UTOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Swift Combine

Programme

1. RxSwift

- Les Observables
- Les subjects
- Les Filtering operators
- Les Transforming Operators

Programme

2. Introduction à Combine

- Les éditeurs (Publishers) et les abonnés (Subscribers)
- Les opérateurs de transformation de flux
- La gestion des erreurs avec "error handling"
- Utilisation de "Schedulers"

RxSwift

Les Observables

Dans RxSwift, un Observable est un type qui représente une séquence d'événements. Un Observable peut émettre zéro, un ou plusieurs événements au fil du temps. Les observables sont le cœur de la programmation réactive, car ils permettent de créer des flux de données et de les manipuler facilement en chaînant des opérations.

RxSwift

Les Observables

```
import RxSwift

// Créer un Observable
let numbersObservable: Observable<Int> = Observable.of(1, 2, 3, 4, 5)

// S'abonner à l'Observable et manipuler les données
let disposeBag = DisposeBag()

numbersObservable
    .map { number in
        return number * 2
    }
    .subscribe(onNext: { doubledNumber in
        print("Le nombre doublé est :", doubledNumber)
    }, onCompleted: {
        print("La séquence est terminée.")
    })
    .disposed(by: disposeBag)
```

RxSwift

Les Observables

- just : Crée un Observable qui émet une seule valeur et se termine
- of : Crée un Observable à partir d'une liste de valeurs
- from : Crée un Observable à partir d'un tableau, d'un dictionnaire ou d'un ensemble
- empty : Crée un Observable qui ne produit aucun élément et se termine immédiatement.
- never : Crée un Observable qui ne produit aucun élément et ne se termine jamais
- error : Crée un Observable qui termine immédiatement avec une erreur
- interval : Crée un Observable qui émet des éléments à intervalles réguliers.
- timer : Crée un Observable qui émet une valeur après un délai spécifié
- range : Crée un Observable qui émet une séquence de valeurs dans une plage spécifiée
- create : Crée un Observable personnalisé en fournissant une fonction qui définit son comportement
- deferred : Crée un Observable qui ne génère la séquence qu'au moment de l'abonnement

RxSwift

Les Subjects

Dans RxSwift, les Subject sont à la fois des observables et des observateurs. Ils peuvent émettre des événements et également réagir aux événements qui leur sont envoyés. Il existe quatre types de subjects dans RxSwift: PublishSubject, BehaviorSubject, ReplaySubject et BehaviorRelay.

RxSwift

Les Subjects

- PublishSubject: Il émet uniquement les événements qui se produisent après qu'un observateur s'est abonné. Les événements précédents ne sont pas reçus par les nouveaux abonnés

```
import RxSwift

let publishSubject = PublishSubject<String>()

publishSubject.onNext("Hello") // Ne sera pas reçu par l'observateur

let disposeBag = DisposeBag()

publishSubject.subscribe(onNext: { value in
    print("PublishSubject:", value)
}).disposed(by: disposeBag)

publishSubject.onNext("World") // Affiche : PublishSubject: World
```

RxSwift

Les Subjects

- BehaviorSubject: Il émet l'événement le plus récent à tout nouvel observateur au moment de l'abonnement et tous les événements ultérieurs

```
import RxSwift

let behaviorSubject = BehaviorSubject<String>(value: "Initial")

behaviorSubject.onNext("Hello")

let disposeBag = DisposeBag()

behaviorSubject.subscribe(onNext: { value in
    print("BehaviorSubject:", value)
}).disposed(by: disposeBag)

behaviorSubject.onNext("World") // Affiche : BehaviorSubject: World
```

RxSwift

Les Subjects

- ReplaySubject: Il émet un nombre spécifié d'événements précédents aux nouveaux observateurs et tous les événements ultérieurs.

```
import RxSwift

let replaySubject = ReplaySubject<String>.create(bufferSize: 2)

replaySubject.onNext("Event 1")
replaySubject.onNext("Event 2")
replaySubject.onNext("Event 3")

let disposeBag = DisposeBag()

replaySubject.subscribe(onNext: { value in
    print("ReplaySubject:", value)
}).disposed(by: disposeBag)

replaySubject.onNext("Event 4") // Affiche : ReplaySubject: Event 4
```

