LEARN BASIC STRING AND ARRAY METHODS BY BUILDING A MUSIC PLAYER

Introduction:

Now let's learn some essential string and array methods like the `find()`, `forEach()`, `map()`, and `join()`. These methods are crucial for developing dynamic web applications.

In this project, you'll code a basic MP3 player using HTML, CSS, and JavaScript. The project covers fundamental concepts such as handling audio playback, managing a playlist, implementing play, pause, next, previous, and shuffle functionalities. You'll even learn how to dynamically update your user interface based on the current song.

Step 1:

In this project you will learn basic string and array methods by building a music player app. You will be able to play, pause, skip, and shuffle songs.

The HTML and CSS of this project have been provided for you, so you can focus on JavaScript.

Start by accessing the `#playlist-songs`, `#play`, and `#pause` elements with the `getElementById()` method. Assign them to variables `playlistSongs`, `playButton` and `pauseButton` respectively.

Step 2:

Access the `#next`, `#previous` and `#shuffle` elements from the HTML file.

Assign them to variables named `nextButton`, `previousButton`, and `shuffleButton`, respectively.

Step 3:

Next, create an array to store all the songs. Create an empty allSongs array.

Step 4:

Inside the allSongs array, create an object with the following properties and values:

Example Code:

id: 0,

title: "Scratching The Surface",

artist: "Quincy Larson",

duration: "4:25",

src:"https://cdn.freecodecamp.org/curriculum/js-music-player/scratching-the-surface.mp3",

Step 5:

Add a second object with the following keys and values:

Example Code:

id: 1,

title: "Can't Stay Down",

artist: "Quincy Larson",

duration: "4:15",

src:
"https://cdn.freecodecamp.org/curriculum/js-music-player/can't-stay-down.mp3",

Step 6:

Add a third object with the following properties and values:

Example Code:

id: 2,

title: "Still Learning",

artist: "Quincy Larson",

duration: "3:51",

src:
"https://cdn.freecodecamp.org/curriculum/js-music-player/still-learning.mp3",

Step 7:

We've added the rest of the songs to the `allSongs` array for you.

Next, you'll learn about the Web Audio API and how to use it to play songs. All modern browsers support the Web Audio API, which lets you generate and process audio in web applications.

Use `const` to create a variable named `audio` and set it equal to `new Audio()`. This will create a new HTML5 `audio` element.

Step 8:

Your music player should keep track of the songs, the current song playing, and the time of the current song. To do this, you will need to create an object to store this information.

Start by using the `let` keyword to declare a new variable called `userData` and assign it an empty object.

Step 9:

Since users will be able to shuffle and delete songs from the playlist, you will need to create a copy of the `allSongs` array without mutating the original. This is where the `spread` operator comes in handy.

The spread operator (`...`) allows you to copy all elements from one array into another. It can also be used to concatenate multiple arrays into one. In the example below, both `arr1` and `arr2` have been spread into `combinedArr`:

Example Code:

```
const arr1 = [1, 2, 3];

const arr2 = [4, 5, 6];

const combinedArr = [...arr1, ...arr2];

console.log(combinedArr); // Output: [1, 2, 3, 4, 5, 6]
```

Inside the `userData` object create a `songs` property. For the value, spread `allSongs` into an array.

Step 10:

To handle the current song's information and track its playback time, create a `currentSong` and `songCurrentTime` properties. Set the values to `null` and `0` respectively.

Step 11:

In the previous projects, you used regular functions. But in the rest of the projects, you will be working with arrow functions. The next few steps will focus on teaching you the basics of arrow functions.

An arrow function is an anonymous function expression and a shorter way to write functions. Anonymous means that the function does not have a name. Arrow functions are always anonymous.

Here is the basic syntax:

Example Code:

```
() => {}
```

To create a named arrow function, you can assign the function to a variable:

Example Code:

```
const exampleFunction = () => {

  // code goes here

}
```

Create a new named arrow function called `printGreeting`. Inside the body of that function use the `console.log()` method to print the string `"Hello there!"`.

Step 12:

To call a named arrow function expression, you can reference the function by its name.

Example Code:

```
exampleArrowFunction();
```

Below your printGreeting function definition, call the function. Open up the console to see the output.

