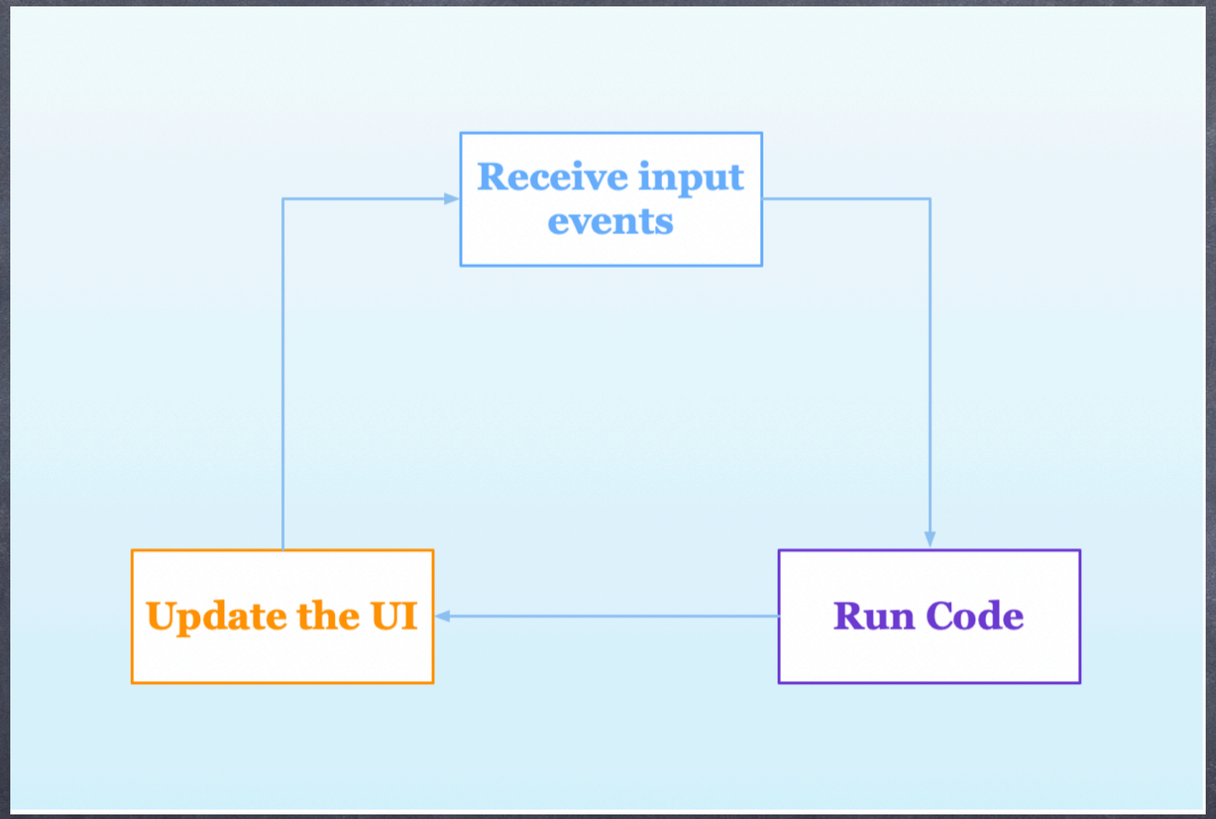
async - await

Structured way of achieving concurrency

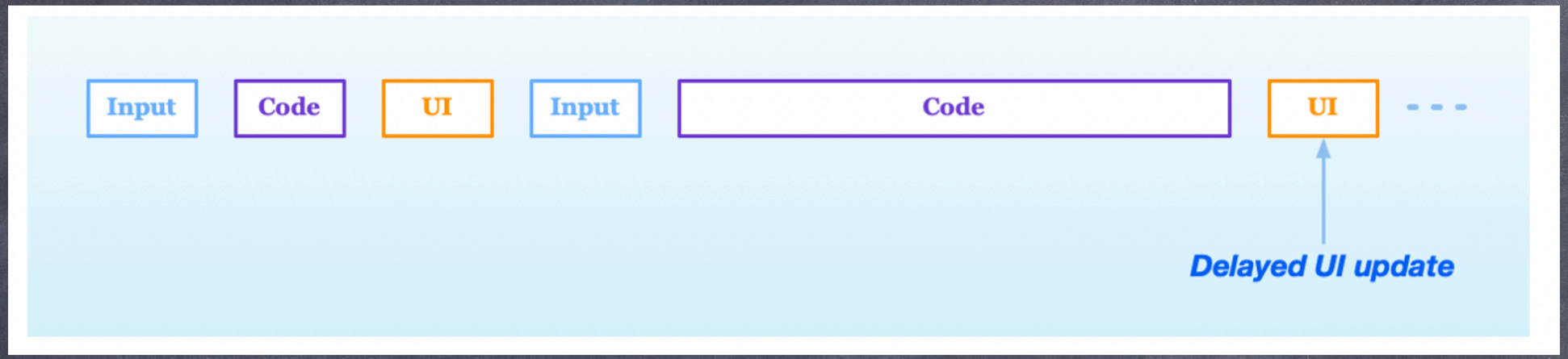
- Devendranath Reddy & Ketan Jogal