RxSwift

Les Subjects

- BehaviorRelay: Il ne permet pas d'émettre des erreurs ou de compléter l'événement. Il est utilisé pour stocker l'état dans les modèles de vue.

```
import RxCocoa
import RxSwift

let behaviorRelay = BehaviorRelay<String>(value: "Initial")

behaviorRelay.accept("Hello")

let disposeBag = DisposeBag()

behaviorRelay.subscribe(onNext: { value in
    print("BehaviorRelay:", value)
}).disposed(by: disposeBag)

behaviorRelay.accept("World") // Affiche : BehaviorRelay: World
```

RxSwift

Les Filtering operators

Les opérateurs de filtrage dans RxSwift sont utilisés pour sélectionner ou filtrer les éléments émis par un Observable en fonction de certaines conditions. Ils permettent de transformer ou de filtrer les séquences en fonction de vos besoins

RxSwift

Les Filtering operators

- filter: Cet opérateur émet uniquement les éléments qui correspondent à une condition spécifiée

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .filter { $0 % 2 == 0 }
    .subscribe(onNext: { value in
        print("Filter:", value) // Affiche uniquement les nombres pairs
    })
    .disposed(by: disposeBag)
```

RxSwift

Les Filtering operators

- distinctUntilChanged: Cet opérateur supprime les éléments consécutifs identiques

```
import RxSwift

let disposeBag = DisposeBag()
let values = Observable.of(1, 1, 2, 2, 3, 3, 4, 4)

values
    .distinctUntilChanged()
    .subscribe(onNext: { value in
        print("DistinctUntilChanged:", value) // Affiche : 1, 2, 3, 4
    })
    .disposed(by: disposeBag)
```

RxSwift

Les Filtering operators

- take: Cet opérateur prend un nombre spécifié d'éléments à partir du début de la séquence

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .take(3)
    .subscribe(onNext: { value in
        print("Take:", value) // Affiche : 1, 2, 3
    })
    .disposed(by: disposeBag)
```

RxSwift

Les Filtering operators

- skip: Cet opérateur ignore un nombre spécifié d'éléments au début de la séquence

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .skip(2)
    .subscribe(onNext: { value in
        print("Skip:", value) // Affiche : 3, 4, 5
    })
    .disposed(by: disposeBag)
```

RxSwift

Les Filtering operators

- takeWhile: Cet opérateur prend les éléments tant qu'ils correspondent à une condition spécifiée

```
import RxSwift

let disposeBag = DisposeBag()
let numbers = Observable.of(1, 2, 3, 4, 5)

numbers
    .takeWhile { $0 < 4 }
    .subscribe(onNext: { value in
        print("TakeWhile:", value) // Affiche : 1, 2, 3
    })
    .disposed(by: disposeBag)
```

RxSwift

Les Transforming Operators

Les opérateurs de transformation (Transforming Operators) dans RxSwift sont utilisés pour transformer ou modifier les événements émis par un observable. Ils permettent de changer la forme ou la structure des événements pour mieux s'adapter à vos besoins.

RxSwift

Les Transforming Operators

- map : Transforme chaque événement en appliquant une fonction de transformation

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
    .map { number in
        return number * 2
    }
    .subscribe(onNext: { doubledNumber in
        print("Transforming Operator - Map:", doubledNumber)
    })
    .disposed(by: disposeBag)

// Output: Transforming Operator - Map: 2
//         Transforming Operator - Map: 4
//         Transforming Operator - Map: 6
//         Transforming Operator - Map: 8
//         Transforming Operator - Map: 10
```

RxSwift

Les Transforming Operators

- flatMap : Transforme chaque événement en un observable et fusionne les observables résultants en une seule séquence

