Q1 - Provide an overview and description of a standard source control process for a large project

Source control for a large project at the most fundamental level remains similar to a smaller project. Developers will fork and clone the codebase from a repository, work on their feature, push it and create a pull request to be reviewed before being merged with the main branch.

As projects get larger, it can be beneficial to split the project into completely separate repositories, this is known as a multi-repo repository strategy. Managing the codebase in this way facilitates modularity and allow for separate teams to work independent of the rest of the project. This has many advantages including ease of modular code release and allows for seamless integration of AGILE principles into the development process. Companies such as Netflix, Amazon and Lyft are examples of large companies with extensive codebases that implement this approach.

This process allows for each library to be versioned independently of the rest of the project. While one advantage to this is that teams can work at their own pace and update their libraries independently, careful management is required to ensure that libraries across repos are adapted together. Teams must communicate to ensure that changes incorporated do not break code by creating changes in another teams repo that will cause tests to fail.

A hybrid-repo approach uses tools to assist in this synchronisation by providing CLI tools to execute commands across all repos at the same time.

https://kinsta.com/blog/monorepo-vs-multi-repo/ https://www.linkedin.com/pulse/how-use-version-control-systems-large-multi-part-software-hadi-safari

Q2 - What are the most important aspects of quality software?

https://www.silasreinagel.com/blog/2016/11/15/the-seven-aspects-of-software-quality/

Reliability - Software needs to be reliable to ensure it functions as intended thorough all possible use cases without failure. For example, a piece of software that controls bank transactions failing due to an uncaught error would greatly and negatively impact its users.

Efficiency - Efficient software is software that is written in such a way that it doesn't use unnecessary resources to accomplish it's task. One example of inefficiency in software is software that is slow due to poorly written API calls that return unnecessary data that the software must process but is not used. Inefficient software can also perform slower than necessary as it is wasting computer resource by performing operations that are more resource intensive than necessary or operations that are unnecessary altogether.

Portable - Portable software contains code that is not tightly coupled to one platform and requires minimal work to move it to another platform. Interpreted languages like JavaScript are intrinsically mobile, though using libraries and writing code that has as few platform specific dependancies helps ensure complete portability. Issues with non-portable software include maintenance problems, scaling difficulties and issues with readability/understandability.

Testable - Testability in code goes hand in hand with reliability. It helps ensure that software is robust and performs as expected. Software testability, when executed using test driven development, helps greatly across the entire development process. By planning and testing discrete portions of definable work within a larger project, each part is ensured to function reliably before being integrated with the larger codebase.

Modifiable - Modifiable software is flexible software than can be readily changed or updated to suit the needs of the end user. For example, it shouldn't require large amounts of code changes or refactoring to make small changes. The qualities above all play a part in ensuring software is modifiable.

Understandable - good quality software contains code that is readily understandable. While not directly related to performance, (this quality is related to human, rather than machine interpretation) though writing code understandable code is easier to maintain, update and troubleshoot because less time is spent by the developer trying to understand how the code functions.

Q3 - Outline a standard high level structure for a MERN stack application and explain the components

https://www.simplilearn.com/tutorials/mongodb-tutorial/what-is-mern-stack-introduction-and-examples

MERN is a web stack of Web development technologies. It consists of MongoDB, Express.JS, React.JS and Node.js. As all the technologies are JavaScript based, it allows for fast, scalable and robust web app development using only JavaScript (along with JSON) to develop.

Stack technologies -

MongoDB is the document based database management system. It uses JSON based document structures instead of tables with columns and rows to store data. React.JS is a front end JavaScript framework, Node.JS allows JavaScript to run in the terminal through a server environment and finally Express.JS is a Node.JS web application backend framework for building restful APIs.

Stack architecture -

https://dev.to/kingsley/mern-stack-project-structure-best-practices-2adk
https://shubhamjha25.medium.com/understanding-the-mvc-architecture-in-the-mern-stack-aff893abce50

MERN stack projects will typically follow an MVC - Model, View, Controller structure. The MVC structure separates the application into three discrete elements as seperate components that can be developed independently.

The Model component deals with the database. MongoDB is used here to store the application data.

React serves as the view component in the application architecture. The React app runs client side and is rendered in the browser by creating the HTML and CSS which the browser renders to screen for the end user to see and interact with. When data is manipulated by the user or returned to react from the backend. The React app also handles the user input which is passed onto the controller. It also processed information received from the controller as JSON into HTML to be rendered by the browser through the DOM.

