Shelby El-rassi, T3A1, Flextrack, MEL

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

What is source control?

Source control (also known as version control) is the tracking and management of files/code and any changes made. The practice of source control is the tracking of any modifications and the ability to revert back to any previous versions in the event of mistakes or backtracking, a key feature is the ability to know which version is the current iteration. An excellent example of this is Git.

Git

Git is a free and open source version control system, it is designed for source code management, however it does have the ability to management other sets of files. Git is currently one of the most popular version control system for software development, as it is a distributed version control system, most Git operations are local, with every user/client having their owner server and full back up of the repository locally. It has the advantages of speed, feature branching, useful staging areas and the ability to run multiple workflows. Below is an overview of the git-flow branching model, which has become a popular version control model for software teams.

Git-flow Workflow

Git-flow workflow first published by Vincent Driessen at nvie. It describes a firm branching model based around a projects release. The hefty framework is ideal for management of larger projects.

Overview of this workflow:

- Getting started with this workflow starts with an actual toolset available for installation, after installation (which is available on multiple operating systems), utilising git-flow for a project involves executing git flow init. It essentially is an extension of the standard git init command, that creates branches.
- A central repository houses the ‘Master’ branch, which is for the core of the project reflecting a production-ready base, along with another branch named ‘Develop’, this branch runs parallel with the master and reflects a state of latest features for next release (also known as an integration branch). An important note is the ‘Develop’ branch will house the entire history of the project, whereas the ‘Master’ will only contain the latest version.
- When git flow init is utilised, this command will create the ‘Develop’ branch.
- When a new feature is started, a new branch is created for this feature. This can be done via git-flow extension by the command git flow feature start feature_branch.
- Each developer pulls and pushes to the origin ‘Develop’ as the parent branch.
- Developers may also pull changes from other peer’s branches, which can be useful for working together in sub-teams.
- Each branch should have a clear descriptive name with a clear purpose.
- Each feature branch should also be pushed to the central repository branch ‘Develop’, this means the code can be shared for collaboration.
- When complete these feature branches get merged back into ‘Develop’, never interacting with the ‘Master’ branch directly.

- Once a feature branch is finalised, the command git flow feature finish feature_branch can be executed.
- When the 'Develop' branch has accumulated the required features for release or a scheduled date is imminent, a 'Release' branch gets forked off 'Develop'. This ensures the next release cycle beings, so new features cannot be added after this point, only fixes, documentation and other release centred tasks.
- Once this 'Release' branch is ready to go, it gets merged into 'Master' and marked with a release number.
- The usage of a 'Release' branch for initial preparation of the release ensures that the rest of the team is able to continue working on features for the succeeding release date.
- It is important that the 'Release' branch also gets merged back into 'Develop' as well as 'Master' for any updates possibly added which new features might need access too. Once this process is complete the current 'Release' branch can be deleted.
- Another type of branch operated in this flow are 'Hotfix' branches, which are in place to swiftly fix up production releases. They operate just like feature branches, however, are parented off the 'Master' branch. A 'Hotfix' branch is the only branch forked directly off the 'Master'. Once the fix is fully complete, these branches get merged immediately into 'Master' and 'Develop', with the 'Master' tagged with an updated version number. Operating a dedicated branch just for branch is a key advantage to this workflow as it ensures issues can be addresses by the team, with total interruption of the rest of the workflow.

In summary, this workflow is great for the large projects operating on a release-based schedule, offering the specific branches enable this workflow management, minimising interruption.

[Sources]---

1. Reference
2. Reference
3. Reference
4. Reference
5. Reference

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

Through the perspective of software engineering, software quality implications involve two related, but separate concepts, these being;

- Software functional quality, being the base product quality model. ISO/IEC 25010 defines a set of eight software quality characteristics, measuring software through these aspects is to define what makes a product worthwhile amongst competitors and in relation to behaviour. These aspects are:

    1. Functional Suitability
    2. Performance Efficiency*
    3. Compatibility
    4. Usability
    5. Reliability*
    6. Security*
    7. Maintainability*

8. Portability.

- Software structural quality, which is the quality-in-use model. By extracting from ISO standards, CISQ (Consortium for Information & Software Quality) defined quality aspects at a source code level in order to support the functional level qualities, but also enabling measurement of quality in meeting these function requirements this determining if the software works as intended. These four important aspects are all derived from the original eight and involve them all, just at a more specified functional level. These aspects are:

    1. Reliability: This is referring to the presence of consistent behaviour of software and the overall stability of the program in the event of unforeseen conditions. Errors should either be correct handled with no effect on the user, or no error occurrence at all. The user should never question the execution of the program. To summarise, reliability is the characteristic of resiliency and structural solidity.

    2. Performance Efficiency: This is referring to how the software utilises resources and their execution time, amount of database calls and overall effect on program scalability, satisfaction and response times. The overall code architecture and design contributes to software efficiency. Additionally, is code compliance with best architecture and database practices. Overall, efficient code is fast, memory-efficient and consumes the fewest CPU cycles.