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
    .flatMap { number in
        return Observable.of(number, number * 2, number * 3)
    }
    .subscribe(onNext: { transformedNumber in
        print("Transforming Operator - FlatMap:", transformedNumber)
    })
    .disposed(by: disposeBag)
```

RxSwift

Les Transforming Operators

- scan : Applique une opération à chaque événement et émet le résultat intermédiaire à chaque étape

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
    .scan(0) { accumulated, next in
        return accumulated + next
    }
    .subscribe(onNext: { accumulatedSum in
        print("Transforming Operator - Scan:", accumulatedSum)
    })
    .disposed(by: disposeBag)

// Output: Transforming Operator - Scan: 1
//         Transforming Operator - Scan: 3
//         Transforming Operator - Scan: 6
//         Transforming Operator - Scan: 10
//         Transforming Operator - Scan: 15
```

RxSwift

Les Transforming Operators

- buffer : Rassemble les événements émis par l'observable dans des tableaux et les émet une fois que le tableau est plein

```
import RxSwift

let numbersObservable = Observable.of(1, 2, 3, 4, 5)

numbersObservable
.buffer(timeSpan: .seconds(3), count: 2, scheduler: MainScheduler.instance)
.subscribe(onNext: { numbersArray in
print("Transforming Operator - Buffer:", numbersArray)
})
.disposed(by: disposeBag)

// Output: Transforming Operator - Buffer: [1, 2]
// Transforming Operator - Buffer: [3, 4]
// Transforming Operator - Buffer: [5]
```

RxSwift

Les Transforming Operators

- groupBy : Divise l'observable en sous-observables en fonction d'une clé.

```
import RxSwift

let wordsObservable = Observable.of("apple", "banana", "cherry", "avocado", "blueberry")

wordsObservable
    .groupBy { word in
        return word.count
    }
    .flatMap { group in
        return group.toArray()
    }
    .subscribe(onNext: { wordsArray in
        print("Transforming Operator - GroupBy:", wordsArray)
    })
    .disposed(by: disposeBag)

// Output: Transforming Operator - GroupBy: ["apple"]
```

Introduction à Combine

- Le framework Combine est basé sur le concept de programmation réactive. Il permet de manipuler des flux de données, appelés publishers, qui émettent des événements, tels que des valeurs, des erreurs ou la fin de la séquence. Ces publishers peuvent être modifiés à l'aide d'opérateurs, tels que map, filter ou reduce, pour produire des valeurs transformées.
- Les valeurs produites peuvent ensuite être transmises à des abonnés, qui peuvent s'inscrire aux publishers à l'aide de la méthode sink. Les abonnés peuvent également être des publishers eux-mêmes, permettant la création de chaînes complexes de flux de données.
- En plus des publishers et des abonnés, le framework Combine fournit également une variété d'autres types pour aider à gérer les flux de données, tels que Subject, qui permet de créer des publishers personnalisés, et Cancellable, qui permet d'annuler des abonnements.
- Le framework Combine est très utile pour les tâches asynchrones dans les applications iOS/macOS, telles que les appels réseau, les accès à la base de données ou les événements d'interface utilisateur. Il permet de gérer de manière élégante les flux de données asynchrones et de les traiter de manière déclarative, ce qui peut rendre le code plus clair, plus concis et plus facile à maintenir.

Combine

Publishers (éditeurs)

- les Publishers (éditeurs) sont des objets qui émettent des valeurs au fil du temps
- Les Publishers dans Combine peuvent être de plusieurs types, tels que les publishers de données (Data Publishers), les publishers de notifications (Notification Publishers), les publishers de temporisation (Timer Publishers) et les publishers personnalisés (Custom Publishers)

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

numbersPublisher
    .map { $0 * 2 }
    .sink { print($0) }
```

Combine

Subscribers (abonnés)

- Les Subscribers sont des objets qui reçoivent et traitent les valeurs émises par les publishers. Les abonnés sont créés à l'aide de la méthode sink, qui permet de spécifier une ou plusieurs closures pour traiter les valeurs produites.