Step 13:

Just like regular functions, arrow functions can accept multiple parameters.

Here is an example of a named arrow function with one parameter:

Example Code:

```
const greet = (name) => {

  console.log(`Hello, ${name}!`);

};
```

If the function only has one parameter, you can omit the parentheses around the parameter list like this:

Example Code:

```
const greet = name => {

  console.log(`Hello, ${name}!`);

};
```

Create a new named arrow function called printMessage that has one parameter called org. Inside the body of that function, add a console statement. Inside that console statement, add the template literal ${org} is awesome!.

Below your printMessage function, call the function and pass in the string "freeCodeCamp" as an argument.

Open up the console to see the result.

Step 14:

Just like regular functions, arrow functions can return values.

Here is an example of an arrow function returning the result of multiplying two numbers:

Example Code:

```
const multiplyTwoNumbers = (num1, num2) => {

  return num1 * num2;

}

// Output: 12

console.log(multiplyTwoNumbers(3, 4));
```

Create a new variable called addTwoNumbers and assign it an arrow function. This arrow function should take two parameters called num1 and num2. Inside the body of the function, return the expression of adding num1 and num2.

Below the addTwoNumbers function, add a console statement. Inside that console statement, call the addTwoNumbers function and pass in the numbers 3 and 4 as arguments.

Open up the console to see the output.

Step 15:

If the arrow function is returning a simple expression, you can omit the `return` keyword and the curly braces `{}`. This is called an implicit return.

Example Code:

```
const multiplyTwoNumbers = (num1, num2) => num1 * num2;
```

If your arrow function has multiple lines of code in the function body, then you need to use the `return` keyword and the curly braces `{}`.

Example Code:

```
const getTax = (price) => {

  const taxRate = 0.08;

  const tax = price * taxRate;

  return tax;

};
```

Refactor, or update, your `addTwoNumbers` function to remove the `return` keyword and the curly braces `{}`. Your `addTwoNumbers` function should instead use an implicit return.

Open up the console to make sure that you are still getting the correct output.

Step 16:

Now it is time to apply what you have learned about arrow functions to your music player project.

Start by removing the printGreeting, printMessage, and addTwoNumbers functions. Also, remove all of the console statements and function calls.

Step 17:

To display the songs in the UI (User Interface), you'll need to create a function.

Use the arrow function syntax to create a renderSongs function that takes in array as its parameter.

Step 18:

When the songs are displayed on the page, it should show the title, artist, duration of each song and a delete button.

In the next few steps, you will learn how to add new HTML for each song using the map() method. This method will be explained in more detail in the next step.

Start by using const to declare a variable named songsHTML and assign it array.map().

Step 19:

The map() method is used to iterate through an array and return a new array. It's helpful when you want to create a new array based on the values of an existing array. For example:

Example Code:

```
const numbers = [1, 2, 3];

const doubledNumbers = numbers.map((number) => number * 2); //
doubledNumbers will be [2, 4, 6]
```

Notice that the `map()` method takes a function as an argument. This is called a callback function, which is a function that is passed to another function as an argument. In the example above, the callback function is `(number) => number * 2`, and it's run on each element in the `numbers` array. The `map()` method then returns a new array with the results.

Pass in a callback function to the `map()` method. The callback function should take `song` as a parameter, use the arrow function syntax, and have an empty body.

Step 20:

Inside the `map()`, add a `return` statement with backticks where you will interpolate all the elements responsible to displaying the song details.

Inside the backticks, create an `li` element with an `id` attribute of `song-${song.id}` and a `class` attribute of `playlist-song`.

Step 21:

Create a `button` element with class `playlist-song-info`. Inside the `button`, add a `span` element with the class `playlist-song-title`, then interpolate `song.title` as the text.

Step 22:

Inside the `button` element, create two more `span` elements.

The first `span` element should have a `class` of `playlist-song-artist`. In between the `span` tags, add `${song.artist}`.

The second span element should have a class of playlist-song-duration. In between the span tags, add ${song.duration}.

