



# ברוחים הבאים לכינוס DevGeekWeek

מתחללים עוד כמה דקות



# Extreme iOS Advanced Swift Programming

- Doron Feldman & Giora Krasilshchik
- CTO & Mobile Tech Lead
- GoTech & Mixin Software | A GoTech Company
- June 21, 2021



Doron Feldman  
CTO @ GoTech



Giora Krasilshchik  
Mobile Tech Lead @ Mixin Software

# Today's Topics

---

- Structs, Classes, Enums & Other Vegetables
- Immutability
- Closures
- Error Handling
- Concurrency
- Reactive Programming
- Combine
- SwiftUI



# Structs, Classes, Enums & Other Vegetables

---

- מה הבדלים בין Class ל-**Struct**?
- متى נרצה להשתמש בכל אחד מהם?
  - Enums
  - Extension Functions

# Structs, Classes, Enums & Other Vegetables

---

## Classes vs Structs

- אפשר להגיד פונקציות ו-Properties
- לשניהם אפשר להגיד פונקציית Init
- את שניהם אפשר הרחיב עם Extensions
- יכולים ממש פרוטוקולים

# Structs, Classes, Enums & Other Vegetables

## מה שונה? - Classes vs Structs

### • אפשרי רק ב-Struct

- לא חייבים לכתוב פונקציית Init, כשמגדירים משתנים, מקבלים init במתנה. אם נגדיר משתנה var עם ערך דיפולטי, קיבל שני פונקציות init, עם ו בלי Value Type
- תמיד יהיה Comparable, זה תמיד ישווה By Value, את הערך של המשתנים

### • אפשרי רק ב-Class

- אפשרי ממש ירושה
- יש פונקציית deinit, המאפשרת לשחרר משאבים ידנית
- תמיד יהיה Reference Type
- שימוש ב-“==” מושווה בין Reference

# Structs, Classes, Enums & Other Vegetables

## מתי משתמש בכל אחד? - Classes vs Structs

- מתי משתמש ב-**Struct**
- מתי משתמש ב-**Class**
- המלצתה הרשמית של אפל ב-Swift להשתמש ב-Struct בכל מקרה אחר -Structים מציעים כמעט את כל היכולות של Class-ים והקד שלנו יהיה הרבה יותר קל למעקב הגדרה של Data Models במיוחד עבור עבודה מרובת Thread-ים
- כאשר אנחנו רוצים להשתמש בירושה או **deinit**
- שעובדים עם Objective C חיבבים לעבוד עם Class מכיוון שאין דבר כזה Objective C ב-Struct
- אם צריכים לוודא שקיים רק **Instance** אחד של האובייקט, לדוגמה ב-Singleton

# Structs, Classes, Enums & Other Vegetables

---

## Enum Swift

- ב-Swift יש ל-Enum הרבה יכולות מעבר לבסיס של הגדרה של קבוצה סגורה של אופציות ולעשות עליו Switch Case
- אפשר להגדיר Properties
- אפשר להגדיר פונקציות Instance וגם פונקציות סטטיות
- אפשר להגדיר לפקציות שימוש את ה-enum עצמו, סוג של מכונת מצבים
- אפשר להגדיר Init, Custom, כלומר להכניס לוגיקה לתוך אתחול ה-enum
- Enum יכול למש פרוטוקול
- Enum יכול להיות גנרי
- אפשר להגדיר ל-Enum Extension Methods

# Structs, Classes, Enums & Other Vegetables

## Extension Functions

- ב-Swift אפשרי להרחיב מימוש של Class או Struct ולהוסיף לו פונקציונליות מסוימת, בין אם זה Class שאנו הגדכנו בקוד שלנו ובין אם זה בرمת השפה
- יש לזה מגבלה אחת בולטת, אי אפשר בתוך Extension להגדירComputed Property חדשם, שהם לא Computed, כלומר, אפשר לתרמן אחרת את המידע, אי אפשר להוסיף מידע
- כן אפשר להגדיר Static Property שימושית משתנה בرمת ה-Class כולו

```
extension Notification.Name {  
    static let statusUpdated = Notification.Name("status_updated")  
}
```

```
NotificationCenter.default.post(name: .statusUpdated, object: nil)
```

# Structs, Classes, Enums & Other Vegetables

---

## Extension Functions - Best Practices

- שימוש של Extensions כ-Utils, בשלב זה או אחר כולנו נעשה String-Extension
- זה משאיר את הקוד שלנו נקי יותר והופך את ה-Utils שלנו לנגישים יותר לאחרים
- הפרדה של Class-ים גדולים לכמה Extensions של אותו ה-Class לפי לוגיקות שונות או מימוש Protocol-ים שונים
- הפרדה של private extension שככלו יהיה private - לשזה Protocol Extension - למעשה מימוש דיפולטי של ה-Protocol, שזה סוג של Abstract Class

# Immutability

- סוגים משתנים ב-Swift
- למה צריך לשאוף לעבוד כמה יותר עם Immutability בקוד?
- Copy On Write •
- Two Phase Initialization •

# Immutability

ב-Swift יש שלוש דרכים להגדיר משתנים:

- var - משתנה שאפשר להחליפ לו ערך אחרி הגדרתו
- let - משתנה שיאפשר להחליפ לו את הערך לאחר האיתחול
- static - משתנה שמאוחתל פעם אחת ומשותף לכל ה-Instance-ים של Class

# Immutability

למה צריך לשאוף לקוד שהוא Immutable?

- ב-Functional Programming אומרים ש-State הוא מקור כל הרשע
- קל יותר להבין את הקוד ולדבаг אותו, מכיוון שאנו יודעים ששם דבר לא משתנה מרגע הגדרתו
- Thread Safe - משתנה מסווג `let` הוא תמיד Thread Safety

# Immutability

## Copy On Write

- ברוב שפות התכנות, סוגי המשתנים מתחלקים לשני קבוצות, Value ו-Type Reference
- ברוב השפות הם למעשה Collections Reference Type Collections
- Swift היא שפה מיוחדת, פה ה-Collections הם Value Types, הסיבה לכך היא שטינה על פשوط הקוד, חלק אחר בקוד לא יכול לשנות את ה-Collection שלנו
- פה עולה השאלה, האם当我们 Assign של מערך למשתנה חדש, האם כל המערך מועתק? הרי זה בזבוני נראה

# Immutability

## Copy On Write

- מה שקרה בפועל הוא כאשרנו עושים **Assign** למשתנה חדש,  
בהתחלת שניהם מצביעים על אותו המקום בזיכרון, עד הרגע שבו  
אנחנו עושים שינוי באחד מהם ובאותו הרגע נוצרים שני מערכיים  
נפרדים
- **קוד!**

# Immutability

## Two Phase Initialization

- ב-Swift, אתחול של Instance מתבצע בשני שלבים
- בשלב הראשון, מתחלים את כל המutableים של אותו ה-Class
- בשלב השני מתחאלות כל ה-Properties והפונקציות שלנו  
מלמעלה למטה
- במקרה של ירושא, זה יקרה בסדר הבא:
  1. המutableים שלנו
  2. המutableים והפונקציות של ה-Parent Class שלנו
  3. הפונקציות של ה-Class שלנו
- קוד!

