JAVASCRIPT

THE INTRO

JavaScript is a programming language developed in the late '90s.

It is a high-level, interpreted language typed dynamically.

Interpreted languages translate the code during execution.

Dynamic typing allows for values reassignment.

JavaScript supports multiple styles of programming.

Imperative programming deals with statements that change the program's state (typically through conditions/loops).

Object-oriented and procedural programming are considered mostly imperative.

Declarative programming deals with building the logic without describing the flow (generally without heavy condition/loop presence).

Functional programming is considered a declarative type.

Two common approaches to creating scripts are:

- in script tags within HTML
- in files with a `.js` extension

Scripts can also be generated through tooling systems and dynamically inserted in the HTML.

VARIABLES

Variables are named containers for values.

There are three ways to declare a variable.

```
var name;
let favoriteFood;
const lovesJavaScript;
```

A variable declared with the `var` keyword is a standard variable.

Anything declared with `var` is function scoped.

```
var name = "Adam";
name = "Syntax the Dog";
var name = "Your Name Here";
```

Notice that the use of `var` allows for:

- reassigning
- redeclaring

When declaring a variable with `let`, the following rules apply to the variable:

- the variable is scope blocked
- redeclaring the variable within the scope is not possible

```
let food = "Tacos";
food = "Nachos";
let food = "Pizza"; // SyntaxError: Identifier 'favoriteFood' has already been declared
```

When declaring a variable with `const`, the following rules apply to the variable:

- all of the rules from `let` carry over to `const`
- reassigning the variable within the scope is not possible

```
const age = 32;
age = 33; // TypeError: Assignment to constant variable
const age = 34; // SyntaxError: Identifier 'age' has already been declared
```

Block scoping refers to variables/functions being accessible within the *code block* they are declared within.

```
{
  var name = "Adam";
  console.log(name); // "Adam"
}
console.log(name); // "Adam"
```

```
{
  let age = 32;
  console.log(age); // 32
}
console.log(age); // ReferenceError: age is not defined
```

```
{
  const favoriteFood = "nachos";
  console.log(favoriteFood); // "nachos"
}
console.log(favoriteFood); // ReferenceError: favoriteFood is not defined
```

RULES, PRACTICES, AND BEHAVIORS

Follow these four simple rules when declaring variables.

1. Must begin with a letter, _, or $
2. Can contain letters, numbers, _, and $
3. Names are case-sensitive
4. Do not use JavaScript keywords for variable or function names

```
var currentYear = 2017; // number
let $firstName = "Grant"; // string
let _lastName = 'Chirpus'; // string - single or double quotes for strings
const detroit_is_cool = true; // boolean
```

`var` is vital to understand; however, in modern times, all variables should be declared using either `let` or `const`.

Using `const` as much as possible is the preferred practice.

`let` is appropriate to use if the conditions within the program demand a value reassignment.

When a script file gets executed, JavaScript collects all variable and function declarations and *hoists* them to the top of whatever scope held the declaration.

It is important to remember that the value does not get brought with the declaration.

Only the declaration gets hoisted.

Consider the following code using `var`:

```
var name = "Adam";
console.log(name); // "Adam"
```

With a quick alteration, the code now acts much differently.

```
console.log(name); // undefined
var name = "Adam";
```

Consider the following code using `const`:

```
const name = "Adam";
console.log(name); // "Adam"
```

With a quick alteration, the code now acts much differently.

```
console.log(name); // ReferenceError: Cannot access 'name' before initialization
const name = "Adam";
```

DATA TYPES

A primitive data type is the lowest level of data a language can use.

Currently, there are seven primitive data types in JavaScript:

- string
- number
- BigInt
- boolean
- null
- undefined
- symbol

The other data type in JavaScript is `Object`.

Strings are used to represent text.

Each character in a string references an element.

The first element within a string is considered to be at index 0.

All strings have a length.

```
const framework = "Angular";
console.log(framework.length); // 7
```

When writing more complex strings, template strings provide an excellent solution.

Template strings allow for the string to contain embedded expressions and allow for multiline strings.

Rather than using quotations, the backtick ` character is what begins and end the template string.

Using template literals produces much cleaner code.

```
const name = "Adam";
console.log(`Hello, my name is ${name}.`); // "Hello, my name is Adam."
console.log(`1 + 1 = ${1 + 1}`); // "1 + 1 = 2";
```

The addition operator can be used to combine strings.

