for...of` loop

The `for...of` loop iterates over the VALUES

- Not the index

```
const names = ['Jorts', 'Jean', 'Nyan'];

for ( let name of names ) {
  console.log( name );
}
```

- Less cognitive overhead
- Easier to follow
- Easier to focus

`for...in` loop

The name is confusingly similar to `for...of`

- Iterates over object property keys
- Not array values

```
const cat = {
  name: 'Jorts',
  age: 3,
  color: 'orange',
};

for ( let key in cat ){
  console.log( key );
  console.log( cat[key] );
}
```

About for...in

Object properties used to be in unpredictable order

- Order long ago became stable

Can get an array of properties using `Object.keys()`

```
const cat = {
  name: 'Jorts',
  age: 3,
  color: 'orange',
};

console.log( Object.keys(cat) );// [ name, age, color ]
console.log( Object.values(cat) );// [ 'Jorts', 3, 'orange' ]
```

Result: `for...in` not used often

Common interview question!

```
for( var i = 0; i < 10; i++ ) {
  console.log(i);
}
console.log(i);
```

What is the output?

- Also seen with a time-delayed output

```
for( var i = 0; i < 10; i++ ) {
  setTimeout( function() { // Called ~1 second later
    console.log(i);
  }, 1000);
}
```

Answer

```
for( var i = 0; i < 10; i++ ) {
  console.log(i);
}
console.log(i);
```

What is the output?

- prints 0 through 10
- `var` is **function-scoped** not **block-scoped**
    - only one `i` variable
- `var` is **hoisted**
    - Exists before and after for loop
    - Not just inside

Delayed answer

```
for( var i = 0; i < 10; i++ ) {
  setTimeout( function() { // Called ~1 second later
    console.log(i);
  }, 1000);
}
```

- Prints `10` 10 times
- Only one `i` variable
- `i` has `10` after loop
    - That is what gets output

Block-scoped loop

```
for( let i = 0; i < 10; i++ ) {
  console.log(i);
}
console.log(i);
```

- prints 0-9
- Errors at end (`i is not defined`)
- `const` would fail (can't reassign (`i++;`))
    - `const` would work with `for..of` loop

Callbacks

A pattern used a lot in JS is the **callback**

A callback is a function passed to another function

- Not passing the result of calling
- Passing the uncalled function itself

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

`checkGrades` doesn't "know" much

- knows WHEN to call callback
- doesn't know what callback does
- **decoupled**

`tellTeacher` doesn't know why it is being called

- knows what to do
- doesn't know when it is called
- doesn't know where info came from

Callbacks are very flexible

Callbacks allow for logic to be used for different things

- `tellTeacher()` can be to report star students
- `checkGrade()` can be used to email the student

Arrays have even more methods

Many of them use callbacks

`forEach` array method

Callback will be called for each element

```
const names = [ 'Jorts', 'Jean', 'Nyan' ];o

function sayName( name ) {
  console.log(name);
}

names.forEach( sayName );
```

Callback functions are often defined "inline"

If a function is only used as a callback

- often defined where it is passed:

```
names.forEach( function( name ) {
  console.log(name);
});
```

This is cumbersome, so a shorter format is often used

- "fat arrow functions"

Fat arrow functions

`->` is commonly called "arrow" in coding

- But isn't used (yet) in JS
- This format uses `=>`
    - thus "fat arrow"
    - coding has lots of fun names
    - Shuttle (`<=>`), Elvis (`?:`) , etc

JS uses the fat arrow to succinctly define a function

- NEVER defines a variable
- HAS to be used as a value

Fat Arrow syntax

- Left of the `=>` are the parameters in parens (`()`)
    - Parens option if EXACTLY one parameter
- Right of the `=>` is the statement
    - Returns expression value if not a block

```
const isEight = (num) => {
    return num === 8;
};
const alsoChecksForEight = num => {
    return num === 8;
};
const sum = (one, two) => {
  return one + two;
};
const alsoSum = (one, two) => one + two;
```

```
names.forEach( name => console.log(name) );
```

Dealing with Arrow Functions

(Fat) Arrow function syntax a lot to process at first

- You can use function keyword functions
- Arrow functions are very common in web dev
    - It is good to get familiar with the syntax