# Immutability

## Two Phase Initialization

- למה הקומפיאילר של Swift מתעלל בנו ככה?
- ב-Swift ה-Class שירוש, הוא יורש כבר מתחילה חייו, ככלומר,  
אם יש פונקציה שעושים לה Override, הוא משתמש בה כבר  
בשלב ה-Init, לכן חובה לאותחל קודם את כל המשתנים, כי  
הfonקציה יכולה לעבוד איתם פנימית
- קוד!

# Closures

---

- מה הם ?Closures
- Closures and Retain Cycle
- Structs and Closures

# Closures

---

מה הם Closures

- Closures הם בעצם פונקציות אונונימיות, בלוק קוד שאפשר להעביר אותו כפרמטר בין פונקציות, להחזיר אותו כערך החזר ולשמור אותו במשתנה אם רוצים
- Reference Type Closures •
- ישנם 4 סוגי Closures :
  - Trailing - הפרמטר האחרון בפונקציה
  - Escaping - כאשר הפונקציה שלנו עלולה להיגמר לפני ביצוע ה-Closure
  - AutoClosure - מאפשר לנו לא לשים {}

# Closures

---

## Closures and Retain Cycle

- ARC - בשמו המלא Automatic Reference Count הדרך בה מנהל ומשחרר זיכרון
- Retain Cycle - בעיה נפוצה ב-Swift, כאשר שני אובייקטים עושים reference אחד לשני ולא משחררים Retain Cycle ו-Closures
- איך מגיעים ל盟בツ (Delegates) (מכירים צזה?)
- איך נמנעים ממצב צזה?
- weak - לא נספר ועובד אותו optional
- unowned - לא נספר וסומך עליך שיהיה שם ערך (t')
- weak delegate

# Closures

---

## מה הטעיה שקיימת בשימוש ב Closures וStructs

- מה הטעיה שקיימת בשימוש ב Closures ו Structs?

```
public struct Foo {  
  
    private var _number: Int = 0  
  
    public init(signal: Signal<String, NoError>) {  
        signal.observeNext {  
            print($0)  
            self._number = 1  
        }  
    }  
}
```

# Error Handling

---

- הבסיס לטיפול בשגיאות ב-Swift
  - הגדרת שגיאות Custom
  - If at first you don't succeed try, try? try! again
  - Rethrows keyword
  - Defer keyword
- טיפול בשגיאות בסביבה אסינכרונית

# Error Handling

## הבסיס לטיפול בשגיאות ב-Swift

- ב-Swift יש פרוטוקול פשוט כדי לסמן מה היא שגיאה, הנקרא Error

```
public protocol Error {  
}
```

# Error Handling

## הבסיס לטיפול בשגיאות ב-Swift

- כל דבר יכול לשמש כ-Error ב-Swift
- פונקציה שעלולה לזרוק שגיאה, צריך לסמן במילה throws
- כדי לתפוס ולטפל ב-Error, אנחנו משתמשים ב-so, do, try ו-catc

```
extension String: Error {}

func parseInt(_ value: String) throws -> Int {
    let parsedInt = Int(value)
    guard let _ = parsedInt else {
        throw "\(value) cannot be parsed into an int"
    }
    return parsedInt!
}

do {
    try parseInt("#@!")
} catch {
    print(error)
}
```

# Error Handling

---

## הגדרת שגיאות Custom

- הדריך המומלצת להגדרת שגיאות היא באמצעות enum
- ב-enum אנחנו יכולים בקלות להפריד Error-ים לסוגים מוגדרים מראש
- אנחנו יכולים לTrap ולטפל בסוגי שגיאות שונים
- אפשר להוסיף אקסטרה מידע על השגיאה שלנו
- אנחנו יכולים להציב תנאים לוגיים בתוך ה-catch
- קוד!

# Error Handling

If at first you don't succeed try, try? try!

- מציאה לנו 3 דרכים לטפל בשגיאות Swift
  - - הדרך הכיו מקיפה עם כל האופציות, אך גם הכיו ארוכה
  - nullable - במקומות לטפל, הופך לנו את ערך ההחזר ל-`try?`
  - - מעביר את האחריות אלינו `try!`

# Error Handling

## Rethrows

- מילה שומרה של השפה אשר מצינית שהפונקציה שלנו לא זורקת שגיאה בעצמה אבל מקבלת closure שעלול לזרוק catch תחביב אותנו לעטוף את הקוד שלנו ב-`do`, `catch` או `try` רק אם ה-block שתקבל יהיה throws בעצמו

```
func logExecutionTime(block: () throws -> Void) rethrows {
    let start = DispatchTime.now()
    try block()
    let end = DispatchTime.now()
    print("\(end.uptimeNanoseconds - start.uptimeNanoseconds ) nanoseconds")
}
```

# Error Handling

## Defer

- מילה שמורה המשמשת אותנו לביצוע פעולות כאשר הפונציה מסיימת
- מתבצעת תמיד, מתפקדת כמו `finally` ברוב השפות המוכרות
- אפשר לשימושים בלוקים רבים של `defer` והם יתבצעו מהסוף להתחלה
- אפשר לגשת למשתנים בכביכול מוגדרים "אחרי" ה-`defer`!- קוד!

# Error Handling

## טיפול בשגיאות בסביבה אסינכרונית

- שימוש ב-`try`, `do`, `catch` לא עוזר לנו בעבודה `async`
- הchl מ-5 Swift יש סטנדרטי בשפה לטיפול זהה

```
// A simplified version
enum Result<Success, Failure> where Failure : Error {
    case success(Success)
    case failure(Failure)
}
```

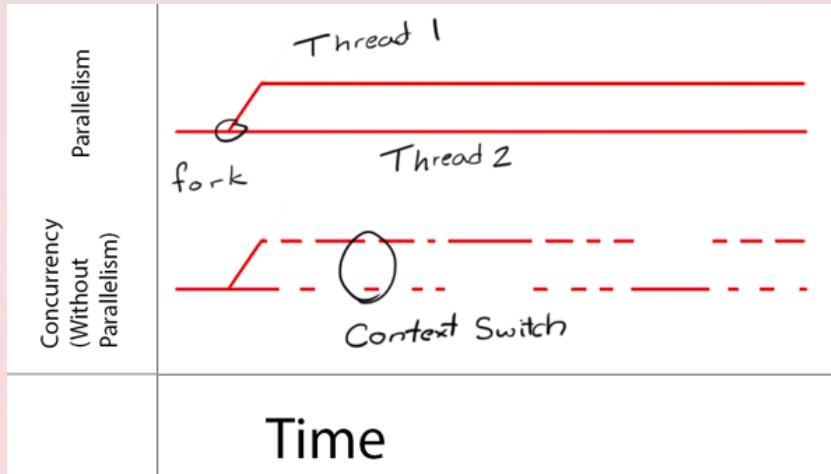
- נשתמש בו בשילוב עם `Callback` כדי להחזיר שגיאות