    3. Security: This is referring to protection in the case of security risks and how well this is executed. The likelihood of attackers breaching the software, interrupt processes and gain access to restricted information is a measure of the security quality and how well the code architecture was formed and put into practice. The measurement of this aspect is via ‘vulnerabilities’, which are known issues possibility resulting in a breach, the ability to detect them, the amount of these vulnerabilities found and the gravity of them indicate the overall security level. Other factors also include the resolution time taken to fix an issue, deployment of security updates and the actual amount of full system breaches as opposed to just vulnerability checks.

    4. Maintainability: This is referring to the ease in which software can be modified, utilised and adapted for alternative purposes and overall portability between environments and development teams. If compliance with software architecture practices and consistent high-quality code is produced across the board then easier streamlined maintenance is likely to be more achievable. Overall high-quality code is clean code that is consistent, easy to understand, well documented, is DRY, has good error handling and can be tested.

[Sources]---

1. Reference
2. Reference
3. Reference
4. Reference
5. Reference
6. Reference
7. Reference
8. Reference

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

The MERN stack is a JavaScript Stack utilised for building web applications and dynamic websites. It is made up of four technologies namely MongoDB, Express, React and Node.js which are all centered around JavaScript and make up a full-stack development framework. The way these technologies interact is based on a 3-tier architecture which is frontend, backend and database. Some key advantages of this stack are the technologies are free and open source, it covers the full development cycle and it supports the MVC architecture.

The MVC framework is the core architectural pattern of a MERN stack application. These being the Model, Views and Controller. The Model is the part of the application dealing and interacting with the database, the view is presenting of the data rendered on a user interface, and the Controller is the glue between Model and View where we can request data from our database (via the model) and then send a response to our view.

Client-side

React

React is a library used for front-end development(the view layer), it operates efficiently and on a declarative basis for client- side application creation in HTML. The overall user interface gets created via small isolated pieces of code called components, which can connect to the server-side data and get sent to the browser for rendering. React being component based credits itself in being able to refresh only parts of the webpage that need refreshing on user interaction. React was built with powerful algorithms that determine when a certain component does need refreshing, meaning it has a powerful performance basis. React offers reusability of components and great control over the state of events, which is key in the dynamic updating of large amounts of data.

A basis example of a React Component, is one that takes in parameters called props (which can come from the server-side) and using that data returns a order of views to display via the render method, this method returns what you want displayed to the UI, more specifically the return is a React element. Most commonly used for rendering is JSX syntax which is JavaScript XML, which enables the power of JavaScript to be used in rendering just by putting any expression within braces.

Server-side

Express

Express JS is a back-end web framework described as fast, unopinionated and minimal and it operates by running on top of Node JS. It is used as to simplify the task of writing server-side code, as it avoids having to use Node JS directly and writing server code from scratch. The benefits of Express include, modularisation, dynamic routing, additional support of middleware usage(ie. cookies, sessions and validation), and its emphasis on simplification means it makes web applications highly scalable and keep a simple code base.

Express acts as apart of the controller portion of the MVC layer, this means it is the bridging layer between the view (client-side) and the model(database tier).

A core task of the controller layer is the accepting of the HTTP requests(GET, PUT, POST, DELETE) and determining the next course of action based on the type of request, url pattern or even based off the data sent from the body. Express then will determine what course of action is next required, whether that be a database

query or other associated tasks. Then a response will be served to the browser which is usually a view sent for rendering with the retrieved data input in the rendering template.

Web Server

Node.js

Node JS is a back-end web server runtime environment allowing for development in JavaScript to be run outside the browser. This means JavaScript web applications can be built and run on a server, which allows for better scalability and performance optimisation. A key component to Node.js is the node package manager (NPM) which allows access to thousands of node modules which are free and reusable in the application and basically automate most of the building cycle. Node JS is the other part of the controller portion of the MVC architecture partnered with express. Once Node is installed and a package.json is initialised, utilisation of node packages is as simple as installing them via npm and importing using:

```
require()
```

Express itself is actually as node package and once is imported has all the needed methods for routing HTTP requests, middleware and rendering configuration and any other application modification for application behaviour.

Database

MongoDB

MongoDB is NoSQL document database which means each record is document key-value pair with similarities to JavaScript objects (technically it saves objects using JSON, however can also use binary JSON which is called BSON in the case of other data not JSON compatible) and are saved in collections as opposed to tables. An advantage of this type of database is the insertion of data without a predefined schema, fast exchanges between client and server and the ability to store very large volumes of data. In MongoDB collections, documents can contain multiple fields, and the keys paired to each field is able to be any data type which can include arrays and even links to other documents. Mongo paired with Mongoose are the basis of the model tier of the MVC architecture. Mongoose is an object data modelling(ODM) module in node that enables schema creation and is used to integrate Mongo with the application. Mongoose as a modelling environment, means when schemas are defined, additional data management can be done also such as forcing consistency and adding data validation which is important even for flexible schemas. Mongoose and Mongo are both installed via NPM and are used by intialisation and then establishing connections. Connecting to MongoDB via Mongoose just requires the URL of the database. Mongoose also provides the query methods, which are asynchronous so can either be accessed via the exec function which returns a promise or a callback function can be passed to the query and then handled as a promise. This data then retrieve is what can be passed through to the view fo rendering via Express routes as previously mentioned.