The controller/backend component in the MERN stack is an Express.js app running within the node.JS server environment. This handles the business logic of the application and makes calls to the MongoDB database to read and write. It handles this data with Models and Schemas to validate and sanitize the data coming from or being sent to the MongoDB database.

Authentication/Validation requests are also handled by the Node/Express backend, making calls to the database to verify user information provided by the react frontend. By placing this business logic in the backend, away from the end user, this protects API endpoints from being accessed by unauthenticated users.

Q4 - A team is about to engage in a project, developing a website for a small business. What knowledge and skills would they need in order to develop the project?

A project such as developing a website requires many varied skills to succeed. These include technical, organisational, social and business skills.

Firstly, the project will require technical skills in order to produce the product for the client. These are web development skills for a given stack which has been decided on for the project. These will include knowledge of a front and back end development systems, database development systems and general computer knowledge required to use and build with these software. The required web development skills also include the problem solving skills specific to web development, such as searching for fixes to encountered bugs online or possible solutions to unforeseen issues that come along in the development process.

Organisational skills will be required to ensure deadlines are met by using the skills required to adequately plan and delegate the project into its tasks. This includes knowledge and experience with a planning system such as Trello along with general knowledge of the web development process in order to assign tasks and accurately estimate completion times.

Business skills are required in order to successfully ensure the team is all paid for and profitable. Knowledge of legal requirements is also required to ensure the team is not exposed to liability and they are insured at their place of work. General business knowledge will also assist in informing the team of the requirements and expectations of a small business client.

High quality communication skills are critical to the success of the project. These is required to ensure the team members communicate effectively between themselves and also the team communicates effectively with the customer. Good client communication ensures that every stakeholder is on the same page. This will help avoid disappointment or confusion that can lead to a poor outcome or expensive miss-understandings that require work to be redone.

Q5 - With reference to one of your own projects, discuss what knowledge or skills were required to complete your project, and to overcome challenges

python API project

necessary knowledge -

- python programming language
- understanding of API model/controller architecture
- understanding of order of development (what needs to be developed first in order for other stuff to work)
- psql
- understanding how a database management system works

- planning skills to hit deadline and efficiently use time
- time management skills to get it done

Q6 - With reference to one of your own projects, evaluate how effective your knowledge and skills were for this project, and suggest changes or improvements for future projects of a similar nature

- Python skills were sufficient to complete the task though deeper understanding would have allowed for a better result with more complex features.
- As the project was a learning experience, system architecture, database knowledge, development process and programming language skills all improved significantly during the process. On near future projects I will need to push myself to see similar knowledge and skill improvements
- As I'm poor at planning generally, I need to overemphasis this element of development in future projects to ensure adequate planning occurs.

and time managements skills all need to be more actively managed, linked to planning, using clear expected outcomes for coding sessions and try not to get side tracked.

Q7 - Explain control flow, using an example from the JavaScript programming language

Control flow is the order in which lines or blocks of code are executed when run. By default, JavaScript is a sequentially executed programming language. This means that without encountering explicit logic or statements to alter the flow, code will be executed line by line, from top to bottom.

Flow altering code can typically be broken down into two categories,iterative and conditional logic. Conditional statements evaluates a set condition and either skip or execute code depending on a given condition being true or false. In JavaScript the main conditional statements are if/else/else if and switch statements.

If, else and else if statements work exactly as can be inferred from the words themselves. An if statement will skip or run code depending on the truthy or falsy value of a condition. An if statement can be used by itself or in conjunction with else and else if statements. An else statement will run code only if the preceding if statement has NOT run and will always execute if the if statement doesn't run. An else if statement works similarly except it will only run if both the preceding if statement did not run and a further condition is met.

A switch statement is an efficient way of executing similar logic where there are multiple possible outcomes for an expression to be tested against. A switch statement executes an expression only once, and will compare the result against multiple cases, executing code when a case matches or a default value when there are no matches.

Iterative statements, also known as loops, will repeat the same code until a condition is met. These can broadly be defined as for loops and while loops. While loops assess a condition and run some code if the condition is true, then reassess and repeat the processes until the condition is evaluated as false.