# Concurrency

---

- מה הם Threads ו Parallelism ,Concurrency
- מימוש Thread Concurrency באמצעות APIs
- Async ו Sync ,DispatchQueue ו GCD
- Custom DispatchQueues
- How to DispatchGroup
- Cancellation ו WorkItems
- דרכי יצרתיות לירוט לעצמכם ברגל

# Concurrency



מה הם Threads ו-Concurrency, Parallelism

- Concurrency - מקבילותות, היכולת של האפליקציה שלנו לקדם יותר משימה אחת בו זמנית, גם במקרה שבו קיים רק מעבד אחד, באמצעות Context Switch

- Parallelism - מקבילותות, ביצוע של יותר משימה אחת בו זמנית, באותו רגע, לא אפשרי אם קיים לנו רק מעבד אחד

# Concurrency

## Mימוש Concurrency באמצעות Threads

- Thread הוא יחידת עבודה שיכולה בצורה מקבילית
- אין קשר ישיר בין מספר המעבדים למספר ה-Thread-ים
- Thread ב-Swift הם הרמה הći נמוכה שאנו יכולים לשנות ולעבוד איתה
- יש הרבה כוח ושליטה ב-Thread, יש הרבה סיבוק ב-Thread
- קוד!

# Concurrency

## מימוש Concurrency באמצעות Threads

- אתגרים שיהיו לנו בעבודה ישירה עם Thread-ים:
  - צריך להתמודד עם נעלות וסנכרון בין Thread-ים
  - הקפדה על שימוש במספר אופטימלי של Thread-ים אחרת נעשה יותר Context Switch מאשר עבודה
  - שימוש יעיל במספר הליבות שזמין לנו במחשב
  - צריך לדעת להעלות ולהוריד את מספר ה-Thread-ים בהתאם לעומס המחשב כולם

# Concurrency

## ה-GCD וה-DispatchQueue

- ה-GCD של Apple מגיע להצלחה
- למשה מימוש של אפל ל-Thread Pool
- מרים מישימות במקביל בשביבנו
- עושים לו אופטימיזציה כל הזמן, משנהים את כמות ה-Thread-ים לפי כמות העומס ברמת מערכת הפעלה
- אופטימיזציה שאפשרי לעשות רק מתוך מערכת הפעלה
- הדרך שלנו להתחבר ל-GCD והוא ירים בשביבנו מישימות היא דרך DispatchQueue
  - אבסטרקציה של Apple לעבודה עם Thread-ים
  - תור שאנו חננו זורקים אליו מישימות שאנו רוצים לבצע
  - ה-GCD יודע לקחת מהתור זהה ולהריץ לפיה כל מיני פרמטרים
- קוד!

# Concurrency

```
static let userInteractive: DispatchQoS
```

The quality-of-service class for user-interactive tasks, such as animations, event handling, or updating your app's user interface.

```
static let userInitiated: DispatchQoS
```

The quality-of-service class for tasks that prevent the user from actively using your app.

```
static let default: DispatchQoS
```

The default quality-of-service class.

```
static let utility: DispatchQoS
```

The quality-of-service class for tasks that the user does not track actively.

```
static let background: DispatchQoS
```

The quality-of-service class for maintenance or cleanup tasks that you create.

```
static let unspecified: DispatchQoS
```

The absence of a quality-of-service class.

## DispatchQueue-1 GCD

- בקוד ראיינו  `DispatchQueue.global()` גלובלי, (global)  `DispatchQueue` הגלובלי יודע לקבל פרמטר `QoS` שאומר למשה באיזה מהתורים הגלובליים נרצה להריץ את המשימה שלנו
- ישנו כמה `QoS` שמוגדרים לנו:

# Concurrency

## ה-Queue GCD

- כל התורים הגלובליים הם Concurrent, הם יכולים להריץ במקביל יותר ממשימה אחת
- ה-Queue Main אשר אחראי לעדכן את ה-UI הוא Serial-י, תמיד ירץ דברים אחד אחד
- אף ממליצים כמה שיותר לעבוד עם התורים הגלובליים
- **קוד!**

# Concurrency

## DispatchQueue Sync & Async

- כאשרנו שולחים משימה ל-`DispatchQueue`, יש שתי דרכים,  
Sync-  
Async-  
Sync-  
- תבצע את הפעולה מתישהו אחר כך  
Sync-  
- תבצע את הפעולה ואני אthesesה מה היא תסימן. Sync יכול  
לזרע על ה-`Thread` של ה-`DispatchQueue` וה-`Thread` שלנו יthesesה או  
על ה-`Thread` שלנו ישירות ניהול ובכל מקרה נחסה עד שישים  
קוד!

# Concurrency

## Custom DispatchQueues

- למה ליצור תורים משלנו?  
Lock the queue so it's Serial ו לסנכרון פעולות ללא  
היקר
- אפשרות לעצור ולהפעיל מחדש את Queue באמצעות  
Resume-Suspended, Activate
- אפשרות לעבוד עם משימות חוסמות, Barrier

## Custom DispatchQueue •

- בדיפולט הם Serial
- בדיפולט הם ב-`QoS` של `default`

קווד!

# Concurrency

## Custom DispatchQueues

### • המלצה Apple :

#### Avoiding Excessive Thread Creation

When designing tasks for concurrent execution, do not call methods that block the current thread of execution. When a task scheduled by a concurrent dispatch queue blocks a thread, the system creates additional threads to run other queued concurrent tasks. If too many tasks block, the system may run out of threads for your app.

Another way that apps consume too many threads is by creating too many private concurrent dispatch queues. Because each dispatch queue consumes thread resources, creating additional concurrent dispatch queues exacerbates the thread consumption problem. Instead of creating private concurrent queues, submit tasks to one of the global concurrent dispatch queues. For serial tasks, set the target of your serial queue to one of the global concurrent queues. That way, you can maintain the serialized behavior of the queue while minimizing the number of separate queues creating threads.