Using the addition operator is not the preferred way, but it is still beneficial to know.

```
const firstName = "Yasmine";
const lastName = "Abdulhamid";
const fullName = firstName + " " + lastName;
console.log(fullName); // "Yasmine Abdulhamid"

let name = "Y";
name += "asmine";
console.log(name); // "Yasmine"
```

Numbers represent numbers.

There are no differences between integers and floats in JavaScript.

All numbers are just numbers.

```
const firstNumber = 10;
const secondNumber = 20;
const sum = firstNumber + secondNumber;
console.log(sum); // 30
```

BigInt is used to represent huge numbers.

BigInt is loosely the same as Number, but for numbers that are outside of the safe limit.

```
const largeNumber = 30n * 39n;
console.log(largeNumber); // 1170n
```

Booleans are the "yes" or "no" values in JavaScript.

Instead, they are true or false.

They can only ever be true or false.

```
const isJavaScriptCool = true;
const iAmTired = false;
console.log(isJavaScriptCool); // true;
console.log(iAmTired); // false;
```

Null is the intentional absence of a value.

Null is commonly used as a variable's value if the variable gets initialized later on.

```
let usersName = null;
usersName = "awesomeDude392";
```

Undefined is the value every variable is assigned when the script executes.

The value of undefined points to the data not being defined.

```
console.log(food); // undefined
var food = "nachos";
console.log(food); // "nachos"
```

Symbol is a unique and immutable identifier.

Use Symbol as a key to an object.

```
const name = Symbol("myName"); // Symbol(myName)
const age = Symbol("myAge"); // Symbol(myAge)
const myInfo = {
  favoriteLanguage: "JavaScript"
};
myInfo[name] = "Adam";
myInfo[age] = 32;

console.log(myInfo);

console.log(myInfo[name]); // "Adam"
```

There isn't a big need to use Symbol early on in development.

Article on Symbol

Objects are collections of key/value pairs (keys are another word for properties).

A key must be of type String or Symbol.

The values of keys can be of any data type.

Objects are references to locations in memory - not a defined value.

```
const myInformation = {
  name: "Adam",
  age: 32,
  favoriteFoods: ["nachos", "pizza", "tacos"],
  dog: {
    name: "Syntax",
    breed: "Mix"
  }
};
```

Objects are extremely robust.

Both Function and Array are of type Object.

All primitive types get wrapped in an object wrapper.

The object wrapper is how primitive types gain access to their methods.

TYPE CHECKING

As programs develop, there may arise the occasional need to check the type of various data sources.

For instance, if there is a question whether or not a value is of type String or Number, a simple type check could be introduced.

There are two "main" ways of type Checking.

The first is with the `typeof` operator.

The second involves more code but is a much more reliable approach.

Using the typeof operator works nicely if the data is of type Number, String, or Boolean.

```
const language = "JavaScript";
const numberOfStudents = 15;
const isHungry = false;
const assistants = ["Carly", "Ben", "Tim"];
const person = { name: "Yasmine" };
const hello = () => "Hello";

console.log(typeof language); // "string"
console.log(typeof numberOfStudents); // "number"
console.log(typeof isHungry); // "boolean"
console.log(typeof assistants); // "object"
console.log(typeof person); // "object"
console.log(typeof hello); // "function"
```

Using the more involved process gives a much more accurate representation of the data's type.

```
const language = "JavaScript";
const numberOfStudents = 15;
const isHungry = false;
const assistants = ["Carly", "Ben", "Tim"];
const person = { name: "Yasmine" };
const hello = () => "Hello";

function checkType(data) { return Object.prototype.toString.call(data); }

console.log(checkType(language)); // "[object String]"
console.log(checkType(numberOfStudents)); // "[object Number]"
console.log(checkType(isHungry)); // "[object Boolean]"
console.log(checkType(assistants)); // "[object Array]"
console.log(checkType(person)); // "[object Object]"
console.log(checkType(hello)); // "[object Function]"
```

CONTROL STRUCTURES: CONDITIONALS

Logic can be inserted to control the flow of programming.

Controlling the flow of a program means programs are structured to perform specific actions under certain conditions.

There are a few basic control structures that are important to know:

- Loops
- Conditions

Any value in JavaScript evaluates to truthy or falsy.

