EBook Gratuito

APPRENDIMENTO Swift Language

Free unaffiliated eBook created from
Stack Overflow contributors.

#swift

Sommario

Di

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: swift-language

It is an unofficial and free Swift Language ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Swift Language.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capitolo 1: Iniziare con Swift Language

Osservazioni

Swift è un linguaggio di programmazione di applicazioni e sistemi sviluppato da Apple e distribuito come open source . Swift interagisce con le API touchive Objective-C e Cocoa / Cocoa per i sistemi operativi Apple macOS, iOS, tvOS e watchOS. Swift attualmente supporta macOS e Linux. Sono in corso attività comunitarie per supportare Android, Windows e altre piattaforme.

Lo sviluppo rapido avviene su GitHub ; i contributi vengono solitamente inviati tramite richieste di pull .

Bug e altri problemi sono rintracciati su bugs.swift.org .

Le discussioni sullo sviluppo, l'evoluzione e l'utilizzo di Swift si svolgono sulle mailing list Swift .

Altre risorse

- Swift (linguaggio di programmazione) (Wikipedia)
- The Swift Programming Language (online)
- Riferimento alla libreria standard Swift (online)
- Linee guida per la progettazione dell'API (online)
- Swift Programming Series (iBooks)
- ... e altro ancora su developer.apple.com .

Versioni

Versione Swift	Versione Xcode	Data di rilascio
sviluppo iniziato (primo commit)	-	2010-07-17
1.0	Xcode 6	2014/06/02
1.1	Xcode 6.1	2014/10/16
1.2	Xcode 6.3	2015/02/09
2.0	Xcode 7	2015/06/08
2.1	Xcode 7.1	2015/09/23
debutto open-source	-	2015/12/03
2.2	Xcode 7.3	2016/03/21

Versione Swift	Versione Xcode	Data di rilascio
2.3	Xcode 8	2016/09/13
3.0	Xcode 8	2016/09/13
3.1	Xcode 8.3	2017/03/27

Examples

Il tuo primo programma Swift

Scrivi il tuo codice in un file chiamato `hello.swift` :

```
print("Hello, world!")
```

- Per compilare ed eseguire uno script in un unico passaggio, utilizzare `swift` dal terminale (in una directory in cui si trova questo file):

  Per avviare un terminale, premi `CTRL` + `ALT` + `T` su *Linux* o trovalo in Launchpad su *macOS* . Per cambiare directory, inserisci `cd` **`directory_name`** (o `cd ..` per tornare indietro)

```
$ swift hello.swift
Hello, world!
```

  Un **compilatore** è un programma per computer (o un insieme di programmi) che trasforma il codice sorgente scritto in un linguaggio di programmazione (la lingua di partenza) in un altro linguaggio (la lingua di destinazione), con quest'ultimo spesso con una forma binaria nota come codice oggetto. (Wikipedia)

- Per compilare ed eseguire separatamente, utilizzare `swiftc` :

```
$ swiftc hello.swift
```

Questo compilerà il tuo codice nel file `hello` . Per eseguirlo, immettere `./` , seguito da un nome file.

```
$ ./hello
Hello, world!
```

- Oppure usa il rapido REPL (Read-Eval-Print-Loop), digitando `swift` dalla riga di comando, quindi inserendo il codice nell'interprete:

Codice:

```
func greet(name: String, surname: String) {
    print("Greetings \(name) \(surname)")
}
```

```
let myName = "Homer"
let mySurname = "Simpson"

greet(name: myName, surname: mySurname)
```

Rompiamo questo grande codice in pezzi:

- `func greet(name: String, surname: String) { // function body }` - crea una *funzione* che prende un `name` e un `surname` .
- `print("Greetings \(name) \(surname)")` - `print("Greetings \(name) \(surname)")` alla console "Greetings", quindi `name` , quindi `surname` . Fondamentalmente `\( variable_name )` stampa il valore di quella variabile.
- `let myName = "Homer"` e `let mySurname = "Simpson"` - crea *costanti* (variabili che valutano che non puoi cambiare) usando `let` con nomi: `myName` , `mySurname` e valori: `"Homer"` , `"Simpson"` rispettivamente.
- `greet(name: myName, surname: mySurname)` - chiama una *funzione* che abbiamo creato in precedenza fornendo i valori di *constanti* `myName` , `mySurname` .

Eseguendolo utilizzando REPL:

```
$  swift
Welcome to Apple Swift. Type :help for assistance.
  1> func greet(name: String, surname: String) {
  2.     print("Greetings \(name) \(surname)")
  3. }
  4>
  5> let myName = "Homer"
myName: String = "Homer"
  6> let mySurname = "Simpson"
mySurname: String = "Simpson"
  7> greet(name: myName, surname: mySurname)
Greetings Homer Simpson
  8> ^D
```

Premere `CTRL` + `D` per uscire da REPL.

Installare Swift

Innanzitutto, scarica il compilatore e i componenti.

Successivamente, aggiungi Swift al tuo percorso. Su macOS, il percorso predefinito per la toolchain scaricabile è / Library / Developer / Toolchains. Esegui il seguente comando nel terminale:

```
export PATH=/Library/Developer/Toolchains/swift-latest.xctoolchain/usr/bin:"${PATH}"
```

Su Linux, dovrai installare clang:

```
$ sudo apt-get install clang
```

Se hai installato la toolchain Swift in una directory diversa dalla root di sistema, dovrai eseguire il seguente comando, utilizzando il percorso effettivo della tua installazione Swift:

```
$ export PATH=/path/to/Swift/usr/bin:"${PATH}"
```

Puoi verificare di avere la versione corrente di Swift eseguendo questo comando:

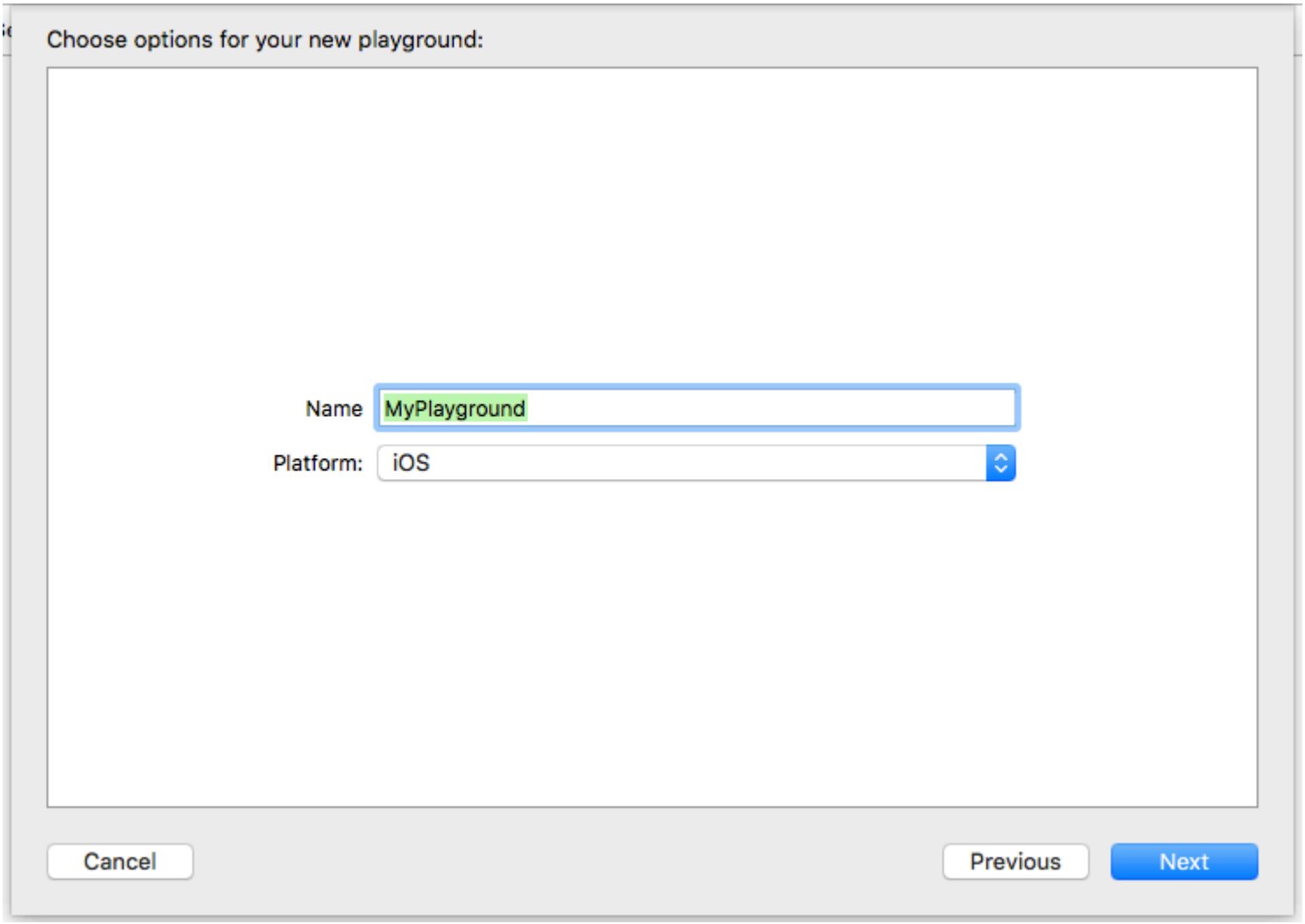
```
$ swift --version
```

Il tuo primo programma in Swift su Mac (utilizzando un parco giochi)

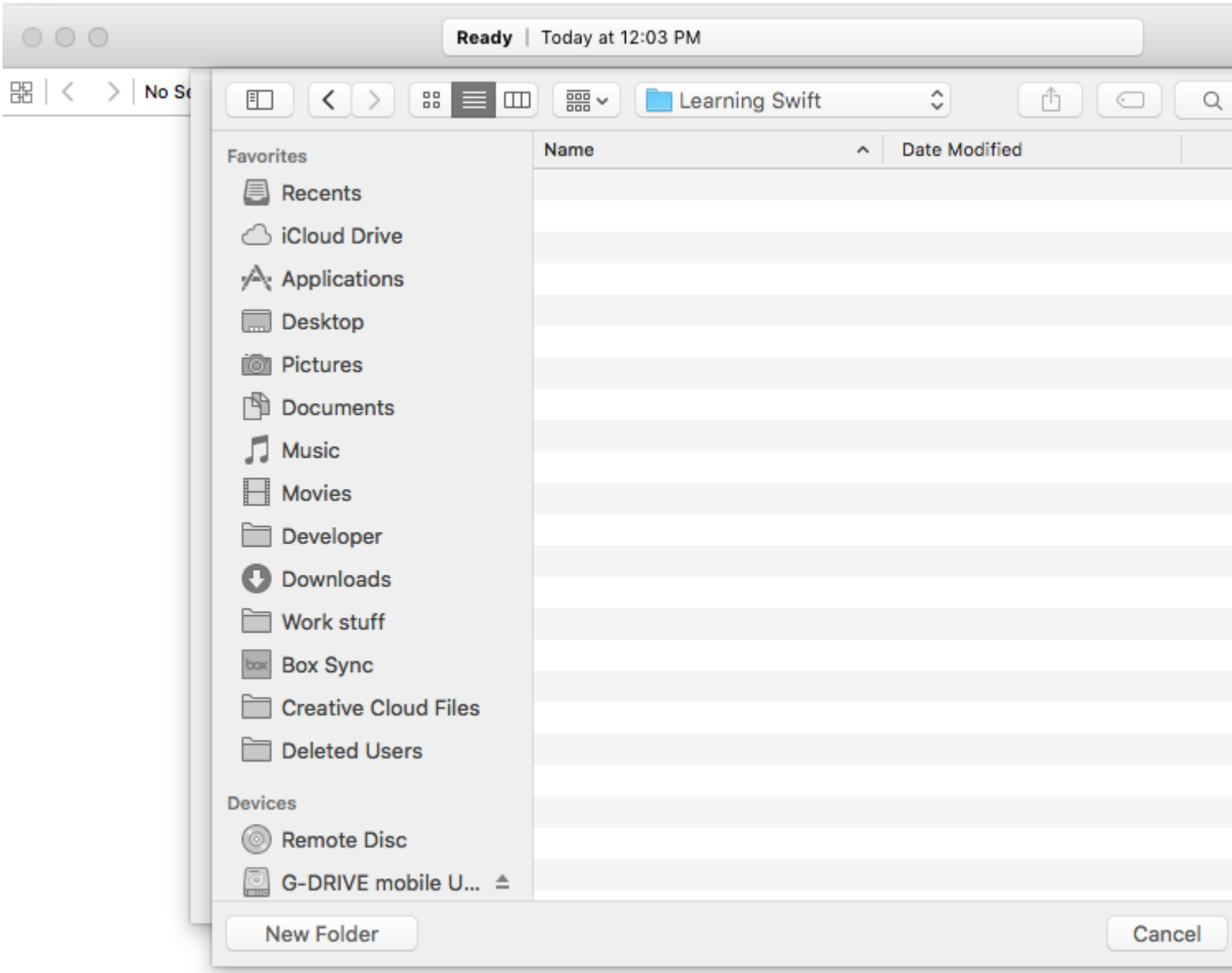
Dal tuo Mac, scarica e installa Xcode dal Mac App Store seguendo questo link .

Al termine dell'installazione, apri Xcode e seleziona **Inizia con un parco giochi** :

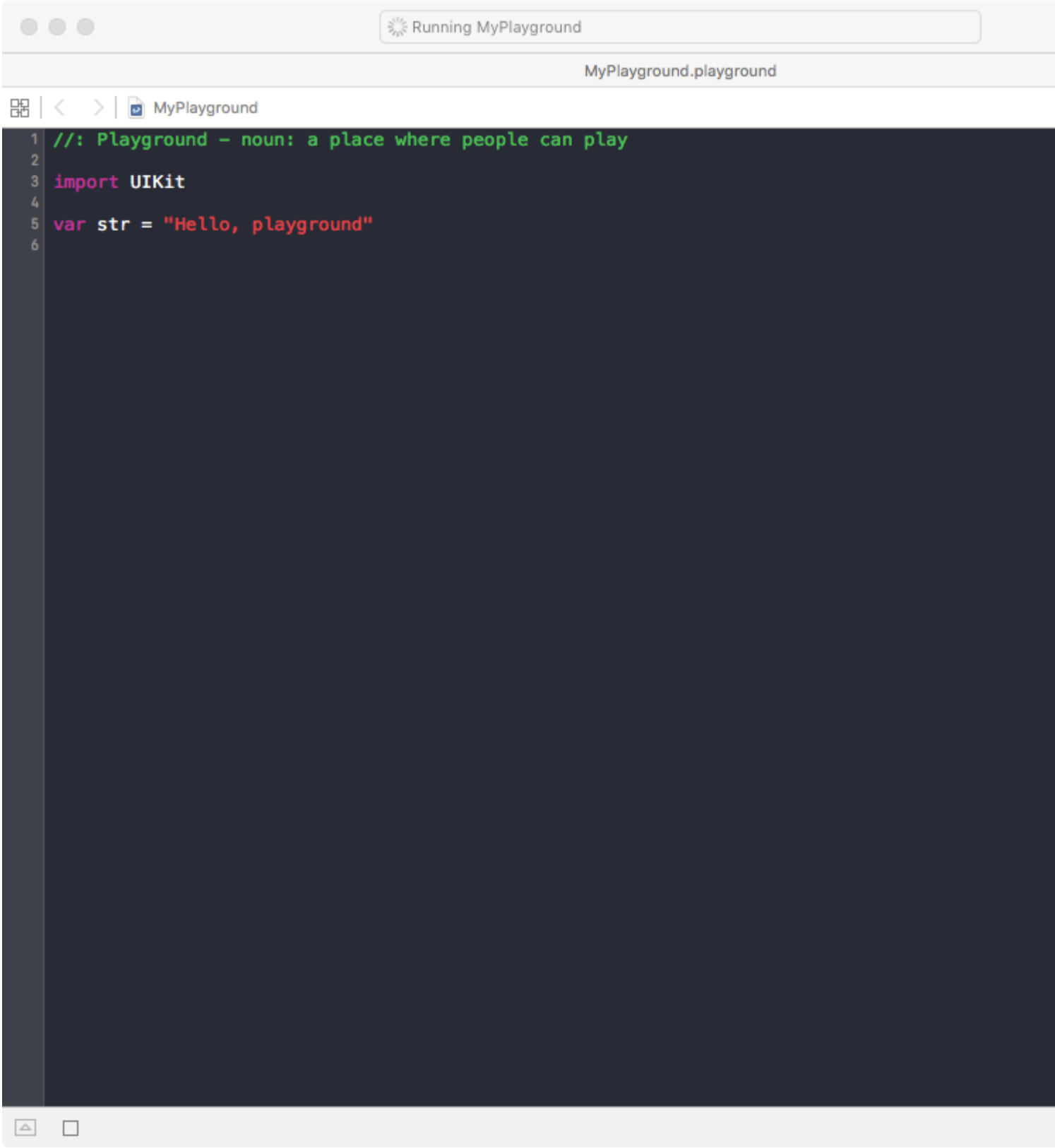
Nel pannello successivo, puoi dare un nome al tuo parco giochi oppure puoi lasciare `MyPlayground` e premere **Avanti** :



Seleziona una posizione in cui salvare il parco giochi e premi **Crea** :



Il Playground si aprirà e lo schermo dovrebbe apparire in questo modo:

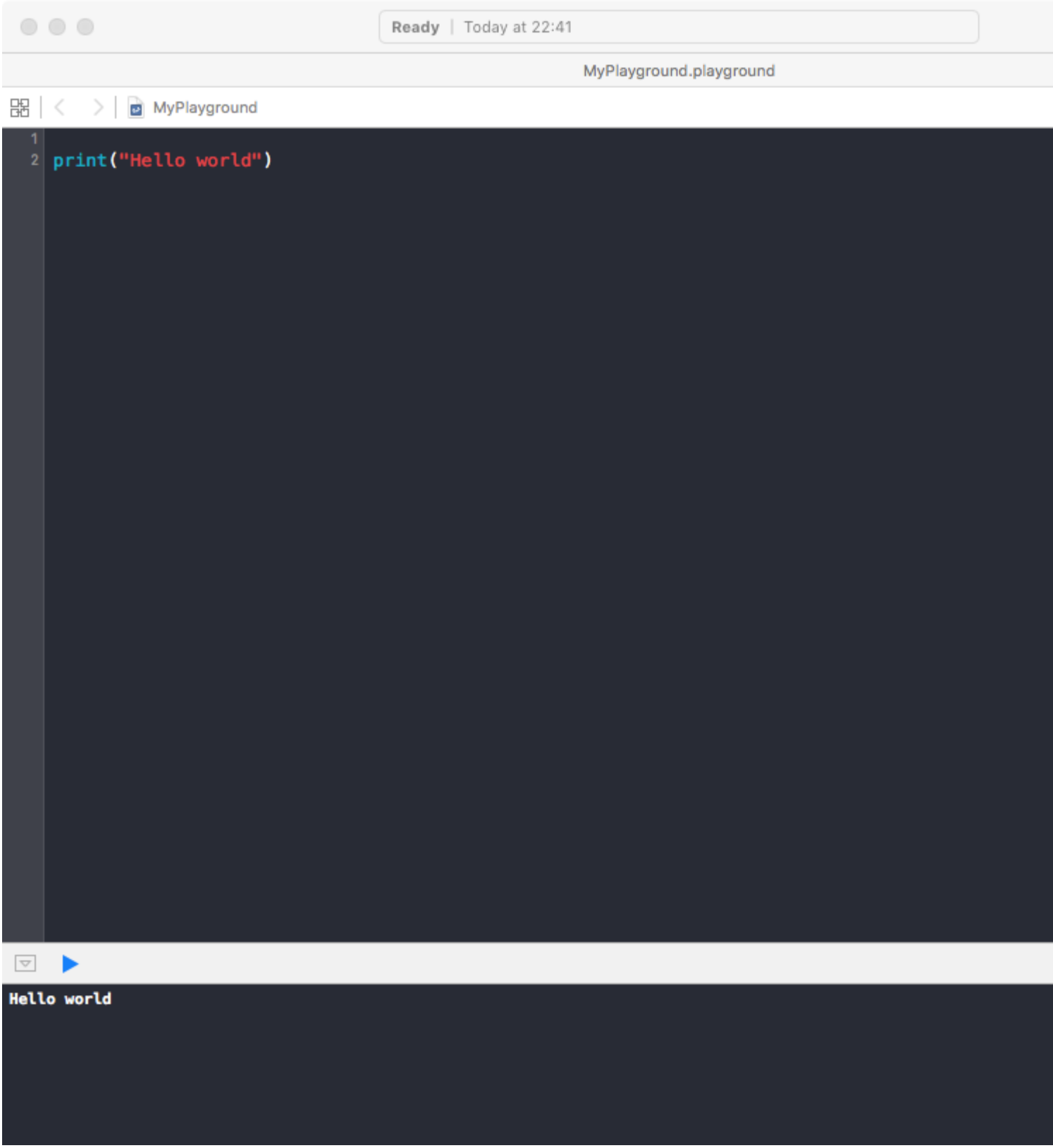


Ora che il parco giochi è sullo schermo, premi + cmd + Y per mostrare l' **area di debug** .

Infine cancella il testo all'interno del Playground e digita:

```
print("Hello world")
```

Dovresti vedere "Hello world" nell'area **Debug** e "Hello world \ n" nella **barra laterale** destra:



Congratulazioni! Hai creato il tuo primo programma in Swift!

Il tuo primo programma nell'app Swift Playgrounds su iPad

L'app Swift Playgrounds è un ottimo modo per iniziare a programmare Swift in movimento. Per usarlo:

1- Scarica Swift Playgrounds per iPad da App Store.

2- Apri l'app

3- Nella scheda I **miei parchi giochi** , tocca **+**

nell'angolo in alto a sinistra e quindi seleziona Modello vuoto.

4- Inserisci il tuo codice.

5- Toccare Esegui il mio codice per eseguire il codice.

6- Nella parte anteriore di ogni riga, il risultato verrà memorizzato in un quadratino. Toccalo per rivelare il risultato.

7- Per scorrere lentamente il codice per tracciarlo, toccare il pulsante accanto a Esegui il mio codice.

Valore opzionale ed enum opzionale

Tipo di opzioni, che gestisce l'assenza di un valore. Gli optionals dicono "c'è un valore, ed è uguale a x" o "non c'è alcun valore".

Un optional è un tipo a sé stante, in realtà una delle nuove enumerazioni potenziate di Swift. Ha due possibili valori, `None` e `Some(T)` , dove T è un valore associato del tipo di dati corretto disponibile in Swift.

Diamo un'occhiata a questo pezzo di codice per esempio:

```
let x: String? = "Hello World"

if let y = x {
   print(y)
}
```

Infatti se aggiungi **`print(x.dynamicType)`** nel codice qui sopra la vedrai nella console:

```
Optional<String>
```

Stringa? è in realtà zucchero sintattico per Opzionale, e Opzionale è un tipo a sé stante.

Ecco una versione semplificata dell'intestazione di Opzionale, che puoi vedere facendo clic con il comando sulla parola Facoltativo nel codice da Xcode:

```
enum Optional<Wrapped> {

 /// The absence of a value.
 case none

 /// The presence of a value, stored as `Wrapped`.
 case some(Wrapped)
}
```

Opzionale è in realtà un enum, definito in relazione a un tipo generico Wrapped. Ha due casi:. *`.none`* per rappresentare l'assenza di un valore, e *`.some`* per rappresentare la presenza di un valore, che viene memorizzato come valore associato di tipo Wrapped.

Lasciarlo passare di nuovo: `String?` non è una `String` ma una `Optional<String>` . Il fatto che `Optional` sia un tipo indica che ha i propri metodi, ad esempio `map` e `flatMap` .

Leggi Iniziare con Swift Language online: https://riptutorial.com/it/swift/topic/202/iniziare-con-swift-language

Capitolo 2: Accesso a Swift

Osservazioni

`println` e `debugPrintln` dove rimosso in Swift 2.0.

fonti:

https://developer.apple.com/library/content/technotes/tn2347/_index.html
http://ericasadun.com/2015/05/22/swift-logging/

http://www.dotnetperls.com/print-swift

Examples

Debug Print

Debug Print mostra la rappresentazione dell'istanza più adatta per il debug.

```
print("Hello")
debugPrint("Hello")

let dict = ["foo": 1, "bar": 2]

print(dict)
debugPrint(dict)
```

I rendimenti

```
>>> Hello
>>> "Hello"
>>> [foo: 1, bar: 2]
>>> ["foo": 1, "bar": 2]
```

Questa informazione extra può essere molto importante, ad esempio:

```
let wordArray = ["foo", "bar", "food, bars"]

print(wordArray)
debugPrint(wordArray)
```

I rendimenti

```
>>> [foo, bar, food, bars]
>>> ["foo", "bar", "food, bars"]
```

Si noti come nel primo output ci siano 4 elementi nell'array rispetto a 3. Per ragioni come questa, è preferibile eseguire il debug per usare `debugPrint`

Aggiornamento di un debug di classi e stampa dei valori

```
struct Foo: Printable, DebugPrintable {
    var description: String {return "Clear description of the object"}
    var debugDescription: String {return "Helpful message for debugging"}
}

var foo = Foo()

print(foo)
debugPrint(foo)

>>> Clear description of the object
>>> Helpful message for debugging
```

cumulo di rifiuti

`dump` stampa il contenuto di un oggetto tramite riflessione (mirroring).

Vista dettagliata di un array:

```
let names = ["Joe", "Jane", "Jim", "Joyce"]
dump(names)
```

stampe:

▿ 4 elementi
- [0]: Joe
- [1]: Jane
- [2]: Jim
- [3]: Joyce

Per un dizionario:

```
let attributes = ["foo": 10, "bar": 33, "baz": 42]
dump(attributes)
```

stampe:

▿ 3 coppie chiave / valore
▿ [0]: (2 elementi)
- .0: bar
- .1: 33
▿ [1]: (2 elementi)

- .0: baz
- .1: 42
▿ [2]: (2 elementi)
- .0: pippo
- .1: 10

`dump` è dichiarato come `dump(_:name:indent:maxDepth:maxItems:)` .

Il primo parametro non ha etichetta.

Sono disponibili altri parametri, come il `name` per impostare un'etichetta per l'oggetto da ispezionare:

```
dump(attributes, name: "mirroring")
```

stampe:

▿ mirroring: 3 coppie chiave / valore
▿ [0]: (2 elementi)
- .0: bar
- .1: 33
▿ [1]: (2 elementi)
- .0: baz
- .1: 42
▿ [2]: (2 elementi)
- .0: pippo
- .1: 10

Puoi anche scegliere di stampare solo un determinato numero di elementi con `maxItems:` per analizzare l'oggetto fino a una certa profondità con `maxDepth:` e per modificare il rientro degli oggetti stampati con il `indent:`

print () vs dump ()

Molti di noi iniziano il debugging con la semplice `print()` . Diciamo che abbiamo una classe simile:

```
class Abc {
    let a = "aa"
    let b = "bb"
}
```

e abbiamo un'istanza di `Abc` in questo modo:

```
let abc = Abc()
```

Quando eseguiamo `print()` sulla variabile, l'output è

```
App.Abc
```

mentre output `dump()`

```
App.Abc #0
- a: "aa"
- b: "bb"
```

Come visto, `dump()` restituisce l'intera gerarchia delle classi, mentre `print()` restituisce semplicemente il nome della classe.

Pertanto, `dump()` è particolarmente utile per il debug dell'interfaccia utente

```
let view = UIView(frame: CGRect(x: 0, y: 0, width: 100, height: 100))
```

Con `dump(view)` otteniamo:

```
 - <UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>> #0
    - super: UIResponder
      - NSObject
```

Durante la `print(view)` otteniamo:

```
<UIView: 0x108a0cde0; frame = (0 0; 100 100); layer = <CALayer: 0x159340cb0>>
```

Ci sono più informazioni sulla classe con `dump()` , e quindi è più utile nel debug della classe stessa.

print vs NSLog

In swift possiamo usare entrambe `print()` funzioni `print()` e `NSLog()` per stampare qualcosa sulla console Xcode.

Ma ci sono molte differenze nelle funzioni `print()` e `NSLog()` , come ad esempio:

1 TimeStamp: `NSLog()` stamperà la data e l'ora con la stringa passata, ma `print()` non stamperà la data / ora.
per esempio

```
let array = [1, 2, 3, 4, 5]
print(array)
NSLog(array.description)
```

Produzione:

> [1, 2, 3, 4, 5]
> 2017-05-31 13: 14: 38.582 ProjetName [2286: 7473287] [1, 2, 3, 4, 5]

Stamperà anche **ProjectName** con timestamp.

2 Solo stringa: `NSLog()` accetta solo String come input, ma `print()` può stampare qualsiasi tipo di input passato ad esso.

per esempio

```
let array = [1, 2, 3, 4, 5]
print(array) //prints [1, 2, 3, 4, 5]
NSLog(array) //error: Cannot convert value of type [Int] to expected argument type 'String'
```

3 Prestazioni: la funzione `NSLog()` è molto **lenta** rispetto alla funzione `print()` .

4 Sincronizzazione: `NSLog()` gestisce l'utilizzo simultaneo dall'ambiente multi-threading e stampa l'output senza sovrapporlo. Ma `print()` non gestirà tali casi e non guasta mentre mostra l'output.

5 Console dispositivo: `NSLog()` output `NSLog()` sulla console del dispositivo, possiamo vedere questa uscita collegando il nostro dispositivo a Xcode. `print()` non stampa l'output sulla console del dispositivo.

Leggi Accesso a Swift online: https://riptutorial.com/it/swift/topic/3966/accesso-a-swift

Capitolo 3: Algoritmi con Swift

introduzione

Gli algoritmi sono una spina dorsale per il calcolo. Fare una scelta di quale algoritmo utilizzare in quale situazione distingue una media da un buon programmatore. Con questo in mente, ecco le definizioni e gli esempi di codice di alcuni degli algoritmi di base là fuori.

Examples

Inserimento Ordina

Insertion sort è uno degli algoritmi più basilari in informatica. L'ordinamento di inserimento classifica gli elementi iterando attraverso una raccolta e posiziona gli elementi in base al loro valore. Il set è diviso in metà ordinate e non ordinate e si ripete fino a quando tutti gli elementi sono ordinati. L'ordinamento di inserzione ha la complessità di O (n2). Puoi inserirlo in un'estensione, come nell'esempio seguente, oppure puoi creare un metodo per questo.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Ordinamento

Bubble Sort

Questo è un semplice algoritmo di ordinamento che passa ripetutamente nell'elenco per essere ordinato, confronta ogni coppia di elementi adiacenti e li scambia se sono nell'ordine sbagliato. Il passaggio attraverso la lista viene ripetuto fino a quando non sono necessari swap. Sebbene l'algoritmo sia semplice, è troppo lento e poco pratico per la maggior parte dei problemi. Ha complessità di O (n2), ma è considerato più lento dell'ordinamento di inserzione.

```
extension Array where Element: Comparable {

func bubbleSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryIndex in 0..<self.count {
        let passes = (output.count - 1) - primaryIndex

        //"half-open" range operator
        for secondaryIndex in 0..<passes {
            let key = output[secondaryIndex]

            //compare / swap positions
            if (key > output[secondaryIndex + 1]) {
                swap(&output[secondaryIndex], &output[secondaryIndex + 1])
            }
        }
    }

    return output
}

}
```

Inserimento sort

Insertion sort è uno degli algoritmi più basilari in informatica. L'ordinamento di inserimento classifica gli elementi iterando attraverso una raccolta e posiziona gli elementi in base al loro valore. Il set è diviso in metà ordinate e non ordinate e si ripete fino a quando tutti gli elementi sono ordinati. L'ordinamento di inserzione ha la complessità di O (n2). Puoi inserirlo in un'estensione, come nell'esempio seguente, oppure puoi creare un metodo per questo.

```
extension Array where Element: Comparable {

func insertionSort() -> Array<Element> {

    //check for trivial case
    guard self.count > 1 else {
        return self
    }
```

```
    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {

        let key = output[primaryindex]
        var secondaryindex = primaryindex

        while secondaryindex > -1 {
            if key < output[secondaryindex] {

                //move to correct position
                output.remove(at: secondaryindex + 1)
                output.insert(key, at: secondaryindex)
            }
            secondaryindex -= 1
        }
    }

    return output
}
}
```

Selezione sort

L'ordinamento della selezione è noto per la sua semplicità. Inizia con il primo elemento dell'array, salvandone il valore come valore minimo (o massimo, a seconda dell'ordine di ordinamento). Quindi ittera attraverso la matrice e sostituisce il valore minimo con qualsiasi altro valore inferiore a quello minimo che trova sulla strada. Questo valore minimo viene quindi posizionato nella parte più a sinistra della matrice e il processo viene ripetuto dall'indice successivo fino alla fine dell'array. L'ordinamento della selezione ha la complessità di O (n2) ma è considerato più lento della sua controparte: Selezione.

func selectionSort () -> Array {// controlla la guardia del caso triviale self.count> 1 else {return self}

```
//mutated copy
var output: Array<Element> = self

for primaryindex in 0..<output.count {
    var minimum = primaryindex
    var secondaryindex = primaryindex + 1

    while secondaryindex < output.count {
        //store lowest value as minimum
        if output[minimum] > output[secondaryindex] {
            minimum = secondaryindex
        }
        secondaryindex += 1
    }

    //swap minimum value with array iteration
    if primaryindex != minimum {
        swap(&output[primaryindex], &output[minimum])
    }
}

return output
```

```
}
```

Ordinamento rapido: tempo di complessità O (n log n)

Quicksort è uno degli algoritmi avanzati. Presenta una complessità temporale di O (n log n) e applica una strategia di divisione e conquista. Questa combinazione si traduce in prestazioni algoritmiche avanzate. Quicksort divide prima un grande array in due sotto-array più piccoli: gli elementi bassi e gli elementi alti. Quicksort può quindi ordinare in modo ricorsivo i sotto-array.

I passaggi sono:

Scegli un elemento, chiamato pivot, dall'array.

Riordinare la matrice in modo che tutti gli elementi con valori inferiori al pivot vengano prima del pivot, mentre tutti gli elementi con valori maggiori del pivot vengono dopo di essa (valori uguali possono andare in entrambi i modi). Dopo questo partizionamento, il perno si trova nella sua posizione finale. Questo è chiamato operazione di partizione.

Applicare ricorsivamente i passaggi precedenti al sotto-array di elementi con valori più piccoli e separatamente al sotto-array di elementi con valori maggiori.

funzione di muting quickSort () -> Array {

```
func qSort(start startIndex: Int, _ pivot: Int) {

    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self
}

mutating func qPartition(start startIndex: Int, _ pivot: Int) -> Int {

var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {

    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
    }
}
    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
```

```
}
```

Selezione sort

L'ordinamento della selezione è noto per la sua semplicità. Inizia con il primo elemento dell'array, salvandone il valore come valore minimo (o massimo, a seconda dell'ordine di ordinamento). Quindi ittera attraverso la matrice e sostituisce il valore minimo con qualsiasi altro valore inferiore a quello minimo che trova sulla strada. Questo valore minimo viene quindi posizionato nella parte più a sinistra della matrice e il processo viene ripetuto dall'indice successivo fino alla fine dell'array. L'ordinamento della selezione ha la complessità di O (n2) ma è considerato più lento della sua controparte: Selezione.

```
func selectionSort() -> Array<Element> {
    //check for trivial case
    guard self.count > 1 else {
        return self
    }

    //mutated copy
    var output: Array<Element> = self

    for primaryindex in 0..<output.count {
        var minimum = primaryindex
        var secondaryindex = primaryindex + 1

        while secondaryindex < output.count {
            //store lowest value as minimum
            if output[minimum] > output[secondaryindex] {
                minimum = secondaryindex
            }
            secondaryindex += 1
        }

        //swap minimum value with array iteration
        if primaryindex != minimum {
            swap(&output[primaryindex], &output[minimum])
        }
    }

    return output
}
```

Analisi asintotica

Dal momento che abbiamo diversi algoritmi tra cui scegliere, quando vogliamo ordinare un array, dobbiamo sapere quale farà il suo lavoro. Quindi abbiamo bisogno di un metodo per misurare la velocità e l'affidabilità di algoritm. È qui che entra in gioco l'analisi asintotica. L'analisi asintotica è il processo di descrizione dell'efficienza degli algoritmi al crescere della loro dimensione di input (n). Nell'informatica, i asintotici sono solitamente espressi in un formato comune noto come Notazione Big O.

- **Tempo lineare O (n)** : quando ogni elemento nell'array deve essere valutato affinché una funzione possa raggiungere il suo obiettivo, ciò significa che la funzione diventa meno

efficace con l'aumentare del numero di elementi. *Si dice che una funzione come questa funzioni in tempo lineare perché la sua velocità dipende dalla sua dimensione di input.*

- **Tempo polinomico O (n2)** : se la complessità di una funzione cresce esponenzialmente (ovvero che per n elementi di una complessità dell'array di una funzione è n al quadrato) quella funzione opera in tempo polinomico. Di solito sono funzioni con cicli annidati. Due loop nidificati determinano la complessità di O (n2), tre cicli nidificati determinano la complessità di O (n3), e così via ...
- **Tempo logaritmico O (log n):** la complessità delle funzioni temporali logaritmiche viene ridotta al minimo quando aumenta la dimensione dei suoi input (n). Questi sono il tipo di funzioni che ogni programmatore cerca.

Ordinamento rapido: tempo di complessità O (n log n)

Quicksort è uno degli algoritmi avanzati. Presenta una complessità temporale di O (n log n) e applica una strategia di divisione e conquista. Questa combinazione si traduce in prestazioni algoritmiche avanzate. Quicksort divide prima un grande array in due sotto-array più piccoli: gli elementi bassi e gli elementi alti. Quicksort può quindi ordinare in modo ricorsivo i sotto-array.

I passaggi sono:

1. Scegli un elemento, chiamato pivot, dall'array.
2. Riordinare la matrice in modo che tutti gli elementi con valori inferiori al pivot vengano prima del pivot, mentre tutti gli elementi con valori maggiori del pivot vengono dopo di essa (valori uguali possono andare in entrambi i modi). Dopo questo partizionamento, il perno si trova nella sua posizione finale. Questo è chiamato operazione di partizione.
3. Applicare ricorsivamente i passaggi precedenti al sotto-array di elementi con valori più piccoli e separatamente al sotto-array di elementi con valori maggiori.

```
mutating func quickSort() -> Array<Element> {

func qSort(start startIndex: Int, _ pivot: Int) {

    if (startIndex < pivot) {
        let iPivot = qPartition(start: startIndex, pivot)
        qSort(start: startIndex, iPivot - 1)
        qSort(start: iPivot + 1, pivot)
    }
}
qSort(start: 0, self.endIndex - 1)
return self
```

}

func mutating qPartition (start startIndex: Int, _ pivot: Int) -> Int {

```
var wallIndex: Int = startIndex

//compare range with pivot
for currentIndex in wallIndex..<pivot {
```

```
    if self[currentIndex] <= self[pivot] {
        if wallIndex != currentIndex {
            swap(&self[currentIndex], &self[wallIndex])
        }

        //advance wall
        wallIndex += 1
    }
}
```

```
    //move pivot to final position
    if wallIndex != pivot {
        swap(&self[wallIndex], &self[pivot])
    }
    return wallIndex
}
```

Grafico, Trie, Stack

Grafico

In informatica, un grafico è un tipo di dati astratti che intende implementare il grafico non orientato e i concetti di grafico diretto dalla matematica. Una struttura di dati del grafico consiste in un insieme finito (e possibilmente mutevole) di vertici o nodi o punti, insieme a un insieme di coppie non ordinate di questi vertici per un grafo non orientato o un insieme di coppie ordinate per un grafico diretto. Queste coppie sono note come bordi, archi o linee per un grafico non orientato e come frecce, bordi diretti, archi diretti o linee dirette per un grafico diretto. I vertici possono far parte della struttura del grafico o possono essere entità esterne rappresentate da indici o riferimenti interi. Una struttura dati del grafico può anche associare a ciascun bordo un valore di bordo, come un'etichetta simbolica o un attributo numerico (costo, capacità, lunghezza, ecc.). (Wikipedia, fonte)

```
//
//  GraphFactory.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 6/7/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


public class SwiftGraph {


    //declare a default directed graph canvas
    private var canvas: Array<Vertex>
    public var isDirected: Bool


    init() {
        canvas = Array<Vertex>()
```

```
        isDirected = true
    }


    //create a new vertex
    func addVertex(key: String) -> Vertex {


        //set the key
        let childVertex: Vertex = Vertex()
        childVertex.key = key


        //add the vertex to the graph canvas
        canvas.append(childVertex)


        return childVertex
    }



    //add edge to source vertex
    func addEdge(source: Vertex, neighbor: Vertex, weight: Int) {


        //create a new edge
        let newEdge = Edge()


        //establish the default properties
        newEdge.neighbor = neighbor
        newEdge.weight = weight
        source.neighbors.append(newEdge)


        print("The neighbor of vertex: \(source.key as String!) is \(neighbor.key as
String!)..")


        //check condition for an undirected graph
        if isDirected == false {


           //create a new reversed edge
           let reverseEdge = Edge()


           //establish the reversed properties
           reverseEdge.neighbor = source
           reverseEdge.weight = weight
           neighbor.neighbors.append(reverseEdge)

           print("The neighbor of vertex: \(neighbor.key as String!) is \(source.key as
String!)..")

        }


    }
```

```
/* reverse the sequence of paths given the shortest path.
   process analagous to reversing a linked list. */

func reversePath(_ head: Path!, source: Vertex) -> Path! {


    guard head != nil else {
        return head
    }

    //mutated copy
    var output = head


    var current: Path! = output
    var prev: Path!
    var next: Path!


    while(current != nil) {
        next = current.previous
        current.previous = prev
        prev = current
        current = next
    }


    //append the source path to the sequence
    let sourcePath: Path = Path()

    sourcePath.destination = source
    sourcePath.previous = prev
    sourcePath.total = nil

    output = sourcePath


    return output

}




//process Dijkstra's shortest path algorthim
func processDijkstra(_ source: Vertex, destination: Vertex) -> Path? {


    var frontier: Array<Path> = Array<Path>()
    var finalPaths: Array<Path> = Array<Path>()


    //use source edges to create the frontier
    for e in source.neighbors {

        let newPath: Path = Path()
```

```
        newPath.destination = e.neighbor
        newPath.previous = nil
        newPath.total = e.weight


        //add the new path to the frontier
        frontier.append(newPath)

    }


    //construct the best path
    var bestPath: Path = Path()


    while frontier.count != 0 {

        //support path changes using the greedy approach
        bestPath = Path()
        var pathIndex: Int = 0


        for x in 0..<frontier.count {

            let itemPath: Path = frontier[x]

            if  (bestPath.total == nil) || (itemPath.total < bestPath.total) {
                bestPath = itemPath
                pathIndex = x
            }

        }



        //enumerate the bestPath edges
        for e in bestPath.destination.neighbors {

            let newPath: Path = Path()

            newPath.destination = e.neighbor
            newPath.previous = bestPath
            newPath.total = bestPath.total + e.weight


            //add the new path to the frontier
            frontier.append(newPath)

        }


        //preserve the bestPath
        finalPaths.append(bestPath)


        //remove the bestPath from the frontier
        //frontier.removeAtIndex(pathIndex) - Swift2
        frontier.remove(at: pathIndex)
```

```
    } //end while



    //establish the shortest path as an optional
    var shortestPath: Path! = Path()


    for itemPath in finalPaths {

        if (itemPath.destination.key == destination.key) {

            if  (shortestPath.total == nil) || (itemPath.total < shortestPath.total) {
                shortestPath = itemPath
            }

        }

    }


    return shortestPath

}



///an optimized version of Dijkstra's shortest path algorthim
func processDijkstraWithHeap(_ source: Vertex, destination: Vertex) -> Path! {


    let frontier: PathHeap = PathHeap()
    let finalPaths: PathHeap = PathHeap()


    //use source edges to create the frontier
    for e in source.neighbors {

        let newPath: Path = Path()


        newPath.destination = e.neighbor
        newPath.previous = nil
        newPath.total = e.weight


        //add the new path to the frontier
        frontier.enQueue(newPath)

    }


    //construct the best path
    var bestPath: Path = Path()


    while frontier.count != 0 {

        //use the greedy approach to obtain the best path
        bestPath = Path()
```

```
            bestPath = frontier.peek()


            //enumerate the bestPath edges
            for e in bestPath.destination.neighbors {

                let newPath: Path = Path()

                newPath.destination = e.neighbor
                newPath.previous = bestPath
                newPath.total = bestPath.total + e.weight


                //add the new path to the frontier
                frontier.enQueue(newPath)

            }


            //preserve the bestPaths that match destination
            if (bestPath.destination.key == destination.key) {
                finalPaths.enQueue(bestPath)
            }


            //remove the bestPath from the frontier
            frontier.deQueue()


        } //end while



        //obtain the shortest path from the heap
        var shortestPath: Path! = Path()
        shortestPath = finalPaths.peek()


        return shortestPath

    }


    //MARK: traversal algorithms


    //bfs traversal with inout closure function
    func traverse(_ startingv: Vertex, formula: (_ node: inout Vertex) -> ()) {


        //establish a new queue
        let graphQueue: Queue<Vertex> = Queue<Vertex>()


        //queue a starting vertex
        graphQueue.enQueue(startingv)


        while !graphQueue.isEmpty() {

            //traverse the next queued vertex
```

```
        var vitem: Vertex = graphQueue.deQueue() as Vertex!


        //add unvisited vertices to the queue
        for e in vitem.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        /*
        notes: this demonstrates how to invoke a closure with an inout parameter.
        By passing by reference no return value is required.
        */

        //invoke formula
        formula(&vitem)


    } //end while


    print("graph traversal complete..")


}




//breadth first search
func traverse(_ startingv: Vertex) {


    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        let vitem = graphQueue.deQueue() as Vertex!

        guard vitem != nil else {
            return
        }

        //add unvisited vertices to the queue
        for e in vitem!.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }
```

```
        vitem!.visited = true
        print("traversed vertex: \(vitem!.key!)..")


    } //end while


    print("graph traversal complete..")


} //end function



//use bfs with trailing closure to update all values
func update(startingv: Vertex, formula:((Vertex) -> Bool)) {


    //establish a new queue
    let graphQueue: Queue<Vertex> = Queue<Vertex>()


    //queue a starting vertex
    graphQueue.enQueue(startingv)


    while !graphQueue.isEmpty() {

        //traverse the next queued vertex
        let vitem = graphQueue.deQueue() as Vertex!

        guard vitem != nil else {
            return
        }

        //add unvisited vertices to the queue
        for e in vitem!.neighbors {
            if e.neighbor.visited == false {
                print("adding vertex: \(e.neighbor.key!) to queue..")
                graphQueue.enQueue(e.neighbor)
            }
        }


        //apply formula..
        if formula(vitem!) == false {
            print("formula unable to update: \(vitem!.key)")
        }
        else {
            print("traversed vertex: \(vitem!.key!)..")
        }

        vitem!.visited = true


    } //end while


    print("graph traversal complete..")
```

```
    }




}
```

prova

Nell'informatica, un trie, chiamato anche albero digitale e talvolta albero radiale o albero prefisso (come possono essere ricercati dai prefissi), è un tipo di albero di ricerca: una struttura dati dell'albero ordinata che viene utilizzata per memorizzare un insieme dinamico o associativo array in cui le chiavi sono solitamente stringhe. (Wikipedia, fonte)

```
//
//  Trie.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 10/14/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


public class Trie {

    private var root: TrieNode!


    init(){
        root = TrieNode()
    }



    //builds a tree hierarchy of dictionary content
    func append(word keyword: String) {


        //trivial case
        guard keyword.length > 0 else {
            return
        }


        var current: TrieNode = root


        while keyword.length != current.level {

            var childToUse: TrieNode!
            let searchKey = keyword.substring(to: current.level + 1)


            //print("current has \(current.children.count) children..")
```

```
        //iterate through child nodes
        for child in current.children {

            if (child.key == searchKey) {
                childToUse = child
                break
            }

        }


        //new node
        if childToUse == nil {

            childToUse = TrieNode()
            childToUse.key = searchKey
            childToUse.level = current.level + 1
            current.children.append(childToUse)
        }


        current = childToUse


    } //end while


    //final end of word check
    if (keyword.length == current.level) {
        current.isFinal = true
        print("end of word reached!")
        return
    }



} //end function




//find words based on the prefix
func search(forWord keyword: String) -> Array<String>! {


    //trivial case
    guard keyword.length > 0 else {
        return nil
    }


    var current: TrieNode = root
    var wordList = Array<String>()


    while keyword.length != current.level {

        var childToUse: TrieNode!
        let searchKey = keyword.substring(to: current.level + 1)
```

```
            //print("looking for prefix: \(searchKey)..")


            //iterate through any child nodes
            for child in current.children {

                if (child.key == searchKey) {
                    childToUse = child
                    current = childToUse
                    break
                }

            }


            if childToUse == nil {
               return nil
            }


        } //end while



        //retrieve the keyword and any descendants
        if ((current.key == keyword) && (current.isFinal)) {
            wordList.append(current.key)
        }


        //include only children that are words
        for child in current.children {

            if (child.isFinal == true) {
                wordList.append(child.key)
            }

        }


        return wordList


    } //end function


}
```

(GitHub, fonte)

Pila

In informatica, uno stack è un tipo di dati astratto che funge da raccolta di elementi, con due operazioni principali: push, che aggiunge un elemento alla raccolta e pop, che rimuove l'ultimo elemento aggiunto che non è stato ancora rimosso. L'ordine in cui gli elementi escono da una pila dà origine al suo nome alternativo, LIFO (per ultimo dentro, prima uscita). Inoltre, un'operazione di

sbirciata può dare accesso alla parte superiore senza modificare lo stack. (Wikipedia, fonte)

Vedi le informazioni sulla licenza qui sotto e la fonte del codice originale su (github)

```
//
//  Stack.swift
//  SwiftStructures
//
//  Created by Wayne Bishop on 8/1/14.
//  Copyright (c) 2014 Arbutus Software Inc. All rights reserved.
//
import Foundation


class Stack<T> {

    private var top: Node<T>

    init() {
        top = Node<T>()
    }


    //the number of items - O(n)
    var count: Int {


        //return trivial case
        guard top.key != nil else {
          return 0
        }


        var current = top
        var x: Int = 1


        //cycle through list
        while current.next != nil {
            current = current.next!
            x += 1
        }

        return x

    }


    //add item to the stack
    func push(withKey key: T) {


        //return trivial case
        guard top.key != nil else {
            top.key = key
            return
        }


        //create new item
```

```
        let childToUse = Node<T>()
        childToUse.key = key


        //set new created item at top
        childToUse.next = top
        top = childToUse

    }


    //remove item from the stack
    func pop() {

        if self.count > 1 {
            top = top.next
        }
        else {
            top.key = nil
        }

    }


    //retrieve the top most item
    func peek() -> T! {


        //determine instance
        if let topitem = top.key {
            return topitem
        }

        else {
            return nil
        }

    }



    //check for value
    func isEmpty() -> Bool {

        if self.count == 0 {
            return true
        }

        else {
            return false
        }

    }


}
```

Leggi Algoritmi con Swift online: https://riptutorial.com/it/swift/topic/9116/algoritmi-con-swift

Capitolo 4: Archi e personaggi

Sintassi

- String.characters // Restituisce una matrice dei caratteri nella stringa
- String.characters.count // Restituisce il numero di caratteri
- String.utf8 // A String.UTF8View, restituisce i punti carattere UTF-8 nella stringa
- String.utf16 // A String.UTF16View, restituisce i punti carattere UTF-16 nella stringa
- String.unicodeScalars // A String.UnicodeScalarView, restituisce i punti carattere UTF-32 nella stringa
- String.isEmpty // Restituisce true se la stringa non contiene alcun testo
- String.hasPrefix (String) // Restituisce true se la stringa è preceduta dall'argomento
- String.hasSuffix (String) // Restituisce true se la stringa è suffissa con l'argomento
- String.startIndex // Restituisce l'indice che corrisponde al primo carattere nella stringa
- String.endIndex // Restituisce l'indice che corrisponde allo spot dopo l'ultimo carattere nella stringa
- String.components (separatedBy: String) // Restituisce una matrice contenente le sottostringhe separate dalla stringa di separazione specificata
- String.append (carattere) // Aggiunge il carattere (fornito come argomento) alla stringa

Osservazioni

Una `String` in Swift è una raccolta di caratteri e, per estensione, una raccolta di scalari Unicode. Poiché le stringhe Swift si basano su Unicode, possono essere qualsiasi valore scalare Unicode, incluse le lingue diverse dall'inglese e dagli emoji.