- אמ;لك:
- אל תחסמו Thread-ים שרצים בתוך Queue במקוון Concurrent Queue
- אף ממליצים לא ליצר הרבה תוריים משלך כאשר הם
- מאחר זה כן מביא לבזבוז של משאבים והקצתה Concurrent Thread-ים מיותרת

# Concurrency

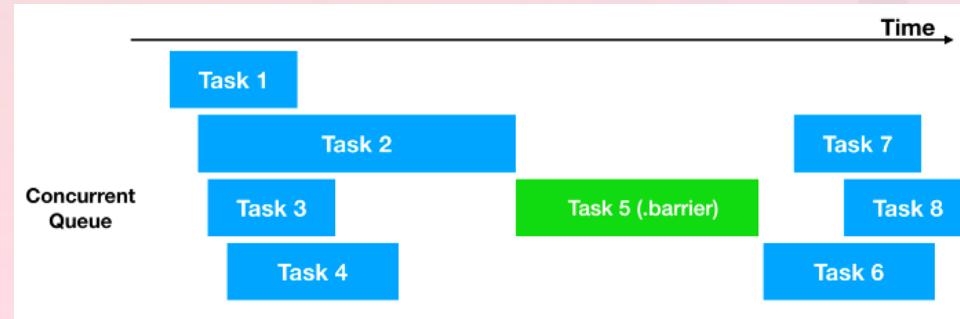
## Custom DispatchQueues

- מה אפל ממליצים לעשות אם אתה יוצרים Queues ורוצים לسانכרן דברים?
- תשתמשו ב-target
- מאפשר לנו ליצור Queue משלנו שמתנהל כמו כל Queue אבל למעשה מפנה את הפעולות שלנו ל-Queue אחר ובכך משתמש Thread

# Concurrency

## Custom DispatchQueues

- משימות Barrier, משימות "חוסמות" רצה לבד ב-Queue שהוא Concurrent
- כל משימה שהתחילה לפניה צריכה להסתיים לפני שהיא מתחילה
- כל משימה שנוספה לאחריה תתחל רק אחרי שהיא תסיים
- אפשר רק ב-Queue שאנחנו הגדרנו
- קוד!



# Concurrency

## Custom DispatchQueues

- מימוש Barrier Task Reader Writer באמצעות רצאים ש��ם יכולים לקרוא כמה שבא להם אבל שימושו כותב כולם ייחכו!
- קוד!

# Concurrency

## DispatchGroup

- שמתם לב שעד עכשו תמיד ריעננטי פעמים את ה-DataTable? לא יעיל!!!
- דרך לсинכרון בין כמה Möglichויות שעשיתם להם Dispatch
- אפשר לנו לחכות שכולן יסינו
- אפשר למש "ידנית" באמצעות Semaphore, לא מומלץ בכלל כי זה יתקע Thread-ים, ואפילו רוצים שנתערב להם ב-GCD
- קוד!

# Concurrency

## Cancellation ו WorkItems

- במקום להעביר ל-Queue שלנו Closure, אפשר ליצור WorkItem
- ולהעביר אותו Queue
- מאפשר לנו לבטל משימות שלחנו ל-Queue
- רק אם המשימה עוד לא התחילה, זה יותר כמו להוציא מהתור
- קוד!

# Concurrency

דרכים יצירתיות לירוט לעצמכם ברgel

- לאפשר לקרוא ל-Sync בתוכה Queue על אותו ה-Async על ה-Main-Sync על ה-
- אי אפשר לקרוא ל-Sync על ה-Main-Sync על ה-

# Concurrency

## סיכום

- חשוב לשים לב שהמשאבים שלנו הם Thread Safe, זוכרים ?Immutability
- תקרו על OperationQueue ו-DispatchQueue שמאפשרת עוד הרבה דברים כמו עבודה עם משימות ארוכות במיוחד והיכולת לבטל אותן גם במקרה, זה פחות נוח לשימוש יומיומי

# Reactive Programming

- מה זה Reactive Programing
- הגדרת מונחי בסיס
  - Observable
  - Observer
  - Scheduler
  - RxSwift
- Basic Observable
- Schedulers
- DisposeBag
- Subjects
- Relays
- Traits
  - RxSwift Traits
  - RxCocoa Traits
  - Reactive Todo App

# Reactive Programming

מה זה Reactive Programming

- נהגה לראשונה ב-1986
- בשנים האחרונות תפס תאוצה בטכנולוגיות רבות
- עוזר מאוד להתמודד עם אפליקציות מורכבות וניהול State מרכיב
- תפיסה תכניתית המושתת סביב זרימת מידע, חילוח שינויים  
והתגובה עליהם
- מה??

# Reactive Programming

## מה זה Reactive Programming

- הרעיון המרכזי הוא הקשבה לזרם של מידע ותגובה אליו
- זרם של מידע = Event Stream
- כל הרשמה ל-Event שאי פעם ביצעת הוא הקשבה לזרם של מידע
- (notificationCenter.addObserver גם הוא הקשבה לזרם של מידע ותגובה אליו)
- Reactive Programming הוא הרעיון הזה רק על סטראודים, הכל הוא קריאה מה-DB, פניה ל-API ואפילו קליק של כפתור

# Reactive Programming

הגדרת מונחי בסיס

- החלק המאתגר ב-Reactive Programming הוא אימוץ החשיבה ה-Reactive

```
@IBOutlet weak var button: UIButton!

@IBAction func click(sender: UIButton) {
    button.backgroundColor = randomColor()
}
```

```
@IBOutlet weak var button: UIButton!

colorObservable.subscribe(onNext: { color in
    button.backgroundColor = color
})

@IBAction func click(sender: UIButton) {
    colorObservable.accept(randomColor())
}
```

# Reactive Programming

## הגדרת מונחי בסיס

- **Observable** - זרם של מידע, הצינור שמאפשר לנו להכנס אליו מידע ולהקשיב בצד השני. בrama ה-*api* בסיסית, שימוש של Observable נדרש לספק לנו כמה פעולות
  - הרשמה לקבלת אירועים מסוגים שונים
    - מידע חדש התקבל
    - קרתה שגיאה
    - השידור הסתויים
  - אפשרות שליחת אירוע חדש
- **Observer** - מתחבר אל זרם המידע ומגיב לאירועים שקוראים
- **Scheduler** - מנהל תחנת המוניות של ה-Thread-ים, לרוב נרצה שאירועים יוכלו לעבור בקלות ובנוחות בין Thread-ים בלי שנוצרך לחשב על זה יותר מדי, ה-*Scheduler* אחראי לקחת אירועים שנכנסו ל-*Observable* ולהעביר אותם ל-Thread המתאים ב-*Observable*

# Reactive Programming

## RxSwift

- מימוש ב-Swift של ספריית Rx (Reactive Extensions) של ספריית-h-Rx. במקור, ספריית-h-Rx פותחה ע"י Microsoft.
- גרסה ראשונה של הספרייה יוצאה ב-2011 ל-.Net. ונהייתה Open Source ב-2012.
- המימוש הכי פופולארי By Far של תוכנות ריאקטיבי, נהיה מילה נרדפת אליו.
- משלבת בין Observer Design Pattern, תוכנות פונקציונלי וריאקטיבי ליצירת ספריה מאוד חזקה ומגוונת.

