* The **console** is a tool that developers use to record the output of their JavaScript programs. The **console.log()**command is used to print, or log, text to the console. **console.log("Hello!");**
* Data type ***Null*** — Can only be null. It represents the absence of value.
* add .property to an instance of an object to return the value of that property e.g. **console.log(“Hello”.length);** //5. You can also add .method() to an instance to return values on the object. methods can be built in or custom made. e.g. **console.log(“Hey”.startsWith());** //H
* Libraries contain methods that you can call without creating an instance e.g. **console.log(Math.random()\*50);** //returns random number 0-50
* **const**, short for constant, is a JavaScript *keyword* that creates a new variable with a value that cannot change.
* You can create and reassign variables that you create with the **let** keyword.
* ***Undefined*** data type is assigned to variables that are not yet assigned a value.
* String interpolation in ES6 version of javascript: **let myPet = 'armadillo' console.log(`I own a pet ${myPet}.`)** this uses backticks instead of quotes and plus symbol
* If it exists but contains one of the following values then it is **falsy** meaning it will evaluate to false when used as conditional logic: ***false, 0 and -0, "" and '' (empty strings), null, undefined, NaN (Not a Number), document.all***
* To check if two things equal each other, we write **===**. To check if two things *do not* equal each other, we write !==. Use a single = to assign a value to a variable. Use ===to compare the values of two different variables.
* Logical operators: To say "both must be true," we use && and To say "either can be true," we use ||.
* A *function declaration* is a function that is bound to an identifier or name. **function square (number) { return number \* number; }** Function declarations require the keyword function, a name, and a function body.
* *arrow function* syntax: **variableName = () => {...}** this is a variable that stores a function
* With a *function expression* the identifier can be omitted, creating an anonymous function. Function expressions are often stored in a variable. **const square = function (number) { return number \* number; };** Also note function expressions end with a semi-colon since they are stored in a variable.
* .**push()**allows us to add items to the end of an array. **arrayName.push('item 3', 'item 4');** Similarly **.pop(),** removes the last item of an array. **arrayName.pop();**
* Variables that contain arrays can be declared with **let** or **const**. Even when declared with const, arrays are still mutable; they can be changed. However, a variable declared with const cannot be reassigned.
* methods called *iterators*, are called on arrays and complete such tasks as altering each element and selecting elements that fit certain criteria.
* **.forEach()**will execute the same code on each element of an array. **arrayName.forEach(function(item) { console.log(' - ' + item); });** alternatively you can use: **arrayname.forEach(item => console.log(' - ' + item));** .forEach() method: is an array method. It must be called upon an array. Any changes to the iterated array value won't be updated in the original array.The return value is undefined.
* **.map()**works just like .forEach but it returns a new array with elements that have been modified by the code in its block. **let bigNumbers = numbers.map(function(number) { return number \* 10; });** or alternatively: **let bigNumbers = numbers.map(numbers => numbers \* 10);**
* **.filter()**returns a new array. However, .filter() returns certain elements from the original array that evaluate to truthy based on conditions written in the block of the method. **let shortWords = words.filter(function(word) { return word.length < 6; });** or **let shortWords = words.filter(word => word.length < 6);**
* JavaScript ***objects***are containers that can store data and functions. The data we store in an object is not ordered — we can only access it by calling its associated *key*. **let restaurant = { name: 'Italian Bistro', seatingCapacity: 120, hasDineInSpecial: true, entrees: ['Penne alla Bolognese', 'Chicken Cacciatore', 'Linguine Pesto'] };**
* Properties in objects are separated by commas. Key-value pairs are always separated by a colon.
* To access the properties within an object, we connect the value's name to the key name with a period: **restaurant.entrees.** Another way to access a key's value is with *bracket notation*. **restaurant['entrees'].** One advantage that bracket notation has over dot notation is that you can use variables inside the brackets to select the keys of anobject.