Poiché due scalari potrebbero combinarsi per formare un singolo carattere, il numero di scalari in una stringa non è necessariamente sempre uguale al numero di caratteri.

Per ulteriori informazioni su Stringhe, vedere The Swift Programming Language e String Structure Reference .

Per dettagli sull'implementazione, vedi "Swift String Design"

Examples

String and Character Literals

I valori letterali delle stringhe in Swift sono delimitati da virgolette doppie (`"`):

```
let greeting = "Hello!"  // greeting's type is String
```

I caratteri possono essere inizializzati da string letterali, purché il letterale contenga solo un grafo grafo:

```
let chr: Character = "H" // valid
let chr2: Character = "" // valid
let chr3: Character = "abc" // invalid - multiple grapheme clusters
```

Interpolazione a stringa

L'interpolazione delle stringhe consente di iniettare un'espressione direttamente in una stringa letterale. Questo può essere fatto con tutti i tipi di valori, comprese stringhe, numeri interi, numeri in virgola mobile e altro.

La sintassi è una barra rovesciata seguita da parentesi che racchiudono il valore: `\(value)` . Qualsiasi espressione valida può apparire tra parentesi, incluse le chiamate di funzione.

```
let number = 5
let interpolatedNumber = "\(number)"  // string is "5"
let fortyTwo = "\(6 * 7)"             // string is "42"

let example = "This post has \(number) view\(number == 1 ? "" : "s")"
// It will output "This post has 5 views" for the above example.
// If the variable number had the value 1, it would output "This post has 1 view" instead.
```

Per i tipi personalizzati, il comportamento predefinito dell'interpolazione delle stringhe è che `"\(myobj)"` è equivalente a `String(myobj)` , la stessa rappresentazione utilizzata da `print(myobj)` . È possibile personalizzare questo comportamento implementando il protocollo `CustomStringConvertible` per il proprio tipo.

3.0

Per Swift 3, in conformità con SE-0089 , `String.init<T>(_:)` è stato rinominato in `String.init<T>(describing:)` .

L'interpolazione stringa `"\(myobj)"` preferirà il nuovo `String.init<T: LosslessStringConvertible>(_:)` initializer, ma ritornerà a `init<T>(describing:)` se il valore non è `LosslessStringConvertible` .

Personaggi speciali

Alcuni personaggi richiedono una **sequenza di escape** speciale per utilizzarli in stringhe letterali:

Personaggio	Senso
`\0`	il carattere null
`\\`	un semplice backslash, `\`
`\t`	un carattere di tabulazione
`\v`	una scheda verticale

Personaggio	Senso
`\r`	un ritorno a capo
`\n`	un avanzamento riga ("newline")
`\"`	una doppia citazione, `"`
`\'`	una sola citazione, `'`
`\u{n}`	il codice Unicode point *n* (in esadecimale)

Esempio:

```
let message = "Then he said, \"I \u{1F496} you!\""

print(message) // Then he said, "I 💖 you!"
```

Stringhe concatenate

Concatena le stringhe con l'operatore `+` per produrre una nuova stringa:

```
let name = "John"
let surname = "Appleseed"
let fullName = name + " " + surname  // fullName is "John Appleseed"
```

Aggiungi a una stringa **mutabile** usando l' operatore di assegnazione composto `+=` o usando un metodo:

```
let str2 = "there"
var instruction = "look over"
instruction += " " + str2  // instruction is now "look over there"

var instruction = "look over"
instruction.append(" " + str2)  // instruction is now "look over there"
```

Aggiungi un singolo carattere a una stringa mutabile:

```
var greeting: String = "Hello"
let exclamationMark: Character = "!"
greeting.append(exclamationMark)
// produces a modified String (greeting) = "Hello!"
```

Aggiungi più caratteri a una stringa mutabile

```
var alphabet:String = "my ABCs: "
alphabet.append(contentsOf: (0x61...0x7A).map(UnicodeScalar.init)
                                         .map(Character.init) )
// produces a modified string (alphabet) = "my ABCs: abcdefghijklmnopqrstuvwxyz"
```

3.0

`appendContentsOf(_:)` è stato rinominato per `append(_:)` .

Unisci una sequenza di stringhe per formare una nuova stringa usando `joinWithSeparator(_:)` :

```
let words = ["apple", "orange", "banana"]
let str = words.joinWithSeparator(" & ")

print(str)   // "apple & orange & banana"
```

3.0

`joinWithSeparator(_:)` è stato rinominato come `joined(separator:)` .

Il `separator` è la stringa vuota per impostazione predefinita, quindi `["a", "b", "c"].joined() == "abc"` .

Esamina e confronta le stringhe

Controlla se una stringa è vuota:

```
if str.isEmpty {
    // do something if the string is empty
}

// If the string is empty, replace it with a fallback:
let result = str.isEmpty ? "fallback string" : str
```

Verifica se due stringhe sono uguali (nel senso dell'equivalenza canonica Unicode):

```
"abc" == "def"          // false
"abc" == "ABC"          // false
"abc" == "abc"          // true

// "LATIN SMALL LETTER A WITH ACUTE" == "LATIN SMALL LETTER A" + "COMBINING ACUTE ACCENT"
"\u{e1}" == "a\u{301}"  // true
```

Controlla se una stringa inizia / finisce con un'altra stringa:

```
"fortitude".hasPrefix("fort")      // true
"Swift Language".hasSuffix("age")  // true
```

Codifica e decomposizione delle stringhe

Una stringa Swift è composta da punti di codice Unicode . Può essere decomposto e codificato in diversi modi.

```
let str = "ที่①!"
```

Stringhe decomponenti

I `characters` una stringa sono grafi di grafi estesi Unicode:

```
Array(str.characters)  // ["ที่","", "①","!"]
```

`unicodeScalars` sono i punti di codice Unicode che costituiscono una stringa (si noti che ที่ è un `unicodeScalars` , ma 3 punti di codice - 3607, 3637, 3656 - quindi la lunghezza dell'array risultante non è la stessa dei `characters`):

```
str.unicodeScalars.map{ $0.value }  // [3607, 3637, 3656, 128076, 9312, 33]
```

È possibile codificare e decomporre le stringhe come UTF-8 (una sequenza di `UInt8` s) o UTF-16 (una sequenza di `UInt16` s):

```
Array(str.utf8)   // [224, 184, 151, 224, 184, 181, 224, 185, 136, 240, 159, 145, 140, 226,
145, 160, 33]
Array(str.utf16)  // [3607, 3637, 3656, 55357, 56396, 9312, 33]
```

Lunghezza delle corde e iterazione

I `characters` una stringa, `unicodeScalars` , `utf8` e `utf16` sono tutti i Collection s, quindi puoi ottenere il loro `count` e iterare su di essi:

```
// NOTE: These operations are NOT necessarily fast/cheap!

str.characters.count     // 4
str.unicodeScalars.count // 6
str.utf8.count           // 17
str.utf16.count          // 7
```

```
for c in str.characters { // ...
for u in str.unicodeScalars { // ...
for byte in str.utf8 { // ...
for byte in str.utf16 { // ...
```

Unicode

Impostazione dei valori

Usando direttamente Unicode

```
var str: String = "I want to visit , Москва, मुंबई, القاهرة, and . "
var character: Character = ""
```

Utilizzo di valori esadecimali

```
var str: String = "\u{61}\u{5927}\u{1F34E}\u{3C0}" // a
```

```
var character: Character = "\u{65}\u{301}" // é = "e" + accent mark
```

Si noti che Swift `Character` può essere composto da più punti di codice Unicode, ma sembra essere un singolo carattere. Questo è chiamato Grapheme Cluster esteso.

conversioni

String -> Hex

```
// Accesses views of different Unicode encodings of `str`
str.utf8
str.utf16
str.unicodeScalars // UTF-32
```

Esadecimale -> stringa

```
let value0: UInt8 = 0x61
let value1: UInt16 = 0x5927
let value2: UInt32 = 0x1F34E

let string0 = String(UnicodeScalar(value0)) // a
let string1 = String(UnicodeScalar(value1)) // 
let string2 = String(UnicodeScalar(value2)) // 

// convert hex array to String
let myHexArray = [0x43, 0x61, 0x74, 0x203C, 0x1F431] // an Int array
var myString = ""
for hexValue in myHexArray {
    myString.append(UnicodeScalar(hexValue))
}
print(myString) // Cat!!
```

Nota che per UTF-8 e UTF-16 la conversione non è sempre così facile perché cose come le emoji non possono essere codificate con un singolo valore UTF-16. Ci vuole una coppia surrogata.

Inversione di archi

2.2

```
let aString = "This is a test string."

// first, reverse the String's characters
let reversedCharacters = aString.characters.reverse()

// then convert back to a String with the String() initializer
let reversedString = String(reversedCharacters)

print(reversedString) // ".gnirts tset a si sihT"
```

3.0

```
let reversedCharacters = aString.characters.reversed()
```

```
let reversedString = String(reversedCharacters)
```

Stringhe maiuscole e minuscole

Per rendere tutti i caratteri in una stringa maiuscoli o minuscoli:

2.2

```
let text = "AaBbCc"
let uppercase = text.uppercaseString // "AABBCC"
let lowercase = text.lowercaseString // "aabbcc"
```

3.0

```
let text = "AaBbCc"
let uppercase = text.uppercased() // "AABBCC"
let lowercase = text.lowercased() // "aabbcc"
```

Controlla se String contiene caratteri da un Set definito

Lettere

3.0

```
let letters = CharacterSet.letters

let phrase = "Test case"
let range = phrase.rangeOfCharacter(from: letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
   print("letters not found")
}
```

2.2

```
let letters = NSCharacterSet.letterCharacterSet()

let phrase = "Test case"
let range = phrase.rangeOfCharacterFromSet(letters)

// range will be nil if no letters is found
if let test = range {
   print("letters found")
}
else {
  print("letters not found")
}
```

La nuova struttura `CharacterSet` che è anche collegata alla classe `NSCharacterSet` Objective-C definisce diversi set predefiniti come:

- `decimalDigits`
- `capitalizedLetters`
- `alphanumerics`
- `controlCharacters`
- `illegalCharacters`
- e altro ancora puoi trovarlo nel riferimento NSCharacterSet .

Puoi anche definire il tuo set di caratteri:

3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive) {
   print("yes")
}
else {
   print("no")
}
```

2.2

```
let charset = NSCharacterSet(charactersInString: "t")

if let _ = phrase.rangeOfCharacterFromSet(charset, options: .CaseInsensitiveSearch, range:
nil) {
   print("yes")
}
else {
    print("no")
}
```

Puoi anche includere la gamma:

3.0

```
let phrase = "Test case"
let charset = CharacterSet(charactersIn: "t")

if let _ = phrase.rangeOfCharacter(from: charset, options: .caseInsensitive, range:
phrase.startIndex..<phrase.endIndex)) {
   print("yes")
}
else {
   print("no")
}
```

Conta le occorrenze di un personaggio in una stringa

Dato una `String` e un `Character`

```
let text = "Hello World"
let char: Character = "o"
```

Possiamo contare il numero di volte in cui il `Character` appare nella `String` usando

```
let sensitiveCount = text.characters.filter { $0 == char }.count // case-sensitive
let insensitiveCount = text.lowercaseString.characters.filter { $0 ==
Character(String(char).lowercaseString) } // case-insensitive
```

Rimuovi i caratteri da una stringa non definita in Set

2.2

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
   return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars =
Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text, set: chars) // "SwiftComeOut"
```

3.0

```
func removeCharactersNotInSetFromText(text: String, set: Set<Character>) -> String {
  return String(text.characters.filter { set.contains( $0) })
}

let text = "Swift 3.0 Come Out"
var chars =
Set([Character]("abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLKMNOPQRSTUVWXYZ".characters))
let newText = removeCharactersNotInSetFromText(text: text, set: chars)
```

Formattare stringhe

Zeri leader

```
let number: Int = 7
let str1 = String(format: "%03d", number) // 007
let str2 = String(format: "%05d", number) // 00007
```

Numeri dopo decimale

```
let number: Float = 3.14159
let str1 = String(format: "%.2f", number) // 3.14
let str2 = String(format: "%.4f", number) // 3.1416 (rounded)
```

Da decimale a esadecimale

```
let number: Int = 13627
let str1 = String(format: "%2X", number) // 353B
let str2 = String(format: "%2x", number) // 353b (notice the lowercase b)
```

In alternativa si potrebbe usare l'inizializzatore specializzato che fa lo stesso:

```
let number: Int = 13627
let str1 = String(number, radix: 16, uppercase: true) //353B
let str2 = String(number, radix: 16) // 353b
```

Decimale a un numero con una radice arbitraria

```
let number: Int = 13627
let str1 = String(number, radix: 36) // aij
```

Radix è `Int` in `[2, 36]` .

Conversione di una stringa Swift in un tipo numerico

```
Int("123") // Returns 123 of Int type
Int("abcd") // Returns nil
Int("10") // Returns 10 of Int type
Int("10", radix: 2) // Returns 2 of Int type
Double("1.5") // Returns 1.5 of Double type
Double("abcd") // Returns nil
```

Si noti che ciò restituisce un valore `Optional` , che deve essere scartato di conseguenza prima di essere utilizzato.

Iterazione delle stringhe

3.0

```
let string = "My fantastic string"
var index = string.startIndex

while index != string.endIndex {
    print(string[index])
    index = index.successor()
}
```

Nota: `endIndex` è dopo la fine della stringa (cioè `string[string.endIndex]` è un errore, ma `string[string.startIndex]` va bene). Inoltre, in una stringa vuota (`""`), `string.startIndex == string.endIndex` è `true` . Assicurati di controllare le stringhe vuote, dal momento che non puoi chiamare `startIndex.successor()` su una stringa vuota.

3.0

In Swift 3, gli indici String non hanno più `successor()` , `predecessor()` , `advancedBy(_:)` , `advancedBy(_:limit:)` o `distanceTo(_:)` .

Invece, quelle operazioni vengono spostate nella raccolta, che ora è responsabile per incrementare e decrementare i suoi indici.

I metodi disponibili sono `.index(after:)` , `.index(before:)` e `.index(_:, offsetBy:)` .

```
let string = "My fantastic string"
var currentIndex = string.startIndex

while currentIndex != string.endIndex {
    print(string[currentIndex])
    currentIndex = string.index(after: currentIndex)
}
```

Nota: stiamo usando `currentIndex` come nome di variabile per evitare confusione con il metodo `.index` .

E, per esempio, se vuoi andare dall'altra parte:

3.0

```
var index:String.Index? = string.endIndex.predecessor()

while index != nil {
    print(string[index!])
    if index != string.startIndex {
        index = index.predecessor()
    }
    else {
        index = nil
    }
}
```

(Oppure potresti solo invertire la stringa per prima, ma se non hai bisogno di andare fino in fondo nella stringa probabilmente preferiresti un metodo come questo)

3.0

```
var currentIndex: String.Index? = string.index(before: string.endIndex)

while currentIndex != nil {
    print(string[currentIndex!])
    if currentIndex != string.startIndex {
        currentIndex = string.index(before: currentIndex!)
    }
    else {
        currentIndex = nil
    }
}
```

Nota, `Index` è un tipo di oggetto e non un `Int` . Non è possibile accedere a un carattere di stringa come segue:

```
let string = "My string"
string[2] // can't do this
string.characters[2] // and also can't do this
```

Ma puoi ottenere un indice specifico come segue:

3.0

```
index = string.startIndex.advanceBy(2)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 2)
```

E può andare indietro come questo:

3.0

```
index = string.endIndex.advancedBy(-2)
```

3.0

```
currentIndex = string.index(string.endIndex, offsetBy: -2)
```

Se è possibile superare i limiti della stringa o se si desidera specificare un limite, è possibile utilizzare:

3.0

```
index = string.startIndex.advanceBy(20, limit: string.endIndex)
```

3.0

```
currentIndex = string.index(string.startIndex, offsetBy: 20, limitedBy: string.endIndex)
```

In alternativa si può semplicemente scorrere i caratteri in una stringa, ma questo potrebbe essere meno utile a seconda del contesto:

```
for c in string.characters {
    print(c)
}
```

Rimuovi WhiteSpace iniziale e finale e NewLine

3.0

```
let someString = "  Swift Language  \n"
let trimmedString =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceAndNewlineCharacterSet())
// "Swift Language"
```

Metodo `stringByTrimmingCharactersInSet` restituisce una nuova stringa creata rimuovendo da entrambe le estremità dei caratteri String contenuti in un determinato set di caratteri.

Possiamo anche rimuovere solo spazi bianchi o newline.

Rimozione solo spazi bianchi:

```
let trimmedWhiteSpace =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.whitespaceCharacterSet())
// "Swift Language  \n"
```

Rimozione solo newline:

```
let trimmedNewLine =
someString.stringByTrimmingCharactersInSet(NSCharacterSet.newlineCharacterSet())
// "  Swift Language  "
```

3.0

```
let someString = "  Swift Language  \n"

let trimmedString = someString.trimmingCharacters(in: .whitespacesAndNewlines)
// "Swift Language"

let trimmedWhiteSpace = someString.trimmingCharacters(in: .whitespaces)
// "Swift Language  \n"

let trimmedNewLine = someString.trimmingCharacters(in: .newlines)
// "  Swift Language  "
```

Nota: tutti questi metodi appartengono a `Foundation`. Usa `import Foundation` se Foundation non è già stato importato tramite altre librerie come Cocoa o UIKit.

Converti stringhe da e verso Dati / NSData

Per convertire String in e da Data / NSData è necessario codificare questa stringa con una codifica specifica. Il più famoso è `UTF-8` che è una rappresentazione a 8 bit di caratteri Unicode, adatta per la trasmissione o l'archiviazione da parte di sistemi basati su ASCII. Ecco un elenco di tutte le `String Encodings` disponibili

`String` a `Data` / `NSData`

3.0

```
let data = string.data(using: .utf8)
```

2.2

```
let data = string.dataUsingEncoding(NSUTF8StringEncoding)
```

`Data` / `NSData` a `String`

3.0

```
let string = String(data: data, encoding: .utf8)
```

2.2

```
let string = String(data: data, encoding: NSUTF8StringEncoding)
```

Divisione di una stringa in una matrice

In Swift puoi facilmente separare una stringa in una matrice tagliandola in un determinato carattere:

3.0

```
let startDate = "23:51"

let startDateAsArray = startDate.components(separatedBy: ":") // ["23", "51"]`
```

2.2

```
let startDate = "23:51"

let startArray = startDate.componentsSeparatedByString(":") // ["23", "51"]`
```

O quando il separatore non è presente:

3.0

```
let myText = "MyText"

let myTextArray = myText.components(separatedBy: " ") // myTextArray is ["MyText"]
```

2.2

```
let myText = "MyText"

let myTextArray = myText.componentsSeparatedByString(" ") // myTextArray is ["MyText"]
```

Leggi Archi e personaggi online: https://riptutorial.com/it/swift/topic/320/archi-e-personaggi

Capitolo 5: Array

introduzione

La matrice è un tipo di raccolta ad accesso casuale ordinato. Le matrici sono uno dei tipi di dati più comunemente utilizzati in un'app. Usiamo il tipo Array per contenere elementi di un singolo tipo, il tipo di elemento dell'array. Un array può memorizzare qualsiasi tipo di elementi --- dagli interi alle stringhe alle classi.

Sintassi

- Array <Elemento> // Il tipo di una matrice con elementi di tipo Elemento
- [Elemento] // Zucchero sintattico per il tipo di una matrice con elementi di tipo Elemento
- [element0, element1, element2, ... elementN] // Un array letterale
- [Element] () // Crea una nuova matrice vuota di tipo [Element]
- Array (count: repeatValue :) // Crea un array di elementi `count` , ciascuno inizializzato su `repeatedValue`
- Matrice (_ :) // Crea una matrice da una sequenza arbitraria

Osservazioni

Gli array sono una *raccolta* di valori *ordinati* . I valori possono ripetersi ma *devono* essere dello stesso tipo.

Examples

Semantica del valore

La copia di un array copierà tutti gli elementi all'interno dell'array originale.

La modifica del nuovo array *non cambierà* la matrice originale.

```
var originalArray = ["Swift", "is", "great!"]
var newArray = originalArray
newArray[2] = "awesome!"
//originalArray = ["Swift", "is", "great!"]
//newArray = ["Swift", "is", "awesome!"]
```

Gli array copiati condividono lo stesso spazio in memoria dell'originale finché non vengono modificati. Di conseguenza, si verifica un impatto sulle prestazioni quando all'array copiato viene assegnato il proprio spazio in memoria mentre viene modificato per la prima volta.

Nozioni di base sugli array

`Array` è un tipo di raccolta ordinato nella libreria standard Swift. Fornisce O (1) accesso casuale e

riallocazione dinamica. La matrice è un tipo generico , quindi il tipo di valori che contiene sono noti al momento della compilazione.

Come `Array` è un tipo di valore , la sua mutabilità è definita dal fatto che sia annotata come `var` (mutabile) o `let` (immutabile).

Il tipo `[Int]` (che significa: un array contenente `Int` s) è zucchero sintattico per `Array<T>` .

Ulteriori informazioni sugli array in Swift Programming Language .

Matrici vuote

Le seguenti tre dichiarazioni sono equivalenti:

```
// A mutable array of Strings, initially empty.

var arrayOfStrings: [String] = []      // type annotation + array literal
var arrayOfStrings = [String]()        // invoking the [String] initializer
var arrayOfStrings = Array<String>()   // without syntactic sugar
```

Letterali di matrice

Un array letterale è scritto con parentesi quadre che circondano gli elementi separati da virgole:

```
// Create an immutable array of type [Int] containing 2, 4, and 7
let arrayOfInts = [2, 4, 7]
```

Il compilatore può solitamente dedurre il tipo di un array in base agli elementi nel letterale, ma le **annotazioni di tipo** esplicito possono ignorare il valore predefinito:

```
let arrayOfUInt8s: [UInt8] = [2, 4, 7]   // type annotation on the variable
let arrayOfUInt8s = [2, 4, 7] as [UInt8] // type annotation on the initializer expression
let arrayOfUInt8s = [2 as UInt8, 4, 7]   // explicit for one element, inferred for the others
```

Matrici con valori ripetuti

```
// An immutable array of type [String], containing ["Example", "Example", "Example"]
let arrayOfStrings = Array(repeating: "Example",count: 3)
```

Creare matrici da altre sequenze

```
let dictionary = ["foo" : 4, "bar" : 6]

// An immutable array of type [(String, Int)], containing [("bar", 6), ("foo", 4)]
```

```
let arrayOfKeyValuePairs = Array(dictionary)
```

Matrici multidimensionali

In Swift, una matrice multidimensionale viene creata nidificando gli array: una matrice bidimensionale di `Int` è `[[Int]]` (o `Array<Array<Int>>`).

```
let array2x3 = [
    [1, 2, 3],
    [4, 5, 6]
]
// array2x3[0][1] is 2, and array2x3[1][2] is 6.
```

Per creare una matrice multidimensionale di valori ripetuti, utilizzare le chiamate nidificate dell'inizializzatore dell'array:

```
var array3x4x5 = Array(repeating: Array(repeating: Array(repeating: 0,count: 5),count:
4),count: 3)
```

Accesso ai valori dell'array

I seguenti esempi useranno questo array per dimostrare l'accesso ai valori

```
var exampleArray:[Int] = [1,2,3,4,5]
//exampleArray = [1, 2, 3, 4, 5]
```

Per accedere a un valore in un indice noto, utilizzare la seguente sintassi:

```
let exampleOne = exampleArray[2]
//exampleOne = 3
```

Nota: il valore *all'indice due è il terzo valore* nella `Array` . `Array` usano un *indice a base zero, il* che significa che il primo elemento `Array` trova all'indice 0.

```
let value0 = exampleArray[0]
let value1 = exampleArray[1]
let value2 = exampleArray[2]
let value3 = exampleArray[3]
let value4 = exampleArray[4]
//value0 = 1
//value1 = 2
//value2 = 3
//value3 = 4
//value4 = 5
```

Accedi a un sottoinsieme di una `Array` usando il filtro:

```
var filteredArray = exampleArray.filter({ $0 < 4 })
//filteredArray = [1, 2, 3]
```

I filtri possono avere condizioni complesse come filtrare solo numeri pari:

```
var evenArray = exampleArray.filter({ $0 % 2 == 0 })
//evenArray = [2, 4]
```

È anche possibile restituire l'indice di un dato valore, restituendo `nil` se il valore non è stato trovato.

```
exampleArray.indexOf(3) // Optional(2)
```

Ci sono metodi per il primo, ultimo, massimo o minimo valore in una `Array` . Questi metodi restituiranno `nil` se l' `Array` è vuoto.

```
exampleArray.first // Optional(1)
exampleArray.last // Optional(5)
exampleArray.maxElement() // Optional(5)
exampleArray.minElement() // Optional(1)
```

Metodi utili

Determina se una matrice è vuota o restituisce la sua dimensione

```
var exampleArray = [1,2,3,4,5]
exampleArray.isEmpty //false
exampleArray.count //5
```

Invertire una matrice **Nota: il risultato non viene eseguito sull'array su cui viene chiamato il metodo e deve essere inserito nella propria variabile.**

```
exampleArray = exampleArray.reverse()
//exampleArray = [9, 8, 7, 6, 5, 3, 2]
```

Modifica dei valori in una matrice

Esistono diversi modi per aggiungere valori a un array

```
var exampleArray = [1,2,3,4,5]
exampleArray.append(6)
//exampleArray = [1, 2, 3, 4, 5, 6]
var sixOnwards = [7,8,9,10]
exampleArray += sixOnwards
//exampleArray = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

e rimuovere i valori da una matrice

```
exampleArray.removeAtIndex(3)
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9, 10]
exampleArray.removeLast()
//exampleArray = [1, 2, 3, 5, 6, 7, 8, 9]
exampleArray.removeFirst()
```

```
//exampleArray = [2, 3, 5, 6, 7, 8, 9]
```

Ordinamento di una matrice

```
var array = [3, 2, 1]
```

Creazione di un nuovo array ordinato

Dato che `Array` conforme a `SequenceType` , possiamo generare una nuova matrice di elementi ordinati usando un metodo di ordinamento incorporato.

2.1 2.2

In Swift 2, questo è fatto con il metodo `sort()` .

```
let sorted = array.sort()  // [1, 2, 3]
```

3.0

A partire da Swift 3, è stato rinominato in `sorted()` .

```
let sorted = array.sorted()  // [1, 2, 3]
```

Ordinamento di un array esistente sul posto

Poiché `Array` conforme a `MutableCollectionType` , possiamo ordinare i suoi elementi in posizione.

2.1 2.2

In Swift 2, questo viene fatto usando il metodo `sortInPlace()` .

```
array.sortInPlace() // [1, 2, 3]
```

3.0

A partire da Swift 3, è stato rinominato in `sort()` .

```
array.sort() // [1, 2, 3]
```

> Nota: per utilizzare i metodi precedenti, gli elementi devono essere conformi al protocollo `Comparable` .

Ordinamento di un array con un ordine personalizzato

Si può anche ordinare un array usando una chiusura per definire se un elemento debba essere ordinato prima di un altro - il che non è limitato agli array in cui gli elementi devono essere

`Comparable` . Ad esempio, non ha senso che un `Landmark` di `Landmark` sia `Comparable` , ma è comunque possibile ordinare una serie di punti di riferimento in base all'altezza o al nome.

```
struct Landmark {
    let name : String
    let metersTall : Int
}

var landmarks = [Landmark(name: "Empire State Building", metersTall: 443),
                 Landmark(name: "Eifell Tower", metersTall: 300),
                 Landmark(name: "The Shard", metersTall: 310)]
```

2.1 2.2

```
// sort landmarks by height (ascending)
landmarks.sortInPlace {$0.metersTall < $1.metersTall}

print(landmarks) // [Landmark(name: "Eifell Tower", metersTall: 300), Landmark(name: "The
Shard", metersTall: 310), Landmark(name: "Empire State Building", metersTall: 443)]

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sort {$0.name < $1.name}

print(alphabeticalLandmarks) // [Landmark(name: "Eifell Tower", metersTall: 300),
Landmark(name: "Empire State Building", metersTall: 443), Landmark(name: "The Shard",
metersTall: 310)]
```

3.0

```
// sort landmarks by height (ascending)
landmarks.sort {$0.metersTall < $1.metersTall}

// create new array of landmarks sorted by name
let alphabeticalLandmarks = landmarks.sorted {$0.name < $1.name}
```

> Nota: il confronto tra stringhe può produrre risultati imprevisti se le stringhe sono incoerenti, vedere Ordinamento di una matrice di stringhe .

Trasformare gli elementi di una matrice con la mappa (_ :)

Dato che `Array` conforme a `SequenceType` , possiamo usare `map(_:)` per trasformare un array di `A` in un array di `B` usando una chiusura di tipo `(A) throws -> B`

Ad esempio, potremmo usarlo per trasformare una matrice di `Int` in una serie di `String` s in questo modo:

```
let numbers = [1, 2, 3, 4, 5]
let words = numbers.map { String($0) }
print(words) // ["1", "2", "3", "4", "5"]
```

`map(_:)` scorrerà l'array, applicando la chiusura data a ciascun elemento. Il risultato di tale chiusura verrà utilizzato per popolare un nuovo array con gli elementi trasformati.

Poiché `String` ha un inizializzatore che riceve un `Int` possiamo usare anche questa sintassi più

chiara:

```
let words = numbers.map(String.init)
```

Una trasformazione di `map(_:)` non ha bisogno di cambiare il tipo di array - per esempio, potrebbe anche essere usato per moltiplicare un array di `Int` s per due:

```
let numbers = [1, 2, 3, 4, 5]
let numbersTimes2 = numbers.map {$0 * 2}
print(numbersTimes2) // [2, 4, 6, 8, 10]
```

Estrazione di valori di un determinato tipo da una matrice con flatMap (_ :)

Le `things` Array contiene valori di `Any` tipo.

```
let things: [Any] = [1, "Hello", 2, true, false, "World", 3]
```

Possiamo estrarre i valori di un determinato tipo e creare una nuova matrice di quel tipo specifico. Diciamo che vogliamo estrarre tutti gli `Int(s)` e inserirli in un `Int` Array in modo sicuro.

```
let numbers = things.flatMap { $0 as? Int }
```

Ora i `numbers` sono definiti come `[Int]` . La funzione `flatMap` tutti gli elementi `nil` e il risultato contiene quindi solo i seguenti valori:

```
[1, 2, 3]
```

Filtraggio di una matrice

È possibile utilizzare il metodo `filter(_:)` su `SequenceType` per creare un nuovo array contenente gli elementi della sequenza che soddisfano un determinato predicato, che può essere fornito come chiusura .

Ad esempio, filtrando i numeri pari da un `[Int]` :

```
let numbers = [22, 41, 23, 30]

let evenNumbers = numbers.filter { $0 % 2 == 0 }

print(evenNumbers)  // [22, 30]
```

Filtrare una `[Person]` , dove la loro età è inferiore a 30:

```
struct Person {
    var age : Int
}

let people = [Person(age: 22), Person(age: 41), Person(age: 23), Person(age: 30)]
```

```
let peopleYoungerThan30 = people.filter { $0.age < 30 }

print(peopleYoungerThan30) // [Person(age: 22), Person(age: 23)]
```

Filtraggio nullo da una trasformazione di array con flatMap (_ :)

È possibile utilizzare `flatMap(_:)` in modo simile alla `map(_:)` per creare un array applicando una trasformazione agli elementi di una sequenza.

```
extension SequenceType {
    public func flatMap<T>(@noescape transform: (Self.Generator.Element) throws -> T?)
rethrows -> [T]
}
```

La differenza con questa versione di `flatMap(_:)` è che si aspetta che la chiusura della trasformazione restituisca un valore opzionale `T?` per ciascuno degli elementi. Sarà quindi in grado di scartare in modo sicuro ciascuno di questi valori opzionali, filtrando `nil` - risultando in una matrice di `[T]` .

Ad esempio, puoi farlo per trasformare una `[String]` in una `[Int]` usando l' inizializzatore di `String` failable di `Int` , filtrando tutti gli elementi che non possono essere convertiti:

```
let strings = ["1", "foo", "3", "4", "bar", "6"]

let numbersThatCanBeConverted = strings.flatMap { Int($0) }

print(numbersThatCanBeConverted) // [1, 3, 4, 6]
```

Puoi anche usare la capacità di `flatMap(_:)` di filtrare `nil` per convertire semplicemente un array di optionals in una serie di non-optionals:

```
let optionalNumbers : [Int?] = [nil, 1, nil, 2, nil, 3]

let numbers = optionalNumbers.flatMap { $0 }

print(numbers) // [1, 2, 3]
```

Sottoscrizione di una matrice con un intervallo

Si può estrarre una serie di elementi consecutivi da una matrice usando un intervallo.

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let range = 2...4
let slice = words[range] // ["Bonjour", "Welcome", "Hi"]
```

La sottoscrizione di una matrice con intervallo restituisce un oggetto `ArraySlice` . È una sottosequenza della matrice.

Nel nostro esempio, abbiamo una matrice di stringhe, quindi torniamo ad `ArraySlice<String>` .

Sebbene ArraySlice sia conforme a `CollectionType` e possa essere utilizzato con `sort` , `filter` , ecc., Il suo scopo non è l'archiviazione a lungo termine ma i calcoli transitori: dovrebbe essere riconvertito in una matrice non appena hai finito di lavorare con essa.

Per questo, usa l'inizializzatore `Array()` :

```
let result = Array(slice)
```

Per riassumere in un semplice esempio senza passaggi intermedi:

```
let words = ["Hey", "Hello", "Bonjour", "Welcome", "Hi", "Hola"]
let selectedWords = Array(words[2...4]) // ["Bonjour", "Welcome", "Hi"]
```

Raggruppamento di valori di matrice

Se abbiamo una struttura come questa

```
struct Box {
    let name: String
    let thingsInside: Int
}
```

e una serie di `Box(es)`

```
let boxes = [
    Box(name: "Box 0", thingsInside: 1),
    Box(name: "Box 1", thingsInside: 2),
    Box(name: "Box 2", thingsInside: 3),
    Box(name: "Box 3", thingsInside: 1),
    Box(name: "Box 4", thingsInside: 2),
    Box(name: "Box 5", thingsInside: 3),
    Box(name: "Box 6", thingsInside: 1)
]
```

possiamo raggruppare le caselle in base alla proprietà `thingsInside` per ottenere un `Dictionary` cui la `key` è il numero di elementi e il valore è un array di caselle.

```
let grouped = boxes.reduce([Int:[Box]]()) { (res, box) -> [Int:[Box]] in
    var res = res
    res[box.thingsInside] = (res[box.thingsInside] ?? []) + [box]
    return res
}
```

Ora raggruppato è un `[Int:[Box]]` e ha il seguente contenuto

```
[
    2: [Box(name: "Box 1", thingsInside: 2), Box(name: "Box 4", thingsInside: 2)],
    3: [Box(name: "Box 2", thingsInside: 3), Box(name: "Box 5", thingsInside: 3)],
    1: [Box(name: "Box 0", thingsInside: 1), Box(name: "Box 3", thingsInside: 1), Box(name:
"Box 6", thingsInside: 1)]
]
```

Appiattimento del risultato di una trasformazione di matrice con flatMap (_ :)

Oltre a essere in grado di creare un array filtrando `nil` dagli elementi trasformati di una sequenza, esiste anche una versione di `flatMap(_:)` che si aspetta che la chiusura della trasformazione restituisca una sequenza `S`

```
extension SequenceType {
    public func flatMap<S : SequenceType>(transform: (Self.Generator.Element) throws -> S)
rethrows -> [S.Generator.Element]
}
```

Ogni sequenza dalla trasformazione sarà concatenata, risultante in una matrice contenente gli elementi combinati di ciascuna sequenza: `[S.Generator.Element]` .

Combinare i personaggi in una serie di stringhe

Ad esempio, possiamo usarlo per prendere una serie di stringhe prime e combinare i loro caratteri in un unico array:

```
let primes = ["2", "3", "5", "7", "11", "13", "17", "19"]
let allCharacters = primes.flatMap { $0.characters }
// => "["2", "3", "5", "7", "1", "1", "1", "3", "1", "7", "1", "9"]"
```

Rompendo l'esempio precedente in basso:

1. `primes` is a `[String]` (Dato che una matrice è una sequenza, possiamo chiamare `flatMap(_:)` su di essa).
2. La chiusura della trasformazione prende uno degli elementi di `primes` , una `String` (`Array<String>.Generator.Element`).
3. La chiusura quindi restituisce una sequenza di tipo `String.CharacterView` .
4. Il risultato è quindi un array contenente gli elementi combinati di tutte le sequenze da ciascuna delle chiamate di chiusura della trasformazione: `[String.CharacterView.Generator.Element]` .

Appiattimento di un array multidimensionale

Quando `flatMap(_:)` concatena le sequenze restituite dalle chiamate di chiusura della trasformazione, può essere utilizzato per appiattire un array multidimensionale, ad esempio un array 2D in un array 1D, un array 3D in un array 2D, ecc.

Questo può essere fatto semplicemente restituendo l'elemento dato `$0` (un array annidato) nella chiusura:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// A 1D array of type [Int]
let flattenedArray = array2D.flatMap { $0 }
```

```
print(flattenedArray) // [1, 3, 4, 6, 8, 10, 11]
```

Ordinamento di una matrice di stringhe

3.0

Il modo più semplice è usare `sorted()` :

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted()
print(sortedWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

o `sort()`

```
var mutableWords = ["Hello", "Bonjour", "Salute", "Ahola"]
mutableWords.sort()
print(mutableWords) // ["Ahola", "Bonjour", "Hello", "Salute"]
```

Puoi passare una chiusura come argomento per l'ordinamento:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted(isOrderedBefore: { $0 > $1 })
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

Sintassi alternativa con una chiusura finale:

```
let words = ["Hello", "Bonjour", "Salute", "Ahola"]
let sortedWords = words.sorted() { $0 > $1 }
print(sortedWords) // ["Salute", "Hello", "Bonjour", "Ahola"]
```

Ma ci saranno risultati inaspettati se gli elementi dell'array non sono coerenti:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let unexpected = words.sorted()
print(unexpected) // ["Hello", "Salute", "ahola", "bonjour"]
```

Per risolvere questo problema, ordinare in una versione minuscola degli elementi:

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.lowercased() < $1.lowercased() }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

Oppure `import Foundation` e usa i metodi di confronto di NSString come `caseInsensitiveCompare` :

```
let words = ["Hello", "bonjour", "Salute", "ahola"]
let sortedWords = words.sorted { $0.caseInsensitiveCompare($1) == .orderedAscending }
print(sortedWords) // ["ahola", "bonjour", "Hello", "Salute"]
```

In alternativa, usa `localizedCaseInsensitiveCompare` , che può gestire i segni diacritici.

Per ordinare correttamente le stringhe in base al valore *numerico* che contengono, utilizzare il `compare` con l'opzione `.numeric` :

```
let files = ["File-42.txt", "File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt"]
let sortedFiles = files.sorted() { $0.compare($1, options: .numeric) == .orderedAscending }
print(sortedFiles) // ["File-01.txt", "File-5.txt", "File-007.txt", "File-10.txt", "File-
42.txt"]
```

Appiattire con parsimonia una matrice multidimensionale con appiattisci ()

Possiamo usare `flatten()` per ridurre pigramente la nidificazione di una sequenza multidimensionale.

Ad esempio, lazy appiattimento di un array 2D in un array 1D:

```
// A 2D array of type [[Int]]
let array2D = [[1, 3], [4], [6, 8, 10], [11]]

// A FlattenBidirectionalCollection<[[Int]]>
let lazilyFlattenedArray = array2D.flatten()

print(lazilyFlattenedArray.contains(4)) // true
```

Nell'esempio sopra, `flatten()` restituirà un `FlattenBidirectionalCollection` , che applicherà pigramente l'appiattimento dell'array. Pertanto `contains(_:)` richiederà solo i primi due array annidati di `array2D` per essere appiattito - in quanto cortocircuiterà nel trovare l'elemento desiderato.

Combinare gli elementi di una matrice con riduci (_: combina :)

`reduce(_:combine:)` può essere utilizzato per combinare gli elementi di una sequenza in un singolo valore. Richiede un valore iniziale per il risultato e una chiusura da applicare a ciascun elemento, che restituirà il nuovo valore accumulato.

Ad esempio, possiamo usarlo per sommare una matrice di numeri:

```
let numbers = [2, 5, 7, 8, 10, 4]

let sum = numbers.reduce(0) {accumulator, element in
    return accumulator + element
}

print(sum) // 36
```

Stiamo passando `0` al valore iniziale, poiché questo è il valore iniziale logico per una sommatoria. Se passassimo in un valore di `N` , la `sum` risultante sarebbe `N + 36` . La chiusura passata per `reduce` ha due argomenti. `accumulator` è il valore accumulato corrente, a cui viene assegnato il valore che la chiusura restituisce ad ogni iterazione. `element` è l'elemento corrente nell'iterazione.

Come in questo esempio, stiamo passando una `(Int, Int) -> Int` chiusura per `reduce` , che sta semplicemente emettendo l'aggiunta dei due input - possiamo effettivamente passare

direttamente nell'operatore `+` , poiché gli operatori sono funzioni in Swift:

```
let sum = numbers.reduce(0, combine: +)
```

Rimozione di un elemento da un array senza conoscere il suo indice

In generale, se vogliamo rimuovere un elemento da un array, dobbiamo conoscere il suo indice in modo che possiamo rimuoverlo facilmente usando `remove(at:)` function.

Ma cosa succede se non conosciamo l'indice, ma sappiamo il valore dell'elemento da rimuovere!