# Reactive Programming

## RxSwift

- למעשה מתחלקת לשולשה חלקים
  - Core - ה- Core של הספרייה RxSwift
  - RxRelay - מספקת לנו קיצורי דרך נוחים לעבודה עם Observables
- RxCocoa - מספקת חיבורים ספציפיים ל-OS ולבוגה עם UI
- החלוקה היא כדי שנוכל לחת מה שאנו צריכים ורוצים ולא "הכל או כלום"

# Reactive Programming

## RxSwift - Basic Observable

- **Sequence** הוא סוג של Observable, אבל אסינכריוני
- ניצור אחד באמצעות `.create`
- נרשם אליו באמצעות `.subscribe`.
- **קוד!**

# Reactive Programming

## RxSwift - Schedulers

- יש 5 סוגי Schedulers מובנים ב-RxSwift:
  - **CurrentThreadScheduler** - סדרתי, מפעיל את זה על אותו thread ממנו הבצעה ההרשמה.
  - **MainScheduler** - סדרתי, מרץ על ה-UI Thread.
  - **SerialDispatchQueueScheduler** - סדרתי, מקבל Queue או QOS ומרץ עליו, גם אם קיבל Queue מקבילי, הוא ממיר את העבודה לסדרתית, מעשה ה-`Main` instance שלו.
  - **ConcurrentDispatchQueueScheduler** - מקבילי, מתאים לעבודות רכע
  - **OperationQueueScheduler** - מקבילי, מתאים לעבודות רכע ממושכת יותר ועובד עם Queue, מאפשר שליטה יותר מדויקת
- אפשר למשם גם Scheduler משלכם, אבל כדאי שתהיה לכם סיבה טוב לזה

# Reactive Programming

## RxSwift - DisposeBag

- ה-Closure שאנו מعتبرים subscribe נשמר ב-`Observable`.
- אם לא נשחרר אותו אנחנו עלולים ליצור Retain Cycle ולייגות זיכרו.
- כשרשימים ל-`Observable`, מקבלים בחזרה `Disposable`, הדך לבטל את ההרשמה היא לעשות לו `dispose`.
- ה-`DisposeBag` גורם לזה להתנהג יותר כמו ARC קלاسي וחוסך לנו קוד.
- כאשר ה-`class` שלנו יעשה `deinit` וישחרר את משאב ה-`DisposeBag`,
- הוא ישחרר את כל מה שנוסף לתוכו ע"י ביטול ההרשמה של כלום או `Complete` או `Error`.

# Reactive Programming

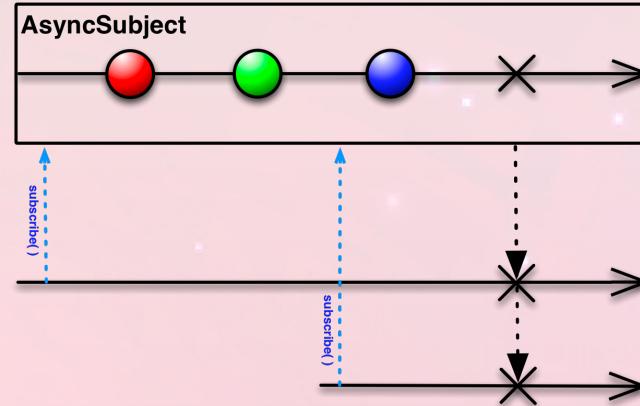
## RxSwift - Subjects

- Subject הוא סוג של גשר אשר מתפקיד גם כ-Observer וגם כ-Observable
- מKeySpecיב לאירועים וגם משדר אותם להאה
- מוסיף לנו אופציות כמו Caching, Shidur מחדש ועוד
- נוח לעבודה יומיומית, לרוב לא כתבו Observables מאפס

# Reactive Programming

## RxSwift - Subjects

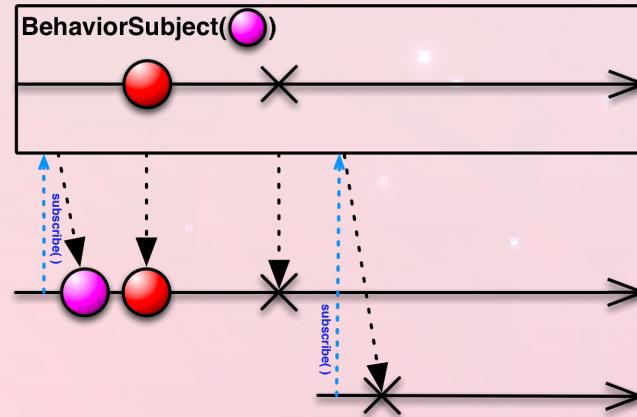
- משדר רק את הערך האחרון שקיבל ורק לאחר  
Error או Complete



# Reactive Programming

## RxSwift - Subjects

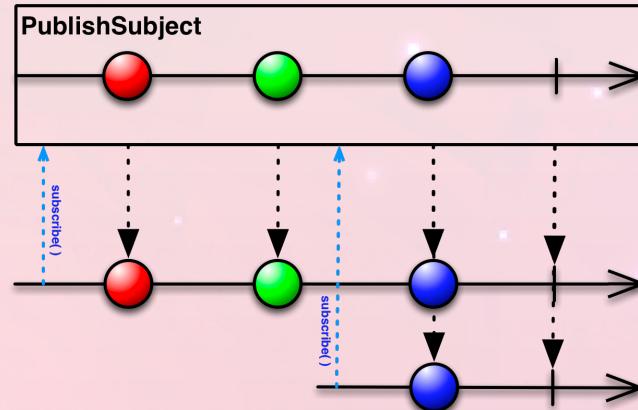
- מתחילה עם ערך הinitial ומשחזר אותו לכל נרשם חדש, אפשר לחשב על זה כמו משתנה מקומי `var` שיש לנו בתוך `the-class` שלנו, אבל אפשרות נוספת לבנוסף להירשם אליו ולעבוד `reactive`.



# Reactive Programming

## RxSwift - Subjects

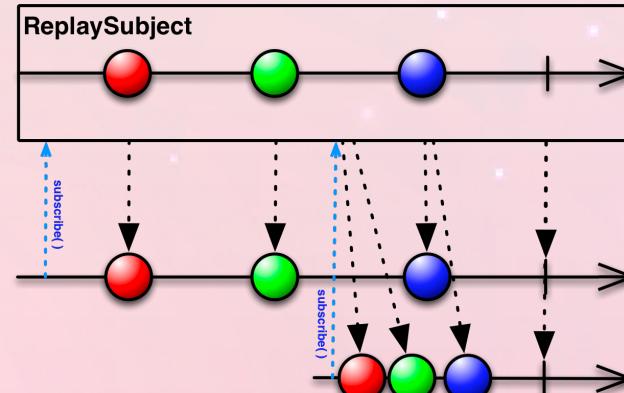
- מואוד דומה למה שראינו ב-`BehaviorSubject`, עם הבדל אחד מהותי, הוא לא שומר כלום, מה שלוחים הוא שולח להלאה, מי שנרשם "מאוחר" מפספס את האירועים שנשלחו



# Reactive Programming

## RxSwift - Subjects

- ReplaySubject - הוא שילוב של שני העולמות שראינו עכשו או אפשר לראות בו כ-Subject עם Cache עם Subject, הוא לא מקבל מצב ראשון, אבל אפשר להעביר לו בפרמטר כמה buffer רוצחים שהוא ישמור, כשמישהו מתחילה לאזין לו והוא יקבל את כל הבאפר שנשמר מיידית ולאחר מכן יעבדו "רגיל"