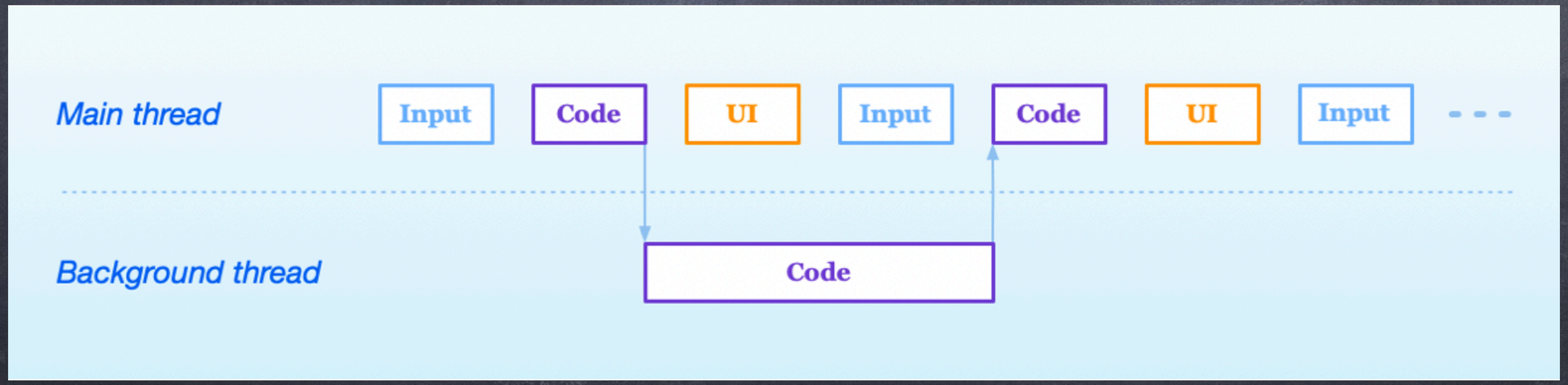
iOS App Behaviour



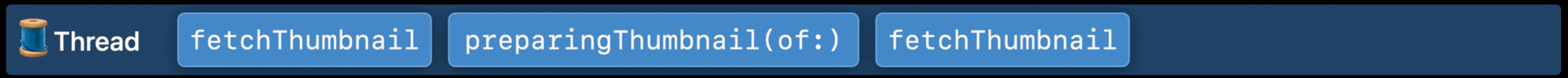
App responds to internal or external events and executes some code and update the User Interface