Quindi ecco la semplice estensione di un array che ci permetterà di rimuovere facilmente un elemento dalla matrice senza sapere che è indice:

Swift3

```
extension Array where Element: Equatable {

    mutating func remove(_ element: Element) {
        _ = index(of: element).flatMap {
            self.remove(at: $0)
        }
    }
}
```

per esempio

```
    var array = ["abc", "lmn", "pqr", "stu", "xyz"]
    array.remove("lmn")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]

    array.remove("nonexistent")
    print("\(array)")    //["abc", "pqr", "stu", "xyz"]
    //if provided element value is not present, then it will do nothing!
```

Anche se, per errore, abbiamo fatto qualcosa di simile: `array.remove(25)` ovvero abbiamo fornito un valore con un tipo di dati diverso, il compilatore genererà un errore che menziona:
`cannot convert value to expected argument type`

Trovare l'elemento minimo o massimo di una matrice

2.1 2.2

È possibile utilizzare i `minElement()` e `maxElement()` per trovare l'elemento minimo o massimo in una determinata sequenza. Ad esempio, con una serie di numeri:

```
let numbers = [2, 6, 1, 25, 13, 7, 9]

let minimumNumber = numbers.minElement() // Optional(1)
let maximumNumber = numbers.maxElement() // Optional(25)
```

3.0

A partire da Swift 3, i metodi sono stati rinominati `min()` e `max()` rispettivamente:

```
let minimumNumber = numbers.min() // Optional(1)
let maximumNumber = numbers.max() // Optional(25)
```

I valori restituiti da questi metodi sono Opzionali per riflettere il fatto che l'array potrebbe essere vuoto - se lo è, verrà restituito `nil` .

> Nota: i metodi precedenti richiedono che gli elementi siano conformi al protocollo `Comparable` .

Trovare l'elemento minimo o massimo con un ordine personalizzato

È inoltre possibile utilizzare i metodi precedenti con una chiusura personalizzata, definendo se un elemento debba essere ordinato prima di un altro, consentendo di trovare l'elemento minimo o massimo in un array in cui gli elementi non sono necessariamente `Comparable` .

Ad esempio, con una matrice di vettori:

```
struct Vector2 {
    let dx : Double
    let dy : Double

    var magnitude : Double {return sqrt(dx*dx+dy*dy)}
}

let vectors = [Vector2(dx: 3, dy: 2), Vector2(dx: 1, dy: 1), Vector2(dx: 2, dy: 2)]
```

2.1 2.2

```
// Vector2(dx: 1.0, dy: 1.0)
let lowestMagnitudeVec2 = vectors.minElement { $0.magnitude < $1.magnitude }

// Vector2(dx: 3.0, dy: 2.0)
let highestMagnitudeVec2 = vectors.maxElement { $0.magnitude < $1.magnitude }
```

3.0

```
let lowestMagnitudeVec2 = vectors.min { $0.magnitude < $1.magnitude }
let highestMagnitudeVec2 = vectors.max { $0.magnitude < $1.magnitude }
```

Accesso sicuro agli indici

Aggiungendo la seguente estensione agli indici di array è possibile accedere senza sapere se l'indice è all'interno dei limiti.

```
extension Array {
    subscript (safe index: Int) -> Element? {
        return indices ~= index ? self[index] : nil
```

```
    }
}
```

esempio:

```
if let thirdValue = array[safe: 2] {
    print(thirdValue)
}
```

Confronto di 2 matrici con zip

La funzione `zip` accetta 2 parametri di tipo `SequenceType` e restituisce un `Zip2Sequence` cui ogni elemento contiene un valore dalla prima sequenza e uno dalla seconda sequenza.

Esempio

```
let nums = [1, 2, 3]
let animals = ["Dog", "Cat", "Tiger"]
let numsAndAnimals = zip(nums, animals)
```

nomsAndAnimals ora contiene i seguenti valori

sequence1	sequence1
1	"Dog"
2	"Cat"
3	"Tiger"

Ciò è utile quando si desidera eseguire una sorta di confronto tra l'n-esimo elemento di ciascuna matrice.

Esempio

Dato 2 matrici di `Int(s)`

```
let list0 = [0, 2, 4]
let list1 = [0, 4, 8]
```

si desidera verificare se ciascun valore in `list1` è il doppio del valore correlato in `list0` .

```
let list1HasDoubleOfList0 = !zip(list0, list1).filter { $0 != (2 * $1)}.isEmpty
```

Leggi Array online: https://riptutorial.com/it/swift/topic/284/array

Capitolo 6: blocchi

introduzione

Da Swift Documentarion

Si dice che una chiusura sfugge ad una funzione quando la chiusura viene passata come argomento alla funzione, ma viene richiamata dopo il ritorno della funzione. Quando dichiari una funzione che accetta una chiusura come uno dei suoi parametri, puoi scrivere @escaping prima del tipo del parametro per indicare che la chiusura può uscire.

Examples

Chiusura senza fuga

In Swift 1 e 2, i parametri di chiusura erano in fuga per impostazione predefinita. Se sapessi che la tua chiusura non sfugge al corpo della funzione, puoi contrassegnare il parametro con l'attributo @noescape.

In Swift 3, è il contrario: i parametri di chiusura non sfuggono di default. Se si desidera che la funzione sfugga, è necessario contrassegnarla con l'attributo @escaping.

```
class ClassOne {
  // @noescape is applied here as default
  func methodOne(completion: () -> Void) {
    //
  }
}

class ClassTwo {
  let obj = ClassOne()
  var greeting = "Hello, World!"

  func methodTwo() {
    obj.methodOne() {
      // self.greeting is required
      print(greeting)
    }
  }
}
```

Chiusura di fuga

Da Swift Documentarion

> @escaping
>
> Applicare questo attributo al tipo di un parametro in una dichiarazione di metodo o funzione per indicare che il valore del parametro può essere memorizzato per

l'esecuzione successiva. Ciò significa che il valore può superare la durata della chiamata. I parametri del tipo di funzione con l'attributo del tipo di escape richiedono l'uso esplicito di self. per proprietà o metodi.

```
class ClassThree {

    var closure: (() -> ())?

    func doSomething(completion: @escaping () -> ()) {
        closure = finishBlock
    }
}
```

Nell'esempio precedente il blocco di completamento viene salvato in chiusura e vivrà letteralmente oltre la chiamata di funzione. Quindi il compilatore costringerà a contrassegnare il blocco di completamento come @escaping.

Leggi blocchi online: https://riptutorial.com/it/swift/topic/8623/blocchi

Capitolo 7: booleani

Examples

Cos'è Bool?

Bool è un tipo booleano con due valori possibili: `true` e `false` .

```
let aTrueBool = true
let aFalseBool = false
```

Le bols sono usate nelle dichiarazioni del flusso di controllo come condizioni. L' istruzione `if` usa una condizione booleana per determinare quale blocco di codice eseguire:

```
func test(_ someBoolean: Bool) {
    if someBoolean {
        print("IT'S TRUE!")
    }
    else {
        print("IT'S FALSE!")
    }
}
test(aTrueBool)  // prints "IT'S TRUE!"
```

Annulla un Bool con il prefisso! operatore

Il prefisso `!` l'operatore restituisce la **negazione logica** del suo argomento. Cioè `!true` restituisce `false` , e `!false` restituisce `true` .

```
print(!true)  // prints "false"
print(!false) // prints "true"

func test(_ someBoolean: Bool) {
    if !someBoolean {
        print("someBoolean is false")
    }
}
```

Operatori logici booleani

L'operatore OR (||) restituisce true se uno dei suoi due operandi restituisce true, altrimenti restituisce false. Ad esempio, il codice seguente restituisce true perché almeno una delle espressioni su entrambi i lati dell'operatore OR è vera:

```
if (10 < 20) || (20 < 10) {
    print("Expression is true")
}
```

L'operatore AND (&&) restituisce true solo se entrambi gli operandi valgono come veri. L'esempio

seguente restituirà false perché solo una delle due espressioni di operando restituisce true:

```
if (10 < 20) && (20 < 10) {
    print("Expression is true")
}
```

L'operatore XOR (^) restituisce true se uno e solo uno dei due operandi restituisce true. Ad esempio, il codice seguente restituirà true poiché solo un operatore valuta essere vero:

```
if (10 < 20) ^ (20 < 10) {
    print("Expression is true")
}
```

Booleans e Inline Conditionals

Un modo pulito per gestire i booleani è l'utilizzo di un condizionale in linea con l'a? b: c operatore ternario, che fa parte delle operazioni di base di Swift.

Il condizionale in linea è costituito da 3 componenti:

```
question ? answerIfTrue : answerIfFalse
```

dove domanda è un valore booleano che viene valutato e answerIfTrue è il valore restituito se la domanda è vera e answerIfFalse è il valore restituito se la domanda è falsa.

L'espressione sopra è la stessa di:

```
if question {
    answerIfTrue
} else {
    answerIfFalse
}
```

Con condizionali in linea si restituisce un valore basato su un valore booleano:

```
func isTurtle(_ value: Bool) {
    let color = value ? "green" : "red"
    print("The animal is \(color)")
}

isTurtle(true) // outputs 'The animal is green'
isTurtle(false) // outputs 'The animal is red'
```

Puoi anche chiamare metodi basati su un valore booleano:

```
func actionDark() {
    print("Welcome to the dark side")
}

func actionJedi() {
    print("Welcome to the Jedi order")
```

```
}

func welcome(_ isJedi: Bool) {
    isJedi ? actionJedi() : actionDark()
}

welcome(true) // outputs 'Welcome to the Jedi order'
welcome(false) // outputs 'Welcome to the dark side'
```

I condizionali in linea consentono valutazioni booleane a una sola riga

Leggi booleani online: https://riptutorial.com/it/swift/topic/735/booleani

Capitolo 8: chiusure

Sintassi

- var closureVar: (<parametri>) -> (<returnType>) // Come variabile o tipo di proprietà
- typealias ClosureType = (<parametri>) -> (<returnType>)
- {[<captureList>] (<parametri>) <throws-ness> -> <returnType> in <istruzioni>} // Completa sintassi di chiusura

Osservazioni

Per ulteriori informazioni sulle chiusure Swift, consultare la documentazione di Apple .

Examples

Nozioni di base sulla chiusura

Le chiusure (note anche come **blocchi** o **lambda**) sono pezzi di codice che possono essere memorizzati e passati all'interno del programma.

```
let sayHi = { print("Hello") }
// The type of sayHi is "() -> ()", aka "() -> Void"

sayHi()  // prints "Hello"
```

Come altre funzioni, le chiusure possono accettare argomenti e restituire risultati o generare errori :

```
let addInts = { (x: Int, y: Int) -> Int in
    return x + y
}
// The type of addInts is "(Int, Int) -> Int"

let result = addInts(1, 2)  // result is 3

let divideInts = { (x: Int, y: Int) throws -> Int in
    if y == 0 {
        throw MyErrors.DivisionByZero
    }
    return x / y
}
// The type of divideInts is "(Int, Int) throws -> Int"
```

Le chiusure possono **acquisire** valori dal loro ambito:

```
// This function returns another function which returns an integer
func makeProducer(x: Int) -> (() -> Int) {
    let closure = { x }  // x is captured by the closure
```

```
    return closure
}

// These two function calls use the exact same code,
// but each closure has captured different values.
let three = makeProducer(3)
let four = makeProducer(4)
three()  // returns 3
four()  // returns 4
```

Le chiusure possono essere trasferite direttamente nelle funzioni:

```
let squares = (1...10).map({ $0 * $0 })  // returns [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
let squares = (1...10).map { $0 * $0 }

NSURLSession.sharedSession().dataTaskWithURL(myURL,
    completionHandler: { (data: NSData?, response: NSURLResponse?, error: NSError?) in
        if let data = data {
            print("Request succeeded, data: \(data)")
        } else {
            print("Request failed: \(error)")
        }
    }).resume()
```

Variazioni di sintassi

La sintassi di chiusura di base è

`{ [` *capture list* `] (` *parametri* `)` *throws-ness* `->` *restituisce type* `in` *body* `}` .

Molte di queste parti possono essere omesse, quindi ci sono diversi modi equivalenti per scrivere semplici chiusure:

```
let addOne = { [] (x: Int) -> Int in return x + 1 }
let addOne = { [] (x: Int) -> Int in x + 1 }
let addOne = { (x: Int) -> Int in x + 1 }
let addOne = { x -> Int in x + 1 }
let addOne = { x in x + 1 }
let addOne = { $0 + 1 }

let addOneOrThrow = { [] (x: Int) throws -> Int in return x + 1 }
let addOneOrThrow = { [] (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { (x: Int) throws -> Int in x + 1 }
let addOneOrThrow = { x throws -> Int in x + 1 }
let addOneOrThrow = { x throws in x + 1 }
```

- La lista di cattura può essere omessa se è vuota.
- I parametri non hanno bisogno di annotazioni di tipo se i loro tipi possono essere dedotti.
- Il tipo di reso non ha bisogno di essere specificato se può essere dedotto.
- I parametri non devono essere nominati; invece possono essere indicati con `$0` , `$1` , `$2` , ecc.
- Se la chiusura contiene una singola espressione, il cui valore deve essere restituito, la parola chiave `return` può essere omessa.
- Se la chiusura è dedotta per lanciare un errore, è scritta in un contesto che si aspetta una chiusura di lancio, o non lancia un errore, i `throws` possono essere omessi.

```
// The closure's type is unknown, so we have to specify the type of x and y.
// The output type is inferred to be Int, because the + operator for Ints returns Int.
let addInts = { (x: Int, y: Int) in x + y }

// The closure's type is specified, so we can omit the parameters' type annotations.
let addInts: (Int, Int) -> Int = { x, y in x + y }
let addInts: (Int, Int) -> Int = { $0 + $1 }
```

Passando le chiusure in funzioni

Le funzioni possono accettare chiusure (o altre funzioni) come parametri:

```
func foo(value: Double, block: () -> Void) { ... }
func foo(value: Double, block: Int -> Int) { ... }
func foo(value: Double, block: (Int, Int) -> String) { ... }
```

Sintassi di chiusura finale

Se l'ultimo parametro di una funzione è una chiusura, le parentesi di chiusura `{` / `}` possono essere scritte **dopo** il richiamo della funzione:

```
foo(3.5, block: { print("Hello") })

foo(3.5) { print("Hello") }

dispatch_async(dispatch_get_main_queue(), {
    print("Hello from the main queue")
})

dispatch_async(dispatch_get_main_queue()) {
    print("Hello from the main queue")
}
```

Se l'argomento di una funzione è solo una chiusura, puoi anche omettere la coppia di parentesi `()` quando la si chiama con la sintassi della chiusura finale:

```
func bar(block: () -> Void) { ... }
```

```
bar() { print("Hello") }

bar { print("Hello") }
```

Parametri `@noescape`

I parametri di chiusura contrassegnati con `@noescape` sono garantiti per l'esecuzione prima che la chiamata della funzione ritorni, quindi utilizzando `self.` non è richiesto all'interno del corpo di chiusura:

```
func executeNow(@noescape block: () -> Void) {
```

```
    // Since `block` is @noescape, it's illegal to store it to an external variable.
    // We can only call it right here.
    block()
}

func executeLater(block: () -> Void) {
    dispatch_async(dispatch_get_main_queue()) {
        // Some time in the future...
        block()
    }
}
```

```
class MyClass {
    var x = 0
    func showExamples() {
        // error: reference to property 'x' in closure requires explicit 'self.' to make
capture semantics explicit
        executeLater { x = 1 }

        executeLater { self.x = 2 }  // ok, the closure explicitly captures self

        // Here "self." is not required, because executeNow() takes a @noescape block.
        executeNow { x = 3 }

        // Again, self. is not required, because map() uses @noescape.
        [1, 2, 3].map { $0 + x }
    }
}
```

Swift 3 note:

Nota che in Swift 3 non contrassegni più blocchi come @noescape. I blocchi ora **non** escaping per impostazione predefinita. In Swift 3, invece di contrassegnare una chiusura come non-escape, contrassegni un parametro di funzione che è una chiusura di escape come escaping usando la parola chiave "@escaping".

`throws` e `rethrows`

Le chiusure, come altre funzioni, possono generare errori :

```
func executeNowOrIgnoreError(block: () throws -> Void) {
    do {
        try block()
    } catch {
        print("error: \(error)")
    }
}
```

La funzione può, naturalmente, passare l'errore al suo chiamante:

```
func executeNowOrThrow(block: () throws -> Void) throws {
    try block()
}
```

Tuttavia, se il blocco inoltrato *non viene* lanciato, il chiamante è ancora bloccato con una funzione di lancio:

```
// It's annoying that this requires "try", because "print()" can't throw!
try executeNowOrThrow { print("Just printing, no errors here!") }
```

La soluzione è **`rethrows`** , che designa che la funzione può lanciare solo **se il suo parametro di chiusura genera** :

```
func executeNowOrRethrow(block: () throws -> Void) rethrows {
    try block()
}

// "try" is not required here, because the block can't throw an error.
executeNowOrRethrow { print("No errors are thrown from this closure") }

// This block can throw an error, so "try" is required.
try executeNowOrRethrow { throw MyError.Example }
```

Molte funzioni di libreria standard utilizzano i `rethrows` , inclusi `map()` , `filter()` e `indexOf()` .

Cattura, riferimenti forti / deboli e conserva i cicli

```
class MyClass {
    func sayHi() { print("Hello") }
    deinit { print("Goodbye") }
}
```

Quando una chiusura acquisisce un tipo di riferimento (un'istanza di classe), mantiene un riferimento forte per impostazione predefinita:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a strong reference to `obj`: the object will be kept alive
    // as long as the closure itself is alive.
    closure = { obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope
closure()  // The object is still alive; prints "Hello"
```

L' **elenco di cattura** della chiusura può essere utilizzato per specificare un riferimento debole o non associato:

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures a weak reference to `obj`: the closure will not keep the object alive;
    // the object becomes optional inside the closure.
    closure = { [weak obj] in obj?.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // `obj` is nil from inside the closure; this does not print anything.
```

```
let closure: () -> Void
do {
    let obj = MyClass()
    // Captures an unowned reference to `obj`: the closure will not keep the object alive;
    // the object is always assumed to be accessible while the closure is alive.
    closure = { [unowned obj] in obj.sayHi() }
    closure()  // The object is still alive; prints "Hello"
} // obj goes out of scope and is deallocated; prints "Goodbye"
closure()  // crash! obj is being accessed after it's deallocated.
```

Per ulteriori informazioni, consultare l'argomento Gestione della memoria e la sezione Conteggio riferimento automatico di Swift Programming Language.

Conservare i cicli

Se un oggetto tiene su una chiusura, che contiene anche un forte riferimento all'oggetto, questo è un **ciclo di conservazione** . A meno che il ciclo non sia interrotto, la memoria che memorizza l'oggetto e la chiusura sarà trapelata (mai bonificata).

```
class Game {
    var score = 0
    let controller: GCController
    init(controller: GCController) {
        self.controller = controller

        // BAD: the block captures self strongly, but self holds the controller
        // (and thus the block) strongly, which is a cycle.
        self.controller.controllerPausedHandler = {
            let curScore = self.score
            print("Pause button pressed; current score: \(curScore)")
        }

        // SOLUTION: use `weak self` to break the cycle.
        self.controller.controllerPausedHandler = { [weak self] in
            guard let strongSelf = self else { return }
            let curScore = strongSelf.score
            print("Pause button pressed; current score: \(curScore)")
        }
    }
}
```

Utilizzo di chiusure per la codifica asincrona

Le chiusure sono spesso utilizzate per attività asincrone, ad esempio quando si prelevano dati da un sito Web.

3.0

```
func getData(urlString: String, callback: (result: NSData?) -> Void) {

    // Turn the URL string into an NSURLRequest.
    guard let url = NSURL(string: urlString) else { return }
    let request = NSURLRequest(URL: url)
```

```
    // Asynchronously fetch data from the given URL.
    let task = NSURLSession.sharedSession().dataTaskWithRequest(request) {(data: NSData?,
response: NSURLResponse?, error: NSError?) in

        // We now have the NSData response from the website.
        // We can get it "out" of the function by using the callback
        // that was passed to this function as a parameter.

        callback(result: data)
    }

    task.resume()
}
```

Questa funzione è asincrona, quindi non bloccherà il thread su cui viene chiamato (non bloccherà l'interfaccia se chiamato sul thread principale dell'applicazione GUI).

3.0

```
print("1. Going to call getData")

getData("http://www.example.com") {(result: NSData?) -> Void in

    // Called when the data from http://www.example.com has been fetched.
    print("2. Fetched data")
}

print("3. Called getData")
```

Poiché l'attività è asincrona, l'output sarà in genere simile a questo:

```
"1. Going to call getData"
"3. Called getData"
"2. Fetched data"
```

Poiché il codice all'interno della chiusura, `print("2. Fetched data")` , non verrà chiamato fino a quando non verranno recuperati i dati dall'URL.

Chiusure e tipo alias

Una chiusura può essere definita con un `typealias` . Questo fornisce un comodo segnaposto di tipo se la stessa firma di chiusura viene utilizzata in più punti. Ad esempio, callback di richieste di rete comuni o gestori di eventi dell'interfaccia utente offrono ottimi candidati per essere "denominati" con un alias di tipo.

```
public typealias ClosureType = (x: Int, y: Int) -> Int
```

È quindi possibile definire una funzione utilizzando le tipealie:

```
public func closureFunction(closure: ClosureType) {
    let z = closure(1, 2)
}
```

```
closureFunction() { (x: Int, y: Int) -> Int in return x + y }
```

Leggi chiusure online: https://riptutorial.com/it/swift/topic/262/chiusure

Capitolo 9: Classi

Osservazioni

`init()` è un metodo speciale in classi che viene utilizzato per dichiarare un inizializzatore per la classe. Ulteriori informazioni possono essere trovate qui: Initializers

Examples

Definire una classe

Definisci una classe come questa:

```
class Dog {}
```

Una classe può anche essere una sottoclasse di un'altra classe:

```
class Animal {}
class Dog: Animal {}
```

In questo esempio, `Animal` potrebbe anche essere un protocollo a cui `Dog` conforma.

Semantica di riferimento

Le classi sono **tipi di riferimento** , il che significa che più variabili possono fare riferimento alla stessa istanza.

```
class Dog {
    var name = ""
}

let firstDog = Dog()
firstDog.name = "Fido"

let otherDog = firstDog  // otherDog points to the same Dog instance
otherDog.name = "Rover"  // modifying otherDog also modifies firstDog

print(firstDog.name)  // prints "Rover"
```

Poiché le classi sono tipi di riferimento, anche se la classe è una costante, le sue proprietà variabili possono comunque essere modificate.

```
class Dog {
    var name: String // name is a variable property.
    let age: Int // age is a constant property.
    init(name: String, age: Int) {
        self.name = name
        self.age = age
```

```
    }
}

let constantDog = Dog(name: "Rover", age: 5)// This instance is a constant.
var variableDog = Dog(name: "Spot", age 7)// This instance is a variable.

constantDog.name = "Fido" // Not an error because name is a variable property.
constantDog.age = 6 // Error because age is a constant property.
constantDog = Dog(name: "Fido", age: 6)
/* The last one is an error because you are changing the actual reference, not
just what the reference points to. */

variableDog.name = "Ace" // Not an error because name is a variable property.
variableDog.age = 8 // Error because age is a constant property.
variableDog = Dog(name: "Ace", age: 8)
/* The last one is not an error because variableDog is a variable instance and
therefore the actual reference can be changed. */
```

Verifica se due oggetti sono *identici* (punta alla stessa identica istanza) usando `===` :

```
class Dog: Equatable {
    let name: String
    init(name: String) { self.name = name }
}

// Consider two dogs equal if their names are equal.
func ==(lhs: Dog, rhs: Dog) -> Bool {
    return lhs.name == rhs.name
}

// Create two Dog instances which have the same name.
let spot1 = Dog(name: "Spot")
let spot2 = Dog(name: "Spot")

spot1 == spot2   // true, because the dogs are equal
spot1 != spot2   // false

spot1 === spot2  // false, because the dogs are different instances
spot1 !== spot2  // true
```

Proprietà e metodi

Le classi possono definire proprietà che possono essere utilizzate dalle istanze della classe. In questo esempio, `Dog` ha due proprietà: `name` e `dogYearAge` :

```
class Dog {
    var name = ""
    var dogYearAge = 0
}
```

È possibile accedere alle proprietà con la sintassi del punto:

```
let dog = Dog()
print(dog.name)
print(dog.dogYearAge)
```

Le classi possono anche definire metodi che possono essere richiamati sulle istanze, sono dichiarati simili alle normali funzioni , solo all'interno della classe:

```
class Dog {
    func bark() {
        print("Ruff!")
    }
}
```

Anche i metodi di chiamata usano la sintassi del punto:

```
dog.bark()
```

Classi e ereditarietà multipla

Swift non supporta l'ereditarietà multipla. Cioè, non puoi ereditare da più di una classe.

```
class Animal { ... }
class Pet { ... }

class Dog: Animal, Pet { ... } // This will result in a compiler error.
```

Invece sei incoraggiato a usare la composizione quando crei i tuoi tipi. Questo può essere realizzato usando i protocolli .

deinit

```
class ClassA {

    var timer: NSTimer!

    init() {
        // initialize timer
    }

    deinit {
        // code
        timer.invalidate()
    }
}
```

Leggi Classi online: https://riptutorial.com/it/swift/topic/459/classi

Capitolo 10: Completamento dell'handler

introduzione

Praticamente tutte le app utilizzano funzioni asincrone per impedire al codice di bloccare il thread principale.

Examples

Gestore di completamento senza argomenti di input

```
func sampleWithCompletion(completion:@escaping (()-> ())){
    let delayInSeconds = 1.0
    DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + delayInSeconds) {

        completion()

    }
}

//Call the function
sampleWithCompletion {
    print("after one second")
}
```

Gestore di completamento con argomento di input

```
enum ReadResult{
    case Successful
    case Failed
    case Pending
}

struct OutpuData {
    var data = Data()
    var result: ReadResult
    var error: Error?
}

func readData(from url: String, completion: @escaping (OutpuData) ->  Void) {
    var _data = OutpuData(data: Data(), result: .Pending, error: nil)
    DispatchQueue.global().async {
        let url=URL(string: url)
        do {
            let rawData = try Data(contentsOf: url!)
            _data.result = .Successful
            _data.data = rawData
            completion(_data)
        }
        catch let error {
            _data.result = .Failed
            _data.error = error
            completion(_data)
```

```
        }

    }
}

readData(from: "https://raw.githubusercontent.com/trev/bearcal/master/sample-data-large.json")
{ (output) in
    switch output.result {
    case .Successful:
        break
    case .Failed:
        break
    case .Pending:
        break

    }
}
```

Leggi Completamento dell'handler online: https://riptutorial.com/it/swift/topic/9378/completamento-dell-handler

Capitolo 11: Concorrenza

Sintassi

- Swift 3.0
- DispatchQueue.main // Ottieni la coda principale
- DispatchQueue (label: "my-serial-queue", attributi: [.serial, .qosBackground]) // Crea la tua coda seriale privata
- DispatchQueue.global (attributi: [.qosDefault]) // Accede a una delle code simultanee globali
- DispatchQueue.main.async {...} // Invia un'attività in modo asincrono al thread principale
- DispatchQueue.main.sync {...} // Invia un'attività in modo sincrono al thread principale
- DispatchQueue.main.asyncAfter (deadline: .now () + 3) {...} // Invio di un'attività in modo asincrono al thread principale da eseguire dopo x secondi
- Swift <3.0
- dispatch_get_main_queue () // Ottieni la coda principale in esecuzione sul thread principale
- dispatch_get_global_queue (dispatch_queue_priority_t, 0) // Ottieni la coda globale con priorità specificata dispatch_queue_priority_t
- dispatch_async (dispatch_queue_t) {() -> Void in ...} // Invio di un'attività in modo asincrono sul dispatch_queue_t specificato
- dispatch_sync (dispatch_queue_t) {() -> Void in ...} // Invio di un'attività in modo sincrono sulla dispatch_queue_t specificata
- dispatch_after (dispatch_time (DISPATCH_TIME_NOW, Int64 (nanoseconds)), dispatch_queue_t, {...}); // Invio di un'attività su dispatch_queue_t specificato dopo nanosecondi

Examples

Ottenere una coda Grand Central Dispatch (GCD)

Grand Central Dispatch lavora sul concetto di "Dispatch Queues". Una coda di invio esegue le attività designate nell'ordine in cui sono state inoltrate. Esistono tre tipi di code di invio:

- **Le code di** invio **seriale** (ovvero le code di invio private) eseguono un compito alla volta, in ordine. Sono spesso utilizzati per sincronizzare l'accesso a una risorsa.
- **Le code simultanee di** invio (ovvero le code di invio globali) eseguono

contemporaneamente una o più attività.
- La **Main Dispatch Queue** esegue le attività sul thread principale.

Per accedere alla coda principale:

3.0

```
let mainQueue = DispatchQueue.main
```

3.0

```
let mainQueue = dispatch_get_main_queue()
```

Il sistema fornisce code di invio globali *simultanee* (globali all'applicazione), con priorità diverse. Puoi accedere a queste code usando la classe `DispatchQueue` in Swift 3:

3.0

```
let globalConcurrentQueue = DispatchQueue.global(qos: .default)
```

equivalente a

```
let globalConcurrentQueue = DispatchQueue.global()
```

3.0

```
let globalConcurrentQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)
```

In iOS 8 o versioni successive, la possibile qualità dei valori di servizio che possono essere passati sono `.userInteractive` , `.userInitiated` , `.default` , `.utility` e `.background` . Sostituiscono le costanti `DISPATCH_QUEUE_PRIORITY_` .

Puoi anche creare le tue code con priorità diverse:

3.0

```
let myConcurrentQueue = DispatchQueue(label: "my-concurrent-queue", qos: .userInitiated,
attributes: [.concurrent], autoreleaseFrequency: .workItem, target: nil)
let mySerialQueue = DispatchQueue(label: "my-serial-queue", qos: .background, attributes: [],
autoreleaseFrequency: .workItem, target: nil)
```

3.0

```
let myConcurrentQueue = dispatch_queue_create("my-concurrent-queue",
DISPATCH_QUEUE_CONCURRENT)
let mySerialQueue = dispatch_queue_create("my-serial-queue", DISPATCH_QUEUE_SERIAL)
```

In Swift 3, le code create con questo inizializzatore sono di serie per impostazione predefinita e il passaggio `.workItem` per la frequenza di autorelease garantisce che un pool di autorelease venga creato e scaricato per ciascun elemento di lavoro. C'è anche. `.never` , il che significa che gestirai i

tuoi pool di autorilease autonomamente, o. `.inherit` che eredita l'impostazione dall'ambiente. Nella maggior parte dei casi probabilmente non lo userai `.never` tranne nei casi di estrema personalizzazione.

Esecuzione di attività in una coda Grand Central Dispatch (GCD)

3.0

Per eseguire attività su una coda di invio, utilizzare i metodi `sync` , `async` e `after` .

Per inviare un'attività a una coda in modo asincrono:

```
let queue = DispatchQueue(label: "myQueueName")

queue.async {
    //do something

    DispatchQueue.main.async {
        //this will be called in main thread
        //any UI updates should be placed here
    }
}
// ... code here will execute immediately, before the task finished
```

Per inviare un'attività a una coda in modo sincrono:

```
queue.sync {
    // Do some task
}
// ... code here will not execute until the task is finished
```

Per inviare un'attività in una coda dopo un determinato numero di secondi:

```
queue.asyncAfter(deadline: .now() + 3) {
    //this will be executed in a background-thread after 3 seconds
}
// ... code here will execute immediately, before the task finished
```

> **NOTA:** Eventuali aggiornamenti dell'interfaccia utente dovrebbero essere richiamati sul thread principale! Assicurati di inserire il codice per gli aggiornamenti dell'interfaccia utente all'interno di `DispatchQueue.main.async { ... }`

2.0

Tipi di coda:

```
let mainQueue = dispatch_get_main_queue()
let highQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_HIGH, 0)
let backgroundQueue = dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_BACKGROUND, 0)
```

Per inviare un'attività a una coda in modo asincrono:

```
dispatch_async(queue) {
    // Your code run run asynchronously. Code is queued and executed
    // at some point in the future.
}
// Code after the async block will execute immediately
```

Per inviare un'attività a una coda in modo sincrono:

```
dispatch_sync(queue) {
    // Your sync code
}
// Code after the sync block will wait until the sync task finished
```

Per inviare un'attività dopo un intervallo di tempo (utilizzare `NSEC_PER_SEC` per convertire secondi in nanosecondi):

```
dispatch_after(dispatch_time(DISPATCH_TIME_NOW, Int64(2.5 * Double(NSEC_PER_SEC))),
dispatch_get_main_queue()) {
    // Code to be performed in 2.5 seconds here
}
```

Per eseguire un'attività in modo asincrono e aggiornare l'interfaccia utente:

```
dispatch_async(queue) {
    // Your time consuming code here
    dispatch_async(dispatch_get_main_queue()) {
        // Update the UI code
    }
}
```

> **NOTA:** Eventuali aggiornamenti dell'interfaccia utente dovrebbero essere richiamati sul thread principale! Assicurati di aver inserito il codice per gli aggiornamenti dell'interfaccia utente all'interno di `dispatch_async(dispatch_get_main_queue()) { ... }`

Cicli concomitanti

GCD fornisce un meccanismo per eseguire un ciclo, in cui i cicli si verificano contemporaneamente l'uno rispetto all'altro. Questo è molto utile quando si eseguono una serie di calcoli computazionalmente costosi.

Considera questo ciclo:

```
for index in 0 ..< iterations {
    // Do something computationally expensive here
}
```

È possibile eseguire questi calcoli contemporaneamente utilizzando `concurrentPerform` (in Swift 3) o `dispatch_apply` (in Swift 2):

3.0

```
DispatchQueue.concurrentPerform(iterations: iterations) { index in
    // Do something computationally expensive here
}
```

3.0

```
dispatch_apply(iterations, queue) { index in
    // Do something computationally expensive here
}
```

La chiusura del ciclo sarà invocata per ogni `index` da `0` a, ma non incluse, `iterations` . Queste iterazioni verranno eseguite simultaneamente l'una rispetto all'altra e quindi l'ordine che esse eseguono non è garantito. Il numero effettivo di iterazioni che si verificano simultaneamente in un dato momento è generalmente dettato dalle capacità del dispositivo in questione (ad esempio quanti core ha il dispositivo).

Un paio di considerazioni speciali:

- Il `concurrentPerform` / `dispatch_apply` può eseguire i loop simultaneamente l'uno rispetto all'altro, ma tutto avviene in modo sincrono rispetto al thread da cui viene chiamato. Quindi, non chiamarlo dal thread principale, in quanto ciò bloccherà quel thread fino a quando il ciclo non sarà terminato.
- Poiché questi cicli si verificano contemporaneamente l'uno rispetto all'altro, si è responsabili di garantire la sicurezza del thread dei risultati. Ad esempio, se si aggiornano alcuni dizionari con i risultati di questi calcoli computazionalmente costosi, assicurarsi di sincronizzarli autonomamente.
- Nota, c'è un sovraccarico associato nell'esecuzione di cicli concomitanti. Pertanto, se i calcoli eseguiti all'interno del ciclo non sono sufficientemente intensi dal punto di vista computazionale, è possibile che le prestazioni ottenute utilizzando loop concomitanti possano essere diminuite, se non completamente compensate, dall'overhead associato alla sincronizzazione di tutti questi thread simultanei.

  Quindi, sei responsabile della determinazione della quantità corretta di lavoro da eseguire in ogni iterazione del ciclo. Se i calcoli sono troppo semplici, puoi usare "striding" per includere più work per loop. Ad esempio, anziché eseguire un ciclo simultaneo con 1 milione di calcoli triviali, è possibile eseguire 100 iterazioni nel ciclo, eseguendo 10.000 calcoli per ciclo. In questo modo viene eseguito abbastanza lavoro su ciascun thread, quindi l'overhead associato alla gestione di questi loop simultanei diventa meno significativo.

Esecuzione di attività in un'operazioneQueue

Puoi pensare a `OperationQueue` come a una linea di compiti in attesa di essere eseguita. A differenza delle code di invio in GCD, le code di operazioni non sono FIFO (first-in-first-out). Invece, eseguono compiti non appena sono pronti per essere eseguiti, a patto che ci siano abbastanza risorse di sistema per consentirne l'esecuzione.

Ottieni l' `OperationQueue` principale:

3.0

```
let mainQueue = OperationQueue.main
```

Creare un `OperationQueue` personalizzato:

3.0

```
let queue = OperationQueue()
queue.name = "My Queue"
queue.qualityOfService = .default
```

Quality of Service specifica l'importanza del lavoro o quanto l'utente è probabile che contenga risultati immediati dall'attività.

Aggiungere un `Operation` ad un `OperationQueue` :

3.0

```
// An instance of some Operation subclass
let operation = BlockOperation {
    // perform task here
}

queue.addOperation(operation)
```

Aggiungi un blocco a `OperationQueue` :

3.0

```
myQueue.addOperation {
    // some task
}
```

Aggiungere più `Operation` s a `OperationQueue` :

3.0

```
let operations = [Operation]()
// Fill array with Operations

myQueue.addOperation(operations)
```

Regola il numero di `Operation` che possono essere eseguite contemporaneamente all'interno della coda:

```
myQueue.maxConcurrentOperationCount = 3 // 3 operations may execute at once

// Sets number of concurrent operations based on current system conditions
myQueue.maxConcurrentOperationCount = NSOperationQueueDefaultMaxConcurrentOperationCount
```

La sospensione di una coda impedirà l'avvio dell'esecuzione di qualsiasi operazione esistente, non avviata o di qualsiasi nuova operazione aggiunta alla coda. Il modo per riprendere questa

coda è di impostare `isSuspended` su `false` :

3.0

```
myQueue.isSuspended = true

// Re-enable execution
myQueue.isSuspended = false
```

Sospensione di `OperationQueue` non si ferma o annullare operazioni che sono già in esecuzione. Si dovrebbe tentare solo di sospendere una coda che è stata creata, non le code globali o la coda principale.

Creazione di operazioni di alto livello

Il framework Foundation fornisce il tipo di `Operation` , che rappresenta un oggetto di alto livello che incapsula una porzione di lavoro che può essere eseguita su una coda. La coda non solo coordina le prestazioni di tali operazioni, ma è anche possibile stabilire dipendenze tra operazioni, creare operazioni cancellabili, limitare il grado di concorrenza utilizzato dalla coda di operazioni, ecc.

`Operation` s sono pronte per essere eseguite al termine dell'esecuzione di tutte le sue dipendenze. La proprietà `isReady` cambia in `true` .

Creare una sottoclasse `Operation` non simultanea semplice:

3.0

```
class MyOperation: Operation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }

}
```

2.3

```
class MyOperation: NSOperation {

    init(<parameters>) {
        // Do any setup work here
    }

    override func main() {
        // Perform the task
    }

}
```

Aggiungi un'operazione a un `OperationQueue` :

1.0

```
myQueue.addOperation(operation)
```

Questo eseguirà l'operazione contemporaneamente in coda.

Gestire le dipendenze su un `Operation` .

Le dipendenze definiscono altre `Operation` che devono essere eseguite su una coda prima che l' `Operation` sia considerata pronta per l'esecuzione.

1.0

```
operation2.addDependency(operation1)

operation2.removeDependency(operation1)
```

Eseguire un `Operation` senza una coda:

1.0

```
   operation.start()
```

Le dipendenze saranno ignorate. Se si tratta di un'operazione simultanea, l'attività può ancora essere eseguita contemporaneamente se il suo metodo di `start` scarica da lavoro alle code in background.

Operazioni simultanee.

Se il compito che l' `Operation` è da eseguire è, a sua volta, asincrono, (ad esempio un `URLSession` compito dati), è necessario implementare l' `Operation` come un'operazione simultanea. In questo caso, l'implementazione `isAsynchronous` deve restituire `true` , in genere è necessario `start` metodo che esegue alcune impostazioni, quindi chiama il suo metodo `main` che esegue effettivamente l'attività.

Quando si implementa un asincrono `Operation` inizia è necessario implementare `isExecuting` , `isFinished` metodi e KVO. Pertanto, quando viene avviata l'esecuzione, `isExecuting` property cambia in `true` . Quando un `Operation` termina il suo compito, `isExecuting` è impostato su `false` , e `isFinished` è impostata su `true` . Se l'operazione è annullata, `isCancelled` e `isFinished` cambiano in `true` . Tutte queste proprietà sono osservabili come valori chiave.

Annullamento di un `Operation` .

Chiamando `cancel` cambia semplicemente la proprietà `isCancelled` su `true` . Per rispondere alla cancellazione dalla propria sottoclasse `Operation` , è necessario verificare il valore di `isCancelled` almeno periodicamente all'interno di `main` e rispondere in modo appropriato.

1.0

```
operation.cancel()
```

Leggi Concorrenza online: https://riptutorial.com/it/swift/topic/1649/concorrenza

Capitolo 12: Condizionali

introduzione

Le espressioni condizionali, che includono parole chiave come if, else if e else, forniscono ai programmi Swift la possibilità di eseguire azioni diverse a seconda della condizione booleana: True o False. Questa sezione copre l'uso di condizionali Swift, logica booleana e dichiarazioni ternarie.

Osservazioni

Per ulteriori informazioni sulle istruzioni condizionali, vedere The Swift Programming Language .

Examples

Usando Guard

2.0

La guardia verifica una condizione e, se è falsa, entra nel ramo. Le filiali di controllo delle guardie devono lasciare il loro blocco di chiusura tramite `return` , `break` o `continue` (se applicabile); non riuscendo a farlo risulta un errore del compilatore. Questo ha il vantaggio che quando una `guard` è scritta non è possibile lasciare che il flusso continui accidentalmente (come sarebbe possibile con un `if`).

L'uso delle protezioni può aiutare a mantenere bassi i livelli di annidamento , il che di solito migliora la leggibilità del codice.

```
func printNum(num: Int) {
    guard num == 10 else {
        print("num is not 10")
        return
    }
    print("num is 10")
}
```

Guard può anche controllare se c'è un valore in un opzionale , e quindi scartarlo nello scope esterno:

```
func printOptionalNum(num: Int?) {
    guard let unwrappedNum = num else {
        print("num does not exist")
        return
    }
    print(unwrappedNum)
}
```

Guard può combinare lo unwrapping opzionale e il controllo delle condizioni utilizzando la parola chiave `where` :

```
func printOptionalNum(num: Int?) {
guard let unwrappedNum = num, unwrappedNum == 10 else {
    print("num does not exist or is not 10")
    return
}
print(unwrappedNum)
}
```

Condizioni condizionali di base: dichiarazioni if

Un'istruzione **`if`** controlla se una condizione di Bool è `true` :

```
let num = 10

if num == 10 {
    // Code inside this block only executes if the condition was true.
    print("num is 10")
}

let condition = num == 10   // condition's type is Bool
if condition {
    print("num is 10")
}
```

`if` istruzioni accettano **`else if`** e **`else`** blocchi, che possono testare condizioni alternative e fornire una riserva:

```
let num = 10
if num < 10 {  // Execute the following code if the first condition is true.
    print("num is less than 10")
} else if num == 10 {  // Or, if not, check the next condition...
    print("num is 10")
} else {  // If all else fails...
    print("all other conditions were false, so num is greater than 10")
}
```

Operatori di base come `&&` e `||` può essere utilizzato per più condizioni:

L'operatore logico AND

```
let num = 10
let str = "Hi"
if num == 10 && str == "Hi" {
    print("num is 10, AND str is \"Hi\"")
}
```

Se `num == 10` era falso, il secondo valore non sarebbe stato valutato. Questo è noto come valutazione di cortocircuito.

L'operatore logico OR

```
if num == 10 || str == "Hi" {
    print("num is 10, or str is \"Hi\")
}
```

Se `num == 10` è true, il secondo valore non verrà valutato.

L'operatore logico NOT

```
if !str.isEmpty {
    print("str is not empty")
}
```

Vincolo facoltativo e clausole "dove"

Gli optionals devono essere *scartati* prima di poter essere utilizzati nella maggior parte delle espressioni. `if let` è un *collegamento facoltativo* , che ha esito positivo se il valore facoltativo **non** era **`nil`** :

```
let num: Int? = 10 // or: let num: Int? = nil

if let unwrappedNum = num {
    // num has type Int?; unwrappedNum has type Int
    print("num was not nil: \(unwrappedNum + 1)")
} else {
    print("num was nil")
}
```

È possibile riutilizzare lo **stesso nome** per la variabile appena associata, ombreggiato all'originale:

```
// num originally has type Int?
if let num = num {
    // num has type Int inside this block
}
```

1.2 3.0

Combina più associazioni opzionali con virgole (`,`):

```
if let unwrappedNum = num, let unwrappedStr = str {
    // Do something with unwrappedNum & unwrappedStr
} else if let unwrappedNum = num {
    // Do something with unwrappedNum
} else {
    // num was nil
}
```

Applicare ulteriori vincoli dopo il legame opzionale utilizzando un `where` clausola:

```
if let unwrappedNum = num where unwrappedNum % 2 == 0 {
    print("num is non-nil, and it's an even number")
}
```

Se ti senti avventuroso, interleave un numero qualsiasi di associazioni opzionali e clausole `where` :

```
if let num = num                            // num must be non-nil
    where num % 2 == 1,                     // num must be odd
    let str = str,                          // str must be non-nil
    let firstChar = str.characters.first    // str must also be non-empty
    where firstChar != "x"                  // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

3.0

In Swift 3, `where` clausole sono state sostituite (SE-0099): basta usare un altro `,` per separare le associazioni opzionali e le condizioni booleane.