# Reactive Programming

## RxSwift - Relays

- לא חלק מ-RxSwift הבסיסי, בתחילת היי חלק מ-RxCocoa, כיום חלק מ-RxRelay
  - יש שלושה, הם מקבילים בדיק ל-Subjects
    - BehaviorRelay
    - PublishRelay
    - ReplayRelay
  - נבדלים אך ורק בשני דברים
    - לא מסיים לעולם, אין להם Complete
    - לא זורקים שגיאות לעולם, אין להם Error
  - הם עוטפים את Subject בפנים ברמת המימוש
    - שימושי למקרים שבהם לא יכולות להיות שגיאות או סוף למידע או שפשות לא מעוניין אותנו ומקוצר לנו את הקוד

# Reactive Programming

## RxSwift - Traits

- קיומם של ה-Traits הוא למעשה דרך לנצל את היותה של Swift שפה Strongly Typed ולשפר את השימושיות, הカリונות והתחזוקתיות של הקוד הריאקטיבי שלנו
- קיימות רק בחלק מהימושים של Rx, Swift היא אחת מהן RxSwift Core מ-
  - יש Traits שהן חלק מ-RxCocoa מ-
  - יש Traits

# Reactive Programming

## RxSwift - RxSwift Traits

- **Single** - משדר רק ערך אחד במקרה של הצלחה או שגיאה אחת במקרה של כישלון, מקרה מאד שימושי לזה הוא קריאות ל-API
- **Completable** - לא מחזיר שום ערך, רק `Success` או `Error`, יכול לשמש אותנו בכל מקרה שבו אנחנו רק רוצים לדעת متى פעולה Observable<void> נגמרה והאם היא הצלחה או נכשלה, למשל כמו
- **Maybe** - שילוב בין `Single` ו-`Completable`, יכול להסתיים בהצלחה בלי ערך, להחזיר שגיאה או להחזיר ערך בודד

# Reactive Programming

## RxSwift - RxCocoa Traits

- עיקרונו RxSwift-ב-Sharing
- כל פעם שעושים הרשמה ל-Observable הוא מרים מחדש את כל שרשרת האופרטורים שלו (create, map וכו')
- הרשמה מרובה יכולה לגרום לבזבוז משאבים
- לעשות Share ל-Observable אומר RxSwift, אל תרים הכל מחדש, תחלוק

# Reactive Programming

## RxSwift - RxCocoa Traits

- שני פרמטרים של Share
  - cache - כמו replay לשמור אחריה, עבור נרשם חדש
  - Observable כמו אירועים אחריה לשולח לו
- scope - שתי אופציונות
  - whileConnected - שהוא ה-cache，默认, נשמר כל עוד מישרו מחובר ל-Observable, ברגע שכולם מתנתקים, הכל ”נעלם”
  - forever - נשמר לנצח, לא מומלץ כמעט אף פעם, יכול להחזיר מידע ישן וליצור זילוגות זיכרון
- קוד!

# Reactive Programming

## RxSwift - RxCocoa Traits

- Driver - מטרתו לספק דרך נוחה ואינטואיטיבית לכתוב קוד ריאקטיבי בשכבה ה-UI
  - לא זורק שגיאות
  - תמיד עובד על ה-MainScheduler
  - תמיד עושה Share עם replay של אחד
  - ל-Driver תהיה פונקציית Drive שיכולה לחבר אותו לשירות ל-Observer אחר ולרכיב UI
  - שמו של ה-Driver היא מכיוון שהוא UI Drives Data to The
  - דומה מאוד ל-Driver אך לא עושה replay ל-event Signal

```
results.asDriver()
    .map { "($0.count)" }
    .drive(resultCount.rx.text)
    .disposed(by: disposeBag)
```

# Reactive Programming

## RxSwift - RxCocoa Traits

- תכונה זו מייצגת **Property** של אלמנט UI כלשהו, **ControlProperty** - היא עליה event-ים רק של הערך הראשון שניתן ל-control ושל הznות או שינויים שבוצעו ע"י המשתמש ולא תסדר שינויים שבוצעו ע"י הקוד.
- לא נכשל ולא זורק שגיאות
- שולח **Complete** כשהרכיב UI עושה **Deallocate**
- עושה **Share** וגם **Replay** של 1, כלומר יחזור על הערך האחרון לנרשמים חדשים
- תמיד יחזיר אירועים שמתרחשים על ה-**MainScheduler**
- דוגמא: `UISearchBar.rx.value`
- ניצור אחד כזה משלנו רק אם נרצה UI View ולבנות לו **RxExtension**

# Reactive Programming

## RxSwift - RxCocoa Traits

- ControlEvent - דומה לרעיון של `ControlProperty`, אך בניגוד אליו, הוא מייצג אירוע שהתרחש, כמו לדוגמה `tap` שעושם על כפתור
- לא נכשל ולא זורק שגיאות
- שולח `Complete` כשהרכיב UI עושה `Deallocate`
- לא שולח ערכים ראשוניים, חיית הרגע
- תמיד יחזיר אירועים שמתרחשים על ה-`MainScheduler` על-
- `UIButton.rx.tap`: לדוגמא:
- ניצור אחד כזה משלנו רק אם נרצה `Custom UI View` ולבנות לו `Event-Extension` `RxExtension` שמתפלת ב-

# Reactive Programming

Reactive ToDo App

- בואו נחבר הכל ביחד!
- מלא קוד!

# Combine

---

- What is Combine
- When to use Combine
- Basic Terms
- Scheduler
- Subjects
- Cancelable
- Future
- Synchronizing Pipelines
- Combine Extensions
- Combine vs RxSwift

# Combine

---

מה זה Combine

Framework שנכתב ע"י Apple על מנת לתמוך בכתיבת קוד ריאקטיבי  
בדומה למספר חיצוניות כמו RxSwift

# Combine

---

מתי להשתמש ב Combine ?

- אם מפתחים אפלקציה ריאקטיבית
- אם עובדים עם SwiftUI
- אם מפתחים אפלקציה ב iOS 13 ומעלה

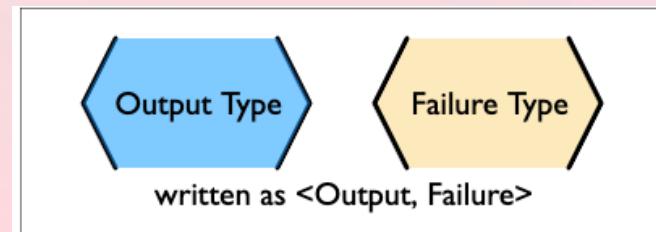
# Combine

---

## Publisher

קומפוננטה ב-Combine אשר מייצרת ערכאים ומשדרת אותם.  
בדומה לתפקידו של ה-`Observable` ב-`RxSwift`.  
ה-`Publisher` מייצר שלושה סוגי ערכאים:

- מידע חדש
- קרתה שגיאה
- השידור הסתיים



# Combine

---

## Subscriber

קומפוננטה ב-Combine אשר נרשמת ל-Publisher ומקבלת את ה-event-ים אשר הוא משדר.  
יש שני Subscriber-ים מובנים ב-Combine :

```
.sink { receivedValue in
    print("The end result was \(receivedValue)")
}
```

```
.assign(to:\.isEnabled, on: signupButton)
```

