

لغة Swift - مسودة


أكاديمية طويق

مقدمة في لغة Swift

هي لغة برمجة قدمتها Apple في عام ٢٠١٤ تستخدم في تطوير التطبيقات لأنظمة التشغيل التالية: iOS, iPadOS, macOS, tvOS, Linux

في 2015 نشرت Apple رسميا ان مشروع Swift اصبح مفتوح المصدر و يمكن العثور على repository الخاص بلغة Swift، في صفحة Apple على موقع GitHub من الرابط التالي

<http://github.com/apple>

 Apple • github.com

أمر الطباعة print()

• مفهوم print()

يستخدم أمر الطباعة `print` في لغة Swift لطباعة رسالة إلى وحدة التحكم console حيث يتم وضع النص المراد طباعته بين القوسين ()

```
print("Welcome to Swift")
```

المثال أعلاه يوضح عملية طباعة عبارة Welcome to Swift باستخدام أمر الطباعة `print`

○ المخرجات:

```
Welcome to Swift
```

○ المراجع:

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

المتغيرات Variables والثوابت Constants

مقدمة في المتغيرات Variables والثوابت Constants

ستحتاج في مرحلة ما أثناء عملية البرمجة إلى التعامل مع البيانات، وعند رغبتك في تخزين تلك البيانات فإنك ستحتاج إلى شيء يقوم بتخزينها وحفظها لك، وهذا هو عمل المتغيرات والثوابت، ويمكن القول أنها الأوعية أو الحاويات التي تحتوي وتُخزن كل ما يوضع فيها، إذاً يمكن النظر إلى المتغير والثابت على أنه أسلوب بسيط لتخزين البيانات واسترجاعها بشكل مؤقت أثناء عمل البرنامج.

الفرق بين المتغيرات والثوابت

- **المتغيرات variables:** تحتوي على قيمة قابلة للتحديث، تستخدم لتعريف القيم التي من الممكن أن تتحدث أثناء سير البرنامج مثل: عمر الشخص ودرجة حرارة الغرفة.
- **الثوابت constants:** تحتوي على قيمة لا يمكن تحديثها، تستخدم لتعريف القيم التي لن تتحدث أبداً بعد أن يتم إسنادها مثل: عدد أيام الأسبوع ودرجة الحرارة التي تجمد الماء.

طريقة تسمية المتغير والثابت

يمكنك استخدام أي اسم تقريباً عند تسمية المتغيرات والثوابت، ولكن هناك بعض القواعد التي يجب عليك اتباعها:

- لا يمكن أن يحتوي الاسم على مسافات
- يمكن استخدام الرمز `_` ولكن لا يمكن استخدام الرموز الأخرى مثل: `^`, `!`, `@`, `%`, `*`, `/`, `=`, `-`, `+`
- لا يمكن أن يبدأ برقم ولكن من الممكن أن يحتوي على أرقام
- لا يمكن أن تستخدم الكلمات المحجوزة في Swift مثل `let`, `var`, `print`, `func`, `class`.. إلخ

إنشاء المتغيرات Variables

لتعريف متغير نستخدم كلمة `var` متبوعة باسم المتغير ولتوضيح الفكرة لاحظ معي المثال التالي:

```
var currentTemperature = 22
var currentSpeed = 55
```

في المثال السابق تم تعريف المتغيرين `currentTemperature` و `currentSpeed` وإسناد القيم لهما.

- إنشاء أكثر من متغير

يمكننا إنشاء أكثر من متغير في سطر واحد وذلك بفصلهم باستخدام الفاصلة , مثال:

```
var applesPrice = 2, bananasPrice = 5
```

استخدام المتغيرات Variables

بعد أن قمنا بإنشاء المتغير يمكن أن نستدعيه لأكثر من مكان في البرنامج، يتم استدعاء المتغير عن طريق كتابة اسمه كما هو موضح أدناه:

```
var studentName = "Mohammad"  
print(studentName)
```

في المثال أعلاه قمنا باستدعاء المتغير `studentName` داخل أمر الطباعة `print`، والذي بدوره سيقوم بطباعة قيمة المتغير إلى وحدة التحكم. لاحظ أنه عند استدعاء المتغير تم كتابة اسمه فقط، بدون استخدام كلمة التعريف `var`.

• المخرجات:

```
Mohammad
```

تحديث قيمة المتغير

لتحديث قيمة المتغير نقوم بكتابة اسم المتغير متبوعًا بمعامل الإسناد `=` ثم القيمة الجديدة. كما هو موضح في المثال التالي:

```
studentName = "Ahmed"  
print(studentName)
```

• المخرجات:

```
Ahmed
```

نلاحظ أنه تم تحديث قيمة المتغير `studentName` إلى `Ahmed`.

إنشاء متغير بدون قيمة

يمكن إنشاء متغير لايحتوي على قيمة ويتم إسناد القيمة إليه فيما بعد، للقيام بذلك يجب علينا تحديد نوع البيانات القيمة التي ستسند لاحقاً للمتغير، ويعرف ذلك بمفهوم `type annotation`، يوضح المثال التالي كيفية إنشاء متغير وتحديد نوع البيانات لقيمتة:

```
var cherriesPrice: Int
cherriesPrice = 4
```

يوضح السطر 1 إنشاء متغير وتحديد نوع البيانات لقيمتة `Int` بدون إسناد القيمة له، بينما يوضح السطر 2 عملية إسناد القيمة إليه، كما نلاحظ أيضاً يجب أن تكون القيمة التي تسند للمتغير من نفس نوع البيانات الذي تم تحديدها.

إنشاء الثوابت Constants

لتعريف ثابت، نستخدم كلمة `let` متبوعةً باسم الثابت. لتوضيح الفكرة لاحظ معي المثال التالي:

```
let birthYear = 1990
let daysInWeek = 7
```

تم إسناد القيمة 1990 إلى الثابت `birthYear` ، والقيمة 7 إلى الثابت `daysInWeek`

• إنشاء أكثر من ثابت

يمكننا إنشاء أكثر من ثابت في سطر واحد وذلك بفصلهم باستخدام الفاصلة , على النحو التالي:

```
let squareCorners = 4, triangleCorners = 3
```

استخدام الثوابت Constants

بعد أن قمنا بإنشاء الثابت، يمكن أن نستدعيه في أكثر من مكان في البرنامج، ونقوم باستدعاء الثابت عن طريق كتابة اسمه كما هو موضح في المثال التالي:

```
print(squareCorners)
```

في المثال السابق قمنا باستدعاء الثابت `SquareCorners` داخل أمر الطباعة `print` والذي بدوره سوف يقوم بطباعة قيمة الثابت إلى وحدة التحكم.

• المخرجات:

ماذا سيحدث لو حاولنا تحديث قيمة ثابت؟

يمكننا تحديث قيمة المتغير ولكن لا يمكننا تحديث قيمة الثابت، وعند القيام بذلك سينتج خطأ كما في المثال التالي:

```
let dateOfBirth = "08/10/1996"
dateOfBirth = "07/11/1997"
```

• المخرجات:

```
Cannot assign to value: 'dateOfBirth' is a 'let' constant
```

تم إنشاء `dateOfBirth` على أنها ثابت لذلك لا يمكننا تحديث قيمتها، وعندما حاولنا القيام بذلك حصلنا على تنبيه ينص على وجود خطأ، كما هو موضح في المخرجات أعلاه.

إنشاء ثابت من غير تحديد قيمة

يمكننا إنشاء ثابت لا يحتوي على قيمة ونسند القيمة إليه فيما بعد، وللقيام بذلك يجب علينا استخدام `type annotation` وتعني تحديد نوع البيانات التي سوف تسند لاحقاً للثابت، يوضح المثال التالي كيفية القيام بذلك:

```
let pentagonCorners: Int
pentagonCorners = 5
```

بما أن `pentagonCorners` تم تعريفه كثابت لا يحتوي على قيمة، فإنه لا يمكن استخدامه إلا بعد أن تقوم بإسناد قيمة إليه، وتكون مرة واحدة فقط.

التعليقات Comments

مقدمة في التعليقات Comments

في حالات معينة أثناء كتابة الكود قد يحتاج المبرمج إلى وضع بعض الملاحظات أو التعليقات. فمثلاً قد يحتاج إلى وضع ملاحظة لتذكيره بتعديل `code` معين، فيقوم المبرمج حينها بكتابة بعض الملاحظات بجانب ذلك الكود للعودة إليه فيما بعد، وفي حالات أخرى قد يعمل على الملف أو المشروع البرمجي أكثر من شخص، وقد يحتاج أحد المبرمجين إلى أن يضع بعض الملاحظات لأعضاء الفريق، وهكذا تساعد التعليقات في Swift المبرمج على كتابة ما يود من ملاحظات في البرنامج. وبالنسبة للغة Swift فإنها ستتجاهل تلك التعليقات ولن تنظر لها على أنها تعليمات ستقوم بتنفيذها.

أنواع التعليقات

- تعليق السطر الواحد Single Line Comment.
- تعليق متعدد الأسطر Multi-line Comment.

• تعليق السطر الواحد Single Line Comment

عند رغبتنا في وضع تعليق في سطر واحد أو ما يُسمى single-line comment والذي سينتهي بنهاية السطر سنستخدم `//` علامة لبداية التعليق ويوضح السطر التالي هذه الفكرة:

```
// This is a comment.
```

ليس بالضرورة أن يبدأ التعليق من بداية السطر فقد يكون التعليق هو جزء من سطر برمجي، ولتوضيح الفكرة لاحظ معي المثال التالي:

```
var age = 25 // This is my age.
```

• تعليق متعدد الأسطر Multi-line Comments

في بعض الحالات قد نحتاج إلى كتابة تعليق طويل يمتد إلى أكثر من سطر في هذه الحالة، يمكننا استخدام أسلوب التعليق متعدد الأسطر Multi-line Comment. ونقوم بذلك عن طريق كتابة الملاحظات بين العلامات `/* */`. يوضح المثال التالي هذا الأمر:

```
/* Write your  
   comments here..  
*/
```

قد يستخدم البعض أسلوب التعليق متعدد الأسطر كتعليق سطر واحد، ولتوضيح الفكرة لاحظ معي المثال التالي:

```
/* Write your comment here.. */
```

بنفس الأسلوب يمكنك استخدامها عند وجود سطر برمجي، ولتوضيح الفكرة لاحظ المثال التالي:

```
var age = 25 /*This is my age. */
```

• المراجع :

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

مقدمة في أنواع البيانات

نظرة على مفهوم أنواع البيانات

أنواع البيانات هي أنواع القيم التي تحتفظ فيها المتغيرات والثوابت والتي يمكن أن نتعامل معها في البرنامج مثل ، `String` ، `Boolean` ، `Character` ، `Integer` ، `Float` ، `Double`.

مقدمة في النوع String

النص `String` هو نوع من أنواع البيانات ويتكون من مجموعة أحرف ورموز داخل علامات التنصيص `""` كما هو موضح في المثال التالي:

```
var myName = "Ahmed"
```

تم إسناد النص `Ahmed` للمتغير `myName`. كما نلاحظ، فالبيانات من نوع `String` تكون بين علامتي التنصيص الثنائية `""`.

نظرة على Multiline String Literal

- إذا أردت إنشاء نص يمتد لأكثر من سطر، يمكنك ذلك عن طريق استخدام ثلاث علامات تنصيص ثنائية `"""` قبل و بعد محتوى النص كما هو موضح في المثال التالي:

```
var message = """
```

```
This is a multiline string literal,  
It can span more than one line  
""  
print(message)
```

○ المخرجات:

```
This is a multiline string literal,  
It can span more than one line
```

- يمكننا استخدام علامتي تنصيص داخل نص فعليًا استخدام ثلاث علامات تنصيص ثنائية "" داخل String

```
var note = ""  
Ahmed said:  
    "I'm on my way home"  
""  
print(note)
```

○ المخرجات:

```
Ahmed said:  
    "I'm on my way home"
```

إدراج قيمة داخل النص String Interpolation

يمكنك إدراج قيمة داخل النص، للقيام بذلك لاحظ معي المثال التالي:

```
var name = "Ahmed"  
var age = 20  
var myMessage = "My name is \(name) and I'am \(age) years old"  
print(myMessage)
```


لاحظ أننا قمنا بتضمين المتغير `name` و `age` داخل النص عن طريق كتابتهما بداخل `() \` .

• المخرجات:

```
My name is Ahmed and I'am 20 years old
```

في المثال السابق تم استدعاء المتغيرين `name` و `age` داخل نص المتغير `myMessage` وذلك باستخدام أسلوب `string interpolation`.

مقدمة في النوع Character

الحرف أو الرمز `Character` هو نوع من أنواع البيانات مشابه لنوع `String` ولكن فقط يحتوي على حرف أو رمز أو رقم واحد داخل علامات التنصيص كما هو موضح في المثال التالي:

```
var myLetter: Character = "A"
```

تم إسناد الحرف `A` للمتغير `myLetter`.

ملاحظة: في المثال السابق قمنا باستخدام علامات `:` ثم حددنا نوع `Character` في جملة إنشاء المتغير `myLetter` وذلك لجعل نوع البيانات التي يحتفظ فيها المتغير من نوع `Character` وإن لم نقم بذلك سوف تعتبر القيمة المسندة من نوع `String`

الفرق بين النوعين String و Character

- نوع البيانات `String` هو نص يحتوي على خانات متعددة من أحرف ورموز وأرقام
○ مثل: `"Ahmed"` `"Hello world!"`, `"top10"`
- نوع البيانات `Character` هو نص يحتوي على خانة واحدة فقط قد تكون حرف أو رمز أو رقم
○ مثل: `"i"`, `"3"`, `"?"`

مقدمة في النوع Integer

النوع `Integer` واختصاره `Int` وهو نوع من أنواع البيانات ويمثل الأعداد الصحيحة أي أنه لا يحتوي على أجزاء كسرية. وقد تكون قيمته موجبة، أو سالبة، أو صفراً. لاحظ معي المثال التالي:

```
var myAge = 20
var currentTemperture = -1
```

تم إسناد القيمة 20 إلى المتغير myAge و القيمة -1 إلى المتغير currentTemperture

مقدمة في نوعي Float و Double

النوع Double هو نوع من أنواع البيانات ويمثل الأعداد الكسرية تصل إلى 15 خانة و قد تكون قيمة Double موجبة، أو سالبة، أو صفرًا. لاحظ معي المثال التالي :

```
let pi = 3.14159265358
var speed = 0.0
```

تم اسناد قيمة 3.14159265358 للثابت pi و 0.0 للمتغير speed استخدام الفاصلة يجعلهما من نوع Double.

ملاحظة : استخدام الفاصلة يجعل العدد من نوع Double بدلا من Int

النوع Float هي نوع من أنواع البيانات و يمثل الأعداد الكسرية يصل إلى 6 خانات، و قد تكون قيمة Float موجبة، أو سالبة، أو صفرًا. لاحظ معي المثال التالي:

```
var height: Float = 160.59
```

لاحظ أنه تم إسناد العدد 160.59 الى المتغير height وقمنا بتحديد النوع باستخدام النقطتان الرأسيتان ثم Float

ملاحظة : في المثال السابق قمنا باستخدام علامات : ثم حددنا نوع Float في جملة إنشاء المتغير height وذلك لجعل نوع البيانات التي يحتفظ فيها المتغير من نوع Float وإن لم نقوم بذلك سوف يعتبر القيمة المسندة من نوع Double

الفرق بين Float و Double

- النوع Double هو عدد كسري بدقة تصل إلى 15 خانة
- النوع Float هو عدد كسري بدقة تصل إلى 6 خانات

لاحظ معي المثال التالي:

```
var number1: Double = 12345.123456789
var number2: Float = 12345.123456789
```

```
print(number1)
print(number2)
```

قمنا في المثال أعلاه بإنشاء متغيرين أحدهما يستقبل `Double` والآخر `Float` ، وتم إسناد قيمة متطابقة لكل منهما ولكن لاحظ اختلاف النتائج حينما نقوم بطباعة قيمة المتغيران.

• المخرجات:

```
12345.123456789
12345.123
```

كما تلاحظ فإننا حصلنا على قيمة مختلفة وذلك لأن نوع `Float` أقل دقة من نوع `Double`.

التحويل بين Integer و Double

في Swift لا يمكن تحويل نوع المتغير بعد تعريفه، ولكن بإمكانك تحويل قيمة المتغير وحفظها بمتغير آخر أو طباعتها، على سبيل المثال التحويل من عدد عشري `Double` إلى عدد صحيح `Int`، لاحظ معي المثال التالي:

```
var hight1 = 1.5
var hight2 = Int(hight1)
print(hight2)
```

قمنا في المثال أعلاه بإنشاء متغيرين أحدهما `Double` ويحتوي القيمة ١.٥ والآخر قمنا بتحويل `hight1` إلى عدد صحيح باستخدام النوع `Int` وكتابته بين أقواس، وتم إسناد قيمته إلى `hight2`، لاحظ النتيجة عندما نقوم بطباعته:

• المخرجات:

```
1
```

كما تلاحظ فإننا حصلنا على قيمة عدد صحيح من دون كسور، وهذا لأن النوع `Int` عدد صحيح فتتم إزالة الأجزاء العشرية عند تحويله.

مقدمة في النوع Boolean

النوع Boolean واختصاره Bool وهو نوع من أنواع البيانات يحتفظ بإحدى القيمتين إما true أو false. لاحظ معي المثال التالي:

```
var isSunny = true
var isRaining = false
```

المثال السابق يفترض أن اليوم يوم مشمس و صافي لذلك تم إسناد true للمتغير isSunny و تم اسناد false للمتغير isRaining.

كيف تتعامل Swift مع أنواع البيانات

• نظرة على مفهوم Type safety

يستخدم المصطلح Type safety في علوم الحاسب للتعبير عن المدى الذي تنهى فيه لغة البرمجة عن أخطاء اختلاف الأنواع. خطأ اختلاف النوع هو خلل أو سلوك غير مرغوب فيه بالبرنامج يسببه التضارب بين أنواع البيانات المختلفة. هذا يعني أنك ستحصل على تنبيه بوجود خطأ عندما تحاول إسناد قيمة لمتغير من نوع مختلف عن القيمة التي تم إنشاء هذا المتغير عليها. مثلاً إذا حاولنا إسناد قيمة نص String في متغير يتوقع integer سوف نحصل على التنبيه التالي:

```
Cannot assign value of type 'String' to type 'Int'
```

تشرح رسالة الخطأ بوضوح تام وجود اختلاف في نوع البيانات وذلك لأننا نحاول إسناد قيمة String في متغير يتوقع Integer.