```
if let unwrappedNum = num, unwrappedNum % 2 == 0 {
    print("num is non-nil, and it's an even number")
}

if let num = num,                           // num must be non-nil
    num % 2 == 1,                           // num must be odd
    let str = str,                          // str must be non-nil
    let firstChar = str.characters.first,   // str must also be non-empty
    firstChar != "x"                        // the first character must not be "x"
{
    // all bindings & conditions succeeded!
}
```

Operatore ternario

Le condizioni possono anche essere valutate in una singola riga usando l'operatore ternario:

Se si desidera determinare il minimo e il massimo di due variabili, è possibile utilizzare le istruzioni if, in questo modo:

```
let a = 5
let b = 10
let min: Int

if a < b {
    min = a
} else {
    min = b
}

let max: Int

if a > b {
    max = a
} else {
    max = b
}
```

L'operatore condizionale ternario accetta una condizione e restituisce uno dei due valori, a seconda che la condizione fosse vera o falsa. La sintassi è la seguente: equivale ad avere l'espressione:

```
(<CONDITION>) ? <TRUE VALUE> : <FALSE VALUE>
```

Il codice sopra riportato può essere riscritto usando l'operatore condizionale ternario come di seguito:

```
let a = 5
let b = 10
let min = a < b ? a : b
let max = a > b ? a : b
```

Nel primo esempio, la condizione è a <b. Se questo è vero, il risultato assegnato a min sarà di a; se è falso, il risultato sarà il valore di b.

Nota: poiché trovare i due numeri più grandi o più piccoli è un'operazione così comune, la libreria standard Swift fornisce due funzioni a questo scopo: max e min.

Nil-Coalescing Operator

L'operatore a coalescenza nulla <OPTIONAL> ?? <DEFAULT VALUE> scava il <OPTIONAL> se contiene un valore o restituisce <DEFAULT VALUE> se è nullo. <OPTIONAL> è sempre di tipo facoltativo. <DEFAULT VALUE> deve corrispondere al tipo archiviato in <OPTIONAL> .

L'operatore a coalescenza nulla è abbreviato per il codice seguente che utilizza un operatore ternario:

```
a != nil ? a! : b
```

questo può essere verificato dal codice qui sotto:

```
(a ?? b) == (a != nil ? a! : b) // ouputs true
```

Tempo per un esempio

```
let defaultSpeed:String = "Slow"
var userEnteredSpeed:String? = nil

print(userEnteredSpeed ?? defaultSpeed) // ouputs "Slow"

userEnteredSpeed = "Fast"
print(userEnteredSpeed ?? defaultSpeed) // ouputs "Fast"
```

Leggi Condizionali online: https://riptutorial.com/it/swift/topic/475/condizionali

Capitolo 13: Controllo di accesso

Sintassi

- Progetto di classe privata
- let car = Car ("Ford", modello: "Escape") // default interno
- public enum Genere
- func privato calculateMarketCap ()
- sovrascrivi func interna setupView ()
- private (set) var area = 0

Osservazioni

1. ***Nota di base:***

Di seguito sono riportati i tre livelli di accesso dall'accesso più alto (meno restrittivo) all'accesso più basso (più restrittivo)

L' accesso **pubblico** consente l'accesso a classi, strutture, variabili, ecc. Da qualsiasi file all'interno del modello, ma soprattutto dal modulo se il file esterno importa il modulo contenente il codice di accesso pubblico. È popolare utilizzare l'accesso pubblico quando si crea un framework.

L' accesso **interno** consente ai file solo con il modulo delle entità di utilizzare le entità. Tutte le entità hanno **un** livello di accesso **interno** per impostazione predefinita (con alcune eccezioni).

L' accesso **privato** impedisce l'utilizzo dell'entità al di fuori di quel file.

2. ***Nota sottoclasse:***

Una sottoclasse non può avere un accesso superiore rispetto alla sua superclasse.

3. ***Getter & Setter Nota:***

Se il setter della proprietà è privato, il getter è interno (che è il valore predefinito). Inoltre è possibile assegnare il livello di accesso sia per il getter che per il setter. Questi principi si applicano anche agli *abbonati*

4. ***Nota generale:***

Altri tipi di entità includono: inizializzatori, protocolli, estensioni, generici e alias di tipo

Examples

Esempio di base usando una Struct

3.0

In Swift 3 ci sono più livelli di accesso. Questo esempio li utilizza tutti tranne che per open :

```
public struct Car {

    public let make: String
    let model: String //Optional keyword: will automatically be "internal"
    private let fullName: String
```

```
    fileprivate var otherName: String

    public init(_ make: String, model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
        self.otherName = "\(model) - \(make)"
    }
}
```

Supponi che myCar stato inizializzato in questo modo:

```
let myCar = Car("Apple", model: "iCar")
```

Car.make (pubblico)

```
print(myCar.make)
```

Questa stampa funzionerà ovunque, compresi gli obiettivi che importano l' Car .

Car.model (interno)

```
print(myCar.model)
```

Questo verrà compilato se il codice si trova nello stesso obiettivo di Car .

Car.otherName (file privato)

```
print(myCar.otherName)
```

Funzionerà solo se il codice è *nello stesso file* di Car .

Car.fullName (privato)

```
print(myCar.fullName)
```

Questo non funzionerà con Swift 3. È possibile accedere private proprietà private solo all'interno della stessa struct / class .

```
public struct Car {

    public let make: String       //public
    let model: String             //internal
    private let fullName: String! //private

    public init(_ make: String, model model: String) {
        self.make = make
        self.model = model
        self.fullName = "\(make)\(model)"
    }
}
```

Se l'entità ha più livelli di accesso associati, Swift cerca il livello di accesso più basso. Se esiste una variabile privata in una classe pubblica, la variabile verrà comunque considerata privata.

Esempio di sottoclasse

```
public class SuperClass {
    private func secretMethod() {}
}

internal class SubClass: SuperClass {
    override internal func secretMethod() {
        super.secretMethod()
    }
}
```

Esempio di getter e setter

```
struct Square {
    private(set) var area = 0

    var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
}

public struct Square {
    public private(set) var area = 0
    public var side: Int = 0 {
        didSet {
            area = side*side
        }
    }
    public init() {}
}
```

Leggi Controllo di accesso online: https://riptutorial.com/it/swift/topic/1075/controllo-di-accesso

Capitolo 14: Convenzioni di stile

Osservazioni

Swift ha una guida di stile ufficiale: Swift.org API Design Guidelines . Un'altra guida popolare è The Official raywenderlich.com Swift Style Guide.

Examples

Cancella uso

Evita l'ambiguità

Il nome di classi, strutture, funzioni e variabili dovrebbe evitare l'ambiguità.

Esempio:

```
extension List {
    public mutating func remove(at position: Index) -> Element {
        // implementation
    }
}
```

La chiamata di funzione a questa funzione sarà quindi simile a questa:

```
list.remove(at: 42)
```

In questo modo, si evita l'ambiguità. Se la chiamata alla funzione fosse solo list.remove(42) non sarebbe chiaro se un Elemento pari a 42 fosse rimosso o se l'Elemento nell'Indice 42 fosse rimosso.

Evitare ridondanza

Il nome delle funzioni non dovrebbe contenere informazioni ridondanti.

Un cattivo esempio potrebbe essere:

```
extension List {
    public mutating func removeElement(element: Element) -> Element? {
        // implementation
    }
}
```

Una chiamata alla funzione può apparire come list.removeElement(someObject) . La variabile someObject indica già che un elemento viene rimosso. Sarebbe meglio che la firma della funzione assomigli a questo:

```
extension List {
    public mutating func remove(_ member: Element) -> Element? {
        // implementation
    }
}
```

La chiamata a questa funzione è la seguente: list.remove(someObject) .

Denominazione delle variabili in base al loro ruolo

Le variabili dovrebbero essere nominate in base al loro ruolo (ad esempio fornitore, saluto) al posto del loro tipo (ad es. Fabbrica, stringa, ecc.)

Elevato accoppiamento tra nome protocollo e nomi variabili

Se il nome del tipo descrive il suo ruolo nella maggior parte dei casi (es. Iterator), il tipo dovrebbe essere nominato con il suffisso `Type`. (ad es. IteratorType)

Fornire ulteriori dettagli quando si utilizzano parametri debolmente tipizzati

Se il tipo di un oggetto non indica chiaramente il suo utilizzo in una chiamata di funzione, la funzione deve essere denominata con un nome precedente per ogni parametro debolmente digitato, descrivendone l'utilizzo.
Esempio:

```
func addObserver(_ observer: NSObject, forKeyPath path: String)
```

a cui apparirebbe una chiamata `object.addObserver (self, forKeyPath: path)

invece di

```
func add(_ observer: NSObject, for keyPath: String)
```

a cui una chiamata sembrerebbe object.add(self, for: path)

Utilizzo fluido

Usando il linguaggio naturale

Le chiamate di funzioni dovrebbero essere vicine alla naturale lingua inglese.

Esempio:

```
list.insert(element, at: index)
```

invece di

```
list.insert(element, position: index)
```

Denominazione dei metodi di fabbrica

I metodi di fabbrica dovrebbero iniziare con il prefisso `make`.

Esempio:

```
factory.makeObject()
```

Denominazione dei parametri negli inizializzatori e nei metodi di fabbrica

Il nome del primo argomento non dovrebbe essere coinvolto nel nominare un metodo factory o un inizializzatore.

Esempio:

```
factory.makeObject(key: value)
```

Invece di:

```
factory.makeObject(havingProperty: value)
```

Denominazione in base agli effetti collaterali

- Le funzioni con effetti collaterali (funzioni mutanti) dovrebbero essere denominate usando verbi o nomi preceduti da form- .
- Le funzioni senza effetti collaterali (funzioni nonmutating) dovrebbero essere denominate usando nomi o verbi con suffisso -ing o -ed .

Esempio: funzioni di muting:

```
print(value)
array.sort()                  // in place sorting
list.add(value)               // mutates list
set.formUnion(anotherSet)     // set is now the union of set and anotherSet
```

Funzioni di nonmutating:

```
let sortedArray = array.sorted()     // out of place sorting
let union = set.union(anotherSet)    // union is now the union of set and another set
```

Funzioni o variabili booleane

Le dichiarazioni che coinvolgono i booleani dovrebbero essere lette come affermazioni.

Esempio:

```
set.isEmpty
line.intersects(anotherLine)
```

Protocolli di denominazione

- I protocolli che descrivono cosa dovrebbe essere chiamato con nomi.
- I protocolli che descrivono le funzionalità dovrebbero avere -able , -ible o -ing come suffisso.

Esempio:

```
Collection        // describes that something is a collection
ProgressReporting // describes that something has the capability of reporting progress
Equatable         // describes that something has the capability of being equal to something
```

Tipi e proprietà

Tipi, variabili e proprietà dovrebbero essere letti come nomi.

Esempio:

```
let factory = ...
let list = [1, 2, 3, 4]
```

capitalizzazione

Tipi e protocolli

I nomi di tipo e protocollo dovrebbero iniziare con una lettera maiuscola.

Esempio:

```
protocol Collection {}
struct String {}
class UIView {}
struct Int {}
enum Color {}
```

Tutto il resto ...

Variabili, costanti, funzioni e casi di enumerazione dovrebbero iniziare con una lettera minuscola.

Esempio:

```
let greeting = "Hello"
let height = 42.0

enum Color {
    case red
    case green
    case blue
}

func print(_ string: String) {
    ...
}
```

Cammello:

Tutti i nomi dovrebbero usare il caso cammello appropriato. Custodia di cammello superiore per nomi di tipo / protocollo e custodia di cammello inferiore per tutto il resto.

Cammello superiore:

```
protocol IteratorType { ... }
```

Cassa inferiore del cammello:

```
let inputView = ...
```

Abbreviazioni

Le abbreviazioni dovrebbero essere evitate a meno che non siano usate comunemente (es. URL, ID). Se viene utilizzata un'abbreviazione, tutte le lettere devono avere lo stesso caso.

Esempio:

```
let userID: UserID = ...
let urlString: URLString = ...
```

Leggi Convenzioni di stile online: https://riptutorial.com/it/swift/topic/3031/convenzioni-di-stile

Capitolo 15: Crittografia AES

Examples

Crittografia AES in modalità CBC con IV casuale (Swift 3.0)

Il iv è prefisso ai dati crittografati

aesCBC128Encrypt creerà un IV casuale e prefisso al codice crittografato.
aesCBC128Decrypt utilizzerà il prefisso IV durante la decrittografia.

Gli input sono i dati e la chiave sono oggetti Data. Se una forma codificata come Base64, se necessario, converte e / o da nel metodo chiamante.

La chiave deve essere esattamente di 128 bit (16 byte), 192 bit (24 byte) o 256 bit (32 byte) di lunghezza. Se viene utilizzata un'altra dimensione della chiave, verrà generato un errore.

Il padding PKCS # 7 è impostato di default.

Questo esempio richiede Common Crypto
È necessario disporre di un'intestazione di bridging per il progetto:
#import <CommonCrypto/CommonCrypto.h>
Aggiungere il Security.framework al progetto.

Questo è un esempio, non il codice di produzione.

```
enum AESError: Error {
    case KeyError((String, Int))
    case IVError((String, Int))
    case CryptorError((String, Int))
}

// The iv is prefixed to the encrypted data
func aesCBCEncrypt(data:Data, keyData:Data) throws -> Data {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let cryptLength = size_t(ivSize + data.count + kCCBlockSizeAES128)
    var cryptData = Data(count:cryptLength)

    let status = cryptData.withUnsafeMutableBytes {ivBytes in
        SecRandomCopyBytes(kSecRandomDefault, kCCBlockSizeAES128, ivBytes)
    }
    if (status != 0) {
        throw AESError.IVError(("IV generation failed", Int(status)))
    }

    var numBytesEncrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = cryptData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCEncrypt),
                        CCAlgorithm(kCCAlgorithmAES),
                        options,
                        keyBytes, keyLength,
```

```
                        cryptBytes,
                        dataBytes, data.count,
                        cryptBytes+kCCBlockSizeAES128, cryptLength,
                        &numBytesEncrypted)
            }
        }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.count = numBytesEncrypted + ivSize
    }
    else {
        throw AESError.CryptorError(("Encryption failed", Int(cryptStatus)))
    }

    return cryptData;
}

// The iv is prefixed to the encrypted data
func aesCBCDecrypt(data:Data, keyData:Data) throws -> Data? {
    let keyLength = keyData.count
    let validKeyLengths = [kCCKeySizeAES128, kCCKeySizeAES192, kCCKeySizeAES256]
    if (validKeyLengths.contains(keyLength) == false) {
        throw AESError.KeyError(("Invalid key length", keyLength))
    }

    let ivSize = kCCBlockSizeAES128;
    let clearLength = size_t(data.count - ivSize)
    var clearData = Data(count:clearLength)

    var numBytesDecrypted :size_t = 0
    let options   = CCOptions(kCCOptionPKCS7Padding)

    let cryptStatus = clearData.withUnsafeMutableBytes {cryptBytes in
        data.withUnsafeBytes {dataBytes in
            keyData.withUnsafeBytes {keyBytes in
                CCCrypt(CCOperation(kCCDecrypt),
                        CCAlgorithm(kCCAlgorithmAES128),
                        options,
                        keyBytes, keyLength,
                        dataBytes,
                        dataBytes+kCCBlockSizeAES128, clearLength,
                        cryptBytes, clearLength,
                        &numBytesDecrypted)
            }
        }
    }

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        clearData.count = numBytesDecrypted
    }
    else {
        throw AESError.CryptorError(("Decryption failed", Int(cryptStatus)))
    }

    return clearData;
}
```

Esempio di utilizzo:

```
let clearData = "clearData0123456".data(using:String.Encoding.utf8)!
```

```
let keyData   = "keyData890123456".data(using:String.Encoding.utf8)!
print("clearData:   \(clearData as NSData)")
print("keyData:     \(keyData as NSData)")

var cryptData :Data?
do {
    cryptData = try aesCBCEncrypt(data:clearData, keyData:keyData)
    print("cryptData:   \(cryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCEncrypt: \(status)")
}

let decryptData :Data?
do {
    let decryptData = try aesCBCDecrypt(data:cryptData!, keyData:keyData)
    print("decryptData: \(decryptData! as NSData)")
}
catch (let status) {
    print("Error aesCBCDecrypt: \(status)")
}
```

Esempio di output:

```
clearData:   <636c6561 72446174 61303132 33343536>
keyData:     <6b657944 61746138 39303132 33343536>
cryptData:   <92c57393 f454d959 5a4d158f 6e1cd3e7 77986ee9 b2970f49 2bafcf1a 8ee9d51a bde49c31
d7780256 71837a61 60fa4be0>
decryptData: <636c6561 72446174 61303132 33343536>
```

Gli appunti:
Un tipico problema con il codice di esempio in modalità CBC è che lascia la creazione e la condivisione della IV casuale all'utente. Questo esempio include la generazione di IV, ha prefisso i dati crittografati e utilizza il prefisso IV durante la decrittografia. Ciò libera l'utente casuale dai dettagli necessari per la modalità CBC .

Per motivi di sicurezza, anche i dati crittografati dovrebbero avere l'autenticazione, questo codice di esempio non lo prevede per essere piccolo e consentire una migliore interoperabilità per altre piattaforme.

Manca anche la derivazione della chiave della chiave da una password, si suggerisce che PBKDF2 sia utilizzato come password per il testo.

Per un codice di crittografia multipiattaforma pronto per la produzione robusta, consultare RNCryptor .

> Aggiornato per utilizzare throw / catch e dimensioni di chiavi multiple in base alla chiave fornita.

Crittografia AES in modalità CBC con IV casuale (Swift 2.3)

Il iv è prefisso ai dati crittografati

aesCBC128Encrypt creerà un IV casuale e prefisso al codice crittografato. aesCBC128Decrypt utilizzerà il prefisso IV durante la decrittografia.

Gli input sono i dati e la chiave sono oggetti Data. Se una forma codificata come Base64, se necessario, converte e / o da nel metodo chiamante.

La chiave dovrebbe essere esattamente a 128 bit (16 byte). Per le altre dimensioni di chiavi, vedi l'esempio di Swift 3.0.

Il padding PKCS # 7 è impostato di default.

Questo esempio richiede Common Crypto È necessario avere un'intestazione di bridging per il progetto: #import <CommonCrypto / CommonCrypto.h> Aggiungi Security.framework al progetto.

Vedi l'esempio di Swift 3 per le note.

Questo è un esempio, non il codice di produzione.

```
func aesCBC128Encrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)
    let cryptDataLength = size_t(data.count + kCCBlockSizeAES128)
    var cryptData = [UInt8](count:ivLength + cryptDataLength, repeatedValue:0)

    let status = SecRandomCopyBytes(kSecRandomDefault, Int(ivLength),
UnsafeMutablePointer<UInt8>(cryptData));
    if (status != 0) {
        print("IV Error, errno: \(status)")
        return nil
    }

    var numBytesEncrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCEncrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              cryptData,
                              data, data.count,
                              &cryptData + ivLength, cryptDataLength,
                              &numBytesEncrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
        cryptData.removeRange(numBytesEncrypted+ivLength..<cryptData.count)
    }
    else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return cryptData;
}

func aesCBC128Decrypt(data data:[UInt8], keyData:[UInt8]) -> [UInt8]? {
    let clearLength = size_t(data.count)
    var clearData   = [UInt8](count:clearLength, repeatedValue:0)

    let keyLength   = size_t(kCCKeySizeAES128)
    let ivLength    = size_t(kCCBlockSizeAES128)

    var numBytesDecrypted :size_t = 0
    let cryptStatus = CCCrypt(CCOperation(kCCDecrypt),
                              CCAlgorithm(kCCAlgorithmAES128),
                              CCOptions(kCCOptionPKCS7Padding),
                              keyData, keyLength,
                              data,
                              UnsafePointer<UInt8>(data) + ivLength, data.count - ivLength,
                              &clearData, clearLength,
                              &numBytesDecrypted)

    if UInt32(cryptStatus) == UInt32(kCCSuccess) {
```

```
        clearData.removeRange(numBytesDecrypted..<clearLength)

    } else {
        print("Error: \(cryptStatus)")
        return nil;
    }

    return clearData;
}
```

Esempio di utilizzo:

```
let clearData = toData("clearData0123456")
let keyData   = toData("keyData890123456")

print("clearData:   \(toHex(clearData))")
print("keyData:     \(toHex(keyData))")
let cryptData = aesCBC128Encrypt(data:clearData, keyData:keyData)!
print("cryptData:   \(toHex(cryptData))")
let decryptData = aesCBC128Decrypt(data:cryptData, keyData:keyData)!
print("decryptData: \(toHex(decryptData))")
```

Esempio di output:

```
clearData:   <636c6561 72446174 61303132 33343536>
keyData:     <6b657944 61746138 39303132 33343536>
cryptData:   <9fce4323 830e3734 93dd93bf e464f72a a653a3a5 2c40d5ea e90c1017 958750a7 ff094c53
6a81b458 b1fbd6d4 1f583298>
decryptData: <636c6561 72446174 61303132 33343536>
```

Crittografia AES in modalità ECB con imbottitura PKCS7

Dalla documentazione Apple per IV,

> Questo parametro viene ignorato se si utilizza la modalità ECB o se è selezionato un algoritmo di cifratura del flusso.

```
func AESEncryption(key: String) -> String? {

        let keyData: NSData! = (key as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let data: NSData! = (self as NSString).data(using: String.Encoding.utf8.rawValue) as
NSData!

        let cryptData    = NSMutableData(length: Int(data.length) + kCCBlockSizeAES128)!

        let keyLength              = size_t(kCCKeySizeAES128)
        let operation: CCOperation = UInt32(kCCEncrypt)
        let algoritm:  CCAlgorithm = UInt32(kCCAlgorithmAES128)
        let options:   CCOptions   = UInt32(kCCOptionECBMode + kCCOptionPKCS7Padding)

        var numBytesEncrypted :size_t = 0


        let cryptStatus = CCCrypt(operation,
                                  algoritm,
                                  options,
                                  keyData.bytes, keyLength,
```

```
                                nil,
                                data.bytes, data.length,
                                cryptData.mutableBytes, cryptData.length,
                                &numBytesEncrypted)

        if UInt32(cryptStatus) == UInt32(kCCSuccess) {
            cryptData.length = Int(numBytesEncrypted)

            var bytes = [UInt8](repeating: 0, count: cryptData.length)
            cryptData.getBytes(&bytes, length: cryptData.length)

            var hexString = ""
            for byte in bytes {
                hexString += String(format:"%02x", UInt8(byte))
            }

            return hexString
        }

        return nil
    }
```

Leggi Crittografia AES online: https://riptutorial.com/it/swift/topic/7026/crittografia-aes

Capitolo 16: Derivazione chiave PBKDF2

Examples

Chiave basata sulla password Derivation 2 (Swift 3)

La derivazione della chiave basata sulla password può essere utilizzata sia per derivare una chiave di crittografia dal testo della password sia per salvare una password per scopi di autenticazione.

Esistono diversi algoritmi di hash che possono essere utilizzati tra cui SHA1, SHA256, SHA512 forniti da questo codice di esempio.

Il parametro rounds viene utilizzato per rendere il calcolo lento in modo che un utente malintenzionato debba trascorrere un tempo considerevole per ogni tentativo. I valori di ritardo tipici scendono tra 100ms e 500ms, valori più brevi possono essere utilizzati se le prestazioni non sono accettabili.

Questo esempio richiede Common Crypto
È necessario disporre di un'intestazione di bridging per il progetto:
#import <CommonCrypto/CommonCrypto.h>
Aggiungere il Security.framework al progetto.

parametri:

```
password     password String
salt         salt Data
keyByteCount number of key bytes to generate
rounds       Iteration rounds

returns      Derived key


func pbkdf2SHA1(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA256(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2SHA512(password: String, salt: Data, keyByteCount: Int, rounds: Int) -> Data? {
    return pbkdf2(hash:CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyByteCount:keyByteCount, rounds:rounds)
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: Data, keyByteCount: Int, rounds:
Int) -> Data? {
    let passwordData = password.data(using:String.Encoding.utf8)!
    var derivedKeyData = Data(repeating:0, count:keyByteCount)

    let derivationStatus = derivedKeyData.withUnsafeMutableBytes {derivedKeyBytes in
        salt.withUnsafeBytes { saltBytes in

            CCKeyDerivationPBKDF(
                CCPBKDFAlgorithm(kCCPBKDF2),
                password, passwordData.count,
                saltBytes, salt.count,
                hash,
```

```
                UInt32(rounds),
                derivedKeyBytes, derivedKeyData.count)
        }
    }
    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKeyData
}
```

Esempio di utilizzo:

```
let password     = "password"
//let salt       = "saltData".data(using: String.Encoding.utf8)!
let salt         = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let keyByteCount = 16
let rounds       = 100000

let derivedKey = pbkdf2SHA1(password:password, salt:salt, keyByteCount:keyByteCount,
rounds:rounds)
print("derivedKey (SHA1): \(derivedKey! as NSData)")
```

Esempio di output:

```
derivedKey (SHA1): <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Password Based Key Derivation 2 (Swift 2.3)

Guarda l'esempio di Swift 3 per informazioni sull'uso e note

```
func pbkdf2SHA1(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA1), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA256(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA256), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2SHA512(password: String, salt: [UInt8], keyCount: Int, rounds: Int) -> [UInt8]? {
    return pbkdf2(CCPBKDFAlgorithm(kCCPRFHmacAlgSHA512), password:password, salt:salt,
keyCount:keyCount, rounds:UInt32(rounds))
}

func pbkdf2(hash :CCPBKDFAlgorithm, password: String, salt: [UInt8], keyCount: Int, rounds:
UInt32!) -> [UInt8]! {
    let derivedKey   = [UInt8](count:keyCount, repeatedValue:0)
    let passwordData = password.dataUsingEncoding(NSUTF8StringEncoding)!

    let derivationStatus = CCKeyDerivationPBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        UnsafePointer<Int8>(passwordData.bytes), passwordData.length,
        UnsafePointer<UInt8>(salt), salt.count,
        CCPseudoRandomAlgorithm(hash),
        rounds,
        UnsafeMutablePointer<UInt8>(derivedKey),
```

```
        derivedKey.count)


    if (derivationStatus != 0) {
        print("Error: \(derivationStatus)")
        return nil;
    }

    return derivedKey
}
```

Esempio di utilizzo:

```
let password = "password"
// let salt  = [UInt8]("saltData".utf8)
let salt     = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let rounds   = 100_000
let keyCount = 16

let derivedKey = pbkdf2SHA1(password, salt:salt, keyCount:keyCount, rounds:rounds)
print("derivedKey (SHA1):   \(NSData(bytes:derivedKey!, length:derivedKey!.count))")
```

Esempio di output:

```
derivedKey (SHA1):   <6b9d4fa3 0385d128 f6d196ee 3f1d6dbf>
```

Taratura di derivazione chiave basata su password (Swift 2.3)

Guarda l'esempio di Swift 3 per informazioni sull'uso e note

```
func pbkdf2SHA1Calibrate(password:String, salt:[UInt8], msec:Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Esempio di utilizzo:

```
let saltData       = [UInt8]([0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Esempio di output:

```
     Per un ritardo di 100 msec, round: 94339
```

Taratura di derivazione chiave basata su password (Swift 3)

Determina il numero di round PRF da utilizzare per un ritardo specifico sulla piattaforma corrente.

Diversi parametri sono impostati in modo predefinito su valori rappresentativi che non dovrebbero influire materialmente sul conteggio del round.

```
password Sample password.
salt     Sample salt.
msec     Targeted duration we want to achieve for a key derivation.

returns  The number of iterations to use for the desired processing time.


func pbkdf2SHA1Calibrate(password: String, salt: Data, msec: Int) -> UInt32 {
    let actualRoundCount: UInt32 = CCCalibratePBKDF(
        CCPBKDFAlgorithm(kCCPBKDF2),
        password.utf8.count,
        salt.count,
        CCPseudoRandomAlgorithm(kCCPRFHmacAlgSHA1),
        kCCKeySizeAES256,
        UInt32(msec));
    return actualRoundCount
}
```

Esempio di utilizzo:

```
let saltData       = Data(bytes: [0x73, 0x61, 0x6c, 0x74, 0x44, 0x61, 0x74, 0x61])
let passwordString = "password"
let delayMsec      = 100

let rounds = pbkdf2SHA1Calibrate(password:passwordString, salt:saltData, msec:delayMsec)
print("For \(delayMsec) msec delay, rounds: \(rounds)")
```

Esempio di output:

```
For 100 msec delay, rounds: 93457
```

Leggi Derivazione chiave PBKDF2 online: https://riptutorial.com/it/swift/topic/7053/derivazione-chiave-pbkdf2

Capitolo 17: Design Patterns - Creazionale

introduzione

I modelli di progettazione sono soluzioni generali ai problemi che si verificano frequentemente nello sviluppo del software. I seguenti sono modelli di migliori pratiche standardizzate nella strutturazione e progettazione del codice, nonché esempi di contesti comuni in cui questi modelli di progettazione sarebbero appropriati.

I modelli di progettazione creazionali astraggono l'istanziazione degli oggetti per rendere un sistema più indipendente dal processo di creazione, composizione e rappresentazione.

Examples

Singleton

I singleton sono uno schema di progettazione frequentemente utilizzato che consiste in una singola istanza di una classe condivisa in tutto il programma.

Nell'esempio seguente, creiamo una proprietà static che contiene un'istanza della classe Foo . Ricorda che una proprietà static è condivisa tra tutti gli oggetti di una classe e non può essere sovrascritta dalla sottoclasse.

```
public class Foo
{
    static let shared = Foo()

    // Used for preventing the class from being instantiated directly
    private init() {}

    func doSomething()
    {
        print("Do something")
    }
}
```

Uso:

```
Foo.shared.doSomething()
```

Assicurati di ricordare l'inizializzatore private :

> Questo assicura che i tuoi singleton siano veramente unici e impedisce agli oggetti esterni di creare le proprie istanze della tua classe attraverso il controllo degli accessi. Poiché tutti gli oggetti sono dotati di un inizializzatore pubblico predefinito in Swift, è necessario sovrascrivere il tuo init e renderlo privato.
> KrakenDev

Metodo di fabbrica

> Nella programmazione basata su classi, lo schema del metodo factory è uno schema creativo che utilizza i metodi factory per affrontare il problema della creazione di oggetti senza dover specificare la classe esatta dell'oggetto che verrà creato.
> Riferimento di Wikipedia

```
protocol SenderProtocol
{
    func send(package: AnyObject)
}
```

```
class Fedex: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Fedex deliver")
    }
}

class RegularPriorityMail: SenderProtocol
{
    func send(package: AnyObject)
    {
        print("Regular Priority Mail deliver")
    }
}

// This is our Factory
class DeliverFactory
{
    // It will be responsable for returning the proper instance that will handle the task
    static func makeSender(isLate isLate: Bool) -> SenderProtocol
    {
        return isLate ? Fedex() : RegularPriorityMail()
    }
}

// Usage:
let package = ["Item 1", "Item 2"]

// Fedex class will handle the delivery
DeliverFactory.makeSender(isLate:true).send(package)

// Regular Priority Mail class will handle the delivery
DeliverFactory.makeSender(isLate:false).send(package)
```

Facendo ciò non dipendiamo dalla reale implementazione della classe, rendendo il sender() completamente trasparente a chi lo sta consumando.

In questo caso tutto ciò che dobbiamo sapere è che un mittente gestirà la consegna e esporrà un metodo chiamato send() . Ci sono molti altri vantaggi: ridurre le classi di accoppiamento, più facile da testare, più facile aggiungere nuovi comportamenti senza dover cambiare chi lo sta consumando.

> All'interno della progettazione orientata agli oggetti, le interfacce forniscono livelli di astrazione che facilitano la spiegazione concettuale del codice e creano una barriera che impedisce le dipendenze. Riferimento di Wikipedia

Osservatore

> Lo schema dell'osservatore è dove un oggetto, chiamato soggetto, mantiene una lista dei suoi dipendenti, chiamati osservatori, e li notifica automaticamente di qualsiasi cambiamento di stato, di solito chiamando uno dei loro metodi. Viene principalmente utilizzato per implementare sistemi di gestione degli eventi distribuiti. Il pattern Observer è anche una parte fondamentale nel modello di architettura modello-vista-controller (MVC) familiare. Riferimento di Wikipedia

Fondamentalmente il pattern di osservatore viene utilizzato quando si dispone di un oggetto che può informare gli osservatori di determinati comportamenti o cambiamenti di stato.

In primo luogo consente di creare un riferimento globale (al di fuori di una classe) per il Centro di notifica:

```
let notifCentre = NotificationCenter.default
```

Ottimo ora possiamo chiamarlo da qualsiasi luogo. Vorremmo quindi registrare una lezione come osservatore ...

```
notifCentre.addObserver(self, selector: #selector(self.myFunc), name: "myNotification",
object: nil)
```

Questo aggiunge la classe come osservatore per "readForMyFunc". Indica anche che la funzione myFunc deve essere chiamata quando viene ricevuta la notifica. Questa funzione dovrebbe essere scritta nella stessa classe:

```
func myFunc(){
    print("The notification has been received")
}
```

> Uno dei vantaggi di questo schema è che è possibile aggiungere molte classi come osservatori e quindi eseguire molte azioni dopo una notifica.

La notifica può ora essere semplicemente inviata (o pubblicata se preferisci) da quasi ovunque nel codice con la linea:

```
notifCentre.post(name: "myNotification", object: nil)
```

> È anche possibile passare informazioni con la notifica come dizionario

```
let myInfo = "pass this on"
notifCentre.post(name: "myNotification", object: ["moreInfo":myInfo])
```

Ma poi devi aggiungere una notifica alla tua funzione:

```
func myFunc(_ notification: Notification){
    let userInfo = (notification as NSNotification).userInfo as! [String: AnyObject]
    let passedInfo = userInfo["moreInfo"]
    print("The notification \(moreInfo) has been received")
    //prints - The notification pass this on has been received
}
```

Catena di responsabilità

> Nella progettazione orientata agli oggetti, il modello della catena di responsabilità è un modello di progettazione costituito da una sorgente di oggetti di command e una serie di oggetti di processing . Ogni oggetto di processing contiene la logica che definisce i tipi di oggetti comando che può gestire; il resto viene passato al successivo oggetto di processing nella catena. Esiste anche un meccanismo per aggiungere nuovi oggetti di processing alla fine di questa catena. Wikipedia

Impostare le classi che costituivano la catena di responsabilità.

Per prima cosa creiamo un'interfaccia per tutti gli oggetti di processing .

```
protocol PurchasePower {
var allowable : Float { get }
  var role : String { get }
  var successor : PurchasePower? { get set }
}

extension PurchasePower {
```

```
  func process(request : PurchaseRequest){
    if request.amount < self.allowable {
      print(self.role + " will approve $ \(request.amount) for \(request.purpose)")
    } else if successor != nil {
      successor?.process(request: request)
    }
  }
}
```

Quindi creiamo l'oggetto command .

```
struct PurchaseRequest {
  var amount : Float
  var purpose : String
}
```

Infine, creando oggetti che costituivano la catena di responsabilità.

```
class ManagerPower : PurchasePower {
  var allowable: Float = 20
  var role : String = "Manager"
  var successor: PurchasePower?
}

class DirectorPower : PurchasePower {
  var allowable: Float = 100
  var role = "Director"
  var successor: PurchasePower?
}

class PresidentPower : PurchasePower {
  var allowable: Float = 5000
  var role = "President"
  var successor: PurchasePower?
}
```

Iniziare e incatenarlo insieme:

```
let manager = ManagerPower()
let director = DirectorPower()
let president = PresidentPower()

manager.successor = director
director.successor = president
```

Il meccanismo per concatenare oggetti qui è l'accesso alla proprietà

Creazione della richiesta per eseguirlo:

```
manager.process(request: PurchaseRequest(amount: 2, purpose: "buying a pen"))  // Manager will
approve $ 2.0 for buying a pen
manager.process(request: PurchaseRequest(amount: 90, purpose: "buying a printer")) // Director
will approve $ 90.0 for buying a printer

manager.process(request: PurchaseRequest(amount: 2000, purpose: "invest in stock")) //
President will approve $ 2000.0 for invest in stock
```

Iterator

> Nella programmazione del computer, un iteratore è un oggetto che consente a un programmatore di attraversare un contenitore, in particolare gli elenchi. Wikipedia

```
struct Turtle {
  let name: String
}

struct Turtles {
  let turtles: [Turtle]
}

struct TurtlesIterator: IteratorProtocol {
  private var current = 0
  private let turtles: [Turtle]

  init(turtles: [Turtle]) {
    self.turtles = turtles
  }

  mutating func next() -> Turtle? {
    defer { current += 1 }
    return turtles.count > current ? turtles[current] : nil
  }
}

extension Turtles: Sequence {
  func makeIterator() -> TurtlesIterator {
    return TurtlesIterator(turtles: turtles)
  }
}
```

E l'esempio di utilizzo sarebbe

```
let ninjaTurtles = Turtles(turtles: [Turtle(name: "Leo"),
                                     Turtle(name: "Mickey"),
                                     Turtle(name: "Raph"),
                                     Turtle(name: "Doney")])
print("Splinter and")
for turtle in ninjaTurtles {
  print("The great: \(turtle)")
}
```

Modello costruttore

> Il modello di builder è un **modello di progettazione di software per la creazione di oggetti** . A differenza del pattern factory astratto e del pattern method factory la cui intenzione è quella di abilitare il polimorfismo, l'intenzione del pattern builder è quella di trovare una soluzione all'anti-pattern del costruttore telescopico. L'anti-pattern del costruttore telescopico si verifica quando l'aumento della combinazione di parametri del costruttore oggetto porta a un elenco esponenziale di costruttori. Anziché utilizzare numerosi costruttori, il pattern di builder utilizza un altro oggetto, un builder, che riceve ogni parametro di inizializzazione passo dopo passo e quindi restituisce l'oggetto costruito risultante in una volta.
>
> *-Wikipedia*

L'obiettivo principale del modello di builder è impostare una configurazione predefinita per un oggetto dalla sua creazione. È un intermediario tra l'oggetto sarà costruito e tutti gli altri oggetti relativi alla sua costruzione.

Esempio:

Per rendere più chiaro, diamo un'occhiata a un esempio di *Car Builder* .

Considera che abbiamo una classe *Car* contiene molte opzioni per creare un oggetto, come ad esempio:

- Colore.
- Numero di posti.
- Numero di ruote
- Genere.
- Tipo di attrezzo.
- Il motore.
- Disponibilità di airbag

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}

enum GearType {
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

class Car: CustomStringConvertible {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
```

```
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}
```

Creare un oggetto auto:

```
let aCar = Car(color: UIColor.black,
               numberOfSeats: 4,
               numberOfWheels: 4,
               type: .saloon,
               gearType: .automatic,
               motor: Motor(id: "101", name: "Super Motor",
                            model: "c4", numberOfCylinders: 6),
               shouldHasAirbags: true)

print(aCar)

/* Printing
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 4
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "101", name: "Super Motor", model: "c4", numberOfCylinders: 6)
 Airbag Availability: true
*/
```

Il **problema** sorge quando si crea un oggetto auto è che l'auto richiede la creazione di molti dati di configurazione.

Per applicare il Pattern Builder, i parametri di inizializzazione dovrebbero avere valori predefiniti *che sono modificabili se necessario* .

Classe CarBuilder:

```
class CarBuilder {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

La classe CarBuilder definisce le proprietà che possono essere modificate per modificare i valori dell'oggetto auto creato.

Costruiamo nuove auto usando il CarBuilder :

```
var builder = CarBuilder()
// currently, the builder creates cars with default configuration.
```

```
let defaultCar = builder.buildCar()
//print(defaultCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
*/

builder.shouldHasAirbags = true
// now, the builder creates cars with default configuration,
// but with a small edit on making the airbags available

let safeCar = builder.buildCar()
print(safeCar.description)
/* prints
 color: UIExtendedGrayColorSpace 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
 */

builder.color = UIColor.purple
// now, the builder creates cars with default configuration
// with some extra features: the airbags are available and the color is purple

let femaleCar = builder.buildCar()
print(femaleCar)
/* prints
 color: UIExtendedSRGBColorSpace 0.5 0 0.5 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: true
*/
```

Il **vantaggio** dell'applicazione del modello di builder è la facilità di creare oggetti che dovrebbero contenere molte configurazioni impostando valori predefiniti, inoltre, la facilità di modificare questi valori predefiniti.

Prendilo ulteriormente:

Come buona pratica, tutte le proprietà che hanno bisogno di valori di default dovrebbero essere in un *protocollo separato* , che dovrebbe essere implementato dalla classe stessa e dal suo costruttore.

Seguendo il nostro esempio, creiamo un nuovo protocollo chiamato CarBluePrint :

```
import UIKit

enum CarType {
    case

    sportage,
    saloon
}
```

```
enum GearType {
    case

    manual,
    automatic
}

struct Motor {
    var id: String
    var name: String
    var model: String
    var numberOfCylinders: UInt8
}

protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }
}

class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\n Type: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags

    }
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.black
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
```

```
                            model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags)
    }
}
```

Il vantaggio di dichiarare le proprietà che necessitano di un valore predefinito in un protocollo è la forzatura di implementare qualsiasi nuova proprietà aggiunta; Quando una classe si conforma a un protocollo, deve dichiarare tutte le sue proprietà / metodi.

Considera che c'è una nuova funzione richiesta che dovrebbe essere aggiunta al progetto di creazione di un'auto chiamata "nome batteria":

```
protocol CarBluePrint {
    var color: UIColor { get set }
    var numberOfSeats: UInt8 { get set }
    var numberOfWheels: UInt8 { get set }
    var type: CarType { get set }
    var gearType: GearType { get set }
    var motor: Motor { get set }
    var shouldHasAirbags: Bool { get set }

    // adding the new property
    var batteryName: String { get set }
}
```

Dopo aver aggiunto la nuova proprietà, notare che si verificheranno due errori in fase di compilazione, notificando che la conformità al protocollo CarBluePrint richiede la dichiarazione della proprietà 'batteryName'. Ciò garantisce che CarBuilder dichiarerà e imposterà un valore predefinito per la proprietà batteryName .

Dopo aver aggiunto batteryName nuova proprietà CarBluePrint protocollo, l'attuazione di entrambe le Car e CarBuilder classi dovrebbe essere:

```
class Car: CustomStringConvertible, CarBluePrint {
    var color: UIColor
    var numberOfSeats: UInt8
    var numberOfWheels: UInt8
    var type: CarType
    var gearType: GearType
    var motor: Motor
    var shouldHasAirbags: Bool
    var batteryName: String

    var description: String {
        return "color: \(color)\nNumber of seats: \(numberOfSeats)\nNumber of Wheels:
\(numberOfWheels)\nType: \(gearType)\nMotor: \(motor)\nAirbag Availability:
\(shouldHasAirbags)\nBattery Name: \(batteryName)"
    }

    init(color: UIColor, numberOfSeats: UInt8, numberOfWheels: UInt8, type: CarType, gearType:
GearType, motor: Motor, shouldHasAirbags: Bool, batteryName: String) {

        self.color = color
        self.numberOfSeats = numberOfSeats
        self.numberOfWheels = numberOfWheels
        self.type = type
        self.gearType = gearType
```

```
        self.motor = motor
        self.shouldHasAirbags = shouldHasAirbags
        self.batteryName = batteryName
    }
}

class CarBuilder: CarBluePrint {
    var color: UIColor = UIColor.red
    var numberOfSeats: UInt8 = 5
    var numberOfWheels: UInt8 = 4
    var type: CarType = .saloon
    var gearType: GearType = .automatic
    var motor: Motor = Motor(id: "111", name: "Default Motor",
                             model: "T9", numberOfCylinders: 4)
    var shouldHasAirbags: Bool = false
    var batteryName: String = "Default Battery Name"

    func buildCar() -> Car {
        return Car(color: color, numberOfSeats: numberOfSeats, numberOfWheels: numberOfWheels,
type: type, gearType: gearType, motor: motor, shouldHasAirbags: shouldHasAirbags, batteryName:
batteryName)
    }
}
```

Ancora una volta, costruiamo nuove auto usando il CarBuilder :

```
var builder = CarBuilder()

let defaultCar = builder.buildCar()
print(defaultCar)
/* prints
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: Default Battery Name
*/

builder.batteryName = "New Battery Name"

let editedBatteryCar = builder.buildCar()
print(editedBatteryCar)
/*
 color: UIExtendedSRGBColorSpace 1 0 0 1
 Number of seats: 5
 Number of Wheels: 4
 Type: automatic
 Motor: Motor(id: "111", name: "Default Motor", model: "T9", numberOfCylinders: 4)
 Airbag Availability: false
 Battery Name: New Battery Name
 */
```

Leggi Design Patterns - Creazionale online: https://riptutorial.com/it/swift/topic/4941/design-patterns---creazionale

Capitolo 18: Digitare Casting

Sintassi

- let name = json["name"] as? String ?? "" // Output: john
- let name = json["name"] as? String // Output: Optional("john")
- let name = rank as? Int // Output: Optional(1)
- let name = rank as? Int ?? 0 // Output: 1
- let name = dictionary as? [String: Any] ?? [:] // Output: ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]]

Examples

downcasting

Una variabile può essere downcasted ad un sottotipo usando gli *operatori di tipo cast* as? e as! .

Il as? l'operatore *tenta* di trasmettere a un sottotipo.
Può fallire, quindi restituisce un optional.

```
let value: Any = "John"

let name = value as? String
print(name) // prints Optional("John")

let age = value as? Double
print(age) // prints nil
```

Il as! l'operatore *costringe* un cast.
Non restituisce un optional, ma si blocca se il cast fallisce.

```
let value: Any = "Paul"

let name = value as! String
print(name) // prints "Paul"

let age = value as! Double // crash: "Could not cast value…"
```

È comune utilizzare gli operatori cast di tipo con lo unwrapping condizionale:

```
let value: Any = "George"

if let name = value as? String {
    print(name) // prints "George"
}

if let age = value as? Double {
    print(age) // Not executed
}
```

Casting con switch

L'istruzione switch può anche essere utilizzata per tentare la trasmissione in tipi diversi:

```
func checkType(_ value: Any) -> String {
    switch value {

    // The `is` operator can be used to check a type
    case is Double:
        return "value is a Double"

    // The `as` operator will cast. You do not need to use `as?` in a `switch`.
    case let string as String:
        return "value is the string: \(string)"

    default:
        return "value is something else"
    }

}

checkType("Cadena")  // "value is the string: Cadena"
checkType(6.28)      // "value is a Double"
checkType(UILabel()) // "value is something else"
```

upcasting

L'operatore as interpreterà un supertipo. Come non può fallire, non restituisce un optional.

```
let name = "Ringo"
let value = string as Any  // `value` is of type `Any` now
```

Esempio di utilizzo di un downcast su un parametro di funzione che coinvolge sottoclassi

Un downcast può essere usato per fare uso del codice e dei dati di una sottoclasse all'interno di una funzione prendendo un parametro della sua superclasse.

```
class Rat {
    var color = "white"
}

class PetRat: Rat {
    var name = "Spot"
}

func nameOfRat( : Rat) -> String {
    guard let petRat = ( as? PetRat) else {
        return "No name"
    }

    return petRat.name
}

let noName = Rat()
let spot = PetRat()

print(nameOfRat(noName))
print(nameOfRat(spot))
```

Digita casting in Swift Language

Digitare Casting

Il cast di tipo è un modo per verificare il tipo di un'istanza o per trattare quell'istanza come una diversa superclasse o sottoclasse da un'altra parte nella propria gerarchia di classi.

Il casting di tipo in Swift è implementato con gli operatori is e as. Questi due operatori forniscono un modo semplice ed espressivo per verificare il tipo di un valore o trasmettere un valore a un tipo diverso.

---

downcasting

Una costante o variabile di un certo tipo di classe può effettivamente fare riferimento a un'istanza di una sottoclasse dietro le quinte. Dove credi che sia questo il caso, puoi provare a downcast al tipo di sottoclasse con un operatore di cast di tipo (come? O come!).

Poiché il downcasting può fallire, l'operatore di cast del tipo si presenta in due forme diverse. La forma condizionale, come?, Restituisce un valore facoltativo del tipo a cui si sta tentando di eseguire il downcast. La forma forzata, come !, tenta il downcast e forza-scartare il risultato come una singola azione composta.

Usa la forma condizionale dell'operatore di tipo cast (come?) Quando non sei sicuro che il downcast avrà successo. Questa forma dell'operatore restituirà sempre un valore opzionale e il valore sarà pari a zero se il downcast non fosse possibile. Ciò ti consente di verificare un downcast di successo.

Usa la forma forzata dell'operatore di tipo cast (come!) Solo quando sei sicuro che il downcast avrà sempre successo. Questa forma dell'operatore attiverà un errore di runtime se si tenta di eseguire il downcast su un tipo di classe errato. Saperne di più

Conversione da stringa a Int e Float: -

```
    let numbers = "888.00"
    let intValue = NSString(string: numbers).integerValue
    print(intValue) // Output - 888


    let numbers = "888.00"
    let floatValue = NSString(string: numbers).floatValue
    print(floatValue) // Output : 888.0
```

Conversione Float to String

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue) // Output : 888.0

    // Get Float value at particular decimal point
    let numbers = 888.00
    let floatValue = String(format: "%.2f", numbers) // Here %.2f will give 2 numbers after
decimal points we can use as per our need
    print(floatValue) // Output : "888.00"
```

Intero al valore di stringa

```
    let numbers = 888
    let intValue = String(numbers)
    print(intValue) // Output : "888"
```

Converti in valore stringa

```
    let numbers = 888.00
    let floatValue = String(numbers)
    print(floatValue)
```

Valore Float facoltativo a stringa

```
    let numbers: Any = 888.00
    let floatValue = String(describing: numbers)
    print(floatValue) // Output : 888.0
```

Stringa facoltativa al valore Int

```
    let hitCount = "100"
    let data :AnyObject = hitCount
    let score = Int(data as? String ?? "") ?? 0
    print(score)
```

Valori di downcast da JSON

```
    let json = ["name" : "john", "subjects": ["Maths", "Science", "English", "C Language"]] as
[String : Any]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Valori di downcast da JSON opzionale

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]]
    let json = response as? [String: Any] ?? [:]
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Gestisci la risposta JSON con condizioni

```
    let response: Any = ["name" : "john", "subjects": ["Maths", "Science", "English", "C
Language"]] //Optional Response

    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
```

```
        return
    }
    let name = json["name"] as? String ?? ""
    print(name) // Output : john
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects) // Output : ["Maths", "Science", "English", "C Language"]
```

Gestisci la risposta negativa con la condizione

```
    let response: Any? = nil
    guard let json = response as? [String: Any] else {
        // Handle here nil value
        print("Empty Dictionary")
        // Do something here
        return
    }
    let name = json["name"] as? String ?? ""
    print(name)
    let subjects = json["subjects"] as? [String] ?? []
    print(subjects)
```

Uscita: **Empty Dictionary**

Leggi Digitare Casting online: https://riptutorial.com/it/swift/topic/3082/digitare-casting

Capitolo 19: dizionari

Osservazioni

Alcuni esempi in questo argomento potrebbero avere un ordine diverso quando vengono utilizzati perché l'ordine dei dizionari non è garantito.