* You can add more properties to an object by using **restaurant['appetizers'] = ['Fried Calamari', 'Bruschetta'];** or **restaurant.desserts = ['Homemade Tiramisu', 'Cannoli'];** Use the same to edit an existing property
* When objects have key-function pairs, we call the function a ***method*.** **const restaurant = { openRestaurant: () => { return 'Unlock the door, flip the open sign. We are open for business!'; }};**
* In the new version of javascript functions are written without arrow syntax or colon like this: **openRestaurant() { return 'Unlock the door, flip the open sign. We are open for business!'; }**
* use the **this** keyword to access properties inside of the same object. It is a dynamic variable that can change depending on the object that is calling the method.
* Getter and setter methods get and set the properties inside of an object.
* Developers use an underscore before a property name to indicate a property or value should not be modified directly by other code.
* Before changing the value of an object key it might be safe to first check if the value is valid, then after confirming all ow the value to be set. This should be done in a setter method: **set seatingCapacity(newCapacity) { if (typeof newCapacity === 'number') { this.\_seatingCapacity = newCapacity; console.log(`${newCapacity} is valid input.`); } else { console.log(`Change ${newCapacity} to a number.`)**
* *Classes* are templates for objects.
* JavaScript calls the constructor() method every time it creates a new *instance* of a class**. constructor(name) { this.name = name; this.behavior = 0; }.** Class names should have first character capitalized
* An *instance* is an object that contains the property names and methods of a class, but with unique property values.
* When multiple classes share properties or methods, they become candidates for *inheritance* — a tool developers use to decrease the amount of code they need to write. With inheritance, you can create a *parent* class (also known as a superclass) with properties and methods that multiple *child* classes (also known as subclasses) share. The child classes inherit the properties and methods from their parent class. **class Cat extends Animal { constructor(name, usesLitter) { super(name); this.\_usesLitter = usesLitter; } }**
* The extends keyword makes the methods of the parent class available inside the child class. The super keyword calls the constructor of the parent class.
* Static methods are called on the class, but not on instances of the class.
* Transpilation is the process of converting one programming language to another. JavaScript library Babel *transpiles* ES6 JavaScript to ES5. **npm install babel-cli** .This installs one of the two required Babel packages. **npm install babel-preset-env** . This installs the second of two required Babel packages. **npm run build**will save a converted version of the file in your lib folder
* The first step is to place your ES6 JavaScript file in a directory called **src**. From your root directory, the path to the ES6 file is **./src/main.js**
* Before we install Babel, we need to setup our project to use the [node package manager (npm)](https://docs.npmjs.com/getting-started/what-is-npm" \t "_blank). Developers use *npm* to access and manage Node packages. Node packages are directories that contain JavaScript code written by other developers. You can use these packages to reduce duplication of work and avoid bugs.
* Before we can add Babel to our project directory, we need to run npm init. The npm init command creates a **package.json** file in the root directory.
* A **package.json** file contains information about the current JavaScript project. Some of this information includes:
* Metadata — This includes a project title, description, authors, and more.
* A list of node packages required for the project — If another developer wants to run your project, npm looks inside **package.json** and downloads the packages in this list.
* Key-value pairs for command line scripts — You can use npm to run these shorthand scripts to perform some process. In a later exercise, we will add a script that runs Babel and transpiles ES6 to ES5.
* If you have Node installed on your computer, you can create a **package.json** file by typing npm init into the terminal.
* The terminal prompts you to fill in fields for the project's metadata (name, description, etc.)
* You are not required to answer the prompts, though we recommend at minimum, you add your own title and description. If you don't want to fill in a field, you can press enter. npm will leave fill these fields with default values or leave them empty when it creates the **package.json** file.
* you need to specify the version of the source JavaScript code. You can specify the initial JavaScript version inside of a file named **.babelrc**. In your root directory
* Inside **.babelrc** you need to define the *preset* for your source JavaScript file. **{ "presets": ["env"] }**
* We need to specify a script in **package.json**that initiates the ES6+ to ES5 transpilation.