Truthy	Falsy
`true`	`false`
`non-zero numbers`	`null`
`Infinity`	`undefined`
`-Infinity`	`0` (this is the number zero)
`non-empty strings`	`NaN` (Not a Number)
`mathematical expressions`	`empty strings`

If a value is either truthy or falsy, said value could be used to drive logic.

A simple statement to check whether a value is truthy of falsy:

```
const language = "JavaScript";
console.log(!!language); // true
const username = "";
console.log(!!username); // false
```

If / else if / else statements determine which parts of the code should run under certain conditions.

```
if (true) {
  // do something
} else if (true) {
  // do something else
} else {
  // do another something else
}
```

Conditions can be quite simple or quite complex.

```
let name = "Adam";
let tired = false;
if (name === "Adam" && !tired) {
  console.log("Adam is not tired");
} else {
  console.log("Something is not right...");
}
```

It is easy to complicate things with the overuse of `else if`.

Switch statements are an excellent alternative.

```
const choice = prompt("Select a size");
switch (choice) {
  case "small":
    console.log("You have ordered a small shirt");
    break;
  case "medium":
    console.log("You have ordered a medium shirt");
    break;
  case "large":
    console.log("You have ordered a large shirt");
    break;
  default:
    console.log("We have no other sizes");
    break;
}
```

A ternary operator is implemented for very basic decision making.

```
const age = 39;
const canVote = age >= 18 ? true : false;
// if age is greater than or equal to 18, canVote is true
// if age is less than 18, canVote is false
```

CONTROL STRUCTURES: LOOPS

Loops allow code blocks to repeat over and over again.

Loop	Use
`for`	Executes a block of code until the condition evaluates to false.
`do...while`	Executes a block of code before checking the condition.
`while`	Checks a condition before executing the block of code.
`for...in`	Iterates over keys of an object (including arrays).
`for...of`	Iterates over values of an object (including arrays).

Control structures have statements that perform specific actions when used in a loop.

Keyword	Description
`break`	Used to break out of a control structure.
`continue`	Used to skip to the next iteration.

The `for` loop has three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement or a set of statements executed in the loop.

```
for (initialization; condition; final-expression) {
  // repeating things
}

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

The `while` loop iterates as long as a given condition evaluates to true.

The condition evaluates before each iteration of the loop.

```
while (condition) {
  // repeating things
}

let a = 0, j = 0;
while (a < 30) {
  a++;
  j += a;
}
```

The `do...while` loop is very similar to a while loop.

The code contained within the code block runs at least once before the condition evaluates.

```
do {
  // repeating things
}
while (condition);

let userPass = null;
do {
  userPass = prompt("What is your password?");
} while (userPass !== "secret");
```

The `for...of` loop iterates through property values.

It is quite simple to set up and acts very similar to the `for...in` loop.

The for...of loop should be your go-to for arrays.

```
let languages = ["Java", "JavaScript", "Ruby", "Python", "C#", "PHP", "HTML", "CSS"];
for (let language of languages) {
  console.log(language);
}
```

The `for...in` loop iterates through property names.

The for...in loop is quite handy for checking properties of an object or extracting those property names.

```
const grandCircus = { location: "Detroit", rooms: 6, colors: ["teal", "orange", "charco
for (let prop in grandCircus) {
  console.log(prop);
}
```

FUNCTIONS

Executing a code block on demand is a widespread task in programming.

Doing such a task would either require the code to be written multiple times or a function to be implemented.

Functions, by default, return a value.

```
function greetClass() {
  console.log("Hello class!!!");
}

greetClass(); // "Hello Class!!!" is logged, however, the function returns undefined
```

The `return` statement causes the function to immediately exit and returns the value to the line of code that called the function.

No code is executed after a return statement so the return statement should always be the last line of code within a function.

With ES6, there are four "types" of functions.

`function declaration`

`function expression`

`arrow function expression`

`anonymous function`

All four require the same fundamental knowledge of functions.

A function declaration is the declaration of a function.

Function declarations do get hoisted.

Declaring a function is relatively similar to declaring variables (keyword, name).

```
function calculateArea(length, width) {
  console.log(length * width);
  return length * width;
}

calculateArea(10, 5); // 50 is logged, however, the function returns the number 50
```

Similar to the function declaration; however, the big difference is that the function is the value of a variable.

The function is also not named, and it is anonymous.