Examples

Dichiarazione dei dizionari

I dizionari sono una collezione non ordinata di chiavi e valori. I valori si riferiscono a chiavi univoche e devono essere dello stesso tipo.

Quando si inizializza un dizionario, la sintassi completa è la seguente:

```
var books : Dictionary<Int, String> = Dictionary<Int, String>()
```

Sebbene un modo più conciso di inizializzare:

```
var books = [Int: String]()
// or
var books: [Int: String] = [:]
```

Dichiarare un dizionario con chiavi e valori specificandoli in un elenco separato da virgole. I tipi possono essere dedotti dai tipi di chiavi e valori.

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
//books = [2: "Book 2", 1: "Book 1"]
var otherBooks = [3: "Book 3", 4: "Book 4"]
//otherBooks = [3: "Book 3", 4: "Book 4"]
```

Modifica dei dizionari

Aggiungi una chiave e un valore a un dizionario

```
var books = [Int: String]()
//books = [:]
books[5] = "Book 5"
//books = [5: "Book 5"]
books.updateValue("Book 6", forKey: 5)
//[5: "Book 6"]
```

updateValue restituisce il valore originale se ne esiste uno o zero.

```
let previousValue = books.updateValue("Book 7", forKey: 5)
//books = [5: "Book 7"]
//previousValue = "Book 6"
```

Rimuovi il valore e le loro chiavi con sintassi simile

```
books[5] = nil
//books [:]
books[6] = "Deleting from Dictionaries"
//books = [6: "Deleting from Dictionaries"]
let removedBook = books.removeValueForKey(6)
```

```
//books = [:]
//removedValue = "Deleting from Dictionaries"
```

Accesso ai valori

È possibile accedere a un valore in un Dictionary utilizzando la sua chiave:

```
var books: [Int: String] = [1: "Book 1", 2: "Book 2"]
let bookName = books[1]
//bookName = "Book 1"
```

I valori di un dizionario possono essere iterati usando la proprietà values :

```
for book in books.values {
    print("Book Title: \(book)")
}
//output: Book Title: Book 2
//output: Book Title: Book 1
```

Allo stesso modo, le chiavi di un dizionario possono essere iterate usando la sua proprietà keys :

```
for bookNumbers in books.keys {
    print("Book number: \(bookNumber)")
}
// outputs:
// Book number: 1
// Book number: 2
```

Per ottenere che tutte key coppie di key e value corrispondano l'una all'altra (non entrerai nell'ordine corretto poiché è un dizionario)

```
for (book,bookNumbers)in books{
print("\(book)  \(bookNumbers)")
}
// outputs:
// 2  Book 2
// 1  Book 1
```

Si noti che un Dictionary , a differenza di una Array , è intrinsecamente non ordinato, ovvero non esiste alcuna garanzia sull'ordine durante l'iterazione.

Se si desidera accedere a più livelli di un dizionario, utilizzare una sintassi ripetuta del pedice.

```
// Create a multilevel dictionary.
var myDictionary: [String:[Int:String]]! =
["Toys":[1:"Car",2:"Truck"],"Interests":[1:"Science",2:"Math"]]

print(myDictionary["Toys"][2]) // Outputs "Truck"
print(myDictionary["Interests"][1]) // Outputs "Science"
```

Cambia valore del dizionario usando la chiave

```
var dict = ["name": "John", "surname": "Doe"]
// Set the element with key: 'name' to 'Jane'
dict["name"] = "Jane"
```

```
print(dict)
```

Ottieni tutte le chiavi nel dizionario

```
let myAllKeys = ["name" : "Kirit" , "surname" : "Modi"]
let allKeys = Array(myAllKeys.keys)
print(allKeys)
```

Unisci due dizionari

```
extension Dictionary {
    func merge(dict: Dictionary<Key,Value>) -> Dictionary<Key,Value> {
        var mutableCopy = self
        for (key, value) in dict {
            // If both dictionaries have a value for same key, the value of the other
dictionary is used.
            mutableCopy[key] = value
        }
        return mutableCopy
    }
}
```

Leggi dizionari online: https://riptutorial.com/it/swift/topic/310/dizionari

Capitolo 20: Enums

Osservazioni

Come le strutture e diversamente dalle classi, le enumerazioni sono tipi di valore e vengono copiate anziché riferite quando passate in giro.

Per ulteriori informazioni sull'enumerazione, vedere The Swift Programming Language .

Examples

Enumerazioni di base

Un enum fornisce un insieme di valori correlati:

```
enum Direction {
    case up
    case down
    case left
    case right
}

enum Direction { case up, down, left, right }
```

I valori Enum possono essere utilizzati dal loro nome completo, ma è possibile omettere il nome del tipo quando può essere desunto:

```
let dir = Direction.up
let dir: Direction = Direction.up
let dir: Direction = .up

// func move(dir: Direction)...
move(Direction.up)
move(.up)

obj.dir = Direction.up
obj.dir = .up
```

Il modo più fondamentale per confrontare / estrarre i valori enum è con un'istruzione switch :

```
switch dir {
case .up:
    // handle the up case
case .down:
    // handle the down case
case .left:
    // handle the left case
case .right:
    // handle the right case
}
```

Le enumerazioni semplici sono automaticamente Hashable , Equatable e hanno conversioni di stringhe:

```
if dir == .down { ... }

let dirs: Set<Direction> = [.right, .left]
```

```
print(Direction.up)  // prints "up"
debugPrint(Direction.up)  // prints "Direction.up"
```

Enum con valori associati

I casi Enum possono contenere uno o più **payload** (**valori associati**):

```
enum Action {
    case jump
    case kick
    case move(distance: Float)  // The "move" case has an associated distance
}
```

Il payload deve essere fornito durante l'istanziazione del valore enum:

```
performAction(.jump)
performAction(.kick)
performAction(.move(distance: 3.3))
performAction(.move(distance: 0.5))
```

L'istruzione switch può estrarre il valore associato:

```
switch action {
case .jump:
    ...
case .kick:
    ...
case .move(let distance):  // or case let .move(distance):
    print("Moving: \(distance)")
}
```

Un'estrazione caso singolo può essere eseguita utilizzando il if case :

```
if case .move(let distance) = action {
    print("Moving: \(distance)")
}
```

La sintassi del guard case può essere utilizzata per l'estrazione successiva:

```
guard case .move(let distance) = action else {
    print("Action is not move")
    return
}
```

Le enumerazioni con valori associati non sono Equatable per impostazione predefinita.
L'implementazione dell'operatore == deve essere eseguita manualmente:

```
extension Action: Equatable { }

func ==(lhs: Action, rhs: Action) -> Bool {
    switch lhs {
    case .jump: if case .jump = rhs { return true }
    case .kick: if case .kick = rhs { return true }
    case .move(let lhsDistance): if case .move (let rhsDistance) = rhs { return lhsDistance ==
rhsDistance }
    }
    return false
```

```
}
```

Carichi utili indiretti

Normalmente, le enumerazioni non possono essere ricorsive (perché richiederebbero una memoria infinita):

```
enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)  // error: recursive enum 'Tree<T>' is not marked 'indirect'
}
```

La parola chiave **indirect** rende l'enumerazione del carico utile con uno strato di riferimento indiretto, anziché memorizzarlo in linea. Puoi utilizzare questa parola chiave in un singolo caso:

```
enum Tree<T> {
    case leaf(T)
    indirect case branch(Tree<T>, Tree<T>)
}

let tree = Tree.branch(.leaf(1), .branch(.leaf(2), .leaf(3)))
```

indirect funziona anche sull'intero enum, rendendolo comunque indiretto quando necessario:

```
indirect enum Tree<T> {
    case leaf(T)
    case branch(Tree<T>, Tree<T>)
}
```

Valori grezzi e hash

Le enumerazioni prive di payload possono avere *valori* non elaborati di qualsiasi tipo letterale:

```
enum Rotation: Int {
    case up = 0
    case left = 90
    case upsideDown = 180
    case right = 270
}
```

Le enumerazioni senza alcun tipo specifico non espongono la proprietà rawValue

```
enum Rotation {
    case up
    case right
    case down
    case left
}

let foo = Rotation.up
foo.rawValue //error
```

Si presuppone che i valori grezzi interi inizino a 0 e aumentino monotonicamente:

```
enum MetasyntacticVariable: Int {
    case foo  // rawValue is automatically 0
```

```
    case bar  // rawValue is automatically 1
    case baz = 7
    case quux  // rawValue is automatically 8
}
```

I valori raw delle stringhe possono essere sintetizzati automaticamente:

```
enum MarsMoon: String {
    case phobos  // rawValue is automatically "phobos"
    case deimos  // rawValue is automatically "deimos"
}
```

Un enum con valore grezzo si adatta automaticamente a RawRepresentable . Puoi ottenere il valore grezzo corrispondente di un valore enum con .rawValue :

```
func rotate(rotation: Rotation) {
    let degrees = rotation.rawValue
    ...
}
```

Puoi anche creare un enum *da* un valore grezzo usando init?(rawValue:) :

```
let rotation = Rotation(rawValue: 0)  // returns Rotation.Up
let otherRotation = Rotation(rawValue: 45)  // returns nil (there is no Rotation with rawValue
45)

if let moon = MarsMoon(rawValue: str) {
    print("Mars has a moon named \(str)")
} else {
    print("Mars doesn't have a moon named \(str)")
}
```

Se si desidera ottenere il valore hash di un enum specifico, è possibile accedere al suo hashValue, il valore hash restituirà l'indice dell'enumerazione a partire da zero.

```
let quux = MetasyntacticVariable(rawValue: 8)// rawValue is 8
quux?.hashValue //hashValue is 3
```

inizializzatori

Le enumerazioni possono avere metodi di inizializzazione personalizzati che possono essere più utili init?(rawValue:) default init?(rawValue:) . Le enumerazioni possono anche memorizzare valori. Questo può essere utile per memorizzare i valori con cui sono stati inizializzati e recuperare quel valore in seguito.

```
enum CompassDirection {
    case north(Int)
    case south(Int)
    case east(Int)
    case west(Int)

    init?(degrees: Int) {
        switch degrees {
        case 0...45:
            self = .north(degrees)
        case 46...135:
            self = .east(degrees)
        case 136...225:
```

```
            self = .south(degrees)
        case 226...315:
            self = .west(degrees)
        case 316...360:
            self = .north(degrees)
        default:
            return nil
        }
    }

    var value: Int = {
        switch self {
            case north(let degrees):
                return degrees
            case south(let degrees):
                return degrees
            case east(let degrees):
                return degrees
            case west(let degrees):
                return degrees
        }
    }
}
```

Usando quell'inizializzatore possiamo fare questo:

```
var direction = CompassDirection(degrees: 0) // Returns CompassDirection.north
direction = CompassDirection(degrees: 90) // Returns CompassDirection.east
print(direction.value) //prints 90
direction = CompassDirection(degrees: 500) // Returns nil
```

Le enumerazioni condividono molte funzionalità con classi e strutture

Enums in Swift sono molto più potenti di alcune delle loro controparti in altre lingue, come C. Condividono molte funzionalità con classi e strutture , come la definizione di inizializzatori , proprietà calcolate , metodi di istanza , conformità e estensioni del protocollo .

```
protocol ChangesDirection {
    mutating func changeDirection()
}

enum Direction {

    // enumeration cases
    case up, down, left, right

    // initialise the enum instance with a case
    // that's in the opposite direction to another
    init(oppositeTo otherDirection: Direction) {
        self = otherDirection.opposite
    }

    // computed property that returns the opposite direction
    var opposite: Direction {
        switch self {
        case .up:
            return .down
        case .down:
            return .up
        case .left:
```

```
            return .right
        case .right:
            return .left
        }
    }
}

// extension to Direction that adds conformance to the ChangesDirection protocol
extension Direction: ChangesDirection {
    mutating func changeDirection() {
        self = .left
    }
}
```

```
var dir = Direction(oppositeTo: .down) // Direction.up

dir.changeDirection() // Direction.left

let opposite = dir.opposite // Direction.right
```

Enumerazioni nidificate

È possibile nidificare le enumerazioni una dentro l'altra, ciò consente di strutturare le enumerazioni gerarchiche per essere più organizzate e chiare.

```
enum Orchestra {
    enum Strings {
        case violin
        case viola
        case cello
        case doubleBasse
    }

    enum Keyboards {
        case piano
        case celesta
        case harp
    }

    enum Woodwinds {
        case flute
        case oboe
        case clarinet
        case bassoon
        case contrabassoon
    }
}
```

E puoi usarlo in questo modo:

```
let instrment1 = Orchestra.Strings.viola
let instrment2 = Orchestra.Keyboards.piano
```

Leggi Enums online: https://riptutorial.com/it/swift/topic/224/enums

Capitolo 21: estensioni

Osservazioni

Puoi leggere ulteriori informazioni sulle estensioni in The Swift Programming Language .

Examples

Variabili e funzioni

Le estensioni possono contenere funzioni e variabili di calcolo calcolate / costanti. Sono nel formato

```
extension ExtensionOf {
    //new functions and get-variables
}
```

Per fare riferimento all'istanza dell'oggetto esteso, è possibile utilizzare self , proprio come potrebbe essere utilizzato

Per creare un'estensione di String che aggiunge una funzione .length() che restituisce la lunghezza della stringa, ad esempio

```
extension String {
    func length() -> Int {
        return self.characters.count
    }
}

"Hello, World!".length() // 13
```

Le estensioni possono contenere anche variabili get . Ad esempio, aggiungendo una variabile .length alla stringa che restituisce la lunghezza della stringa

```
extension String {
    var length: Int {
        get {
            return self.characters.count
        }
    }
}

"Hello, World!".length // 13
```

Inizializzatori in estensioni

Le estensioni possono contenere gli inizializzatori di convenienza. Ad esempio, un inizializzatore disponibile per Int che accetta una NSString :

```
extension Int {
    init?(_ string: NSString) {
        self.init(string as String)  // delegate to the existing Int.init(String) initializer
    }
}

let str1: NSString = "42"
Int(str1) // 42
```

```
let str2: NSString = "abc"
Int(str2) // nil
```

Quali sono le estensioni?

Le estensioni vengono utilizzate per estendere la funzionalità dei tipi esistenti in Swift. Le estensioni possono aggiungere pedici, funzioni, inizializzatori e proprietà calcolate. Possono anche rendere i tipi conformi ai protocolli .

Supponiamo che tu voglia essere in grado di calcolare il fattoriale di un Int . È possibile aggiungere una proprietà calcolata in un'estensione:

```
extension Int {
    var factorial: Int {
        return (1..<self+1).reduce(1, combine: *)
    }
}
```

Quindi puoi accedere alla proprietà come se fosse stata inclusa nell'API Int originale.

```
let val1: Int = 10

val1.factorial  // returns 3628800
```

Estensioni di protocollo

Una caratteristica molto utile di Swift 2.2 è la possibilità di estendere i protocolli.

Funziona praticamente come le classi astratte quando si tratta di una funzionalità che si desidera essere disponibile in tutte le classi che implementa un protocollo (senza dover ereditare da una classe comune di base).

```
protocol FooProtocol {
    func doSomething()
}

extension FooProtocol {
    func doSomething() {
        print("Hi")
    }
}

class Foo: FooProtocol {
    func myMethod() {
        doSomething() // By just implementing the protocol this method is available
    }
}
```

Questo è anche possibile usando i generici.

restrizioni

È possibile scrivere un metodo su un tipo generico che è più restrittivo usando la frase.

```
extension Array where Element: StringLiteralConvertible {
  func toUpperCase() -> [String] {
```

```
      var result = [String]()
      for value in self {
          result.append(String(value).uppercaseString)
      }
      return result
    }
}
```

Esempio di utilizzo

```
let array = ["a","b","c"]
let resultado = array.toUpperCase()
```

Quali sono le estensioni e quando usarle

Le estensioni aggiungono nuove funzionalità a una classe esistente, struttura, enumerazione o tipo di protocollo. Ciò include la possibilità di estendere i tipi per i quali non si ha accesso al codice sorgente originale.

Le estensioni in Swift possono:

- Aggiungi proprietà calcolate e proprietà del tipo calcolato
- Definire metodi di istanza e metodi di tipo
- Fornire nuovi inizializzatori
- Definisci pedici
- Definire e utilizzare nuovi tipi nidificati
- Rendere un tipo esistente conforme a un protocollo

Quando usare Swift Extensions:

- Funzionalità aggiuntive per Swift
- Funzionalità aggiuntive per UIKit / Foundation
- Funzionalità aggiuntive senza incasinare il codice di altre persone
- Classi di suddivisione in: Dati / Funzionalità / Delegato

Quando non usare:

- Estendi le tue classi da un altro file

Semplice esempio:

```
extension Bool {
    public mutating func toggle() -> Bool {
        self = !self
        return self
    }
}

var myBool: Bool = true
print(myBool.toggle()) // false
```

fonte

pedici

Le estensioni possono aggiungere nuove iscrizioni a un tipo esistente.

Questo esempio ottiene il carattere all'interno di una stringa utilizzando l'indice specificato:

2.2

```
extension String {
    subscript(index: Int) -> Character {
        let newIndex = startIndex.advancedBy(index)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

3.0

```
extension String {
    subscript(offset: Int) -> Character {
        let newIndex = self.index(self.startIndex, offsetBy: offset)
        return self[newIndex]
    }
}

var myString = "StackOverFlow"
print(myString[2]) // a
print(myString[3]) // c
```

Leggi estensioni online: https://riptutorial.com/it/swift/topic/324/estensioni

Capitolo 22: Funzione come cittadini di prima classe in Swift

introduzione

Funziona come un membro di prima classe significa che può godere dei privilegi proprio come fanno gli oggetti. Può essere assegnato a una variabile, passato a una funzione come parametro o può essere usato come tipo di ritorno.

Examples

Assegnazione di una funzione a una variabile

```
struct Mathematics
{
    internal func performOperation(inputArray: [Int], operation: (Int)-> Int)-> [Int]
    {
        var processedArray = [Int]()

        for item in inputArray
        {
            processedArray.append(operation(item))
        }

        return processedArray
    }


    internal func performComplexOperation(valueOne: Int)-> ((Int)-> Int)
    {
        return
            ({
                 return valueOne + $0
            })
    }

}


let arrayToBeProcessed = [1,3,5,7,9,11,8,6,4,2,100]

let math = Mathematics()

func add2(item: Int)-> Int
{
    return (item + 2)
}

// assigning the function to a variable and then passing it to a function as param
let add2ToMe = add2
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2ToMe))
```

Produzione:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Allo stesso modo, quanto sopra potrebbe essere ottenuto usando una `closure`

```
// assigning the closure to a variable and then passing it to a function as param
```

```
let add2 = {(item: Int)-> Int in return item + 2}
print(math.performOperation(inputArray: arrayToBeProcessed, operation: add2))
```

Passaggio della funzione come argomento a un'altra funzione, creando così una funzione ordine superiore

```
func multiply2(item: Int)-> Int
{
    return (item + 2)
}


let multiply2ToMe = multiply2

// passing the function directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: multiply2ToMe))
```

Produzione:

```
[3, 5, 7, 9, 11, 13, 10, 8, 6, 4, 102]
```

Allo stesso modo, quanto sopra potrebbe essere ottenuto usando una closure

```
// passing the closure directly to the function as param
print(math.performOperation(inputArray: arrayToBeProcessed, operation: { $0 * 2 }))
```

Funziona come tipo di ritorno da un'altra funzione

```
// function as return type
print(math.performComplexOperation(valueOne: 4)(5))
```

Produzione:

```
9
```

Leggi Funzione come cittadini di prima classe in Swift online:
https://riptutorial.com/it/swift/topic/8618/funzione-come-cittadini-di-prima-classe-in-swift

Capitolo 23: funzioni

Examples

Uso di base

Le funzioni possono essere dichiarate senza parametri o un valore di ritorno. L'unica informazione richiesta è un nome (hello in questo caso).

```
func hello()
{
    print("Hello World")
}
```

Chiama una funzione senza parametri scrivendo il suo nome seguito da una coppia vuota di parentesi.

```
hello()
//output: "Hello World"
```

Funziona con i parametri

Le funzioni possono assumere parametri in modo che la loro funzionalità possa essere modificata. I parametri sono indicati come una lista separata da virgole con i loro tipi e nomi definiti.

```
func magicNumber(number1: Int)
{
    print("\(number1) Is the magic number")
}
```

Nota: la sintassi \(number1) è \(number1) base della stringa e viene utilizzata per inserire il numero intero nella stringa.

Le funzioni con i parametri vengono chiamate specificando la funzione in base al nome e fornendo un valore di input del tipo utilizzato nella dichiarazione della funzione.

```
magicNumber(5)
//output: "5 Is the magic number
let example: Int = 10
magicNumber(example)
//output: "10 Is the magic number"
```

Potrebbe essere stato utilizzato qualsiasi valore di tipo Int.

```
func magicNumber(number1: Int, number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}
```

Quando una funzione utilizza più parametri, il nome del primo parametro non è richiesto per il primo ma è sui parametri successivi.

```
let ten: Int = 10
let five: Int = 5
magicNumber(ten,number2: five)
//output: "15 Is the magic number"
```

Utilizzare i nomi dei parametri esterni per rendere più leggibili le chiamate di funzione.

```
func magicNumber(one number1: Int, two number2: Int)
{
    print("\(number1 + number2) Is the magic number")
}

let ten: Int = 10
let five: Int = 5
magicNumber(one: ten, two: five)
```

L'impostazione del valore predefinito nella dichiarazione della funzione consente di chiamare la funzione senza fornire alcun valore di input.

```
func magicNumber(one number1: Int = 5, two number2: Int = 10)
{
    print("\(number1 + number2) Is the magic number")
}

magicNumber()
//output: "15 Is the magic number"
```

Valori di ritorno

Le funzioni possono restituire valori specificando il tipo dopo l'elenco di parametri.

```
func findHypotenuse(a: Double, b: Double) -> Double
{
    return sqrt((a * a) + (b * b))
}

let c = findHypotenuse(3, b: 5)
//c = 5.830951894845301
```

Le funzioni possono anche restituire più valori usando tuple.

```
func maths(number: Int) -> (times2: Int, times3: Int)
{
    let two = number * 2
    let three = number * 3
    return (two, three)
}
let resultTuple = maths(5)
//resultTuple = (10, 15)
```

Errori di lancio

Se si desidera che una funzione sia in grado di generare errori, è necessario aggiungere la parola chiave throws dopo le parentesi che contengono gli argomenti:

```
func errorThrower()throws -> String {}
```

Quando vuoi generare un errore, usa la parola chiave throw :

```
func errorThrower()throws -> String {
  if true {
    return "True"
```

```
  } else {
    // Throwing an error
    throw Error.error
  }
}
```

Se si desidera chiamare una funzione che può generare un errore, è necessario utilizzare la parola chiave try in un blocco do :

```
do {
  try errorThrower()
}
```

Per ulteriori informazioni sugli errori Swift: Errori

metodi

I metodi di istanza sono funzioni che appartengono a istanze di un tipo in Swift (una classe , una struttura , un'enumerazione o un protocollo). **I metodi Type** vengono chiamati su un tipo stesso.

Metodi di istanza

I metodi di istanza sono definiti con una dichiarazione func all'interno della definizione del tipo o in un'estensione .

```
class Counter {
    var count = 0
    func increment() {
        count += 1
    }
}
```

Il metodo di istanza increment() viene chiamato su un'istanza della classe Counter :

```
let counter = Counter()  // create an instance of Counter class
counter.increment()      // call the instance method on this instance
```

Digitare metodi

I metodi di tipo sono definiti con le parole chiave static func . (Per le classi, la class func definisce un metodo di tipo che può essere sovrascritto dalle sottoclassi).

```
class SomeClass {
    class func someTypeMethod() {
        // type method implementation goes here
    }
}
```

```
SomeClass.someTypeMethod()  // type method is called on the SomeClass type itself
```

Inout Parameters

Le funzioni possono modificare i parametri passati a loro se sono contrassegnati con la parola chiave inout . Quando si passa un parametro inout a una funzione, il chiamante deve aggiungere un & alla variabile che si sta passando.

```
func updateFruit(fruit: inout Int) {
    fruit -= 1
}

var apples = 30 // Prints "There's 30 apples"
print("There's \(apples) apples")

updateFruit(fruit: &apples)

print("There's now \(apples) apples") // Prints "There's 29 apples".
```

Ciò consente di applicare la semantica di riferimento a tipi che normalmente hanno una semantica del valore.

Sintassi della chiusura finale

Quando l'ultimo parametro di una funzione è una chiusura

```
func loadData(id: String, completion:(result: String) -> ()) {
    // ...
    completion(result:"This is the result data")
}
```

la funzione può essere richiamata utilizzando la sintassi della chiusura del trailing

```
loadData("123") { result in
    print(result)
}
```

Gli operatori sono funzioni

Operatori come + , - , ?? sono un tipo di funzione denominata utilizzando simboli anziché lettere. Sono invocati diversamente dalle funzioni:

- Prefisso: **-** x
- Infix: x **+** y
- Postfix: x **++**

Puoi leggere ulteriori informazioni sugli operatori di base e sugli operatori avanzati in Swift Programming Language.

Parametri Variadici

A volte, non è possibile elencare il numero di parametri di cui una funzione potrebbe avere bisogno. Considera una funzione sum :

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

Questo funziona bene per trovare la somma di due numeri, ma per trovare la somma di tre dovremmo scrivere un'altra funzione:

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

e uno con quattro parametri avrebbe bisogno di un altro, e così via. Swift rende possibile

definire una funzione con un numero variabile di parametri usando una sequenza di tre periodi: ... Per esempio,

```
func sum(_ numbers: Int...) -> Int {
    return numbers.reduce(0, combine: +)
}
```

Si noti come il parametro numbers , che è variadico, è fuso in una singola Array di tipo [Int] . Questo è vero in generale, i parametri variadici di tipo T... sono accessibili come [T] .

Questa funzione può ora essere chiamata così:

```
let a = sum(1, 2) // a == 3
let b = sum(3, 4, 5, 6, 7) // b == 25
```

Un parametro variadico in Swift non deve venire alla fine della lista dei parametri, ma può essercene uno solo in ogni firma di funzione.

A volte, è conveniente inserire una dimensione minima sul numero di parametri. Ad esempio, non ha senso prendere la sum di nessun valore. Un modo semplice per far rispettare questo è mettere alcuni parametri richiesti non-variabili e quindi aggiungere il parametro variadic dopo. Per essere sicuri che la sum possa essere chiamata solo con almeno due parametri, possiamo scrivere

```
func sum(_ n1: Int, _ n2: Int, _ numbers: Int...) -> Int {
    return numbers.reduce(n1 + n2, combine: +)
}

sum(1, 2) // ok
sum(3, 4, 5, 6, 7) // ok
sum(1) // not ok
sum() // not ok
```

pedici

Classi, strutture ed enumerazioni possono definire pedici, che sono scorciatoie per accedere agli elementi membri di una raccolta, un elenco o una sequenza.

Esempio

```
struct DaysOfWeek {

  var days = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"]

  subscript(index: Int) -> String {
    get {
      return days[index]
    }
    set {
      days[index] = newValue
    }
  }
}
```

Uso degli abbonati

```
var week = DaysOfWeek()
//you access an element of an array at index by array[index].
debugPrint(week[1])
debugPrint(week[0])
```

```
week[0] = "Sunday"
debugPrint(week[0])
```

Opzioni di iscrizione:

Gli indici di sottoscrizione possono prendere qualsiasi numero di parametri di input e questi parametri di input possono essere di qualsiasi tipo. Gli abbonati possono anche restituire qualsiasi tipo. Gli indici di sottoscrizione possono utilizzare parametri variabili e parametri variadici, ma non possono utilizzare parametri in-out o fornire valori di parametro predefiniti.

Esempio:

```
struct Food {

  enum MealTime {
    case Breakfast, Lunch, Dinner
  }

  var meals: [MealTime: String] = [:]

  subscript (type: MealTime) -> String? {
    get {
      return meals[type]
    }
    set {
      meals[type] = newValue
    }
  }
}
```

uso

```
var diet = Food()
diet[.Breakfast] = "Scrambled Eggs"
diet[.Lunch] = "Rice"

debugPrint("I had \(diet[.Breakfast]) for breakfast")
```

Funzioni con chiusure

L'uso di funzioni che accolgono ed eseguono chiusure può essere estremamente utile per inviare un blocco di codice da eseguire altrove. Possiamo iniziare consentendo alla nostra funzione di accettare una chiusura opzionale che (in questo caso) restituirà Void .

```
func closedFunc(block: (()->Void)? = nil) {
    print("Just beginning")

    if let block = block {
        block()
    }
}
```

Ora che la nostra funzione è stata definita, chiamiamola e passa un codice:

```
closedFunc() { Void in
    print("Over already")
}
```

Usando una **chiusura finale** con la nostra chiamata di funzione, possiamo passare il codice (in questo caso, print) per essere eseguito in qualche punto all'interno della nostra funzione closedFunc() .

Il registro dovrebbe stampare:

Appena iniziato

Già finito

Un caso d'uso più specifico di questo potrebbe includere l'esecuzione di codice tra due classi:

```
class ViewController: UIViewController {

    override func viewDidLoad() {
        let _  = A.init(){Void in self.action(2)}
    }

    func action(i: Int) {
        print(i)
    }
}


class A: NSObject {
    var closure : ()?

    init(closure: (()->Void)? = nil) {
        // Notice how this is executed before the  closure
        print("1")
        // Make sure closure isn't nil
        self.closure = closure?()
    }
}
```

Il registro dovrebbe stampare:

1

2

Passare e restituire funzioni

La seguente funzione restituisce come risultato un'altra funzione che può essere successivamente assegnata a una variabile e chiamata:

```
func jediTrainer () -> ((String, Int) -> String) {
  func train(name: String, times: Int) -> (String) {
    return "\(name) has been trained in the Force \(times) times"
  }
  return train
}

let train = jediTrainer()
train("Obi Wan", 3)
```

Tipi di funzioni

Ogni funzione ha un proprio tipo di funzione, costituito dai tipi di parametri e dal tipo di ritorno della funzione stessa. Ad esempio la seguente funzione:

```
func sum(x: Int, y: Int) -> (result: Int) { return x + y }
```

ha una funzione tipo di:

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

I tipi di funzione possono quindi essere utilizzati come tipi di parametri o come tipi di ritorno per le funzioni di nidificazione.

Leggi funzioni online: https://riptutorial.com/it/swift/topic/432/funzioni

Capitolo 24: Funzioni di Swift Advance

introduzione

Le funzioni avanzate come map , flatMap , filter e reduce vengono utilizzate per operare su vari tipi di raccolta come Array e Dictionary. Le funzioni avanzate richiedono in genere un codice ridotto e possono essere concatenate per creare una logica complessa in modo conciso.

Examples

Introduzione con funzioni avanzate

Prendiamo uno scenario per capire le funzioni avanzate in modo migliore,

```
struct User {
    var name: String
    var age: Int
    var country: String?
}

//User's information
let user1 = User(name: "John", age: 24, country: "USA")
let user2 = User(name: "Chan", age: 20, country: nil)
let user3 = User(name: "Morgan", age: 30, country: nil)
let user4 = User(name: "Rachel", age: 20, country: "UK")
let user5 = User(name: "Katie", age: 23, country: "USA")
let user6 = User(name: "David", age: 35, country: "USA")
let user7 = User(name: "Bob",age: 22, country: nil)

//User's array list
let arrUser = [user1, user2, user3, user4, user5, user6, user7]
```

Funzione mappa:

Utilizza la mappa per eseguire il loop su una raccolta e applicare la stessa operazione a ciascun elemento della raccolta. La funzione mappa restituisce una matrice contenente i risultati dell'applicazione di una funzione di mappatura o trasformazione a ciascun elemento.

```
//Fetch all the user's name from array
let arrUserName = arrUser.map({ $0.name }) // ["John", "Chan", "Morgan", "Rachel", "Katie",
"David", "Bob"]
```

Funzione Flat-Map:

L'uso più semplice è come suggerisce il nome per appiattire una collezione di collezioni.

```
// Fetch all user country name & ignore nil value.
let arrCountry = arrUser.flatMap({ $0.country }) // ["USA", "UK", "USA", "USA"]
```

Funzione filtro:

Utilizza il filtro per eseguire il loop su una raccolta e restituire una matrice contenente solo quegli elementi che corrispondono a una condizione di inclusione.

```
// Filtering USA user from the array user list.
let arrUSAUsers = arrUser.filter({ $0.country == "USA" }) // [user1, user5, user6]

// User chaining methods to fetch user's name who live in USA
let arrUserList = arrUser.filter({ $0.country == "USA" }).map({ $0.name }) // ["John",
```

```
"Katie", "David"]
```

Ridurre:

Usa riduci per combinare tutti gli elementi di una raccolta per creare un nuovo valore singolo.

Swift 2.3: -

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, combine: { $0 + $1 }) //174

//Prepare all user name string with seperated by comma
let strUserName = arrUserName.reduce("", combine: { $0 == "" ? $1 : $0 + ", " + $1 }) // John,
Chan, Morgan, Rachel, Katie, David, Bob
```

Swift 3: -

```
//Fetch user's total age.
let arrUserAge = arrUser.map({ $0.age }).reduce(0, { $0 + $1 }) //174

//Prepare all user name string with seperated by comma
let strUserName = arrUserName.reduce("", { $0 == "" ? $1 : $0 + ", " + $1 }) // John, Chan,
Morgan, Rachel, Katie, David, Bob
```

Appiattisci array multidimensionale

Per appiattire la matrice multidimensionale in una singola dimensione, vengono utilizzate le funzioni avanzate di FlatMap. Altro caso d'uso è quello di trascurare il valore nullo da valori di matrice e mappatura. Vediamo con esempio: -

Supponiamo di disporre di una serie di città multidimensionale e di voler ordinare l'elenco dei nomi delle città in ordine crescente. In tal caso, possiamo usare la funzione flatMap come: -

```
let arrStateName = [["Alaska", "Iowa", "Missouri", "New Mexico"], ["New York", "Texas",
"Washington", "Maryland"], ["New Jersey", "Virginia", "Florida", "Colorado"]]
```

Preparare una lista monodimensionale dall'array multidimensionale,

```
let arrFlatStateList = arrStateName.flatMap({ $0 }) // ["Alaska", "Iowa", "Missouri", "New
Mexico", "New York", "Texas", "Washington", "Maryland", "New Jersey", "Virginia", "Florida",
"Colorado"]
```

Per ordinare i valori dell'array, possiamo usare l'operazione di concatenamento o ordinare l'array di appiattimento. Qui sotto l'esempio mostra l'operazione di concatenamento,

```
// Swift 2.3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sort(<) // ["Alaska",     "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]

// Swift 3 syntax
let arrSortedStateList = arrStateName.flatMap({ $0 }).sorted(by: <) // ["Alaska", "Colorado",
"Florida", "Iowa", "Maryland", "Missouri", "New Jersey", "New Mexico", "New York", "Texas",
"Virginia", "Washington"]
```

Leggi Funzioni di Swift Advance online: https://riptutorial.com/it/swift/topic/9279/funzioni-di-swift-advance

Capitolo 25: Genera UIImmagine delle iniziali dalla stringa

introduzione

Questa è una classe che genererà una UIImmagine delle iniziali di una persona. Harry Potter genererebbe un'immagine di HP.

Examples

InitialsImageFactory

```
class InitialsImageFactory: NSObject {

class func imageWith(name: String?) -> UIImage? {

let frame = CGRect(x: 0, y: 0, width: 50, height: 50)
let nameLabel = UILabel(frame: frame)
nameLabel.textAlignment = .center
nameLabel.backgroundColor = .lightGray
nameLabel.textColor = .white
nameLabel.font = UIFont.boldSystemFont(ofSize: 20)
var initials = ""

if let initialsArray = name?.components(separatedBy: " ") {

  if let firstWord = initialsArray.first {
    if let firstLetter = firstWord.characters.first {
      initials += String(firstLetter).capitalized
    }

  }
  if initialsArray.count > 1, let lastWord = initialsArray.last {
    if let lastLetter = lastWord.characters.first {
      initials += String(lastLetter).capitalized
    }

  }
} else {
  return nil
}

nameLabel.text = initials
UIGraphicsBeginImageContext(frame.size)
if let currentContext = UIGraphicsGetCurrentContext() {
  nameLabel.layer.render(in: currentContext)
  let nameImage = UIGraphicsGetImageFromCurrentImageContext()
  return nameImage
}
return nil
}

}
```

Leggi Genera UIImmagine delle iniziali dalla stringa online:
https://riptutorial.com/it/swift/topic/10915/genera-uiimmagine-delle-iniziali-dalla-stringa

Capitolo 26: Generics

Osservazioni

Il codice generico consente di scrivere funzioni e tipi flessibili e riutilizzabili che possono funzionare con qualsiasi tipo, in base ai requisiti definiti dall'utente. È possibile scrivere codice che evita la duplicazione ed esprime il suo intento in modo chiaro e astratto.

I generici sono una delle funzionalità più potenti di Swift e gran parte della libreria standard di Swift è costruita con codice generico. Ad esempio, i tipi di Array e Dictionary di Swift sono entrambi raccolte generiche. È possibile creare una matrice che contiene valori Int o una matrice che contiene valori String , o in effetti una matrice per qualsiasi altro tipo che può essere creato in Swift. Allo stesso modo, è possibile creare un dizionario per memorizzare valori di qualsiasi tipo specificato e non ci sono limitazioni su cosa possa essere quel tipo.

Fonte: Apple Swift Programming Language

Examples

Vincolare i tipi di segnaposto generici

È possibile forzare i parametri di tipo di una classe generica per implementare un protocollo , ad esempio, Equatable

```
class MyGenericClass<Type: Equatable>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func valueEquals(anotherValue: Type) -> Bool{
        return self.value == anotherValue
    }
}
```

Ogni volta che creiamo una nuova MyGenericClass , il parametro type deve implementare il protocollo Equatable (assicurando che il parametro type possa essere confrontato con un'altra variabile dello stesso tipo usando ==)

```
let myFloatGeneric = MyGenericClass<Double>(value: 2.71828) // valid
let myStringGeneric = MyGenericClass<String>(value: "My String") // valid

// "Type [Int] does not conform to protocol 'Equatable'"
let myInvalidGeneric = MyGenericClass<[Int]>(value: [2])

let myIntGeneric = MyGenericClass<Int>(value: 72)
print(myIntGeneric.valueEquals(72)) // true
print(myIntGeneric.valueEquals(-274)) // false

// "Cannot convert value of type 'String' to expected argument type 'Int'"
print(myIntGeneric.valueEquals("My String"))
```

Le basi di Generics

I generici sono segnaposti per i tipi, consentendo di scrivere codice flessibile che può essere applicato su più tipi. Il vantaggio dell'utilizzo di generici su Any è che consentono ancora al compilatore di rafforzare la sicurezza dei caratteri.

Un segnaposto generico è definito tra parentesi angolari <> .

Funzioni generiche

Per le funzioni , questo segnaposto viene posizionato dopo il nome della funzione:

```
/// Picks one of the inputs at random, and returns it
func pickRandom<T>(_ a:T, _ b:T) -> T {
    return arc4random_uniform(2) == 0 ? a : b
}
```

In questo caso, il segnaposto generico è T Quando si arriva a chiamare la funzione, Swift può dedurre il tipo di T per te (in quanto agisce semplicemente come segnaposto per un tipo effettivo).

```
let randomOutput = pickRandom(5, 7) // returns an Int (that's either 5 or 7)
```

Qui stiamo passando due interi alla funzione. Quindi Swift sta inferendo T == Int - quindi la firma della funzione è dedotta per essere (Int, Int) -> Int .

A causa della forte sicurezza del tipo offerta dai generici, sia gli argomenti che il ritorno della funzione devono essere dello *stesso* tipo. Pertanto quanto segue non verrà compilato:

```
struct Foo {}

let foo = Foo()

let randomOutput = pickRandom(foo, 5) // error: cannot convert value of type 'Int' to expected
argument type 'Foo'
```

Tipi generici

Per utilizzare i generici con classi , strutture o enumerazioni , è possibile definire il segnaposto generico dopo il nome del tipo.

```
class Bar<T> {
    var baz : T

    init(baz:T) {
        self.baz = baz
    }
}
```

Questo segnaposto generico richiede un tipo quando si usa la classe Bar . In questo caso, può essere dedotto dall'inizializzatore init(baz:T) .

```
let bar = Bar(baz: "a string") // bar's type is Bar<String>
```

Qui il segnaposto generico T è dedotto per essere di tipo String , creando così un'istanza Bar<String> . Puoi anche specificare il tipo esplicitamente:

```
let bar = Bar<String>(baz: "a string")
```

Se utilizzato con un tipo, il segnaposto generico specificato manterrà il suo tipo per l'intera durata dell'istanza data e non potrà essere modificato dopo l'inizializzazione. Pertanto, quando si accede alla proprietà baz , sarà sempre di tipo String per questa determinata istanza.

```
let str = bar.baz // of type String
```

Passando intorno ai tipi generici

Quando si arriva a passare tipi generici, nella maggior parte dei casi è necessario essere espliciti sul tipo di segnaposto generico che ci si aspetta. Ad esempio, come input di una funzione:

```
func takeABarInt(bar:Bar<Int>) {
    ...
}
```

Questa funzione accetta solo una Bar<Int> . Il tentativo di passare in un'istanza Bar cui il tipo generico di segnaposto non è Int causerà un errore del compilatore.

Denominazione generica del segnaposto

I nomi generici dei segnaposto non si limitano alle sole lettere singole. Se un determinato segnaposto rappresenta un concetto significativo, dovresti dargli un nome descrittivo. Ad esempio, Swift's Array ha un segnaposto generico chiamato Element , che definisce il tipo di elemento di una data istanza di Array .

```
public struct Array<Element> : RandomAccessCollection, MutableCollection {
    ...
}
```

Esempi di classi generiche

Una classe generica con il parametro type Type

```
class MyGenericClass<Type>{

    var value: Type
    init(value: Type){
        self.value = value
    }

    func getValue() -> Type{
        return self.value
    }

    func setValue(value: Type){
        self.value = value
    }
}
```

Ora possiamo creare nuovi oggetti usando un parametro di tipo

```
let myStringGeneric = MyGenericClass<String>(value: "My String Value")
let myIntGeneric = MyGenericClass<Int>(value: 42)

print(myStringGeneric.getValue()) // "My String Value"
print(myIntGeneric.getValue()) // 42

myStringGeneric.setValue("Another String")
```

```
myIntGeneric.setValue(1024)

print(myStringGeneric.getValue()) // "Another String"
print(myIntGeneric.getValue()) // 1024
```

Generics può anche essere creato con più parametri di tipo

```
class AnotherGenericClass<TypeOne, TypeTwo, TypeThree>{

    var value1: TypeOne
    var value2: TypeTwo
    var value3: TypeThree
    init(value1: TypeOne, value2: TypeTwo, value3: TypeThree){
        self.value1 = value1
        self.value2 = value2
        self.value3 = value3
    }

    func getValueOne() -> TypeOne{return self.value1}
    func getValueTwo() -> TypeTwo{return self.value2}
    func getValueThree() -> TypeThree{return self.value3}
}
```

E usato allo stesso modo

```
let myGeneric = AnotherGenericClass<String, Int, Double>(value1: "Value of pi", value2: 3,
value3: 3.14159)

print(myGeneric.getValueOne() is String) // true
print(myGeneric.getValueTwo() is Int) // true
print(myGeneric.getValueThree() is Double) // true
print(myGeneric.getValueTwo() is String) // false

print(myGeneric.getValueOne()) // "Value of pi"
print(myGeneric.getValueTwo()) // 3
print(myGeneric.getValueThree()) // 3.14159
```

Eredità di classe generica

Le classi generiche possono essere ereditate:

```
// Models
class MyFirstModel {
}

class MySecondModel: MyFirstModel {
}

// Generic classes
class MyFirstGenericClass<T: MyFirstModel> {

    func doSomethingWithModel(model: T) {
        // Do something here
    }

}

class MySecondGenericClass<T: MySecondModel>: MyFirstGenericClass<T> {
```

```
    override func doSomethingWithModel(model: T) {
        super.doSomethingWithModel(model)

        // Do more things here
    }

}
```

Utilizzo di Generics per semplificare le funzioni di array

Una funzione che estende la funzionalità dell'array creando una funzione di rimozione orientata agli oggetti.

```
// Need to restrict the extension to elements that can be compared.
// The `Element` is the generics name defined by Array for its item types.
// This restriction also gives us access to `index(of:_)` which is also
// defined in an Array extension with `where Element: Equatable`.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        if let index = self.index(of: element ) {
            self.remove(at: index)
        } else {
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
    }
}
```

uso

```
var myArray = [1,2,3]
print(myArray)

// Prints [1,2,3]
```

Utilizzare la funzione per rimuovere un elemento senza necessità di un indice. Basta passare l'oggetto per rimuovere.

```
myArray.remove(2)
print(myArray)

// Prints [1,3]
```

Usa i generici per migliorare la sicurezza del tipo

Prendiamo questo esempio senza usare i generici

```
protocol JSONDecodable {
    static func from(_ json: [String: Any]) -> Any?
}
```

La dichiarazione del protocollo sembra soddisfacente a meno che tu non la usi effettivamente.

```
let myTestObject = TestObject.from(myJson) as? TestObject
```

Perché devi TestObject il risultato a TestObject ? A causa del tipo Any ritorno nella dichiarazione del protocollo.

Usando i generici puoi evitare questo problema che può causare errori di runtime (e non vogliamo averli!)