Step 23:

Create another button element with the class playlist-song-delete and the aria-label attribute set to Delete interpolated with song.title. For the content of the delete icon, paste in the following SVG:

Example Code:

```
<svg width="20" height="20" viewBox="0 0 16 16" fill="none"
xmlns="http://www.w3.org/2000/svg"><circle cx="8" cy="8" r="8"
fill="#4d4d62"/><path fill-rule="evenodd" clip-rule="evenodd"
d="M5.32587 5.18571C5.7107 4.90301 6.28333 4.94814 6.60485 5.28651L8
6.75478L9.39515 5.28651C9.71667 4.94814 10.2893 4.90301 10.6741
5.18571C11.059 5.4684 11.1103 5.97188 10.7888 6.31026L9.1832
7.99999L10.7888 9.68974C11.1103 10.0281 11.059 10.5316 10.6741
10.8143C10.2893 11.097 9.71667 11.0519 9.39515 10.7135L8
9.24521L6.60485 10.7135C6.28333 11.0519 5.7107 11.097 5.32587
10.8143C4.94102 10.5316 4.88969 10.0281 5.21121 9.68974L6.8168
7.99999L5.21122 6.31026C4.8897 5.97188 4.94102 5.4684 5.32587
5.18571Z" fill="white"/></svg>
```

Step 24:

Right now the songsHTML is an array. If you tried to display this as is, you would see the songs separated by commas. This is not the desired outcome because you want to display the songs as a list. To fix this, you will need to join the array into a single string by using the join() method.

The join() method is used to concatenate all the elements of an array into a single string. It takes an optional parameter called a separator which is used to separate each element of the array. For example:

Example Code:

```
const exampleArr = ["This", "is", "a", "sentence"];

const sentence = exampleArr.join(" "); // Separator takes a space
character

console.log(sentence); // Output: "This is a sentence"
```

Chain the `join()` method to your `map()` method and pass in an empty string for the separator.

To chain multiple methods together, you can call the `join()` method on the result of the `map()` method. For example:

Example Code:

```
array.map().join();
```

Step 25:

Next, you will need to update the playlist in your HTML document to display the songs.

Assign `songsHTML` to the `innerHTML` property of the `playlistSongs` element. This will insert the `li` element you just created into the `ul` element in the already provided `HTML` file.

Step 26:

Now you need to call the `renderSongs` function and pass in `userData?.songs` in order to finally display the songs in the UI.

Optional chaining (`?.`) helps prevent errors when accessing nested properties that might be null or undefined. For example:

Example Code:

```
const user = {

  name: "Quincy",

  address: {

    city: "San Francisco",

    state: "CA",

    country: "USA",

  },

};


// Accessing nested properties without optional chaining

const state = user.address.state; // CA


// Accessing a non-existent nested property with optional chaining

const zipCode = user.address?.zipCode; // Returns undefined instead of
throwing an error
```

Step 27:

Now that you have the list of songs displayed on the screen, it would be nice to sort them in alphabetical order by title.

Start by creating an arrow function called `sortSongs`.

Step 28:

Now that you have the list of songs displayed on the screen, it would be nice to sort them in alphabetical order by title. You could manually update the allSongs array, but JavaScript has an array method you can use called sort().

The sort() method converts elements of an array into strings and sorts them in place based on their values in the UTF-16 encoding.

Example Code:

```
const names = ["Tom", "Jessica", "Quincy", "Naomi"];

names.sort() // ["Jessica", "Naomi", "Quincy", "Tom"]
```

Inside your sortSongs function, add the sort() method to userData?.songs.

Step 29:

To sort the songs in alphabetical order by title, you will need to pass in a compare callback function into your sort() method.

Here is an example of sorting a list of fruits by name.

Example Code:

```
const fruits = [

  { name: "Apples", price: 0.99 },

  { name: "Blueberries", price: 1.49 },

  { name: "Grapes", price: 2.99 },

];

fruits.sort((a, b) => {

  if (a.name < b.name) {

    return -1;
  }

  if (a.name > b.name) {

    return 1;

  }

  return 0;

});
```

In the next few steps, you will learn what each of those `if` statements is doing inside that callback function. But for now, add an empty callback function to your `sort()` method and use `a` and `b` for the parameter names.

Step 30:

The `sort()` method accepts a compare callback function that defines the sort order.

In this example, the first condition `(a.name < b.name)` checks if the name of the first fruit is less than the name of the second fruit. If so, the first fruit is sorted before the second fruit.