```
import Combine
let numbersPublisher = [1, 2, 3].publisher()

let numbersSubscriber = Subscribers.Sink<Int, Never>(receiveValue: { value in
    print(value)
})

numbersPublisher.subscribe(numbersSubscriber)
```

Combine

Publishers - Publishers de données

- Publishers de données (Data Publishers) : Ce type de publisher émet une séquence de valeurs, telles que des nombres, des chaînes de caractères, des tableaux, etc. Les publishers de données sont créés à l'aide de la méthode publisher() ou de la méthode Just().

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

numbersPublisher
    .map { $0 * 2 }
    .sink { print($0) }
```

Combine

Publishers - Publishers de notifications

- Publishers de notifications (Notification Publishers) : Ce type de publisher émet des notifications, telles que des notifications de fin de vie d'application, de changement de langue, etc. Les publishers de notifications sont créés à l'aide des notifications NotificationCenter.

```
import Combine

let notificationPublisher = NotificationCenter.default.publisher(for: UIApplication.didEnterBackgroundNotification)

notificationPublisher
    .sink { notification in
        print("Application did enter background")
    }
```

Combine

Publishers - Publishers de temporisation

- Publishers de temporisation (Timer Publishers) : Ce type de publisher émet des événements à des intervalles réguliers, tels que des timers. Les publishers de temporisation sont créés à l'aide de la méthode Timer.publish().

```
import Combine

let timerPublisher = Timer.publish(every: 1.0, on: .main, in: .common)

timerPublisher
    .sink { _ in
        print("Timer fired")
    }
```

Combine

Publishers - Publishers personnalisés

- Publishers personnalisés (Custom Publishers) : Ce type de publisher permet de créer des publishers personnalisés en conformité avec le protocole Publisher. Les publishers personnalisés peuvent être créés pour émettre des valeurs spécifiques ou pour se connecter à des sources de données personnalisées.

```
import Combine

class MyCustomPublisher: Publisher {
    typealias Output = String
    typealias Failure = Never

    func receive<S>(subscriber: S) where S : Subscriber, Failure == S.Failure, Output == S.Input {
        subscriber.receive("Hello, world!")
        subscriber.receive(completion: .finished)
    }
}
let customPublisher = MyCustomPublisher()

customPublisher
    .sink { print($0) }
```

Combine

Subscribers - Sink

- Sink : Le subscriber Sink est utilisé pour recevoir des valeurs émises par un publisher, et effectuer une ou plusieurs opérations sur ces valeurs.

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

let numbersSubscriber = Subscribers.Sink<Int, Never>(receiveValue: { value in
    print(value)
})

numbersPublisher.subscribe(numbersSubscriber)
```

Combine

Subscribers - Assign

- Assign : Le subscriber Assign est utilisé pour assigner les valeurs émises par un publisher à une propriété d'un objet.

```
import Combine

class MyClass {
    var myValue: Int = 0
}

let myClass = MyClass()
let numbersPublisher = [1, 2, 3].publisher()

numbersPublisher.assign(to: \.myValue, on: myClass)
```

Combine

Subscribers - Future

- Future : Le subscriber Future est utilisé pour recevoir une seule valeur émise par un publisher, et retourner une valeur de sortie future.

```
import Combine

let numbersPublisher = Just(42)

let future = numbersPublisher
    .delay(for: .seconds(1), scheduler: DispatchQueue.main)
    .map { $0 * 2 }
    .future()

print(future.result ?? "No result yet")
```

Combine

Subscribers - AnySubscriber

- AnySubscriber : Le subscriber AnySubscriber permet de créer un subscriber générique pour recevoir des valeurs de n'importe quel type.

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

let anySubscriber = AnySubscriber<Int, Never>(
    receiveSubscription: { subscription in
        subscription.request(.unlimited)
    },
    receiveValue: { value -> Subscribers.Demand in
        print(value)
        return .unlimited
    },
    receiveCompletion: { completion in
        print(completion)
    }
)
numbersPublisher.subscribe(anySubscriber)
```