A standard for loop consist of 3 expressions, the first expression executes once at the start of the loop and typically defines a variable to be used by the loop logic. The second expression contains a conditional expression that defines when the loop repeats or stops. The third expression runs after each iteration of the loop.

The following example demonstrates these concepts, the code will run from top down, initialising the 2 variables and entering the for loop.

The for loop runs while the index variable is less than the length of someArray, 5 times in total.

Inside the loop the code meets a conditional if statement. The condition is assessed (is the new array 3 elements long?) and uses the index value of the for loop to access the index within the someArray variable and pushes this value to the new array.

It's important to note that the loop will still run even after the if condition has been met, skipping the if statement (as it now returns false) and logging the array.

```
let someArray = [0, 1, 2, 3, 4]
let newArray = []


for (index = 0; index < someArray.length; index ++) {


    if (newArray.length != 3) {
        newArray.push(someArray[index])
    }
    console.log(newArray)
}

// => [ 0 ]
// [ 0, 1 ]
// [ 0, 1, 2 ]
// [ 0, 1, 2 ]
// [ 0, 1, 2 ]
```

Functions, which are essentially code that the code itself can call, must be defined and used with sequential logic, ie, they must be defined before they are called as the program will not have read the function definition if the program attempts to call it first.

Q8 - Explain type coercion, using examples from the JavaScript programming language

Java Script is a dynamically typed language, this means that while they can be, data types do not need to be explicitly defined when declaring a variable. If no data type is explicitly defined, JavaScript will automatically define it for you. In much the same way, JavaScript can also change a variable's type automatically when needed to perform a task in the program.

Implicit type coercion is the process of converting data between data types automatically without the programmer explicitly stating the conversion. While extremely useful to the developer as its use can create cleaner and simpler code to read, implicit type coercion is limited in JavaScript and not all data types can be coerced.

JavaScript will only implicitly coerce values into strings, numbers or boolean values. It is important to understand how JavaScript 'decides' which type to coerce into so unforeseen issues do not arise. For example, in the following code, JS will convert the boolean value 'true' to the number value '1', use the mathematical + operator to add the number 3.

```
let someNum = 3
let someBool = true

someResult = someNum + someBool

console.log(someResult) // => 4
console.log(typeof(someResult)) // => number
```

wheras in the following code, providing 3 as a string value will result in both values being coerced into strings and the + concatinating them together.

```
let someString = "3"
let someNum = 3
let someBool = true

someResult = someString + someBool

console.log(someResult) // => 3true
console.log(typeof(someResult)) // => string
```

JavaScript provides two equality comparison operators. If the developer wishes for JS to use type coercion while performing the comparison the '==' (loose quality operator) can be used. Otherwise the '===' operator (known as the strict equality operator) only returns true when the data is both equal value and equal type.

for example the following code will log 'not the same type' because while the value is equal, the type is not.

```
let someNum = 3
let someString = "3"

if (someNum === someString) {
    console.log("the same type")
} else {
    console.log("not the same type")
}

//  => 'not the same type'
```

by comparison, the following code will log 'the same type' because javascript used type coercion to compare equal data types

```
if (someNum == someString) {
    console.log("the same type")
} else {
console.log("not the same type")
}

// => "the same type"
```

Types can also be coerced explicitly in JavaScript. Typically using built in JavaScript methods. Using the example above, using explicit type conversion, the string can be coerced into a number or vice versa to control how the program executes the operation.

```
let someString = "3"
let someBool = true

someResult = Number(someString) + someBool

console.log(someResult) // => 4
console.log(typeof(someResult)) // => number
```

Similarly, the boolean value can also be coerced into a string

```
let someString = "3"
let someBool = true

someResult = Number(someString) + String(someBool)

console.log(someResult) // => 3true
console.log(typeof(someResult)) // => string
```

It is important to note that there are limits to type coercion and not all values can be, or will return a useable value once coerced. While the array in the following code contains values that can individually coerced into a number. Though,the array object itself cannot be.

```
let someArray = [6, 4, "6", [6,"230", true]]

console.log(Number(someArray)) // => NaN
```

While the String function will output the values contained within the array as a string

```
let someArray = [6, 4, "6", [6,"230", true]]

console.log((String(someArray)))  // => 6,4,6,6,230,true
```

Q9 - Explain data types, using examples from the JavaScript programming language

In order for a program to use and interact with some data within a variable, it must first know what that data is. Data types represent the classifications of different kinds of data and inform a compiler or interpreter how to treat it.

