Springboard

Callback Functions

js-callback-functions-demo.zip 1.5KB

Goals

- Learn how Javascript treats functions as first-class data types
- Introduce callback functions
- Explore a pattern for using callbacks to write declarative code
- Introduce Anonymous functions
- Understand the difference between synchronous and asynchronous code
- Learn about setTimeout and setInterval

First Class Functions

- Most programming languages do not treat functions the same as other data types
- Javascript functions are considered first-class because they are treated like any other data type

- Functions in Javascript are first-class because of the following 3 features
  1. Functions can be passed as arguments to other functions
  2. The return value of a function can be another function
  3. A function can be assigned to a variable

```
function name(arg) { arg(); } function fn() { console.log('This is function 2'); } name(fn);
```

- Here we are passing the function 'fn' to the 'name' function as an argument
- This function can be invoked later by the 'name' function

```
function fn1(cb) { cb(); cb(); cb(); } function fn2() { console.log('This is function 2'); } fn1(fn2); // This is function 2 // This is function 2 // This is function 2
```

- Just as with any other argument to a function we can rename that argument anything we like
- The function can be invoked multiple times from within the 'fn1', it's just a function

Callback functions

- All functions are considered first-class in Javascript
- A callback functions is how we refer to functions that has been passed as an argument

```
function fn1(cb) { cb();// this is a callback function} function fn2() { console.log('This is function 2'); } fn1(fn2);
```

```
function greet(argument, cb) { cb(argument); } function fn2(nam
e) { console.log('Hi, ' + name); } greet('Lena', fn2);
```

- Callback functions are just like any regular function
- We are allowed to pass arguments to callback functions

Declarative callback pattern

```
function greet(argument, cb) { return cb(argument); } function
casual(name) { return 'Hi, ' + name; } function warm(name) {
return 'Hello, ' + name; } function bro(name) { return 'Yo, ' +
name; } greet('Lena', casual); // declarative code for greeting
Lina casually greet('Lena', warm);
```

- The ability to pass functions as arguments gives us a lot of flexibility in how we structure our programs
- In this example we use a declarative pattern
- Declarative patterns are built into the language with Array methods like ‘map’, ‘forEach’, and ‘filter’

Anonymous functions

- We aren’t restricted to declaring our functions before we pass then in as arguments
- A common pattern is to declare a function as we are invoking the function we are passing it to

```
function greet(argument, cb) { return cb(); } greet('Lena',
function() {});
```

If we are declaring a function as an argument, the name is optional

```
function greet(argument, cb) { return cb(); } greet('Lena',
function() { // this function will run when called by greet });
```

- Here the function declaration does not include a name
- This is known as an anonymous function

```
function greet(argument, cb) { return cb(argument); }
greet('Lena', function() { return 'Hi, ' + name; });
greet('Lena', function() { return 'Hello, ' + name; });
greet('Lena', function() { return 'Yo, ' + name; });
```

- Here is the pattern we saw previously refactored with anonymous functions

Asynchronous code introduction

- First-class functions in Javascript give us the ability to run code both synchronously and asynchronously
- Many programming languages are strictly synchronous meaning that we execute one request at a time in order
- Asynchronous code allows us to make a request, put it off to the side, then initiate other requests without stopping
- For each request that needs time to finish, we pass a callback to be executed once the request is complete
- It is difficult to demonstrate asynchronous code without having some sort of backend that we can make requests to
- Instead we will use pseudo code to demonstrate what an asynchronous request to a server might look like

Typical callback pattern

- Often, a callback function will not be called until after another function has finished executing

```
function getUser(username, cb) { // a request to the database
is made // after the request, the result is passed to the
cb(userInfo) } function updateView(userInfo) { // update the
webpage with the user info } getUser('Lina');
```

- Our 'getUser' function might take some time to resolve, we don't want to block our program to wait for the response from the server

```
function getUser(username, cb) { // a request to the database
is made // after the request, the result is passed to the
cb(userInfo) } function updateView(userInfo) { // update the
webpage with the user info } getUser('Lina');
getFriends('Lina')
```

- Let's imagine we have another request to get all of Lina's friends
- The getFriends request does not wait until getUser is finished
- This is an example of asynchronous code
    - We can no longer rely on the order of the code
    - but we don't have to stop and wait before making additional requests

Asynchronous code

- It's difficult to demonstrate asynchronous code without a backend but there are examples of asynchronous code built into the Javascript library
- In the following section we will explore two examples of asynchronous timers

Timers

setTimeout

It's quite common to write code that we want to be executed after a specific amount of time.

Maybe you want to perform some animation on the page after a fixed amount of time has elapsed.

To do this, we use the setTimeout

- We typically pass two arguments to setTimeout
- The first argument is a callback function
- The second is the number of milliseconds we want to wait before executing the callback function

setTimeout example

```
setTimeout(function() { console.log("Hello!"); }, 1000);
```

- This will log Hello! after one second

A more declarative example of setTimeout

```
function greet() { console.log("Hello!"); } setTimeout(greet,
1000);
```

setInterval

- setInterval is similar to setTimeout
- With setTimeout you invoke the callback function only once after the specified number of milliseconds
- With setInterval you continuously invoke the callback function until the timer is cleared
- The second argument is the number of milliseconds between each function invocation

setInterval example

```
setInterval(function() { console.log("Hello!"); }, 1000);
```

- This will log Hello! every second until the timer is cleared

Stopping the timer

What happens if we want to stop the timer?

setTimeout and setInterval return a special value called a timer id.

If we pass this value into the clearInterval method, we can stop our timer!

```
let timerId = setInterval(function() { console.log("Hello!");
}, 1000); clearInterval(timerId);
```

It may not be as common to want to stop a setTimeout before it executes but doing so looks similar to stopping a setInterval

```
let timerId = setTimeout(function() { console.log("Hello!"); },
1000); clearTimeout(timerId);
```

- The callback function will not actually if clearTimeout is invoked before 1000 milliseconds
- Once the callback function is invoked there is no longer any need to clear setTimeout

Timers

- Both timers are examples of asynchronous code
- If we pass a callback function to setTimeout or setInterval we can move on to the next request
- Each timer will automatically execute the callback function once the specified milliseconds have passed