Strings are compared lexicographically which means they are compared character by character. For example, `"Apples"` is less than `"Bananas"` because `"A"` comes before `"B"` in the alphabet.

The reason why this example is returning numbers is because the `sort()` method is expecting a number to be returned. If you return a negative number, the first item is sorted before the second item.

Example Code:

```
const fruits = [

  { name: "Apples", price: 0.99 },
  { name: "Blueberries", price: 1.49 },

  { name: "Grapes", price: 2.99 },

];

fruits.sort((a, b) => {

  if (a.name < b.name) {

    return -1;

  }

  if (a.name > b.name) {

    return 1;

  }

  return 0;

});
```

Inside your callback function, add an `if` statement to check if `a.title` is less than `b.title`. If so, return `-1`.

Step 31:

The second condition in this example checks if `a.name > b.name`. If so, the function returns `1`, which sorts the first fruit after the second fruit.

Example Code:

```
const fruits = [

  { name: "Apples", price: 0.99 },

  { name: "Blueberries", price: 1.49 },
  { name: "Grapes", price: 2.99 },
];


fruits.sort((a, b) => {
  if (a.name < b.name) {
    return -1;
  }
  if (a.name > b.name) {
    return 1;
  }
  return 0;
});
```

Inside your callback function, add another `if` statement to check if `a.title` is greater than `b.title`. If so, return the number `1`.

Step 32:

In the example, if `a.name` is equal to `b.name`, then the function returns `0`. This means that nothing changes and the order of `a` and `b` remains the same.

Example Code:

```
const fruits = [
  { name: "Apples", price: 0.99 },
  { name: "Blueberries", price: 1.49 },
  { name: "Grapes", price: 2.99 },
];

fruits.sort((a, b) => {
  if (a.name < b.name) {
    return -1;
  }
  if (a.name > b.name) {
    return 1;
  }
  return 0;
});
```

Below your `if` statements, return the number `0` to leave the order of the two elements unchanged.

Step 33:

The last step for the `sortSongs` function is to return `userData?.songs`.

Step 34:

Right now the song order has not changed. That is because the updates you made using the `sort` method will not happen until the `sortSongs` function is called.

Change your `renderSongs` function to call the `sortSongs` function.

Now you should see the songs in alphabetical order.

Step 35:

It's time to begin implementing the functionality for playing the displayed songs.

Define a playSong function using const. The function should take an id parameter which will represent the unique identifier of the song you want to play.

Step 36:

The find() method retrieves the first element within an array that fulfills the conditions specified in the provided callback function. If no element satisfies the condition, the method returns undefined.

In the example below, the find() method is used to find the first number greater than 25:

Example Code:

```
const numbers = [10, 20, 30, 40, 50];


// Find the first number greater than 25

const foundNumber = numbers.find((number) => number > 25);

console.log(foundNumber); // Output: 30
```

Use const to create a variable named song and assign it result of the find() method call on the userData?.songs array. Use song as the parameter of the find() callback and check if song.id is strictly equal to id.

This will iterate through the `userData?.songs` array, searching for a song that corresponds to the `id` passed into the `playSong` function.

Step 37:

Inside the `playSong` function, set the `audio.src` property equal to `song.src`. This tells the audio element where to find the audio data for the selected song.

Also, set the `audio.title` property equal to `song.title`. This tells the audio element what to display as the title of the song.

Step 38:

Before playing the song, you need to make sure it starts from the beginning. This can be achieved by the use of the `currentTime` property on the `audio` object.

Add an `if` statement to check whether the `userData?.currentSong` is falsy *OR* if `userData?.currentSong.id` is strictly not equal `song.id`. This condition will check if no current song is playing or if the current song is different from the one that is about to be played.

Inside the `if` block, set the `currentTime` property of the `audio` object to `0`.

Step 39:

Add an `else` block to handle the song's current playback time. This allows you to resume the current song at the point where it was paused.

Within the `else` block, set the `currentTime` property of the `audio` object to the value stored in `userData?.songCurrentTime`.

Step 40:

You need to update the current song being played as well as the appearance of the `playButton` element.

Assign `song` to the `currentSong` property on the `userData` object.

Note: You should not use the optional chaining operator `?.` in this step because `userData.currentSong` will not be `null` or `undefined` at this point.

