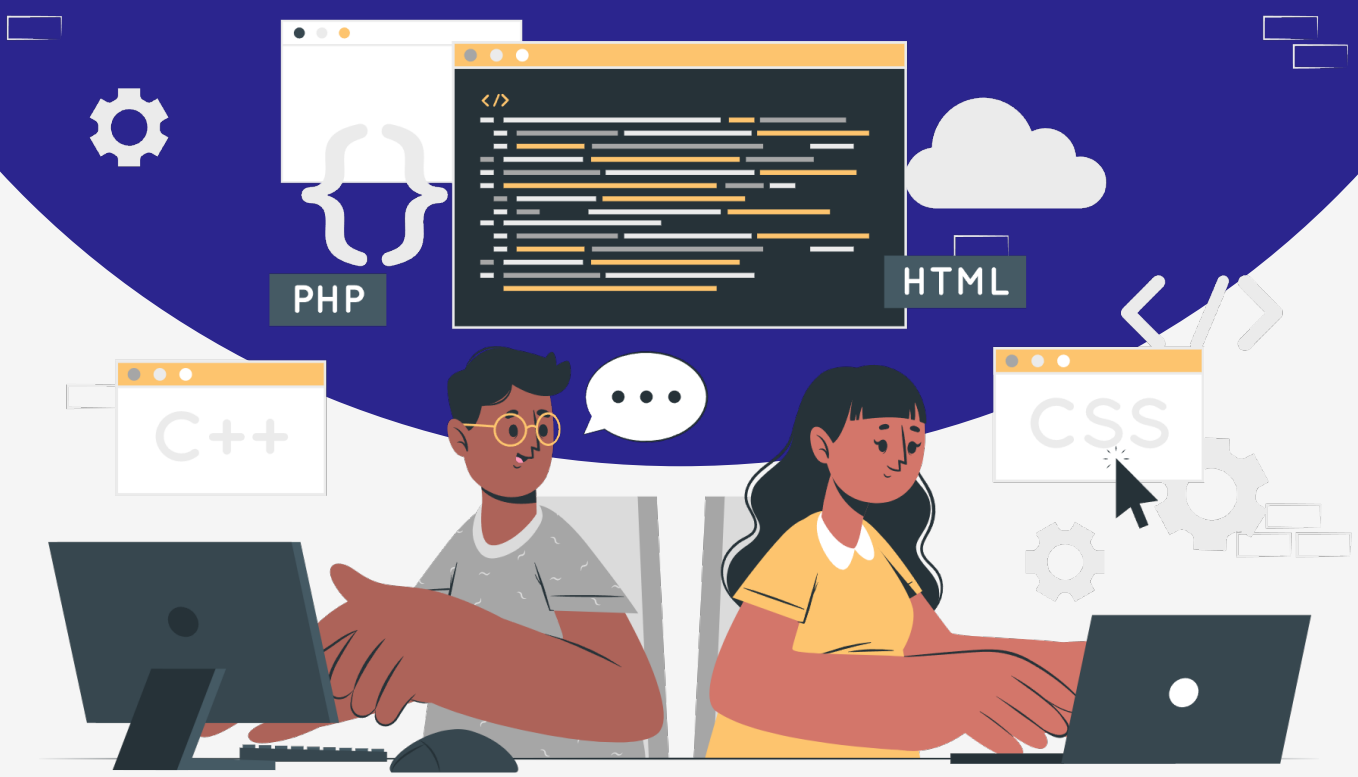
Lesson:

What is Higher Order Function (HOF)



PW SKILLS

Topics

1. **Higher-order functions**
2. **Examples of HOF**
3. **Use case of higher order functions**

Higher-order functions

The functions that use only primitives (predefined data types) or objects as arguments, and only return primitives or objects are named first-order functions.

However, functions are treated as first-class citizens in JavaScript which means that functions can be

- assigned to different variables

```
const hello = function() {
  return 'Hello!'
};
hello();
```

- passed as arguments to different functions

```
function useFunction(func) {
  return func();
}
useFunction(function () { return 7 });
```

- returned from different functions

```
function returnFunction() {
  return function() { return 7 };
}
const exFunc = returnFunction();
exFunc();
```

Higher-order functions are, in fact, functions that accept another function as an argument or return another function. In the above examples, useFunction() is a higher-order function because it accepts a function as an argument. Also, returnFunction() is a higher-order function because it returns another function.