[Sources]---

1. Reference
2. Reference

3. Reference
4. Reference
5. Reference
6. Reference
7. Reference
8. Reference
9. Reference
10. Reference

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?

[Sources]---

1. label

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

During week commencing the 19th of October as part of the Coder Academy curriculum myself and my classmates all participated in an Express JS hackathon. The briefing was:

- 2 Developers per team.
- Develop a solution to a problem of choice with a chosen theme.
- The application must have:
    - Server-side: Express and Mongo
    - Client-side: HTML/CSS/JavaScript that requests the server and manages sent data.
- Git Workflow utilisation

In the process of this project myself and my partner needed a wide variety of knowledge and skills. These being:

- Git Workflow: For our Git workflow we decided on a forking workflow. This involved having a clear core understanding of the workflow and the processes and order of action, we both agreed we had. This was determined at the beginning stages of the Hackathon and a repository was set up immediately and forked by my partner. Setting up this workflow at the beginning and talking through the process gave us both a clear understanding of what we needed to do whilst working on particular area/features of the project. We were able to work on separate feature branches, then merge them into the main with very minimal merge conflicts due to keeping our communication front and centre in the Hackathon process. This meant during the timeframe the Git Workflow did not cause any issues or interruptions, which gave us the opportunity to focus on the application.

- Planning and documentation: The start of the project involved some initial planning which was done on a simple google doc. This involved talking through ideas and research and noting them down to form a basis of our application idea. Once an idea and them was decided we set up a Trello board to implement

a basic, but clear development plan for the 2.5days of coding. For this process to be successful we needed to have a good planning focus and methodology in which we could outline an implementation plan. This ability is what gave the Hackathon a direction to work towards. Contained in the Trello board were cards for the Server and client-side work, cards for doing done and a backlog. Additionally, a card for some user story outlining proved useful for reference. A nice to have card was added when we determined certain features and components were better on this card given the short time frame. Whilst usually more detailed mock-ups and wireframes we accompany planning, we still implemented a simple wireframe for the main application dashboard, this enabled direction for the front-end creation. Knowledge of creating wireframes was necessary, and Figma was the tool of choice in this process.

- MVC File Structure: Knowledge of the MVC architecture was required in this build process. This framework was the core architectural pattern of the application component separation. These being the Model, Views and Controller. The Model is the part of the application dealing and interacting with the database, the view is presenting of the data rendered on a user interface, and the Controller is the glue between Model and View where we can request data from our database(via the model) and then send a response to our view. Understanding the way this architecture works and operates together was required. In addition, another architecture component for an Express App was the Utilities, in order to keep our controller from getting too inflated, keeping logic in the utilities was beneficial.

- CRUD application concepts: The type of application created in this Hackathon was a CRUD application. Knowledge of CRUD (Create, Read, Update, Delete) functions and paired with the corresponding HTTP request was needed. This required an understanding of HTTP routing and requests. As team we did face certain challenges with our PUT and DELETE requests, this was due to not fully grasping the link to client-side HTTP requests connecting to the server-side, particular with integrating this with our rendering engine of choice PUG. We were able to work through this as a team, and through research and refreshing our knowledge on these topics during the Hackathon process.

- JavaScript: Knowledge of fundamental JavaScript was key in this process as this was the core of the project, without this basis the following could not be implemented.

- Server-side: A good knowledge basis for the following was required in order the create our server-side code.

    - Express: Web framework for Node. Core knowledge of setting up this framework and how it is implemented was needed. Express involved some heavy initial plumbing so understanding this process and how it connects was crucial. Along with this involved setting up Express routing.
    - Node: Understanding the basis of Node as a web server to process and deliver our data was needed and how these requests and responses were operating with routing.
    - Node Packages: These packages were utilised for a variety of application components, knowledge of using node packages and setting up the initializing of them was needed.
    - MongoDb: NoSQL database. Knowledge of databases particularly how NoSQL databases operate proved essential in utilising MongoDB as the document database. Understanding how to query the database and the methods to go about this was needed.
    - Mongoose: In order to persist the data in our Mongo database Mongoose was integrated. Understanding the creation of Model via a Mongoose Schema and how to define a Schema was needed, along with how to query the DB via Mongoose.
    - View Rendering: Implementation of a rendering engine was needed to render our data to a user interface. Whilst Express-Handlebars were the initial rendering engine outlined, Pug JS was

implemented due to its concise and clean structure. Understanding how to send the data to the view for rendering was needed, additional was implementation of adding styling to the Pug files.
    - Overall Plumbing: Implementing the basis plumbing and set up for the above components was needed for this application. An idea of their interaction and base set up is important, especially for troubleshooting if errors came along.