• مفهوم Type inference

ميزة Type inference تسمح لنا بإسقاط فقرة تحديد نوع البيانات من جملة إنشاء المتغير و الثابت وذلك لأن Swift تستطيع استنتاج نوع البيانات من القيمة الأولية التي تم إسنادها عند إنشاء المتغير أو الثابت كما هو موضح في المثال التالي:

```
var numberOfApples: Int = 10
var numberOfBananas = 10
```

في المثال السابق تم إنشاء متغيران أحدهما يحدد نوع البيانات بطريقة يدوية والآخر يستفيد من ميزة Type inference بإسقاط فقرة Int:

لاحظ أن المتغير numberOfApples تم تحديده من نوع integer بطريقة يدوية وذلك من خلال وضع النقطتان الرأسيتان ثم كتابة Int، على خلاف المتغير numberOfBananas الذي يقوم بإسقاط فقرة تحديد نوع البيانات وذلك لأنه يستنتج أنه من نوع integer بشكل تلقائي من القيمة الأولية التي تم إسنادها إليه.

○ الاستفادة من Type inference

في المثال التالي سوف تستنتج Swift نوع بيانات المتغير بناءً على القيم الأولية التي يتم إسنادها :

```
var age = 2102
var message = "Hello"
var isRaining = true
```

سوف تستنتج Swift أن المتغير `age` هو `Integer` وأن المتغير `message` هو `String`، وأن المتغير `isRaining` هو `Boolean`.

• مفهوم Type Annotation

هناك أوقات نود فيها تحديد نوع المتغير بشكل مباشر و صريح، مثال:

```
var height = 3.14
```

سوف تستنتج Swift أن المتغير `height` هو `Double`، لكن ماذا لو أردنا أن يكون `Float`؟ يمكننا تحديد نوع المتغير بشكل صريح بإضافة نقطتان رأسيّتان متبوعتان بالنوع الذي نود تحديده للمتغير كما هو موضح بالكود التالي:

```
var height: Float = 3.14
```

لاحظ أن فقرة تحديد نوع المتغير أنت بعد اسم المتغير.

عندما نعرف متغير بهذه الطريقة نحتاج إلى التأكد من أن القيمة الأولية من نفس النوع الذي تم تحديده، إذا حاولنا إعطاء متغير قيمة أولية من نوع مختلف عما قمنا بتحديدده فسوف نحصل على تنبيه بوجود خطأ.

• المراجع :

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

المعاملات Operators

مقدمة في العمليات Operators

هناك عدد من العمليات المختلفة التي يمكنك استخدامها أثناء البرمجة مثل العمليات الرياضية وعمليات المقارنة والعمليات المنطقية وغيرها من العمليات المختلفة، وسنتحدث في هذا الجزء عن مجموعة من أهم العمليات التي توفرها لغة swift.

المعاملات الحسابية Arithmetic Operators

ببساطة، يمكنك تنفيذ العمليات الرياضية المختلفة باستخدام الصيغة التالية:

```
result = left op right
```

حيث يمثل **op** نوع العملية الرياضية المراد استخدامها، ويمثل كل من **left** و **right** القيمتين (أو المتغيرين أو الثابتين) اللذين سيتم تنفيذ العملية **op** عليهما. يوضح الجدول التالي أنواع العمليات الحسابية:

وظيفتها	رمز العملية	اسم العملية
تقوم بتنفيذ عملية الجمع.	+	Addition
تقوم بتنفيذ عملية الطرح.	-	Subtraction
تقوم بتنفيذ عملية القسمة.	/	Division
تقوم بتنفيذ عملية الضرب.	*	Multiplication
تقوم بتنفيذ عملية إرجاع باقي القسمة.	%	Modulus (Remainder)

لتوضيح الفكرة، دعنا نقوم باستبدال **op** بأحد العمليات السابقة، وسنقوم هنا باختيار الجمع **+** كمثال يمكن تطبيقه على باقي العمليات الأخرى. يوضح السطر التالي هذا الأمر:

```
var result = 5 + 2  
print(result)
```

• المخرجات:

7

في المثال أعلاه، قمنا بتنفيذ عملية الجمع باستخدام **+** وسيتم تخزين ناتج العملية وهو في هذه الحالة 7 في المتغير **result**.

معاملات الإسناد المركبة Compound assignment operators

هي عبارة عن عملية تتم على مرحلتين، في المرحلة الأولى تنفذ عملية حسابية على قيمة المتغير الأساسية ومن ثم تقوم في المرحلة الثانية بإسناد الناتج كقيمة جديدة للمتغير، على النحو التالي:

```
result op right
```

حيث يمثل `op` نوع العملية الرياضية المُراد استخدامها، ويمثل `right` القيمة (أو المتغيرين أو الثابتين) اللذين سيتم تنفيذ العملية `op` عليهما. يوضح الجدول التالي أنواع العمليات :

وظيفتها	رمز العملية	اسم المعامل
تقوم بتنفيذ عملية الجمع ثم تسند القيمة إلى المتغير <code>result</code> .	<code>+=</code>	Addition assignment operator
تقوم بتنفيذ عملية الطرح ثم تسند القيمة إلى المتغير <code>result</code> .	<code>-=</code>	Subtraction assignment operator
تقوم بتنفيذ عملية القسمة ثم تسند القيمة إلى المتغير <code>result</code> .	<code>/=</code>	Division assignment operator
تقوم بتنفيذ عملية الضرب ثم تسند القيمة إلى المتغير <code>result</code> .	<code>*=</code>	Multiplication assignment operator
تقوم بتنفيذ عملية إرجاع باقي القسمة ثم تسندھا إلى المتغير <code>result</code> .	<code>%=</code>	Modulus (Remainder) assignment operator

لتوضيح الفكرة، دعنا نقوم باستبدال `op` بأحد العمليات السابقة، وسنقوم هنا باختيار الجمع `+=` كمثال يمكن تطبيقه على باقي العمليات الأخرى. يوضح السطر التالي هذا الأمر:

```
var result = 6
result += 2 // result = result + 2
print(result)
```

• المخرجات:

8

في المثال السابق قمنا بإنشاء متغير `result` وأسندنا إليه قيمة 6 ثم أضفنا إليه 2 باستخدام عملية الإسناد المركبة `+=` و الآن `result` تحتفظ بقيمة 8

معاملات المقارنة Comparison Operators

يمكنك تنفيذ عمليات المقارنة المختلفة باستخدام الصيغة التالية (مع التنبيه على أنه يمكنك استخدامها في سياقات برمجية أخرى دون إسنادها إلى قيمة، مثل استخدامها كشرط مع جملة if كما سنرى لاحقاً):

```
result = left op right
```

يمثل `op` نوع عملية المقارنة المُراد استخدامها ويمثل كل من `left` و `right` القيمتين (أو المتغيرين أو الثابتين) اللذين سيتم تنفيذ العملية `op` عليهما. وستكون نتيجة عمليات المقارنة هي قيمة من نوع `boolean`، أي أن ناتج المقارنة سيكون إما `true` أو `false`. يوضح الجدول التالي هذا الأمر:

وظيفتها	رمز العملية	اسم العملية
تعيد <code>true</code> في حال كان <code>left</code> أكبر من <code>right</code> .	>	Greater Than
تعيد <code>true</code> في حال كان <code>left</code> أصغر من <code>right</code> .	<	Less Than
تعيد <code>true</code> في حال كان <code>left</code> أكبر من أو يساوي <code>right</code> .	>=	Greater Than or Equal
تعيد <code>true</code> في حال كان <code>left</code> أصغر من أو يساوي <code>right</code> .	<=	Less Than or Equal
تعيد <code>true</code> في حال كان <code>left</code> يساوي <code>right</code> من حيث القيمة فقط.	==	Equal
تعيد <code>true</code> في حال كان <code>left</code> لا يساوي <code>right</code> من حيث القيمة فقط.	!=	Not Equal

دعنا نقوم الآن باستبدال `op` بأحد المعاملات السابقة، وسنقوم هنا باستبدالها بأكبر من `>`، ونستخدمها بصيغة `swift` على النحو التالي:

```
var result = 5 > 2
```

في هذه الحالة، قمنا بتنفيذ عملية المقارنة باستخدام `>` وسيتم تخزين الناتج وهو `true` في المتغير `result`.

معامل الشرط الثلاثي Ternary Conditional Operator

عند استخدام معامل الشرط الثلاثي سيتم اختبار شرط ما، وفي حال تحقق الشرط فإنه يعود بالقيمة الأولى، وفي حال لم يتحقق فإنه يعود بالقيمة الثانية كما هو موضح بالشكل التالي:

```
result = condition ? firstExpression : secondExpression
```

- تمثل `condition` التعبير الشرطي
- تمثل `firstExpression` الأمر البرمجي الذي سيتم تنفيذه عندما تكون نتيجة الشرط صحيحة ويكون بعد ؟

- تمثل `secondExpression` الأمر البرمجي الذي سيتم تنفيذه عندما تكون نتيجة الشرط خاطئة ويكون بعد :

لتوضيح الفكرة، لاحظ معي المثال التالي:

```
var khalidAge = 22
var salehAge = 30
var result = khalidAge > salehAge ? "Khalid is older than Saleh" : "Khalid is not older than Saleh"
print(result)
```

- المخرجات:

Khalid is not older than Saleh

في المثال السابق قمنا باستخدام معاملي الشرط الثلاثي، والذي بدوره سيقوم باختبار ما إذا المتغير `khalidAge` أكبر من المتغير `salehAge`، وبناءً على ذلك قام بإسناد القيمة المناسبة إلى المتغير `result`. لاحظ أنه تم إسناد القيمة `Khalid is not older than Saleh` للمتغير `result`، وذلك لعدم تحقق الشرط `khalidAge > salehAge`.

المعاملات المنطقية Logical Operators

هناك ثلاث معاملات منطقية اثنتان منهما تكتب بالصيغة التالية (مع التنبيه على أنه يمكنك استخدامها في سياقات برمجية أخرى دون إسنادها إلى قيمة، مثل استخدامها كشرط مع جملة `if` كما سنرى لاحقاً):

```
result = left op right
```

يمثل `op` نوع المعامل المنطقي المراد استخدامه ويمثل كل من `left` و `right` القيمتين (أو المتغيرين أو الثابتين) اللذين سيتم تنفيذ المعامل `op` عليهما. وستكون نتيجة العمليات المنطقية هي قيمة من نوع `boolean`، أي أن ناتج المقارنة سيكون إما `true` أو `false`.

أما بالنسبة للمعامل المتبقي، أي المعامل الثالث، فإنه يكتب بالصيغة التالية:

```
result = op right
```

يوضح الجدول التالي المعاملات المنطقية:

وظيفتها	رمز	اسم
---------	-----	-----

العملية	العملية	
And	&&	ستكون النتيجة true في حالة واحدة فقط، وهي إن كانت left و right كلاهما true.
Or		ستكون النتيجة false في حالة واحدة فقط، وهي إن كانت left و right كلاهما false.
Not	!	يقوم بعكس قيمة right في حال كانت right تساوي true فستكون النتيجة false والعكس صحيح.

توضح الأسطر التالية استخدام المعاملات السابقة برمجياً:

```
var first = true, second = false
var andResult = first && second // false
var orResult = first || second // true
var notResult = !first // false
notResult = !second // true
```

• المراجع :

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

مقدمة في جمل القرار Control Flow

في المواضيع السابقة تطرقنا إلى عدة مفاهيم برمجية في لغة swift وجميع هذه المفاهيم يتم تنفيذها سطر تلو السطر بشكل تسلسلي من الأعلى إلى الأسفل في ملف swift واحد، ولكن في التطبيقات الفعلية قد يكون لديك أكثر من ملف تتعامل معه ويكون الكود موزع بين هذه الملفات وقد ترغب بتنفيذ جزء من الكود وتجاهل بعضه.

في هذا الفصل سنتعرف على طريقة التحكم بتنفيذ الأكواد عن طريق Control Flow باستخدام هذه المفاهيم:

• الشروط Conditions

- جملة if الشرطية.
- جملة switch الشرطية.

• جمل التكرار Loops

- جملة for in.
- جملة while.
- جملة repeat while.

الشروط Conditions

عند القيام بكتابة برنامج ستواجه عددًا من الحالات التي تريد فيها تنفيذ أوامر معينة عند تحقق شرط محدد، أي أنك لا تريد تنفيذ تلك الأوامر بشكل مباشر، بل تريد أن يتم تنفيذها فقط عند تحقق شرط أو مجموعة من الشروط لتوضيح الفكرة، افترض أنك تريد تنفيذ مجموعة من الأوامر في حال كان عمر الطالب أكبر من 20 وتريد تنفيذ أوامر أخرى في حال لم يكن كذلك، هنا سنكون بحاجة إلى استخدام ما يُسمى بالشروط Conditions لنتمكن من تحقيق هذا الغرض.

• نظرة على if statement

أحد الطرق التي يمكنك استخدامها لربط الأوامر بالشروط هي استخدام جملة `if`، ببساطة كل ما ستقوم به هو ربط مجموعة من الأوامر بشرط أو مجموعة من الشروط، وعندها لن يتم تنفيذ تلك الأوامر إلا إذا تحقق ذلك الشرط أو تلك الشروط لفهم الفكرة الأساسية، لاحظ معي الشكل العام لجملة `if` في الأسطر التالية:

```
if condition {  
statement1  
statement2  
...  
statementN  
}
```

سيتم تنفيذ جميع الأوامر الموجودة بين الأقواس `{ }` والمشار إليها هنا بكلمة `statement` فقط عندما يتحقق الشرط أو مجموعة الشروط التي أُشير لها بكلمة `condition`. أي أن جميع الأوامر الموجودة "بداخل" جملة `if` هي أوامر لن يتم تنفيذها حتى يتحقق الشرط `condition` (أي تكون نتيجته `true`)، عدا ذلك سيتم تجاوزها، توضح الأسطر التالية مثال على استخدام جملة `if` في `swift`:

```
var age = 25  
if age > 18 {  
print("You are an adult")  
}
```

في هذا المثال ستتم طباعة الرسالة في حالة واحدة فقط وهي أن يكون `age` أكبر من 18، وبما أن الشرط تحقق فعلاً في هذه الحالة فسوف تتم طباعة الرسالة.

○ المخرجات:

```
You are an adult
```

• استخدام else مع if

في حال عدم تحقق الشرط في if statement سيتم تجاهل الأوامر، ولكن ماذا لو أردنا تنفيذ أوامر برمجية أخرى عند عدم تحقق الشرط داخل if statement عندها يمكننا استخدام else لهذا الغرض ولتوضيح الفكرة دعنا نُعيد كتابة المثال السابق ليحتوي على else على النحو الموضح في الأسطر التالية:

```
var age = 17
if age > 18 {
  print("You are an adult")
} else{
  print("You are a minor")
}
```

في هذه الحالة سيتم طباعة الرسالة الموجودة في جزء else، لأن الشرط في if لم يتحقق وذلك لكون العمر المخزن في المتغير age أقل من 18.
o المخرجات:

```
You are a minor
```

• اختبار أكثر من شرط باستخدام else if

هناك حالات برمجية نحتاج فيها إلى اختبار أكثر من مسار وليس مسارين فقط مثل if-else. ولتوضيح الفكرة دعنا ننظر إلى إشارة المرور كمثال ففيها سنجد أن لدينا ثلاث خيارات: أحمر، أصفر، أخضر، وفي كل حالة أو لون من ألوان الإشارة سيكون لدينا أمر أو أكثر سيتبعه أصحاب المركبات، ففي حالة الأحمر: قف، وفي حالة الأصفر: قلل السرعة، وفي حالة الأخضر: سر. يمكنك استخدام else if مع جملة if في حال كان لديك أكثر من شرط ولتوضيح الفكرة دعنا نقوم بتنفيذ مثال إشارة المرور على النحو الموضح في الأسطر التالية:

```
var trafficLight = "red"
if trafficLight == "green" {
  print("Go!")
} else if trafficLight == "yellow"{
  print("Slow Down")
} else {
  print("Stop!")
}
```

○ المخرجات:

Stop!

ببساطة، سيتم اختبار الشرط الأول إذا لم يتحقق، وسيتم الانتقال للشرط الثاني إذا لم يتحقق، سيتم الانتقال لجزء `else` وتنفيذه بشكل مباشر كونه غير مرتبط بشرط، لاحظ كيف تم استخدام `else if` والتي سيتم التحقق منها في حال عدم تحقق الشرط الموجود في `if`.

● نظرة على Switch Statement

نستخدم `switch` لاختبار قيمة معينة مع عدة احتمالات تستقبل `switch` قيمة وتقارنها بعدد من الاحتمالات، وتنفذ مجموعة التعليمات البرمجية التابعة للحالة `case` المتوافقة ولتوضيح الفكرة لاحظ معي المثال التالي:

```
var trafficLight = "red"
switch trafficLight {
case "green":
print("Go!")
case "yellow":
print("Slow Down")
case "red":
print("Stop!")
default:
print("Invalid light")
}
```

○ المخرجات:

Stop!

في المثال السابق قمنا بإنشاء متغير `trafficLight` وأسندنا إليه القيمة `red`، ثم قمنا باستخدام `switch` وأسندنا إليها قيمة `trafficLight` لتتم عملية المقارنة عليها وتنفيذ مجموعة التعليمات البرمجية التابعة للحالة المطابقة لها.

● استخدام default للتعامل مع الحالات الغير متوقعة

تستخدم default في نهاية switch وتقوم بالتعامل مع الحالات الغير متوقعة، مما يعني أنها تنفذ في حال لم تتحقق أي من الحالات case السابقة، لفهم هذا الأمر لاحظ معي المثال التالي:

```
var trafficLight = "blue"
switch trafficLight {
case "green":
print("Go!")
case "yellow":
print("Slow Down")
case "red":
print("Stop!")
default:
print("Invalid light")
}
```

○ المخرجات:

Invalid light

في المثال السابق قمنا بتعديل قيمة المتغير trafficLight وأسندنا إليه القيمة blue، لاحظ أن القيمة المسندة لم تتطابق مع أي case وبالتالي تم تنفيذ مجموعة التعليمات التابعة للحالة الافتراضية default.

ملاحظة: في Swift لانحتاج لاستخدام break في نهاية كل case لإيقاف عملية switch وذلك لأنها فقط تنفذ مجموعة التعليمات التابعة للحالة المطابقة ثم تتوقف ولا تنتقل للحالة التالية على خلاف بعض لغات البرمجة وللقيام بذلك يمكننا استخدام fallthrough.

• استخدام fallthrough في Switch

نستخدم عبارة fallthrough لتنفيذ الحالة التي تلي الحالة الصحيحة يوضح المثال التالي كيفية استخدام fallthrough:

```
var trafficLight = "yellow"

switch trafficLight {
```

```
case "green":

print("Go!")

case "yellow":

print("Slow Down")

fallthrough

case "red":

print("Stop!")

default:

print("Invalid light")

}
```

○ المخرجات:

```
Slow Down
Stop!
```

في المثال السابق تم استخدام `fallthrough` في نهاية الحالة `yellow` وهي الحالة المتطابقة مع الشرط، مما سيؤدي إلى تنفيذ الحالة التي تليها وبالتالي تم تنفيذ مجموعة التعليمات التابعة لكل من `yellow` و `red`.

ملاحظة: يمكنك استخدام `switch` بدلاً من `if` بل وقد يفضل في بعض الحالات استخدام `switch` حيث أنها تستخدم لمقارنة قيمة متغير واحد فقط بعدة حالات.