With / without Concurrency





Functions: synchronous and asynchronous



Use cases

- Calling complex API calls (Calling multiple APIs one after another and Dependent APIs)
- Performing long running tasks (Operations on File System and Databases etc)

Existing solutions

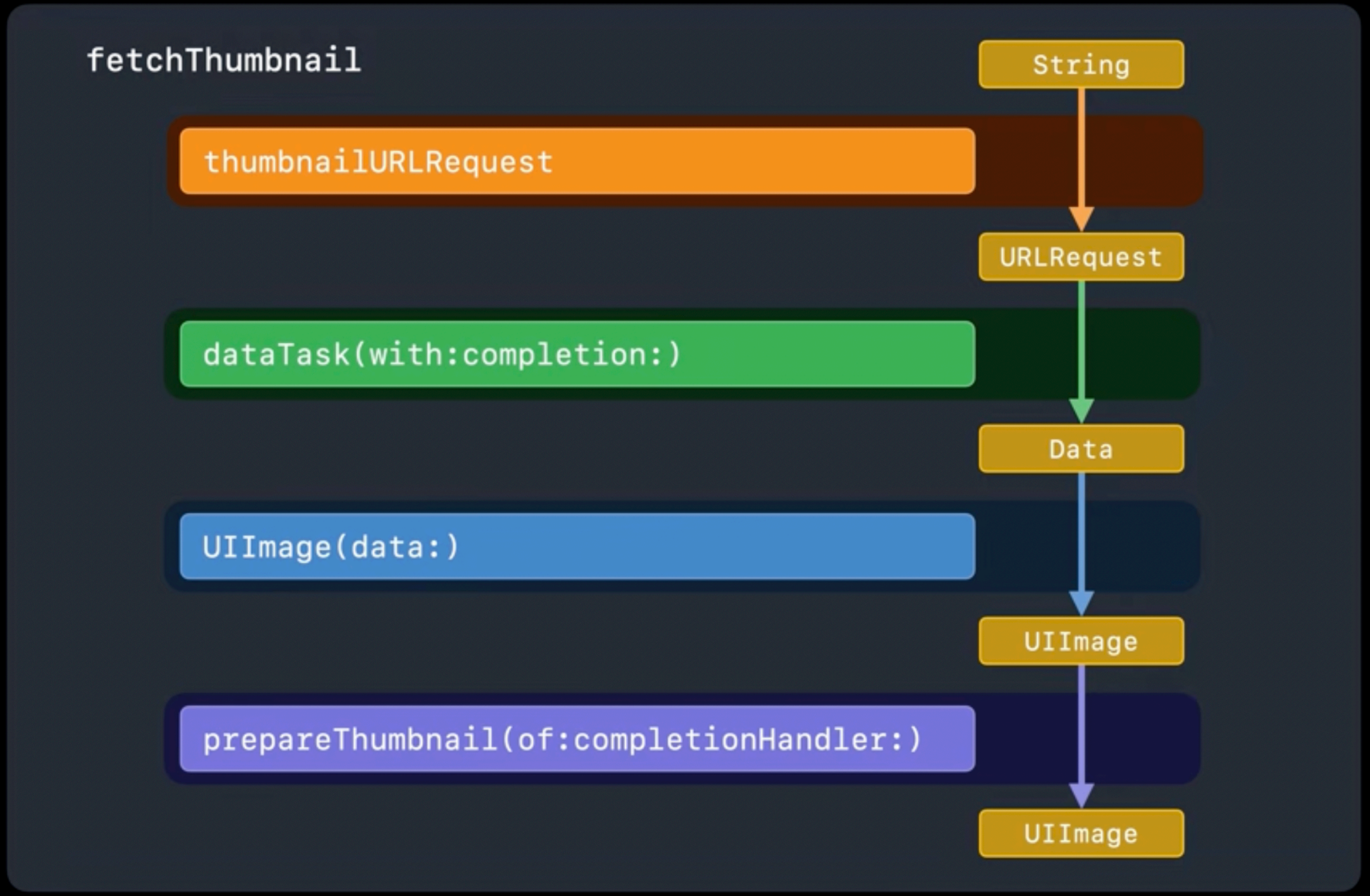
DispatchQueue

Nested Closures

DispatchGroup

OperationQueue

Fetching a thumbnail

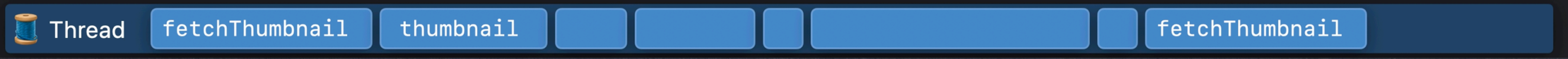


Current version

```
func fetchThumbnail(for id: String, completion: @escaping (UIImage?, Error?) -> Void) {
    let request = thumbnailURLRequest(for: id)
    let task = URLSession.shared.dataTask(with: request) { data, response, error in
        if let error = error {
            completion(nil, error)
        } else if (response as? HTTPURLResponse)?.statusCode != 200 {
            completion(nil, FetchError.badID)
        } else {
            guard let image = UIImage(data: data!) else {
                completion(nil, FetchError.badImage)
                return
            }
            image.prepareThumbnail(of: CGSize(width: 40, height: 40)) { thumbnail in
                guard let thumbnail = thumbnail else {
                    completion(nil, FetchError.badImage)
                    return
                }
                completion(thumbnail, nil)
            }
        }
    }
    task.resume()
}
```

Async / Await version

```
func fetchThumbnail(for id: String) async throws -> UIImage {
    let request = thumbnailURLRequest(for: id)
    let (data, response) = try await URLSession.shared.data(for: request)
    guard (response as? HTTPURLResponse)?.statusCode == 200 else { throw FetchError.badID }
    let maybeImage = UIImage(data: data)
    guard let thumbnail = await maybeImage?.thumbnail else { throw FetchError.badImage }
    return thumbnail
}
```



Current version

```
func getUser(completionHandler: @escaping (_ response: UserResponse?, _ error: Error?) -> Void) {