- Client-side

    - HTML/CSS: Knowledge of employing CSS for styling was required. Additionally, Bootstrap a styling framework was used to speed up the styling process of the front end due to time constraints. Knowledge of implementing this framework and the classes and particular components was required.
    - Scripts: The client-side JavaScript involved aspects including, but not limited too API calls, HTTP requests, DOM manipulation and utilising special JavaScript library which was Chart.js.

A core piece of knowledge/skill is the ability to problem solve and knowing where to go to learn a piece of information. Problem solving is a huge component in coding and working on a project. Knowing how to research and search for a solution when a problem arises was essential. All the pieces needed to successfully build an Express JS application are very large and comprehensive, an important part of being able to use them is to know how to use the documentation to find the process for a piece of code. This process was a fundamental process in the application implementation plan. Understanding the breakdown of steps and knowing where to find the answers when facing a block was crucial. The ability to breakdown a problem into smaller steps was also key. This proved very useful when faced broad code errors not easily able to be pin pointed, understanding how to break them down the debug one step at time was an important process. During this application building, I found my debugging process was improved by reflecting on this breakdown process.

Communication amongst the team. This important soft skill was essential in our application implementation process. In the starting code process, we pair programmed which involved a lead coder and a navigator. During this process we both held these roles at different times and both roles involve communication. Some key components to this effective communication were the explanation of our own code, the communication of steps when in the navigation position, expressing problems when they arose and the ability to convey them to the other teammate.

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.

In continuation of the aforementioned project in question 5, I would like to conduct a personal evaluation of my skills in relation to this project.

Evaluation:

Git Workflow:

This part of the project was important as it was the management of the project version repository. In relation to this part of the project my knowledge of Git was at a level in which I could comfortable operate a feature branch and submit pull requests for merging. This aspect of the project proved to run smoothly due to mine and my partners knowledge of this component. Some improvements for next time would be clearer feature separation on branches(branches were used, but only a few), and attention to detail on meaningful commit messages.

Commits were clear, but due to the time-frame, there were some not properly thought out. These improvements can easily be implemented in my next project.

Application Build

Using Express, Node and MongoDB for this project, meant it involved a sound understanding of some heavy concepts.

A key part of this process of server-side code is the initial implementation of the Express application 'plumbing'. Whilst the initial set up of all the code requirement for Express, Node and MongoDB was done with minimal error, my core understanding of these pieces of code could definitely be improved upon. I felt this was a case of it is implemented and it works, but is my understanding as well rounded of how the code is interacting? Some examples would be reviewing how the middleware implemented, such as Cors, is working and how Mongoose connects to MongoDB.

The application had the basis of a CRUD application with a MVC architecture, I felt my knowledge of this concept was effective as a feature I worked on was building the basis CRUD for the tasks, during which the implementation time of this was swift and with minimal interruption. This process involves setting up the routes, the task controller which connects to the routes, the code in the task utilities, which houses the core logic of CRUD and connects queries the database via the model. Understanding the request and response parameters was an important factor of this process. Database queries and implementing Mongoose for document modelling were realised. My knowledge of querying MongoDb was improved during this project build process, which proved effective during implementation. Our model design played an important role in our the game style of the tasks, with a completed boolean and points tally in the schema. Correct accessibility of this to update and retrieve points was necessary. This was able to be done with correct database queries and updating. In the build process we came across some errors due to using the required parameter in our task model, so due to limited time we took validation that out of the implementation, so future my Mongoose Modelling processes and validating could be improved and refined. In relation to the queries in the utils and also the functions with the controller, for future I would like to implement a stronger emphasis on error handling. Whilst some error handling was inputted, I felt there was not a clear focus on correct error handling and also uniform error handling throughout the application. Doing this in future I feel would smoothen the build process as errors would be clear or perhaps easily discovered.