```
protocol JSONDecodable {
    associatedtype Element
    static func from(_ json: [String: Any]) -> Element?
}

struct TestObject: JSONDecodable {
    static func from(_ json: [String: Any]) -> TestObject? {
    }
}

let testObject = TestObject.from(myJson) // testObject is now automatically `TestObject?`
```

Vincoli di tipo avanzato

E 'possibile specificare numerosi vincoli di tipo per i medicinali generici che utilizzano la where clausola:

```
func doSomething<T where T: Comparable, T: Hashable>(first: T, second: T) {
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

E 'valido anche a scrivere la where clausola dopo la lista degli argomenti:

```
func doSomething<T>(first: T, second: T) where T: Comparable, T: Hashable {
    // Access hashable function
    guard first.hashValue == second.hashValue else {
        return
    }
    // Access comparable function
    if first == second {
        print("\(first) and \(second) are equal.")
    }
}
```

Le estensioni possono essere limitate a tipi che soddisfano le condizioni. La funzione è disponibile solo per le istanze che soddisfano le condizioni del tipo:

```
// "Element" is the generics type defined by "Array". For this example, we
// want to add a function that requires that "Element" can be compared, that
// is: it needs to adhere to the Equatable protocol.
public extension Array where Element: Equatable {
    /// Removes the given object from the array.
    mutating func remove(_ element: Element) {
        // We could also use "self.index(of: element)" here, as "index(of:_)"
        // is also defined in an extension with "where Element: Equatable".
        // For the sake of this example, explicitly make use of the Equatable.
        if let index = self.index(where: { $0 == element }) {
            self.remove(at: index)
        } else {
```

```
            fatalError("Removal error, no such element:\"\(element)\" in array.\n")
        }
    }
}
```

Leggi Generics online: https://riptutorial.com/it/swift/topic/774/generics

Capitolo 27: Gestione degli errori

Osservazioni

Per ulteriori informazioni sugli errori, vedere The Swift Programming Language .

Examples

Errore nella gestione delle basi

Le funzioni in Swift possono restituire valori, **generare errori** o entrambi:

```
func reticulateSplines()                // no return value and no error
func reticulateSplines() -> Int         // always returns a value
func reticulateSplines() throws         // no return value, but may throw an error
func reticulateSplines() throws -> Int  // may either return a value or throw an error
```

Qualsiasi valore conforme al protocollo ErrorType (inclusi gli oggetti NSError) può essere generato come un errore. Le enumerazioni forniscono un modo conveniente per definire errori personalizzati:

2.0 2.2

```
enum NetworkError: ErrorType {
    case Offline
    case ServerError(String)
}
```

3.0

```
enum NetworkError: Error {
    // Swift 3 dictates that enum cases should be `lowerCamelCase`
    case offline
    case serverError(String)
}
```

Un errore indica un errore non irreversibile durante l'esecuzione del programma ed è gestito con i costrutti di flusso di controllo specializzati che do / catch , throw e try .

```
func fetchResource(resource: NSURL) throws -> String {
    if let (statusCode, responseString) = /* ...from elsewhere...*/ {
        if case 500..<600 = statusCode {
            throw NetworkError.serverError(responseString)
        } else {
            return responseString
        }
    } else {
        throw NetworkError.offline
    }
}
```

Gli errori possono essere catturati con do / catch :

```
do {
    let response = try fetchResource(resURL)
    // If fetchResource() didn't throw an error, execution continues here:
    print("Got response: \(response)")
```

```
    ...
} catch {
    // If an error is thrown, we can handle it here.
    print("Whoops, couldn't fetch resource: \(error)")
}
```

Qualsiasi funzione che può generare un errore ***deve*** essere chiamata usando try , try? , o try! :

```
// error: call can throw but is not marked with 'try'
let response = fetchResource(resURL)

// "try" works within do/catch, or within another throwing function:
do {
    let response = try fetchResource(resURL)
} catch {
    // Handle the error
}

func foo() throws {
    // If an error is thrown, continue passing it up to the caller.
    let response = try fetchResource(resURL)
}

// "try?" wraps the function's return value in an Optional (nil if an error was thrown).
if let response = try? fetchResource(resURL) {
    // no error was thrown
}

// "try!" crashes the program at runtime if an error occurs.
let response = try! fetchResource(resURL)
```

Cattura diversi tipi di errore

Creiamo il nostro tipo di errore per questo esempio.

2.2

```
enum CustomError: ErrorType {
    case SomeError
    case AnotherError
}

func throwing() throws {
    throw CustomError.SomeError
}
```

3.0

```
enum CustomError: Error {
    case someError
    case anotherError
}

func throwing() throws {
    throw CustomError.someError
}
```

La sintassi Do-Catch consente di rilevare un errore generato e crea *automaticamente* un error denominato costante disponibile nel blocco catch :

```
do {
    try throwing()
} catch {
    print(error)
}
```

Puoi anche dichiarare una variabile tu stesso:

```
do {
    try throwing()
} catch let oops {
    print(oops)
}
```

È anche possibile collegare diverse dichiarazioni di catch . Questo è utile se possono essere lanciati diversi tipi di errori nel blocco Do.

Qui Do-Catch tenterà prima di lanciare l'errore come CustomError , quindi come NSError se il tipo personalizzato non è stato abbinato.

2.2

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch let error as NSError {
    print(error)
}
```

3.0

In Swift 3, non c'è bisogno di downcast esplicitamente a NSError.

```
do {
    try somethingMayThrow()
} catch let custom as CustomError {
    print(custom)
} catch {
    print(error)
}
```

Catch and Switch Pattern for Explicit Error Handling

```
class Plane {

    enum Emergency: ErrorType {
        case NoFuel
        case EngineFailure(reason: String)
        case DamagedWing
    }

    var fuelInKilograms: Int

    //... init and other methods not shown

    func fly() throws {
        // ...
        if fuelInKilograms <= 0 {
```

```
            // uh oh...
            throw Emergency.NoFuel
        }
    }

}
```

Nella classe cliente:

```
let airforceOne = Plane()
do {
    try airforceOne.fly()
} catch let emergency as Plane.Emergency {
    switch emergency {
    case .NoFuel:
        // call nearest airport for emergency landing
    case .EngineFailure(let reason):
        print(reason) // let the mechanic know the reason
    case .DamagedWing:
        // Assess the damage and determine if the president can make it
    }
}
```

Disabilitare la propagazione degli errori

I creatori di Swift hanno prestato molta attenzione a rendere il linguaggio espressivo e la gestione degli errori è esattamente questo, espressivo. Se si tenta di richiamare una funzione che può generare un errore, la chiamata alla funzione deve essere preceduta dalla parola chiave try. La parola chiave try non è magica. Tutto ciò che fa è rendere lo sviluppatore consapevole della capacità di lancio della funzione.

Ad esempio, il codice seguente utilizza una funzione loadImage (atPath :), che carica la risorsa immagine in un determinato percorso o genera un errore se l'immagine non può essere caricata. In questo caso, poiché l'immagine viene fornita con l'applicazione, non verrà generato alcun errore in fase di runtime, quindi è opportuno disabilitare la propagazione dell'errore.

```
let photo = try! loadImage(atPath: "./Resources/John Appleseed.jpg")
```

Crea un errore personalizzato con descrizione localizzata

Crea enum di errori personalizzati

```
enum RegistrationError: Error {
    case invalidEmail
    case invalidPassword
    case invalidPhoneNumber
}
```

Crea extension di RegistrationError per gestire la descrizione localizzata.

```
extension RegistrationError: LocalizedError {
    public var errorDescription: String? {
        switch self {
        case .invalidEmail:
            return NSLocalizedString("Description of invalid email address", comment: "Invalid
Email")
        case .invalidPassword:
            return NSLocalizedString("Description of invalid password", comment: "Invalid
```

```
Password")
        case .invalidPhoneNumber:
            return NSLocalizedString("Description of invalid phoneNumber", comment: "Invalid
Phone Number")
        }
    }
}
```

Errore di gestione:

```
let error: Error = RegistrationError.invalidEmail
print(error.localizedDescription)
```

Leggi Gestione degli errori online: https://riptutorial.com/it/swift/topic/283/gestione-degli-errori

Capitolo 28: Gestione della memoria

introduzione

Questo argomento descrive come e quando il runtime di Swift deve allocare memoria per le strutture di dati dell'applicazione e quando tale memoria deve essere recuperata. Per impostazione predefinita, le istanze della classe di supporto della memoria vengono gestite tramite il conteggio dei riferimenti. Le strutture sono sempre passate attraverso la copia. Per disattivare lo schema di gestione della memoria integrato, si potrebbe usare la struttura [Unmanaged] [1]. [1]: https://developer.apple.com/reference/swift/unmanaged

Osservazioni

Quando utilizzare la parola chiave weak:

La parola chiave weak deve essere utilizzata, se un oggetto di riferimento può essere deallocato durante la vita dell'oggetto che detiene il riferimento.

Quando utilizzare la parola chiave sconosciuta:

Il unowned -parola chiave deve essere utilizzato, se un oggetto di riferimento non dovrebbe essere deallocato durante la vita dell'oggetto tiene il riferimento.

insidie

Un errore frequente è quello di dimenticare di creare riferimenti agli oggetti, che sono necessari per sopravvivere al termine di una funzione, come i gestori di località, i gestori del movimento, ecc.

Esempio:

```
class A : CLLocationManagerDelegate
{
    init()
    {
        let locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

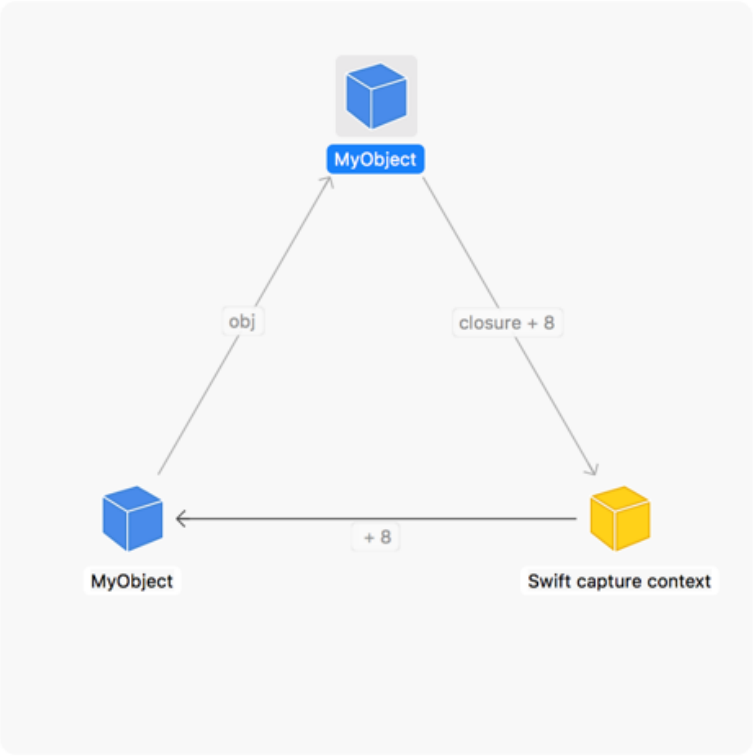
Questo esempio non funzionerà correttamente, in quanto il gestore di posizione viene deallocato dopo il ritorno dell'inizializzatore. La soluzione corretta è creare un riferimento forte come variabile di istanza:

```
class A : CLLocationManagerDelegate
{
    let locationManager:CLLocationManager

    init()
    {
        locationManager = CLLocationManager()
        locationManager.delegate = self
        locationManager.startLocationUpdates()
    }
}
```

Examples

Cicli di riferimento e riferimenti deboli

Un *ciclo di riferimento* (o *ciclo di mantenimento*) è così chiamato perché indica un ciclo nel grafico dell'oggetto :



Ogni freccia indica un oggetto che ne conserva un altro (un riferimento forte). A meno che il ciclo non venga interrotto, la memoria di questi oggetti **non** verrà **mai liberata** .

Un ciclo di conservazione viene creato quando due istanze di classi si richiamano l'una con l'altra:

```
class A { var b: B? = nil }
class B { var a: A? = nil }

let a = A()
let b = B()

a.b = b  // a retains b
b.a = a  // b retains a -- a reference cycle
```

Entrambe le istanze vivranno fino alla fine del programma. Questo è un ciclo di conservazione.

Riferimenti deboli

Per evitare i cicli di conservazione, utilizzare la parola chiave weak o unowned durante la creazione di riferimenti per interrompere i cicli di conservazione.

```
class B { weak var a: A? = nil }
```

I riferimenti deboli o non condivisi non aumentano il numero di riferimenti di un'istanza. Questi riferimenti non contribuiscono a mantenere i cicli. Il riferimento debole **diventa nil** quando l'oggetto a cui fa riferimento viene deallocato.

```
a.b = b  // a retains b
```

```
b.a = a  // b holds a weak reference to a -- not a reference cycle
```

Quando si lavora con chiusure, è possibile utilizzare anche gli elenchi di acquisizione weak e unowned .

Gestione manuale della memoria

Quando si interfaccia con le API C, si potrebbe voler bloccare il contatore di riferimento Swift. Ciò avviene con oggetti non gestiti.

Se è necessario fornire un puntatore punteggiato a una funzione C, utilizzare il metodo toOpaque della struttura Unmanaged per ottenere un puntatore raw e fromOpaque per ripristinare l'istanza originale:

```
setupDisplayLink() {
  let pointerToSelf: UnsafeRawPointer = Unmanaged.passUnretained(self).toOpaque()
  CVDisplayLinkSetOutputCallback(self.displayLink, self.redraw, pointerToSelf)
}

func redraw(pointerToSelf: UnsafeRawPointer, /* args omitted */) {
  let recoveredSelf = Unmanaged<Self>.fromOpaque(pointerToSelf).takeUnretainedValue()
  recoveredSelf.doRedraw()
}
```

Si noti che, se si utilizza passUnretained e controparti, è necessario prendere tutte le precauzioni come con riferimenti unowned citati.

Per interagire con API legacy Objective-C, è possibile che si desideri modificare manualmente il conteggio dei riferimenti di un determinato oggetto. Per questo Unmanaged ha i rispettivi metodi di retain e release . Tuttavia, è più desiderabile utilizzare passRetained e takeRetainedValue , che eseguono il mantenimento prima di restituire il risultato:

```
func preferredFilenameExtension(for uti: String) -> String! {
  let result = UTTypeCopyPreferredTagWithClass(uti, kUTTagClassFilenameExtension)
  guard result != nil else { return nil }

  return result!.takeRetainedValue() as String
}
```

Queste soluzioni dovrebbero essere sempre l'ultima risorsa e le API native della lingua potrebbero sempre essere preferite.

Leggi Gestione della memoria online: https://riptutorial.com/it/swift/topic/745/gestione-della-memoria

Capitolo 29: Hashing crittografico

Examples

MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

Queste funzioni eseguiranno l'hash di String o Data con uno degli otto algoritmi di hash crittografico.

Il parametro name specifica il nome della funzione hash come una stringa
Le funzioni supportate sono MD2, MD4, MD5, SHA1, SHA224, SHA256, SHA384 e SHA512

Questo esempio richiede Common Crypto
È necessario disporre di un'intestazione di bridging per il progetto:
#import <CommonCrypto/CommonCrypto.h>
Aggiungere il Security.framework al progetto.

Questa funzione accetta un nome hash e dati da hash e restituisce un dato:

```
name: A name of a hash function as a String
data: The Data to be hashed
returns: the hashed result as Data
```

```
func hash(name:String, data:Data) -> Data? {
    let algos = ["MD2":    (CC_MD2,    CC_MD2_DIGEST_LENGTH),
                 "MD4":    (CC_MD4,    CC_MD4_DIGEST_LENGTH),
                 "MD5":    (CC_MD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA1":   (CC_SHA1,   CC_SHA1_DIGEST_LENGTH),
                 "SHA224": (CC_SHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (CC_SHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (CC_SHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (CC_SHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[name]  else { return nil }
    var hashData = Data(count: Int(length))

    _ = hashData.withUnsafeMutableBytes {digestBytes in
        data.withUnsafeBytes {messageBytes in
            hashAlgorithm(messageBytes, CC_LONG(data.count), digestBytes)
        }
    }
    return hashData
}
```

Questa funzione accetta un nome hash e String da hash e restituisce un dato:

```
name: A name of a hash function as a String
string: The String to be hashed
returns: the hashed result as Data
```

```
func hash(name:String, string:String) -> Data? {
    let data = string.data(using:.utf8)!
    return hash(name:name, data:data)
}
```

Esempi:

```
let clearString = "clearData0123456"
let clearData   = clearString.data(using:.utf8)!
print("clearString: \(clearString)")
print("clearData: \(clearData as NSData)")

let hashSHA256 = hash(name:"SHA256", string:clearString)
print("hashSHA256: \(hashSHA256! as NSData)")

let hashMD5 = hash(name:"MD5", data:clearData)
print("hashMD5: \(hashMD5! as NSData)")
```

Produzione:

```
clearString: clearData0123456
clearData: <636c6561 72446174 61303132 33343536>

hashSHA256: <aabc766b 6b357564 e41f4f91 2d494bcc bfa16924 b574abbd ba9e3e9d a0c8920a>
hashMD5: <4df665f7 b94aea69 695b0e7b baf9e9d6>
```

HMAC con MD5, SHA1, SHA224, SHA256, SHA384, SHA512 (Swift 3)

> Queste funzioni eseguiranno l'hash di String o Data con uno degli otto algoritmi di hash crittografico.

Il parametro name specifica il nome della funzione hash come stringa Le funzioni supportate sono MD5, SHA1, SHA224, SHA256, SHA384 e SHA512

Questo esempio richiede Common Crypto
È necessario disporre di un'intestazione di bridging per il progetto:
#import <CommonCrypto/CommonCrypto.h>
Aggiungere il Security.framework al progetto.

Queste funzioni accettano un nome hash, un messaggio da hash, una chiave e restituiscono un digest:

```
hashName: name of a hash function as String
message:  message as Data
key:      key as Data
returns:  digest as Data
```

```
func hmac(hashName:String, message:Data, key:Data) -> Data? {
    let algos = ["SHA1":   (kCCHmacAlgSHA1,   CC_SHA1_DIGEST_LENGTH),
                 "MD5":    (kCCHmacAlgMD5,    CC_MD5_DIGEST_LENGTH),
                 "SHA224": (kCCHmacAlgSHA224, CC_SHA224_DIGEST_LENGTH),
                 "SHA256": (kCCHmacAlgSHA256, CC_SHA256_DIGEST_LENGTH),
                 "SHA384": (kCCHmacAlgSHA384, CC_SHA384_DIGEST_LENGTH),
                 "SHA512": (kCCHmacAlgSHA512, CC_SHA512_DIGEST_LENGTH)]
    guard let (hashAlgorithm, length) = algos[hashName]  else { return nil }
    var macData = Data(count: Int(length))

    macData.withUnsafeMutableBytes {macBytes in
        message.withUnsafeBytes {messageBytes in
            key.withUnsafeBytes {keyBytes in
                CCHmac(CCHmacAlgorithm(hashAlgorithm),
                       keyBytes,     key.count,
                       messageBytes, message.count,
                       macBytes)
            }
```

```
        }
    }
    return macData
}
```

```
hashName: name of a hash function as String
message:  message as String
key:      key as String
returns:  digest as Data
```

```
func hmac(hashName:String, message:String, key:String) -> Data? {
    let messageData = message.data(using:.utf8)!
    let keyData = key.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:keyData)
}
```

```
hashName: name of a hash function as String
message:  message as String
key:      key as Data
returns:  digest as Data
```

```
func hmac(hashName:String, message:String, key:Data) -> Data? {
    let messageData = message.data(using:.utf8)!
    return hmac(hashName:hashName, message:messageData, key:key)
}
```

```
// Esempi
```

```
let clearString = "clearData0123456"
let keyString   = "keyData8901234562"
let clearData   = clearString.data(using:.utf8)!
let keyData     = keyString.data(using:.utf8)!
print("clearString: \(clearString)")
print("keyString:   \(keyString)")
print("clearData: \(clearData as NSData)")
print("keyData:   \(keyData as NSData)")

let hmacData1 = hmac(hashName:"SHA1", message:clearData, key:keyData)
print("hmacData1: \(hmacData1! as NSData)")

let hmacData2 = hmac(hashName:"SHA1", message:clearString, key:keyString)
print("hmacData2: \(hmacData2! as NSData)")

let hmacData3 = hmac(hashName:"SHA1", message:clearString, key:keyData)
print("hmacData3: \(hmacData3! as NSData)")
```

Produzione:

```
clearString: clearData0123456
keyString:   keyData8901234562
clearData: <636c6561 72446174 61303132 33343536>
keyData:   <6b657944 61746138 39303132 33343536 32>

hmacData1: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
hmacData2: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
```

```
hmacData3: <bb358f41 79b68c08 8e93191a da7dabbc 138f2ae6>
```

Leggi Hashing crittografico online: https://riptutorial.com/it/swift/topic/7885/hashing-crittografico

Capitolo 30: Il Defer Statement

Examples

Quando utilizzare una dichiarazione di differimento

Una dichiarazione di defer costituita da un blocco di codice, che verrà eseguito quando una funzione ritorna e dovrebbe essere utilizzata per la pulizia.

Poiché le dichiarazioni di guard di Swift incoraggiano uno stile di ritorno anticipato, possono esistere molti possibili percorsi per un ritorno. Una dichiarazione di defer fornisce il codice di pulizia, che non deve essere ripetuto ogni volta.

Può anche risparmiare tempo durante il debugging e il profiling, poiché è possibile evitare perdite di memoria e risorse aperte non utilizzate a causa di una pulizia dimenticata.

Può essere usato per deallocare un buffer alla fine di una funzione:

```
func doSomething() {
    let data = UnsafeMutablePointer<UInt8>(allocatingCapacity: 42)
    // this pointer would not be released when the function returns
    // so we add a defer-statement
    defer {
        data.deallocateCapacity(42)
    }
    // it will be executed when the function returns.

    guard condition else {
        return /* will execute defer-block */
    }

}   // The defer-block will also be executed on the end of the function.
```

Può anche essere usato per chiudere le risorse alla fine di una funzione:

```
func write(data: UnsafePointer<UInt8>, dataLength: Int) throws {
    var stream:NSOutputStream = getOutputStream()
    defer {
        stream.close()
    }

    let written = stream.write(data, maxLength: dataLength)
    guard written >= 0 else {
        throw stream.streamError! /* will execute defer-block */
    }

}    // the defer-block will also be executed on the end of the function
```

Quando NON usare una dichiarazione di differimento

Quando si utilizza una dichiarazione di differimento, assicurarsi che il codice rimanga leggibile e che l'ordine di esecuzione rimanga chiaro. Ad esempio, il seguente uso dell'istruzione differimento rende l'ordine di esecuzione e la funzione del codice difficili da comprendere.

```
postfix func ++ (inout value: Int) -> Int {
    defer { value += 1 } // do NOT do this!
    return value
}
```

Leggi Il Defer Statement online: https://riptutorial.com/it/swift/topic/4932/il-defer-statement

Capitolo 31: Imposta

Examples

Dichiarazione di insiemi

Gli insiemi sono raccolte non ordinate di valori unici. I valori unici devono essere dello stesso tipo.

```
var colors = Set<String>()
```

È possibile dichiarare un set con valori utilizzando la sintassi letterale dell'array.

```
var favoriteColors: Set<String> = ["Red", "Blue", "Green", "Blue"]
// {"Blue", "Green", "Red"}
```

Modifica dei valori in un set

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

È possibile utilizzare il metodo insert(_:) per aggiungere un nuovo elemento in un set.

```
favoriteColors.insert("Orange")
//favoriteColors = {"Red", "Green", "Orange", "Blue"}
```

È possibile utilizzare il metodo remove(_:) per rimuovere un elemento da un set. Restituisce il valore contenente facoltativo che è stato rimosso o zero se il valore non era nel set.

```
let removedColor = favoriteColors.remove("Red")
//favoriteColors = {"Green", "Orange", "Blue"}
// removedColor = Optional("Red")

let anotherRemovedColor = favoriteColors.remove("Black")
// anotherRemovedColor = nil
```

Verifica se un set contiene un valore

```
var favoriteColors: Set = ["Red", "Blue", "Green"]
//favoriteColors = {"Blue", "Green", "Red"}
```

È possibile utilizzare il metodo contains(_:) per verificare se un set contiene un valore. Restituirà vero se il set contiene quel valore.

```
if favoriteColors.contains("Blue") {
    print("Who doesn't like blue!")
}
// Prints "Who doesn't like blue!"
```

Esecuzione di operazioni sui set

Valori comuni di entrambi i set:

È possibile utilizzare il metodo intersect(_:) per creare un nuovo set contenente tutti i valori

comuni a entrambi i set.

```
let favoriteColors: Set = ["Red", "Blue", "Green"]
let newColors: Set = ["Purple", "Orange", "Green"]

let intersect = favoriteColors.intersect(newColors) // a AND b
// intersect = {"Green"}
```

Tutti i valori di ogni set:

È possibile utilizzare il metodo union(_:) per creare un nuovo set contenente tutti i valori univoci di ciascun set.

```
let union = favoriteColors.union(newColors) // a OR b
// union = {"Red", "Purple", "Green", "Orange", "Blue"}
```

Si noti come il valore "Verde" appare solo una volta nel nuovo set.

Valori che non esistono in entrambi i set:

È possibile utilizzare il metodo exclusiveOr(_:) per creare un nuovo set contenente i valori univoci da entrambi, ma non da entrambi.

```
let exclusiveOr = favoriteColors.exclusiveOr(newColors) // a XOR b
// exclusiveOr = {"Red", "Purple", "Orange", "Blue"}
```

Nota come il valore "Verde" non appare nel nuovo set, poiché era in entrambi i set.

Valori che non sono in un set:

È possibile utilizzare il metodo subtract(_:) per creare un nuovo set contenente valori che non si trovano in un set specifico.

```
let subtract = favoriteColors.subtract(newColors) // a - (a AND b)
// subtract = {"Blue", "Red"}
```

Notare come il valore "Verde" non appare nel nuovo set, poiché era anche nel secondo set.

Aggiunta di valori del mio tipo a un Set

Per definire un Set del tuo tipo devi conformare il tuo tipo a Hashable

```
struct Starship: Hashable {
    let name: String
    var hashValue: Int { return name.hashValue }
}

func ==(left:Starship, right: Starship) -> Bool {
    return left.name == right.name
}
```

Ora puoi creare un Set di Starship(s)

```
let ships : Set<Starship> = [Starship(name:"Enterprise D"), Starship(name:"Voyager"),
Starship(name:"Defiant") ]
```

CountedSet

3.0

Swift 3 introduce la classe CountedSet (è la versione Swift della NSCountedSet Objective-C NSCountedSet).

CountedSet, come suggerito dal nome, tiene traccia di quante volte è presente un valore.

```
let countedSet = CountedSet()
countedSet.add(1)
countedSet.add(1)
countedSet.add(1)
countedSet.add(2)

countedSet.count(for: 1) // 3
countedSet.count(for: 2) // 1
```

Leggi Imposta online: https://riptutorial.com/it/swift/topic/371/imposta

Capitolo 32: Iniezione di dipendenza

Examples

Iniezione delle dipendenze con View Controller

Iniezione dell'iniezione di Dependenct

Un'applicazione è composta da molti oggetti che collaborano tra loro. Gli oggetti di solito dipendono da altri oggetti per eseguire alcune attività. Quando un oggetto è responsabile del riferimento alle proprie dipendenze, conduce a un codice altamente accoppiato, difficile da testare e difficile da modificare.

L'iniezione di dipendenza è un modello di progettazione software che implementa l'inversione del controllo per la risoluzione delle dipendenze. Un'iniezione sta passando dipendenza a un oggetto dipendente che lo userebbe. Ciò consente una separazione delle dipendenze del client dal comportamento del client, che consente all'applicazione di essere accoppiata liberamente.

Da non confondere con la definizione di cui sopra - un'iniezione di dipendenza significa semplicemente dare a un oggetto le sue variabili di istanza.

È così semplice, ma offre molti vantaggi:

- più facile testare il codice (utilizzando test automatici come test di unità e UI)
- se utilizzato in tandem con la programmazione orientata ai protocolli, semplifica il cambio di implementazione di una determinata classe, più semplice da refactoring
- rende il codice più modulare e riutilizzabile

Esistono tre modi più comuni in cui la Dependency Injection (DI) può essere implementata in un'applicazione:

1. Iniezione inizializzatore
2. Iniezione di proprietà
3. Utilizzo di framework DI di terze parti (come Swinject, Cleanse, Dip o Typhoon)

C'è un articolo interessante con collegamenti ad altri articoli su Dipendenza Iniezione, quindi dai un'occhiata se vuoi approfondire i principi DI e Inversion of Control.

Mostriamo come usare DI con View Controller - un compito quotidiano per uno sviluppatore iOS medio.

Esempio senza DI

Avremo due View Controller: **LoginViewController** e **TimelineViewController** . LoginViewController viene utilizzato per accedere e, una volta completato lo schema, passerà a TimelineViewController. Entrambi i controller di visualizzazione dipendono da **FirebaseNetworkService** .

LoginViewController

```
class LoginViewController: UIViewController {

    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService = FirebaseNetworkService()

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService.logutCurrentUser()
    }
}
```

FirebaseNetworkService

```
class FirebaseNetworkService {

    func loginUser(username: String, passwordHash: String) {
        // Implementation not important for this example
    }

    func logutCurrentUser() {
        // Implementation not important for this example
    }
}
```

Questo esempio è molto semplice, ma supponiamo di avere 10 o 15 diversi controller di visualizzazione e alcuni di essi dipendono anche da FirebaseNetworkService. Ad un certo momento, desideri cambiare Firebase come servizio di back-end con il servizio di back-end della tua azienda. Per farlo dovrai passare attraverso ogni controller di visualizzazione e modificare FirebaseNetworkService con CompanyNetworkService. E se alcuni dei metodi in CompanyNetworkService sono cambiati, avrai molto lavoro da fare.

Il test dell'Unità e dell'UI non è lo scopo di questo esempio, ma se si volessero testare i controller delle viste di prova con dipendenze strettamente accoppiate, si avrebbe davvero molto tempo a farlo.

Riscriviamo questo esempio e inseriamo il servizio di rete nei nostri controller di visualizzazione.

Esempio con iniezione di dipendenza

Per sfruttare al meglio l'iniezione delle dipendenze, definiamo la funzionalità del servizio di rete in un protocollo. In questo modo, i controller di visualizzazione dipendenti da un servizio di rete non dovranno nemmeno sapere della reale implementazione di esso.

```
protocol NetworkService {
    func loginUser(username: String, passwordHash: String)
    func logutCurrentUser()
}
```

Aggiungi un'implementazione del protocollo NetworkService:

```
class FirebaseNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Firebase implementation
    }
```

```
    func logutCurrentUser() {
        // Firebase implementation
    }
}
```

Cambiamo LoginViewController e TimelineViewController per utilizzare il nuovo protocollo NetworkService invece di FirebaseNetworkService.

LoginViewController

```
class LoginViewController: UIViewController {

    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }
}
```

TimelineViewController

```
class TimelineViewController: UIViewController {

    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    @IBAction func logoutButtonPressed(_ sender: UIButton) {
        networkService?.logutCurrentUser()
    }
}
```

Ora, la domanda è: come iniettare l'implementazione corretta di NetworkService in LoginViewController e TimelineViewController?

Poiché LoginViewController è il controller di visualizzazione iniziale e mostrerà ogni volta che l'applicazione si avvia, è possibile iniettare tutte le dipendenze in **AppDelegate** .

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
    // This logic will be different based on your project's structure or whether
    // you have a navigation controller or tab bar controller for your starting view
controller
    if let loginVC = window?.rootViewController as? LoginViewController {
        loginVC.networkService = FirebaseNetworkServiceImpl()
    }
    return true
}
```

In AppDelegate stiamo semplicemente prendendo il riferimento al primo controller di visualizzazione (LoginViewController) e iniettando l'implementazione NetworkService usando il metodo di iniezione di proprietà.

Ora, il prossimo compito è quello di iniettare l'implementazione NetworkService in TimelineViewController. Il modo più semplice è farlo quando LoginViewController sta passando a TimlineViewController.

Aggiungeremo il codice di iniezione nel metodo prepareForSegue nel LoginViewController (se si sta utilizzando un approccio diverso per navigare tra i controller di vista, inserire lì il codice di iniezione).

La nostra classe LoginViewController si presenta ora come questa:

```
class LoginViewController: UIViewController {
    // No need to initialize it here since an implementation
    // of the NetworkService protocol will be injected
    var networkService: NetworkService?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func prepare(for segue: UIStoryboardSegue, sender: Any?) {
        if segue.identifier == "TimelineViewController" {
            if let timelineVC = segue.destination as? TimelineViewController {
                // Injecting the NetworkService implementation
                timelineVC.networkService = networkService
            }
        }
    }
}
```

Abbiamo finito ed è così facile.

Ora immagina di voler passare dall'implementazione di NetworkService da Firebase all'implementazione del backend della nostra azienda personalizzata. Tutto ciò che dovremmo fare è:

Aggiungi una nuova classe di implementazione NetworkService:

```
class CompanyNetworkServiceImpl: NetworkService {
    func loginUser(username: String, passwordHash: String) {
        // Company API implementation
    }

    func logutCurrentUser() {
        // Company API implementation
    }
}
```

Passa a FirebaseNetworkServiceImpl con la nuova implementazione in AppDelegate:

```
func application(_ application: UIApplication, didFinishLaunchingWithOptions launchOptions:
[UIApplicationLaunchOptionsKey: Any]?) -> Bool {
        // This logic will be different based on your project's structure or whether
        // you have a navigation controller or tab bar controller for your starting view
controller
        if let loginVC = window?.rootViewController as? LoginViewController {
            loginVC.networkService = CompanyNetworkServiceImpl()
        }
        return true
    }
```

È così, abbiamo cambiato l'intera implementazione underlay del protocollo NetworkService senza nemmeno toccare LoginViewController o TimelineViewController.

Dato che questo è un semplice esempio, potresti non vedere tutti i benefici al momento, ma se provi a usare DI nei tuoi progetti, vedrai i benefici e userai sempre Dependency Injection.

Tipi di iniezione delle dipendenze

Questo esempio mostrerà come utilizzare il pattern di progettazione Dipendenza Iniezione (**DI**) in Swift usando questi metodi:

1. **Iniezione inizializzatore** (il termine corretto è Iniezione costruttore, ma poiché Swift ha inizializzatori si chiama iniezione inizializzatore)
2. **Proprietà dell'iniezione**
3. **Metodo di iniezione**

Esempio di installazione senza DI

```
    protocol Engine {
        func startEngine()
        func stopEngine()
    }

    class TrainEngine: Engine {
        func startEngine() {
            print("Engine started")
        }

        func stopEngine() {
            print("Engine stopped")
        }
    }

    protocol TrainCar {
    var numberOfSeats: Int { get }
    func attachCar(attach: Bool)
}

class RestaurantCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 30
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class PassengerCar: TrainCar {
    var numberOfSeats: Int {
        get {
            return 50
        }
    }
    func attachCar(attach: Bool) {
        print("Attach car")
    }
}

class Train {
    let engine: Engine?
    var mainCar: TrainCar?
}
```

Iniezione delle dipendenze dell'inizializzatore

Come dice il nome, tutte le dipendenze vengono iniettate attraverso l'inizializzatore della classe. Per iniettare le dipendenze attraverso l'inizializzatore, aggiungeremo l'inizializzatore alla classe Train .

Il corso di formazione ora si presenta così:

```
class Train {
    let engine: Engine?
    var mainCar: TrainCar?

    init(engine: Engine) {
        self.engine = engine
    }
}
```

Quando vogliamo creare un'istanza della classe Train utilizzeremo l'inizializzatore per iniettare una specifica implementazione del motore:

```
let train = Train(engine: TrainEngine())
```

NOTA: il vantaggio principale dell'iniezione dell'inizializzatore rispetto all'iniezione di proprietà è che possiamo impostare la variabile come variabile privata o addirittura renderla costante con la parola chiave let (come abbiamo fatto nel nostro esempio). In questo modo possiamo assicurarci che nessuno possa accedervi o cambiarlo.

Iniezione delle dipendenze delle proprietà

L'utilizzo delle proprietà di DI è ancora più semplice con l'uso di un inizializzatore. Inseriamo una dipendenza PassengerCar dall'oggetto del treno che abbiamo già creato utilizzando le proprietà DI:

```
train.mainCar = PassengerCar()
```

Questo è tutto. Il mainCar del nostro treno è ora un'istanza di PassengerCar .

Metodo Iniezione di dipendenza

Questo tipo di iniezione delle dipendenze è un po 'diverso dai precedenti perché non influenzerà l'intero oggetto, ma verrà iniettato solo una dipendenza da utilizzare nell'ambito di uno specifico metodo. Quando una dipendenza viene utilizzata solo in un singolo metodo, di solito non è buono per rendere l'intero oggetto dipendente da esso. Aggiungiamo un nuovo metodo alla classe Train:

```
func reparkCar(trainCar: TrainCar) {
    trainCar.attachCar(attach: true)
    engine?.startEngine()
    engine?.stopEngine()
    trainCar.attachCar(attach: false)
}
```

Ora, se chiamiamo il metodo della nuova classe del treno, inietteremo la TrainCar usando l'iniezione della dipendenza del metodo.

```
train.reparkCar(trainCar: RestaurantCar())
```

Leggi Iniezione di dipendenza online: https://riptutorial.com/it/swift/topic/8198/iniezione-di-

dipendenza

Capitolo 33: inizializzatori

Examples

Impostazione dei valori di proprietà predefiniti

È possibile utilizzare un inizializzatore per impostare valori di proprietà predefiniti:

```
struct Example {
    var upvotes: Int
    init() {
        upvotes = 42
    }
}
let myExample = Example() // call the initializer
print(myExample.upvotes) // prints: 42
```

Oppure, specifica i valori di proprietà predefiniti come parte della dichiarazione della proprietà:

```
struct Example {
    var upvotes = 42 // the type 'Int' is inferred here
}
```

Classi e strutture **devono** impostare tutte le proprietà archiviate su un valore iniziale appropriato al momento della creazione di un'istanza. Questo esempio non verrà compilato, perché l'inizializzatore non ha fornito un valore iniziale per i downvotes :

```
struct Example {
    var upvotes: Int
    var downvotes: Int
    init() {
         upvotes = 0
    } // error: Return from initializer without initializing all stored properties
}
```

Personalizzazione dell'inizializzazione con i parametri

```
struct MetricDistance {
    var distanceInMeters: Double

    init(fromCentimeters centimeters: Double) {
        distanceInMeters = centimeters / 100
    }
    init(fromKilometers kilos: Double) {
        distanceInMeters = kilos * 1000
    }
}

let myDistance = MetricDistance(fromCentimeters: 42)
// myDistance.distanceInMeters is 0.42
let myOtherDistance = MetricDistance(fromKilometers: 42)
// myOtherDistance.distanceInMeters is 42000
```

Nota che non puoi omettere le etichette dei parametri:

```
let myBadDistance = MetricDistance(42) // error: argument labels do not match any available
```

```
overloads
```

Per consentire l'omissione delle etichette dei parametri, utilizzare un carattere di sottolineatura _ come etichetta:

```
struct MetricDistance {
    var distanceInMeters: Double
    init(_ meters: Double) {
        distanceInMeters = meters
    }
}
let myDistance = MetricDistance(42) // distanceInMeters = 42
```

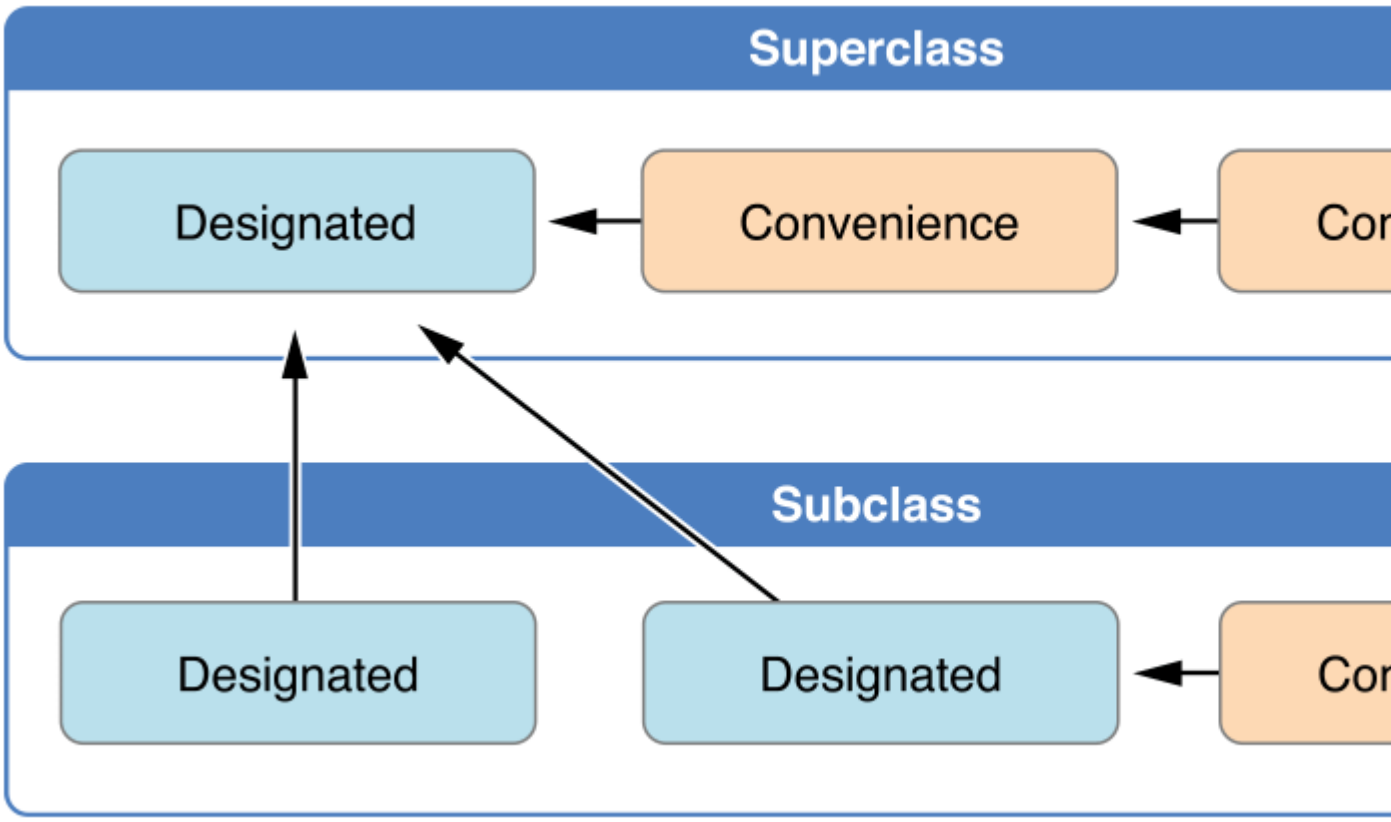
Se le etichette dei tuoi argomenti condividono nomi con una o più proprietà, usa self per impostare esplicitamente i valori delle proprietà:

```
struct Color {
    var red, green, blue: Double
    init(red: Double, green: Double, blue: Double) {
        self.red = red
        self.green = green
        self.blue = blue
    }
}
```

Convenienza init

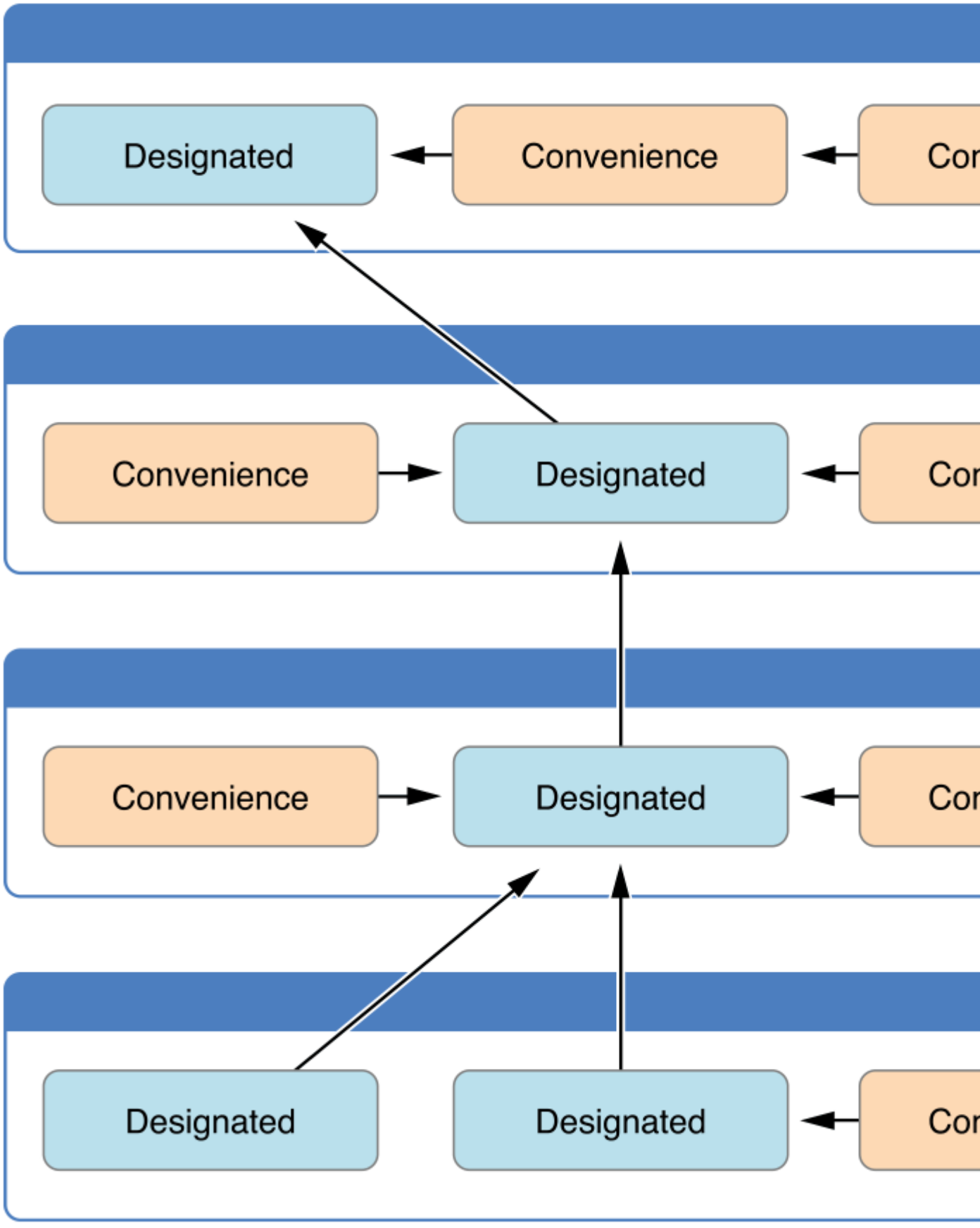
Le classi Swift supportano diversi modi di essere inizializzati. Seguendo le specifiche di Apple, queste 3 regole devono essere rispettate:

1. Un inizializzatore designato deve chiamare un inizializzatore designato dalla sua superclasse immediata.



2. Un inizializzatore conveniente deve chiamare un altro inizializzatore dalla stessa classe.

3. Un inizializzatore conveniente deve infine chiamare un inizializzatore designato.




