SWIFT CLOSURES ADVANCE LEVEL

PREPARED BY FARHAJ AHMED

COMPACT MAP

- The **compactMap** function is used to transform a collection by applying a closure and removing nil values.
- It's particularly useful when working with optionals.
- **compactMap** is similar to map, but it also removes nil values from the resulting array.

```swift
let optionalNumbers: [Int?] = [1, 2, nil, 4, nil, 6, 7, nil, 9, nil]

// Map to convert optional integers to integers, removing nil values
let nonNilNumbers = optionalNumbers.compactMap { $0 }
print(nonNilNumbers) // Output: [1, 2, 4, 6, 7, 9]

// Map to convert optional integers to strings, removing nil values
let nonNilNumberStrings = optionalNumbers.compactMap { $0.map { String($0) } }
print(nonNilNumberStrings) // Output: ["1", "2", "4", "6", "7", "9"]
```

COMPACT MAP

```swift
let strings = ["1", "2.5", "three", nil, "4.2"]

let numbers = strings.compactMap { str in
    if let number = Float(str) {
        return number
    } else {
        return nil
    }
}

print(numbers) // [1.0, 2.5, 4.2]
```

- **compactMap** is useful when you want to transform a collection and remove elements that don't produce a desired value (like nil in this case). It combines the functionality of map and filtering for nil values in a single step.
- This is particularly useful when you have an array of elements that need to be converted, but the conversion process might fail.

```swift
let scores = ["1", "2", "three", "four", "5"]

// Using map: Converts strings to optional Ints
let mapped: [Int?] = scores.map { str in Int(str) }
// Result: [1, 2, nil, nil, 5]

// Using flatMap: Filters out nil values
let compactMapped: [Int] = scores.compactMap { str in Int(str) }
// Result: [1, 2, 5]
```

FLAT MAP

- The **flatMap** function is used to **flatten nested collections** (such as arrays of arrays) into a **single-level collection**. It applies a closure to each element and returns a new collection by concatenating the results.

```
let arrayOfArrays = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

// Use flatMap to flatten the array of arrays into a single array
let flattenedArray = arrayOfArrays.flatMap { $0 }
print(flattenedArray) // Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]

// Applying a transformation while flattening
let transformedFlattenedArray = arrayOfArrays.flatMap { $0.map { $0 * 2 } }
print(transformedFlattenedArray) // Output: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```
let multiDimArray = [[1, 2], [3, 4], [5, 6]]
let flattenedArray = multiDimArray.flatMap { $0 }
```

FLAT MAP

```swift
let arrays = [["1", "2"], ["3", "4"], ["5", "6"]]

let strings = arrays.flatMap { array in
    return array
}

print(strings) // ["1", "2", "3", "4", "5", "6"]
```

```swift
let arrayOfOptionalArrays: [[Int?]] = [[1, 2, nil], [nil, 4, 5], [6, nil, 7]]

// Flatten the array of arrays of optionals while removing nil values
let flattenedNonNilNumbers = arrayOfOptionalArrays.flatMap { $0.compactMap { $0 } }
print(flattenedNonNilNumbers) // Output: [1, 2, 4, 5, 6, 7]
```

- flatMap is particularly useful when working with collections containing optional values. It allows you to both transform elements and flatten the resulting collection while automatically unwrapping and discarding nil values

FOR EACH

- In Swift 5, the **forEach** method is used to perform an operation on each element in an array or a set. It’s a concise way to iterate through the elements without explicitly writing a for-in loop.

```swift
let numbers = [1, 2, 3, 4, 5]

// Using a for-in loop
for number in numbers {
    print(number)
}

// Using forEach
numbers.forEach { print($0) }
```

FOR EACH

forEach is a simpler and more concise option for basic iteration without needing element indexes or modifying the collection.

```swift
let numbers = [1, 2, 3, 4, 5]

numbers.forEach { number in
    print(number)
}
```

```swift
let names = ["Alice", "Bob", "Charlie"]
names.forEach { print($0) }  // Prints each name on a new line
```

```swift
// Iterate over a dictionary and print key-value pairs
let fruits = ["apple": "red", "banana": "yellow", "orange": "orange"]
fruits.forEach { key, value in
  print("\(key): \(value)")
}
```

