Week 07

Monday, May 31, 2021 11:53 AM

In JavaScript, functions are first-class objects, which means they can be passed around in the same way as every other value. They can have their own properties and methods, as well as accepting other functions as parameters and being returned by other functions.

The call() method can be used to set the value of this inside a function to an object that is provided as the first argument

```
function sayHello(){
        return `Hello, my name is ${ this.name }`; }

const clark = { name: 'Clark' };
const bruce = { name: 'Bruce' };

sayHello.call(clark);
<< 'Hello, my name is Clarke'

sayHello.call(bruce);
 << 'Hello, my name is Bruce'
```

An Immediately Invoked Function Expression – or IIFE (pronounced “iffy”) – is an anonymous function that, as the name suggests, is invoked as soon as it’s defined.

Placing any code that uses the temporary variable inside an IIFE will ensure it’s only available while the IIFE is invoked, then it will disappear.

Recursive Functions

A recursive function is one that invokes itself until a certain condition is met. It’s a useful tool to use when iterative processes are involved. A common example is a function that calculates the factorial1 of a number:

```
function factorial(n) { if (n === 0) { return 1; } else { return n * factorial(n - 1); } };
```

This function will return 1 if 0 is provided as an argument (0 factorial is 1), otherwise it will multiply the argument by the result of invoking itself with an argument of one less. The function will continue to invoke itself until finally the argument is 0 and 1 is returned. This will result in a multiplication of 1, 2, 3 and all the numbers up to the original argument.

Callbacks

You’ll recall that they’re functions passed to other functions as arguments and then invoked inside the function they are passed to.

Event-driven Asynchronous Programming

Callbacks can be used to facilitate event-driven asynchronous programming. JavaScript is a **single-threaded environment**, which means only one piece of code will ever be processed at a time. This may seem like a limitation, but nonblocking techniques can be used to ensure that the program continues to run. Instead of waiting for an event to occur, a callback can be created that’s invoked when the event happens. This means that the code is able to run out of order, or asynchronously.

The increase in the use of asynchronous programming in JavaScript has meant that more and more callbacks are being used. This can result in messy and confusing “spaghetti code”. This is when more than one callback is used in the same function, resulting in a large number of nested blocks that are difficult to comprehend. **Callback hell** is the term used to refer to this tangled mess of code.

Promises

A promise represents the future result of an asynchronous operation. Promises don't do anything that can't already be achieved using callbacks, but they help simplify the process, and avoid the convoluted code that can result from using multiple callbacks.

When a promise is created, it calls an asynchronous operation and is then said to be pending. It remains in this state while the operation is taking place. At this stage, the promise is said to be unsettled. Once the operation has completed, the promise is said to have been settled.

Creating a Promise

```
const promise = new Promise( (resolve, reject) => {
        // initialization code goes here
        if (success) {
                resolve(value);
        } else {
                reject(error);
        }
});
```

Promises come into their own when multiple asynchronous tasks are required to be carried out one after the other. If each function that performs an asynchronous operation returns a promise, we can chain the then() methods together to form a sequential piece of code that's easy to read. Each promise will only begin once the previous promise has been settled.

```
login(userName)
 .then(user => getPlayerInfo(user.id))
 .then(info => loadGame(info))
 .catch( throw error)
```

A closure is a reference to a variable that was created inside the scope of another function, but is then kept alive and used in another part of the program.

Functional Programming

Functional programming is a programming paradigm. Other examples of programming paradigms include object oriented programming and procedural programming. JavaScript is a multi-paradigm language, meaning that it can be used to program in a variety of paradigms (and sometimes a mash-up of them!). This flexibility is an attractive feature of the language, but it also makes it harder to adopt a particular coding style as the principles are not enforced by the language. A language such as Haskell, which is a purely functional language, is much stricter about adhering to the principles of functional programming.

Pure Functions

A key aspect of functional programming is its use of pure functions. A pure function is a function that adheres to the following rules:

1) The return value of a pure function should only depend on the values provided as arguments. It doesn't rely on values from somewhere else in the program.
2) There are no side-effects. A pure function doesn't change any values or data elsewhere in the program. It only makes non-destructive data transformations and returns new values, rather than altering any of the underlying data.
3) Referential transparency. Given the same arguments, a pure function will always return the same result.

Functional programming uses pure functions as the building blocks of a program. The functions perform a series of operations without changing the state of any data. Each function forms an abstraction that should perform a single task, while encapsulating the details of its implementation inside the body of the function. This means that a program becomes a sequence of expressions based on the return values of pure functions. The emphasis is placed on using function composition to combine pure functions together to complete more complex tasks.

Ajax

Ajax is a technique that allows web pages to communicate asynchronously with a server, and it dynamically updates web pages without reloading. This enables data to be sent and received in the background, as well as portions of a page to be updated in response to user events, while the rest of the program continues to run

Ajax was a neat acronym that referred to the different parts of the process being used: Asynchronous JavaScript and XML:

Asynchronous	When a request for data is sent, the program doesn't have to stop and wait for the response. It can carry on running, waiting for an event to fire when a response is received. By using callbacks to manage this, programs are able to run in an efficient way, avoiding lag as data is transferred back and forth.
JavaScript	JavaScript was always considered a front-end language, not used to communicate with the server. Ajax enabled JavaScript to send requests and receive responses from a server, allowing content to be updated in real time.
XML	When the term Ajax was originally coined, XML documents were often used to return data. Many different types of data can be sent, but by far the most commonly used in Ajax nowadays is JSON, which is more lightweight and easier to parse than XML. (Although it has never really taken off, the termAjaj is sometimes used to describe the technique.) JSON also has the advantage of being natively supported in JavaScript, so you can deal with JavaScript objects rather than having to parse XML files using DOM methods.

The Fetch API

It has since been superseded by the Fetch API, which is currently a living standard for requesting and sending data asynchronously across a network. The Fetch API uses promises to avoid callback hell, and also streamlines a number of concepts that had become cumbersome when using the XMLHttpRequest object.

JSON Responses

JSON is probably the most common format for AJAX responses. The json() method is used to deal with these by transforming a stream of JSON data into a promise that resolves to a JavaScript object.

```
fetch(url)
 .then( response => response.json() ); // transforms the
➥ JSON data into a JavaScript object
 .then( data => console.log(Object.entries(data)) )
  .catch( error => console.log('There was an error: ',
➥ error))
```

The Fetch API is, at the time of writing, what is known as a "living standard", which means that the specification is being developed in the wild . This means that, despite it being available to use, it's still subject to change as developers, browser vendors and end-users provide feedback about how it works. It's an experimental technology, and new features might get added, or the syntax and behavior of some properties and methods might change in the future.