# Combine

---

## Operators

קומפוננטות שעובdot גם כ-Publisher וגם כ-Subscriber. למעשה ה-**Subscriber** מאפשרות לשנות מידע מ-**Publisher** ולהזרים אותו להלה. ניתן לשרשר

קומפוננטות כאלו, לדוגמה:

- filter •
- map •

# Combine

---

## Pipeline

אם מחברים את כל הקומפוננטות נוצר Pipeline

```
let _ = Just(5)
    .map { value -> String in
        switch value {
            case _ where value < 1:
                return "none"
            case _ where value == 1:
                return "one"
            case _ where value == 2:
                return "couple"
            case _ where value == 3:
                return "few"
            case _ where value > 8:
                return "many"
            default:
                return "some"
        }
    }
    .sink { receivedValue in
        print("The end result was \(receivedValue)")
    }
```

# Combine

---

## EraseToAnyPublisher

כשאנו יוצרים Swift ,pipeline משרשרת את כל ה-Type-ים שיש ב-**Type**-ל Pipeline-ב-Kotlin:

```
let x = PassthroughSubject<String, Never>()
    .flatMap { name in
        return Future<String, Error> { promise in
            promise(.success(""))
                .catch { _ in
                    Just("No user found")
                }.map { result in
                    return "\((result) foo"
                }
        }
    }
```

```
Publishers.FlatMap<Publishers.Map<Publishers.Catch<Future<String, Error>, Just<String>>, String>, PassthroughSubject<String, Never>>
```

כדי שנוכל לעבוד עם Type-ים בנוחות, אפשר להוסיף את הפונקציה לשנות את Type-ים eraseToAnyPublisher למשהו קרייא:

```
AnyPublisher<String, Never>
```

# Combine

---

## Scheduler

קובע את ה-Thread שעליו ה-Pipeline יroz.  
Swift-Combine משתמש בכל האלמנטים הקיימים ב-  
DispatchQueue, OperationQueue, RunLoop

ה-Scheduler היחיד ש-Combine מוסיף זה ה-ImmediateScheduler אשר מבצע פעולות סינכרוניות באופן מיידי

במידה ולא נקבע Scheduler בצורה יזומה ה-Combine קובע Scheduler עצמו.

# Combine

---

Scheduler

```
let subject = PassthroughSubject<Int, Never>()
// 1
let token = subject.sink(receiveValue: { value in
    print(Thread.isMainThread)
})
// 2
subject.send(1)
// 3
DispatchQueue.global().async {
    subject.send(2)
}
```

# Combine

---

## Scheduler

```
let _ = Just(5)
    .map { value -> String in
        switch value {
            case _ where value < 1:
                return "none"
            case _ where value == 1:
                return "one"
            case _ where value == 2:
                return "couple"
            case _ where value == 3:
                return "few"
            case _ where value > 8:
                return "many"
            default:
                return "some"
        }
    }
    .subscribe(on: DispatchQueue.global(qos: .background))
    .receive(on: DispatchQueue.main)
    .sink { receivedValue in
        print("The end result was \(receivedValue)")
    }
```

# Combine

---

## Subjects

Subject הוא מקרה מיוחד של Publisher שמשמש את פרוטוקול Subject, אשר מחייב שימוש של פונקציית Send. יש שני סוגי Subjects מובנים ב-Combine:

- CurrentValueSubject
- PassthroughSubject

# Combine

---

## Future

נוסך שווה להתקבב עליו הוא Future. הרעיון שלו דומה Javascript של Promise-

```
func performAsyncAction(completionHandler: @escaping () -> Void) {
    DispatchQueue.main.asyncAfter(deadline:.now() + 2) {
        completionHandler()
    }
}
func performAsyncActionAsFuture() -> Future <Void, Never> {
    return Future() { promise in
        DispatchQueue.main.asyncAfter(deadline:.now() + 2) {
            promise(Result.success(()))
        }
    }
}
performAsyncActionAsFuture()
    .sink() { _ in print("Future succeeded.") }
```

# Combine

---

## Canclables

דומה ל-RxSwift `DisposeBag`. מאפשר לעשות `cancel` ב-`Swift`, `Garbage Collector` ומשתמשים `Subscribers` ב-`Swift`, `Cancellable` אחראי לשחרר את המשתמשים של `Subscribers`

```
private var cancellableSet: Set<AnyCancellable> = []  
  
let mySinkSubscriber = remotePublisher  
    .sink { data in  
        print("received ", data)  
    }  
    .store(in: &cancellableSet)
```

# Combine

---

## Synchronizing Pipelines

לפעמים נרצה לשלב כמה pipeline-ים ביחד ולבצע פעולות בSubscriber אחד. קיימים מספר אופרטורים אשר יכולים לעזור במקרה זה

# Combine

---

## Synchronizing Pipelines

```
let germanCities = PassthroughSubject<String, Never>()
let italianCities = PassthroughSubject<String, Never>()
let europeanCities = Publishers.Merge(germanCities, italianCities)

_ = europeanCities.sink(receiveValue: { city in
    print("\(city) is a city in europe")
})

germanCities.send("Munich")
germanCities.send("Berlin")
italianCities.send("Milano")
italianCities.send("Rome")
```

# Combine

---

## Synchronizing Pipelines

```
let selectedFilter = PassthroughSubject<String, Never>()
let searchText = PassthroughSubject<String, Never>()

let publisher =
    Publishers.CombineLatest(selectedFilter, searchText, transform:
        { selectedFilter, searchText in
            "\n(selectedFilter) \n(searchText)"
        })
    _ = publisher.sink { value in
        print(value)
    }
```

# Combine

---

## Combine Extension

אחד הפיצ'ירים המוגנים של Combine הוא יכולת להתחבר בצורה מאוד אלגנטית להרבה מהkomפוננטות הקיימות ב-Swift וליצור מהם Publishers באמצעות Extensions. אפשר גם ליצור Extensions משלנו לדברים שעוד לא קיימים.