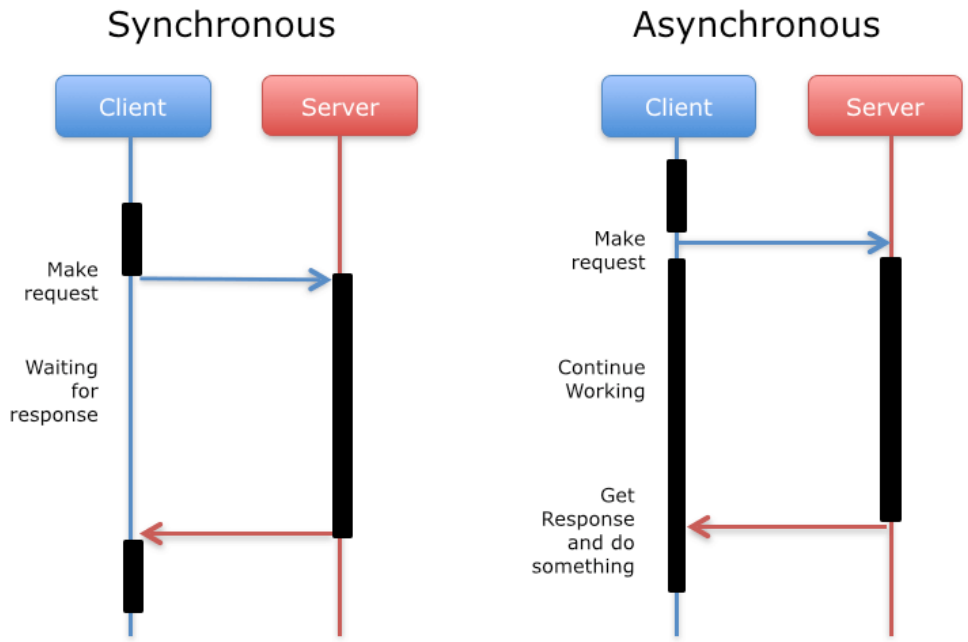
ALL SATISFY

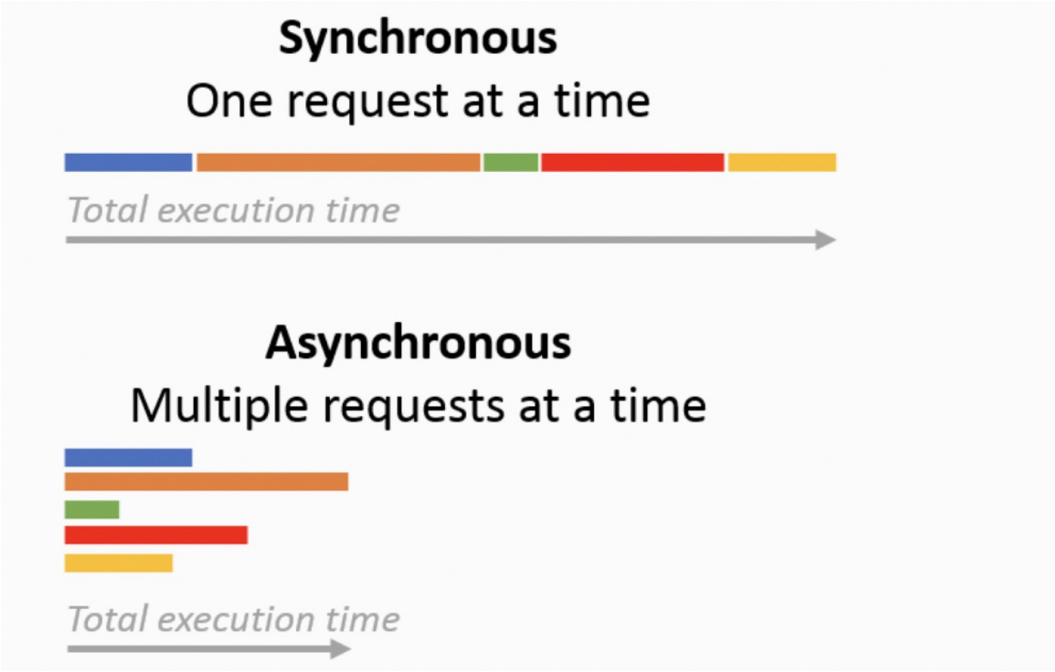
The allSatisfy closure in Swift 5 is used to check if **all** elements in a collection meet a certain condition. It returns true if every element satisfies the condition, and false otherwise

```swift
// Sample student grades
let grades = [85, 92, 75, 88, 60]

// Check if all grades are above 70 (using allSatisfy)
func allGradesPassing(grades: [Int]) -> Bool {
  return grades.allSatisfy { grade in
    grade > 70
  }
}

let allPassed = allGradesPassing(grades: grades)
print(allPassed) // Output: false (because one grade is below 70)
```

Synchronous Function

- **Execution:** A synchronous function or code block will **execute completely** before continuing to the next line of code.
- **Blocking:** The thread that calls the synchronous function is **blocked** until the function finishes its execution. This means the thread can't perform any other tasks while waiting.