    let url = URL(string: "https://reqres.in/api/users/2")

    URLSession.shared.dataTask(with: url!) { data, response, error in
        do {
            guard let data = data, error == nil, let statusCode = (response as? HTTPURLResponse)?.statusCode else {
                completionHandler( nil, error)
                return
            }

            if (200...299).contains(statusCode) {
                let delayedResponse = try JSONDecoder().decode(UserResponse.self, from: data)
                print(delayedResponse)
                completionHandler(delayedResponse, nil)
            }
        } catch {
            completionHandler( nil, error)
            return
        }
    }.resume()
}
```

Async / Await version

```
func getUser() async throws -> UserResponse {
    do {
        let url = URL(string: "https://reqres.in/api/users/2")
        let (userResponse, response) = try await URLSession.shared.data(from: url!)

        guard let statusCode = (response as? HTTPURLResponse)?.statusCode else {
            throw URLError(.badURL)
        }
        if (200...299).contains(statusCode) {
            return try JSONDecoder().decode(UserResponse.self, from: userResponse)
        }
    } catch {
        throw URLError(.badURL)
    }
}
```

Issues with Existing solutions

Unknowingly creating Strong References

Forget to call closures and calling closures multiple times

Compromising in Code Readability & Code Maintainability

Difficult to understand and handle Errors

Missing resume() call

Introduction

- Async - Await is introduced with Swift 5.5 in WWDC 2021
- As it allows structured concurrency which improves readability of complex asynchronous code
- It works with Xcode 13.2, iOS 13+ and Swift 5.5

Concepts

- Async / Await
- Task
- Task Cancellation
- Task.detached
- Task.sleep and Task.yield
- Async let
- Task Group
- Async Sequence
- Continuations
- Refactoring existing code

