Asynchronous Javascript

Asynchronous JavaScript is a way of controlling the flow of your code by running certain functions at a later time or based on certain conditions. This can be useful for handling time-consuming tasks, such as making API calls, without freezing the rest of your application, or delaying things (for example animations, sounds). In this introduction to asynchronous code in JavaScript, we'll use examples such as the **setTimeout** function and the **addEventListener** function.

Using setTimeout

The setTimeout function is a built-in JavaScript function that allows you to run a function after a specified amount of time. You might have used it before. Here is an example:

```
setTimeout(() => {
 console.log("Hi");
}, 1000);
console.log("Outside");
```

The `setTimeout` function takes two parameters:

- A callback function, in the above example we use an anonymous arrow function `() =>` But its also possible to use a normal named function instead.
- The amount of time in milliseconds before the callback function is run.

In this example, we are using setTimeout to run the function `console.log("Hi")` after 1 second (1000 milliseconds). As you can see, the `console.log("Outside")` statement is executed before the `console.log("Hi")` statement, even though it appears after it in the code. This is because the setTimeout function is asynchronous, and it runs the function inside of it at a later time, allowing the rest of the code to continue to execute.

Using addEventListener

The addEventListener function is another way to use asynchronous code in JavaScript. We've used it before. This function allows you to attach an event listener to an HTML element, and run a function when that event is triggered. Here is an example:

```
const button = document.querySelector("button");
button.addEventListener("click", () => {
 console.log("Clicked");
});
console.log("Outside");
```

In this example, we are selecting a button element and attaching a click event listener to it. When the button is clicked, the function console.log("Clicked") will be executed. As you can see, this function will not be executed until the button is clicked, allowing the rest of the code to continue to execute.

Callbacks

Both setTimeout and addEventListener, use callbacks. A callback is a function that you pass to another function in order to be executed at a later time based on certain conditions. This can include executing after a specific amount of time or when an event occurs.

Callbacks are not only used in asynchronous code but also in synchronous code. For example, the forEach method takes a callback function as an argument, which is executed for each element in the array its executed on, for example:

```
const numbers = [1,2,3,4,5];
numbers.forEach((number) => {
   console.log(number);
});
```

Callbacks can be very useful way to structure code. However, using too many callbacks, nested inside each other, can lead to a phenomenon known as "callback hell." For example, consider the following code:

```
setTimeout(() => {
   console.log("Callback 1");
   setTimeout(() => {
       console.log("Callback 2");
       setTimeout(() => {
           console.log("Callback 3");
           setTimeout(() => {
               console.log("Callback 4");
               // and so on...
           }, 1000);
       }, 1000);
   }, 1000);
}, 1000);
```

All we're trying to do here is run a setTimeout after another, but because we can only get the delay when we're inside the callback function code-block the code becomes nested and difficult to read as the number of setTimeout callbacks increases. Each callback is waiting for the previous one to complete before executing, resulting in a "pyramid" of nested code that can be hard to understand and manage.

Promises

One way to avoid callback hell is by using promises. Promises are a way to handle asynchronous code that makes it easier to reason about and avoid callback hell by chaining multiple asynchronous calls together in a more readable way. They provide a more elegant way to handle asynchronous operations and make the code look more organized. Promises are objects that represent the eventual completion (or failure) of an asynchronous operation, and its resulting value. They have several states, such as "pending", "fulfilled", and "rejected", and provide methods to handle the success and failure cases.

Here's an example of a simple setTimeout function that uses a promise:

```
function setTimeoutPromise(duration) {
 return new Promise(function(resolve) {
   setTimeout(resolve, duration);
 });
}
```

In this example, the setTimeoutPromise function returns a new promise that resolves after the specified duration.

If we just try to run:

```
setTimeoutPromise(2000)
```

Nothing will happen, because all the line above does is create a promise, but it doesn't resolve it.

One way to resolve the promise (in our case to create a delay that then does something) is to use the `.then` method to handle the resolved value of a promise. For example:

```
setTimeoutPromise(2000)
 .then(function() {
   console.log("Hello, World!");
 });
```

While this can be useful for simple things, it does not help with that fact that in order to chain multiple delays together we'd still have to recreate a "callback hell", a more modern approach to deal with promises is "async-await".

Async/Await

Async/await is an alternative way of handling asynchronous code that can make the code more readable. In order to use async/await we need to define an function, this allows us to use the await keyword inside of them, for example:

```
async function doStuff() {
 console.log("Starting...")
 await setTimeoutPromise(250)
 console.log("First delay done!")
 await setTimeoutPromise(250)
 console.log("Second delay done!")
}
doStuff()
```