JS can be run...client side or server side?	variables type can be changed dynamically, functions don't have to define return type, method overloading can be done without defining multiple functions with diff signatures, arrays can store objects of different types
callback function	FALSY: undefined, null, 0, false, '', NaN TRUTHY: anything that is not falsy
hoisting	conditional operator that provides shorthand for if else statements, let type = points > 100 ? "gold" : "silver";
Primitive/value data types	(==): basic defn is that only checks for value, (===): basic defn is that checks for value and data type
REFERENCE data types	break: keyword used to jump or break out of a loop, continue: keyword used to end current iteration, and then resumes execution of next iteration

difference between primitive and reference data types?	// for objects: variable defined in loop is key of object
	// for arrays: variable defined in loop is index
	// returns all members (instance + prototype),

Js implements DYNAMIC TYPING, what are some characteristics of dynamic typing?	Both. Js can be run client-side with JS Engine in browser, or can be run server-side in NODE, (JS Engine embedded in c++ program)
If Boolean function is evaluated with non Boolean values, the JS Engine will determine value based on truthy/falsy values:	function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action
ternary operator:	the process of moving function declarations to top of file at runtime by JS Engine
loose equality operator (==) vs strict equality operator (===) ?	String, Number, Boolean, undefined, null, Symbol(a unique identifier generated by Symbol() function)
break and continue keyword	Objects, arrays, functions (arrays and functions are objects too)
FOR-IN LOOP: mainly used to iterate through an objects' properties,	PRIMITIVES ARE COPIED BY THEIR VALUE, OBJECTS OR REFERENCE TYPES ARE COPIED BY THEIR REFERENCE

FOR-OF LOOP: ideal way to iterate through arrays, or for arrays you can use foreach()	object.property // dot notation, //bracket notation allows us to create dynamic references to property: const propertyName = 'location'; object[propertyName]
BLOCK SCOPED VARIABLES: are only accessible inside current block and blocks within this block	circle.location = {x:1}; delete circle.location;
FUNCTION SCOPED VARIABLES: are accessible to blocks within this block, AND anywhere within function, attaches variables to Window object if outside function	if ('location' in circle)//checks object + prototype if (circle.hasOwnProperty('location'))//checks object
this keyword: references the object that is executing the current function	const circle = { radius: 1, draw: function() {} };
apply(), bind(), call() methods	//returns object it creates, property values are supplied as arguments, // if key and value have same name, you can write 'name: name' as 'name' // when defining function inside object we can drop : and function keyword
Arrow functions: don't rebind this keyword (inherit this value from containing function)	function createCircle(radius) { return { radius, draw() {}} } const circle = createCircle(3);

Object: 2 ways to access properties:	// loop variable is value of each element in array // can only be used on objects with iterator
Since properties in JS are dynamic, you can add/update/delete properties	let, const
To see if an object has a given property:	var
object literal syntax	if current function is method of object, or constructor function, THIS references current object the method belongs to, if current function is regular standalone function (outside class, or method inside a function inside class, or prototype method) THIS references global object (in browser window, or in node global), important to remember
//************************* */ // FACTORY FUNCTIONS, //************************* */	can change value of this for functions
ex of factory function	useful because standalone functions or callback functions reference window/global object with this keyword

//************************ */
// CONSTRUCTOR FUNCTIONS,
//************************ */

prototypes

ex of constructor function

an object that is the parent of another object

TEMPLATE LITERALS: indicated by backtick character, helps write cleaner code with strings

when an object performs a method/property call, the JS Engine first looks at that object to see if it has implementation, and if not found then goes up the Prototype chain until found or until it gets to the single root object in memory.
Every object references the single root object in memory as its prototype.

ex of function declaration

The proto property is deprecated, but it still viewable for other purposes. Just need to know that EVERY OBJECT CREATED BY SAME CONSTRUCTOR WILL HAVE THE SAME PROTOTYPE.

ex of function expression

const x = {};
const y = {};
Object.getPrototypeOf(x) === Object.getPrototypeOf(y); // returns true

2 ways to declare function: function declaration or function expression. What are differences?