```
const calculateArea = function(length, width) {
  return length * width;
}

calculateArea(10, 5); // the function returns the number 50
```

Arrow functions are quite similar to function expressions but require slightly less code, and arrow functions handle referencing `this` differently.

`this` is a word that comes back up later in the course.

```
// function declaration
function feedPeople(person1, person2) {
  return `${person1} and ${person2} are now fed.`;
}

// arrow function
const feedPeople = (person1, person2) => `${person1} and ${person2} are now fed.`;
```

When a function needs any information to complete its task, `parameters` get defined within the declaration.

Parameters act like variables; however, they are not indeed variables.

Parameters are just placeholders.

When a function is called or executed, data may get passed into the function as `arguments`.

The `arguments` used become assigned to the parameters for use within the function.

Let us take a few minutes to discuss what is happening.

`number` is the parameter.

`4` is the argument.

```
function isEven(number) {
  if (number % 2 === 0) {
    return true;
  } else {
    return false;
  }
}

isEven(4);
```

Parameters may also have default values.

If the function executes with an undefined argument, the default parameter kicks into play.

```
const feedPeople = (person1, person2 = "Carol") => `${person1} and ${person2} are now f

feedPeople("Steve"); // "Carol" is assigned to person2
```

Functions can return functions.

Returning functions from functions is a common practice.

```
function success() {
  console.log("The number is even.");
}

function isEven(num) {
  if (num % 2 === 0) {
    return success;
  } else {
    return function() {
      console.log("The number is not even.");
    }
  }
}

let result = isEven(4); // value of result is now the success function

result(); // result is now a function and can be executed to see "The number is even."
```

SCOPE

A variable declared within a function is scoped to that function.

Likewise, declaring a variable with the `let` or `const` keyword, the variable is only accessible within that code block.

Declaring a variable or function anywhere out of a code block establishes said variable or function as global.

Global variables and functions can be accessed anywhere but can cause problems with larger, more complex applications.

```
const dayOfWeek = "Monday"; // scoped globally
function processUser() {
  const newUser = "Adam"; // scoped to processUser
  console.log(newUser); // the string "Adam"
  console.log(dayOfWeek); // the string "Monday";
}

processList();
console.log(newUser); // doesn't exist outside of processUser
```

A significant problem in JavaScript is the global scope and keeping it clean.

As such, several critical techniques and design patterns in JavaScript were developed to address this problem.

One such technique is the Immediately Invoked Function Expression (IIFE).

An IIFE is an anonymous function that executes immediately.

The IIFE creates a *closure* that keeps all the variables inside private and contained.

```
(function() {
  // awesome code
})();
```

```
"use strict";
{
  // the new ES6 block can also be used to keep variables scoped properly!
}
```

CLOSURES

What are closures?

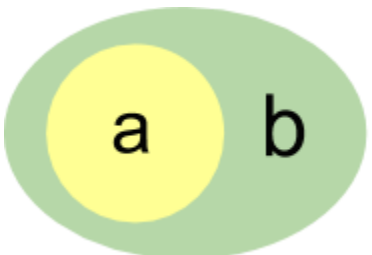
A closure is a function that has references to variables and functions that are coming from an outer scope.

As a nifty side effect, developers can create closures as a form of encapsulation in JavaScript.

```
function yellow() {
  const a = "yellow";
  function green() {
    const b = "green";
    console.log("a: " + a); // > a: yellow
    console.log("b: " + b); // > b: green
  }
  green();
}
yellow();
```

The set of variables visible to yellow:

The set of variables visible to green:



Why are closures so powerful?

Programmers can create numerous functions that all run the same base code, but have different values associated with them.

```
function sayHello(name) {
  const greeting = function(message) {
    return `${message} ${name}?`;
  }
  return greeting;
}

let hiAdam = sayHello("Adam");
console.log(hiAdam("How are you"));

let hiYasmine = sayHello("Yasmine");
console.log(hiYasmine("How are you"));

let hiDavid = sayHello("David");
console.log(hiDavid("How are you"));
```

Variables inside a closure are only accessible *within* that closure.

In JavaScript, closure is very close to the same concept as a function's scope.

A closure "remembers" the context in which it executes in.

Even if that function is removed from that context and used elsewhere, it still has access to all the variables of its original context.

OBJECTS

Objects are a data structure that allows us to store unorganized collections of data.

An object consists of two parts: state and behavior.

That is, data about the object (state) and methods that can act on that data (behavior).

There are a few ways to create an object.

Object literal notation is the first option for creating an object.

The second way is using a constructor.

Objects require curly braces (unlike arrays, which use square brackets).

Objects have key/value pairs (properties/values), separated by a colon.

If there are multiple properties, a comma separates them.