JavaScript uses the following primitive data types; null, undefined, boolean, string, number, bigint and symbol. It has one complex data type - object.

String data are type cast by wrapping the text in double or single quotes around the text. All primitive data types can be coerced or type cast into the string data type. In the example below, JavaScript will automatically coerce data types into string for string interpolation.

```
let someString = "this is a string"
console.log(typeof(someString))// => string

let someNumber = 0
console.log(typeof(someNumber)) // => number

let someBool = true
console.log(typeof(someBool)) // => Boolean

console.log(typeof(someString + someNumber + someBool)) // => string
```

The number data type represent whole number ints as well as float point numbers. The plus symbol acts as a mathematical operator rather than a joining operator. Numbers can be a maximum of 15 digits long. For larger numbers, the bigint type must be used. Any variable that contains only a numeric value will be coerced into the number type by JavaScript.

```
let someNum = 42
let someOtherNum = 1329.333
let notANum = "$30.00"

console.log(typeof(someNum)) // => number
console.log(typeof(notANum)) // => string
```

Bigints are used to store number variables that are too large for the number type. They are defined by placing an n after the last digit of the variable.

The bigint type is similar to the number type except that is cannot be used on any methods in the built in JavaScript Math object. In order to use Math object methods, they must be coerced to number types first, though precision will be lost beyond 15 digits.

```
let someBigint = 123456789123456789123n

console.log(typeof(someBigint)) // => bigint

bigintNumber = Number(someBigint)

console.log(bigintNumber) // => 123456789123456800000
console.log(someBigint) // => 123456789123456789123n
```

Boolean data types are either true or false. Any other value that isn't a null value will be true when coerced into a boolean type and a boolean value will be represented as 1 or 0 for true and false respectively when coerced into a number.

The object data type is used to store a collection of indexed data that can be iterated over. It will typically refer to a 'thing' that has properties or methods. Arrays are a type of object that use numbers to index their data.

```
let list = ["one", "two", "three", 4]

console.log(typeof(list)) // => object
console.log(list[2]) // => three
```

Q10 - Explain how arrays can be manipulated in JavaScript, using examples from the JavaScript programming language

There are many ways to manipulate arrays in JavaScript. The primary way we ado this is through the use of built in JavaScript methods to achieve just about anything you would wish to do to an array.

for removing or deleting elements, the pop() and push() methods can be used to remove the last element or add a new element to the end respectively -

```
let someArray = [0, 1, 2, 3, 4]

someArray.pop()
console.log(someArray) // => [ 0, 1, 2, 3 ]

someArray.push("new element")
console.log(someArray) // => [ 0, 1, 2, 3, 'new element' ]
```

the push() method is highly flexible as it can be used to add any data type or multiple elements to the array -

```
let someArray = [0, 1, 2, 3, 4]

someArray.push(["a","sub", "array"])
someArray.push(5,6,7,)
console.log(someArray) // => [ 0, 1, 2, 3, 4, [ 'a', 'sub', 'array' ], 5,
6, 7 ]
```

Some other examples of the many array manipulation methods are the slice() and splice() methods. The slice() method is used to extract the value of some elements from an array and returning them to a new array, it takes the start and finish index of the elements to be extracted and does not alter the original array. splice() works similarly but removes the elements from the original array.

```
let someArray = [0, 1, 2, 3, 4, 5, 6, 7]

let newArray = someArray.slice(2,5)

console.log(someArray) // => [ 0, 1, 2, 3, 4, 5, 6, 7 ]
console.log(newArray) // => [ 2, 3, 4 ]

let splicedArray = someArray.splice(2,5)

console.log(someArray) // =>[ 0, 1, 7 ]
console.log(splicedArray) // => [ 2, 3, 4, 5, 6 ]
```

the map() method iterates over each element, calling a callback function on each element and creating a new array of the returned values. This is extremely useful for many tasks, one example is creating an array of HTML elements from an array ready to pass onto the DOM -

```
let someArray = ["pink", "blue", "green", "purple","yellow"]

let list = someArray.map((element) => {
    return '<li>' + element + '</li>'
})

console.log(list) // =>

[
    '<li>pink</li>',
    '<li>blue</li>',
    '<li>green</li>',
    '<li>purple</li>',
    '<li>yellow</li>'
  ]
```

Array manipulation methods that do not alter the array, but rather make a copy (such as slice() and map()) are known as non-destructive methods. In contrast, destructive methods alter the array variable itself in some way (such as splice() and pop()).