As mentioned before, the controller functions were set up next with these being was the route will call. These functions are what executes the query functions in the utils. Understanding the flow of the function calls was crucial in this project. Along with this is what comes next which is being able to render the response received from the controller. In the case of the project PUG JS was chosen as the HTML template rendering engine, this was due to its clean syntax and it was a library I had recently started learning and utilising, so being able to practice implementation of it in this project was really beneficial to my learning. Initially understanding the format and syntax of PUG was a little confusing, but after some practice became relatively simple. So the API requests executed were able to be sent to the browser using PUG to render the desired content. An example is the listing of all the tasks on the users dashboard. Pug was able to render the tasks, the user(user logged in), and the tally of tasks and the total points. The utils houses the functions that tallied up the total tasks and total completed tasks, these functions were able to be called on the sent data in the controller. Doing this kept the controller 'skinny' in other words the logic was able to stay in the utils. Additionally the response sent to PUG to render included all the tasks which were sent as an object. Then in the dashboard PUG file, the object could be iterated over using clean PUG syntax. I found breaking down all the tasks in the PUG file using the one object sent to the browser was a simple and clear approach to rendering all the tasks for this project. In a real world

application situation, this approach, I feel, might not be the best for time, speed and memory as a user might eventually have a very large number of tasks (since it sends even tasks that are completed). In that case, a future alternative approach could include having a filter for just relevant tasks (the active ones) or other alternative approaches. PUG proved useful as a rendering engine as simple conditional statements were able to be implemented to do with certain specific data being rendered on conditions. I found this syntax and format simple and easily implemented. Integrating CSS with PUG was also done with ease in this project as CSS classes could be used, just with a different format. One area that I did get stuck on for awhile was including a separate JavaScript Script file for the client-side JavaScript. In the build process, I am not sure why, but I could not successfully include a separate Script JS file for the client-side JS and HTTP requests. Due to the time-frame, this was able to be managed by just using a PUG script tag at the bottom of the PUG files and house the JS there. For future having the client JS in a file on its own is ideal and more clearly organised, so that is a clear future improvement. Additionally to with HTTP requests, for future I would like to refine my knowledge of this process and implemented these requests, particular the DELETE and PUT requests. In this project, I felt my knowledge in this area was lacking as I was stuck on the implementation of these methods. In the end they were successfully done, however the PUT method was done with method override in PUG as that was a popular NPM for a PUT method form (updating a task) in PUG, however I would like to look further into what is the best practice for this. Additionally the DELETE method request works, however is a little 'buggy' in that the page does not correctly refresh and redirect to the desired location, it just stalls then manually page refreshing will finalise the delete. The best future improvements for this is revising my knowledge on these methods and the correct way to implement them.

Overall, my knowledge and experience was at a very good level to successfully implement (along with my partner) a MVP application. The best thing about the build process was the extra experience, practice and knowledge refinement I gained.

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

Control flow is the flow or order in which statements in a script are executed. Code is run line by line started from the first line then continuing until the whole code is implemented. Factors that alter the control flow are structures such as conditionals and loops. So overall control flow involves not just start to finish script reading, but the reading and analysing the overall structure of the program to determine order of execution dependant on how many instructions present that alternative the start to finish flow.

An example in JavaScript is when code execution reaches a control structure of a conditional if/else statement. A conditional statement is a set of instructions that executes if a specification is true. When the statement is reached the given expression is analysed and then chooses the direction the program takes based on if the logic is true or not. Else is an optional statement utilised to enable an alternate path. In the example below, is the condition is true, statement 1 is executed, if the condition is false, statement 2 will be executed.

```
    if (condition) {
        statement_1;
      } else {
        statement_2;
      }
```

[Sources]---

1. Reference

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

Type Coercion is the automatic or implicit conversion process of converting a value from one type to another type. Such as a string to a number. Adjacent to type coercion is type conversion, which is explicit conversion of data types, which involves the explicit requiring of a conversion to take place rather than automatic as aforementioned.

An example of this is the toString() method, which can return a converting number to a string as shown below. In this snippet to code is specifically calling a method of conversion on the number 15 which then results in a string 15.

```
(15).toString() //=> “15”
```

Type Coercion has differences to Conversion. Coercion is the term used when unexpected type casting happens in JavaScript. Coercion of a data type is often a side effect of different operations. For example, like the code shown above of a number to a string, a type coercion version of the same example would be:

```
15 + “” //=> “15”
```

Adding a string to a number will always result in a string. A reverse of the above example, coercing a string to be a number is below. In this example using the “-“ operator means JavaScript will cast the values to be numbers.

```
“15” - 1 //=> 14
```

These snippets are in themselves simple examples of Type Coercion, by they do show the way JavaScript when it is executing the code will even out the datatypes and cast them automatically to enable to operator to work.

[Sources]---

1. Reference
2. Reference
3. Reference

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

Data types like in any programming language tell the characteristic of a piece of data; this is conveyed to the compiler so that is can perform the applicable operation. In JavaScript data types can include primitive, which means these data types are immutable or unchangeable, and non-primitive or objects which are collections of properties in data usually referenced by an identifier. Objects are not immutable as they can have properties added and removed. The value properties in an object can be of any type, also including other objects, so this enables the creation of complex data structures.