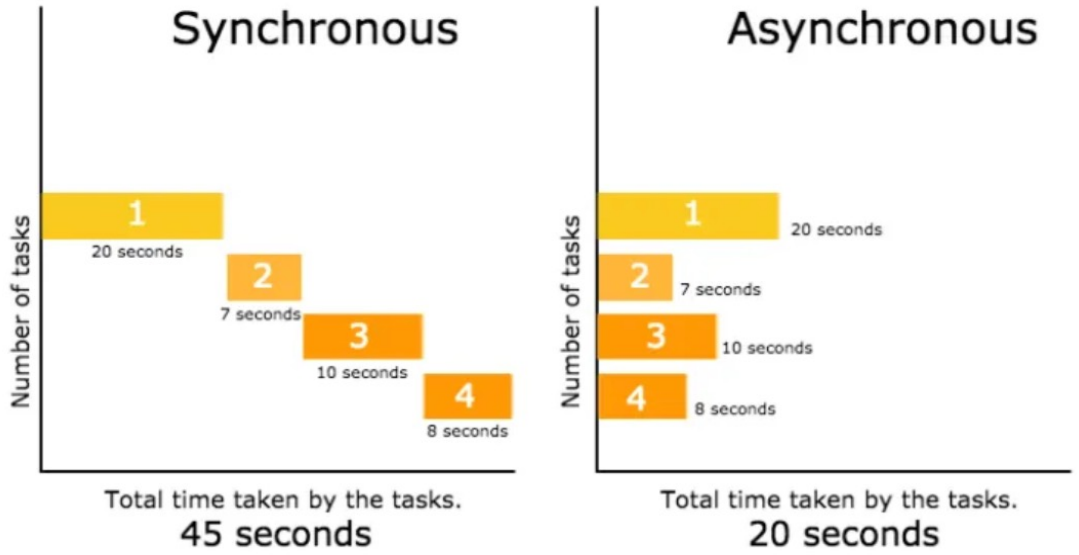
Example:

Swift

```swift
func addNumbers(num1: Int, num2: Int) -> Int {
  let sum = num1 + num2
  return sum
}

let result = addNumbers(num1: 5, num2: 10)
print(result) // This line will wait until the function finishes
```

Synchronous Function

- In this example, the **addNumbers** function is synchronous. The code calling it **(print(result))** will wait until the function finishes calculating the sum and returns the result before proceeding.



Asynchronous Function

- **Execution:** An asynchronous function or code block can **start execution** and then continue to the next line of code **without blocking the thread**. The actual work might happen later, in the background, or when a specific event occurs.
- **Non-Blocking:** The thread that calls the asynchronous function can continue executing other code while the asynchronous operation is ongoing.

Asynchronous Function

```swift
func downloadData(from url: URL, completion: @escaping (Data?) -> Void) {
  // Simulate asynchronous download
  DispatchQueue.global().asyncAfter(deadline: .now() + .seconds(2)) {
    let data = "Downloaded data".data(using: .utf8)
    completion(data)
  }
}

let someURL = URL(string: "https://www.geeksforgeeks.org/")!

downloadData(from: someURL) { data in
  if let data = data {
    print(String(decoding: data, as: UTF8.self))
  } else {
    print("Download failed")
  }
}

print("Continuing with other code...") // This line executes before download finishes
```

```
Continuing with other code...
Downloaded data
```

In this example, the **downloadData** function is asynchronous. The code calling it **(print("Continuing with other code..."))** executes right after the function is called. The actual download happens in the background, and the completion closure is called later when the download is complete (simulated by a delay here).

KEY TAKEAWAYS

Use **synchronous functions** for simple tasks where you need the result immediately.

Use **asynchronous functions** for long-running tasks to keep your app responsive and avoid blocking the main thread.

Be mindful of how you handle asynchronous operations and their results using mechanisms like closures and **DispatchQueues.**

THANK YOU