Exploration — Functions and Functional Programming

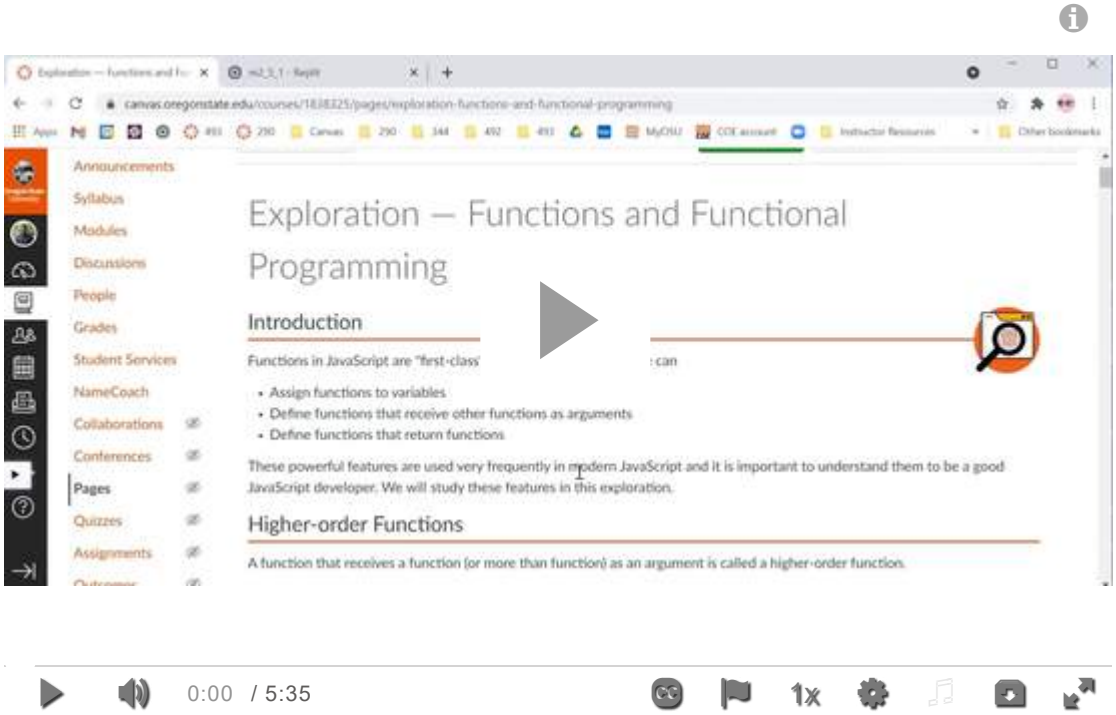
Introduction

Functions in JavaScript are "first-class" values. This means that we can

- Assign functions to variables
- Define functions that receive other functions as arguments
- Define functions that return functions

These powerful features are used very frequently in modern JavaScript and it is important to understand them to be a good JavaScript developer. We will study these features in this exploration.

Higher-order Functions



A function that receives a function(s) as an argument is called a higher-order function.

Example

Run

open in replit

index.js

```
'use strict';

/**
 * Apply the function f on each element of the array arrayIn
 * and return an array with the result
 * @param {function} A function that takes one argument and returns a value
 * @param {array}
 * @return An array
```

Console Shell

[redacted]

In this example, we define a function `ourMap` which takes two arguments:

- `f`: This argument should be a function that takes one argument
- `arrayIn`: This argument should be an array. `ourMap` calls the function `f` on every element of the array `arrayIn`, collecting the result in another array called `arrayOut` and returns `arrayOut`. `ourMap` is thus a higher-order function.

We also define another function `square` which returns the square of the argument passed to it.

In the example, we call `ourMap` with argument `f` set to `square`. `arrayOut` thus includes the square of elements of `arrayIn`.

The built-in map function

JavaScript provides many built-in higher-order functions and methods.

The **Map** object is provided by **ES6**. A map in JS is "a collection of elements where each element is stored as a *Key, value* pair. *Map* object can hold both *objects and primitive* values as either key or value. When we iterate over the map object it returns the key, value pair in the same order as inserted." (**Geeks for Geeks** **(https://www.geeksforgeeks.org/map-in-javascript/)**)

To reduce the number of lines of code for the `ourMap` function, we can replace it with the built-in **`Array.map`** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/map)**.

Run

open in replit

index.js

```
1  'use strict';
2
3  function square(x) {
4      return x * x;
5  }
6
7  const numbers = [23, 44, 15, 18];

8  console.log(numbers);
```

Console Shell

[redacted]

Exercise

JavaScript provides a method **Array.filter()** **(https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter)** which

- Applies a predicate function, i.e., a function that receives one argument and returns a Boolean value, to the array, and
- Returns an array containing only those element of the array for which the predicate function returns true.

