Higher order functions

In JavaScript, functions are first-class citizens, which means that they can be treated as values, assigned to variables, passed as arguments to other functions, and even returned from functions.

In JavaScript, a higher-order function is a function that takes one or more functions as arguments or returns a function as its result. In other words, it operates on functions like other functions operate on data types such as numbers, strings, and objects.

Functions as argument

In JavaScript, you can pass a function as an argument to another function by simply providing the function name (without parentheses) as an argument to the higher-order function. Here's an example:

```
// Define a function that takes a function as an argument
function higherOrderFunction(callback) {
  // Call the callback function
  callback();
}

// Define a function to be passed as an argument
function myCallback() {
  console.log("Hello from myCallback!");
}

// Call the higher-order function and pass in the callback function
higherOrderFunction(myCallback);
```

In the above example, we have defined a higher-order function called **higherOrderFunction** that takes a function as an argument. We have also defined a function called **myCallback** that we want to pass as an argument to **higherOrderFunction**.

When we call **higherOrderFunction(myCallback)** the **myCallback** function is passed as an argument and is then invoked inside the **higherOrderFunction** function by calling **callback()**.

Example 1:

```
// Define a function that takes another function as an argument
function applyOperation(x, y, operation) {
  return operation(x, y);
}

// Define a function to be passed as an argument
function add(x, y) {
  return x + y;
}
// Call applyOperation with add function as the third argument
let result = applyOperation(3, 4, add);
console.log(result); // Output: 7
```

In the above example, **applyOperation** is a higher-order function that takes three arguments: **x**, **y**, and **operation**. The **operation** argument is expected to be a function that takes two arguments and returns a value. When **applyOperation** is called with the arguments **(3, 4, add)**, it passes the **add** function as the **operation** argument. The **add** function is then called with **x** and **y** as its arguments, and its return value (7) is returned by **applyOperation**.

Note that the **add** function is defined separately from the **applyOperation** function. This allows for greater flexibility and reusability of functions in your code.

Example 2:

```
function applyOperation(num1, num2, operation) {
  return operation(num1, num2);
}

function add(a, b) {
  return a + b;
}

function multiply(a, b) {
  return a * b;
}

console.log(applyOperation(2, 3, add)); // Output: 5
console.log(applyOperation(2, 3, multiply)); // Output: 6
```

In this example, the **applyOperation()** function takes three arguments: **num1**, **num2**, and **operation**. The **operation** argument is a function that takes two arguments and returns a value.

The **add()** and **multiply()** functions are defined as separate functions that take two arguments and return the sum and product of those arguments, respectively.

When we call **applyOperation()** with **2**, **3**, and **add**, the **add()** function is passed as the **operation** argument, so **applyOperation()** calls **add(2, 3)** and returns the result, which is **5**.

Similarly, when we call **applyOperation()** with **2**, **3**, and **multiply**, the **multiply()** function is passed as the **operation** argument, so **applyOperation()** calls **multiply(2, 3)** and returns the result, which is **6**.

Example 3:

```
// Define a function that takes another function as an argument
function processArray(arr, func) {
  let result = [];
  for(let i = 0; i < arr.length; i++) {
    result.push(func(arr[i]));
  }
  return result;
}

// Define a function that we will pass as an argument
function double(x) {
  return x * 2;
}

// Call the processArray function and pass the double function as an argument
let arr = [1, 2, 3, 4];
let doubledArr = processArray(arr, double);
console.log(doubledArr); // Output: [2, 4, 6, 8]
```

In this example, we define a function called **processArray** that takes two arguments: an array **arr** and a function **func**. Inside the **processArray** function, we loop through the **arr** array and apply the **func** function to each element, storing the result in a new **result** array. Finally, we return the **result** array.

We then define a function called **double** that takes a single argument and returns that argument multiplied by 2. We then call the **processArray** function and pass in the **arr** array and the **double** function as arguments. The **processArray** function applies the **double** function to each element of the **arr** array, resulting in a new array called **doubledArr** containing the doubled values of the original array.

Example 4:

```
function greet(name, callback) {
  console.log('Hello, ' + name + '!');
  callback();
}


function sayGoodbye() {
  console.log('Goodbye!');
}


greet('Alice', sayGoodbye);
```

In this example, we define a function called **greet** that takes two arguments: **name** and **callback**. The **name** argument is a string, and the **callback** argument is a function that will be called after the greeting has been printed. Inside the **greet** function, we first print a greeting using the **console.log** function, and then we call the **callback** function.

We also define a separate function called **sayGoodbye**, which simply prints "Goodbye!" to the console.

Finally, we call the **greet** function and pass in two arguments: the name "Alice" and the **sayGoodbye** function. This means that when the **greet** function is executed, it will first print "Hello, Alice!", and then it will call the **sayGoodbye** function, which will print "Goodbye!" to the console.

Note that when you pass a function as an argument, you do not include parentheses after the function name. This is because including parentheses would actually call the function, whereas we want to pass the function itself as an argument.

Return function from function

A function which returns another function is also called as Higher Order Function.

Example:

```
function outerFunction() {
  // define a function inside the outer function
  function innerFunction() {
    console.log('Hello from inner function!');
  }
  // return the inner function
  return innerFunction;
}

// call the outer function to get the inner function
const myFunction = outerFunction();

// call the inner function
myFunction(); // logs 'Hello from inner function!'
```

In this example, the **outerFunction** defines a new function called **innerFunction** and then returns it. When the **outerFunction** is called, it returns the **innerFunction**, which can be assigned to a variable **myFunction**. Finally, when **myFunction** is called, it logs the message **'Hello from inner function!'** to the console.

Example 1:

```
function createMultiplier(factor) {
  function multiplier(number) {
    return number * factor;
  }
  return multiplier;
}

const double = createMultiplier(2);
const triple = createMultiplier(3);

console.log(double(5)); // Output: 10
console.log(triple(5)); // Output: 15
```

In this example, the **createMultiplier** function returns a new function called **multiplier**. When you call **createMultiplier(2)**, it returns the **multiplier** function with **factor** set to 2. You can then assign the returned function to a variable and call it later, passing in a value to be multiplied.

So, when you call **double(5)**, it multiplies 5 by 2, and returns 10. Similarly, when you call **triple(5)**, it multiplies 5 by 3, and returns 15.