Primitives

- Boolean: This type is representative as a logical entity, either the value true or the value false. Other data type values such as NaN, null and undefined are by definition a falsey value along with false, 0 and an empty string. All other values truthy by nature.
- Null: This value is representative of empty, or the intentional absence of a value.
- Undefined: This value is utilised when a variable has not been assigned a value yet, by default it becomes undefined.
- Number: Number is representative of numeric values. JavaScript is different to other languages in that it cannot distinct automatically an integer and float. So technically the type only has one integer, this being 0 and all other numbers are represented as a double-precision 64-bit floating point format. Other languages will know to round to the ideal representation, this is not the case for JavaScript. That is why methods like .round() exist in JavaScript to enable the rounding of integers. The number type can also including +Infinitity, -infinity and NaN(not a number).
- BigInt: A BigInt is representative of an integer with arbitrary precision, it is also numeric in nature. This value is utilised to store and operator on large integers safely when they go beyond the double-precision floating-point number range.
- String: A string is representative of textual data and comes in the form of a string of characters. JavaScript strings are immutable in that once created; they are unmodifiable directly. However, utilising .substr() to pick individual letters or .concat() to concatenate two strings together are alternatives to ‘modifying’ a string.
- Symbol: A symbol is representative of a unique and immutable value, such as the key in a JavaScript object.

Objects (non-primitive)

- Object: A object in JavaScript is representing of key,value pairs which are a list of zero or more property names and corresponding values, which are enclosed in curly braces.
- Array: Arrays are also by definition objects, however their relationship differs in that the key is the integer set index paired with the length property of the array.

There are many other built-in JavaScript objects that come in the form of a standard library, an example of these are:

- Error
- Math
- Set

[Sources]---

1. Reference

2. Reference
3. Reference

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

JavaScript arrays are by definition JavaScript objects, with a key being the integer set index paired with the value being the length property of the array. This enables data to be stored in array in an adjacent fashion. Arrays are not immutable, meaning the elements are not fixed as in without the ability to be changed. This means an arrays length is able to be altered at any point, because of this an array can be empty or it can be of great substance, depending of the needs of the program. Array methods which allow for this alteration of length are methods which enable operations of traverse and mutative nature. Below are examples of array manipulation methods.

Adding and Removing Array Elements

unshift(), shift(), push(), pop() and splice()

These powerful array methods operate by allowing to add or remove specific array elements. These methods are powerful as they do not return a new array, rather alter the original array.

unshift(), shift()

unshift() operates by taking the given value and adding it/them to the start of the array and then returns the altered original arrays length.

```
let letters = [ "a", "b" ];
let newLength = letters.unshift ( "c", "d" );
console.log(letters);    //"c,d,a,b"
console.log(newLength); //"4"
```

shift() operates removing the first element in the array and then returning it.

```
let letters = [ "c","d", "a", "b" ];
let letter = letters.shift ();
console.log(letters);    //"d,a,b"
console.log(letter); //"c"
```

push(), pop()

push() operates by taking the given value and adding it/them to the start of the array and then returns the altered original arrays length.

```
let letters = [ "a", "b" ];
let newLength = letters.push ( "c", "d" );
```

```
console.log(letters);   // "a,b,c,d"
console.log(newLength); //"4"
```

pop() operates removing the last element in the array and then returning it.

```
let letters = [ "a", "b", "c", "d"];
let letter = letters.pop ();
console.log(letters);  // "a,b,c"
console.log(letter); //"d"
```

splice()

splice() operates removing an element or elements from an original array with the option to replace them (the replacement parameters are not required if only removing). The return of this methods is the elements that were removed from the array. The parameters for splice are the index to which start removing/adding (below it is index 1), the second parameters is the number of elements desired for removal(below it is 2), then the optional third parameter is the replacement elements(below we have no arguments).

```
let letters = [ "a", "b", "c", "d"];
let removedLetters = letters.splice ( 1, 2 );
console.log(letters);   // "a, d"
console.log(removedLetters); //"b, c"
```

Array conversion to a string

toString(), join()

In programming conversion of an array to a string is a very useful tool, often utilised when wanting to display data held in an array. The below JavaScript methods both produce strings.

toString()

toString() operates by converting an array to a long string, with the automatic use of a comma as a separator

```
let letters = [ "a", "b", "c", "d"];
console.log(letters.toString()); //"a, b, c, d"
```

join()

join() operates by converting an array to a long string except it has the ability to use an alternate separator instead of a comma. If no alternate separator is specified a comma will be utilised automatically.

```

```

```
let letters = [ "a", "b", "c", "d"];
console.log(letters.join()); //"a, b, c, d"
console.log(letters.join("*")); //"a * b * c * d"
```

Multiple Array Joining

concat()

concat() operates by processing the joining of two arrays and the return is the resulting new array. The method enables to original arrays to remain unchanged.

```
let letters = [ "a", "b", "c", "d"];
let numbers = [ "1", "2", "3", "4"];
let lettersAndNumbers = letters.concat(numbers)
console.log(lettersAndNumbers) //["a", "b", "c", "d","1", "2", "3", "4"]
```