```
class Foo {

    var someString: String
```

```
    var someValue: Int
    var someBool: Bool

    // Designated Initializer
    init(someString: String, someValue: Int, someBool: Bool)
    {
        self.someString = someString
        self.someValue = someValue
        self.someBool = someBool
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(otherString: "")
    }

    // A convenience initializer must ultimately call a designated initializer.
convenience init(otherString: String)
    {
        self.init(someString: otherString, someValue:  0, someBool: false)
    }
}


class Baz: Foo
{
    var someFloat: Float

    // Designed initializer
    init(someFloat: Float)
    {
        self.someFloat = someFloat

        // A designated initializer must call a designated initializer from its immediate
superclass.
        super.init(someString: "", someValue: 0, someBool: false)
    }

    // A convenience initializer must call another initializer from the same class.
    convenience init()
    {
        self.init(someFloat: 0)
    }
}
```

Inizializzatore designato

```
let c = Foo(someString: "Some string", someValue: 10, someBool: true)
```

Convenienza init ()

```
let a = Foo()
```

Convenience init (otherString: String)

```
let b = Foo(otherString: "Some string")
```

Inizializzatore designato (chiamerà la superclasse Designated Initializer)

```
let d = Baz(someFloat: 3)
```

Convenienza init ()

```
let e = Baz()
```

Fonte immagine: The Swift Programming Languag e

Initilizer lanciabile

Utilizzo di Error Handling per rendere l'inizializzatore di Struct (o classe) come inizializzatore lanciabile:

Esempio di errore Gestione enum:

```
enum ValidationError: Error {
    case invalid
}
```

È possibile utilizzare enum Gestione errori per verificare il parametro per il requisito previsto di Struct (o classe)

```
struct User {
    let name: String

    init(name: String?) throws {

        guard let name = name else {
           ValidationError.invalid
        }

        self.name = name
    }
}
```

Ora, puoi usare l'iniziatore throwable di:

```
do {
   let user = try User(name: "Sample name")

   // success
}
catch ValidationError.invalid {
     // handle error
}
```

Leggi inizializzatori online: https://riptutorial.com/it/swift/topic/1778/inizializzatori

Capitolo 34: Interruttore

Parametri

Parametro	Dettagli
Valore da testare	La variabile da confrontare

Osservazioni

Fornisci un caso per ogni possibile valore del tuo contributo. Utilizzare un default case per coprire i valori di input rimanenti che non si desidera specificare. Il caso predefinito deve essere l'ultimo caso.

Per impostazione predefinita, gli switch in Swift non continueranno a controllare altri casi dopo che un caso è stato abbinato.

Examples

Uso di base

```
let number = 3
switch number {
case 1:
    print("One!")
case 2:
    print("Two!")
case 3:
    print("Three!")
default:
    print("Not One, Two or Three")
}
```

le istruzioni switch funzionano anche con tipi di dati diversi dagli interi. Funzionano con qualsiasi tipo di dati. Ecco un esempio di accensione di una stringa:

```
let string = "Dog"
switch string {
case "Cat", "Dog":
  print("Animal is a house pet.")
default:
  print("Animal is not a house pet.")
}
```

Questo stamperà il seguente:

```
Animal is a house pet.
```

Abbinamento di più valori

Un singolo caso in un'istruzione switch può corrispondere su più valori.

```
let number = 3
switch number {
case 1, 2:
```

```
    print("One or Two!")
case 3:
    print("Three!")
case 4, 5, 6:
    print("Four, Five or Six!")
default:
    print("Not One, Two, Three, Four, Five or Six")
}
```

Abbinare una gamma

Un singolo caso in un'istruzione switch può corrispondere a un intervallo di valori.

```
let number = 20
switch number {
case 0:
    print("Zero")
case 1..<10:
    print("Between One and Ten")
case 10..<20:
    print("Between Ten and Twenty")
case 20..<30:
    print("Between Twenty and Thirty")
default:
    print("Greater than Thirty or less than Zero")
}
```

Utilizzando l'istruzione where in un interruttore

L'istruzione where può essere utilizzata all'interno di una corrispondenza caso switch per aggiungere ulteriori criteri richiesti per una corrispondenza positiva. L'esempio seguente controlla non solo l'intervallo, ma anche se il numero è pari o dispari:

```
switch (temperature) {
      case 0...49 where temperature % 2 == 0:
        print("Cold and even")

      case 50...79 where temperature % 2 == 0:
        print("Warm and even")

      case 80...110 where temperature % 2 == 0:
        print("Hot and even")

      default:
        print("Temperature out of range or odd")
}
```

Soddisfare uno dei molteplici vincoli utilizzando l'interruttore

Puoi creare una tupla e usare un interruttore in questo modo:

```
var str: String? = "hi"
var x: Int? = 5

switch (str, x) {
case (.Some,.Some):
    print("Both have values")
```

```
case (.Some, nil):
    print("String has a value")
case (nil, .Some):
    print("Int has a value")
case (nil, nil):
    print("Neither have values")
}
```

Corrispondenza parziale

L'istruzione switch utilizza l'abbinamento parziale.

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (0, 0, 0): // 1
  print("Origin")
case (_, 0, 0): // 2
  print("On the x-axis.")
case (0, _, 0): // 3
  print("On the y-axis.")
case (0, 0, _): // 4
  print("On the z-axis.")
default:        // 5
  print("Somewhere in space")
}
```

1. Corrisponde esattamente al caso in cui il valore è (0,0,0). Questa è l'origine dello spazio 3D.
2. Corrisponde a y = 0, z = 0 e qualsiasi valore di x. Ciò significa che la coordinata è sull'asse x.
3. Corrisponde a x = 0, z = 0 e qualsiasi valore di y. Ciò significa che la coordinata si trova sull'asse dell'asse.
4. Corrisponde a x = 0, y = 0 e qualsiasi valore di z. Ciò significa che la coordinata è sull'asse z.
5. Corrisponde al resto delle coordinate.

Nota: usare il carattere di sottolineatura per indicare che non ti interessa il valore.

Se non vuoi ignorare il valore, puoi utilizzarlo nella tua istruzione switch, in questo modo:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (0, 0, 0):
  print("Origin")
case (let x, 0, 0):
  print("On the x-axis at x = \(x)")
case (0, let y, 0):
  print("On the y-axis at y = \(y)")
case (0, 0, let z):
  print("On the z-axis at z = \(z)")
case (let x, let y, let z):
  print("Somewhere in space at x = \(x), y = \(y), z = \(z)")
}
```

Qui, i casi degli assi usano la sintassi let per estrarre i valori pertinenti. Il codice quindi stampa i valori utilizzando l'interpolazione della stringa per creare la stringa.

Nota: non è necessario un valore predefinito in questa istruzione switch. Questo perché il caso

finale è essenzialmente quello di default: corrisponde a qualsiasi cosa, perché non ci sono vincoli su nessuna parte della tupla. Se l'istruzione switch esaurisce tutti i valori possibili con i relativi casi, non è necessario alcun valore predefinito.

Possiamo anche usare la sintassi let-where per abbinare casi più complessi. Per esempio:

```
let coordinates: (x: Int, y: Int, z: Int) = (3, 2, 5)

switch (coordinates) {
case (let x, let y, _) where y == x:
  print("Along the y = x line.")
case (let x, let y, _) where y == x * x:
  print("Along the y = x^2 line.")
default:
break
}
```

Qui, corrispondiamo alle linee "y equals x" e "y equals x squared".

Scorri le innovazioni

Vale la pena notare che in modo rapido, a differenza di altre lingue con cui le persone hanno familiarità, c'è una interruzione implicita alla fine di ogni dichiarazione di un caso. Per seguire il caso successivo (es. fallthrough più casi) è necessario utilizzare la dichiarazione fallthrough .

```
switch(value) {
case 'one':
    // do operation one
    fallthrough
case 'two':
    // do this either independant, or in conjunction with first case
default:
    // default operation
}
```

questo è utile per cose come i flussi.

Cambia ed Enum

L'istruzione Switch funziona molto bene con i valori Enum

```
enum CarModel {
    case Standard, Fast, VeryFast
}

let car = CarModel.Standard

switch car {
case .Standard: print("Standard")
case .Fast: print("Fast")
case .VeryFast: print("VeryFast")
}
```

> Poiché abbiamo fornito un caso per ogni possibile valore della macchina, omettiamo il caso default .

Switch e Optionals

Alcuni esempi di casi in cui il risultato è facoltativo.

```
var result: AnyObject? = someMethod()

switch result {
case nil:
    print("result is nothing")
case is String:
    print("result is a String")
case _ as Double:
    print("result is not nil, any value that is a Double")
case let myInt as Int where myInt > 0:
    print("\(myInt) value is not nil but an int and greater than 0")
case let a?:
    print("\(a) - value is unwrapped")
}
```

Interruttori e tuple

Gli interruttori possono accendere le tuple:

```
public typealias mdyTuple = (month: Int, day: Int, year: Int)

let fredsBirthday =   (month: 4, day: 3, year: 1973)




switch theMDY
{
//You can match on a literal tuple:
case (fredsBirthday):
  message = "\(date) \(prefix) the day Fred was born"

//You can match on some of the terms, and ignore others:
case (3, 15, _):
  message = "Beware the Ides of March"

//You can match on parts of a literal tuple, and copy other elements
//into a constant that you use in the body of the case:
case (bobsBirthday.month, bobsBirthday.day, let year) where year > bobsBirthday.year:
  message = "\(date) \(prefix) Bob's \(possessiveNumber(year - bobsBirthday.year))" +
    "birthday"

//You can copy one or more elements of the tuple into a constant and then
//add a where clause that further qualifies the case:
case (susansBirthday.month, susansBirthday.day, let year)
  where year > susansBirthday.year:
  message = "\(date) \(prefix) Susan's " +
    "\(possessiveNumber(year - susansBirthday.year)) birthday"

//You can match some elements to ranges:.
case (5, 1...15, let year):
  message = "\(date) \(prefix) in the first half of May, \(year)"
}
```

Abbinamento basato sulla classe - ottimo per preparare ForSegue

Puoi anche fare un cambio di istruzione switch in base alla **classe** della cosa che stai

accendendo.

Un esempio in cui questo è utile è in prepareForSegue . Ho usato passare in base all'identificatore di seguito, ma è fragile. se cambi lo storyboard in un secondo momento e rinomina l'identificatore dei passaggi, il tuo codice verrà violato. Oppure, se si utilizza segues per più istanze della stessa classe di controller di visualizzazione (ma scene di storyboard diverse), non è possibile utilizzare l'identificatore di follow per calcolare la classe della destinazione.

Dichiarazioni di commutazione rapida in soccorso.

Usa case let var as Class Swift e case let var as Class sintassi di case let var as Class , come questo:

3.0

```
override func prepareForSegue(segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
      break
  }
}
```

3.0

In Swift 3 la sytax è leggermente cambiata:

```
override func prepare(for segue: UIStoryboardSegue, sender: AnyObject?) {
  switch segue.destinationViewController {
    case let fooViewController as FooViewController:
      fooViewController.delegate = self

    case let barViewController as BarViewController:
      barViewController.data = data

    default:
      break
  }
}
```

Leggi Interruttore online: https://riptutorial.com/it/swift/topic/207/interruttore

Capitolo 35: Introduzione alla programmazione orientata ai protocolli

Osservazioni

Per ulteriori informazioni su questo argomento, consultare la programmazione orientata al protocollo talk di WWDC 2015 in Swift .

C'è anche una grande guida scritta sullo stesso: Introduzione alla programmazione orientata ai protocolli in Swift 2 .

Examples

Utilizzo della programmazione orientata ai protocolli per il test delle unità

La programmazione orientata al protocollo è uno strumento utile per scrivere facilmente test unitari migliori per il nostro codice.

Diciamo che vogliamo testare un UIViewController che si basa su una classe ViewModel.

I passaggi necessari sul codice di produzione sono:

1. Definire un protocollo che espone l'interfaccia pubblica della classe ViewModel, con tutte le proprietà e i metodi necessari da UIViewController.
2. Implementare la classe ViewModel reale, conforme a tale protocollo.
3. Utilizzare una tecnica di iniezione delle dipendenze per consentire al controller di visualizzazione di utilizzare l'implementazione desiderata, passandola come protocollo e non come istanza concreta.

```
protocol ViewModelType {
   var title : String {get}
   func confirm()
}

class ViewModel : ViewModelType {
   let title : String

   init(title: String) {
       self.title = title
   }
   func confirm() { ... }
}

class ViewController : UIViewController {
   // We declare the viewModel property as an object conforming to the protocol
   // so we can swap the implementations without any friction.
   var viewModel : ViewModelType!
   @IBOutlet var titleLabel : UILabel!

   override func viewDidLoad() {
       super.viewDidLoad()
       titleLabel.text = viewModel.title
   }

   @IBAction func didTapOnButton(sender: UIButton) {
       viewModel.confirm()
   }
}

// With DI we setup the view controller and assign the view model.
// The view controller doesn't know the concrete class of the view model,
```

```
// but just relies on the declared interface on the protocol.
let viewController = //... Instantiate view controller
viewController.viewModel = ViewModel(title: "MyTitle")
```

Quindi, sul test unitario:

1. Implementare un finto ViewModel conforme allo stesso protocollo
2. Passalo a UIViewController sotto test usando dependency injection, invece dell'istanza reale.
3. Test!

```
class FakeViewModel : ViewModelType {
   let title : String = "FakeTitle"

   var didConfirm = false
   func confirm() {
       didConfirm = true
   }
}

class ViewControllerTest : XCTestCase {
    var sut : ViewController!
    var viewModel : FakeViewModel!

    override func setUp() {
        super.setUp()

        viewModel = FakeViewModel()
        sut = // ... initialization for view controller
        sut.viewModel = viewModel

        XCTAssertNotNil(self.sut.view) // Needed to trigger view loading
    }

    func testTitleLabel() {
        XCTAssertEqual(self.sut.titleLabel.text, "FakeTitle")
    }

    func testTapOnButton() {
        sut.didTapOnButton(UIButton())
        XCTAssertTrue(self.viewModel.didConfirm)
    }
}
```

Utilizzo dei protocolli come tipi di prima classe

La programmazione orientata al protocollo può essere utilizzata come un modello di base di Swift.

Diversi tipi sono in grado di conformarsi allo stesso protocollo, i tipi di valore possono persino conformarsi a più protocolli e persino fornire l'implementazione del metodo di default.

Inizialmente vengono definiti protocolli che possono rappresentare proprietà e / o metodi comunemente utilizzati con tipi specifici o generici.

```
protocol ItemData {

    var title: String { get }
    var description: String { get }
    var thumbnailURL: NSURL { get }
```

```
    var created: NSDate { get }
    var updated: NSDate { get }

}

protocol DisplayItem {

    func hasBeenUpdated() -> Bool
    func getFormattedTitle() -> String
    func getFormattedDescription() -> String

}

protocol GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void)
}
```

È possibile creare un'implementazione predefinita per il metodo get, anche se i tipi conformi desiderati potrebbero sovrascrivere l'implementazione.

```
extension GetAPIItemDataOperation {

    static func get(url: NSURL, completed: ([ItemData]) -> Void) {

        let date = NSDate(
        timeIntervalSinceNow: NSDate().timeIntervalSince1970
            + 5000)

        // get data from url
        let urlData: [String: AnyObject] = [
            "title": "Red Camaro",
            "desc": "A fast red car.",
            "thumb":"http://cars.images.com/red-camaro.png",
            "created": NSDate(), "updated": date]

        // in this example forced unwrapping is used
        // forced unwrapping should never be used in practice
        // instead conditional unwrapping should be used (guard or if/let)
        let item = Item(
            title: urlData["title"] as! String,
            description: urlData["desc"] as! String,
            thumbnailURL: NSURL(string: urlData["thumb"] as! String)!,
            created: urlData["created"] as! NSDate,
            updated: urlData["updated"] as! NSDate)

        completed([item])

    }
}

struct ItemOperation: GetAPIItemDataOperation { }
```

Un tipo di valore conforme al protocollo ItemData, questo tipo di valore è anche in grado di conformarsi ad altri protocolli.

```
struct Item: ItemData {

    let title: String
    let description: String
```

```
    let thumbnailURL: NSURL
    let created: NSDate
    let updated: NSDate

}
```

Qui la struttura dell'articolo viene estesa per conformarsi a un elemento di visualizzazione.

```
extension Item: DisplayItem {

    func hasBeenUpdated() -> Bool {
        return updated.timeIntervalSince1970 >
            created.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return description.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}
```

Un sito di chiamata di esempio per l'utilizzo del metodo get statico.

```
ItemOperation.get(NSURL()) { (itemData) in

    // perhaps inform a view of new data
    // or parse the data for user requested info, etc.
    dispatch_async(dispatch_get_main_queue(), {

        // self.items = itemData
    })

}
```

Diversi casi d'uso richiederanno diverse implementazioni. L'idea principale qui è mostrare la conformità da tipi diversi in cui il protocollo è il punto principale del focus nel design. In questo esempio, forse i dati API vengono salvati condizionalmente in un'entità Core Data.

```
// the default core data created classes + extension
class LocalItem: NSManagedObject { }

extension LocalItem {

    @NSManaged var title: String
    @NSManaged var itemDescription: String
    @NSManaged var thumbnailURLStr: String
    @NSManaged var createdAt: NSDate
    @NSManaged var updatedAt: NSDate
}
```

Qui la classe di supporto Core Data può anche essere conforme al protocollo DisplayItem.

```
extension LocalItem: DisplayItem {

```

```
    func hasBeenUpdated() -> Bool {
        return updatedAt.timeIntervalSince1970 >
            createdAt.timeIntervalSince1970
    }

    func getFormattedTitle() -> String {
        return title.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }

    func getFormattedDescription() -> String {
        return itemDescription.stringByTrimmingCharactersInSet(
            .whitespaceAndNewlineCharacterSet())
    }
}

// In use, the core data results can be
// conditionally casts as a protocol
class MyController: UIViewController {

    override func viewDidLoad() {

        let fr: NSFetchRequest = NSFetchRequest(
        entityName: "Items")

        let context = NSManagedObjectContext(
        concurrencyType: .MainQueueConcurrencyType)

        do {

            let items: AnyObject = try context.executeFetchRequest(fr)
            if let displayItems = items as? [DisplayItem] {

                print(displayItems)
            }

        } catch let error as NSError {
            print(error.localizedDescription)
        }

    }
}
```

Leggi Introduzione alla programmazione orientata ai protocolli online: https://riptutorial.com/it/swift/topic/2502/introduzione-alla-programmazione-orientata-ai-protocolli

Capitolo 36: Lavorare con C e Objective-C

Osservazioni

Per ulteriori informazioni, consultare la documentazione di Apple sull'utilizzo di Swift con Cocoa e Objective-C .

Examples

Utilizzo delle classi Swift dal codice Objective-C

Nello stesso modulo

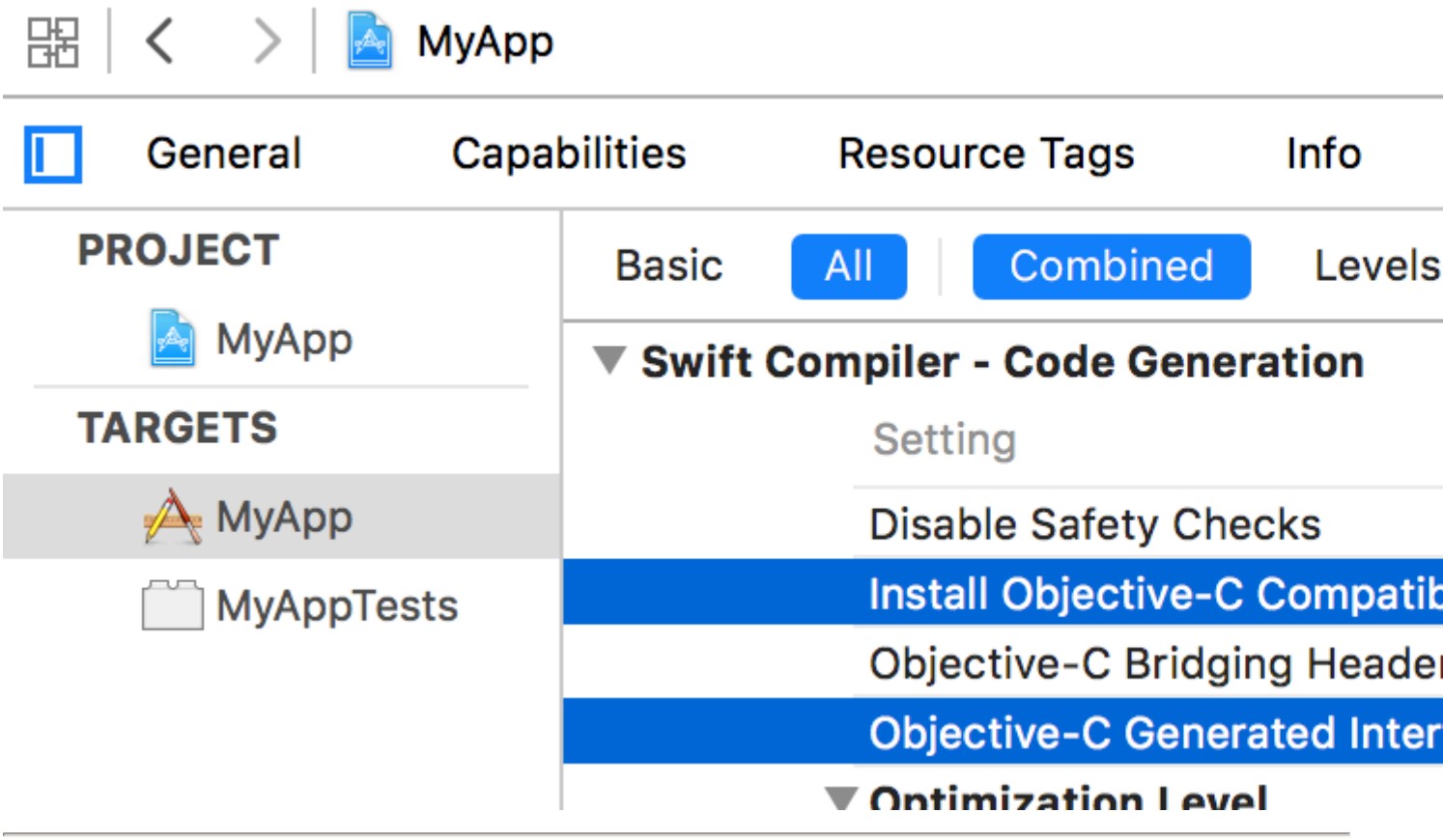
All'interno di un modulo chiamato " **MyModule** ", Xcode genera un'intestazione denominata **MyModule-Swift.h** che espone le classi Swift pubbliche a Objective-C. Importa questa intestazione per utilizzare le classi Swift:

```
// MySwiftClass.swift in MyApp
import Foundation

// The class must be `public` to be visible, unless this target also has a bridging header
public class MySwiftClass: NSObject {
    // ...
}
```

```
// MyViewController.m in MyApp

#import "MyViewController.h"
#import "MyApp-Swift.h"                     // import the generated interface
#import <MyFramework/MyFramework-Swift.h>   // or use angle brackets for a framework target

@implementation MyViewController
- (void)demo {
    [[MySwiftClass alloc] init];  // use the Swift class
}
@end
```

Impostazioni di costruzione rilevanti:

- **Nome intestazione interfaccia Objective-C** : controlla il nome dell'intestazione Obj-C generata.
- **Installa Intestazione di compatibilità Objective-C** : se l'intestazione -Swift.h deve essere un'intestazione pubblica (per gli obiettivi del framework).



In un altro modulo

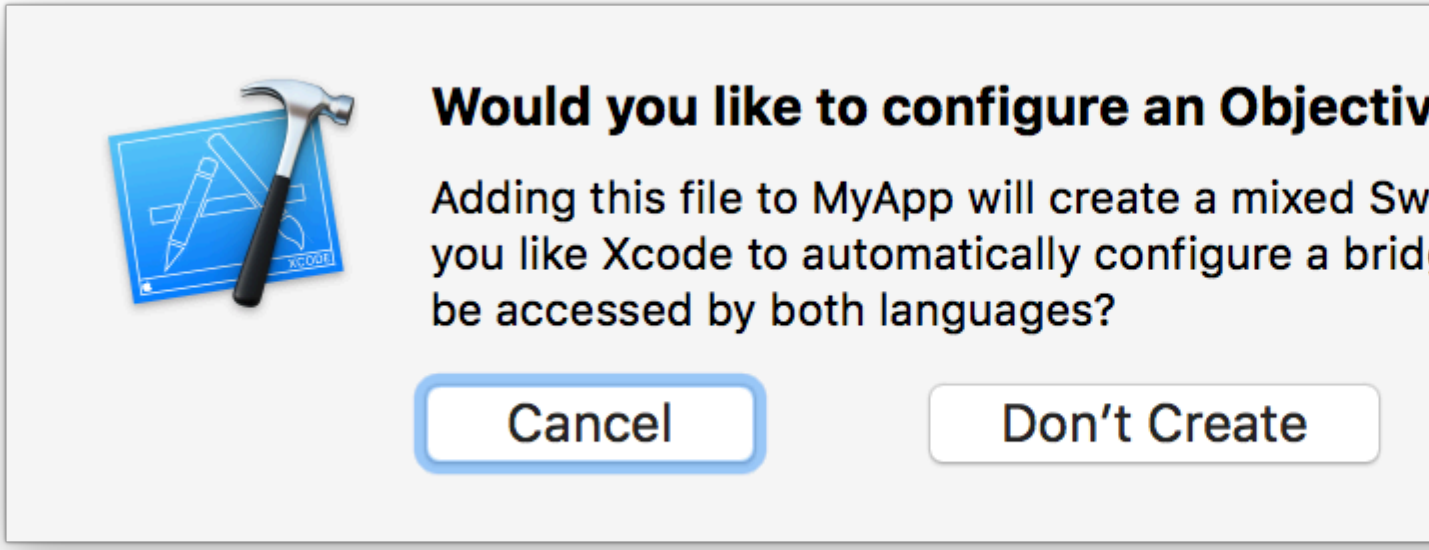
Uso di **@import MyFramework;** importa l'intero modulo, comprese le interfacce Obj-C per le classi Swift (se è abilitata la suddetta impostazione di build).

Utilizzo delle classi Objective-C dal codice Swift

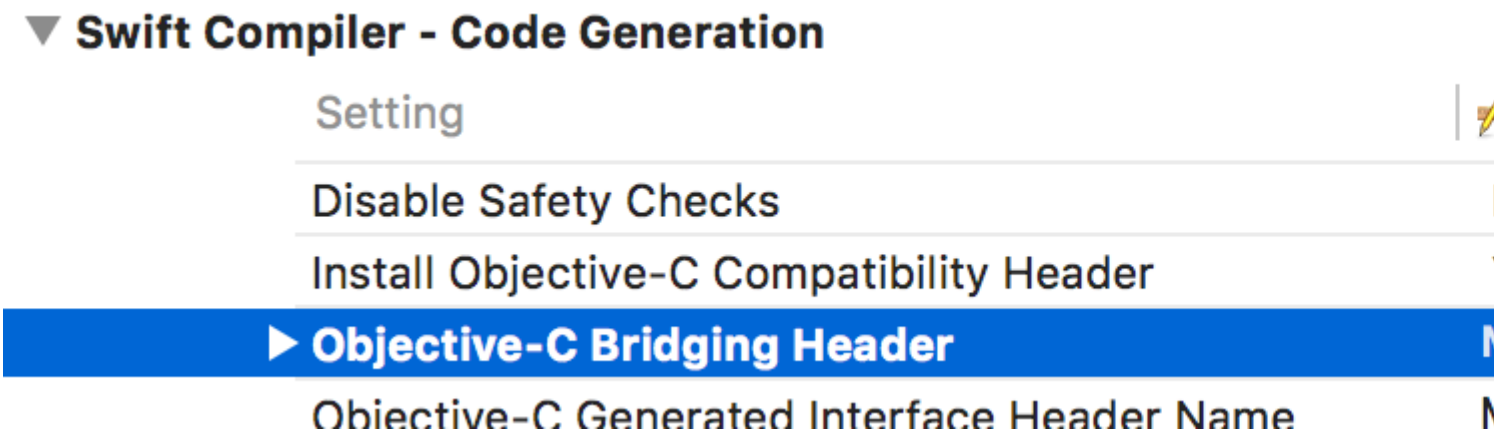
Se MyFramework contiene classi Objective-C nelle intestazioni pubbliche (e nell'intestazione dell'ombrello), quindi **import MyFramework** è tutto ciò che è necessario per utilizzarli da Swift.

Colpire le intestazioni

Un'intestazione di bridging rende visibili ulteriori dichiarazioni Objective-C e C al codice Swift. Quando si aggiungono file di progetto, Xcode può offrire di creare automaticamente un'intestazione di bridging:



Per crearne uno manualmente, modifica l'impostazione di costruzione **dell'intestazione di Bridging Objective-C** :

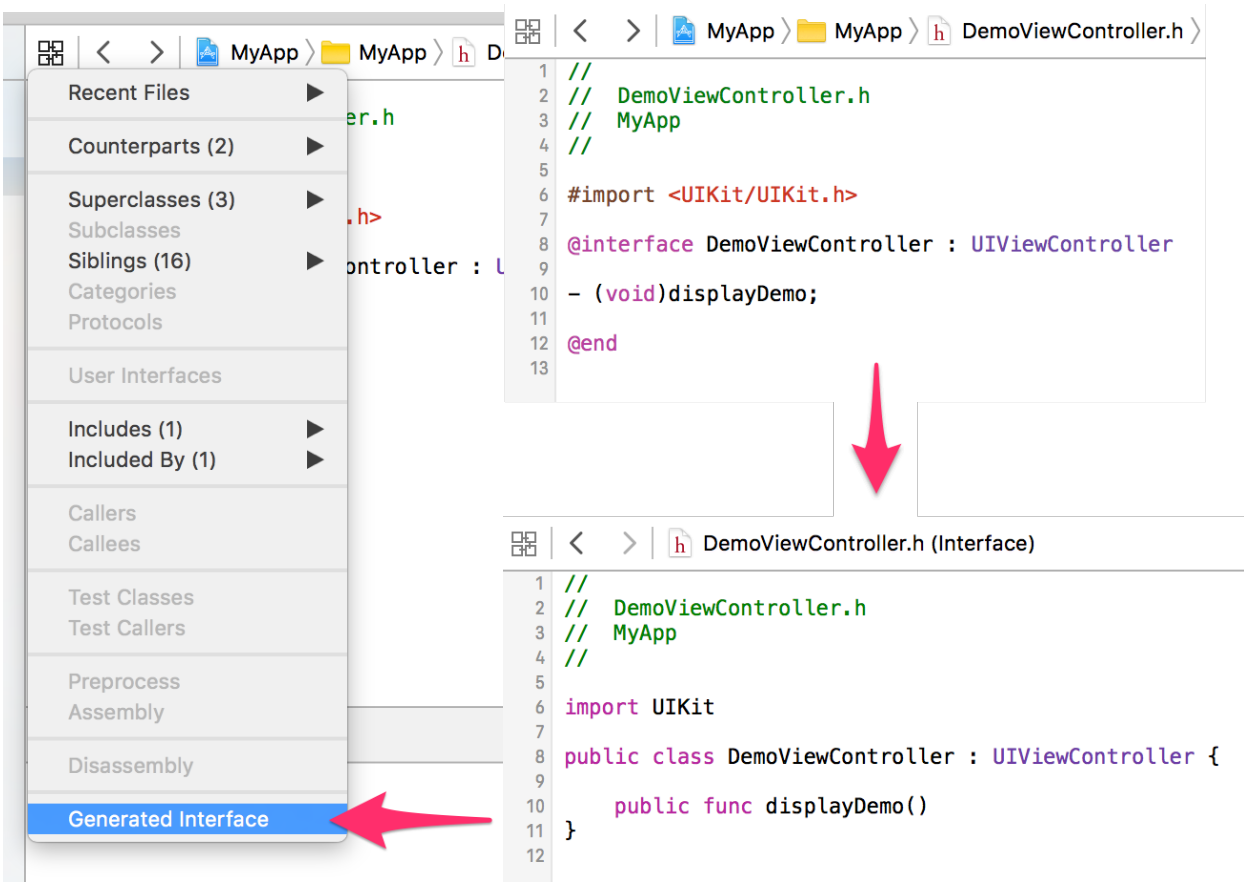


All'interno dell'intestazione del bridging, importa qualsiasi file sia necessario utilizzare dal codice:

```
// MyApp-Bridging-Header.h
#import "MyClass.h"  // allows code in this module to use MyClass
```

Interfaccia generata

Fai clic sul pulsante Elementi correlati (o premi ^ 1), quindi seleziona **Interfaccia generata** per visualizzare l'interfaccia Swift che verrà generata da un'intestazione Objective-C.



Specifica un'intestazione di bridging per swiftc

Il -import-objc-header specifica un'intestazione per l'importazione da swiftc :

```
// defs.h
struct Color {
    int red, green, blue;
};

#define MAX_VALUE 255
```

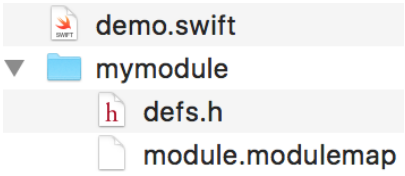
```
// demo.swift
extension Color: CustomStringConvertible {  // extension on a C struct
    public var description: String {
        return "Color(red: \(red), green: \(green), blue: \(blue))"
    }
}
print("MAX_VALUE is: \(MAX_VALUE)")  // C macro becomes a constant
let color = Color(red: 0xCA, green: 0xCA, blue: 0xD0)  // C struct initializer
print("The color is \(color)")
```

```
$ swiftc demo.swift -import-objc-header defs.h && ./demo
MAX_VALUE is: 255
The color is Color(red: 202, green: 202, blue: 208)
```

Utilizzare una mappa modulo per importare intestazioni C

Una **mappa modulo** può semplicemente import mymodule configurandolo per leggere i file header C e farli apparire come funzioni Swift.

Inserire un file denominato module.modulemap in una directory denominata mymodule :



All'interno del file di mappa del modulo:

```
// mymodule/module.modulemap
module mymodule {
    header "defs.h"
}
```

Quindi import il modulo:

```
// demo.swift
import mymodule
print("Empty color: \(Color())")
```

Usa il flag -I *directory* per dire a swiftc dove trovare il modulo:

```
swiftc -I . demo.swift   # "-I ." means "search for modules in the current directory"
```

Per ulteriori informazioni sulla sintassi della mappa dei moduli, consultare la documentazione Clang sulle mappe dei moduli .

Interoperabilità a grana fine tra Objective-C e Swift

Quando un'API è contrassegnata con NS_REFINED_FOR_SWIFT , verrà anteposta a due caratteri di sottolineatura (__) se importati in Swift:

```
@interface MyClass : NSObject
- (NSInteger)indexOfObject:(id)obj NS_REFINED_FOR_SWIFT;
@end
```

L' interfaccia generata si presenta così:

```
public class MyClass : NSObject {
    public func __indexOfObject(obj: AnyObject) -> Int
}
```

Ora puoi **sostituire l'API** con un'estensione più "Swifty". In questo caso, possiamo utilizzare un valore di ritorno opzionale , filtrando NSNotFound :

```
extension MyClass {
    // Rather than returning NSNotFound if the object doesn't exist,
    // this "refined" API returns nil.
    func indexOfObject(obj: AnyObject) -> Int? {
        let idx = __indexOfObject(obj)
        if idx == NSNotFound { return nil }
        return idx
    }
}

// Swift code, using "if let" as it should be:
let myobj = MyClass()
```

```
if let idx = myobj.indexOfObject(something) {
    // do something with idx
}
```

Nella maggior parte dei casi potresti voler limitare o meno un argomento a una funzione Objective-C che potrebbe essere nil . Questo viene fatto usando la parola chiave _Nonnull , che qualifica qualsiasi riferimento a puntatore o blocco:

```
void
doStuff(const void *const _Nonnull data, void (^_Nonnull completion)())
{
    // complex asynchronous code
}
```

Con quello scritto, il compilatore deve emettere un errore ogni volta che proviamo a passare nil a quella funzione dal nostro codice Swift:

```
doStuff(
    nil,  // error: nil is not compatible with expected argument type 'UnsafeRawPointer'
    nil)  // error: nil is not compatible with expected argument type '() -> Void'
```

L'opposto di _Nonnull è _Nullable , il che significa che è accettabile passare nil in questo argomento. _Nullable è anche il valore predefinito; tuttavia, specificarlo consente esplicitamente un codice più auto-documentato e a prova di futuro.

Per aiutare ulteriormente il compilatore a ottimizzare il tuo codice, potresti anche voler specificare se il blocco sta eseguendo l'escape:

```
void
callNow(__attribute__((noescape)) void (^_Nonnull f)())
{
    // f is not stored anywhere
}
```

Con questo attributo promettiamo di non salvare il riferimento del blocco e di non chiamare il blocco dopo che la funzione ha terminato l'esecuzione.

Utilizzare la libreria standard C

L'interoperabilità di Swift's C consente di utilizzare funzioni e tipi dalla libreria standard C.

Su Linux, la libreria standard C viene esposta tramite il modulo Glibc ; sulle piattaforme Apple si chiama Darwin .

```
#if os(macOS) || os(iOS) || os(tvOS) || os(watchOS)
import Darwin
#elseif os(Linux)
import Glibc
#endif

// use open(), read(), and other libc features
```

Leggi Lavorare con C e Objective-C online: https://riptutorial.com/it/swift/topic/421/lavorare-con-c-e-objective-c

Capitolo 37: Le tuple

introduzione

Un tipo di tupla è un elenco di tipi separati da virgola, racchiusi tra parentesi.

Questo elenco di tipi può anche avere il nome degli elementi e usare quei nomi per fare riferimento ai valori dei singoli elementi.

Il nome di un elemento è costituito da un identificatore seguito immediatamente da due punti (:).

Uso comune -

Possiamo usare un tipo di tupla come tipo di ritorno di una funzione per consentire alla funzione di restituire una singola tupla contenente più valori

Osservazioni

Le tuple sono considerate tipi di valore. Maggiori informazioni sulle tuple sono disponibili nella documentazione:
developer.apple.com/library/content/documentation/Swift/Conceptual/Swift_Programming_Language/TheBasics.

Examples

Cosa sono le tuple?

Le tuple raggruppano più valori in un singolo valore composto. I valori all'interno di una tupla possono essere di qualsiasi tipo e non devono essere dello stesso tipo l'uno dell'altro.

Le tuple vengono create raggruppando qualsiasi quantità di valori:

```
let tuple = ("one", 2, "three")

// Values are read using index numbers starting at zero
print(tuple.0) // one
print(tuple.1) // 2
print(tuple.2) // three
```

Anche i singoli valori possono essere nominati quando viene definita la tupla:

```
let namedTuple = (first: 1, middle: "dos", last: 3)

// Values can be read with the named property
print(namedTuple.first)  // 1
print(namedTuple.middle) // dos

// And still with the index number
print(namedTuple.2)      // 3
```

Possono anche essere nominati quando vengono utilizzati come variabili e persino avere la possibilità di avere valori opzionali all'interno:

```
var numbers: (optionalFirst: Int?, middle: String, last: Int)?

//Later On
numbers = (nil, "dos", 3)

print(numbers.optionalFirst)// nil
```

```
print(numbers.middle)//"dos"
print(numbers.last)//3
```

Scomponendosi in singole variabili

Le tuple possono essere scomposte in singole variabili con la seguente sintassi:

```
let myTuple = (name: "Some Name", age: 26)
let (first, second) = myTuple

print(first) // "Some Name"
print(second)  // 26
```

Questa sintassi può essere utilizzata indipendentemente dal fatto che la tupla abbia proprietà senza nome:

```
let unnamedTuple = ("uno", "dos")
let (one, two) = unnamedTuple
print(one) // "uno"
print(two) // "dos"
```

Le proprietà specifiche possono essere ignorate utilizzando il carattere di sottolineatura (_):

```
let longTuple = ("ichi", "ni", "san")
let (_, _, third) = longTuple
print(third) // "san"
```

Tuple come valore di ritorno delle funzioni

Le funzioni possono restituire tuple:

```
func tupleReturner() -> (Int, String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.0) // 3
print(myTuple.1) // "Hello"
```

Se assegni i nomi dei parametri, possono essere utilizzati dal valore restituito:

```
func tupleReturner() -> (anInteger: Int, aString: String) {
    return (3, "Hello")
}

let myTuple = tupleReturner()
print(myTuple.anInteger) // 3
print(myTuple.aString) // "Hello"
```

Usando un typealias per nominare il tuo tipo di tupla

Occasionalmente potresti voler usare lo stesso tipo di tupla in più punti del codice. Questo può diventare rapidamente disordinato, specialmente se la tua tupla è complessa:

```
// Define a circle tuple by its center point and radius
```

```
let unitCircle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat) = ((0.0, 0.0), 1.0)

func doubleRadius(ofCircle circle: (center: (x: CGFloat, y: CGFloat), radius: CGFloat)) ->
(center: (x: CGFloat, y: CGFloat), radius: CGFloat) {
    return (circle.center, circle.radius * 2.0)
}
```

Se si utilizza un determinato tipo di tupla in più di una posizione, è possibile utilizzare la parola chiave typealias per denominare il tipo di tupla.

```
// Define a circle tuple by its center point and radius
typealias Circle = (center: (x: CGFloat, y: CGFloat), radius: CGFloat)

let unitCircle: Circle = ((0.0, 0.0), 1)

func doubleRadius(ofCircle circle: Circle) -> Circle {
    // Aliased tuples also have access to value labels in the original tuple type.
    return (circle.center, circle.radius * 2.0)
}
```

Se ti accorgi di farlo troppo spesso, dovresti prendere in considerazione l'utilizzo di una struct .

Scambiare valori

Le tuple sono utili per scambiare valori tra 2 (o più) variabili senza utilizzare variabili temporanee.

Esempio con 2 variabili

Dato 2 variabili

```
var a = "Marty McFly"
var b = "Emmett Brown"
```

possiamo facilmente scambiare i valori

```
(a, b) = (b, a)
```

Risultato:

```
print(a) // "Emmett Brown"
print(b) // "Marty McFly"
```

Esempio con 4 variabili

```
var a = 0
var b = 1
var c = 2
var d = 3

(a, b, c, d) = (d, c, b, a)

print(a, b, c, d) // 3, 2, 1, 0
```

Tuple come Case in Switch

Usa le tuple in un interruttore

```
let switchTuple = (firstCase: true, secondCase: false)

switch switchTuple {
 case (true, false):
    // do something
 case (true, true):
    // do something
 case (false, true):
    // do something
 case (false, false):
    // do something
}
```

O in combinazione con un Enum Ad esempio con Classi di Dimensione:

```
    let switchTuple = (UIUserInterfaceSizeClass.Compact, UIUserInterfaceSizeClass.Regular)

    switch switchTuple {
    case (.Regular, .Compact):
        //statement
    case (.Regular, .Regular):
        //statement
    case (.Compact, .Regular):
        //statement
    case (.Compact, .Compact):
        //statement
    }
```

Leggi Le tuple online: https://riptutorial.com/it/swift/topic/574/le-tuple

Capitolo 38: Lettura e scrittura JSON

Sintassi

- NSJSONSerialization.JSONObjectWithData (jsonData, options: NSJSONReadingOptions) // Restituisce un oggetto da jsonData. Questo metodo si basa sul fallimento.
- NSJSONSerialization.dataWithJSONObject (jsonObject, options: NSJSONWritingOptions) // Restituisce NSData da un oggetto JSON. Passa a NSJSONWritingOptions.PrettyStampato in opzioni per un output più leggibile.

Examples

Serializzazione, codifica e decodifica JSON con Apple Foundation e Swift Standard Library

La classe JSONSerialization è integrata nel framework Foundation di Apple.

2.2

Leggi JSON

La funzione JSONObjectWithData prende NSData e restituisce AnyObject . Puoi usare as? per convertire il risultato nel tipo atteso.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".dataUsingEncoding(NSUTF8StringEncoding) else
{
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try NSJSONSerialization.JSONObjectWithData(jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joinWithSeparator(", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}
```

Puoi passare le options: .AllowFragments invece delle options: [] per consentire la lettura di JSON quando l'oggetto di livello superiore non è un array o un dizionario.

Scrivi JSON

Chiamando dataWithJSONObject converte un oggetto compatibile con JSON (array nidificati o dizionari con stringhe, numeri e NSNull) su NSData raw codificato come UTF-8.

```
do {
    // Convert object to JSON as NSData
    let jsonData = try NSJSONSerialization.dataWithJSONObject(jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: NSUTF8StringEncoding)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

È possibile passare options: .PrettyPrinted anziché options: [] per la stampa carina.

3.0

Stesso comportamento in Swift 3 ma con una sintassi diversa.

```
do {
    guard let jsonData = "[\"Hello\", \"JSON\"]".data(using: String.Encoding.utf8) else {
        fatalError("couldn't encode string as UTF-8")
    }

    // Convert JSON from NSData to AnyObject
    let jsonObject = try JSONSerialization.jsonObject(with: jsonData, options: [])

    // Try to convert AnyObject to array of strings
    if let stringArray = jsonObject as? [String] {
        print("Got array of strings: \(stringArray.joined(separator: ", "))")
    }
} catch {
    print("error reading JSON: \(error)")
}

do {
    // Convert object to JSON as NSData
    let jsonData = try JSONSerialization.data(withJSONObject: jsonObject, options: [])
    print("JSON data: \(jsonData)")

    // Convert NSData to String
    let jsonString = String(data: jsonData, encoding: .utf8)!
    print("JSON string: \(jsonString)")
} catch {
    print("error writing JSON: \(error)")
}
```

> Nota: il seguente è attualmente disponibile solo in **Swift 4.0** e versioni successive.

A partire da Swift 4.0, la libreria standard di Swift include i protocolli Encodable e Decodable per definire un approccio standardizzato alla codifica e alla decodifica dei dati. L'adozione di questi protocolli consentirà implementazioni dei protocolli Encoder e Decoder prendere i dati e codificarli o decodificarli da e verso una rappresentazione esterna come JSON. La conformità al protocollo Codable combina entrambi i protocolli Encodable e Decodable . Questo è ora il mezzo raccomandato per gestire JSON nel tuo programma.

Codifica e decodifica automaticamente

Il modo più semplice per rendere un tipo codificabile è dichiarare le sue proprietà come tipi già Codable . Questi tipi includono tipi di libreria standard come String , Int e Double ; e tipi di base come Date , Data e URL . Se le proprietà di un tipo sono codificabili, il tipo stesso si conformerà automaticamente a Codable semplicemente dichiarando la conformità.

Si consideri il seguente esempio, in cui la struttura del Book è conforme al Codable .

```
struct Book: Codable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

> Nota che le raccolte standard come Array e Dictionary conformi a Codable se contengono tipi codificabili.

Adottando Codable , la struttura del Book può ora essere codificata e decodificata da JSON

utilizzando le classi di Apple Foundation JSONEncoder e JSONDecoder , anche se Book stesso non contiene codice per gestire specificamente JSON. Anche i codificatori e decodificatori personalizzati possono essere scritti, rispettando i protocolli Encoder e Decoder , rispettivamente.

Codifica dati JSON

```
// Create an instance of Book called book
let encoder = JSONEncoder()
let data = try! encoder.encode(book) // Do not use try! in production code
print(data)
```

> Imposta encoder.outputFormatting = .prettyPrinted per facilitare la lettura. ## Decodifica dai dati JSON

Decodifica dai dati JSON

```
// Retrieve JSON string from some source
let jsonData = jsonString.data(encoding: .utf8)!
let decoder = JSONDecoder()
let book = try! decoder.decode(Book.self, for: jsonData) // Do not use try! in production code
print(book)
```

> Nell'esempio sopra, Book.self informa il decodificatore del tipo a cui il JSON deve essere decodificato.

Codifica o decodifica in esclusiva

A volte potrebbe non essere necessario che i dati siano entrambi codificabili e decodificabili, ad esempio quando è necessario leggere solo i dati JSON da un'API o se il programma invia solo dati JSON a un'API.