Examples of HOF

EXAMPLE 1

```
function calculatorFunction(operation, initialValue,
numbers) {
  let total = initialValue;
  for (const number of numbers) {
    total = operation(total, number);
  }
  return total;
}
function sum(n1, n2) {
  return n1 + n2;
}
function multiply(n1, n2) {
  return n1 * n2;
}
calculatorFunction(sum, 0, [1, 3, 4]);    // 8
calculatorFunction(multiply, 1, [1, 3, 4]); // 12
```

Here, calculatorFunction(operation, initialValue, numbers) is a higher-order function as it accepts a function as the first argument.

sum() is the function that describes the addition operation. calculatorFunction(sum, 0, [1, 3, 4]) is using sum() function to perform the sum of numbers.

Similarly, multiply() describes the multiplication operation. calculatorFunction(multiply, 1, [1, 3, 4]) is using multiply() function to perform the product of numbers.

EXAMPLE 2

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

const double = createMultiplier(2);
console.log(double(5)); // Output: 10

const triple = createMultiplier(3);
console.log(triple(5)); // Output: 15
```

In the above example, the createMultiplier function is a higher-order function because it returns a new function. The createMultiplier function takes a **factor** as an argument and returns an inner function. The inner function takes a **number** as an argument and multiplies it by the factor provided to the createMultiplier function. We can then use the createMultiplier function to create specific multiplier functions, such as double and triple. The double function multiplies a number by 2, and the triple function multiplies a number by 3. We can call these functions with a number as an argument to perform the respective multiplication.

Use case of higher order functions

- ### Callback Functions

  One common use case of higher-order functions is the use of callback functions. Callback functions are passed as arguments to other functions and are executed at a certain point in the code. They allow you to perform actions asynchronously or handle events.
  In the above examples of HOF we had seen the usage of callback functions.

- **Array Iteration and Transformation**

  Higher-order functions like map(), filter(), reduce(), and forEach() are extensively used for iterating over arrays and transforming their elements. They take a function as an argument and apply it to each element of the array, allowing you to perform various operations.
  Let's take an example where we want to double the value of all the elements of the array -

```
// Original array of numbers
const numbers = [1, 2, 3, 4, 5];

// function to double a number
function doubleNumber(number) {
  return number * 2;
}

// Using the map function with the callback to create a
new array
const doubledNumbers = numbers.map(doubleNumber);

// Displaying the original and doubled arrays
console.log("Original Numbers:", numbers);
console.log("Doubled Numbers:", doubledNumbers);

// output
// Original Numbers: [ 1, 2, 3, 4, 5 ]
// Doubled Numbers: [ 2, 4, 6, 8, 10 ]
```

- **Function Composition**

  Function composition is the process of combining two or more functions to produce a new function. Higher-order functions enable composing functions by taking one function's output and passing it as an input to another function.
  In the below example, we are composition two simple functions into one by using HOF -

```
// Two basic functions
const double = (x) ⇒ x * 2;
const square = (x) ⇒ x * x;

// Function composition using a higher-order function
const compose = (func1, func2) ⇒ (value) ⇒
func1(func2(value));
```

```
// Compose the `double` and `square` functions
const doubleThenSquare = compose(square, double);

const result = doubleThenSquare(3);

// Display the result
console.log(result); // Output: 36
```

Question 1: Implement a findIndex function that returns the index of the first element in an array that satisfies a given condition.

```
// Implementing findIndex function
function findIndex(arr, callback) {
  for (let i = 0; i < arr.length; i++) {
    if (callback(arr[i])) {
      return i;
    }
  }
  return -1; // Return -1 if no element satisfies the
condition
}

// Example usage
const numbers = [1, 3, 5, 7, 9];
const index = findIndex(numbers, (num) => num > 5);
console.log(index); // Output: 3 (because 7 is the first
element > 5)
```

Question 2: Implement a customForEach function that iterates over an array and applies a callback to each element.

```
// Implementing customForEach function
function customForEach(arr, callback) {
  for (let i = 0; i < arr.length; i++) {
    callback(arr[i], i, arr);
  }
}

// Example usage
const numbers = [1, 2, 3, 4];
customForEach(numbers, (num) ⇒ console.log(num));
```

PW SKILLS

THANK YOU !