- الفرق بين **switch statement** و **if statement**
 - تستقبل **switch statement** قيمة **value** ويتم تنفيذ الأوامر التابعة للحالة **case** المتطابقة.
 - تستقبل **if statement** شرط **condition** ويتم تنفيذ الأوامر التابعة للشرط المتحقق.

مقدمة في جمل التكرار Loop

قد نحتاج في بعض الحالات إلى تكرار مجموعة من التعليمات البرمجية لعدد من المرات، 10، 20

- جملة التكرار **while**

يمكننا تكرار تنفيذ أوامر برمجية وفقاً لشرط معين باستخدام **while**، أي أنه سيتم تكرار مجموعة تعليمات برمجية طالما أن الشرط صحيح، ولاتتوقف عملية التكرار إلا عندما يصبح الشرط غير صحيح، يوضح المثال التالي كيفية القيام بذلك:

```
var number = 5

while number > 0 {

print("Hello world")

number -= 1

}
```

- المخرجات:

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

في المثال السابق قمنا بإسناد الرقم 5 إلى المتغير **number** ثم قمنا باستخدام **while** ووضعنا لها شرطاً، أن تكون قيمة المتغير **number** أكبر من 0 ، وفي كل مرة يتحقق فيها هذا الشرط تتم طباعة **Hello world** وطرح 1 من قيمة المتغير **number** حتى أصبحت قيمة **number** هي 0، أي أنها لم تعد أكبر من 0 وبذلك لا يتحقق الشرط وعندها تتوقف عملية التكرار.

- **جملة التكرار repeat-while**

يمكننا أيضاً تكرار تنفيذ أمر برمجي وفقاً لشرط معين باستخدام `repeat-while`، ولكنها تنفذ الأوامر البرمجية مرة واحدة قبل أن تبدأ في التحقق من الشرط، ويوضح المثال التالي كيفية استخدامها:

```
var number = 5

repeat {

print("Hello world")

number -= 1

} while number > 0
```

- **المخرجات:**

```
Hello world
Hello world
Hello world
Hello world
Hello world
```

في المثال السابق قمنا بإسناد الرقم 5 إلى المتغير `number` ثم قمنا باستخدام `repeat-while` ووضعنا لها شرطاً وهو أن تكون قيمة المتغير `number` أكبر من 0.

ملاحظة: تقوم `while` بالتحقق من الشرط قبل تنفيذ الأوامر بينما `repeat-while` تتحقق من الشرط بعد عملية التكرار الأولى.

- **جملة التكرار for-in**

يمكننا تكرار تنفيذ أمر برمجي باستخدام `for-in` والتي تقوم بتكرار التعليمات البرمجية لكل عنصر في نطاق معين أو مجموعة، على سبيل المثال `Array` أو `Set`.

- **استخدام for-in مع المصفوفات**

تستخدم `for-in` مع المصفوفات لتكرار مجموعة من الأوامر على كل عنصر من عناصر المصفوفة ويوضح المثال التالي كيفية القيام بذلك:

```
var basketOfFruits = ["Apples", "Bananas", "Oranges"]

for fruit in basketOfFruits {

    print(fruit)

}
```

○ المخرجات:

```
Apples
Bananas
Oranges
```

لاحظ أن قيمة `fruit` تغيرت في كل مرة تم فيها تنفيذ أمر الطباعة، ففي المرة الأولى من التكرار كانت قيمتها تمثل العنصر الأول في المصفوفة وهو `Apples` ثم في المرة الثانية العنصر الثاني `Bananas` وأخيرًا في المرة الثالثة العنصر الثالث والأخير `.Oranges`.

● استخدام `for-in` مع النطاقات

تستخدم `for-in` مع النطاقات لتكرار مجموعة تعليمات برمجية لكل عدد في النطاق، على النحو التالي:

```
for index in 1...3 {

    print("Hello", index)

}
```

يمثل النطاق `1...3` الأعداد من 1 إلى 3 أي أنه سيتم تنفيذ الأمر ثلاث مرات.

○ المخرجات:

```
Hello 1
Hello 2
Hello 3
```

في المثال أعلاه قام البرنامج بتكرار طباعة Hello مع قيمة index والتي تغيرت في كل مرة من مرات التكرار، ففي المرة الأولى كانت 1 وهي بداية النطاق وفي المرة الثانية كانت 2 وفي المرة الثالثة كانت 3 وهي نهاية النطاق وعندها توقفت عملية التكرار.

• استخدام continue

تجعل continue جملة التكرار تتوقف وتنتقل لتكرار التعليمات البرمجية للعنصر التالي. يوضح المثال التالي كيف يمكننا استخدام عبارة continue لطباعة الأرقام الفردية فقط في نطاق الأعداد من 1 إلى 10:

```
for index in 1...10 {
  if index % 2 == 0 {
    continue
  }
  print("\(index) is odd")
}
```

○ المخرجات:

```
1 is odd
3 is odd
5 is odd
7 is odd
9 is odd
```

في المثال السابق قمنا بتكرار مجموعة تعليمات برمجية لكل عدد في نطاق الأعداد من 1 إلى 10، وفي كل مرة يقوم البرنامج بالتحقق مما إذا كان العدد يقبل القسمة على العدد 2، فإن كان كذلك فسيتم استدعاء continue والتي ستقوم بالانتقال إلى العدد التالي في النطاق، وفي حال كان لا يقبل القسمة على 2 سوف تتم طباعة العدد مع عبارة is odd.

• استخدام break

تجعل break جملة التكرار تتوقف بشكل كامل ويوضح المثال التالي كيفية استخدام break مع for-in.

```
for index in 1...10 {  
    if index % 2 == 0 {  
        break  
    }  
  
    print("\(index) is odd")  
}
```

○ المخرجات:

```
1 is odd
```

في المثال السابق قمنا بتكرار مجموعة تعليمات برمجية لكل عدد في نطاق الأعداد من 1 إلى 10، وفي كل مرة يقوم البرنامج بالتحقق مما إذا كان العدد يقبل القسمة على العدد 2، فإن كان كذلك فسيتم استدعاء `break` والتي سوف تنهي عملية التكرار بشكل كامل.

• المراجع :

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

المصفوفات Array

مقدمة في مفهوم المصفوفات Array

فكر في المصفوفة على أنها متغير أو ثابت يتكون من مجموعة من القيم أو العناصر، ويمكن الوصول لكل خانة توجد بها قيمة أو عنصر من خلال رقم يدعى `index` والذي يمثل ترتيب القيمة أو العنصر بداخل المصفوفة.

إنشاء المصفوفة Array

لتعريف مصفوفة سنستخدم الأقواس المربعة `[]` وسنقوم بوضع القيم بداخلها، ونفصل بين كل قيمة والأخرى بفاصلة `,`، لتوضيح الفكرة، دعنا نقوم بتعريف مصفوفة بإسم `arrayNumbers` تحتوي على ثلاثة أعداد 1، 2 و 3 كما هو موضح في السطر التالي:

```
var arrayNumbers = [1, 2, 3];
```

في المثال السابق قمنا بإنشاء متغير باسم `arrayNumbers` يحتفظ بمصفوفة من الأعداد. لاحظ ان جميع القيم من نوع بيانات واحد وهو `Int`.

ملاحظة: يجب ان تكون عناصر المصفوفة من نفس النوع اذا لم تكن المصفوفة من نوع `[Any]`

إنشاء مصفوفة لا تحتوي على عناصر

إذا أردنا إنشاء مصفوفة لا تحتوي على قيم، يجب تحديد نوع القيم المراد تخزينها في المصفوفة بين الأقواس المربعة `[]` متبوعةً بقوسين `()` ، كما هو موضح في المثال التالي:

```
var arrayA = [String]()
var arrayB = [Double]()
var arrayC = [Int]()
var arrayD = [Bool]()
var arrayE = [Any]()
```

في المثال السابق تم إنشاء خمس مصفوفات لا تحتوي على قيم مع تحديد نوع القيم.

الوصول لعناصر المصفوفة

ذكرنا سابقاً أن المصفوفة تحتوي على أكثر من قيمة، وأن كل قيمة مُرتبطة برقم يُسمى `index`، والذي يساعدنا على الوصول إلى تلك القيمة سواء لجلبها أو لتغييرها إلى قيمة أخرى. يبدأ ترقيم خانات وقيم المصفوفة من اليسار لليمين، ويبدأ العد من `index` رقم 0. لتوضيح الفكرة، سنقوم بوضع رقم `index` فوق كل خانة أو قيمة من قيم المصفوفة كما هو موضح في المثال التالي:

```
// index:           0           1           2
var arrayNames = ["Ahmed","Khaled","Muhammad"]
print(arrayNames[0])
print(arrayNames[2])
```

في المثال السابق، قمنا بإنشاء مصفوفة تحتفظ بمجموعة من الأسماء. ثم قمنا بطباعة القيمة التابعة للعنصر رقم 0 و 2 وستكون المخرجات على النحو التالي:

Ahmed

Muhammad

إضافة عنصر جديد إلى المصفوفة

قد نرغب في بعض الحالات في إضافة عنصر جديد إلى المصفوفة، وللقيام بذلك يمكننا استخدام دالة `append` والتي ستقوم بإضافة العنصر إلى آخر المصفوفة كما هو موضح في المثال التالي:

```
var arrayOne = [1,2]
arrayOne.append(3)
print(arrayOne)
```

• المخرجات:

[1, 2, 3]

سوف تتضمن مصفوفة `arrayOne` الآن الأعداد 1 ، 2 و 3 وذلك لأننا قمنا بإضافة قيمة 3 باستخدام الأمر البرمجي `.append`.

ملاحظة : يمكن إضافة عناصر إلى المصفوفات فقط إذا كانت من المتغيرات وسوف نحصل على خطأ لو حاولنا القيام بذلك لمصفوفات الثوابت

إزالة العناصر من المصفوفة

عند رغبتنا في إزالة عنصر أو أكثر من المصفوفة فإن هناك أكثر من طريقة يمكننا استخدامها للقيام بذلك، ومنها:

- دالة `removeFirst` لإزالة العنصر الأول في المصفوفة
- دالة `removeLast` لإزالة العنصر الأخير في المصفوفة
- دالة `remove` لإزالة عنصر في موقع محدد
- دالة `removeAll` لإزالة كل عناصر المصفوفة

يوضح المثال التالي كيف يمكن استخدام هذه الدوال:

```
var arrayTwo = [1,2,3,4,5]
arrayTwo.removeFirst()
```

في البداية كانت مصفوفة `arrayTwo` تحتوي على خمس عناصر ولكننا قمنا بإزالة العنصر الأول في المصفوفة و بالتالي أصبحت الآن تحتفظ بالأعداد [5 , 4 , 3 , 2]

```
arrayTwo.removeLast()
```

الآن سوف تتضمن مصفوفة `arrayTwo` الأعداد [2 , 3 , 4] وذلك لأننا قمنا بإزالة العنصر الأخير

```
arrayTwo.remove(at: 1)
```

الآن سوف تتضمن مصفوفة `arrayTwo` الأعداد [2 , 4] وذلك لأننا قمنا بإزالة العدد 3 الموجود في `index` رقم 1

```
arrayTwo.removeAll()
```

الآن تمت إزالة كل عناصر `arrayTwo` ، لأننا قمنا باستخدام أمر `removeAll`

ملاحظة : يمكن إزالة عناصر من مصفوفة فقط إذا كانت من المتغيرات و سوف نحصل على خطأ لو حاولنا القيام بذلك للشوايت

حساب عدد عناصر المصفوفة

لمعرفة عدد العناصر في المصفوفة، يمكننا استخدام `count` والتي ستعود بعدد عناصر المصفوفة. يوضح المثال التالي كيفية القيام بذلك:

```
var arrayNames = ["Ahmed","Khaled","Muhammad"]
print(arrayNames.count)
```

• المخرجات:

3

كما تلاحظ فإن مصفوفة `arrayNames` تحتوي على ثلاث أسماء و بالتالي فإن العدد الذي سوف تتم طباعته عند استخدام `count` هو 3

ملاحظة : لاحظ انه يتم حساب عدد العناصر داخل المصفوفة ابتداء من 1 وليس 0 كما في `index`.

التحقق من أن المصفوفة لا تحتوي على عناصر:

للتحقق مما إذا كانت المصفوفة تحتوي على عناصر، فإننا نستخدم `isEmpty`. والتي ستعود بالقيمة `true` إذا كانت المصفوفة لا تحتوي على عناصر و `false` إذا كانت تحتوي على أي عناصر. يوضح المثال التالي كيف يتم ذلك:

```
var arrayOne = [1,2]
var arrayTwo = [Int]()
var boolOne = arrayOne.isEmpty
var boolTwo = arrayTwo.isEmpty
```

سوف تسند `false` للمتغير `boolOne` لأن المصفوفة `arrayOne` تحتوي على عنصرين و سوف تسند `true` للمتغير `boolTwo` لأن المصفوفة `arrayTwo` لا تحتوي على عناصر.

توزيع عناصر المصفوفة بشكل عشوائي

يمكن إعادة ترتيب عناصر المصفوفة وتوزيعها بشكل عشوائي بسهولة باستخدام دالة `shuffle`. يوضح المثال التالي كيف نقوم بذلك:

```
var arrayOne = [1, 2, 3, 4, 5, 6]
arrayOne.shuffle()
```

لو قمنا بطباعة عناصر المصفوفة `arrayOne` سوف نجد أنه تم توزيعها بشكل عشوائي وذلك لأننا قمنا باستخدام أمر `shuffle`.

مقدمة في Two-dimensional array

وهي عبارة عن مصفوفة تحتوي على مصفوفات أخرى أي أن كل عنصر من عناصرها يمثل مصفوفة. يوضح المثال التالي كيف يمكن إنشاؤها:

```
var multiArrayOne = [[1, 2], [3, 4], [5, 6]]
```

في هذا المثال السابق قمنا بإنشاء `Two-dimensional array` مما يعني أنها مصفوفة داخلها مصفوفات، و كما تلاحظ فإن هذه المصفوفات تحتوي على أعداد

الوصول لعنصر من عناصر Two-dimensional array

للوصول لعنصر من عناصر Two-dimensional array ، يتم استدعاء معرف المصفوفة متبوعاً [رقم العنصر
في البعد الأول] ثم [رقم العنصر في البعد الثاني]

```
let multiArray = [[1, 2], [3, 4], [5, 6]]  
let value = multiArray[0][1]
```

الآن المتغير **value** سوف يحتفظ بالقيمة العدد 2

• المراجع :

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

مقدمة في مفهوم Dictionary

يخزن Dictionary مجموعة من القيم value ويربط كل قيمة بمعرف خاص بها يسمى بالمفتاح (Key)، أي أن كل عنصر من عناصر Dictionary يتكون من مفتاح Key وقيمة Value، تتميز العناصر في Dictionary أنها غير مرتبة، وتتميز المفاتيح بأنها فريدة ولا تتكرر، بعكس القيم يُمكن تكرارها.

إنشاء Dictionary

يوضح المثال التالي كيفية إنشاء Dictionary يحتوي على القيم التالية:

```
value, key : 100, Ahmed 70, Khaled, 100, Muhammad  
var studentGrades = ["Ahmed": 100, "Khaled": 70, "Muhammad":  
100]
```

في المثال أعلاه قمنا بإنشاء Dictionary اسمناه studentGrades يتكون من ثلاث عناصر وهي عبارة عن درجات الطلاب. كل عنصر يحتوي على مفتاح وقيمة، والمفتاح في هذا المثال هو اسم الطالب والقيمة هي درجته.

إنشاء Dictionary لاحتوي على عناصر

إذا أردنا إنشاء Dictionary لاحتوي على قيم، فسنحتاج إلى تحديد نوع القيم التي سوف تخزن فيه لاحقاً متبوعةً بالقوسين () كما هو موضح في المثال التالي:

```
var dictionaryOne = [String: String]()  
var dictionaryTwo = [Int: String]()  
var dictionaryThree = [String: Bool]()  
var dictionaryFour = [String: Double]()
```

لاحظ أنه تم تحديد نوع عناصر Dictionary على النحو التالي:

() [نوع قيمة : نوع المفتاح]

الوصول لعناصر Dictionary

للوصول لعنصر من عناصر Dictionary يتم استدعاء المتغير أو الثابت متبوعًا بالمفتاح الخاص بالقيمة المطلوبة، يوضح المثال التالي كيفية القيام بذلك:

```
let studentGrades = ["Ahmed": 100, "Khaled": 70, "Muhammad": 95]
var grade = studentGrades["Ahmed"]
print(grade)
```

سيقوم أمر الطباعة في المثال السابق بطباعة درجة الطالب Ahmed وهي 100.

المخرجات:

```
Optional(100)
```

لاحظ أن قيمة 100 تأتي داخل قوس تسبقها كلمة Optional وسيتم مناقشة هذا النوع لاحقًا.

إضافة عنصر جديد إلى Dictionary

لإضافة عنصر جديد إلى Dictionary نقوم باستدعاء المتغير متبوعًا بالمفتاح الجديد بين قوسين [] ثم نسند إليه القيمة ويوضح المثال التالي كيفية القيام بذلك:

```
var countries = ["US": "United States", "IN": "India", "UK": "United Kingdom"]
countries["FR"] = "France"
```

في السطر الثاني قمنا بإضافة عنصر جديد إلى countries، المفتاح FR وقيمته هي France

تحديث قيمة عنصر في Dictionary

لتحديث قيمة عنصر في Dictionary، نقوم باستدعاء المتغير متبوعًا بالمفتاح المراد تحديث قيمته، ثم نقوم بإسناد القيمة الجديدة له، يوضح المثال التالي كيفية القيام بذلك:

```
var countriesShortNames = ["US": "United States", "IN": "India", "UK": "United Kingdom"]
countries["UK"] = "Great Britain"
```

لاحظ في السطر الثاني قمنا بتحديث القيمة التابعة لـ UK إلى Great Britain.

ملاحظة: عند تنفيذ هذا الأمر البرمجي سيقوم البرنامج من التحقق من وجود المفتاح UK داخل المتغير countriesShortNames، في حال تواجد المفتاح سيتم تحديث قيمته (كما هو موضح في المثال أعلاه)، ولكن في حال عدم تواجده فإنه سيقوم بإضافته كعنصر جديد إلى countriesShortNames.

إزالة العناصر من Dictionary

عند رغبتنا في إزالة عنصر أو أكثر من Dictionary فإن هناك أكثر من طريقة يُمكن استخدامها للقيام بذلك، ومنها:

- إزالة عنصر محدد باستخدام المفتاح `removeValue`.
- إزالة كل العناصر `removeAll`.

يوضح المثال التالي كيفية استخدام هذه الدوال:

```
var countries = ["US": "UnitedStates", "IN": "India", "UK": "United Kingdom"]
countries.removeValue(forKey:"UK")
```

```
print(countries)
countries.removeAll()
print(countries)
```

المخرجات:

```
["US": "UnitedStates", "IN": "India"]
[:]
```

في السطر الثاني تم إزالة العنصر الذي يحتوي على المفتاح UK وفي السطر الرابع تم إزالة جميع العناصر. ملاحظة: يمكن إزالة العناصر من Dictionary فقط إذا كانت من المتغيرات وسوف نحصل على خطأ لو حاولنا القيام بهذا للشوابة.