Array .sort() method

`.sort()` on an array sorts it

- IN PLACE (mutation)
- default: "asciibetically" ascending
    - NOT numeric

You can pass a callback that compares two params

- to sort in the order you like

```
const nums = [ 100, 5, 2 ];
nums.sort(); // default sorting
console.log( nums ); // [ 100, 2, 5 ]
nums.sort( (a, b) => a - b ); // neg to sort "earlier"
console.log( nums ); // [ 2, 5, 100 ];
```

Array .join() method

Returns a string of the elements joined together

- uses a passed string in between elements

```
const names = [ 'Jorts', 'Jean', 'Nyan' ];
console.log( names.join() ); // 'JortsJeanNyan'
console.log( names.join('-') ); // 'Jorts-Jean-Nyan'
```

We will make use of this method, don't ignore it

Array .filter() method

Returns NEW array with elements that pass test

- test is a callback passed to filter
- callback is called with each element
    - if callback returns truthy value, element passes

```
const names = [ 'Jorts', 'Jean', 'Nyan' ];
const shortNames = names.filter( name => name.length < 5 );
console.log( names ); // [ 'Jorts', 'Jean', 'Nyan' ]
console.log( shortNames ); // [ 'Jean', 'Nyan' ]
```

Array .map() method

- Returns a NEW array
- Results of calling callback on each source element

Example of translating data into HTML strings:

```
const names = [ 'Jorts', 'Jean', 'Nyan' ];
const listHtml = names
  .map( name => `<li>${name}</li>` ) // returns new array
  .join(''); // joins new array into one string

console.log(`
  <ul>
    ${listHtml}
  </ul>
`);
```

Spread operator

The **spread operator** works on arrays and attributes

- We will use this a lot later

The operator is `...` before the variable name

- like the "stuff here" in `for...in` and `for...of`

The spread operator "spreads out" the contents

- used to fill up another container

Spreading arrays

```
const names = [ 'Jorts', 'Jean' ];

console.log( [ 'Nyan', names ] ); // array nested in array
console.log( [ 'Nyan', ...names ] ); // only one array
```

Fills array-like value with values from inside an array

- Rather than nesting array

Spreading objects

```
const cat = {
  name: 'Jorts',
  age: 3,
};

console.log({ color: 'orange', ...cat });
console.log({ ...cat, age: 5 }); // overwrites age from cat
```

Provides the key:value pairs from object

- if duplicate property key last value wins

Destructuring

- **destructure** - removing structure
- Declares new variables
    - Assigns matching values from array/objects
- Leaves original structures unchanged

Not something you immediately need

- Used often in web dev

Destructuring Arrays

Creates new variables with values from array

- You don't need all values
- Destructure with `[ ]` around variable names

```
const names = ['Jorts', 'Jean', 'Nyan'];

const [ first, second ] = names;

console.log(second); // Jean
```

Why would we do such a thing?

Why Destructure Arrays

Usually we don't

Sometimes we have an array of different parts

- Ex: function that returns multiple values
    - Functions can only return 1 value
    - 1 array is 1 value (containing many)

```
const [ studentInfo, semesterInfo ] = lookupStudent('Amit');

// Then do stuff with studentInfo and/or semesterInfo
```

- You will see this in React
- Mentioning now to avoid a surprise later

Destructuring Objects

Creates new variables named after keys

- with matching values
- destructure with `{ }` around variable names

```
const cat = {
  name: 'Jorts',
  color: 'orange',
  age: 3,
};

const { name, age } = cat;

console.log( name, age ); // there is no variable "color"
```

Why Destructure Objects

More common than destructuring arrays

- Work with values without object
- Reverse of Object shorthand

Can fake "named function parameters"

- Pass a function an object of parameters
- Function destructures object into parameters
- Order of params (key/value pairs) doesn't matter!
- Each param is labeled with the name (the key)
- Good for functions with many parameters
- Good for boolean parameters

Named Function Parameters Example

```
function reportAge ( name, useUpperCase, age ) {
  const useName = useUpperCase ? name.toUpperCase() : name;
  console.log( `${useName} is ${age} years old` );
}

const age = 3;

reportAge('Jorts', true, age );
```

```
function giveAge ({ name, useUpperCase, age }) {
  const useName = useUpperCase ? name.toUpperCase() : name;
  console.log( `${useName} is ${age} years old` );
}

const age = 3;

giveAge({ name: 'Jorts', age, useUpperCase: true });
```