```
const car = {
  make: "Ford",
  model: "Ranger",
  year: 2019,
  color: "blue"
};
```

Object keys can be accessed or altered using dot or bracket notation.

Deleting properties can be done by using the `delete` keyword.

Adding properties is done by either dot or bracket notation.

```
const car = { make: "Ford", model: "Ranger", year: 2019, color: "blue" };

console.log(car["make"]); // "Ford" is logged
console.log(car.model); // "Ranger" is logged

delete car.color; // car no longer has a color key

car.ecoFriendly = true; // car has an ecoFriendly key that is true
```

Methods are very similar to functions but have a more specific purpose.

Methods belong to objects which allow the objects to perform actions.

Often, methods will be used to do the following:

- Return some information about the object
- Change the state of the object

Adding a method to an object is quite easy.

```
const car = {
  make: "Ford",
  model: "Ranger",
  year: 2019,
  color: "blue",
  turnOn() {
    console.log("The car turns on.");
  }
  turnOff() {
    console.log("The car turns off.");
  }
  paintTheCar(color) {
    this.color = color;
    console.log(`The car is now ${this.color}.`);
  }
};

car.turnOn();
car.paintTheCar("orange");
```

Classes allow for a clean and easy way to create objects.

Behind the scenes, a class is a function.

Think of a class as a blueprint or model.

Objects are instantiated (created) from the blueprint or model.

Class declarations are *not* hoisted.

Imagine writing a program to manage all students for bootcamp.

There would probably be a single class (blueprint), which acts as the structure for each student (object).

Each student (object) may have unique attributes, but they are all derived from the same structure (class).

```
class Student {
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
}

const tSwift = new Student("Taylor Swift", 30);
console.log(tSwift);
```

The object is constructed with the provided state.

Adding behavior to the object requires writing a method to the class.

```
class Student {
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log("Hello, front-end!");
  }
}

const tSwift = new Student("Taylor Swift", 30);
console.log(tSwift);
tSwift.sayHello();
```

Inspecting these objects after creation will not display the methods.

The object's prototype contains the methods.

The *prototype* holds all the common properties and methods (push, pop, splice, and the other pre-defined methods for an array).

For example: when an array is defined, it may have unique elements but also inherits conventional methods and properties from the array's prototype.

The `hasOwnProperty` method can determine whether a property is part of the individual object or part of its common class.

```
console.log(tSwift.hasOwnProperty("name")); // true
console.log(tSwift.hasOwnProperty("sayHello")); // false

let newArray = new Array();
console.log(newArray.length);
console.log(newArray.pop);
console.log(newArray.hasOwnProperty("length")); // true (individual object)
console.log(newArray.hasOwnProperty("pop")); // false (common class)
```

By using extends and super, "subclasses" can be created from the `Student` class.

```
class Student {
  constructor(name, age){
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log("Hello, front-end!");
  }
}

class FrontEndStudent extends Student {
}

const tSwift = new FrontEndStudent("Taylor Swift", 30);
```

The `extends` keyword is used to set the prototype of the new class to a prior class.

`super` is the reference point to the prototype, and without it, the subclass has no reference to `this`.

```
class FrontEndStudent extends Student {
  constructor(name, age) {
    super();
    this.name = name;
    this.age = age;
  }
  sayHello() {
    console.log(`Nice to meet you. My name is ${this.name}.`);
  }
}

const tSwift = new FrontEndStudent("Taylor Swift", 30);
tSwift.sayHello();
```

`this` always refers to a single object.

The object `this` references is determined by when and where it gets used.

In the global context, `this` refers to the global window object.

When using `this` within a method, it refers to that particular object.

```
const javaScriptStudent = {
  name: "Stephanie",
  age: 30,
  sayName() {
    console.log(`My name is ${this.name}`);
  }
};
javaScriptStudent.sayName(); // "My name is Stephanie"
```

ARRAYS

Arrays are similar to objects in that they are collections of data.

An array is a data structure consisting of an ordered collection of elements.

This numerical index starts at 0, not 1.