حساب عدد عناصر Dictionary

لمعرفة عدد العناصر في Dictionary يمكننا استخدام count ويوضح المثال التالي كيفية القيام بذلك:

```
let countries = ["US": "United States", "IN": "India", "UK":
"United Kingdom"]
print(countries.count)
```

المخرجات:

3

ملاحظة: القيمة التي يتم إرجاعها بواسطة خاصية count هي عدد الأزواج الموجودة في Dictionary.

التحقق من أن Dictionary لا تحتوي على عناصر:

للتحقق مما إذا كان Dictionary يحتوي على عناصر، فإننا نستخدم الخاصية isEmpty. والتي ستعود بالقيمة true إذا كان Dictionary لا يحتوي على عناصر، false إذا كان يحتوي على عناصر ويوضح المثال التالي كيف يتم ذلك:

```
let countries = ["US": "United States", "IN": "India", "UK":
"United Kingdom"]
var boolOne = countries.isEmpty
print(boolOne)
```

المخرجات:

false

نلاحظ أن قيمة المتغير boolOne تساوي false وذلك لأن المتغير countries يحتوي على عناصر.

المراجع:

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

المجموعات Set

مقدمة في مفهوم Set

تقوم Set بتخزين مجموعة قيم من نفس النوع كما هو الحال في المصفوفات، ولكن تتميز القيم في Set بأنها غير مرتبة وغير قابلة للتكرار.

ملاحظة: العناصر التي تحتفظ فيها المجموعات يجب أن تكون من نفس النوع (String, Int , Bool, Double).

إنشاء المجموعات Set

يوضح المثال التالي كيفية إنشاء مجموعة ثابتة تحتفظ بعناصر من نوع String:

```
let fruits: Set = ["Apple", "Banana", "Orange"]
print(fruits)
```

في المثال السابق تم انشاء Set تحتفظ بعدد من اسماء الفواكة.

ملاحظة: يجب أن نستخدم Set : لانه في حال لم نقم بذلك فسوف تعتبر Swift أنها Array بدلا من Set

المخرجات:

```
["Orange", "Apple", "Banana"]
```

لاحظ أنه عند طباعة القيم في Set، لم تتم طباعة العناصر على نفس الترتيب الذي أنشأت عليه المجموعة وفي كل مرة نقوم فيها بالطباعة سيختلف الترتيب مجدداً.

إنشاء Set تحتوي على عناصر مكررة

في حال إنشاء Set تحتوي على قيم مكررة فسيتم حذف تلك القيم تلقائياً، لاحظ معي المثال التالي:

```
let names: Set = ["Ahmed", "Ali", "Omar", "Ali"]
print(names)
```

المخرجات:

```
["Ahmed", "Omar", "Ali"]
```

لاحظ في نتيجة الطباعة أعلاه عدم تكرار طباعة القيمة Ali.

إنشاء Set لا تحتوي على عناصر

إذا أردنا إنشاء Set لا تحتوي على عناصر، فسنحتاج إلى تحديد نوع القيم التي سوف تخزن فيه العناصر متنوعة بالقوسين () كما هو موضح في المثال التالي:

```
var mySetA = Set<String>()
var mySetB = Set<Double>()
var mySetC = Set<Int>()
```

لاحظ تم تحديد نوع عناصر المجموعة بين <>

إضافة عنصر جديد إلى Set

لإضافة عنصر جديد في Set، يُكتب اسم المتغير متبوعاً بالدالة insert كما هو موضح في المثال التالي:

```
mySetA.insert("One")
mySetA.insert("Two")
mySetA.insert("Three")
print(mySetA)
```

المخرجات:

```
["Three", "Two", "One"]
```

اصبح المتغير mySetA الآن يحتوي على القيم One, Two, Three.
ملاحظة: يمكن اضافة عناصر الى مجموعة فقط إذا كانت من المتغيرات.

إزالة العناصر من Set

عند رغبتنا في إزالة عنصر أو أكثر من المجموعة فإن هناك أكثر من طريقة يُمكن استخدامها للقيام بذلك، ومنها:

- استخدام دالة remove لإزالة عنصر محدد من المجموعة.
 - استخدام دالة removeAll لإزالة جميع العناصر من داخل المجموعة
- يوضح المثال التالي كيفية استخدام هذه الدوال:

```
var numbers: Set = ["one", "two", "three"]
numbers.remove("two")
print(numbers)
numbers.removeAll()
print(numbers)
```

المخرجات:

```
["three", "one"]
```

```
[]
```

كما تلاحظ أعلاه، في السطر الثاني تم حذف العنصر ذو القيمة two من المتغير numbers، وفي السطر الرابع تم حذف جميع العناصر من المتغير numbers.

ملاحظة: يمكن إزالة عناصر من مجموعة فقط إذا كانت من المتغيرات.

التحقق من وجود عنصر في Set

للتحقق من وجود عنصر محدد في Set يمكنك استخدام دالة contains، كما هو موضح في المثال التالي:

```
var fruits: Set = ["Apple", "Banana", "Orange"]
var isAppleAvailable = fruits.contains("Apple")
print(isAppleAvailable)
```

المخرجات:

true

لأن المجموعة fruits تحتوي على العنصر Apple فستكون قيمة isAppleAvailable تساوي true.
حساب عدد عناصر Set
لمعرفة عدد عناصر المجموعة يمكننا استخدام الخاصية count كما هو موضح في المثال التالي:

```
var mySetB: Set = ["one", "two", "three"]  
print(mySetB.count)
```

المخرجات:

3

استخدام Set Operations

- مما يميز المجموعات انه يمكننا إجراء بعض العمليات عليها منها:
- دالة union تقوم بإيجاد جميع العناصر داخل المجموعتين من دون تكرار العناصر المشتركة.
 - دالة intersection تقوم بإيجاد العناصر المشتركة بين المجموعتين.
 - دالة subtracting تقوم بإيجاد جميع العناصر داخل المجموعة بشرط ان لا يكون العنصر موجود داخل المجموعة الأخرى.
 - دالة symmetricDifference تقوم بإيجاد جميع العناصر داخل المجموعتين باستثناء العناصر المشتركة.

لاحظ معي المثال التالي

```
let setA: Set = [1, 2, 4, 6, 7]  
let setB: Set = [2, 3, 5, 6, 8]  
print("the union is:", setA.union(setB))  
print("the intersection is:", setA.intersection(setB))  
print("the subtracting is:", setA.subtracting(setB))  
print("the symmetric difference is:", setA.symmetricDifference  
(setB))
```

المخرجات:

the union is: [5, 8, 7, 1, 4, 2, 3, 6]

the intersection is: [2, 6]

the subtracting is: [1, 7, 4]

the symmetric difference is: [1, 5, 7, 8, 3, 4]

استخدام Set membership

ايضا يمكنك إيجاد العلاقات بين المجموعات ومنها:

- دالة isSubset تقوم بالتأكد من أن المجموعة موجودة بكاملها داخل المجموعة الأخرى .
- دالة isSuperset تقوم بالتأكد من أن المجموعة تحوي المجموعة الأخرى بكاملها.
- دالة isDisjoint تقوم بالتأكد من أن المجموعة لا تحتوي على اي عنصر موجود داخل المجموعة الأخرى.

```
let setC: Set = [1, 2, 3, 4]
let setD: Set = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(setC.isSubset(of: setD))
print(setD.isSuperset(of: setC))
print(setC.isDisjoint(with: setD))
```

المخرجات:

```
true
true
false
```

تحويل Array إلى Set

قد نرغب في تحويل Array إلى Set لعدد من الأسباب؛ كالتخلص من العناصر المكررة الموجود في مصفوفة. ويتم ذلك عن طريق استخدام Set(), كما هو موضح بالمثل التالي :

```
let studentNames = ["Ahmed", "Ali", "Omar", "Ali", "Ahmed", "Khalid"]
let uniqueStudentNames = Set(studentNames)
print(studentNames)
```



```
print(uniqueStudentNames)
```

المخرجات:

```
["Ahmed", "Ali", "Omar", "Ali", "Ahmed", "Khalid"]  
["Omar", "Khalid", "Ahmed", "Ali"]
```

كما تلاحظ، فإنه عندما حولنا studentNames الى Set وأسندنا قيمتها إلى uniqueStudentNames فإننا بذلك تخلصنا من الأسماء المكررة

المراجع :

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman
- <https://www.programiz.com/swift-programming/sets>

مقدمة في الدوال Functions

الدالة عبارة عن مجموعة من الأوامر البرمجية مصممة لأداء مهمة معينة يتم تنفيذ الدالة عند استدعائها باسمها. تهدف الدوال إلى تقسيم الأوامر البرمجية الكبيرة إلى مجموعات صغيرة تشترك لتحقيق برنامج واحد.

قواعد تسمية الدالة

يمكنك اختيار أي اسم تقريباً عند تسمية الدوال، ولكن هناك بعض القواعد التي يجب عليك اتباعها:

- أن لا يحتوي الاسم على أي مسافة.
- يمكن استخدام - لكن لا يمكن استخدام الرموز الأخرى مثل:
- +، -، =، /، *، %، \$، @، !، ^، ... إلخ
- أن لا يبدأ الاسم برقم، لكن يمكن أن يحتوي على أرقام.
- أن لا يكون الاسم أحد الكلمات الرئيسية المستخدمة من قبل Swift مثل:

```
let, var, print, func, class .. إلخ
```

إنشاء الدالة

يمكننا أن نقوم بإنشاء دالة عن طريق كتابة func ثم اسم الدالة متبوعة بقوسين () ثم نقوم بكتابة جميع الأوامر البرمجية الخاصة بهذه الدالة داخل القوسين {} كما هو موضح في المثال التالي:

```
func helloWorld() {
    print("Hello world")
    print("Hola Mundo")
    print("مرحبا بالعالم")
}
```

كما تلاحظ قمنا بإنشاء دالة باسم `helloWorld` ووظيفتها طباعة ثلاثة جمل كما هو موضح أعلاه ولن يتم تنفيذ الدالة حتى يتم استدعائها، وهذا ما سنتناوله في الموضوع التالي.

استدعاء الدالة

بعد أن قمنا بإنشاء الدالة يمكن استدعاؤها في أكثر من مكان في البرنامج، ولن يتم تنفيذ الأوامر البرمجية التابعة لها إلا عندما تُستدعى فقط. نقوم باستدعاء الدالة بكتابة اسمها متبوعاً بالأقواس () ، كما هو موضح في المثال التالي:

```
helloWorld()
```

المخرجات:

```
Hello world
Hola Mundo
```

مرحبا بالعالم

لاحظ أنه تم تنفيذ الأوامر البرمجية التابعة للدالة `helloWorld` عندما تم استدعائها.

معاملات الدوال: Arguments و Parameters

مقدمة عن معاملات الدوال Argument & Parameter

يمكننا جعل الدوال تستقبل قيم كمداخلات تسمى `parameters`، يمكن تعريف `parameters` وتصورها على أنها ثوابت خاصة بالدالة، يحدد لها اسم ونوع البيانات أثناء تعريف الدالة، تُمرر القيم عند استدعاء الدالة وهذا ما يشار إليه بمفهوم

`.argument`.

استخدام parameters

نقوم بكتابة اسم المدخل `parameter` ونوع البيانات الذي تنتمي له قيمة المدخل بين قوسي الدالة () على النحو التالي:

```
func sayHelloTo(name: String) {
    print("Hello", name)
}
```

في المثال أعلاه قمنا بإنشاء دالة تستقبل مدخل واحد باسم `name` ويستقبل قيمة من نوع `String` والتي سيتم إسنادها له لاحقاً أثناء الاستدعاء. وبما أننا أضفنا للدالة هذا المدخل فيمكننا استخدامه داخلها وهذا ما قمنا بفعله في السطر الثاني.

ملاحظة: تُعامل المداخلات `parameters` كالثوابت، أي أنه لا يمكن تغيير قيمتها داخل الدالة.

استخدام arguments

هي القيم التي يتم تمريرها للمدخلات parameter عند استدعاء الدالة، يوضح المثال التالي طريقة تمرير القيم وقت استدعاء الدالة:

```
sayHelloTo(name: "Ahmed")
sayHelloTo(name: "Abdallah")
sayHelloTo(name: "Muhammad")
```

المخرجات:

```
Hello Ahmed
Hello Abdallah
Hello Muhammad
```

كما تلاحظ في المثال أعلاه قمنا باستدعاء الدالة ثلاث مرات، وفي كل مرة أسندنا قيمة مختلفة إلى المدخل name، إذاً يمكن الإشارة للمفهوم parameter على أنه الثابت الذي سيستقبل قيمة تُستخدم داخل الدالة والمفهوم argument القيم التي تُمرر عند استدعاء الدالة.

إنشاء الدالة ذات المدخلات المتعددة

يمكننا كتابة أكثر من parameter والفصل بينهم بفاصلة كما هو موضح في المثال التالي:

```
func sayWelcomeTo(name: String, title: String) {
print("Welcome " + title + name)
}
```

تحتوي الدالة أعلاه على مدخلين الأول name والآخر title وكلاهما من نوع String، أي أن الدالة ستستقبل قيمتين من نوع String، لاحظ أنه تم استخدام المدخلين داخل الدالة في عملية الطباعة.

استدعاء الدالة ذات المدخلات المتعددة

سنقوم باستدعاء الدالة السابقة ثم بين القوسين سنذكر اسم كل مدخل والقيمة المراد إسنادها له، ويكون ترتيب arguments مطابق لترتيب parameters الدالة ونفصل بين كل مدخل والآخر باستخدام الفاصلة كما هو موضح في المثال التالي:

```
sayWelcomeTo(name: "Ahmed", title: "Mr.")
sayWelcomeTo(name: "Abdallah", title: "Eng.")
sayWelcomeTo(name: "Muhammad", title: "Dr.")
```

المخرجات:

```
Welcome Mr.Ahmed
Welcome Eng.Abdallah
Welcome Dr.Muhammad
```

مفهوم واستخدامات return

مقدمة عن مفهوم return

في جميع الأمثلة السابقة كنا نقوم بكتابة أوامر طباعة في جميع الدوال، لتقوم بطباعة المخرجات لنا في لوحة التحكم، ولكن في بعض الحالات قد لا يكون هذا مانريد من الدالة القيام به، فقد نريد منها أن تعود لنا بقيمة عند استدعائها.

إنشاء الدالة مع return

لإنشاء دالة تعود بقيمة يجب علينا أولاً أن نحدد نوع البيانات المراد إعادتها كما هو موضح في المثال التالي:

```
func greeting() -> String {  
    return "Hello Ahmed,"  
}
```

لاحظ تمت كتابة نوع القيمة المراد إعادتها عند استدعاء الدالة بعد القوسين () وقبل القوس { على النحو التالي String →. تتم كتابة كلمة return داخل الدالة متبوعةً بالقيمة المراد أن تعود بها الدالة، وفي المثال أعلاه تعود الدالة بنص ترحيبي. الآن دعنا نقوم بطباعة قيمة الدالة باستدعائها داخل أمر طباعة على النحو التالي:

```
print(greeting() , "my name is Abdallah")
```

المخرجات:

```
Hello Ahmed, my name is Abdallah
```

لاحظ أن المخرج Hello Ahmed يمثل قيمة الدالة greeting .

تنبيه: عند استخدام علامة السهم التي تشير إلى نوع البيانات المرجو إعادتها، فإنه يجب أن تعود الدالة بقيمة من نفس النوع.

المراجع :

- <https://swift.org>
- Mastering Swift 5 - Fifth Edition by Jon Hoffman

Function Type

لو سئلت عن نوع القيمة 10 , ستجيب مباشرة ب Int , ولكن لو سئلت عن نوع الدالة التالية :

```
func addTwoInts(_ a: Int, _ b: Int) -> Int {  
    return a + b  
}
```

قد يبدو الأمر غريباً، ولكن في سويفت نعامل الدوال كأى قيمة أخرى وبالتالي لكل دالة نوع بياني. يتكون نوع الدالة من أنواع parameters ونوع القيمة المرجعة.

يمكنك استخدام هذا النوع مثل أي نوع آخر في Swift ، وبالتالي يمكنك تمرير functions إلى functions أخرى ، وإرجاع function من function . ويمكن أيضًا كتابة function داخل function . أي كما تعامل أي قيمة من أي نوع بيانات (تذكر كيف كنت تعامل القيمة 10 في البرامج!).

مثال:

نوع الدالة addTwoInts السابقة:

```
(Int, Int) -> Int
```

ولكن ما الذي يقدمه لنا التعامل مع الدوال بهذه الطريقة ؟ لاحظ الأمثلة التالية.

- استخدام نوع الدالة

بما أن الدوال في سويقت يتم معاملتها كأبي قيمة أخرى إذا يمكن إسنادها إلى متغير من نوع مطابق يمكنك تعريف متغير (أو ثابت) وإسناد الدالة addTwoInts إليه .

```
var mathFunction: (Int, Int) -> Int = addTwoInts
```

الآن بإمكانك معاملة mathFunction كما كنت تعامل addTwoInts.

```
print(mathFunction(2,3))
```

النتيجة:

5

ماذا لو أردنا إسناد قيمة أخرى إلى mathFunction ؟
أجل يمكنك إسناد أي دالة من النوع:

```
(Int, Int) -> Int
```

مثلا يمكن إسناد الدالة التالية إلى mathFunction:

```
func multiplyTwoInts(_ a: Int, _ b: Int) -> Int {  
    return a * b  
}  
  
mathFunction = multiplyTwoInts  
print(mathFunction(2,3))
```

النتيجة:

6

- دالة تمثل parameter لدالة أخرى

افترض اننا نريد كتابة دالة `printMathResult` والتي تستقبل قيمتين من النوع `Int` وتقوم بطباعة نتيجة عملية رياضية على العددين , ولكن ماهي العملية الرياضية؟ الجواب أن العملية سيتم إرسالها كدالة إلى `printMathResult` سيكون نوع الدالة الي سيتم إرسالها إلى `printMathResult` في هذا المثال هو :

```
(Int,Int)->Int
```

بحيث يتم تمرير قيمتين من النوع `Int` إلى هذه الدالة , وإرجاع القيمة التي ستقوم `printMathResult` بطباعتها.

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    print("Result: \(mathFunction(a, b))")  
}
```

والآن ما رأيك أن نقوم بتمرير الدالة `addTwoInts` التي رأيناها مسبقاً إلى الدالة `printMathResult`.