Write a function `myFilter` which has the same functionality as `Array.filter()`. This means that `myFilter`:

- Takes two arguments:
  - `p`: a predicate function
  - `arrayIn`: an array
- Returns an array containing only those elements of `arrayIn` for which `p` returns true.

Here is an example of calling `myFilter` and the result it produces.

Run

open in replit

index.js

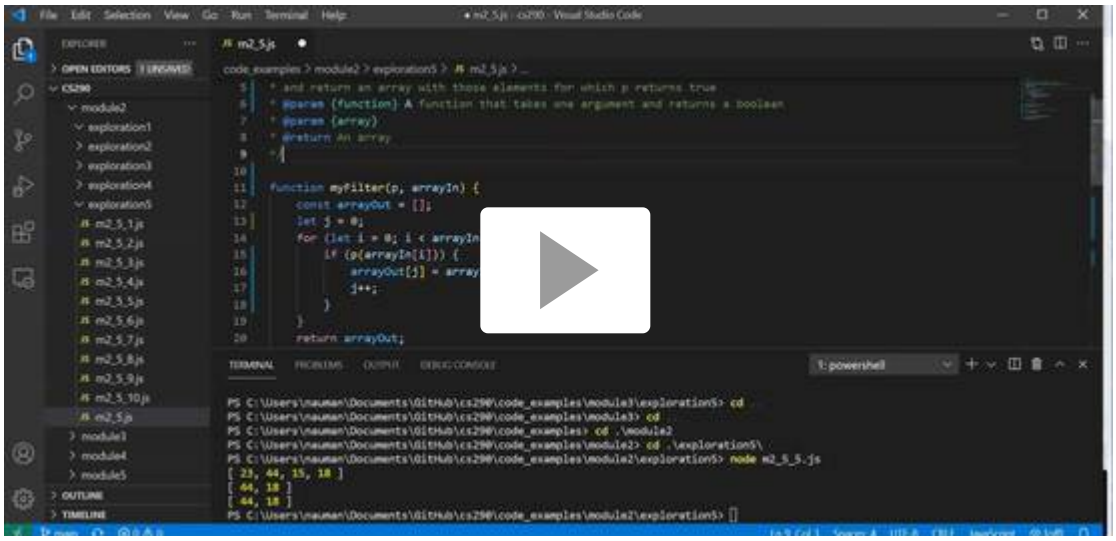
```
'use strict';

/**
 * Apply the function p on each element of the array arrayIn
 * and return an array with those elements for which p returns true
 * @param {function} A function that takes one argument and returns a boolean
 * @param {array}
 * @return An array
```

Console Shell

[redacted]

Function Expressions



0:00 / 8:27

Many times the function we pass to a higher-order function may not be useful other than for this call. In our examples of using `myMap` and `myFilter`, the functions `square` and `isEven` aren't particularly useful as general functions. In such cases, we can use **anonymous functions**, i.e., functions that don't have a name. Anonymous functions are defined using **function expressions**, i.e., expressions that return a function. Function expressions can be defined using two different syntaxes.

Anonymous function expression using the function keyword

The definition of the function is similar to how we have declared functions up till now, except that the function does not have a name.

Example

In the following example, we modify the code so that the first argument to `myFilter` is an anonymous function defined using a function expression. Note that this function expression is the same as the `isEven` declared function, except that we removed the name `isEven`.

Run

open in replit

index.js

```
'use strict';

/**
* Apply the function p on each element of the array arrayIn
* and return an array with those elements for which p returns true
* @param {function} A function that takes one argument and returns a boolean
* @param {array}
* @return An array
```

Console Shell

Anonymous function expression using arrow function syntax

JavaScript provides a more concise syntax for function expressions using the **arrow operator** which is `=>`. An arrow function has the following syntax:

- The parameter variables in parentheses.
  - If there are multiple parameters, the variables are separated by commas.
  - If there is only one parameter, parenthesis are not needed.
  - If there are no parameters, the parentheses are empty.
- The arrow operator, i.e., `=>`.
- The function body.
  - If the function body is only an expression, we don't need to use the `return` keyword and don't need to enclose the body in `{}`.

Example

In the following example, we modify the code to so that the first argument to `myFilter` is an anonymous arrow function that returns `true` if the argument is even.

Run

open in replit

index.js

```
'use strict';

/**
* Apply the function p on each element of the array arrayIn
* and return an array with those elements for which p returns true
* @param {function} A function that takes one argument and returns a boolean
* @param {array}
* @return An array
```

Console Shell

[redacted]

Notice the arrow function is very succinct `x => x % 2 === 0`.