Combine

Subscribers - PassthroughSubject

- PassthroughSubject : Le subscriber PassthroughSubject est utilisé pour créer un publisher personnalisé qui émet des valeurs, et pour recevoir des valeurs émises par ce publisher.

```
import Combine

let passthroughSubject = PassthroughSubject<String, Never>()

passthroughSubject
    .sink { value in
        print("Received value: \(value)")
    }

passthroughSubject.send("Hello, world!")
```

Combine

Les opérateurs de transformation de flux

Les opérateurs de transformation de flux dans Combine sont des fonctions qui prennent un ou plusieurs éditeurs (Publishers) en entrée et renvoient un nouvel éditeur en sortie. Ces opérateurs transforment les flux d'événements en appliquant des opérations telles que le filtrage, le mappage, le regroupement et la combinaison. Voici quelques-uns des opérateurs de transformation de flux les plus couramment

Combine

Les opérateurs de transformation de flux

- map: L'opérateur map transforme chaque élément de l'éditeur en appliquant une fonction à l'élément. Il renvoie un nouvel éditeur avec les éléments transformés

```
let numbers = [1, 2, 3, 4, 5]
let publisher = numbers.publisher
let mappedPublisher = publisher.map { $0 * 2 }
mappedPublisher.sink { value in
    print(value) // Affiche 2, 4, 6, 8, 10
}
```

Combine

Les opérateurs de transformation de flux

- filter: L'opérateur filter élimine certains éléments de l'éditeur en appliquant une fonction de filtrage à chaque élément. Il renvoie un nouvel éditeur contenant uniquement les éléments qui satisfont la condition de filtrage.

```
let numbers = [1, 2, 3, 4, 5]
let publisher = numbers.publisher
let filteredPublisher = publisher.filter { $0 % 2 == 0 }
filteredPublisher.sink { value in
    print(value) // Affiche 2, 4
}
```

Combine

Les opérateurs de transformation de flux

- flatMap: L'opérateur flatMap transforme chaque élément de l'éditeur en un ou plusieurs nouveaux éditeurs en appliquant une fonction à l'élément. Les éditeurs générés sont ensuite combinés en un seul éditeur. Il renvoie un nouvel éditeur contenant les éléments combinés.

```
let numbers = [1, 2, 3, 4, 5]
let publisher = numbers.publisher
let flatMapPublisher = publisher.flatMap { value in
    return [value, value * 2, value * 3].publisher
}
flatMapPublisher.sink { value in
    print(value) // Affiche 1, 2, 3, 2, 4, 6, 3, 6, 9, 4, 8, 12, 5, 10, 15
}
```

Combine

Les opérateurs de transformation de flux

- scan: L'opérateur scan combine les éléments successifs de l'éditeur en appliquant une fonction d'accumulateur. Il renvoie un nouvel éditeur qui émet les résultats intermédiaires de l'accumulateur.

```
let numbers = [1, 2, 3, 4, 5]
let publisher = numbers.publisher
let scanPublisher = publisher.scan(0) { accumulator, value in
    return accumulator + value
}
scanPublisher.sink { value in
    print(value) // Affiche 1, 3, 6, 10, 15
}
```

Combine

Les opérateurs de transformation de flux

- merge: L'opérateur merge combine plusieurs éditeurs en un seul éditeur qui émet des éléments de tous les éditeurs sources. Il renvoie un nouvel éditeur qui combine les éléments de tous les éditeurs sources.

```
let publisher1 = [1, 2, 3].publisher
let publisher2 = [4, 5, 6].publisher
let mergedPublisher = Publishers.Merge(publisher1, publisher2)
mergedPublisher.sink { value in
    print(value) // Affiche 1, 2, 3, 4, 5, 6
}
```

Combine

Les opérateurs de transformation de flux

- combineLatest: L'opérateur combineLatest combine les derniers éléments de plusieurs éditeurs en appliquant une fonction à ces éléments. Il renvoie un nouvel éditeur qui émet les résultats de la fonction appliquée aux derniers éléments de tous les éditeurs.

```
let publisher1 = [1, 2, 3].publisher
let publisher2 = [4, 5, 6].publisher
let combinedPublisher = Publishers.CombineLatest(publisher1, publisher2)
    .map { $0 + $1 }
combinedPublisher.sink { value in
    print(value) // Affiche 5, 6, 7, 6, 7, 8, 7, 8, 9
}
```