```
func printMathResult(_ mathFunction: (Int, Int) -> Int, _ a: Int, _ b: Int) {  
    print("Result: \(mathFunction(a, b))")  
}  
  
printMathResult(addTwoInts, 3, 5)
```

النتيجة:

8

وبالتأكيد يمكنك إرسال الدالة `multiplyTwoInts` أو أي دالة من نوع بيانات مطابق .

- التعامل مع الدوال كقيمة مرجعة

لدينا الدالتان التاليتان :

```
func stepForward(_ input: Int) -> Int {  
    return input + 1  
}  
  
func stepBackward(_ input: Int) -> Int {  
    return input - 1  
}
```

ولدينا الدالة `chooseStepFunction` التي تستقبل قيمة من النوع `Bool` وتقوم بإرجاع أحد الدالتين السابقتين على النحو التالي:

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
  return backward ? stepBackward : stepForward  
}
```

إذا كانت قيمة `backward` هي `true` سيتم إرجاع الدالة `stepBackward` أما غير ذلك سيتم إرجاع `stepForward`. لاحظ نتيجة استدعاء الدالة `chooseStepFunction`:

```
let inReverse = chooseStepFunction(backward:true)  
print(inReverse(10))
```

بما أن القيم المرسلة إلى الدالة `chooseStepFunction` هي `true` فسيتم إرجاع الدالة `stepBackward` وإسنادها إلى `inReverse`, والآن بإمكانك استخدام `inReverse` كأبي دالة من النوع:

```
(Int)->Int
```

• الدوال المتداخلة

كل الدوال التي رأيناها حتى الآن كانت أمثلة على دوال عامة، تم تعريفها في النطاق العام. ولكن يمكنك أيضًا تعريف الدوال داخل دوال أخرى، هو ما يعرف باسم الدوال المتداخلة `Nested Functions`. يتم إخفاء الدالة الداخلية عن النطاق العام افتراضيًا، ويمكن استدعاؤها واستخدامها بواسطة الدالة الخارجية التي تحتويها. ويمكن للدالة الخارجية أيضًا إرجاع الدالة الداخلية للسماح باستخدامها في نطاق آخر. يمكننا التعديل على المثال السابق وكتابة `stepForward` و `stepBackward` كدوال داخلية في `chooseStepFunction`

```
func chooseStepFunction(backward: Bool) -> (Int) -> Int {  
  func stepForward(input: Int) -> Int { return input + 1 }  
  func stepBackward(input: Int) -> Int { return input - 1 }  
  return backward ? stepBackward : stepForward  
}
```

}

المراجع:

- the swift programming language swift 5.5

البرمجة الكائنية Object Oriented Programming ((OOP

عبارة عن أسلوب خاص في كتابة الأسطر البرمجية لتكون أكثر سهولة وقابلية لإعادة الاستخدام، حيث يتم تجميع مجموعة من المتغيرات والدوال داخل وحدات منفصلة وتُسمى هذه الوحدات بالكائن Object، وقد يحتوي البرنامج الواحد على أكثر من كائن حيث يتم ربطها مع بعضها البعض، ويمكن تطبيق هذا الأسلوب في كتابة البرامج لأغلب اللغات البرمجية الشائعة الآن.

الكائن Object

لكي تعرف ما هو الكائن Object في البرمجة عليك أن تعرف ماهو الكائن Object في الحياة الواقعية، الكائن وهو أي شيء يمكن تعريفه وقد تكون له صفات وأفعال، حيث أن كل شيء من حولنا يعتبر كائن Object، بمعنى أي شيء تراه أمامك الآن هو كائن، فالمقعد أو الكرسي الذي تجلس عليه الآن هو كائن Object والأشجار من حولنا هي Object وحتى الأشخاص هم Object بل حتى الأشياء التي لا تستطيع رؤيتها بالعين المجردة أو لمسها هي Object مثل الوقت والتاريخ... إلخ وبإمكاننا القول أن الأمثلة التالية تمثل Object:

- السيارة.
- الأشجار.
- القلم.
- الأشخاص.
- الباب.
- التاريخ.
- الحساب البنكي.

كل كائن Object يحتوي على ثلاث خصائص رئيسية:

- معرف Identity.
- صفات Properties.
- أفعال Behaviors.

أولاً: **المعرف Identity** حتى نُميّز كل كائن داخل البرنامج يجب أن يكون له اسم يدل عليه.

ثانيًا: الصفات Properties كل كائن Object في الحياة له صفات نستطيع أن نصفها به، مثلًا السيارة هي كائن Object ولكل سيارة عدة صفات خاصة بها من لون وعدد للأبواب وعدد للمقاعد وبلد الصنع... إلخ جميع هذه الصفات بالإمكان تمثيلها داخل البرنامج.

ثالثًا: الأفعال Behaviors وتمثل جميع الأفعال التي يمكن أن يقوم بها الكائن Object على سبيل المثال للسيارة عدة أفعال تقوم بها:

- التحرك للأمام.
- الرجوع للخلف.
- التوقف.

جميع ما ذكر يعتبر أفعال تقوم بها السيارة وبالإمكان تمثيلها داخل البرنامج وعند تحويل هذه الأفعال إلى أسطر برمجية تُسمى Method.

مفهوم Class

يعتبر Class وصفًا لتفاصيل الكائن Object وشرحًا للأفعال التي يستطيع القيام بها، حيث إنه بمجرد إنشاء Class في البرنامج يصبح بإمكانك إنشاء كائن منه أو عدة كائنات منفصلة وكلما تنوعت Classes في برنامج كلما تنوعت الكائنات Objects وحتى تتضح لك الصورة أكثر، تخيل بأن لديك مصنع لصنع السيارات وتحتاج لمخطط واحد فقط لكل أنواع السيارات التي ستنتجها، فمن هذا المخطط ستستطيع صنع العديد من السيارات المختلفة (العديد من الكائنات Objects) حيث أن المخطط هنا ليس السيارة والهدف منه مساعدتك في صنع السيارة المخطط في هذا المثال هو Class، ولكل Class ثلاث خصائص رئيسية نفس خصائص الكائن التي تعرفنا عليها من قبل:

- معرف Identity.
- صفات Properties.
- أفعال Behaviors.

تعريف Class

الشكل العام لتعريف Class في Swift:

```
class ClassName{
```

```
// property 1
```

```
// property 2
```

```
// methods N
```

```
}
```

والمثال التالي يعبر عن class:

```
class Car{  
var color: String = "White"
```

```
}
```

في المثال السابق تم تعريف Class وإعطائه الاسم Car وتم تعريف خاصية property للون السيارة وتسميتها Color. شروط تسمية Class

هي نفس شروط تسمية المتغيرات ولكن ينصح بشدة أن يكون الحرف الأول من الاسم حرف كبير Capital.

تعريف الكائن Object

الشكل العام لتعريف الكائن Object في Swift:

```
let ObjectName = ClassName()
```

ملاحظة: عند تعريف الكائن Object بالإمكان البدء بالكلمة المفتاحية let أو var

ولنأخذ مثالاً على طريقة تعريف Object

```
let car1 = Car()
```

```
let car2 = Car()
```

في المثال السابق تم تعريف كائن Object باسم car1 من Class اسمه Car ومن ثم تم تعريف كائن Object باسم car2 من Class اسمه Car.

بعد تعريف Object من Class سيحتوي Object على جميع خصائص وأفعال Class، ففي المثال أعلاه يحتوي Object car1 على صفة اللون color وبالإمكان الوصول إليها واستعراضها بهذا الشكل:

```
print(car1.color)
```

المُخرج:

```
white
```

وفي حال رغبت بتعديل هذه الصفة من Object من اللون white إلى اللون red بإمكانك القيام بالتالي:

```
car1.color = "red"
```

مقدمة في الخصائص Properties

الخصائص بشكل عام تمثل الثوابت constants أو المتغيرات variables المستخدمة في class ويطلق عليها عدة مسميات أخرى، مثل: الخصائص، الصفات، المتغيرات... وغيرها Attributes, properties, characteristics, state, field, variables ولكننا سنعتمد الإشارة إليها باسمها المعروف (الخصائص properties).

ولنكون أكثر تحديداً نستطيع القول أن الخصائص تعبر عن القيم المخزنة داخل class التي قد نستدعيها أو نعدل على قيمها في البرنامج لأي سبب كان، ونستطيع تصنيفها في نوعان أساسيان:

- الخصائص المخزنة Stored Properties.
- الخصائص المحسوبة Computed Properties.

تقوم الخصائص المخزنة **Stored Properties** بتخزين القيم وحفظها من البداية على عكس الخصائص المحسوبة **Computed Properties** والتي تقوم بحساب القيمة ومن ثم حفظها، وسنتحدث عن كل واحدة منها بالتفصيل.

الخصائص المخزنة **Stored Properties**

تقوم **بتخزين وحفظ** القيم بشكل مباشر سواء كانت متغيرات **variables** أو ثوابت **constants** ويتم تعريفها من داخل **class** على سبيل المثال:

```
class Number {  
    var number : Int  
    let pi = 3.1415  
}
```

في المثال أعلاه قمنا بتعريف **class** باسم **Number** وعرفنا بداخله **Stored Properties** أحدهما متغير **variable** **number** اسمه **stored property** والآخر ثابت **constant stored property** **pi** اسمناه **Int** من نوع **Int**.

الخصائص المحسوبة **Computed Properties**

لاتخزن القيم بعينها ولا تسند لها قيم في وقت تعريفها، وإنما توفر إمكانية حساب القيمة وإرجاعها، وبإمكانك استخدام بقية الصفات **properties** بشكل غير مباشر في الحساب.
مثال:

```
class Rectangle {  
    var width:Double = 5.5  
    var height:Double = 6.6  
    var area:Double {  
        width * height  
    }  
}
```

المثال أعلاه يحتوي على متغير **variable stored property** من نوع **Double** اسمه **width** ليمثل العرض وله قيمة ابتدائية 5.5 ومتغير آخر **variable stored property** من نوع **Double** اسمه **height** ليمثل الطول وله قيمة ابتدائية 6.6، ولدينا أخيراً متغير ليحسب المساحة **area** من نوع **Double** وداخل الأقواس عملية حسابية نضرب الطول في العرض لنحسب المساحة، وبهذا الشكل نكتب **variable computed Properties**.
لو أردنا استدعاه سيكون بالشكل التالي:

```
var reocrangle = Rectangle()
```

```
print(reocrangle.area)
```

قمنا بإنشاء **object** من نوع **Rectangle** وطبعنا المساحة عن طريق استخدام **reocrangle.area**، بهذا الشكل سنقوم باستدعاء **area** لتحسب المساحة وبما أن **area** تعتبر **variable computed Properties** لن تتم عملية الحساب فيها إلا إذا قمنا بدائها، في اللحظة التي نناديها ستقوم بعملية الحساب وترجع لنا القيمة، فكما نلاحظ لم نسند لها قيمة أساسية أو ابتدائية، إنما استخدمناها لتحسب لنا قيمة محددة باستخدام عملية قامت بتطبيقها على الطول والعرض بشكل غير مباشر.

المخرجات:

36.3

سنقوم بطباعة النتيجة وحساب المساحة استناداً على الطول والعرض.

هل الخصائص المحسوبة تطبق فقط على العمليات الحسابية والأرقام؟ الخصائص المحسوبة Computed Properties لا تطبق فقط على العمليات الحسابية، وإنما حتى على الكلمات فعلى سبيل المثال لديك البرنامج التالي:

```
class Person {  
    var firstName:String = "Amani"  
    var lastName:String = "Ahmed"  
    var fullName:String {  
        return firstName + " " + lastName  
    }  
}
```

يحتوي البرنامج أعلاه على class اسمناه Person وأنشأنا ثلاث متغيرات من نوع String تمثل الاسم الأول والأخير وقمنا بإسناد قيم لها، Amani و Ahmed والمتغير الأخير fullName سنلاحظ بداخله جملة تبدأ بكلمة return لترجع لنا القيمة المحسوبة والتي ستكون الاسم الأول والاسم الأخير، بإمكانك استخدام return أو بدونها فهي مجرد اختلاف في طريقة الكتابة. لنقوم بطباعتها سننشئ object من Person ، ونستدعي fullName وذلك بالشكل التالي:

```
var person = Person()  
person.fullName
```

المخرجات:

Amani Ahmed

قمنا بطباعة Amani Ahmed .

تخصيص الخصائص المحسوبة باستخدام Getters & Setters

سبق وعرفنا أن الخصائص المحسوبة، تقوم بحساب قيمة معينة وترجعها لنا وإمكانك إضافة setter مخصصة و getter مخصصة كذلك.

- ما هي getter؟
- بشكل بسيط تعمل على إرجاع القيمة للخصائص المحسوبة كما ذكر في الأمثلة أعلاه، باستخدام الكلمة المفتاحية get ويتم تنفيذ مابداخلها عند مناداتها وهي الخاصية الأساسية في computed properties.
- ما هي setter؟
- يمكنك استخدامها لتغيير القيم وإسنادها فقط باستخدام الكلمة المفتاحية set وتعتبر خطوة اختيارية في computed properties.

هيكل كتابة الخصائص المحسوبة باستخدام Getters & Setters

- تكتب بالشكل التالي:

```
var property : type {  
    get {  
        //code
```

```

}
set(value) {
// code
}
}

```

مقدمة في Instance Methods

تعتبر أنها الأفعال Behaviors التي يستطيع أن يقوم بها class وعند كتابة هذه الأفعال داخل البرنامج تُسمى Method. Method ولكي نقوم بتحويل الأفعال الخاصة بالسيارة إلى Class عندما تكون داخل Function عبارة عن Method نقوم بالتالي Class Car داخل:

```

class Car{
var color = "White"
func moveForward(){
print("move forward")
}
}

```

في المثال السابق قمنا بتحويل فعل من أفعال السيارة وهو المشي للأمام إلى Method، ومن ثم تم إضافة جملة طباعة للتعبير عن حركة السيارة للأمام ولكن بإمكانك كتابة أي أمر برمجي بين أقواس Method حتى يتم تنفيذها عند استدعاء Method. ملاحظة: بالإمكان أن يحتوي Class الواحد على أكثر من Method في نفس الوقت.

طريقة استدعاء Method: ليتم استدعاء Method نحتاج بأن نقوم بتعريف Object من class بهذا الشكل:

```

class Car{
var color = "White"
func moveForward(){
print("move forward")
}
}

let MyCar = Car()
MyCar.moveForward()

```

في المثال السابق تم تعريف Object من Class Car ومن ثم تم استدعاء moveForward وهي عبارة عن Method

ما هي Initializer function؟

قد تتساءل لماذا علينا أن نقوم بكتابة الأقواس () بعد تعريف كل Object!

```
let car1 = Car()
```

ويعود السبب لذلك أنه عند تعريف كل Object يتم استدعاء function موجودة في أي class يتم إنشائه، و تُسمى Initializer function هذه function موجودة ولكن مخفية لن تراها مباشرة داخل Class وفي حال لم ترغب بتنفيذ شيء ما عند تعريف Object لن تحتاج الوصول لها والتعديل عليها، ولكن إذا رغبت بتنفيذ شيء ما عند إنشاء Object ستحتاج كتابتها، والصيغة العامة لها هي:

```
init(parameters) {  
  //statements  
}
```

في الصيغة العامة لطريقة كتابتها سنتوقف عند ثلاث نقاط:

النقطة الأولى: لاحظ لا توجد الكلمة المفتاحية الخاصة بتعريف الدوال func في بداية تعريف function حيث لا داع لكتابتها في البداية.

النقطة الثانية: وجود الكلمة المفتاحية init حيث أنها كلمة ثابتة يتم كتابتها في كل مره ترغب بكتابة Initializer function.

النقطة الثالثة: بالإمكان كتابة مُعاملات Parameters لهذه Method مثل أي دالة تحتوي على معاملات Parameters.

وبالإضافة إلى أنه داخل أقواس Method بإمكانك كتابة أي أوامر ترغب بأن يتم تنفيذها عند إنشاء Object جديد.

لنأخذ مثال على initializer function:

```
init() {  
  print("new object has been created!")  
}
```

وللتوضيح أكثر لنرى المثال التالي:

```
class Celsius {  
  var temperature: Double  
  init() {  
    temperature = 25.0  
  }  
}  
  
var c = Celsius()  
print("The default temperature is \(c.temperature)° celsius")
```

في المثال السابق لاحظ أنه تم تعريف Class باسم Celsius ومن ثم تم تعريف Property باسم temperature من نوع Double و تم كتابة Initializer function لاحتوي على Parameters وتم إسناد قيمة افتراضية لدرجة الحرارة temperature وسيتم تنفيذ هذا السطر عند إنشاء Object جديد وتم إنشاء Object جديد باسم c.

المُخرج

The default temperature is 25.0° celsius

المقصود بالكلمة المفتاحية **self**

حتى تعرف المقصود بالكلمة المفتاحية **self** لنقم بكتابة الأسطر البرمجية التالية:

```
class Car {  
    var color = ""  
    init(color: String) {  
        color = color  
    }  
}
```

عند كتابة الأسطر البرمجية أعلاه في Xcode سيظهر لك خطأ! وذلك بسبب أن Xcode لم يستطيع أن يفصل بين **color** التي تخص **Class** والموجودة في السطر الثاني و **color** التي تخص الدالة والموجودة في السطر الرابع كمعال **Parameter**، وحتى ندل Xcode للوصول للخاصية **color** سنقوم بكتابة الكلمة المفتاحية **self** بهذا الشكل:

```
class Car {  
    var color = ""  
    init(color: String) {  
        self.color = color  
    }  
}
```

لاحظ في المثال السابق تم إضافة الكلمة المفتاحية **self** مباشرة أمام الخاصية **color** حيث أن **self** تمثل اسم **Class** التي تتم كتابتها فيه، وفي هذا المثال **self** تعني **Class Car** بالتالي عند كتابتها نعني الخاصية الموجودة داخل هذا **Class**.

مقدمة في Inheritance

الوراثة تعني inheritance وفي البرمجة تعني أن ترث صفات وأفعال من class محدد. مفهوم الوراثة يسهل عليك إنشاء class جديد بإعادة استخدام صفات وأفعال موجودة مسبقاً في class آخر.

مفهوم Subclass و Superclass

- إذا كان لديك class يرث من class آخر يطلق عليه اسم Subclass
- أما class الذي يرث محتوياته إلى class آخر يسمى Superclass ويسمى أيضاً (base class)

طريقة كتابتها:

عند كتابة class الذي تريد أن يرث الصفات من class آخر، عليك أن تكتب : النقطتين الرأسيتين بعد اسمه، ويتبعها class الذي تريد أن ترث منه، لنأخذ نظرة على المثال التالي:

```
class Person {  
    var name = "Ahmed"  
}  
  
class Student : Person {  
  
}
```

في الأسطر البرمجية أعلاه أنشأنا class اسمناه Person ويحتوي على متغير به الاسم وأعطيناه قيمة مبدئية Ahmed ، وبعدها قمنا بإنشاء class جديد اسمه Student يرث صفات Person كما هو موضح بهذا الشكل نستطيع أن نقول أن class Student يرث صفات 'class Person'.