Named Functions Using Function Expressions

A function expression returns a function. We can define named functions using function expressions by simply assigning a function expression to a variable.

Example

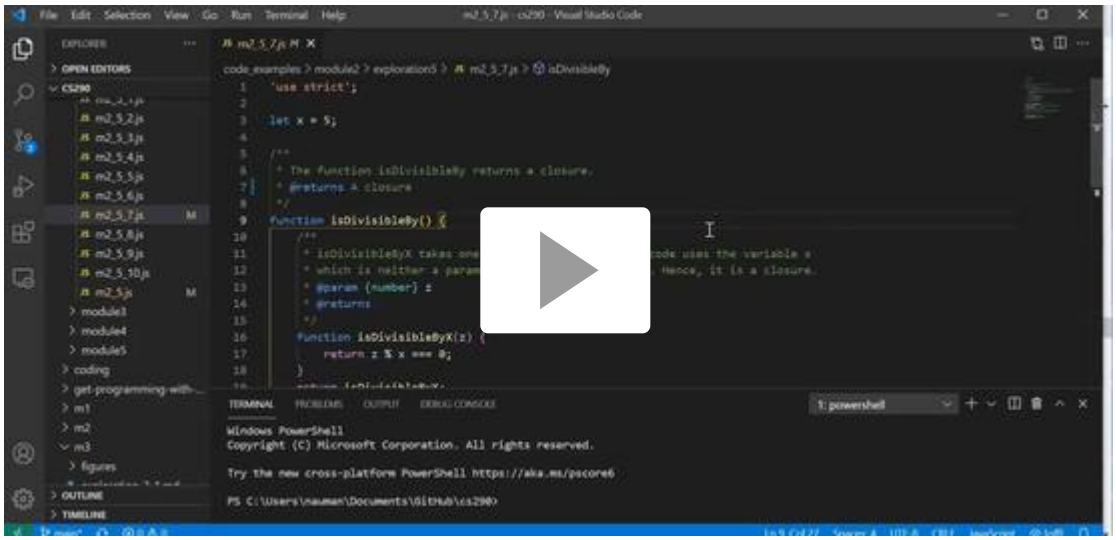
We can define a function `isEven` using arrow syntax as follows:

```
const isEven = (x => x % 2 === 0);
```

We can also define `isEven` using a function expression with the `function` keyword as follows:

```
const isEven = function(x) { return x % 2 === 0;};
```

Closure



The fundamental idea of closures is best explained by drawing a contrast between a closure and a function which is not a closure:

- The variables used in the code of a "regular" function (i.e., a function that is not a closure) are either parameters passed to the function or local variables.
- As opposed to this, the variables in a function which is a closure include one or more variables that are neither parameters passed to the function nor are local variables. Such variables are called **free variables**. These variables exist when the function was defined, but they may or may not exist when the closure is executed.
- A closure thus has the additional capability that it can store or **capture** variables in the environment in which it is defined. The closure can then access these variables at the time of its execution which may be later than when the closure was defined.

- In JavaScript, the free variable is a reference to the captured variable. It is not a reference to the value of this captured variable at the time the closure was defined. This means that if the value of the captured variable changes after the closure has been defined, the value of the free variable inside the closure also changes.

Example

In the following example, the function `isDivisibleByX` takes one parameter `z`. However, its code also uses the variable `x` which is neither a parameter nor a local variable in this function. Thus, `x` is a free variable and `isDivisibleByX` is a closure.

When `isDivisibleBy` is executed, the closure `isDivisibleByX` is defined by capturing the variable `x`. Inside this closure, the variable `x` refers to the current value of the variable `x`. When we change the value of `x`, the behavior of the function changes, because now the value of `x` inside the closure is set to the new value of `x` outside the closure.

Run

open in replit

index.js

```
1  'use strict';
2
3  let x = 5;
4
5  /**
6   * The function isDivisibleBy returns a closure.
    * @returns
    */
7  function isDivisibleBy() {
```

Console Shell

Uses of closure

Closures have been a feature of functional programming languages for a long time. Closures are very widely used in JavaScript code. Even if you don't write closures that much, it is very likely that

you will read code, e.g., written by others, in docs, etc., that uses closures. They have many uses, but we will look at their use for information hiding. See the references provided in the section **Additional Resources** if you are interested in learning more about closures.

Example

Here is an example adapted from a **post in Stack Overflow (https://stackoverflow.com/questions/2728278/what-is-a-practical-use-for-a-closure-in-javascript)**

Run

open in replit

index.js

```
'use strict';

/**
 * Returns a closure which is a counter incremented on each call.
 * @returns A closure
 */
function createCounter() {
    let count = 0;
    return function counter() {
```

Console Shell

[redacted]

The variable `count` is hidden inside the closure and its value cannot be accessed outside the closure.