Step 41:

Next, use the `classList` property and the `add()` method to add the `"playing"` class to the `playButton` element. This will look for the class `"playing"` in the CSS file and add it to the `playButton` element.

To finally play the song, use the `play()` method on the `audio` variable. `play()` is a method from the web audio API for playing an mp3 file.

Step 42:

In previous steps you built out the functionality for playing a song. Now you need to add the functionality to the play button so that it will play the current song when it is clicked on.

Use the `addEventListener()` method and pass in a `"click"` event for the first argument and an empty callback function with arrow syntax for the second argument, e.g., `() => {}`.

Step 43:

Within the arrow function of the event listener, add an `if` to check if `userData?.currentSong` is `null`.

Inside the `if` block, call the `playSong()` function with the `id` of the first song in the `userData?.songs` array. This will ensure the first song in the playlist is played first.

Step 44:

Add an `else` block. Inside the `else` block, call the `playSong` function with the `id` of the currently playing song as an argument.

This ensures that the currently playing song will continue to play when the play button is clicked.

Step 45:

To play the song anytime the user clicks on it, add an `onclick` attribute to the first button element. Inside the `onclick`, call the `playSong` function with `song.id`.

Don't forget you need to interpolate with the dollar sign here.

Step 46:

Now you need to work on pausing the currently playing song.

Define a `pauseSong` function using the `const` keyword and arrow function syntax. The function should take no parameters.

Step 47:

To store the current time of the song when it is paused, set the `songCurrentTime` of the `userData` object to the `currentTime` of the `audio` variable.

Note: You should not use optional chaining for this step because `userData.songCurrentTime` will not be `null` or `undefined` at this point.

Step 48:

Use `classList` and `remove()` method to remove the `playing` class from the `playButton`, since the song will be paused at this point.

To finally pause the song, use the `pause()` method on the `audio` variable. `pause()` is a method of the Web Audio API for pausing music files.

Step 49:

Now it is time to test out the pause button.

Add a `"click"` event listener to the `pauseButton` element, then pass in `pauseSong` as the second argument of the event listener. This is the function the event listener will run.

Test out your app by first clicking on the play button followed by the pause button. You should see that everything is working as expected.

Step 50:

Before you start working on playing the next and previous song, you need to get the index of each song in the `songs` property of `userData`.

Start by creating an arrow function called `getCurrentSongIndex`.

Step 51:

To get the index for the current song, you can use the indexOf() method. The `indexOf()` array method returns the first index at which a given element can be found in the array, or -1 if the element is not present.

Example Code:

```
const animals = ["dog", "cat", "horse"];

animals.indexOf("cat") // 1
```

Inside your getCurrentSongIndex function, return userData?.songs.indexOf(). For the indexOf() argument, set it to userData?.currentSong.

Step 52:

You need to work on playing the next song and the previous song. For this, you will need a playNextSong and playPreviousSong function.

Use const and arrow syntax to create an empty playNextSong function.

Step 53:

Inside the playNextSong function, create an if statement to check if the currentSong of userData is strictly equal to null. This will check if there's no current song playing in the userData object.

If the condition is true, call the playSong function with the id of the first song in the userData?.songs array as an argument.

Step 54:

Add an else block to the if statement. Inside the else block, call the getCurrentSongIndex() function and assign it to a constant named currentSongIndex.

Step 55:

Next, you will need to retrieve the next song in the playlist. For that, you will need to get the index of the current song and then add `1` to it.

Create a constant called `nextSong` and assign `userData?.songs[currentSongIndex + 1]` to it.

Lastly, call the `playSong` function and pass in `nextSong.id` as the argument.

Step 56:

Now it is time to test out the `playNextSong` function.

Add a `"click"` event listener to the `nextButton` element, then pass in `playNextSong` as the second argument of your event listener. This is the function the event listener will run.

Test out your app by first clicking on the play button followed by the next button. You should see that everything is working as expected.

Step 57:

Use `const` and arrow syntax to create an empty `playPreviousSong` function.

Step 58:

Within the `playPreviousSong` function, add an `if` statement with a condition of `userData?.currentSong === null`. This will check if there is currently no song playing. If there isn't any, exit the function using a `return`.

Inside the `else` block, create a constant named `currentSongIndex` and assign it `getCurrentSongIndex()`.