Constructors have a "prototype" property that returns reference to baseObject prototype

How do we achieve inheritance in js?

//we use the new operator and this keyword with constructor functions
//called by new operator and new operator provides empty object
//this keyword binds the members to the empty object

prototype

```
function Circle(radius) {
this.radius = radius;
this.draw = function() {}
}
const anotherCircle = new Circle(3);
```

Prototypical Inheritance

${}// allows for substitution of any expression, variable, function within string,
// whitespaces in backticks are included in string

__proto__

```
function greet() {
console.log("hello world");
}
```

// All objects created with the same constructor will have the same prototype. Their prototype is called baseObject, ie a circle derived from Circle constructor will have baseCircle prototype
// A single instance of this prototype will be stored in the memory.

```
let run = function() {}; //
anonymous function expression
let run = function walk() {}; //
named function expression
```

objInstance.__proto__ === ObjectConstructor.prototype

// expression needs ; at end of {} and declaration does not, function declarations are hoisted and function expressions are not

Mixins: We use Mixins to achieve COMPOSITION in JS

extends // use extends keyword, resets prototype and constructor fn() under the hood,
// -use super constructor to call parent constructor and super keyword to access prototype members/methods,if parent has constructor and you wish to put constructor in child then you HAVE to call super constructor first

REST OPERATOR, ...

related grouping of code to improve maintainability, resusability, and abstraction,
high level rule is things that are highly related should be grouped together

DEFAULT PARAMETERS:

// all you really need to know is ES6 Modules for browser and CommonJS which is used with NodeJS

the class declaration with class keyword are used to mimic class syntax most people are familiar with, under the hood it is just syntactical sugar for constructor function and prototypical inheritance

```
// Named exports: use export keyword to export one or more objects,
export class Square {}
import {ObjectName[s]} from 'module(path)';
// Default exports: use export default keyword to export main object that is exported from a module,
export default class Square {}
import ObjectName from 'module(path)';
// to import both from a file we use:
import ObjectName, {OtherObjectName} from  ‘ path ’ ;
// to understand import line we need to use Webpack or within html file set type attribute to 'module' in script tag
```

properties and methods for classes

converts our JS code into es5 versions so every browser can understand, babeljs.io is a website that shows you the conversion of es6+ code to es5

```
//************************* */
// PRIVATE PROPERTIES AND METHODS
//************************* */
// can be implemented via:
```

combindes all JS files and other files into a bundle, minify code, uglify

//*************************** */ // ES6 INHERITANCE: //*************************** */	// favor composition over inheritance! composition: flexible technique where you combine a few objects to create new object const canSwim = {swim: function(){}}; const canEat = {eat: function(){}}; const canWalk = {walk: function(){}};
modules:	//used before parameter of function so the varying number of arguments given will be put in array and can be used inside function, cleaner than using arguments property that is in every function and is actual array
// Module formats	//can be used by assigning parameter a default value instead of using logical expressions (||) inside function body, must be last parameter in function

//*********************** */
// ES6 Modules (Used in Browser)
//*********************** */
	class Circle{
	constructor(param1){this.param1 = param1;}
	methodName() {logic here}
	}

ES6 TOOLING(for client SIDE) transpiler(BABEL): translator+ compiler	// all properties and methods are added to baseCircle prototype (in these notes Circle was constructor function), to add to instance you you need to define members in constructor, OR USE ARROW FUNCTION defined in class (preferred to reference current object with this keyword and puts method on the object instead of prototype)
ES6 TOOLING(for BROWSER SIDE) bundler(WEBPACK):	symbols(sort of), implement with es6 computed properties, weakmaps, implement with getters/setters,

//**************************** */
// STRICT MODE:
//**************************** */

	// any reference to global object is undefined
	// throws error if you try to call/modify Window object
	// raises errors on some things that are otherwise silent errors
	// does more but beyond scope of this intro class