# Combine

---

## Combine Extension

```
let sub = NotificationCenter.default.publisher(for:  
    NSControl.textDidChangeNotification, object: filterField)  
.map { ($0.object as! NSTextField).stringValue }  
.assign(to: \MyViewModel.filterString, on: myViewModel)
```

# Combine

---

## Combine Extension

```
let cancellable = Timer.publish(every: 1.0, on: RunLoop.main, in: .common)
    .autoconnect()
    .sink { receivedTimeStamp in
        print("passed through: ", receivedTimeStamp)
    }
```

# Combine

---

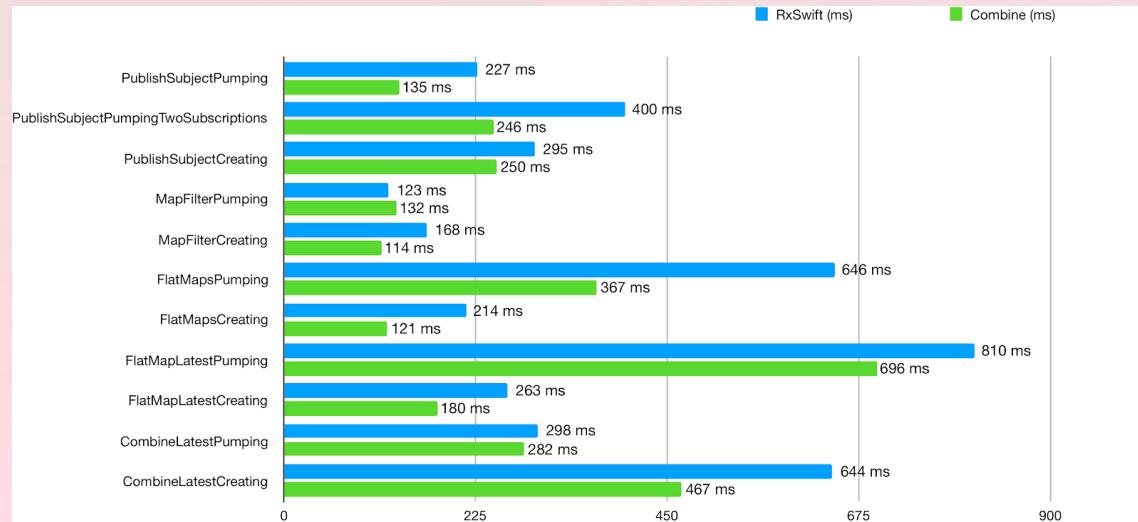
## Combine vs RxSwift

- שתי הפתרונות מאפשרים לפתח את האפליקציה שלנו בארכיטקטורה ריאקטיבית.
- RxSwift זו ספריה חיצונית, לעומת זאת Combine היא חלק מ-iOS Framework
- יש דמיון בין ה-API של RxSwift ו-Combine, אבל ה-API של Combine פשוט יותר וכולל Publishers ו-Subscribers בלבד

# Combine

## Combine vs RxSwift

- מבחינה ביצועים Combine מנצח בקלות ומציג שיפור של כ-40% ביצועים לעומת RxSwift



# Combine

---

## Combine vs RxSwift

- RxSwift הוא פיצ'ר שלא קיים בכלל ב-Combine Back Pressure
- ב-RxSwift משתמשים ב-RxCocoa Bind כה לעשות יישירות
- לkomponenotot של ה-UI וב-Combine משתמשים ב-Assign
- **publisher.assign(to: \.text, on: label)**
- Combine מאפשר להתחבר לכל komponenotah ב-Framework של iOS
- וליצור ממנה Publisher בצורה אינטואיטיבית ונווחה
- במקומם DisposeBag משתמשים ב-Cancellable.
- iOS 13 החל מ-Combine נתרמן
- **קוד!**

# SwiftUI

---

- What is SwiftUI
- Basic SwiftUI example
- States and SwiftUI
- Combine + SwiftUI

# SwiftUI

---

## What is SwiftUI

- שינוי גישה בבנייה UI והחיבור ללוגיקה
- מקום לבנות UI בעזרת Storyboard ו-XIB כתובים את כל ה-UI בעזרת קוד דקלרטיבי
- היתרונות העיקריים:
  - אין יותר הפרדה בין ה-Storyboard לבין הקוד, לא צריך לחבר IBOutlets, IBActions וכו'
  - נותן אפשרות לעשות reuse ל-UI ביתר קלות
  - נותן אפשרות לעבוד בנוחות עם Colors, Theming, כמו הגדרת Colors גלובלי
  - אין בעיות של Merge ב git
  - אפשר לעשות Code Review ל-UI בקלות
  - אין קריסות בגלל ששכחנו להגדיר IBOutlet וכו'

# SwiftUI

---

## What is SwiftUI

- במקום לשנות State של קומפוננטות לפי Events שmagיעים מהמשתמש או מכל מקום אחר, ה UI נבנה מחדש בכל שינוי State SwiftUI מחליף לוחוטין את UIKit
- הכל ב-UI הוא View, החל מה-View הcy קטן עד למסך כולו.
- והכל מוגדר ב-Struct SwiftUI מאפשר לנו לראות שינויים באפליקציה בזמן אמת בלי צורך לCOMM פל לבנות מחדש
- אפשר לבנות אפליקציה הכוללת גם UIKit וגם SwiftUI במקביל לצערנו Apple זה Apple, שכן אין תמיכה לאחר מכן, אפשר לבנות אפליקציות SwiftUI רק למערכות הפעלה iOS 13 ומעלה

# SwiftUI

---

```
struct TestView: View {  
    var body: some View {  
        VStack {  
            Text("Text")  
            Spacer()  
            Button("button", action: {  
                print("button pressed")  
            })  
        }  
    }  
}
```

# SwiftUI

---

## States and SwiftUI

- כמו שאמרנו מוקדם ה-UI ב-UI SwiftUI נבנה מחדש מוחדש בכל שינוי State. ישנו כמה סוגים של הגדרות ל State
  - @State - משתמש אותו ל primitive types
  - @StateObject - משמש אותו ל custom class, class, ObservableObject. וכל שדה שיצרנו חייב למשר פרוטוקול ObservableObject. וכל שדה שאנחנו מעוניינים שהוא יבנה מחדש בהתאם לשינוי בו, מגדרים כ-Published
  - @ObservedObject - מאפשר להעביר ל-Child View את ה-Observable Object
  - @EnvironmentObject - מאפשר ליצור state שימושי על כמה views בלי הצורך לחחל את ה state פנימה

# SwiftUI

---

```
struct MyView: View {
    @State private var text: String = ""

    var body: some View {
        VStack {
            Text(text)
        }
    }
}
```

# SwiftUI

---

```
struct TodoList: View {
    @StateObject private var viewModel = ToDoListViewModel()

    var body: some View {
        VStack() {
            List(viewModel.items) { item in
                TodoCell(viewModel: TodoItemViewModel(todoItem: item))
            }
        }
    }
}

class ToDoListViewModel: ObservableObject {
    @Published var items: [Item] = []
}
```

# SwiftUI

---

```
NavigationLink("To child", destination: ChildView(observedObject:  
stateObject))  
  
struct ChildView: View {  
    @ObservedObject var observedObject: TestObject  
  
    var body: some View {  
        VStack {  
            Text("ObservedObject: \(observedObject.num)")  
            Button("Increase observedObject", action: {  
                observedObject.num += 1  
                print("ObservedObject \(observedObject.num)")  
            })  
        }  
    }  
}
```

# SwiftUI

---

```
struct TestApp: App {  
    @StateObject var environmentObject = TestObject()  
    var body: some Scene {  
        WindowGroup {  
            ContentView().environmentObject(environmentObject)  
        }  
    }  
}  
  
@EnvironmentObject var environmentObject: TestObject
```

# SwiftUI + Combine

---

- משtałב בצורה מאד מובנה לתוך ה-UI SwiftUI
- השילוב בין SwiftUI ו-Combine יוצר קוד נקי, קריא וקצר



# Thank You!