Step 59:

To get the previous song, subtract `1` from the `currentSongIndex` of `userData?.songs` and assign it to the constant `previousSong`. After that, call the `playSong` function and pass `previousSong.id` as an argument.

Step 60:

Add a `"click"` event listener to the `previousButton` element, then pass in `playPreviousSong` as the second argument.

Step 61:

If you check closely, you'd see the currently playing song is not highlighted in the playlist, so you don't really know which song is playing. You can fix this by creating a function to highlight any song that is being played.

Using an arrow syntax, create a `highlightCurrentSong` function. Inside the function, use `querySelectorAll` to get the `.playlist-song` element and assign to a `playlistSongElements` constant.

Step 62:

You need to get the `id` of the currently playing song. For this, you can use `userData?.currentSong?.id`.

Use `getElementById()` to get the `id` of the currently playing song, then use template literals to prefix it with `song-`. Assign it to the constant `songToHighlight`.

Step 63:

Loop through the `playlistSongElements` with a `forEach` method.

The `forEach` method is used to loop through an array and perform a function on each element of the array. For example, suppose you have an array of numbers and you want to log each number to the console.

Example Code:

```
const numbers = [1, 2, 3, 4, 5];

// Using forEach to iterate through the array

numbers.forEach((number) => {

  console.log(number); // 1, 2, 3, 4, 5

});
```

Use the `forEach` method on `playlistSongElements`. Pass in `songEl` as the parameter and use arrow syntax to add in an empty callback.

Step 64:

Within the callback function, use the `removeAttribute()` method to remove the `"aria-current"` attribute. This will remove the attribute for each of the songs.

Step 65:

Now you need to add the attribute back to the currently playing song.

Create an `if` statement with the condition `songToHighlight`. For the statement, use `setAttribute` on `songToHighlight` to pass in `"aria-current"` and `"true"` as the first and second arguments.

Step 66:

Inside the `playSong` function, call the `highlightCurrentSong` function.

After that, play around with the control buttons to see how the `highlightCurrentSong` function works.

Step 67:

Next, you need to display the current song title and artist in the player display. Use `const` and arrow syntax to create an empty `setPlayerDisplay` function.

Step 68:

Inside the function, obtain references to the HTML elements responsible for displaying the song title and artist.

Access the `#player-song-title` and `#player-song-artist` elements with the `getElementById()` method. Assign them to variables `playingSong` and `songArtist` respectively.

Step 69:

Access the `userData?.currentSong?.title` and `userData?.currentSong?.artist` properties and assign them to a `currentTitle` and `currentArtist` variables respectively.

Step 70:

`textContent` sets the text of a node and allows you to set or retrieve the text content of an HTML element.

Example Code:

```
<div id="example">This is some text content</div>
```

Example Code:

```
const element = document.getElementById('example');

console.log(element.textContent); // Output: This is some text content
```

You can use a ternary operator to conditionally set the text content value. Here is an example of assigning the result of a ternary operator to a variable:

Example Code:

```
const example = condition ? "I'm true" : "I'm false";
```

Use a ternary operator to check if `currentTitle` evaluates to a truthy value. If it does, set `playingSong.textContent` to `currentTitle`. Otherwise, set it to an empty string.

Then below that, use a `ternary` operator to check if `currentArtist` is truthy. If so, set `songArtist.textContent` to `currentArtist`. Otherwise, set it to empty string.

Step 71:

To ensure the player's display updates whenever a new song begins playing, call the `setPlayerDisplay()` function within the `playSong()` function.

Now you should see the song title and the artist show up in the display.

Step 72:

To make the application more accessible, it is important that the play button describes the current song or the first song in the playlist.

Start by creating an empty arrow function called `setPlayButtonAccessibleText`.

Step 73:

You need to get the currently playing song or the first song in the playlist. To do that, create a `song` constant and use the OR operator (`||`) to set it to the current song of `userData`, **or** the first song in the `userData?.songs` array.

Don't forget to use optional chaining.

Step 74:

The `setPlayButtonAccessibleText` function will set the `aria-label` attribute to the current song, or to the first song in the playlist. And if the playlist is empty, it sets the `aria-label` to `"Play"`.