مقدمة في Overriding

فلنفترض أن لديك class يحتوي على function ويقوم بتوريثها إلى class آخر، وتود أن تكتب تعريف ومحتوى مختلف لها بداخل class كأن تكون على سبيل المثال تطبع جملة مختلفة ولكن بنفس اسم function الموجودة بداخل super class الذي ترث منه، في هذه الحالة ستستخدم مفهوم Overriding، والذي يسمح لك بكتابة function ولكن بمحتوى مختلف في Sub class، ولنفهم الفكرة بشكل أقرب، لاحظ المثال التالي:

```
class Vehicle {  
    var speed: Int = 100  
    var color: String = "white"  
    func makeNoise() {  
        print("play a vehicle sound!")  
    }  
}
```



```
class Train: Vehicle {
    override func makeNoise() {
        print("play a train sound 🚂")
    }
}
```

```
var myTrain = Train()
myTrain.makeNoise()
```

في المثال السابق لدينا class اسمه Vehicle يحتوي على خصائص و method اسمها makeNoise والتي تقوم بطباعة الجملة التالية:

```
play a vehicle sound!
```

ويحتوي أيضًا على class آخر باسم Train يرث من class Vehicle داخله تم تعريف method بنفس الاسم ولكن تقوم بطباعة مُخرج آخر وهو:

```
play a train sound 🚂
```

حتى نتمكن من تعريف method بنفس الاسم الموجود في super class تم كتابة الكلمة المفتاحية **override** وبهذا الشكل استطعنا كتابة method بنفس الاسم في super class و sub class ولكن بإختلاف المحتوى. عند تنفيذ myTrain.makeNoise() سيكون المخرج بهذا الشكل:

```
play a train sound 🚂
```

وذلك لأنه سيقوم بتنفيذ جملة الطباعة الموجودة في الدالة makeNoise التي بداخل class Train.

كيف تمنع Overriding Methods؟

بإمكانك منع override عن طريق استخدام الكلمة المفتاحية **final** أمام تعريف method داخل class بهذا الشكل:

```
class Vehicle{
    var speed: Int = 100
    var color: String = "white"
    final func makeNoise() {
        print("play a vehicle sound!")
    }
}

class Train: Vehicle {
}
```

لاحظ وجود الكلمة المفتاحية `final` أمام تعريف `method makeNoise` وبهذا الشكل سيتم منع أي `class` يرث منه بأن يقوم بعمل `override` (تكرار الاسم) لنفس الدالة.

مفهوم التحكم في الوصول Access Control

في مرحلة ما أثناء كتابتك لبرنامج ما قد تحتاج لتحديد من يستطيع الوصول إلى `class` محدد، أو أن تضع قيودًا محددة على متغيرات `variables` في برنامجك لمنع تغييرها من خارج `class`، هذا المفهوم يسمى `Access Control`، ويصنف في أربع درجات من الوصول:

الأولى: `private`

- عند تعريفك متغير في `class` باستخدام الكلمة المفتاحية `private` سيكون المتغير قابلاً للوصول والتعديل فقط في داخل `class`.

الثانية: `fileprivate`

- عند تعريفك متغير في ملف باستخدام الكلمة المفتاحية `fileprivate` سيكون المتغير قابلاً للوصول والتعديل من أي مكان في داخل نفس الملف.

الثالثة: `internal`

- عند تعريفك متغير في ملف باستخدام الكلمة المفتاحية `internal` سيكون المتغير قابلاً للوصول والتعديل من داخل تطبيقك بشكل طبيعي، تعتبر الحالة الأساسية لكل متغير و سطر يكتب داخل برنامجك بشكل تلقائي حتى وإن لم تكتب الكلمة `internal`.

الرابعة: `public`

- عند تعريفك متغير بداخل `class` ما باستخدام الكلمة المفتاحية `public` سيكون المتغير قابلاً للوصول من خارج `class` الخاص به، على سبيل المثال لو كنت في `class` آخر ستستطيع الوصول له. وهناك أيضًا درجتان من الوصول إضافة إلى الأنواع السابقة وهما:

الأولى : `open`

- عند تعريفك `class` باستخدام الكلمة المفتاحية `open` سيكون بإمكانك الوصول له من أي مكان خارج برنامجك، وكذلك تستطيع تطبيق `subclassing` و `overridden` عليه.

الثانية : `final`

- عند تعريفك `class` باستخدام الكلمة المفتاحية `final` سيتمنعك من تطبيق `subclassing` و `overridden` عليه.

مفهوم Optionals

من الخصائص التي تمتاز بها لغة Swift هي Optional، ولكي تعرف ما هو Optional ستحتاج قبلها أن تعرف ما هو nil. Optional هو الشيء الذي لاوجود له وهو المتغير الذي لا توجد له قيمة، وبالتالي فإن nil متغير var أو ثابت let قد يحتوي أو لا يحتوي على قيمة أي أنه nil، إذا عند تعريف Optional قد تكون له أي قيمة من أي نوع من Data type أو قد لا تكون له قيمة مطلقاً.

طريقة تعريف Optional

الصيغة العامة لتعريف Optional هي:

```
var VariableName : Data_Type?
```

لاحظ وجود علامة الاستفهام ؟ والتي ستحتاج أن تكتبها بعد نوع البيانات مباشرة للدلالة بأن هذا المتغير أصبح Optional. مثال على تعريف متغير Optional:

```
var age: Int?
```

في المثال أعلاه تم تعريف Optional Int ولم يتم إسناد قيمة له.

```
var username: String? = "user1"
```

وفي هذا المثال تم تعريف Optional String ولكن تم إسناد قيمة له، ومن هذا المثال نستنتج بأنه عند تعريف Optional بالإمكان تحديد قيمة له أو تركه بدون إسناد قيمة.

التعامل مع Optional

قد يخطر على ذهنك عند التعامل مع Optional مثل طباعته على سبيل المثال بأنك ستكتب اسمه فقط بهذا الشكل:

```
let age : Int? = 22
print(age)
```

ستلاحظ بأن المخرج من هذا البرنامج هو:

```
Optional(22)
```

ولم تظهر القيمة التي نرغب بطباعتها وحدها، ولحل هذه المشكلة وطباعة القيمة فقط لديك ثلاث خيارات للتعامل مع Optional. الخيار الأول: **force unwrap**:

وهو عبارة عن إضافة علامة التعجب بعد Optional في جملة الطباعة بهذا الشكل:

```
let age : Int? = 22
```

```
print(age!)
```

هذا الرمز يعني بأنك كمبرمج تضمن ومتأكد من وجود قيمة في Optional بشكل لا يدع للشك، بحيث أنه في حال تم تنفيذ البرنامج وكان Optional بدون قيمة أي nil سيفشل تشغيل البرنامج ويظهر لك خطأ، ولا يحذر الكثير من المبرمجين استخدام **force unwrapping**.

الخيار الثاني: nil coalescing

في هذا الخيار سيتم عرض قيمة افتراضية وفي حال لم تكن هناك قيمة أي أنه nil. مثال:

```
let age : Int? = 22
```

```
print(age ?? 0)
```

في المثال أعلاه لاحظ وجود الرمز ؟؟ في السطر الثاني ثم القيمة صفر، هذا الرمز يعني في حال كان age يساوي nil سيتم عرض القيمة 0

- ملاحظة: لا يجب استخدام القيمة صفر في كل مره تكتب الكود وبإمكانك إسناد أي قيمة أخرى، مثال 18 بهذا الشكل:

```
swift let age : Int? = 22``
```

```
print(age ?? 18)
```

****الخيار الثالث: Optional binding****

في هذا الخيار يتم كتابة جملة شرطية ويتم التحقق من وجود قيمة في Optional بحيث إذا كانت توجد قيمة سيتم تمريرها إلى متغير جديد يمكنك استخدامه والتأكد من وجود قيمه فيه بهذا الشكل:

```
let age : Int? = 22
if let temp = age {
print(temp)
} else {
print("age is nil")
}
```

شرح المثال:

السطر الثالث: في هذا السطر يتم التحقق من وجود قيمة في Optional age وفي حال تم وجود قيمة وكان الشرط true سيتم نسخ القيمة إلى المتغير المؤقت temp. السطر الرابع: بإمكانك في هذا السطر الاستفادة من القيمة temp بعد أن تأكدت من وجود قيمة فيها مثل أن تقوم بطباعتها مثل ما تم فعله في هذا السطر. السطر السادس: سيتم تنفيذ جملة else إذا كان Optional يساوي nil.

مفهوم Optional Chaining

يُعتبر Optional Chaining بمثابة السلسلة التي إذا انقطع جزء منها لن تستطيع للوصول إلى البقية منها. مثال:

```
class Rectangle {
var rectangleHeight : Height?
}
class Height {
var height = 5
}
```

تم تعريف class واسمه Rectangle، بداخله متغير من نوع Height وتم تعريفه على شكل optional ولدينا class آخر تم تعريفه باسم Height وبداخله متغير اسمه height وأسندنا له قيمة مبدئية 5. بمعنى أنه لدينا قيمة متأكدين من وجودها

وهي `height = 5` ولدينا قيمة أخرى ولكن من الممكن أن تكون `nil` وهي `rectangleHeight : Height`? لنحاول الوصول إلى القيمة `height` والتي تأكدنا من وجود قيمة لها، مثال:

```
var myRectangle = Rectangle()
```

```
print(myRectangle.rectangleHeight?.height)
```

قمنا بإنشاء `object` من نوع `Rectangle` وقمنا بالوصول إلى `height` عن طريق السطر الثاني وحاولنا طباعته، سيكون الآن المخرج:

```
nil
```

ولعلك تتعجب كيف لها أن تكون `nil` على الرغم من تأكدنا من وجود قيمة `height`! وسأخبرك السبب بأن `rectangleHeight` تعتبر `nil` لأننا لم ننشئ `object` منها، ففي `Optional Chaining` إذا وصلت إلى قيمة غير موجودة لن نستطيع الوصول إلى بقية البرنامج والمتغيرات التي تعتمد عليه. فلنجرب أن نعرف `rectangleHeight` ونرى ماذا سيحدث:

```
var myRectangle = Rectangle()
myRectangle.rectangleHeight = Height()
print(myRectangle.rectangleHeight?.height)
```

قمنا بإنشاء `Object` لكي لا تكون `myRectangle.rectangleHeight` فارغة. عند تشغيل البرنامج ستكون المخرجات كالتالي:

```
Optional(5)
```

ولأن `rectangleHeight` الآن تم تعريفها وليست `nil` ظهرت لنا 5، ولكنها ظهرت على شكل `optional` على الرغم من أنها ليست `optional` وهذا كذلك بسبب `optional chaining` بشكل تلقائي أصبحت `optional` في هذه السلسلة لأنها قد لا تكون موجودة إذا لم تكن `rectangleHeight` موجودة. وهذا بشكل مبسط مفهوم `optional Chaining` كالسلسلة، إن توقفت في المنتصف لعدم وجود قيم، لن يكمل البرنامج بقية السلسلة المكتوبة.

مقدمة في Enumeration

تعرف `Enumeration` والتي تختصر بالاسم `enum` بكونها أحد الأنواع الشائعة لمجموعة مترابطة من القيم والتي تسمح لك بالعمل معها بطريقة آمنة في برنامجك ولتوضيح الفكرة بشكل أكبر لنفكر بها وكأنها نوع من أنواع المتغيرات المفيدة جدًا في حال تم استخدامها مع `switch` و `conditions`.

طريقة تعريفها

لنوضح الفكرة سننشئ `Enumeration` للاتجاهات:

```
enum Direction {
    case forward
```

```

    case back
    case left
    case right
}

```

في المثال أعلاه قمنا بتعريف Enumeration باسم Direction. لاحظ أنه تم استخدام الكلمة المفتاحية enum عند تعريف Enumeration. ولإنشاء قيم بداخله نستخدم الكلمة case ومن ثم القيمة.

الوصول إلى قيم Enumeration

للوصول إلى أحد قيم enumeration نستطيع ذلك بالشكل التالي:

```

var movingDirection = Direction.forward
print(movingDirection)

```

المخرجات

```
forward
```

في المثال أعلاه قمنا بإسناد القيمة forward إلى المتغير movingDirection وعند طباعتها ستكون النتيجة forward.

استخدامها مع Switch

لاستخدام enumeration مع switch لاحظ المثال التالي:

```

switch movingDirection {
    case .forward:
        print("you moved forward")
    case .back:
        print("you moved backwards")
    case .left:
        print("you moved to the left")
    case .right:
        print("you moved to the right")
}

```

المخرجات

```
you moved forward
```

لاحظ في المثال السابق اكتفينا بكتابة نقطة ومن ثم الحالة المحتملة لقيمة movingDirection.

إسناد قيم إلى حالات Enumeration

بإمكاننا إسناد قيمة لكل حالة لدينا في Enumeration باستخدام طريقتين Raw Value Enumerations أو

Associated Value Enumerations

إسناد قيم باستخدام Raw Value Enumerations

لإسناد قيم إلى حالات enumerations يجب أن تكون جميع القيم المسندة من نفس النوع ومحددة مسبقاً وثابتة، لذلك تحتاج أولاً لتحديد نوع القيم المضافة ومن ثم إعطاء كل حالة قيمة، ولتوضيح الفكرة لاحظ المثال التالي:

```
enum Direction : String {  
    case forward = "Move forward"  
    case back = "Move back"  
    case left = "Move left"  
    case right = "Move right"  
}  
  
var m = Direction.back.rawValue
```

في المثال السابق قمنا بتحديد النوع ليكون String وأعطينا لكل حالة قيمة باستخدام علامة المساواة لاحظ أيضاً أننا استخدمنا الخاصية `rawValue` للوصول إلى قيمة الحالة `.back`.

إسناد قيم باستخدام Associated Value Enumerations

نستطيع استخدام Associated Value في حال أردنا إضافة قيم غير ثابتة لحالات enumeration، ولتوضيح الفكرة لاحظ معي المثال التالي:

```
enum WorkingDay {  
    case sunday(hours: Int)  
    case monday  
    case tuesday  
    case wednesday  
    case thursday  
}  
  
var workingDay = WorkingDay.sunday(hours: 8)
```

لاحظ في المثال السابق أن الحالة `sunday` عبارة عن Associated Value من نوع `Int` لذلك قمنا بإسناد القيمة 8

مقدمة في Tuples

المجموعات أو كما تسمى Tuples

نستخدم Tuple عندما نريد إنشاء مجموعة من عدة قيم مختلفة ولتوضيح الفكرة لاحظ معي المثال التالي:

```
let tuple1 = (1, ["One"]) // inferred as (Int, [String])
```

```
let tuple2 = (3, true, "Two") // inferred as (Int, Bool, String)
let tuple3 = ("a", tuple1) // inferred as (String, (Int, [String]))
```

لاحظ في المثال السابق استخدمنا الأقواس عند تعريف Tuple وبادخلها وضعنا القيم. أيضا لاحظ في السطر الثالث أننا قمنا بوضع tuple1 كقيمة داخل tuple3.

```
let tuple3 = ("a", tuple1) // inferred as (String, (Int, [String]))
```

الوصول إلى محتويات Tuple

تستطيع الوصول إلى محتوياته وطباعة ما بداخله بعدد من الطرق:

- الوصول إلى عناصر tuple باستخدام رقم الخانة للقيمة index
- يبدأ ترقيم الخانات في tuple ابتداء من صفر إلى نهاية tuple ولتوضيح الفكرة لاحظ معي المثال التالي:

```
let month = (9, "Ramadan")
print("Order in a year: \(month.0)")
// Prints "Order in a year: 9"
print("Month full name: \(month.1)")
// Prints "Month full name: Ramadan"
```

المخرجات

```
Order in a year: 9
Month full name: Ramadan
```

في المثال السابق السطر الثاني والرابع قمنا بالوصول إلى قيم tuple وذلك باستخدام اسم tuple وهو month متبوع برقم الخانة.

- الوصول إلى عناصر tuple باستخدام Decompose the values
- بإمكاننا إسناد قيم tuple إلى متغيرات أو ثوابت كما في المثال التالي:

```
let month = (9, "Ramadan")
let (order, fullName) = month
//alternatively, you can combine the above:...
print("Order is \(order)")
//Prints "Order is 9"
print("Full name is \(fullName)")
//Prints "Full name is Ramadan"
```

المخرجات

```
Order is 9
```



```
Full name is Ramadan
```

كما تلاحظ في المخرجات أن قيمة الثابت `order` هي 9 وذلك لأن القيمة 9 يحملان نفس رقم الخانة.

- الوصول إلى عناصر معينة باستخدام `Decompose the values`
- عندما نريد تجاهل بعض القيم في `tuple` نقوم بوضع `_` مكان رقم خانة القيمة المراد تجاهلها ولتوضيح الفكرة لاحظ معي المثال التالي:

```
let month = (9, "Ramadan")
let (_, justTheFullName) = month
print("Full name is \(justTheFullName)")
```

المخرجات

```
Full name is Ramadan
```

في المثال السابق قمنا بتجاهل قيمة العنصر الأول وذلك بوضع `_`.

- الوصول إلى عناصر `tuple` باستخدام اسم العنصر
- نستطيع فعل ذلك بإعطاء أسماء للعناصر داخل `tuple` لنصل إليها بسهولة، مثال:

```
let month = (order: 9, fullName: "Ramadan")
print("Order is \(month.order)")
//Prints "Order is 9"
print("Full name is \(month.fullName)")
//Prints "Full name is Ramadan"
```

في المثال أعلاه عند تعريف `Tuple` قمنا بإعطاء أسماء للعناصر بداخلها، وبهذا الشكل نستطيع الوصول إليها بسهولة.

ما هي فائدة استخدام `Tuple`؟

تفيدنا في حال كنّا نستخدم `Function` نقوم بإرجاع قيمة واحدة، ونريد أن نجعلها تقوم بإرجاع قيمتين مختلفتين بالشكل التالي:

```
func getMonth() -> (number : Int, month : String) {
    return ( 3 , "March")
}

let thisMonth = getMonth().month
print(thisMonth)
```

في المثال أعلاه، قمنا بتعريف `getMonth` والتي تقوم بإرجاع قيمتين `Int` و `String` عن طريق استخدام `tuple`.

نظرة على Struct

في بعض الحالات أثناء تطويرك لتطبيق ما ستحتاج لمتغير خاص فيك، ومتغير نوعه ليس String وليس Int وليس Double، متغير من الممكن أن يجمع كل هذه البيانات أو جزء منها، كأن تقوم بتطوير تطبيق تحفظ فيه المعلومات الشخصية للمستخدم مثل الاسم ورقم الجوال والعمر، وقد تقوم بإنشاء ثلاث متغيرات منفصلة لكل صفة بهذا الشكل:

```
let name: String
var phoneNumber: String
var age: Int
var addresses: [String]
```

ولكن بالإمكان أن نقوم بتجميع هذه المعلومات في متغير واحد فقط عن طريق إضافتها جميعًا داخل مجموعة واحدة داخل struct. الصيغة العامة لتعريف Struct:

```
struct StructName { }
```

يبدأ تعريف Struct بكتابة الكلمة المفتاحية struct ثم يتم تسميته بحيث أن شروط تسمية Struct هي نفس شروط تسمية Class، ثم يتم فتح وإغلاق الأقواس بهذا الشكل { } وداخلها بالإمكان إضافة Properties و Methods و Initializer. مثال على تعريف struct:

```
struct UserInfo {
    let name: String
    var phoneNumber: String
    var age: Int
    var addresses: [String]
}
```

نلاحظ في المثال السابق أنه في:

السطر الأول: تم تعريف struct وتسميته UserInfo، لاحظ أن أول حرف هو حرف كبير capital.