Array Extraction

slice()

slice() operates by removing a specific section of an array and returning it as a new array. The method operates by specifying a start index and an up to index.

```
let letters = [ "a", "b", "c", "d"];
console.log(letters.slice(1,3)) // ["b, c"]
```

Powerful array methods allowing callbacks

map(), filter(). reduce(), every(), some()

These powerful are very useful in programming as they allow executing of a callback function on each item in array, however they all operate uniquely.

map() is a very open ended method is that is operates by returning a new array of elements which have had the callback executed on them.

filter() operates by returning a new array of elements in which the callback returned true.

reduce() operates by returning single output value due to a 'reducer' callback function being executed on each element in the array. The purpose of this method is to reduce the collection down to the desired specified value. As in the example below, the parameters are the callback(which is executed on each element except the first if no initial value is specified), accumulator takes the return value and as the name suggests accumulates a running value, current value is the particular element being processed in the first iteration, index is the index of the element being processed and is option and finally the array is the array reduce was called upon. Initial value is an optional value which can be utilised with the first argument.

```
  arr.reduce(callback( accumulator, currentValue[, index[, array]] ) {
    // return result from executing something for accumulator or
currentValue
  }[, initialValue]);
```

every() operates by returning a value of true if the callback returns true for all array elements.

some() operates by returning true if the callback returns true for at least one array element.

[Sources]---

1. Reference
2. Reference
3. Reference
4. Reference

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

An object in JavaScript is a data structure of key, value pairs which are a list of zero or more property names and corresponding property values, which are enclosed in curly braces. Objects are integral in JavaScript as they allow for encapsulation of data, functions and other objects into one accessible and manipulatable entity.

Creation:

Creation of an object can be done via an object literal, also called Object Initializer which is defining and initializing an object via directly stating. These literal objects are expressions, and results in a new object whenever the statement is executed. Below is an example:

```
// Initialize object literal with curly brackets
const objectLiteral = {};
const objectLiteral2 = {prop1: val1, prop2: val2};
```

The assigning of the object to the variable is not required, however is needed when the object is to be referred to elsewhere. As per the first example an object can be initialized empty.

Creation of an object can also be done via an object constructor which is creation of an object via the new keyword. Below are examples:

```
// Initialize object constructor with new Object
const objectConstructor = new Object();
const objectConstructor2 = new Object();
objectConstructor2.prop1 = “val1”;
objectConstructor2.prop2 = “val2”;
```

In these examples, an object can constructor can initialize an empty object, then you are able to populate the object via dot notation. The object constructor is also useful in creating a constructor function to be able to create multiple objects with the same properties without needed to literally state them all.

Accessing:

Accessing the properties of objects can be done via dot notation in which the object name is entered first, then a dot, then the property requiring access. Similarly, when accessing a function within an object brackets are needed to call the function.

```
const dotObject = {prop1: “val1”, prop2: “val2”, ”, prop3: function() {
return “hello there”};
dotObject.prop1 //=> “val1”
dotObject.prop3() //=> “hello there”
```

Accessing an object is also possible via bracket notation which is similar to array accessibility. However, in this situation, the property is used to select a value. This method is useful for dynamic accessibility, accessing property names which are alternative to standard name creation, and using an object as a dictionary.

```
const bracketObject = {prop1: “val1”, prop2: “val2”};
bracketObject [prop1] //=> “val1”
```

Changing:

Using object accessibility, object values are able to be changed/overwritten.

```
const changeObject = {prop1: “val1”, prop2: “val2”};
changeObject.prop1 = “newval1”
```

Adding:

Using object accessibility, object values are able to be added.

```
const changeObject = {prop1: “val1”, prop2: “val2”};
changeObject.prop3 = “val3”
```

Deleting:

Deleting from an object is done via utilising the delete keyword. Delete returns true is the removal was successful.

```
const delObject = {prop1: “val1”, prop2: “val2”};
delete delObject.prop1 //=> true
console.log(delObject) //=> {prop2: “val2”};
```

Iteration:

Using a specific for loop in JavaScript objects are able to be iterated through without knowing particular property names. The below example shows this via counting all the keys in the example object.

```
const iterationObject = {prop1: “val1”, prop2: “val2”, prop3: “val3”};

let countKeys = 0
for(let key in iterationObject) {
    countKeys++;
};
console.log(countKeys) //=> 3
```

for...in loop is a useful method however it returns based off the original protoype chain. In order to iterate on an object so that all JavaScript methods can be utilised on the object other enumerating iteration methods that will not affect the original object are useful. Object.entries() is a method that can be applied on an object that will return an array of key,value pairs. Similarly, Object.keys() returns a new array of the object’s keys.

```
const iterationObject2 = {prop1: “val1”, prop2: “val2”, prop3: “val3”};
Object.keys(iterationObject2)
//=> [“prop1”, “prop2”, “prop3”]
```

Copying:

To deep copy an object using JSON.stringify and JSON.parse. This works by making the original object into a new string, then transforming it into an object again and assigning it to a variable. Therefore becoming a new object. This process however only works on plain objects, meaning the object cannot contain other code like functions.

```
const copyObject = {prop1: “val1”, prop2: “val2”, prop3: “val3”};
const newCopyObject = JSON.parse(JSON.stringify(copyObject))
```

[Sources]---

1. Reference

2. Reference
3. Reference
4. Reference

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

JSON (JavaScript Object Notation) is a format for data storage and transportation that is lightweight, easily understandable as it uses human-readable text consisting of key-value pairs that can include JavaScript data types such as strings, numbers, arrays, Booleans and object literals. Some key notes about the JSON object is the syntax requires name/value pairs to be double quoted, with correct comma's, within curly brackets for objects and arrays must be in square brackets. Null is allowed in JSON however NaN and infinity are not supported, JSON strings can be empty, but undefined is not permitted, and irrelevant whitespace is permitted anywhere except within a number or a string. Whilst it resembles JavaScript object literal syntax it is a very common data format often utilised diversely amongst applications that include code that can generate and parse JSON. Commonly utilised for transmitting data across a network from web server to webpage. In this process JSON exists as a string, that requires conversion to a JavaScript object for data accessibility, this act is called deserialization. This static method is:

```
JSON.parse(text,[reviver])

let jsonObj = '{"foo": 1, "bar": 2, "abc": "abc" }'
JSON.parse(jsonObj)
console.log(jsonObj)//  {"foo": 1, "bar": 2, "abc": "abc" }
```

This method enables the JSON string text to be parsed/transformed into a JavaScript object and the value gets returned. Once parsed the object returned will obtained all the object methods inbuilt for JavaScript.

In order to then send JSON objects over a network they correspondingly have to be transformed to a JSON string text. This static method of serialization is:

```
JSON.stringify (value[, replacer[, space]])

let obj = {"foo": 1, "bar": 2, "abc": "abc" }
JSON.stringify(obj)
console.log(obj)//  '{"foo": 1, "bar": 2, "abc": "abc" }'
```

This method returns a JSON string which consists of the given parameter object value. There are optional parameters that can be used for specifying only certain properties to be stringified and/or replacing certain property values

[Sources]---

1. Reference
2. Reference
3. Reference
4. Reference

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.

```
class Car { //Creating a JavaScript Class called Car which is like
Template
    constructor(brand) { //Setting the class constructor method which is
used for intialisation
      this.carname = brand; //One intial property is set which is Car
name.
    }
    present() { //This is a object method called present
      return 'I have a ' + this.carname; //It returns a String that has
the class Car Name interpolated
    }
  }

  class Model extends Car { //Creating a Class which is a Template, this
is a Model Class that inherits methods from the Car Class
    constructor(brand, mod) { //Setting the constructor which intialises
the properties
      super(brand); //super refers to parent class, using super means the
Car constructor gets called and gets access to the Car properties and
methods.
      this.model = mod; //A inital property is set which is the Model
    }
    show() { //This is an object method called Show
      return this.present() + ', it was made in ' + this.model; //It
returns Car Object Method which is Present, connecting another string with
interpolation the Model.
    }
  }

  let makes = ["Ford", "Holden", "Toyota"] //Setting a new array called
makes with 3 properties
  let models = Array.from(new Array(40), (x,i) => i + 1980) //Setting an
new array using the static method that shallow copies an array from a new
Array which is set with 40 elements, then a map function is called on each
element adding 1980 so that there is 40 years in the array from 1980-2019

  function randomIntFromInterval(min,max) { // min and max included
      return Math.floor(Math.random()*(max-min+1)+min); //Function with
two arguments that returns a random number
```

```
  }
  // gives us a random number between 0 and 1 .0 but never quite return a
1
  //We dont want decimal we want random number
  //To do this we multiple Math.random by the range, which is set with the
min and max arguments
  //To make the random number start at the min value we need to add min to
whatever number we got.
  //Math.floor takes off the decimal

  for (model of models) {  //for loop that loops over all the models in
the models array and executes the below

    make = makes[randomIntFromInterval(0,makes.length-1)] //sets each make
to be accessing the makes array with an index chosen randomly with the
random number function with the min being 0 and max being the array length
take 1
    model = models[randomIntFromInterval(0,makes.length-1)]  //sets each
model to be accessing the models array with an index chosen randomly with
the random number function with the min being 0 and max being the makes
array(this is an error I think, it should be models, so a random element
selected is only ever 1980-1982) length take 1

    mycar = new Model(make, model); //setting mycar to be a new Model
object with the parameters of make and model set into the object using
above make and model set
    console.log(mycar.show()) //printing in the console is the show method
on the mycar object which prints the statement returned in the show method
  }
  //the for loop, loops over the 40 models in the models array and that is
what is printed in the console. The main error being the getting random
models is called on the makes array length. Therefore the years 1983 and
above do not get randomly chosen.
```