Use the `setAttribute` method on the `playButton` element to set an attribute named `"aria-label"`. Using a ternary, set the attribute value to `Play ${song.title}` or `"Play"` if `song?.title` is not available.

Don't forget you need string interpolation here, so you need to use backticks.

Step 75:

Now, call the `setPlayButtonAccessibleText` function inside the `playSong` function.

Step 76:

Using `const` and arrow syntax to create an empty function called `shuffle`.

This function is responsible for shuffling the songs in the playlist and performing necessary state management updates after the shuffling.

Step 77:

In earlier steps, you learned how to work with the `sort()` method to sort the songs in alphabetical order. Another use case for the callback function is to randomize an array.

One way to randomize an array of items would be to subtract `0.5` from `Math.random()` which produces random values that are either positive or negative. This makes the comparison result a mix of positive and negative values, leading to a random ordering of elements.

Example Code:

```
const names = ["Tom", "Jessica", "Quincy", "Naomi"];

names.sort(() => Math.random() - 0.5);
```

Use the `sort()` method on the `userData?.songs` array. Pass a callback to the method, and return the result of `Math.random() - 0.5`.

Step 78:

When the shuffle button is pressed, you want to set the `currentSong` to nothing and the `songCurrentTime` to `0`.

Set `userData.currentSong` to `null` and `userData.songCurrentTime` to `0`.

Note: You should not use optional chaining for this step because you are explicitly setting the `currentSong` and `songCurrentTime` properties to be `null` and `0` respectively.

Step 79:

You should also re-render the songs, pause the currently playing song, set the player display, and set the play button accessible text again.

Call the `renderSongs` function and pass in `userData?.songs` as an argument. Also, call the `pauseSong`, `setPlayerDisplay`, and `setPlayButtonAccessibleText` functions.

Step 80:

Add a `"click"` event listener to the `shuffleButton` element. For the function to run, pass in the `shuffle` function.

Note: You don't need a callback inside this particular event listener. You also don't need to call the `shuffle` function, just pass in its identifier.

Step 81:

It's time to implement a delete functionality for the playlist. This would manage the removal of a song from the playlist, handle other related actions when a song is deleted, and create a `Reset Playlist` button.

Use `const` and arrow syntax to create an empty `deleteSong` function and pass in `id` as a parameter.

Step 82:

Use the filter() method to remove the song object that matches the id parameter from the userData?.songs array.

The filter method keeps only the elements of an array that satisfy the callback function passed to it:

Example Code:

```
const numArr = [1, 10, 8, 3, 4, 5]

const numsGreaterThanThree = numArr.filter((num) => num > 3);

console.log(numsGreaterThanThree) // Output: [10, 8, 4, 5]
```

Use the filter() method on userData?.songs. Pass in song as the parameter of the arrow function callback and use implicit return to check if song.id is strictly not equal to id. Assign all of that to the userData.songs.

Note: You should not use optional chaining when you assign the result of userData?.songs.filter to userData.songs because the allSongs array will not be undefined or null at that point.

Step 83:

You need to re-render the songs, highlight it and set the play button's accessible text since the song list will change.

Call the renderSongs function and pass in the userData?.songs array as an argument, this displays the modified playlist.

After that, call the highlightCurrentSong function to highlight the current song if there is any also and the setPlayButtonAccessibleText function to update the play button's accessible text.

Step 84:

Before deleting a song, you need to check if the song is currently playing. If it is, you need to pause the song and play the next song in the playlist.

Use an `if` statement to check if the `userData?.currentSong?.id` is equal to the `id` of the song you want to delete.

Step 85:

If there is a match then set `userData.currentSong` to `null` and `userData.songCurrentTime` to `0`.

After that, call the `pauseSong()` function to stop the playback and the `setPlayerDisplay()` function to update the player display.

Step 86:

Within the button element in the `renderSongs` function, add an `onclick` attribute. For the value, call the `deleteSong` function and interpolate `song.id`.

Step 87:

Next, you need to check if the playlist is empty. If it is, you should reset the `userData` object to its original state.

Use an `if` statement to check if the `userData?.songs` has a length of `0`.

