




# Welcome to the **Co**Grammar Functional Programming Lecture

The session will start shortly...

Questions? Drop them in the chat. We'll have dedicated moderators answering questions.



# Full Stack Web Development Session Housekeeping

---

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly.  
**(Fundamental British Values: Mutual Respect and Tolerance)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Academic Sessions. You can submit these questions here: [Questions](#)

## Full Stack Web Development Session Housekeeping cont.

---

- For all **non-academic questions**, please submit a query:  
[www.hyperiondev.com/support](http://www.hyperiondev.com/support)
- Report a **safeguarding** incident:  
[www.hyperiondev.com/safeguardreporting](http://www.hyperiondev.com/safeguardreporting)
- We would love your **feedback** on lectures: [Feedback on Lectures](#)

# Skills Bootcamp

## 8-Week Progression Overview

### Fulfil 4 Criteria to Graduation

#### ✓ Criterion 1: Initial Requirements

Timeframe: First 2 Weeks

Guided Learning Hours (GLH):

Minimum of 15 hours

Task Completion: First four tasks

**Due Date: 24 March 2024**

#### ✓ Criterion 2: Mid-Course Progress

**60** Guided Learning Hours

Data Science - **13 tasks**

Software Engineering - **13 tasks**

Web Development - **13 tasks**

**Due Date: 28 April 2024**

# Skills Bootcamp Progression Overview

## ✓ Criterion 3: Course Progress

Completion: All mandatory tasks,  
including Build Your Brand and  
resubmissions by study period end  
Interview Invitation: Within 4 weeks  
post-course  
Guided Learning Hours: Minimum of  
112 hours by support end date  
(10.5 hours average, each week)

## ✓ Criterion 4: Demonstrating Employability

Final Job or Apprenticeship  
Outcome: Document within 12  
weeks post-graduation  
Relevance: Progression to  
employment or related  
opportunity

**SKILLS  
FOR LIFE**

**SKILLS BOOTCAMPS**



Department  
for Education

# CoGrammar

# Functional Programming

March 2024



# Lecture Overview

- Higher Order Functions
- Callback Functions
- Function Composition



# Pure Functions

A function with no side effects that always return the same output, given the same input.

- ❖ Pure functions do not **use or modify any external variables**.
- ❖ They are **deterministic** functions which lead to **consistent results**, regardless of the external state of the program.
- ❖ The main benefits of pure functions are:
  - They are **independent**
  - They make your code more **readable**
  - You can **clone** the **external state into a pure function** without affecting the **internal state**



# Closure

A function along with its lexical scope.

- ❖ This concept refers to the concept that a **nested inner function** has access to the **outer function's variables and parameters**, even after the **end of the outer function's execution**.
- ❖ **Lexical Scope** refers to the parent function's variables.
- ❖ A nested function along with its parent function's variables and parameters forms a closure.

```
function outerFunction(outerParam) {  
  let outerFunctionVar;  
  function innerFunction(innerParam) {  
    console.log(outerParam);  
    outerFunctionVar = "initialise";  
    return innerParam;  
  }  
  return innerFunction;  
}
```

# Arrow Functions

Shorthand syntax for writing function expressions.

- ❖ We use an arrow ( `=>` ) to define these function shorthands.
- ❖ They are specifically used when the **function block** is **one line of code**.
- ❖ Arrow functions can improve the **readability** and **organisation** of code.

# Functional Programming

**Functional programming is a programming paradigm centered around the concept of functions as first-class citizens. It revolves around a set of principles.**

- ❖ Functions as First-Class Citizens:

- Functions can be assigned to variables, passed as arguments, and returned from other functions.

```
const add = (a, b) => a + b;
const multiply = (a, b) => a * b;

const calculate = (operation, a, b) => operation(a, b);

console.log(calculate(add, 3, 4)); // Output: 7
console.log(calculate(multiply, 2, 5)); // Output: 10
```

# Functional Programming

Functional programming is a programming paradigm centered around the concept of functions as first-class citizens. It revolves around a set of principles.