Async

Async enables a function to suspend to perform some long running operation asynchronously

```
func fetchUser() async -> User {
    // Perfrom some asynchronous operation
    return User()
}


func fetchUser() async throws -> User {
    // Perfrom some asynchronous operation
    return User()
}
```

Await

Await marks an async function may suspend execution

Await suspends the execution and waits for resume the task once done while running other synchronous code

```
func callFetchUser() async {
    let user = await fetchUser()
}
```

Converting Existing Methods

```swift
func fetchUser(completionHandler: @escaping ((Result<User, Error>) -> Void)) {
    let url = URL(string: "https://reqres.in/api/users/2")!
    URLSession.shared.dataTask(with: url) { data, response, error in
        do {
            let user = try JSONDecoder().decode(User.self, from: data!)
            completionHandler(.success(user))
        } catch {
            completionHandler(.failure(error))
        }
    } .resume()
}

func fetchUser() async throws -> User {
    do {
        let url = URL(string: "https://reqres.in/api/users/2")
        let (userResponse, _) = try await URLSession.shared.data(from: url!)
        let user = try JSONDecoder().decode(User.self, from: userResponse)
        return user
    } catch {
        throw error
    }
}
```

Task

- Task provides async context for executing code concurrently
- Task represents an unit of asynchronous code
- Multiple Asynchronous operations in a Task runs sequentially

Example

Synchronous Context
Line 1
Line 2
...
...
...
Task { (Provides Asynchronous Context)
Await method
}
...
...
Line n

How multiple Tasks work?

```
func test() {
    T1 (async 1)
    T2 (async 2)
    T3 (async 3)
    T4 (async 4)
}
```

How multiple async methods works in a Task?

```
func test() {
    T1 {
        async 1
        async 2
        async 3
        async 4
    }
}
```

Code which is wrapped inside a Task executes sequentially

Task Priorities

```
public struct TaskPriority : RawRepresentable, Sendable {

            public static let high: TaskPriority

            public static var medium: TaskPriority { get }

            public static let low: TaskPriority

            public static let userInitiated: TaskPriority

            public static let utility: TaskPriority

            public static let background: TaskPriority

            @available(*, deprecated, renamed: "medium")
            public static let `default`: TaskPriority
}
```

```
Task(priority: .high) {
    await asyncOperation()
}
```

Task.detached()

In nested tasks, the inner tasks inherits the priorities of Outer Tasks

Task.detached ignores the outer Task priorities and performs independently in nested Tasks

Try Avoid using Task.detached()

Don’t use a detached task if it’s possible to model the operation using structured concurrency features like child tasks. Child tasks inherit the parent task’s priority and task-local storage, and canceling a parent task automatically cancels all of its child tasks. You need to handle these considerations manually with a detached task.

You need to keep a reference to the detached task if you want to cancel it by calling the `Task.cancel()` method. Discarding your reference to a detached task doesn’t implicitly cancel that task, it only makes it impossible for you to explicitly cancel the task.

Task.detached()

```
Task(priority: .userInitiated) {

    Task {
        // .userInitiated Priority
    }

    Task.detached {
        // nil / default priority
    }
}
```

Async let

Async let represents an async operation

Async let allows to perform multiple tasks at the same time and waits for result of all operations at once

Async let is quicker compared to running Tasks sequentially

Async let tasks run Parallel

Non Async Let

```
let image1 = try await HTTPClient.shared.loadImage(number: 1)
let image2 = try await HTTPClient.shared.loadImage(number: 2)
let image3 = try await HTTPClient.shared.loadImage(number: 3)
let image4 = try await HTTPClient.shared.loadImage(number: 4)

loadingIndicator.stopAnimating()
imageView1.image = image1
imageView2.image = image2
imageView3.image = image3
imageView4.image = image4
```