Step 88:

If the playlist is empty, you need to create a `resetButton` element and a text for it. This button will only show up if the playlist is empty.

createElement() is a DOM method you can use to dynamically create an element using JavaScript. To use createElement(), you call it, then pass in the tag name as a string:

Example Code:

```
// syntax

document.createElement(tagName)

// example

document.createElement('div')
```

You can also assign it to a variable:

Example Code:

```
const divElement = document.createElement('div')
```

Inside your if statement, declare a resetButton constant, then use createElement() to create a "button".

Step 89:

Now that you've created the button, you need to assign it a text. To do this, you need to use the createTextNode() method of DOM.

The createTextNode() method is used to create a text node. To use it, you call it and pass in the text as a string:

Example Code:

```
document.createTextNode("your text")
```

You can also assign it to a variable:

Example Code:

```
const myText = document.createTextNode("your text")
```

Use the `createTextNode()` method to create a `"Reset Playlist"` text, then assign it to a `resetText` constant.

Step 90:

Now that you've created the `resetButton`, you need to assign it an `id` and `aria-label` attributes. JavaScript provides the `id` and `ariaLabel` properties you need to use for this.

For example, `element.id` would set an `id` attribute, and `element.ariaLabel` would set an `aria-label` attribute. Both of them accept their values as a string.

Set the `id` attribute of the `resetButton` element to `"reset"` and its `aria-label` attribute to `"Reset playlist"`.

Step 91:

You need to add the `resetText` to the `resetButton` element as a child, and also the `resetButton` to the `playlistSongs` element as a child. For this, there is an `appendChild()` method to use.

`appendChild()` lets you add a node or an element as the child of another element. In the example below, the text "Click me" would be attached to the button:

Example Code:

```
const parentElement = document.createElement("button")

const parentElementText = document.createTextNode("Click me")

// attach the text "Click me" to the button

parentElement.appendChild(parentElementText)
```

Use `appendChild()` to attach `resetText` to `resetButton` element, and `resetButton` to the `playlistSongs` element.

Step 92:

Now, it's time to add the reset functionality to the `resetButton`. This will bring back the songs in the playlist when clicked.

Add a click event listener to the `resetButton` variable. Pass in a callback using arrow syntax and leave it empty for now.

Step 93:

To reset the playlist to its original state, spread `allSongs` into an array and assign it to `userData.songs`.

Note: You should not use optional chaining for the `userData.songs` because the song will not be `null` or `undefined` at this point.

Step 94:

Finally, you should render the songs again, update the play button's accessible text, and remove the reset button from the playlist. You also need to remove the `resetButton` from the DOM.

Call the `renderSongs()` function with `sortSongs()` as an argument to render the songs again in alphabetical order.

Call the `setPlayButtonAccessibleText()` function to update the play button's accessible text.

Remove the reset button from the playlist by calling the `remove()` method on the `resetButton` variable.

Note: Now you can try removing all the songs to see what happens.

Step 95:

All the core functionalities are now in place. The only issue now is that the next song does not automatically play when the currently playing song ends.

To fix that, you can set up an event listener which will detect when the currently playing song ends. The `"ended"` event listener is appropriate for this. It is fired when the playback of a media reaches the end.

Add an event listener to the `audio` element which listens for the `"ended"` event. Pass in a callback using arrow syntax with empty curly braces.

Step 96:

Notice that the album art in the HTML and songs in the `userData.songs` array have changed. We've swapped out the original songs for shorter ones that you can use to test your app in the upcoming steps.

Next, you need to check if there is a next song to play. Retrieve the current song index by calling the `getCurrentSongIndex()` function, and save it in a `currentSongIndex` constant.

After that, create a `nextSongExists` constant that contains the boolean value `true` or `false` depending on if the next song exists.

Step 97:

Use an `if` statement to check if `nextSongExists` exists, then call the `playNextSong()` function in the `if` block. This will automatically play the next song when the current song ends.

Step 98:

If there is no next song in the playlist, use the `else` block to reset the `currentSong` key of `userData` to `null`, and its `songCurrentTime` property to `0`.

Step 99:

With everything set in place, call the `pauseSong()`, `setPlayerDisplay()`, `highlightCurrentSong()`, and `setPlayButtonAccessibleText()` functions to correctly update the player.

Congratulations on completing your music player! Now that we've finished testing and using the shorter songs, we've replaced them with the original tracks specially selected by Quincy for you to enjoy.