```
const names = ["Adam", "Anna", "David", "Josh", "Yasmine"];
// Index:        0        1       2        3         4
```

Similar to objects, arrays can be defined through literal syntax or a constructor.

The literal syntax is the preferred way of doing things, so stick with that.

Bracket notation is used to access the value in an index.

```
const listOfPeople = ["Adam", "Amanda", "Ashley", "Ajay", "Jonah"];
const firstPerson = listOfPeople[0]; // "Adam"
const lastPerson = listOfPeople[listOfPeople.length - 1]; // "Jonah"
```

There are a few options to add an element to an array.

The following methods allow for the adding of elements to an array:

Method	What the method does
`Array.unshift()`	Adds the element to the beginning of the array
`Array.push()`	Adds the element to the end of the array
`Array.splice()`	Adds the element to a specific index in the array

There are a few options for removing an element from an array.

The following methods allow for the removing of elements to an array:

Method	What the method does
`Array.shift()`	Removes the element at the beginning of the array
`Array.pop()`	Removes the element at the end of the array
`Array.splice()`	Removes element(s) at a specific index in the array

A method that can both add and remove is Array.splice().

```
const drinks = ["Tea", "Coffee", "Soda", "Water"];
drinks.splice(2, 1, "Pop"); // start at index 2, remove 1 element, add "Pop"
console.log(drinks); // ["Tea", "Coffee", "Pop", "Water"];
```

COMMON METHODS

Commonly used methods

There are many methods provided by JavaScript.

There is hardly a need to memorize the list of methods or trying to understand precisely *how* they all work.

Focus on knowing how to look up methods and use them.

Method List

REST

The purpose of the rest parameter is to bundle up all of the remaining arguments of the function.

The rest paramater returns an array which can use methods such as:

- Array.find()
- Array.map()
- Array.reduce()
- Array.forEach()
- Array.includes()

This function can only ever use two parameters (num1 and num2) even if the function is called with three arguments

```
function addNumbers(num1, num2) {
  let total = 0;
  total = num1 + num2;
  return total;
}
addNumbers(1, 2, 3, 4, 5); // returns 3
```

This function, using the rest parameter and reduce, allows the function to use any amount of arguments.

```
function addNumbers(...numbers) {
  return numbers.reduce((prev, next) => prev + next);
}
addNumbers(1, 2); // returns 3
addNumbers(1, 2, 3, 4, 5); // returns 15
```

The rest parameter must be the last parameter assigned within the function.

The "rest" of the arguments get bundled in an array.

```
function sayHello(person1, person2, ...everyoneElse) {
  console.log(`Hello ${person1}`); // Hello Adam
  console.log(`Hello ${person2}`); // Hello Ivan
  console.log(`Hello ${everyoneElse[2]}`); // Hello Celena
  console.log(`Hello ${everyoneElse[3]}`); // Hello Yasmine
}

sayHello("Adam", "Ivan", "Kim", "Antonella", "Celena", "Yasmine");
```

SPREAD

The spread operator is used to extract elements of an array or the keys of an object.

Think about the following situations:

- An array/object is needing a copy
- A function requires multiple arguments from the same object/array

Encountering each of the above situations is very likely.

Copying an object to manipulate the copied version, not the original, is quite common.

In this situation, it is quite common to construct an object literal from the original object's properties.

```
const obj1 = { name: "Snoopy", age: 2 };
const obj2 = { name: obj1.name, age: obj1.age };
const obj3 = { ...obj1 }; // obj3 contains the same keys/values as obj1

const names1 = ["Adam", "Grace"];
const names2 = [ ...names1, "Cody" ]; // names2 contains the same elements as names1 ar
```

The spread operator can also be used as a single argument that refers to a collection of elements.