السطر الثاني: تم تعريف ثابت لاسم المستخدم من نوع String.

السطر الثالث: تم تعريف ثابت لرقم جوال المستخدم من نوع String.

السطر الرابع: تم تعريف متغير لعمر المستخدم من نوع Int.

السطر الرابع: تم تعريف مصفوفة لعناوين المستخدم من نوع String.

ملاحظة: عند تعريف struct بالإمكان إسناد عدة قيم ابتدائية له:

```
struct UserInfo {
    let name: String = "Amani"
    var phoneNumber: String = "000"
    let age: Int = 22
    var addresses: [String] = ["Abha" , "Dammam"]
}
```

إنشاء Instance من Struct

الصيغة العامة لإنشاء Instance من Struct، حتى نستفيد من struct بعد تعريفه بالإمكان أن ننشئ منه Instance:

```
var instance = StructName
```

عند إنشاء Instance من Struct علينا أن نكتب الكلمة المفتاحية **var** أو **let** ثم نقوم بتسميته، حيث أن شروط التسمية هي نفس شروط تسمية المتغيرات، بعد ذلك نحتاج أن نقوم بكتابة علامة = ثم نكتب اسم struct الذي نرغب بإنشاء Instance منه

```
var user1 = UserInfo(name: "Amani", phoneNumber: "000", age: 22, addresses: ["Abha" , "Dammam"])
```

نلاحظ في المثال السابق:

تم إنشاء Instance من struct UserInfo وتسميته user1 ثم تمت إضافة معلومات المستخدم له.

طريقة الوصول إلى محتويات Struct:

بالإمكان الوصول إلى Properties و Methods الموجودة داخل Instance عن طريق كتابة اسم Instance الذي قمنا بإنشائه ثم إضافة النقطة . بعد ذلك سيظهر لك Xcode جميع Properties و Methods داخل هذا Instance. بعد الوصول لأي من Properties بالإمكان استعراض القيمة الموجودة داخله أو تحريرها:

```
user1.phoneNumber = "001"
print(user1.name)
print(user1.phoneNumber)
```

الخصائص المشتركة بين Class و Struct

من أول نظرة إلى Struct نلاحظ بأنه يوجد شبه بين Struct و Class حيث أن الاثنان لديهما هذه الخصائص المشتركة:

- بالإمكان تعريف Properties و Methods في struct و class.
- بالإمكان تعريف Initializers في struct و class.
- بالإمكان إضافة Extended لهما لإضافة المزيد من الخصائص.
- بالإمكان تطبيق Protocols عليهما.

الفرق بين Class و Struct

- في Class يمكن إضافة خاصية الوراثة بينما لا يمكن أن يرث Struct من Struct آخر.
 - أثناء تعريف Class يجب إسناد قيم له أو إضافة دالة init function عكس Struct.
- بهذا الشكل Class فعند إنشاء كائن من Reference type هو Class:

```
class Animal {
    var playSound = ""
}

let horse1 = Animal()
let horse2 = horse1
```

لاحظ في السطر الخامس عند قراءته قد تعتقد بأنه تم إنشاء كائن جديد من horse1، ولكن في الحقيقة هو نفس الكائن horse1، أي كأنه أصبح لدينا كائن واحد له اسمين، بينما لو تم تحويل هذا Class إلى Struct كما هو موضح في المثال التالي:

```
struct Animal {
```

```
var playSound = ""
    }
let horse1 = Animal()
let horse2 = horse1
```

الآن أصبح لدينا كائنين مستقلين الأول اسمه horse1 والثاني اسمه horse2.

Type Casting

من نوع آخر instance وكأنه instance أو طريقة لمعالجة instance هو طريقة للتحقق من نوع type casting

ولكن ما فائدة ذلك؟

يعد مفهوم polymorphism من المفاهيم الأساسية في أسلوب البرمجة بالكائنات OOP.

استخدم المثال التالي لتوضيح هذا المفهوم:

```
class Person {
var name :String
init(name:String) {
self.name = name }
func greeting(){ }
}
class Teacher: Person {
var salary = 10_000
override func greeting(){
print ("hello, my name is \(name), and I am a teacher") }
func printSalary(){
print("My salary is : \(salary)")
}}
class Student: Person {
var gpa = 4.5
override func greeting(){
print ("hello, my name is \(name), and I am a student") }
func printGpa()
{print("My GPA is : \(gpa)")}
}}
```

```
var people : [Person] = [Teacher(name : "Maha"),
Student(name:"Noura"),Teacher(name : "Ahmed")]
for person in people{
person.greeting() }
```

المخرجات

```
hello, my name is Maha, and I am a teacher
hello, my name is Noura, and I am a student
hello, my name is Ahmed, and I am a teacher
```

لاحظ التنفيذ المختلف ل `greeting` بناء على نوع `instance` ! وهذا هو المعنى الأول لل `Type Casting` وهو التحقق من نوع `instance` ليتم تنفيذ الدالة بناء عليه.

افتراض أنك تريد الالتفاف حول المصفوفة السابقة واستدعاء `greeting` على العناصر من النوع `Teacher` فقط. إذا كان بالإمكان خلال كل دورة من دورات الحلقة التحقق من نوع `instance` الحالي سيكون من السهل تنفيذ هذه العملية. توفر لنا لغة سويفت طريقة للتحقق من نوع ال `instance` عن طريق العامل `is` والذي يستخدم للتحقق مما إذا كان ال `instance` ينتمي لنوع معين ويرجع `true` أو `false` بناء على ذلك.

مثال

لاحظ نتيجة إضافة جملة `is` على الحلقة في الكود السابق:

```
for person in people{
if person is Teacher{
person.greeting()
}
}
```

المخرجات

```
hello, my name is Maha, and I am a teacher
hello, my name is Ahmed, and I am a teacher
```

خلال كل دورة من دورات الحلقة سيتم التحقق من نوع `instance` الحالي، فإذا كان من النوع `Teacher` ستكون القيمة المرجعة `true` وبالتالي يتم نداء `greeting`.

يمكن استخدام العامل `is` مع جملة `switch` , الكود التالي يعطي نتيجة مماثلة للكود السابق:

```

for person in people {
switch person{
case is Teacher:
person.greeting()
default:
break
} }

```

Downcasting

والآن افترض أننا أردنا الالتفاف على المصفوفة people ونداء printSalary على instances من النوع Teacher و printGpa على instances من النوع Student؟
القاعدة الأساسية هي أن methods التي يمكن مناداتها هي فقط methods المتوفرة في reference أما التنفيذ فيتم تحديده بحسب نوع instance.

أي أنه في المثال السابق لا يمكن مناداة printSalary من instance يكون reference الخاص به هو Person لأن printSalary غير موجودة في Person .

ولحل هذه المشكلة نستخدم المفهوم الآخر ل type casting وهو معاملة instance وكأنه instance من نوع آخر.

as operator

حين تريد معاملة instance نوعه من superclass وكأنه instance من نوع subclass سيكون عليك إجراء downcasting وذلك بواسطة العامل as.

ولأن عملية downcasting قد تفشل، فإن as تأتي على شكلين :

من النوع المطلوب التحويل إليه optional والتي ترجع قيمة ?as

في الوقت ذاته force-unwraps والتي تقوم بمحاولة التحويل و !as

بالتالي نستخدم ?as حين لا نكون متأكدين من نجاح عملية التحويل , هذه العملية سترجع القيمة nil في حال عدم نجاح عملية التحويل , وترجع قيمة optional من النوع المطلوب التحويل إليه في حال نجاح العملية .
ونستخدم !as فقط حين نكون متأكدين من نجاح عملية downcasting , وفي حال فشل عملية التحويل فسيحدث خطأ وقت التشغيل.

والآن لنعد إلى المثال السابق ونقوم باستدعاء methods بحسب نوع instance:

```

for person in people{
if let teacher = person as? Teacher {
teacher.printSalary()
} else if let student = person as? Student {
student.printGpa()
}
}

```

```
}  
}
```

النتيجة :

```
My salary is : 10000  
My GPA is : 4.5  
My salary is : 10000
```

بالنوع as! لاحظ المثال التالي:

```
var person : Person = Student(name:"Khalid")  
(person as! Student).printGpa()
```

النتيجة :

```
My GPA is : 4.5
```

ولكن ما الذي سيحصل في الحالة التالية:

```
var person : Person = Teacher(name:"Khalid")  
(person as! Student).printGpa()
```

النتيجة خطأ وقت التشغيل والسبب أن العامل as! سيقوم بمحاولة التحويل بالقوة في حين أنه لا يمكن التحويل من Teacher إلى Student .

جملة switch التالية تكافئ الحلقة في الكود السابق:

```
for person in people{  
  switch person{  
    case let teacher as Teacher:  
      teacher.printSalary()  
    case let student as Student:  
      student.printGpa()  
    default:  
      break  
  } }  
}
```

مقدمة في Closures

يمكن تعريف Closure أنه عبارة عن دالة بدون اسم، أي أنه مجموعة من الأوامر بين أقواس { } وبالإمكان تمرير هذه الأقواس كمُدخل إلى دالة، حتى تتضح لك الصورة أكثر لاحظ المثال التالي:

```
myFunction1( 8921 )
myFunction2( "Welcome!" )
myFunction3({
    print(" this is inside a closure " )
})
```

السطر الأول: يحتوي على دالة تستقبل مُدخل من نوع Int.

السطر الثاني: يحتوي على دالة تستقبل مُدخل من نوع String .

السطر الثالث: يحتوي على دالة تستقبل مُدخل عبارة عن Closure .

تعريف Closure:

ذكرنا سابقاً أن Closure هو عبارة عن مجموعة من الأوامر بين أقواس { } وبالإمكان تمرير هذه الأقواس كمُدخل إلى دالة، والآن سنقوم بتعريف دالة تستقبل Closure كمُدخل:

```
func calculator(number1:Int , number2:Int , operation: (Int,Int)-> Int) -> Int {
    return operation(number1,number2)
}
```


في هذا المثال تم تعريف دالة تستقبل مدخل أول نوعه `Int` ومدخل ثاني نوعه `Int` ومدخل ثالث نوعه دالة، ثم تقوم بتنفيذ عملية حسابية بناءً على الدالة المدخلة. وحتى نستطيع استدعاء الدالة `calculator` سنحتاج إلى رقمين ودالة للمدخلات، في البداية سنقوم بتعريف دالة جديدة تجمع بين رقمين لأننا نرغب باستخدامها كمدخل في الدالة `calculator` لاحقاً:

```
func add(number1:Int , number2:Int)-> Int{
    return number1 + number2
}
```

لو نظرنا في الدالة التالية سنجد بأن هيكلها الأساسي بهذا الشكل:

```
(Int , Int ) -> Int
```

عبارة عن مدخلين من نوع `Int` وتقوم بإرجاع قيمة نوعها `Int` الآن سنقوم بأخذ هذا السطر:

```
Int , Int ) -> Int
```

ونضيفه كمدخل إلى الدالة `calculator`:

```
func calculator(number1:Int , number2:Int , operation: (Int,Int)-> Int) -> Int {
    return operation(number1,number2)
}

func add(number1:Int , number2:Int)-> Int{
    return number1 + number2
}

calculator(number1: 3, number2: 7, operation: add)
```

لاحظ أنه في السطر السابع تم استدعاء الدالة `calculator` وتم إعطاؤها مُدخلين من نوع `Int` ومدخل ثالث عبارة عن الدالة `add` التي قمنا بتعريفها في المثال أعلاه.

تحويل الدالة إلى Closure:

لاحظ في الدالة `add` في المثال أعلاه أنه لم يتم كتابة أقواس الاستدعاء () بل تم تمريرها داخل الدالة `calculator` وحيث أننا لن نحتاج إلى استدعائها في مكان آخر في البرنامج فيمكن الاستغناء عن تعريفها الحالي وتحويلها إلى Closure في ثلاث خطوات:

```
func add(number1:Int , number2:Int)-> Int{
    return number1 + number2
}
```

● الخطوة الأولى: إزالة الكلمة المفتاحية `func` و اسم الدالة:

```
(number1:Int , number2:Int)-> Int{
    return number1 + number2
}
```

● الخطوة الثانية: تحريك قوس بداية الدالة في السطر الأول وجعله في بداية السطر:

```
{ (number1:Int , number2:Int)-> Int
    return number1 + number2
}
```

- الخطوة الثالثة: لاحظ الآن عدم وجود فاصل بين المدخلات **parameters** وباقي الدالة، لذا سنحتاج إلى إضافة الكلمة المفتاحية **in** نهاية السطر الأول:

```
{ (number1:Int , number2:Int)-> Int in
    return number1 + number2
}
```

الآن بهذا الشكل تم تحويل الدالة إلى **Closure** وأصبح جاهز للتمرير داخل دالة أخرى، لذا سنقوم بتمريره إلى الدالة **:calculator**

```
func calculator(number1:Int , number2:Int , operation: (Int,Int)-> Int) -> Int {
    return operation(number1,number2)
}
```

```
calculator(number1: 3, number2: 7, operation: { (number1:Int , number2:Int)-> Int in
    return number1 + number2
})
```

لاحظ أن المدخل الثالث والموجود في السطر الخامس عندما استدعينا الدالة **calculator** أصبح عبارة عن **Closure**.

الفرق بين **NonEscaping Closure** و **Escaping Closure**

النوع **NonEscaping Closure** عندما يتم تنفيذ **Closure** مباشرة داخل الدالة بدون تأخير عن باقي أكواد الدالة:

```
func nonEscaping ( closure : (String)-> Void){
print("function start")
    closure("closure start")
    print("function end")
}
```

في هذا المثال تم تعريف دالة تحتوي على مدخل واحد عبارة عن **Closure**، وعند استدعائها سيكون تنفيذ **Closure** مباشرة داخلها، ولكي تستدعيها سنقوم بالتالي:

```
nonEscaping { (myString) in
    print(myString)
}
```

وعند تشغيل البرنامج ستكون المخرجات بهذا الترتيب:

```
function start
closure start
function end
```

لاحظ أنه تم تنفيذ Closure داخل الدالة مباشرة، وهذه هي Non Escaping Closure. النوع Escaping Closure وهو عندما يتم تنفيذ Function كاملة ولكن تبقى جزئية Closure لم يتم تنفيذها بعد وقد تتأخر لعدة أسباب، كأن يتم الاستعلام عن حالة الطقس عبر الإنترنت وتتأخر البيانات في الوصول أو يتم تحميل ملف ويستغرق تحميله بعض الوقت، سنقوم بالتعديل على المثال الأول ونقوم بتأخير تنفيذ Closure خمس ثواني كما هو موضح في المثال التالي:

```
func escaping (closure: (String)-> Void) {
    print("function start")
}
```

```
DispatchQueue.main.asyncAfter(deadline: .now() + 5) {
    closure("closure start")
}

print("function end")
}
```

لوقمت بكتابة هذا الكود في xcode ستظهر لك رسالة خطأ تدل على أنه سيتم تنفيذ هذه الدالة ولكن سينتهي تنفيذها وسيكون هناك Closure لم يتم تنفيذه وقت تنفيذ الدالة escaping فما الحل في هذه الحالة؟
الحل بكل بساطة تحويلها إلى escaping closure عن طريق إضافة الكلمة المفتاحية @escaping بهذا الشكل:

```
func escaping (closure: @escaping(String)-> Void) {
    print("function start")
}
```

```
DispatchQueue.main.asyncAfter(deadline: .now() + 5) {
    closure("closure start")
}

print("function end")
}
```

في السطر الخامس تم تأخير تنفيذ Closure ٥ ثواني
سنقوم بالاستدعاء:

```
escaping { (value) in
    print(value)
}
```

ستكون المخرجات بهذا الشكل:

```
function start
function end
closure start
```

لاحظ أن الترتيب الآن اختلف بالإضافة إلى وقت تنفيذ Closure.

مقدمة في Protocol و Extension و Delegate

مقدمة في Protocol

مفهوم Protocol يعبر عن مجموعة من القوانين التي يتم تطبيقها على Struct أو Class.

طريقة كتابته

تستطيع كتابته باستخدام الكلمة المفتاحية protocol بالشكل التالي:

```
protocol SomeProtocol {
    // protocol definition goes here
}
```

محتوياته

بإمكانك إضافة Computed Properties بداخله وكذلك Methods، بالشكل التالي:

```
protocol FullyNamed {
    var fullName: String { get }
    func someTypeMethod()
}
```

نلاحظ في المثال أعلاه أننا قمنا بتعريف Computed Property باسم fullName و Method باسم someTypeMethod.

استخدامه

نستطيع استخدام Protocol عن طريق إضافته إلى Struct أو Class ويكون اسم Protocol بعدها ويسمى Adopt، والذي يعني أن Struct مثلاً يتبع Protocol ما، بغض النظر عن ما إذا كان قد قام بتنفيذ ما بداخل Protocol أو لا، وستكون بالشكل التالي:

```
protocol FirstProtocol {
    // protocol definition goes here
}

protocol AnotherProtocol {
    // protocol definition goes here
}
```

```
struct SomeStructure: FirstProtocol, AnotherProtocol {
    // structure definition goes here
}
```

كما نرى في السطر الثامن `SomeStructure` نقوم باستخدام أكثر من `Protocol` بعد علامة النقطتان الرأسيتان. إذا كان `Protocol` له محتويات وتريد استخدامها فعليك إذا استخدام جميع ما بداخله وهذا يُعرف بمسمى `Conform to protocol`، ويكون بالشكل التالي:

```
protocol Flyable {
    func fly()
}
struct Plane : Flyable {
    func fly() {
        print("plane is flying")
    }
}
```

في المثال أعلاه لدينا `protocol Flyable` و `struct Plane` سنقوم باستخدامه، وكما ترى `struct Plane` قامت باستخدام الدالة `fly()` بالفعل.