It is important to understand how JavaScript treats the data in each array element in some manipulation methods. For example, the sort() method will sort an array in the following example, we can see how the method rearranges the elements into alphabetical order.

```
let someArray = ["orange", "apple", "watermelon", "lemon"]
console.log(someArray.sort())

// => [ 'apple', 'lemon', 'orange', 'watermelon' ]
```

If we attempt to use the same logic to sort a list of numbers we find that even though the elements are the number data type, the method coerces them into strings and arranges them into alphabetic order.

```
let someArray = [100, 2, 390, 28, 9, 6000]

console.log(someArray.sort()) // => [ 100, 2, 28, 390, 6000, 9 ]
console.log(typeof(someArray[0])) // => number
```

In order to use the sort method for the intended purpose of sorting by numerical order, it is necessary to pass a function to the sort method to compare the values.

```
let someArray = [100, 2, 390, 28, 9, 6000]

console.log(someArray.sort((num1, num2) => {
    return num1 - num2
})); // => [ 2, 9, 28, 100, 390, 6000 ]
```

This callback function subtracts one element from the if num1 is less than num2, it will return a negative value and num1 will be sorted below num2 and vice versa if num1 is greater than num2.

Q11 Explain how objects can be manipulated in JavaScript, using examples from the JavaScript programming language

Objects can be manipulated in many ways. For example, elements can be accessed using dot or bracket notation, values and properties can be enumerated in several ways and elements can be manipulated with operators.

Constructor functions can be used to create object instances -

```
function Instrument(name, family, pitch, volume) {
    this.name = name
    this.family = family
    this.pitch = pitch;
    this.volume = volume;
}
let violin = new Instrument("violin", "String", "treble", "")
```

The values of the violin object can be manipulated by both dot and bracket notation

```
violin.volume = "soft"
console.log(violin)

//  =>  Instrument {
    //      name: 'violin',
    //      family: 'String',
    //      pitch: 'treble',
    //      volume: 'soft'
//    }

violin["volume"] = "loud"
console.log(violin)

//  =>  Instrument {
    //      name: 'violin',
    //      family: 'String',
    //      pitch: 'treble',
    //      volume: 'loud'
    //    }
```

We can similarly use the same notation to add new properties to the object. Though only brackets notation can be used if the property name is not a valid JavaScript identified (ie, no spaces or hyphens, cannot be within another variable, cannot begin with a number etc.). In these cases, bracket notation must be used.

```
violin.timber = "maple"
violin["size of instrument"] = "small"
console.log(violin)

//  =>  Instrument {
    //      name: 'violin',
    //      family: 'String',
    //      pitch: 'treble',
    //      volume: 'loud',
    //      timber: 'maple',
    //      'size of instrument': 'small'
    //    }
```

The Object.keys() and Object.Values() methods return arrays of the keys or values of an object respectively.

```
console.log(Object.keys(violin))

// => [ 'name', 'family', 'pitch', 'volume', 'timber', 'size of
instrument' ]

console.log(Object.values(violin))
```

```
//  => [ 'violin', 'String', 'treble', 'loud', 'maple', 'small' ]
```

Q12 - Explain how JSON can be manipulated in JavaScript, using examples from the JavaScript programming language

JSON syntax is very similar to JavaScript objects. Because of this, a JSON string can be seamlessly parsed into a JavaScript array object with the JSON.parse() method to be manipulated. Conversely, in order to send data to an API or for any other reason, a JavaScript object can also be easily converted into a JSON string with the JSON.stringify().