```
const things = ["Computer", "TV", "Radio", "Cellphone"];
function printThings(item1, item2, item3, item4) {
  console.log(item1);
  console.log(item2);
  console.log(item3);
  console.log(item4);
}
printThings(...things);
```

DESTRUCTURING

Destructuring extracts elements from arrays or keys from objects and assigns them to variables.

Destructuring is very common when working with frameworks/libraries.

The goal with destructuring is to cut down on repeating code like:

```
const computer = {
  cpu: "Ryzen 7", motherboard: "Crosshair 7",
  gpu: "NVidia 2080 TI", ram: "32GB G.SKILL TridentZ"
};
const gpu = computer.gpu;
const ram = computer.ram;
```

Rather than repeating that style of code, a single one-line statement can be used.

The below code will create two variables: `gpu` and `ram` which have values coming from the keys of `gpu` and `ram` of the computer object.

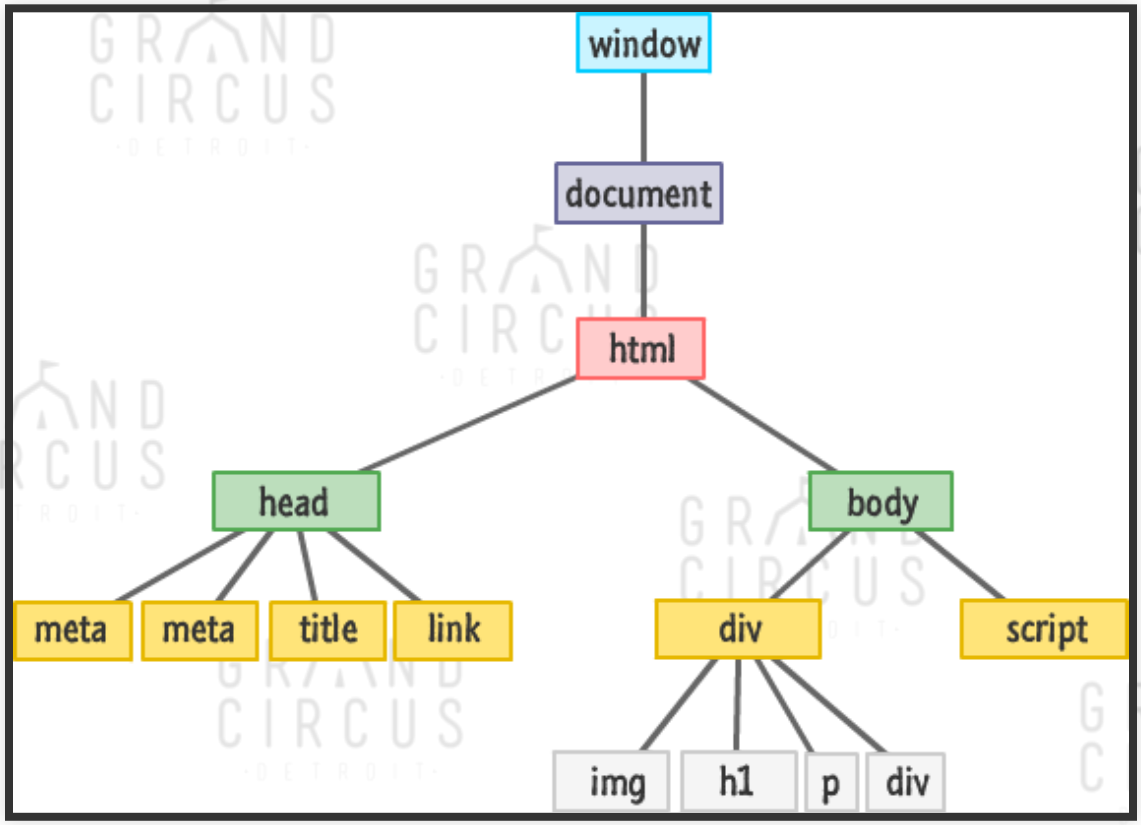
```
const computer = {
  cpu: "Ryzen 7", motherboard: "Crosshair 7",
  gpu: "NVidia 2080 TI", ram: "32GB G.SKILL TridentZ"
};

const { gpu, ram } = computer;
console.log(gpu); // "NVidia 2080 TI"
console.log(ram); // "32GB G.SKILL TridentZ"
```

DOM

The Document Object Model (DOM) is an interface which allows dynamic access to the HTML.

The DOM is a logical tree structure where each element in HTML represents a node (object).



The DOM exposes methods that are used to manipulate the structure of a page.

Each web page loaded in the browser has a unique `document` object.

Open any web page in a browser, open the developer tools, and run this command in the console.