Combine

La gestion des erreurs

La gestion des erreurs (error handling) dans le framework Combine permet de gérer les erreurs émises par les publishers lors de la production des valeurs. Les erreurs peuvent être générées par des erreurs de réseau, des erreurs d'accès à la base de données, des erreurs de parsing, etc. Pour gérer les erreurs, le framework Combine propose plusieurs opérateurs.

Combine

La gestion des erreurs

- catch: Cet opérateur permet de remplacer une erreur par une autre valeur ou par un nouveau publisher.

```
import Combine
enum CustomError: Error {
    case networkError
    case parsingError
}
let numbersPublisher = [1, 2, 3].publisher()
numbersPublisher
    .tryMap { value -> Int in
        guard value != 2 else {
            throw CustomError.parsingError
        }
        return value
    }
    .catch { error -> Just<Int> in
        if let customError = error as? CustomError {
            switch customError {
            case .parsingError:
                return Just(0)
            default:
                return Just(1)
            }
        }
        return Just(2)
    }
    .sink { print($0) }
```

Combine

La gestion des erreurs

- retry: Cet opérateur permet de retenter la production des valeurs en cas d'erreur.

```
import Combine
enum CustomError: Error {
    case networkError
}
var counter = 0
let numbersPublisher = PassthroughSubject<Int, CustomError>()
let subscription = numbersPublisher
    .tryMap { value -> Int in
        guard value != 2 else {
            throw CustomError.networkError
        }
        return value
    }
    .retry(2)
    .sink(
        receiveCompletion: { completion in
            print(completion)
        },
        receiveValue: { value in
            print(value)
        }
    )
numbersPublisher.send(1)
numbersPublisher.send(2)
numbersPublisher.send(3)
```

Combine

La gestion des erreurs

- replaceError: Cet opérateur permet de remplacer toutes les erreurs par une autre valeur.

```
import Combine

enum CustomError: Error {
    case networkError
}

let numbersPublisher = [1, 2, 3].publisher()

numbersPublisher
    .tryMap { value -> Int in
        guard value != 2 else {
            throw CustomError.networkError
        }
        return value
    }
    .replaceError(with: 0)
    .sink { print($0) }
```

Combine

Schedulers

Les "schedulers" dans le framework Combine sont utilisés pour définir la façon dont les opérations sont exécutées, notamment pour définir l'ordonnancement des tâches et la file d'attente d'exécution. Le framework Combine fournit plusieurs schedulers pour les différentes tâches, comme le travail sur la file d'attente principale, l'exécution sur un thread spécifique ou un travail sur un thread background.

Combine

Schedulers

- MainScheduler : Ce scheduler est utilisé pour exécuter les opérations sur la file d'attente principale.

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

numbersPublisher
    .receive(on: DispatchQueue.main)
    .sink { print($0) }
```

Combine

Schedulers

- DispatchQueue : Ce scheduler est utilisé pour exécuter les opérations sur une file d'attente personnalisée.

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

let backgroundQueue = DispatchQueue(label: "my-background-queue")

numbersPublisher
    .subscribe(on: backgroundQueue)
    .sink { print($0) }
```

Combine

Schedulers

- OperationQueue : Ce scheduler est utilisé pour exécuter les opérations sur une file d'attente de type OperationQueue.

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

let operationQueue = OperationQueue()

numbersPublisher
    .subscribe(on: operationQueue)
    .sink { print($0) }
```

Combine

Schedulers

- ImmediateScheduler : Ce scheduler est utilisé pour exécuter les opérations immédiatement.

```
import Combine

let numbersPublisher = [1, 2, 3].publisher()

numbersPublisher
    .subscribe(on: ImmediateScheduler.shared)
    .sink { print($0) }
```