مفهوم Delegation

هذا المفهوم الذي يستخدم `Protocol` في تطبيقه وسترى استخدامه في لغة `Swift` بشكل كبير، ويعني كأنك تقوم بتوكيل شخص آخر وهو `class` أو `struct` ليتقوم بتنفيذ مهمة ما لا تستطيع تنفيذها بنفسك، ولتوضيح الفكرة تخيل أن لديك طفل بحاجة لأن يأكل، هذا يعني أننا نستطيع إنشاء `Protocol` يخص الطبخ وليكن:

```
protocol Cook {
    func makingFood()
}
```

وإن كانت الأم هي من ستعد الطعام لنقم بإنشائها بالشكل التالي:

```
struct Mom: Cook {
    func makingFood() {
        print("making food!")
    }
}
// Make a mom
var mom = Mom()
```

والآن لنقم بإنشاء ما يمثل الطفل، وبما أن الطفل غير قادر على الطبخ فسننشئ بالغا بداخله، شخص ما قادر على إعداد الطعام له ولنسمه `delegate`:

```
// Make a baby
struct Baby {
    var delegate: Cook? // delegate = someone with special skills
}
```

والآن لنقم بإنشاء الطفل ونحدد له من سيعد الطعام ونسند له الأم في `delegate`:

```
// Create a baby
var babe = Baby()
babe.delegate = mom
```

والآن وحتى نتمكن من الوصول إلى إعداد الطعام للطفل سيكون ذلك عن طريق:

```
babe.delegate?.makingFood()
```

بهذا الشكل استطعنا الوصول إلى `makingFood` والتي تقوم بها الأم باستخدام `delegate`

مفهوم Extension

يمكن القول أن `Extension` هي توصيلة، تستطيع أن تضيفها على أي نوع شئت بعد تعريفه مثل `struct` و `class` و `enum` و `protocol`، والتي تمكّنك من إضافة خصائص جديدة لم تكن موجودة عند تعريفها للمرة الأولى.

طريقة كتابتها

تستخدم الكلمة المفتاحية `extension` يتبعها النوع الذي تريد إضافة خصائص جديدة أو `Methods` له، لاحظ معي المثال التالي:

```
extension Int {
    var cubed: Int {
        return self * self * self
    }
}
```

لاحظ في المثال أعلاه أننا استخدمنا `Extension` مع النوع `Int`، وبداخله قمنا بإضافة `Computed Property` جديدة له وهي `cubed`.

إضافة محتوى لها

باستخدام `Extension`، يمكننا إضافة إما `Properties` أو `Methods`.

إضافة Properties

النوع الوحيد من `Properties` الذي يمكن إضافته باستخدام `Extension` هو `Computed Properties`، فمثلاً لدينا النوع `Double` ونريد إضافة خصائص عليه، على سبيل المثال لو كنا نريد أن يرجع لنا بقيمة من نوع `Double`، ويقوم بتحويلها مثلاً إلى كيلو متر نستطيع تحقيق ذلك عن طريق استخدام `Extension` بالشكل التالي:

```
extension Double {
```

```
var km: Double { return self * 1_000.0 }  
}
```

ويتم استخدامها بالشكل التالي:

```
let oneKm = 25.4.km
```

لاحظ في المثال أعلاه أننا قمنا باستدعاء `km property` الذي سبق وقمنا بإنشائها في النوع `Double` بذلك سيقوم بتحويل القيمة المُسندة إلى كيلومتر مباشرة وقت مناداتنا لها وسيتم إسنادها للمتغير.

إضافة Method

في حال أردنا إضافة `Method` على النوع `Int`، فيمكن إضافتها بالشكل التالي:

```
extension Int {  
    func printMyInt() {  
        print("my integer is \(self)")  
    }  
}
```

ونستطيع الآن مناداتها واستخدامها كذلك.

ملاحظة: أحد عيوب `Extension` أننا لانستطيع تغيير الخصائص الموجودة مسبقاً في النوع `Extension` مصممة لإضافة الخصائص وليست مصممة لتغيير الخصائص الموجودة مسبقاً.

مقدمة في Error Handling و Defer

مقدمة في Error handling

الأخطاء عديدة وبالإمكان معرفة سببها حين تكون خطأ في الكتابة Syntax Error أو حين تكون Logic Error بمعنى خطأ في منطقية سير البرنامج على سبيل المثال أن تكون لديك Method تقوم بالجمع ولكنك بالخطأ جعلتها تقوم بالضرب. ولكن هناك أخطاء قد لا تكون خطأ من المبرمج نفسه كالحالات الأنف ذكرها، بل قد تكون بسبب أمر يحدث فجأة كإنقطاع الانترنت عن الجهاز مما يتسبب في تعذر الوصول للصفحة المطلوبة على سبيل المثال، وتسمى هذه الأخطاء Recoverable Errors والتي تعني أخطاء يمكن التعامل معها بشكل أو بآخر في حال حدوثها، كأن تخبر المستخدم على سبيل المثال بأن السبب في عدم فتح الصفحة ليس مشكلة فيها بل مشكلة في الانترنت وأن البيانات سيتم تحميلها في حال قام بالاتصال بالانترنت مرة أخرى. للتعامل مع الأخطاء نحتاج لثلاث خطوات أساسية:

- تحديد وتعريف الخطأ.
- تحديد مكان ظهوره.
- كيفية التعامل معه في حال حدوثه.

تحديد الأخطاء

بعد أن تكتب برنامجك فكر بالأخطاء الوارد حدوثها، ولا يقصد بها هنا الأخطاء التي يمكن ملاحظتها برمجياً، بل الأخطاء التي من الممكن أن تحصل وتريد أن تتعامل معها في حال حصلت. مثال: طول كلمة المرور، فلنفترض أنك تريد أن يدخل المستخدم كلمة مرور يتراوح طولها بين ٧ إلى ١٥ حرف، وتريد أن يظهر له توضيح أو رسالة خطأ في حال أنه قام بإدخال عدد أحرف أقل أو أكثر من ذلك. ومن أفضل الطرق لتحديد مجموعة متشابهة من البيانات هي استخدام enum لذا سنستخدمها لتوضيح أنواع الأخطاء الواردة بالشكل التالي:

```
enum LengthError{  
    case maxLength  
    case minLength  
}
```

ولكن الآن لا أحد يعرف أنها أخطاء من الممكن ظهورها سوانا، لذلك نحتاج أن نخبر Swift بأن هذا enum يحتوي على أخطاء واردة الحدوث. ويكون ذلك عن طريق إضافة Protocol يمثل الأخطاء في Swift، يسمى Error ويتميز بكونه لا يجبرنا على إضافة Properties أو Methods إضافية ولكنه يسمح لبرنامجك بالتعامل مع الأخطاء التي حددتها كما لو كانت أخطاء حقيقية.

```
enum LengthError : Error {  
    case maxLength  
    case minLength  
}
```

أين سيظهر الخطأ؟

لنستطيع التعامل مع كلمة المرور لنفترض أن لدينا الدالة التالية التي تقوم بالتأكد من الطول وطباعة جملة تحدد نوع الخطأ وترجع لنا قيمة نصية:

```
func checkPassLength(length : Int) -> String{
```



```

    if length < 7 {
        print("length is less than 7")
    } else if length > 15 {
        print("length is greater than 15 ")
    }
    return "length is valid"
}

```

بما أن الخطأ قد يظهر داخل if أو if else فنحن بحاجة لأن نخبر Swift بذلك، عن طريق إضافة الكلمة المفتاحية **throws** إلى عنوان **checkPassLength** بالشكل التالي:

```

func checkPassLength(length : Int) throws -> String{
    //
}

```

بهذا الشكل كأننا نقول أن هذه **Method** قد يحدث بداخلها خطأ ما. ولكن يتبقى لنا أن نجعل الخطأ يحدث! وذلك باستخدام **throw** **إحداث الخطأ**

لنقوم بإحداث الخطأ بعد أن حددنا أن **checkPassLength** قد يحدث بداخلها خطأ ما، نحدد الأماكن في برنامجنا التي نريد أن نظهر الخطأ بها وهي في حالتنا عندما يكون طول كلمة المرور اقل من ٧ أو أكثر من ١٥، لذلك سنقوم بإضافة جملة داخل كل شرط ونظهر الأخطاء التي قمنا بتحديددها سابقاً بداخل **LengthError** بالشكل التالي:

```

func checkPassLength(length : Int) throws -> String{
    if length < 7 {
        print("length is less than 7")
        throw LengthError.minLength
    } else if length > 15 {
        print("length is greater than 15 ")
        throw LengthError.maxLength
    }
    return "length is valid"
}

```

لاحظ استخدامنا للكلمة **throw** والتي تعني إحداث الخطأ، ونلاحظ تحديدنا لنوع الخطأ باستخدام **LengthError**. **كيف نتعامل مع الخطأ؟**

عند محاولتنا لاستخدام **checkPassLength** بالشكل التالي:

```

let passwordLength = checkPassLength(length: 10)
print(passwordLength)

```

يظهر لنا خطأ بسبب أننا حددنا `checkPassLength` على أنه قد يحدث بداخلها خطأ ما، ولكننا لم نتعامل معه حتى الآن ولم نحدد ماذا سنفعل في حال حدوثه.

طريقة `do..catch`

هناك العديد من الطرق للتعامل مع الأخطاء، أحدها `do..catch` وتكتب بالشكل التالي:

```
do {  
  try expression  
    //statements  
catch {  
  //statements  
}
```

تقوم `do` بتنفيذ ما بداخلها وتحتوي على جملة `try` والتي تأتي قبل استدعاء `checkPassLength` لأنه من المحتمل أن يحدث فيها الخطأ، وفي داخل `catch` نقوم بإمساك الخطأ وبإمكاننا طباعته حتى عن طريق استخدام `error` والذي سيقوم بطباعة نوع الخطأ الذي قد يظهر.

لنقوم بتطبيقها على برنامجنا ستكون بالشكل التالي:

```
do {  
  let passwordLength = try checkPassLength(length:17)  
  print(passwordLength)}  
catch{  
  print(error)  
}
```

عند إرسالنا للرقم 17 سيظهر خطأ `LengthError.maxLength` وفي نفس الوقت ستحمي جملة `catch` برنامجنا من أن يُعطب بسبب ظهور الخطأ أو أن يتوقف حتى!

استخدام `try?`

في حال لم نكن نريد معرفة نوع الخطأ، ولا تهمة معرفته بإمكاننا استخدام `optional try` والتي تمكننا من التعامل مع `checkPassLength` وتظهر لنا القيمة إن كانت تحتوي على قيمة، وإن ظهر خطأ ما فلن يحدث شيء وستكون `nil` بطبيعة الحال بالشكل التالي:

```
let passwordLength = try? checkPassLength(length: 17)  
print(passwordLength)
```

استخدام `try!`

في حال كنا متأكدين من `checkPassLength` لن يظهر لنا خطأ ابداً، بإمكاننا استخدام `Force unwrapping try` ولكن استخدامها بهذا الشكل خطير جداً، حيث أنه في حال حصل خطأ ما فجأة فسيوقف برنامجك عن العمل، لذلك الأفضل ألا نستخدمها، وعلى كل حال ستكون بالشكل التالي:

```
let passwordLength = try! checkPassLength(length: 10)
```

```
print(passwordLength)
```

مفهوم Defer

نستخدم الكلمة المفتاحية **defer** عندما نريد تأجيل تنفيذ مبادخلها إلى وقت ما قبل انتهاء تنفيذ الدالة، بمعنى لا يتم تنفيذ مبادخلها حتى ينتهي تنفيذ جميع ما بداخل الدالة.
على سبيل المثال:

```
func simpleDefer() {  
    defer { print("later") }  
    print("Print First")  
}
```

في المثال أعلاه قمنا بإنشاء دالة بداخلها **defer** بداخله جملة طباعة لاحظ أنه عند تشغيل البرنامج ستتم طباعة **Print First** ومن ثم **later**.

مقدمة في Error Handling و Defer

مقدمة في Error handling

الأخطاء عديدة وبالإمكان معرفة سببها حين تكون خطأ في الكتابة **Syntax Error** أو حين تكون **Logic Error** بمعنى خطأ في منطقية سير البرنامج على سبيل المثال أن تكون لديك **Method** تقوم بالجمع ولكنك بالخطأ جعلتها تقوم بالضرب. ولكن هناك أخطاء قد لا تكون خطأ من المبرمج نفسه كالحالات الأنف ذكرها، بل قد تكون بسبب أمر يحدث فجأة كإنقطاع الانترنت عن الجهاز مما يتسبب في تعذر الوصول للصفحة المطلوبة على سبيل المثال، وتسمى هذه الأخطاء **Recoverable Errors** والتي تعني أخطاء يمكن التعامل معها بشكل أو بآخر في حال حدوثها، كأن تخبر المستخدم على سبيل المثال بأن السبب في عدم فتح الصفحة ليس مشكلة فيها بل مشكلة في الانترنت وأن البيانات سيتم تحميلها في حال قام بالاتصال بالانترنت مرة أخرى.
لنتعامل مع الأخطاء نحتاج لثلاث خطوات أساسية:

- تحديد وتعريف الخطأ.
- تحديد مكان ظهوره.
- كيفية التعامل معه في حال حدوثه.

تحديد الأخطاء

بعد أن تكتب برنامجك فكر بالأخطاء الوارد حدوثها، ولا يقصد بها هنا الأخطاء التي يمكن ملاحظتها برمجياً، بل الأخطاء التي من الممكن أن تحصل وتريد أن تتعامل معها في حال حصلت.
مثال: طول كلمة المرور، فلنفترض أنك تريد أن يدخل المستخدم كلمة مرور يتراوح طولها بين ٧ إلى ١٥ حرف، وتريد أن يظهر له توضيح أو رسالة خطأ في حال أنه قام بإدخال عدد أحرف أقل أو أكثر من ذلك.
ومن أفضل الطرق لتحديد مجموعة متشابهة من البيانات هي استخدام **enum** لذا سنستخدمها لتوضيح أنواع الأخطاء الواردة بالشكل التالي:

```
enum LengthError{  
    case maxLength  
    case minLength  
}
```

ولكن الآن لا أحد يعرف أنها أخطاء من الممكن ظهورها سوانا، لذلك نحتاج أن نخبر Swift بأن هذا enum يحتوي على أخطاء واردة الحدوث. ويكون ذلك عن طريق إضافة Protocol يمثل الأخطاء في Swift، يسمى Error ويتميز بكونه لا يجبرنا على إضافة Properties أو Methods إضافية ولكنه يسمح لبرنامجك بالتعامل مع الأخطاء التي حددتها كما لو كانت أخطاء حقيقية.

```
enum LengthError : Error {  
    case maxLength  
    case minLength  
}
```

أين سيظهر الخطأ؟

لنستطيع التعامل مع كلمة المرور لنفترض أن لدينا الدالة التالية التي تقوم بالتأكد من الطول وطباعة جملة تحدد نوع الخطأ وترجع لنا قيمة نصية:

```
func checkPassLength(length : Int) -> String{  
    if length < 7 {  
        print("length is less than 7")  
    } else if length > 15 {  
        print("length is greater than 15 ")  
    }  
    return "length is valid"  
}
```

بما أن الخطأ قد يظهر داخل if أو if else فنحن بحاجة لأن نخبر Swift بذلك، عن طريق إضافة الكلمة المفتاحية throws إلى عنوان checkPassLength بالشكل التالي:

```
func checkPassLength(length : Int) throws -> String{  
    //  
}
```

بهذا الشكل كأننا نقول أن هذه Method قد يحدث بداخلها خطأ ما. ولكن يتبقى لنا أن نجعل الخطأ يحدث! وذلك باستخدام throw

إحداث الخطأ

لنقوم بإحداث الخطأ بعد أن حددنا أن checkPassLength قد يحدث بداخلها خطأ ما، نحدد الأماكن في برنامجنا التي نريد أن نظهر الخطأ بها وهي في حالتنا عندما يكون طول كلمة المرور اقل من ٧ أو أكثر من ١٥، لذلك سنقوم بإضافة جملة داخل كل شرط ونظهر الأخطاء التي قمنا بتحديدتها سابقاً بداخل LengthError بالشكل التالي:

```
func checkPassLength(length : Int) throws -> String{  
    if length < 7 {  
        print("length is less than 7")  
        throw LengthError.minLength  
    }
```

```

    } else if length > 15 {
        print("length is greater than 15 ")
        throw LengthError.maxLength
    }
    return "length is valid"
}

```

لاحظ استخدامنا للكلمة **throw** والتي تعني إحداث الخطأ، ونلاحظ تحديدنا لنوع الخطأ باستخدام **LengthError**.
كيف نتعامل مع الخطأ؟

عند محاولتنا لاستخدام **checkPassLength** بالشكل التالي:

```

let passwordLength = checkPassLength(length: 10)
print(passwordLength)

```

يظهر لنا خطأ بسبب أننا حددنا **checkPassLength** على أنه قد يحدث بداخلها خطأ ما، ولكننا لم نتعامل معه حتى الآن ولم نحدد ماذا سنفعل في حال حدوثه.

طريقة *do..catch*

هنالك العديد من الطرق للتعامل مع الأخطاء، أحدها **do..catch** وتكتب بالشكل التالي:

```

do {
    try expression
    //statements
} catch {
    //statements
}

```

تقوم **do** بتنفيذ ما بداخلها وتحتوي على جملة **try** والتي تأتي قبل استدعاء **checkPassLength** لأنه من المحتمل أن يحدث فيها الخطأ، وفي داخل **catch** نقوم بإمساك الخطأ وإمكاننا طباعته حتى عن طريق استخدام **error** والذي سيقوم بطباعة نوع الخطأ الذي قد يظهر.

لنقوم بتطبيقها على برنامجنا ستكون بالشكل التالي:

```

do {
    let passwordLength = try checkPassLength(length:17)
    print(passwordLength)}
catch{
    print(error)
}

```

عند إرسالنا للرقم 17 سيظهر خطأ `LengthError.maxLength` وفي نفس الوقت ستحمي جملة `catch` برنامجنا من أن يُعطب بسبب ظهور الخطأ أو أن يتوقف حتى!

استخدام `try?`

في حال لم نكن نريد معرفة نوع الخطأ، ولا تهمة معرفته بإمكاننا استخدام `optional try` والتي تمكننا من التعامل مع `checkPassLength` وتظهر لنا القيمة إن كانت تحتوي على قيمة، وإن ظهر خطأ ما فلن يحدث شيء وستكون `nil` بطبيعة الحال بالشكل التالي:

```
let passwordLength = try? checkPassLength(length: 17)
print(passwordLength)
```

استخدام `try!`

في حال كنا متأكدين من `checkPassLength` لن يظهر لنا خطأ ابداً، بإمكاننا استخدام `Force unwrapping try` ولكن استخدامها بهذا الشكل خطير جداً، حيث أنه في حال حصل خطأ ما فجأة فسيوقف برنامجك عن العمل، لذلك الأفضل ألا نستخدمها، وعلى كل حال ستكون بالشكل التالي:

```
let passwordLength = try! checkPassLength(length: 10)
print(passwordLength)
```

مفهوم `Defer`

نستخدم الكلمة المفتاحية `defer` عندما نريد تأجيل تنفيذ مبادخلها إلى وقت ما قبل انتهاء تنفيذ الدالة، بمعنى لا يتم تنفيذ مبادخلها حتى ينتهي تنفيذ جميع ما بداخل الدالة. على سبيل المثال:

```
func simpleDefer() {
    defer { print("later") }
    print("Print First")
}
```

في المثال أعلاه قمنا بإنشاء دالة بداخلها `defer` بداخله جملة طباعة `Print First` ومن ثم `later`. لاحظ أنه عند تشغيل البرنامج ستتم طباعة `Print First` ومن ثم `later`.