- ❖ Immutability:

- Data is treated as immutable to avoid unintended side effects.

```
const numbers = [1, 2, 3, 4, 5];  
  
// Using map to create a new array without modifying the original  
const doubledNumbers = numbers.map(num => num * 2);  
  
console.log(doubledNumbers); // Output: [2, 4, 6, 8, 10]
```

# Functional Programming

**Functional programming is a programming paradigm centered around the concept of functions as first-class citizens. It revolves around a set of principles.**

## ❖ Pure Functions:

- Pure functions always return the same output for the same input and have no side effects.

```
// Impure function
let total = 0;
const addToTotal = num => {
  total += num;
  return total;
};

console.log(addToTotal(5)); // Output: 5
console.log(addToTotal(3)); // Output: 8 (Side effect: modifies external state)

// Pure function
const pureAdd = (a, b) => a + b;

console.log(pureAdd(2, 3)); // Output: 5
console.log(pureAdd(2, 3)); // Output: 5 (No side effects)
```



# Higher Order Functions

Higher order functions are functions that can accept other functions as arguments or return functions as results.

- ❖ They enable **abstraction** and code **reusability**, crucial principles in functional programming.
- ❖ Some notable examples include **map()**, **filter()**, and **reduce()** in JavaScript.



# Higher Order Functions

Higher order functions are functions that can accept other functions as arguments or return functions as results.

- ❖ The **map()** function **applies** a provided function to each element of an array and returns a new array with the results.

```
const numbers = [1, 2, 3, 4, 5];  
const doubled = numbers.map(num => num * 2);  
console.log(doubled);
```

# Higher Order Functions

Higher order functions are functions that can accept other functions as arguments or return functions as results.

- ❖ The **filter()** function creates a new array with all elements that pass the test implemented by the provided function.

```
const scores = [80, 90, 60, 45, 75];  
const passed = scores.filter(score => score >= 70);  
console.log(passed);
```

# Higher Order Functions

Higher order functions are functions that can accept other functions as arguments or return functions as results.

- ❖ The **reduce()** function executes a **reducer** function on each element of the array, resulting in a single output value.

```
const numbers = [1, 2, 3, 4, 5];  
const sum = numbers.reduce((acc, num) => acc + num, 0);  
console.log(sum);
```

# Let's Breathe!

Let's take a small break  
before moving on to  
the next topic.





# Callback Functions

Callback functions are functions passed as arguments to other functions and executed later.

- ❖ They are commonly used in **asynchronous programming** and **event handling**.
- ❖ Callbacks are vital in handling asynchronous tasks, such as fetching data from an API.
- ❖ Callbacks play a crucial role in **event-driven programming**, responding to user interactions.





# Callback Functions

Callback functions are functions passed as arguments to other functions and executed later.

```
function fetchData(callback) {  
  setTimeout(() => {  
    const data = 'Data fetched asynchronously';  
    callback(data);  
  }, 2000);  
}  
  
fetchData(data => {  
  console.log(data);  
});
```



# Callback Functions

Callback functions are functions passed as arguments to other functions and executed later.

```
document.getElementById('myButton').addEventListener('click', () => {  
  console.log('Button clicked!');  
});
```

# Function Composition

Function composition is a technique used to combine multiple functions to create a new function.

- ❖ It involves **chaining** functions together, where the **output** of one function becomes the **input** of the **next**.
- ❖ Function **composition** enhances code modularity and readability by breaking down complex operations into smaller, composable units.

```
const add = x => x + 1;
const multiplyByTwo = x => x * 2;

// Compose two functions
const composedFunction = x => multiplyByTwo(add(x));

console.log(composedFunction(3)); // Output: 8
```

# Questions and Answers



# Thank you for attending



Department  
for Education

CoGrammar