Fancier function param defaults

Normal function arguments can have defaults

- But omitting "middle" parameters is ugly

```
function greet( message='Hello', target='World' ) {
  console.log(`${message} ${target}`);
}

greet(); // Hello World
greet('Heya'); // Heya World
greet('Heya', 'Class'); // Heya Class
greet(undefined, 'Class'); // Hello Class (ugly)
```

Function defaults with param object

With named function params this gets nicer

- can omit any params
- usage is more clear

```
function greet({ message='Hello', target='World' }) {
  console.log(`${message} ${target}`);
}
greet({});
greet({ message: 'Heya' });
greet({ message: 'Heya', target: 'Class' });
greet({ target: 'Class' });
```

But `greet()` fails with an error!

- `TypeError: Cannot read properties of undefined (reading 'message')`

Read the Error messages!

The stacktrace may not be helpful after a while

- The first few lines are essential
- The message says what is wrong!

It is trying to destructure the passed object

- We passed no object
    - It is trying to destructure `undefined`

Don't guess randomly with errors! (not at first)

- You'll make a mess and not learn
- Understanding errors is a skill to learn

Defaulting the destructured object

We can default the object parameter too

```
function greet({ message='Hello', target='World' }={}) {
  console.log(`${message} ${target}`);
}
greet();
greet({target: 'Class'});
```

No more error, defaults even work

Using Named Function Parameters

I recommend using named function parameters

- Whenever you have 3+ params
- and/or function name has unclear param order
- and/or if you have boolean parameters
- and/or you have assorted defaults

Immediately Invoked Function Expression (IIFE)

IIFEs are used in browser JS code

- All code in an anonymous function
- This function is immediately run (invoked)

Because everything is in a function

- Not in global scope
- No accidental global variables

```
(function() {
  // ... code here
})();
```

Some tools remove the need for this

Remember for now!

- Always 'use strict'
- Always use an IIFE in browser JS
    - Not NodeJS server code
- Do these even when my examples don't
- Later tools will do this for you

Javascript is probably not your first language

So you have some habits that may not apply to JS

- A common issue
- I'll highlight some common stumbling blocks

Javascript CAN be Object-Oriented

- But usually isn't
- JS Objects are rarely instances of classes
    - when they are the rules are different
    - inheritance works differently
- Native structures and syntax used a lot
    - Almost never instantiate typed Objects
        - e.g. `new Array()`, `new Object()`, etc
        - just use the native literal syntax `[ ]`
    - Some concepts `Map()`, `Set()` are barely used
        - usually plain objects work fine!

Javascript uses duck-typing

If it walks like a duck and quacks like a duck...

- `instanceof` and `typeof` very rare
- usually only used to identify passed params
    - and there are traps there!
        - Example: `typeof []` is 'object'
        - instead `Array.isArray([])`

Typescript has different type-safety options

- but it too is unlike, say, Java
- not run-time enforced

`null` is fairly rare

Even though we never explicitly assign `undefined`

- We end up rarely assign `null`
- If you aren't initializing to a value, leave it as `undefined`
- Explicitly assigning `null` is visually noisy
    - only do it if you're unsetting a value

`const` is common

~80-90% of variables end up being `const`

- Reassignment is just not that common
- Preferring `const` means using `let` stands out!
    - `let` isn't BAD
        - More informative when `const` preferred
- We also use a lot of objects
    - `const` doesn't complain if contents change

`this` works differently

Keyword `this` *almost* like other languages

- Can trip up new devs because it FEELS the same
    - Until it isn't
- Some code rarely uses `this`
    - Fat arrow functions reduce complications
    - Can write entire web apps and never use `this`
- Not a direct focus of this course
    - We use JS, this isn't a JS course
    - JS devs should learn all about `this`

Regular Expressions

- Regular Expressions (RegEx/RegExp)
    - Allow for powerful text parsing
    - Easy, once you know it
    - Easy to mess up
- Many Regex jokes: indecipherable and arcane
- Included in all major languages for a reason
    - Have native syntax support in JS!
    - This matters!
    - WebDev involves a lot of text

Try...Catch and Handling Errors

- JS does not have compile-time checking
    - no compile time!
- Errors are found at run-time
    - Often not much to do to "fix" it
- Errors only "handled" when useful response
    - But don't silence unexpected errors!
- Bad input handled before an error is thrown
    - No error trapping involved