* Inside of the **package.json** file, there is a property named "scripts" that holds an object for specifying command line shortcuts
* **"scripts": { "test": "echo \"Error: no test specified\" && exit 1", "build": "babel src -d lib" }** When run, the "build" script should use Babel to transpile JavaScript code inside of the src folder and write it to a folder called lib. Then From the command line, we type: **npm run build**
* The command above runs the build script in **package.json**. Babel writes the ES5 code to a file named **main.js** (it's always the same name as the original file), inside of a folder called lib
* JavaScript ***modules***are reusable pieces of code that can be exported from one program and imported for use in another program.
* The pattern we use to export **modules** is: Define an object to represent the module. Add data or behavior to the module. Export the module. **let Menu = {}; Menu.specialty = "Roasted Beet Burger with Mint Sauce"; module.exports = Menu;**
* To make use of the exported module and the behavior we define within it, we import the module. A common way to do this is with the **require()**function. **const Menu = require('./menu.js');**
* As of ES6, JavaScript has implemented a new more readable and flexible syntax for exporting modules. **let Menu = {}; export default Menu;** export default uses the JavaScript export statement to export JavaScript objects, functions, and primitive data types.
* ES6 module syntax also introduces the **import** keyword for importing objects to replace require(). **import ModuleName from './filePath';**
* ES6 introduced a second common approach to export modules. In addition to default export, *named exports* allow us to export data through the use of variables. **let specialty = ''; function isVegetarian() { }; export { specialty, isVegetarian };**
* To import (named import) objects stored in a variable, we use the import keyword and include the variables in a set of {}. **import { specialty, isVegetarian } from './menu';** We can then use these objects as in within our code. For example, we would use**specialtyinstead** insteadof**Menu.specialty.**
* In the case of requesting information from another site, we separate the request (asking another website for information) from what we want to do with the response (the information the website returns to us). We can do this using a system of technologies called *Asynchronous JavaScript and XML*, **or AJAX.**
* **GET**requests receive information from other sites by sending a *query*. **POST** requests can change information on another site and will receive information or data in response. A GET request is like a search. It is not introducing any new information. POST requests, on the other hand, introduce new information to another website. Instead of sending this information in the URL of the request, it is sent as part of the body of the request.
* *XMLHttpRequest object*, a JavaScript object that is used to retrieve data. Alternatively you can use **$.ajax({});** in jQuery to send and retrieve data with less code. Another further alternative is to use **$.get()** or **$.post()** or **$.getJSON()**
* In ES6 you can also use the **fetch** function. We call the fetch() function and pass it a single argument - the URL of the API endpoint. Because this is a GET request, this URL will contain the URL to the API and may also contain query parameters, an API key, a client ID, or other information necessary to make the request . fetch() is a web API that can be used to create requests. fetch() will return Promises.
* A Promise is an object that acts as a placeholder for data that has been requested but not yet received. Eventually, a Promise will *resolve* to the value requested or to a reason why the request failed. If the requested information or any error except a network error is received, the Promise is *fulfilled* and calls a function to handle the response. If there is a network error, the Promise is *rejected* and will call a function to handle the error.
* We chain a **.then()**method to the closing parentheses of the fetch() function. This is where the asynchronicity of JavaScript comes in - the fetch() function makes the request and returns the response, and we don't call the function that will handle the response until it has been received. .then() takes two callback functions as parameters, the first of which handles success and the second of which handles failure. The first callback function takes response as a parameter. response is the resolution of the Promise returned by the fetch() function.
* ES7 introduces an alternative to fetch/then called **async/await.** async is a keyword that is used to create functions that will return Promises. await is a keyword that is used to tell a program to continue moving through the message queue while a Promise resolves. await can only be used within functions declared with async.