Each method waits for the other method to finish

Async let

```
@IBAction func fetchAllAtOnce() {
    Task {
        async let image6 = HTTPClient.shared.loadImage(number: 6)
        async let image7 = HTTPClient.shared.loadImage(number: 7)
        async let image8 = HTTPClient.shared.loadImage(number: 8)
        async let image9 = HTTPClient.shared.loadImage(number: 9)

        let images = await [try image6, try image7, try image8, try image9]

        print("All images fetched!")
        imageView6.image = images[0]
        imageView7.image = images[1]
        imageView8.image = images[2]
        imageView9.image = images[3]
    }
}
```

Note: We can use different return typed async methods

Task Cancellations

A task can be cancelled by calling cancel() method on task instance.

```
taskInstance.cancel()
```

Tasks which are part of .task modifier are auto cancelled

```
.task {
  //Auto cancels when view disappears in SwiftUI
}
```

Cancelling a parent task automatically cancels all child tasks except detached tasks

Detached tasks must be cancelled annually, so try avoid using detachedTasks.

Group Tasks can be cancelled by calling `group.cancelALL()`

TaskGroup

You can see a Task Group as a container of several child tasks that are dynamically added.

Child tasks can run in parallel or in serial, but the Task Group will only be marked as finished once all its child tasks are finished

Tasks in the TaskGroup runs concurrently, so we can not predict the order of execution

Task Group is faster than tasks and async let

Task Group Limitations

All async functions in the Task Group should return the same Type

Ex:

Collection of Images
Collection of Users

Task Group

```
let images = TaskGroup {
    getImage() -> UIImage
    getImage() -> UIImage
    getImage() -> UIImage
    getImage() -> UIImage
}
```

Delayed Task

```
await Task.sleep(nanoseconds:
2_000_000_0000)
```

Execution waits for the given number of nanoseconds

Voluntarily Suspend Task

await Task.yield()

Task.yield() suspends the current task for some nanoseconds and gives chances to run other waiting tasks.

```
func playWithTasks() {
    Task {
        for i in 1...10000 {
            print("1 ------ Task -------- \(i)")
            if i % 1000 == 0 {
                await Task.yield()
            }
        }
    }

    Task {
        for i in 1...100 {
            print("2 ------ Task -------- \(i)")
        }
    }
}
```

When to use yield()

if you're writing computationally intensive code, you should call Task.yield() So that the system gives opportunity to perform any other work, such as updating the UI.

If you don't do this, your app might appear to be frozen to the user, even though it is actively running some computationally intensive code.

Refactoring

The Swift community strongly recommends using async-await construct wherever possible

Apple has already provided async-await variants of many of their existing APIs

Use Continuations and Async Alternatives to convert for non converted code

Continuations

Swift provides a few methods to convert callback-based code / Delegate based code into async/await

One of the ways to migrate is using

1. withCheckedContinuation

2. withCheckedThrowingContinuation
(code that throws errors)

Continuations

```swift
// Callback based code
func fetchUser(completionHandler: @escaping ((Result<User, Error>) -> Void)) {
    let url = URL(string: "https://reqres.in/api/users/2")!
    URLSession.shared.dataTask(with: url) { data, response, error in
        do {
            let user = try JSONDecoder().decode(User.self, from: data!)
            completionHandler(.success(user))
        } catch {
            completionHandler(.failure(error))
        }
    } .resume()
}

// Migrated Code
func fetchUser() async throws -> User {
    let url = URL(string: "https://reqres.in/api/users/2")!
    return try await withCheckedThrowingContinuation { continuation in
        URLSession.shared.dataTask(with: url) { data, response, error in
            do {
                let user = try JSONDecoder().decode(User.self, from: data!)
                continuation.resume(with: .success(user))
            } catch {
                continuation.resume(with: .failure(error))
            }
        } .resume()
    }
}
```

Refactoring - to Async

```
// Existing way of asynchronous API Calling
func fetchU                                    ing ((Result<User, Error>) -> Void)) {
    let url                                    s.in/api/users/2")!
    URLSess                                    ) { data, response, error in
        do

                                               .decode(User.self, from: data!)

        } c

        }
    } .resu
}
```