In the following example a const 'max' is created by parsing the 'info' JSON string const.

```
const info = `{
    "name": "max",
    "age": 40,
    "nationality": "Italian",
    "favouriteFood": [
        "pasta",
        "pizza",
        "gnocci"
    ],
    "livesInAustralia": true
}`

const max = JSON.parse(info);

console.log(max)

// => {
//   name: 'max',
//   age: 40,
//   nationality: 'Italian',
//   favouriteFood: [ 'pasta', 'pizza', 'gnocci' ],
//   livesInAustralia: true
// }
```

This new object can now be manipulated as per any other array object. It is also worth noting that JavaScript seamlessly coerces the data types as we would expect.

```
console.log(typeof(max.age)) // => number
console.log(typeof(max.livesInAustralia)) // => boolean
console.log(typeof(max.favouriteFood)) // => object
console.log(typeof(max.name)) // => string
```

In order to convert an object into JSON, we use the JSON.stringify() method.

```
const max = {
  name: 'max',
  age: 40,
  nationality: 'Italian',
  favouriteFood: [ 'pasta', 'pizza', 'gnocci' ],
  livesInAustralia: true
}

const maxJSON = JSON.stringify(max)

console.log(maxJSON)
// => {"name":"max","age":40,"nationality":"Italian","favouriteFood":
["pasta","pizza","gnocci"],"livesInAustralia":true}
console.log(typeof(maxJSON))
// => string
```

Q13 - For the code snippet provided below, write comments for each line of code to explain its functionality. In your comments you must demonstrates your ability to recognise and identify functions, ranges and classes

```
// Defines car class
class Car {
    // constructor method assigns properties to the class
    // it is automatically called when creating an object and used to pass
unique values to the new object
    constructor(brand) {
        // the 'this' keyword is used to refer to a newly created object
        // when creating a new car instance, the carname property will be
assigned the brand value
        // passed as an argument to the class constructor

        // for example -
        // let newCar = new Car("Toyota") will create a new object
instance "newCar" of the Car class with
        // the value "Toyota" will be assigned to its carname property
      this.carname = brand;
    }
    // Declaring present method - returns a string interpolating 'I have a
' and the carname of the object
    // Method will belong to each object as per a property
    present() {
      return 'I have a ' + this.carname;
    }
  }

// Defines Model class which inherits all properties and methods from the
Car class
// Model is the child class and Car is the parent class
  class Model extends Car {
    constructor(brand, mod) {
        // super keyword referes to the parent class, used to invoke its
constructor
        // in this case, the Car class' 'brand' constructor is invoked and
a new Model object can be
        // passed a 'brand' argument and the instance will contain the
'carname' property
      super(brand);
    //   mod argument passed to the passed to the constructor will be
assigned to the model property
      this.model = mod;
    }
    // show() method returns inherited present() method and interpolates
it with ', it was made in ' and the model property for the object
    show() {
      return this.present() + ', it was made in ' + this.model;
    }
  }

//   define array variable with 3 string elements
  let makes = ["Ford", "Holden", "Toyota"]


// defines models variable and uses Array.from method to create a new
array
// Array.from takes an array/array like or other iterable oject to convert
to an array and a function to call on each element as arguments

// The Array constructor method as the first argument,
    // In this case the Array constructor is passed an integer (40) so
creates an array of 40 empty slots

// the second argument is the mapFn, an annonymous callback function,
called on each element of the array
    // the mapFn is passed the current element (x, in this case empty and
unused) and the index currently being processed (i)
    //  the function adds 1980 to the current element and this value is
assigned to the index in the models array

  let models = Array.from(new Array(40), (x,i) => i + 1980)

//   Declare function to pick a random number, takes 2 arguments, the
upper and lower range (inclusive) for the random number to be between
  function randomIntFromInterval(min,max) { // min and max included
    // Math.floor method rounds a number down to the nearest integer
    // Math.random method returns a random number between 0 and 1

    // the function calculates the difference between min and max +1 (to
make it inclusive of the min and max value)
    // It multiplies this by a random value between o and 1 to get a
random number somewhere within the difference of min and max
    // the min value is added to this to get a random value that is now
between the min and max value
    // Math.floor converts this into an Integer
      return Math.floor(Math.random()*(max-min+1)+min);
  }

//   for loop iterates over the models array
  for (model of models) {
    // defines make variable
    // assigns it the value of the element found in the makes array at an
index between 0 and the number of elements in the makes array
    make = makes[randomIntFromInterval(0,makes.length-1)]
    // defines model variable
    // assigns it the value of the element found in the models array at an
index between 0 and the number of elements in the makes array
    model = models[randomIntFromInterval(0,makes.length-1)]
    // creates a new object instance of the Model class, passing the make
and model variable just created as arguments
    mycar = new Model(make, model);
    // logs the .show() method to the console
    console.log(mycar.show())
}
```