Function Arguments

If we call a function with fewer arguments then the number of parameters it declares, the missing arguments are set to `undefined`. On the other hand, if we call a function with more arguments than the number of parameters declared in the function, the extra arguments are ignored.

Specifying Default Arguments

It is possible to define a function with default values for one or more parameters. If such a function is called so that for the argument with a default value, either we do not provide a value or we provide the value `undefined`, then the default value of this argument is used instead.

Example

Run

open in replit

index.js

```
'use strict';

/**
 * Returns a closure which is a counter incremented on each call.
 * @param {number} initValue defaults to 0
 * @returns A clousure
 */
function createCounter(initValue = 0) {
    let count = initValue;
```

Console Shell

[redacted]

Exceptions

When an error occurs in a function, instead of returning the value `undefined` the function can throw an **exception (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#exception_handling_statements)**. Exceptions can be caught using the `catch` clause to take appropriate action.

Throwing Exceptions

An exception is thrown using the `throw` statement. As soon as the `throw` statement is executed, the execution doesn't continue to the next statement in this function. Instead, the nearest `catch` clause is executed. This `catch` clause may be in the function which threw the exception, or it might be in another function that has called this function either directly or indirectly. If the exception is not caught by any function, then the program terminates.

We can throw an exception with any type of value. For example:

```
throw 'Value is too big';
```

However, JavaScript provides a built-in `Error` object which has a name and a message. We can create an `Error` object with a custom message by using the function `Error()`. For example:

```
throw Error('Value is too big');
```

JavaScript also provides many different built-in **Error objects (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error#error_types)**.

Catching an exception

To catch an exception, we have to use a `try` statement and the `catch` clause. The general form of the statement is as follows:

```
try{
    // Code that may throw an exception
} catch (err){
    // Code to handle the exception, whose value is in the variable named err
}
```

As soon as any code inside the `try` block throws an exception, the program skips the rest of the code in the `try` block and executes the code in inside the `catch` clause. This code is also termed the exception handler.

Example

In the following example, we call `JSON.parse()` on a string which is not valid JSON (the key `name` should have been in double quotes). The method `JSON.parse()` throws the exception `SyntaxError` when its argument doesn't have valid syntax. We catch the exception and log it.

Run

open in replit

index.js

```
'use strict';

const invalidJSON = '{name: "John Doe"}';

try{
  let person = JSON.parse(invalidJSON);
  console.log(person);
} catch (err) {
  console.log(`Execution failed with exception ${err}`);
```

Console Shell

Custom handlers based on error type

Using the `instanceof` keyword, we can determine what type of exception has been thrown and tailor our response based on the type of exception. The general form of such code is as follows:

```
try {
    // Code that may throw an exception
} catch (err) {
  if (err instanceof SyntaxError) {
      // Do what you want to do for SyntaxError
  } else if (err instanceof RangeError) {
      // Do what you want to do for RangeError
  }
  // ... etc
}
```

Using a finally block

A `try` statement can also **optionally** have a `finally` clause. The code in the `finally` clause is always executed regardless of whether an exception occurs or not. A very common use of the `finally` clause is to make sure that resource acquired during the `try` block, e.g., a database connection, an open file, a network connection, are always released even if an exception occurs. The general form of the `finally` clause is as follows:

```
try{
    // Code that acquires resources
    // other code
} finally{
    // Release resources
}
```

The `finally` clause can also be used when a `catch` clause is being used (but there is no requirement that the `finally` clause must be used when you use a `catch` clause). The general form in that case is as follows:

```
try {
    // Code that acquires resources
    // other code
} catch(err){
    // Exception handler
} finally{
    // Release resources
}
```

Summary

In this exploration, we studied how functions in JavaScript are first-class values. We can define functions that return other functions, pass functions as arguments to other functions, and assign functions to variables. We studied anonymous functions and the two different syntaxes for defining them. We studied the concept of closures and how to define them in JavaScript.

Additional Resources

Here are some references to learn more about the topics we discussed in this exploration.

- Here are two good references on closures in JavaScript. **Closures in JavaScript: Why Do We Need Them? (https://blog.bitsrc.io/closures-in-javascript-why-do-we-need-them-2097f5317daf)** and **What is a closure (https://medium.com/javascript-scene/master-the-javascript-interview-what-is-a-closure-b2f0d2152b36)**. Should you search for other resources, filter out the ones where examples declare variables using `var` or declare global variables.
- For a discussion of error handling on MDN, see the **relevant page (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#exception_handling_statements)**.