Continuations

Continuation is the resume point when asynchronous method returned with result

Which should be called only once for the path

Refactor - Async Alternative

```swift
@available(*, renamed: "fetchUser()")
func fetchUser(completionHandler: @escaping ((Result<User, Error>) -> Void)) {
    Task {
        do {
            let result = try await fetchUser()
            completionHandler(.success(result))
        } catch {
            completionHandler(.failure(error))
        }
    }
}

func fetchUser() async throws -> User {
    let url = URL(string: "https://reqres.in/api/users/2")!
    return try await withCheckedThrowingContinuation { continuation in
        URLSession.shared.dataTask(with: url) { data, response, error in
            do {
                let user = try JSONDecoder().decode(User.self, from: data!)
                continuation.resume(with: .success(user))
            } catch {
                continuation.resume(with: .failure(error))
            }
        } .resume()
    }
}
```

Refactor - Async Wrapper

```swift
@available(*, renamed: "fetchUser()")
func fetchUser(completionHandler: @escaping ((Result<User, Error>) -> Void)) {
    let url = URL(string: "https://reqres.in/api/users/2")!
    URLSession.shared.dataTask(with: url) { data, response, error in
        do {
            let user = try JSONDecoder().decode(User.self, from: data!)
            completionHandler(.success(user))
        } catch {
            completionHandler(.failure(error))
        }
    } .resume()
}

func fetchUser() async throws -> User {
    return try await withCheckedThrowingContinuation { continuation in
        fetchUser() { result in
            continuation.resume(with: result)
        }
    }
}
```

Refactor - Current Delegate Based

```
protocol ExpensiveOperationPerformable: AnyObject {
    func didSucceed(with output: String)
    func didFail(with error: Error)
}

class ViewController: UIViewController {
    override func viewDidLoad() {
        super.viewDidLoad()

        performExpensiveOperation(id: 100)
    }

    func performExpensiveOperation(id: Int) {
        let thirdPartyUtility = ThirdPartyUtility()
        thirdPartyUtility.delegate = self
        thirdPartyUtility.performOperation(with: id)
    }
}



extension ViewController: ExpensiveOperationPerformable {
    func didSucceed(with output: String) {
        print("Succeeded with output \(output)")
    }

    func didFail(with error: Error) {
        print("Failed with error \(error.localizedDescription)")
    }
}
```

Refactor - Current Delegate Based

```
class ViewController: UIViewController {

    var activeContinuation: CheckedContinuation<String, Error>?

    override func viewDidLoad() {
        super.viewDidLoad()

        Task(priority: .medium) {
            do {
                let result = try await performExpensiveOperation(id: 100)
            } catch {
                print("Error Occurred \(error.localizedDescription)")
            }
        }
    }

    func performExpensiveOperation(id: Int) async throws -> String {
        let thirdPartyUtility = ThirdPartyUtility()

        return try await withCheckedThrowingContinuation { continuation in
            self.activeContinuation = continuation
            thirdPartyUtility.delegate = self
            thirdPartyUtility.performOperation(with: id)
        }
    }
}
```

```
extension ViewController: ExpensiveOperationPerformable {
    func didSucceed(with output: String) {
        activeContinuation?.resume(returning: output)
    }

    func didFail(with error: Error) {
        activeContinuation?.resume(throwing: error)
    }
}
```

Useful Resources

https://www.youtube.com/watch?v=p6q1RmYUsNU&list=PLwvDm4Vfkdphr2DL4sY4rS9PLzPdyi8PM

https://www.youtube.com/watch?v=8iBK0QxyjgU&t=202s

https://developer.apple.com/videos/play/wwdc2021/10134/

https://developer.apple.com/videos/play/wwdc2021/10132/

https://developer.apple.com/videos/play/wwdc2021/10058/

https://developer.apple.com/videos/play/wwdc2021/10133/

https://deepu.tech/concurrency-in-modern-languages/

Thanks