```
document.write("JavaScript is pretty cool.");
```

There are numerous ways to access various nodes within the DOM.

- Node.childNodes - all direct children of this node
- Node.parentNode - the parent of this node
- Node.firstChild - all direct children of this node

```
// body element
const bodyNode = document.body;

// html element
const htmlNode = document.body.parentNode;

// Array of all body's children
const childNodes = document.body.childNodes;
```

There are two methods that can be used to do the selecting which are:

- document.querySelectorAll()
- document.querySelector()

CSS selectors are used as arguments to the methods.

```
document.querySelectorAll(".rows"); // returns NodeList
document.querySelectorAll("#heading-title"); // returns NodeList
document.querySelector("p"); // returns Node
```

The objects returned from these selectors contain a large amount of keys/values.

Remembering dot and bracket notation is going to be quite substantial.

JavaScript provides the power to change anything and everything about an HTML element.

```
const el = document.querySelector("#myLink");
el.textContent = "Hello World";
el.classList.add("selected");
el.setAttribute("href", "/hello.html");
```

Creating elements is quite simple.

Use the document.createElement() method.

The tag name of the element is the parameter of document.createElement().

```
const header = document.createElement("div");
const headerPara = document.createElement("p");
```

Adjust the new element by defining properties.

```
header.style.width = "500px";
header.style.height = "200px";
header.style.backgroundColor = "pink";
headerPara.innerText = "Welcome to the Our Site!";
headerPara.style.fontSize = "30px";
```

To place these objects into the DOM, use the ParentNode.append() method.

This method will add the argument as the last child of the said parent.

```
document.body.append(header);
header.append(headerPara);
```

Memorizing all of the properties and methods that the DOM provides is nearly impossible.

Each DOM node has a massive list of properties.

Do not stress trying to memorize everything.

Document methods/properties

Element methods/properties

EVENTS

The browser can register user interactions or events.

If a registered event fires it will execute a function.

These functions can be bound to elements through event handlers.

To attach an event to a node the EventTarget.addEventListener() method is used.

The first argument is the event name.

The second argument is a function that executes when the event fires.

```
element.addEventListener("event_type", handlerFunction);
```

The handler function takes a parameter which has all of the information about the event.

"e" or "event" is a common name for the parameter.

```
function clickHandler(event) {
  console.log("Clicked", event);
}

element.addEventListener("click", clickHandler);
```

Event listeners can also be removed.

If there is a need to remove an event listener it is vital to use a function reference as the event handler and *not* an anonymous function.

```
function clickHandler(event) {
  console.log("Clicked", event);
}

element.removeEventListener("click", clickHandler);
```

Event	Description
`load`	When a page finishes loading.
`unload`	When a page is unloading.
`error`	When a JavaScript or asset error occurs.
`resize`	When the browser window resizes.
`scroll`	When the user scrolls.

Event	Description
`keydown`	Pushing down on a key.
`keyup`	Releasing the key.
`keypress`	When a key is pressed.

Event	Description
`click`	A standard mouse click.
`dblclick`	A standard double click.
`mousedown`	Initial press of the mouse.
`mouseup`	Releasing the mouse.
`mouseover`	Mouse moved over an element (not on touch screen).
`mouseout`	Mouse moved off of an element (not on touch screen).

There are also:

- focus events when the focus is on or leaves an object
- form events for form interactions
- also, many others

Click Demo

Mouse Over Demo

EVENT DELEGATION

The process of adding events to nodes gets complicated rather quickly.

Consider the following situation:

- When the script executes, it adds a click event to all paragraphs that exist in the DOM.
- A button click adds a new paragraph to the DOM.
- Clicking on the newly appended paragraph does nothing.

The click event mentioned above does not execute.

Why not?

When elements are dynamically inserted into the DOM (after the script executes), elements have no event listeners.

A common fix for this is to add the event listener to a common ancestor.

By doing so, all children of the ancestor or parent will inherit the click event.

Now the event handler requires some additional logic to check if we clicked the correct element.

For example, rather than adding a click event to all listed items, the click event can be added to the unordered list.

Following this process allows for any listed item dynamically inserted into the unordered list to inherit the click event.

EVENT BUBBLING

When an event fires, the event will "bubble" up to the highest level node in the DOM.

If any ancestors have a click event, that click event is fired during the "bubbling" phase.

```
<div>
  <button>Click Me</button>
</div>
```

```
document.querySelector("button").addEventListener("click", function(event) {
  event.stopPropagation();
  console.log("button was clicked");
});

document.querySelector("div").addEventListener("click", function(e) {
  console.log("div was clicked");
});
```