Se si intende solo scrivere dati JSON, conformare il proprio tipo a Encodable .

```
struct Book: Encodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Se si intende solo leggere i dati JSON, conformare il proprio tipo a Decodable .

```
struct Book: Decodable {
    let title: String
    let authors: [String]
    let publicationDate: Date
}
```

Utilizzo dei nomi chiave personalizzati

Le API utilizzano spesso convenzioni di denominazione diverse dal caso cammello standard Swift, ad esempio il caso serpente. Questo può diventare un problema quando si tratta di decodificare JSON, poiché per impostazione predefinita le chiavi JSON devono essere allineate esattamente con i nomi delle proprietà del tipo. Per gestire questi scenari è possibile creare chiavi personalizzate per il proprio tipo utilizzando il protocollo CodingKey .

```
struct Book: Codable {
    // ...
    enum CodingKeys: String, CodingKey {
        case title
```

```
        case authors
        case publicationDate = "publication_date"
    }
}
```

CodingKeys vengono generati automaticamente per i tipi che adottano il protocollo Codable , ma creando la nostra implementazione nell'esempio precedente, permettiamo al nostro decodificatore di abbinare la publicationDate caso cammello publicationDate con il caso snake publication_date come viene fornito dall'API.

SwiftyJSON

SwiftyJSON è un framework Swift creato per rimuovere la necessità di concatenamento opzionale nella normale serializzazione JSON.

Puoi scaricarlo qui: https://github.com/SwiftyJSON/SwiftyJSON

Senza SwiftyJSON, il tuo codice sarebbe simile a questo per trovare il nome del primo libro in un oggetto JSON:

```
if let jsonObject = try NSJSONSerialization.JSONObjectWithData(data, options: .AllowFragments)
as? [[String: AnyObject]],
let bookName = (jsonObject[0]["book"] as? [String: AnyObject])?["name"] as? String {
    //We can now use the book name
}
```

In SwiftyJSON, questo è estremamente semplificato:

```
let json = JSON(data: data)
if let bookName = json[0]["book"]["name"].string {
    //We can now use the book name
}
```

Rimuove la necessità di controllare ogni campo, in quanto restituirà nulla se nessuno di essi è valido.

Per usare SwiftyJSON, scarica la versione corretta dal repository Git - c'è un ramo per Swift 3. Basta trascinare il "SwiftyJSON.swift" nel tuo progetto e importarlo nella tua classe:

```
import SwiftyJSON
```

Puoi creare il tuo oggetto JSON usando i seguenti due inizializzatori:

```
let jsonObject = JSON(data: dataObject)
```

o

```
let jsonObject = JSON(jsonObject) //This could be a string in a JSON format for example
```

Per accedere ai tuoi dati, usa gli indici:

```
let firstObjectInAnArray = jsonObject[0]
let nameOfFirstObject = jsonObject[0]["name"]
```

È quindi possibile analizzare il valore su un determinato tipo di dati, che restituirà un valore facoltativo:

```
let nameOfFirstObject = jsonObject[0]["name"].string //This will return the name as a string
```

```
let nameOfFirstObject = jsonObject[0]["name"].double //This will return null
```

Puoi anche compilare i tuoi percorsi in una matrice rapida:

```
let convolutedPath = jsonObject[0]["name"][2]["lastName"]["firstLetter"].string
```

Equivale a:

```
let convolutedPath = jsonObject[0, "name", 2, "lastName", "firstLetter"].string
```

SwiftyJSON ha anche funzionalità per stampare i propri errori:

```
if let name = json[1337].string {
    //You can use the value - it is valid
} else {
    print(json[1337].error) // "Array[1337] is out of bounds" - You cant use the value
}
```

Se hai bisogno di scrivere sul tuo oggetto JSON, puoi usare di nuovo gli abbonati:

```
var originalJSON:JSON = ["name": "Jack", "age": 18]
originalJSON["age"] = 25 //This changes the age to 25
originalJSON["surname"] = "Smith" //This creates a new field called "surname" and adds the
value to it
```

Se hai bisogno della stringa originale per il JSON, ad esempio se devi scriverlo in un file, puoi ottenere il valore grezzo:

```
if let string = json.rawString() { //This is a String object
    //Write the string to a file if you like
}

if let data = json.rawData() { //This is an NSData object
    //Send the data to your server if you like
}
```

Freddy

Freddy è una libreria di analisi JSON gestita da Big Nerd Ranch . Ha tre principali vantaggi:

1. Sicurezza del tipo: ti aiuta a lavorare con l'invio e la ricezione di JSON in modo da prevenire arresti anomali del runtime.
2. Idiomatico: sfrutta i generici, le enumerazioni e le funzionalità funzionali di Swift, senza complicata documentazione o magici operatori personalizzati.
3. Gestione degli errori: fornisce informazioni di errore informative per gli errori JSON comunemente presenti.

Esempio di dati JSON

Definiamo alcuni dati JSON di esempio da utilizzare con questi esempi.

```
{
  "success": true,
  "people": [
```

```
    {
      "name": "Matt Mathias",
      "age": 32,
      "spouse": true
    },
    {
      "name": "Sergeant Pepper",
      "age": 25,
      "spouse": false
    }
  ],
  "jobs": [
    "teacher",
    "judge"
  ],
  "states": {
    "Georgia": [
      30301,
      30302,
      30303
    ],
    "Wisconsin": [
      53000,
      53001
    ]
  }
}
```

```
let jsonString = "{\"success\": true, \"people\": [{\"name\": \"Matt Mathias\",\"age\":
32,\"spouse\": true},{\"name\": \"Sergeant Pepper\",\"age\": 25,\"spouse\": false}],\"jobs\":
[\"teacher\",\"judge\"],\"states\": {\"Georgia\": [30301,30302,30303],\"Wisconsin\":
[53000,53001]}}"
let jsonData = jsonString.dataUsingEncoding(NSUTF8StringEncoding)!
```

Deserializzazione dei dati grezzi

Per deserializzare i dati, inizializziamo un oggetto JSON quindi accediamo a una particolare chiave.

```
do {
    let json = try JSON(data: jsonData)
    let success = try json.bool("success")
} catch {
    // do something with the error
}
```

try qui perché l'accesso a json per la chiave "success" potrebbe fallire - potrebbe non esistere, o il valore potrebbe non essere un booleano.

Possiamo anche specificare un percorso per accedere agli elementi nidificati nella struttura JSON. Il percorso è un elenco separato da virgole di chiavi e indici che descrive il percorso verso un valore di interesse.

```
do {
    let json = try JSON(data: jsonData)
    let georgiaZipCodes = try json.array("states", "Georgia")
    let firstPersonName = try json.string("people", 0, "name")
} catch {
    // do something with the error
}
```

Deserializzare i modelli direttamente

JSON può essere analizzato direttamente in una classe del modello che implementa il protocollo JSONDecodable .

```
public struct Person {
    public let name: String
    public let age: Int
    public let spouse: Bool
}

extension Person: JSONDecodable {
    public init(json: JSON) throws {
        name = try json.string("name")
        age = try json.int("age")
        spouse = try json.bool("spouse")
    }
}

do {
    let json = try JSON(data: jsonData)
    let people = try json.arrayOf("people", type: Person.self)
} catch {
    // do something with the error
}
```

Serializzazione di dati grezzi

Qualsiasi valore JSON può essere serializzato direttamente su NSData .

```
let success = JSON.Bool(false)
let data: NSData = try success.serialize()
```

Serializzare i modelli direttamente

Qualsiasi classe di modello che implementa il protocollo JSONEncodable può essere serializzata direttamente su NSData .

```
extension Person: JSONEncodable {
    public func toJSON() -> JSON {
        return .Dictionary([
            "name": .String(name),
            "age": .Int(age),
            "spouse": .Bool(spouse)
        ])
    }
}

let newPerson = Person(name: "Glenn", age: 23, spouse: true)
let data: NSData = try newPerson.toJSON().serialize()
```

Freccia

Arrow è un'elegante libreria di analisi JSON in Swift.

Permette di analizzare JSON e associarlo a classi di modelli personalizzati con l'aiuto di un operatore <-- :

```
identifier <-- json["id"]
```

```
name <-- json["name"]
stats <-- json["stats"]
```

Esempio:

Modello Swift

```
struct Profile {
    var identifier = 0
    var name = ""
    var link: NSURL?
    var weekday: WeekDay = .Monday
    var stats = Stats()
    var phoneNumbers = [PhoneNumber]()
}
```

File JSON

```
{
    "id": 15678,
    "name": "John Doe",
    "link": "https://apple.com/steve",
    "weekdayInt" : 3,
    "stats": {
        "numberOfFriends": 163,
        "numberOfFans": 10987
    },
    "phoneNumbers": [{
                     "label": "house",
                     "number": "9809876545"
                     }, {
                     "label": "cell",
                     "number": "0908070656"
                     }, {
                     "label": "work",
                     "number": "0916570656"
    }]
}
```

Mappatura

```
extension Profile: ArrowParsable {
    mutating func deserialize(json: JSON) {
        identifier <-- json["id"]
        link <-- json["link"]
        name <-- json["name"]
        weekday <-- json["weekdayInt"]
        stats <- json["stats"]
        phoneNumbers <-- json["phoneNumbers"]
    }
}
```

uso

```
let profile = Profile()
profile.deserialize(json)
```

Installazione:

Carthage

```
github "s4cha/Arrow"
```

CocoaPods

```
pod 'Arrow'
use_frameworks!
```

manualmente

Basta copiare e incollare Arrow.swift nel tuo progetto Xcode

https://github.com/s4cha/Arrow

Come un quadro

Scarica Arrow dal repository GitHub e crea l'obiettivo Framework nel progetto di esempio. Quindi Link contro questo framework.

Semplice JSON che analizza gli oggetti personalizzati

Anche se le librerie di terze parti sono buone, un semplice modo per analizzare il JSON è fornito da protocolli. Puoi immaginare di avere un oggetto Todo come

```
struct Todo {
    let comment: String
}
```

Ogni volta che si riceve il JSON, è possibile gestire il normale NSData come mostrato nell'altro esempio usando NSJSONSerialization oggetto NSJSONSerialization .

Successivamente, utilizzando un semplice protocollo JSONDecodable

```
typealias JSONDictionary = [String:AnyObject]
protocol JSONDecodable {
    associatedtype Element
    static func from(json json: JSONDictionary) -> Element?
}
```

E il trucco è rendere la struttura di Todo conforme a JSONDecodable

```
extension Todo: JSONDecodable {
    static func from(json json: JSONDictionary) -> Todo? {
        guard let comment = json["comment"] as? String else { return nil }
        return Todo(comment: comment)
    }
}
```

Puoi provarlo con questo codice JSON:

```
{
    "todos": [
        {
            "comment" : "The todo comment"
        }
    ]
}
```

Quando l'hai ottenuto dall'API, puoi serializzarlo come gli esempi precedenti mostrati in un'istanza AnyObject . Successivamente, è possibile verificare se l'istanza è un'istanza di JSONDictionary

```
guard let jsonDictionary = dictionary as? JSONDictionary else { return }
```

L'altra cosa da controllare, specifica per questo caso perché hai un array di Todo nel JSON, è il dizionario di todos i todos

```
guard let todosDictionary = jsonDictionary["todos"] as? [JSONDictionary] else { return }
```

Ora che hai la gamma di dizionari, puoi convertirli in un oggetto Todo usando flatMap (cancellerà automaticamente i valori nil dall'array)

```
let todos: [Todo] = todosDictionary.flatMap { Todo.from(json: $0) }
```

JSON Parsing Swift 3

Ecco il file JSON che useremo chiamato animals.json

```
{
    "Sea Animals": [
    {
     "name": "Fish",
     "question": "How many species of fish are there?"    },
                {
                "name": "Sharks",
                "question": "How long do sharks live?"
                },
                {
                "name": "Squid",
                "question": "Do squids have brains?"
                },
                {
                "name": "Octopus",
                "question": "How big do octopus get?"
                },
                {
                "name": "Star Fish",
                "question": "How long do star fish live?"
                }
                ],
    "mammals": [
                {
      "name": "Dog",
      "question": "How long do dogs live?"
    },
    {
      "name": "Elephant",
      "question": "How much do baby elephants weigh?"
    },
    {
      "name": "Cats",
      "question": "Do cats really have 9 lives?"
    },
    {
      "name": "Tigers",
      "question": "Where do tigers live?"
    },
    {
      "name": "Pandas",
      "question": "WHat do pandas eat?"
    }   ] }
```

Importa il tuo file JSON nel tuo progetto

È possibile eseguire questa semplice funzione per stampare il file JSON

```
   func jsonParsingMethod() {
        //get the file
        let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
        let content = try! String(contentsOfFile: filePath!)

        let data: Data = content.data(using: String.Encoding.utf8)!
        let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//Call which part of the file you'd like to pare
        if let results = json["mammals"] as? [[String: AnyObject]] {

            for res in results {
                //this will print out the names of the mammals from out file.
                if let rates = res["name"] as? String {
                    print(rates)
                }
           }
       }
    }
```

Se vuoi metterlo in una vista tabella, vorrei prima creare un dizionario con un NSObject.

Crea un nuovo file rapido chiamato ParsingObject e crea le tue variabili stringa.

     Assicurarsi che il nome della variabile sia lo stesso del file JSON

. Ad esempio, nel nostro progetto abbiamo name e question così nel nostro nuovo file swift, che useremo

```
var name: String?
var question: String?
```

Inizializza il NSObject che abbiamo creato nel nostro ViewController.swift var array = ParsingObject Quindi eseguiremo lo stesso metodo che avevamo prima con una piccola modifica.

```
     func jsonParsingMethod() {
            //get the file
            let filePath = Bundle.main.path(forResource: "animals", ofType: "json")
            let content = try! String(contentsOfFile: filePath!)

            let data: Data = content.data(using: String.Encoding.utf8)!
            let json: NSDictionary =  try! JSONSerialization.jsonObject(with: data as Data,
options:.mutableContainers) as! NSDictionary

//This time let's get Sea Animals
            let results = json["Sea Animals"] as? [[String: AnyObject]]

//Get all the stuff using a for-loop
            for i in 0 ..< results!.count {

//get the value
                let dict = results?[i]
                let resultsArray = ParsingObject()

//append the value to our NSObject file
                resultsArray.setValuesForKeys(dict!)
                array.append(resultsArray)
```

```
            }

        }
```

Quindi lo mostriamo nella nostra tableview facendo questo,

```
    func tableView(_ tableView: UITableView, numberOfRowsInSection section: Int) -> Int {
            return array.count
        }

        func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) ->
UITableViewCell {
            let cell = tableView.dequeueReusableCell(withIdentifier: "Cell", for: indexPath)
//This is where our values are stored
            let object = array[indexPath.row]
            cell.textLabel?.text = object.name
            cell.detailTextLabel?.text = object.question
            return cell
        }
```

Leggi Lettura e scrittura JSON online: https://riptutorial.com/it/swift/topic/223/lettura-e-scrittura-json

Capitolo 39: Loops

Sintassi

- per costante in sequenza {statement}
- per costante in sequenza dove condizione {statement}
- per variabile var in sequenza {statement}
- per _ in sequenza {istruzioni}
- per caso lascia costante in sequenza {statement}
- per caso lascia costante in sequenza dove condizione {statement}
- per caso var variable in sequence {statements}
- while condition {statement}
- ripeti {dichiarazioni} mentre la condizione
- sequence.forEach (body: (Element) throws -> Void)

Examples

Ciclo For-in

Il ciclo **for-in** consente di eseguire iterazioni su qualsiasi sequenza.

Iterare su un intervallo

Puoi scorrere su intervalli sia semiaperti che chiusi:

```
for i in 0..<3 {
    print(i)
}

for i in 0...2 {
    print(i)
}

// Both print:
// 0
// 1
// 2
```

Iterare su un array o un set

```
let names = ["James", "Emily", "Miles"]

for name in names {
   print(name)
}

// James
// Emily
// Miles
```

2.1 2.2

Se è necessario l'indice per ciascun elemento nell'array, è possibile utilizzare il metodo enumerate() su SequenceType .

```
for (index, name) in names.enumerate() {
```

```
   print("The index of \(name) is \(index).")
}

// The index of James is 0.
// The index of Emily is 1.
// The index of Miles is 2.
```

enumerate() restituisce una sequenza lazy che contiene coppie di elementi con Int consecutivi, a partire da 0. Pertanto con gli array, questi numeri corrisponderanno all'indice dato di ciascun elemento, tuttavia questo potrebbe non essere il caso di altri tipi di collezioni.

3.0

In Swift 3, enumerate() è stato rinominato enumerated() :

```
for (index, name) in names.enumerated() {
   print("The index of \(name) is \(index).")
}
```

Iterare su un dizionario

```
let ages = ["James": 29, "Emily": 24]

for (name, age) in ages {
    print(name, "is", age, "years old.")
}

// Emily is 24 years old.
// James is 29 years old.
```

Iterando al contrario

2.1 2.2

È possibile utilizzare il metodo reverse() su SequenceType per eseguire un'iterazione su qualsiasi sequenza al contrario:

```
for i in (0..<3).reverse() {
    print(i)
}

for i in (0...2).reverse() {
    print(i)
}

// Both print:
// 2
// 1
// 0

let names = ["James", "Emily", "Miles"]

for name in names.reverse() {
    print(name)
}

// Miles
// Emily
// James
```

3.0

In Swift 3, reverse() è stato rinominato in reverse() reversed() :

```
for i in (0..<3).reversed() {
    print(i)
}
```

Iterare su intervalli con passo personalizzato

2.1 2.2

Usando i metodi stride(_:_:) su Strideable puoi Strideable su un intervallo con un passo personalizzato:

```
for i in 4.stride(to: 0, by: -2) {
    print(i)
}

// 4
// 2

for i in 4.stride(through: 0, by: -2) {
    print(i)
}

// 4
// 2
// 0
```

1.2 3.0

In Swift 3, i metodi stride(_:_:) su Stridable sono stati sostituiti dalle funzioni stride(_:_:_:) globali stride(_:_:_:) :

```
for i in stride(from: 4, to: 0, by: -2) {
    print(i)
}

for i in stride(from: 4, through: 0, by: -2) {
    print(i)
}
```

Ciclo di ripetizione

Simile al ciclo while, solo l'istruzione di controllo viene valutata dopo il ciclo. Pertanto, il ciclo verrà sempre eseguito almeno una volta.

```
var i: Int = 0

repeat {
   print(i)
   i += 1
} while i < 3

// 0
// 1
// 2
```

mentre ciclo

Un ciclo while verrà eseguito fintanto che la condizione è vera.

```
var count = 1

while count < 10 {
    print("This is the \(count) run of the loop")
    count += 1
}
```

Tipo di sequenza per Ogni blocco

Un tipo conforme al protocollo SequenceType può scorrere gli elementi all'interno di una chiusura:

```
collection.forEach { print($0) }
```

Lo stesso potrebbe essere fatto anche con un parametro chiamato:

```
collection.forEach { item in
    print(item)
}
```

* Nota: le istruzioni di flusso di controllo (come interruzione o continuazione) non possono essere utilizzate in questi blocchi. È possibile chiamare un ritorno e, se chiamato, restituirà immediatamente il blocco per l'iterazione corrente (molto simile a un continuazione). La successiva iterazione verrà quindi eseguita.

```
let arr = [1,2,3,4]

arr.forEach {

    // blocks for 3 and 4 will still be called
    if $0 == 2 {
        return
    }
}
```

Ciclo For-in con filtraggio

1. **where** clausola

Con l'aggiunta di una where la clausola è possibile limitare le iterazioni a quelli che soddisfano la condizione data.

```
for i in 0..<5 where i % 2 == 0 {
    print(i)
}

// 0
// 2
// 4


let names = ["James", "Emily", "Miles"]

for name in names where name.characters.contains("s") {
    print(name)
```

```
}

// James
// Miles
```

2. clausola **case**

È utile quando è necessario iterare solo attraverso i valori che corrispondono ad alcuni pattern:

```
let points = [(5, 0), (31, 0), (5, 31)]
for case (_, 0) in points {
    print("point on x-axis")
}

//point on x-axis
//point on x-axis
```

Inoltre puoi filtrare i valori opzionali e scartarli se necessario aggiungendo ? segnare dopo il binding costante:

```
let optionalNumbers = [31, 5, nil]
for case let number? in optionalNumbers {
    print(number)
}

//31
//5
```

Rompere un ciclo

Un ciclo verrà eseguito fino a quando la sua condizione rimarrà true, ma è possibile interromperlo manualmente usando la parola chiave **break** . Per esempio:

```
var peopleArray = ["John", "Nicole", "Thomas", "Richard", "Brian", "Novak", "Vick", "Amanda",
"Sonya"]
var positionOfNovak = 0

for person in peopleArray {
    if person == "Novak" { break }
    positionOfNovak += 1
}

print("Novak is the element located on position [\(positionOfNovak)] in peopleArray.")
//prints out: Novak is the element located on position 5 in peopleArray. (which is true)
```

Leggi Loops online: https://riptutorial.com/it/swift/topic/1186/loops

Capitolo 40: Markup della documentazione

Examples

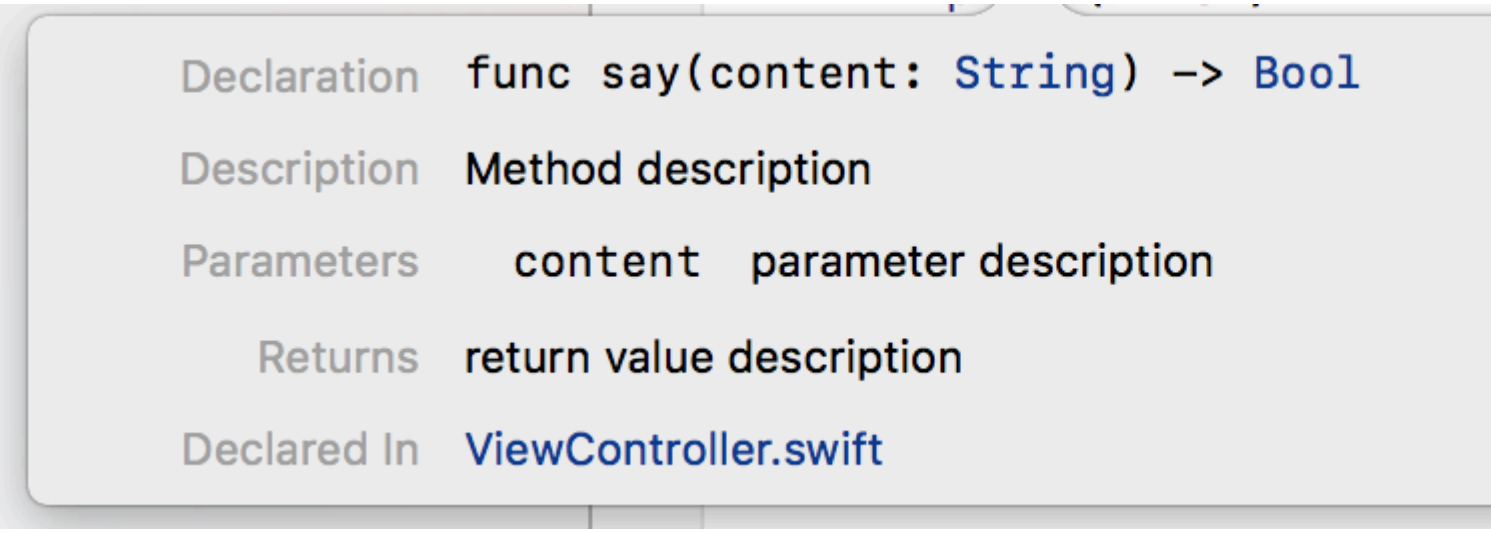
Documentazione di classe

Ecco un esempio di documentazione di classe base:

```
/// Class description
class Student {

    // Member description
    var name: String

    /// Method description
    ///
    /// - parameter content:   parameter description
    ///
    /// - returns: return value description
    func say(content: String) -> Bool {
        print("\(self.name) say \(content)")
        return true
    }
}
```

Nota che con *Xcode 8* puoi generare lo snippet di documentazione con comando + opzione + / .

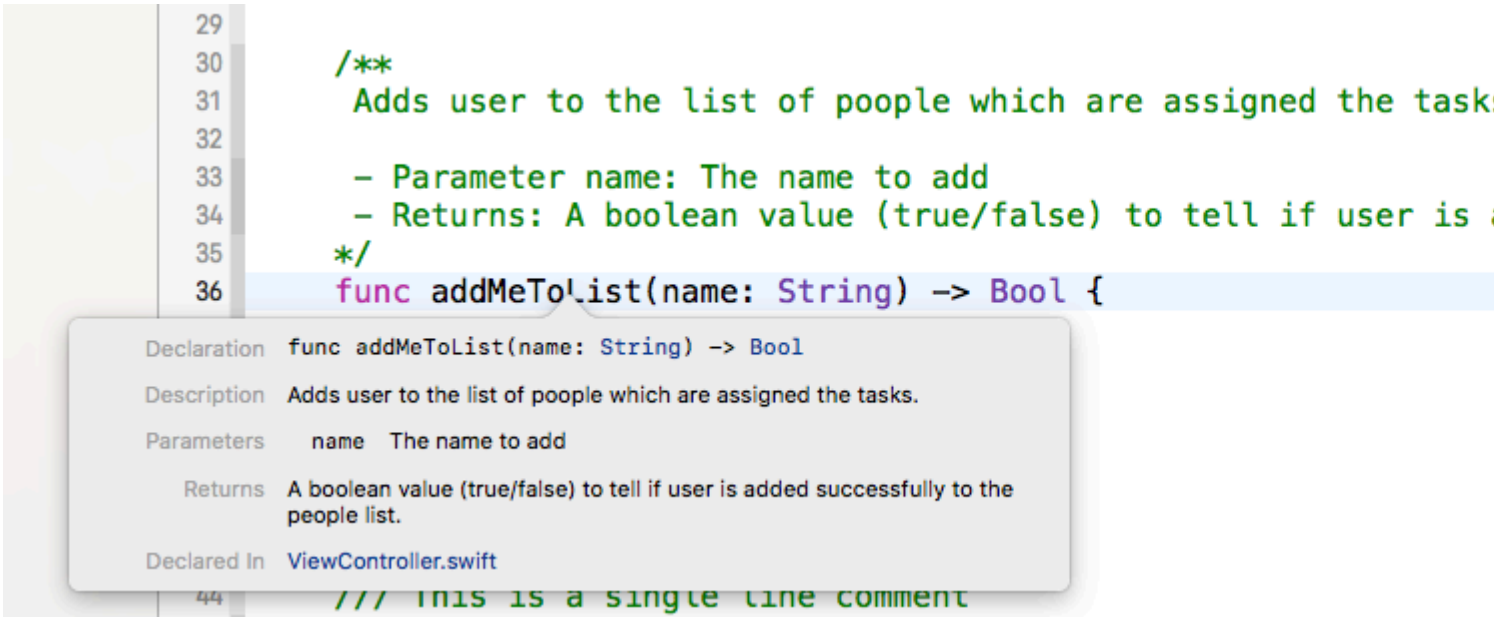
Questo ritornerà:



Stili di documentazione

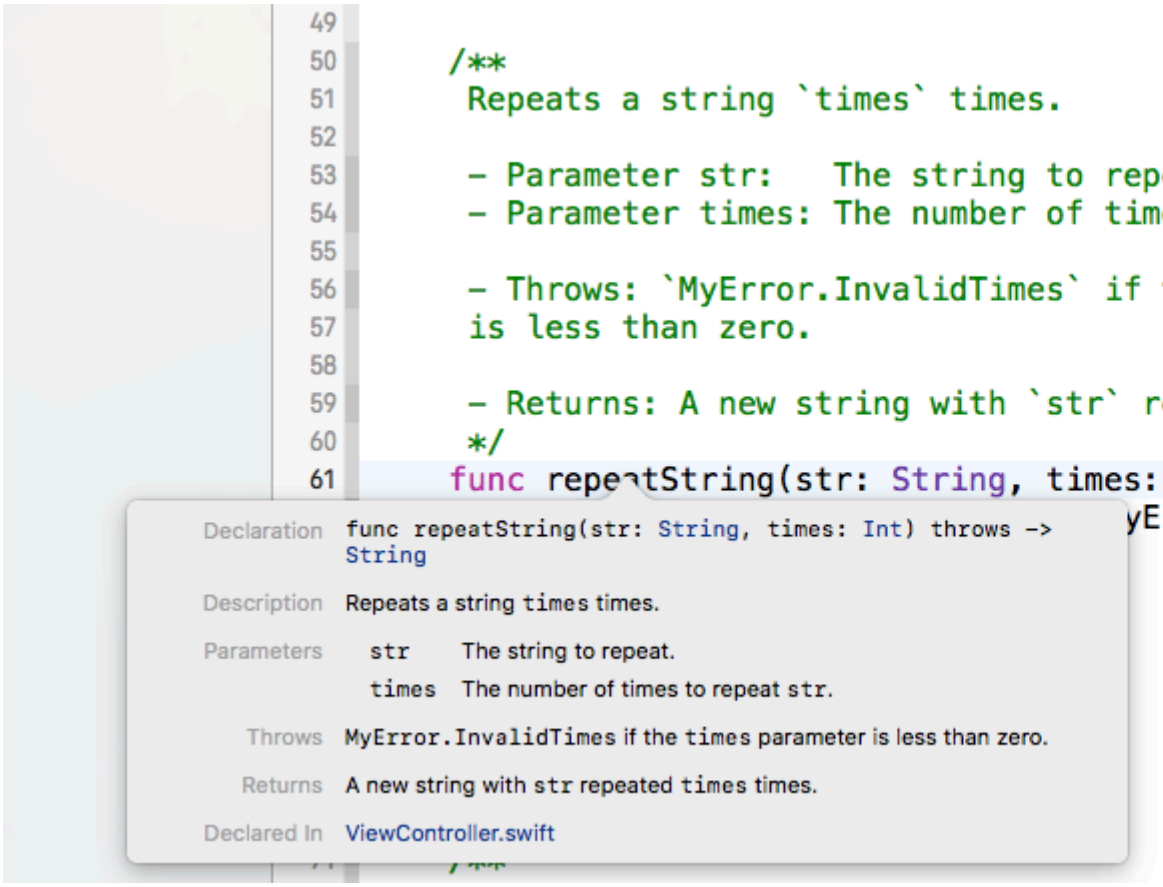
```
/**
 Adds user to the list of poople which are assigned the tasks.

 - Parameter name: The name to add
 - Returns: A boolean value (true/false) to tell if user is added successfully to the people
list.
*/
func addMeToList(name: String) -> Bool {

    // Do something....
```

```
    return true
}
```

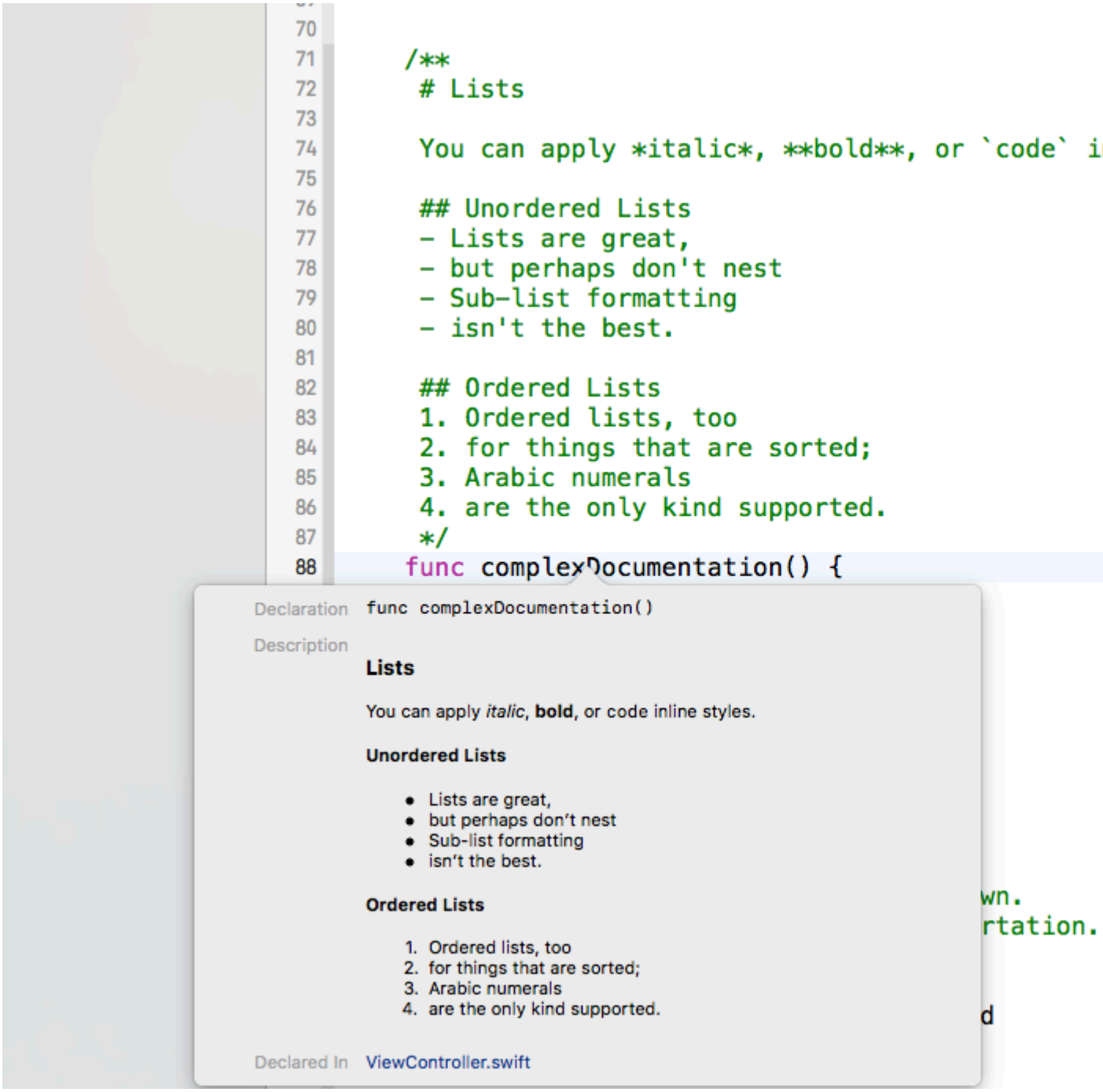



```
/// This is a single line comment
func singleLineComment() {

}
```

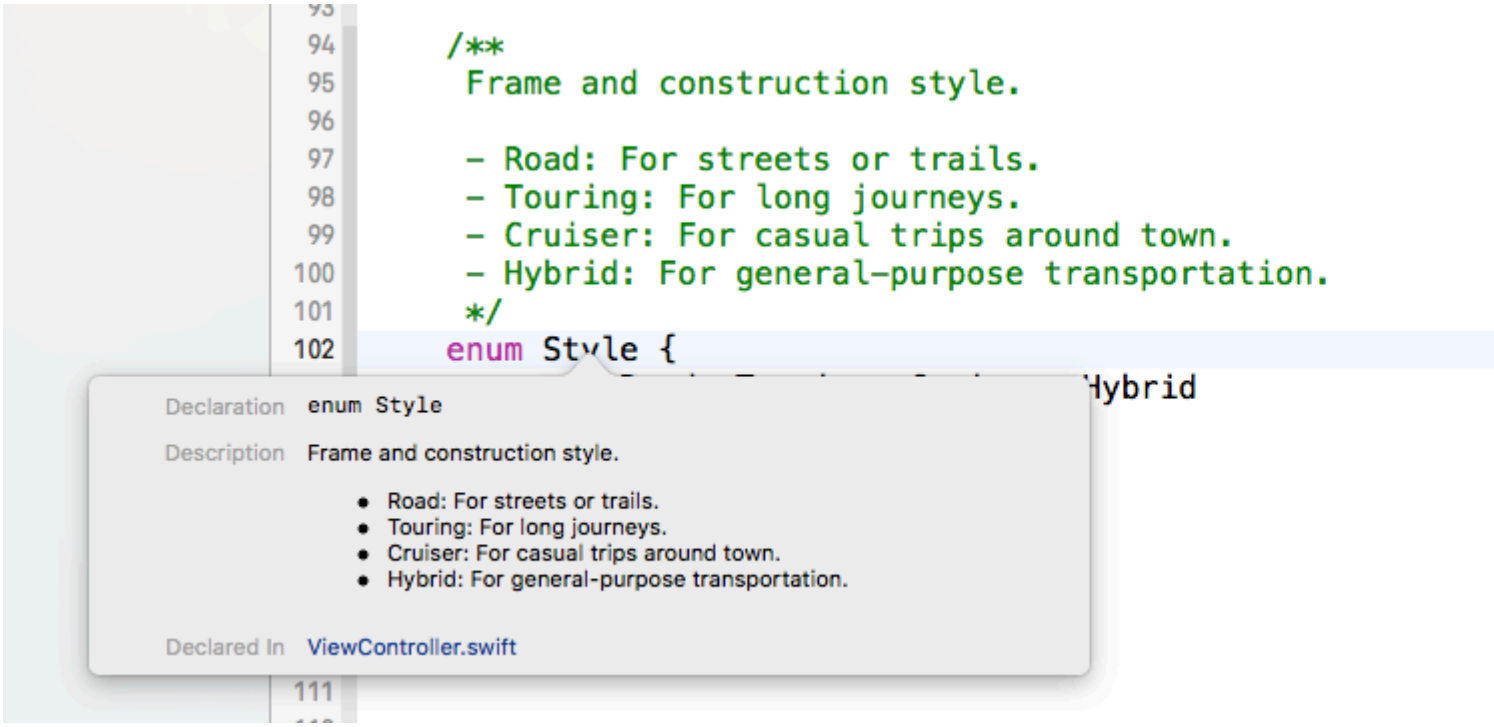

```
/**
 Repeats a string `times` times.

 - Parameter str:   The string to repeat.
 - Parameter times: The number of times to repeat `str`.

 - Throws: `MyError.InvalidTimes` if the `times` parameter
 is less than zero.

 - Returns: A new string with `str` repeated `times` times.
*/
func repeatString(str: String, times: Int) throws -> String {
    guard times >= 0 else { throw MyError.invalidTimes }
    return "Hello, world"
}
```




```
/**
 # Lists

 You can apply *italic*, **bold**, or `code` inline styles.

 ## Unordered Lists
 - Lists are great,
 - but perhaps don't nest
 - Sub-list formatting
 - isn't the best.

 ## Ordered Lists
 1. Ordered lists, too
 2. for things that are sorted;
 3. Arabic numerals
 4. are the only kind supported.
*/
func complexDocumentation() {

}
```




```
/**
 Frame and construction style.

 - Road: For streets or trails.
 - Touring: For long journeys.
 - Cruiser: For casual trips around town.
 - Hybrid: For general-purpose transportation.
*/
enum Style {
    case Road, Touring, Cruiser, Hybrid
}
```



Leggi Markup della documentazione online: https://riptutorial.com/it/swift/topic/6937/markup-della-documentazione

Capitolo 41: Memorizzazione nella cache dello spazio su disco

introduzione

Memorizzazione nella cache di video, immagini e audio utilizzando URLSession e FileManager

Examples

Salvataggio

```
let url = "https://path-to-media"
let request = URLRequest(url: url)
let downloadTask = URLSession.shared.downloadTask(with: request) { (location, response, error)
in
  guard let location = location,
        let response = response,
        let documentsPath = NSSearchPathForDirectoriesInDomains(.documentDirectory,
.userDomainMask, true).first else {
    return
  }
  let documentsDirectoryUrl = URL(fileURLWithPath: documentsPath)
  let documentUrl = documentsDirectoryUrl.appendingPathComponent(response.suggestedFilename)
  let _ = try? FileManager.default.moveItem(at: location, to: documentUrl)

  // documentUrl is the local URL which we just downloaded and saved to the FileManager
}.resume()
```

Lettura

```
let url = "https://path-to-media"
guard let documentsUrl = FileManager.default.urls(for: .documentDirectory, in:
.userDomainMask).first,
      let searchQuery = url.absoluteString.components(separatedBy: "/").last else {
  return nil
}

do {
  let directoryContents = try FileManager.default.contentsOfDirectory(at: documentsUrl,
includingPropertiesForKeys: nil, options: [])
  let cachedFiles = directoryContents.filter { $0.absoluteString.contains(searchQuery) }

  // do something with the files found by the url
} catch {
  // Could not find any files
}
```

Leggi Memorizzazione nella cache dello spazio su disco online:
https://riptutorial.com/it/swift/topic/8902/memorizzazione-nella-cache-dello-spazio-su-disco

Capitolo 42: Metodo Swizzling

Osservazioni

Quando usi il metodo swizzling in Swift ci sono due requisiti che le tue classi / metodi devono rispettare:

- La tua classe deve estendere NSObject
- Le funzioni che vuoi far girare devono avere l'attributo dynamic

Per una spiegazione completa del motivo per cui è necessario, consulta Usare Swift con Cocoa e Objective-C :

> **Richiesta di invio dinamico**
>
> Mentre l'attributo @objc espone la tua API Swift al runtime Objective-C, non garantisce l'invio dinamico di una proprietà, metodo, pedice o inizializzatore. *Il compilatore Swift può ancora devirtualizzare o accedere in linea ai membri per ottimizzare le prestazioni del codice, ignorando il runtime Objective-C* . Quando contrassegni una dichiarazione membro con il modificatore dynamic , l'accesso a quel membro viene sempre inviato dinamicamente. Poiché le dichiarazioni contrassegnate con il modificatore dynamic vengono inviate utilizzando il runtime Objective-C, vengono implicitamente contrassegnate con l'attributo @objc .
>
> Richiedere la spedizione dinamica è raramente necessario. **Tuttavia, devi utilizzare il modificatore dynamic quando sai che l'implementazione di un'API viene sostituita in fase di runtime** . Ad esempio, è possibile utilizzare la funzione method_exchangeImplementations nel runtime Objective-C per sostituire l'implementazione di un metodo mentre è in esecuzione un'app. Se il compilatore Swift ha delineato l'implementazione del metodo o l'accesso devirtualizzato ad esso, *la nuova implementazione non verrebbe utilizzata* .

link

Riferimento runtime Objective-C

Metodo Swizzling su NSHipster

Examples

Estensione di UIViewController e Swizzling viewDidLoad

In Objective-C, il metodo swizzling è il processo di modifica dell'implementazione di un selettore esistente. Ciò è possibile a causa del modo in cui i selettori sono mappati su una tabella di distribuzione o di una tabella di puntatori a funzioni o metodi.

I metodi Pure Swift non vengono inviati dinamicamente dal runtime Objective-C, ma possiamo ancora trarre vantaggio da questi trucchi su qualsiasi classe che eredita da NSObject .

Qui, estenderemo UIViewController e UIViewController viewDidLoad per aggiungere alcune registrazioni personalizzate:

```
extension UIViewController {

    // We cannot override load like we could in Objective-C, so override initialize instead
    public override static func initialize() {

        // Make a static struct for our dispatch token so only one exists in memory
        struct Static {
            static var token: dispatch_once_t = 0
```

```
        }

        // Wrap this in a dispatch_once block so it is only run once
        dispatch_once(&Static.token) {
            // Get the original selectors and method implementations, and swap them with our
new method
            let originalSelector = #selector(UIViewController.viewDidLoad)
            let swizzledSelector = #selector(UIViewController.myViewDidLoad)

            let originalMethod = class_getInstanceMethod(self, originalSelector)
            let swizzledMethod = class_getInstanceMethod(self, swizzledSelector)

            let didAddMethod = class_addMethod(self, originalSelector,
method_getImplementation(swizzledMethod), method_getTypeEncoding(swizzledMethod))

            // class_addMethod can fail if used incorrectly or with invalid pointers, so check
to make sure we were able to add the method to the lookup table successfully
            if didAddMethod {
                class_replaceMethod(self, swizzledSelector,
method_getImplementation(originalMethod), method_getTypeEncoding(originalMethod))
            } else {
                method_exchangeImplementations(originalMethod, swizzledMethod);
            }
        }
    }

    // Our new viewDidLoad function
    // In this example, we are just logging the name of the function, but this can be used to
run any custom code
    func myViewDidLoad() {
        // This is not recursive since we swapped the Selectors in initialize().
        // We cannot call super in an extension.
        self.myViewDidLoad()
        print(#function) // logs myViewDidLoad()
    }
}
```

Nozioni di base di Swift Swizzling

methodOne() l'implementazione di methodOne() e methodTwo() nella nostra classe TestSwizzling :

```
class TestSwizzling : NSObject {
    dynamic func methodOne()->Int{
        return 1
    }
}

extension TestSwizzling {

    //In Objective-C you'd perform the swizzling in load(),
    //but this method is not permitted in Swift
    override class func initialize()
    {

        struct Inner {
            static let i: () = {

                let originalSelector = #selector(TestSwizzling.methodOne)
                let swizzledSelector = #selector(TestSwizzling.methodTwo)
                let originalMethod = class_getInstanceMethod(TestSwizzling.self,
```

```
originalSelector);
                let swizzledMethod = class_getInstanceMethod(TestSwizzling.self,
swizzledSelector)
                method_exchangeImplementations(originalMethod, swizzledMethod)
            }
        }
        let _ = Inner.i
    }

    func methodTwo()->Int{
        // It will not be a recursive call anymore after the swizzling
        return methodTwo()+1
    }
}

var c = TestSwizzling()
print(c.methodOne())
print(c.methodTwo())
```

Nozioni di base di Swizzling - Objective-C

Esempio Objective-C di swizzling initWithFrame: metodo

```
static IMP original_initWithFrame;

+ (void)swizzleMethods {
    static BOOL swizzled = NO;
    if (!swizzled) {
        swizzled = YES;

        Method initWithFrameMethod =
            class_getInstanceMethod([UIView class], @selector(initWithFrame:));
        original_initWithFrame = method_setImplementation(
            initWithFrameMethod, (IMP)replacement_initWithFrame);
    }
}

static id replacement_initWithFrame(id self, SEL _cmd, CGRect rect) {

    // This will be called instead of the original initWithFrame method on UIView
    // Do here whatever you need...

    // Bonus: This is how you would call the original initWithFrame method
    UIView *view =
        ((id (*)(id, SEL, CGRect))original_initWithFrame)(self, _cmd, rect);

    return view;
}
```

Leggi Metodo Swizzling online: https://riptutorial.com/it/swift/topic/1436/metodo-swizzling

Capitolo 43: Modelli di design - Strutturali

introduzione

I modelli di progettazione sono soluzioni generali ai problemi che si verificano frequentemente nello sviluppo del software. I seguenti sono modelli di migliori pratiche standardizzate nella strutturazione e progettazione del codice, nonché esempi di contesti comuni in cui questi modelli di progettazione sarebbero appropriati.

I modelli di progettazione strutturale si concentrano sulla composizione di classi e oggetti per creare interfacce e ottenere una maggiore funzionalità.

Examples

Adattatore

Gli adattatori sono usati per convertire l'interfaccia di una data classe, nota come **Adaptee** , in un'altra interfaccia, chiamata **Target** . Le operazioni sul bersaglio sono chiamate da un **cliente** e tali operazioni vengono *adattate* dall'adapter e trasmesse a Adaptee.

In Swift, gli adattatori possono spesso essere formati attraverso l'uso di protocolli. Nell'esempio seguente, un Client in grado di comunicare con il Target viene fornito con la capacità di eseguire funzioni della classe Adaptee tramite l'uso di un adattatore.

```
// The functionality to which a Client has no direct access
class Adaptee {
    func foo() {
        // ...
    }
}

// Target's functionality, to which a Client does have direct access
protocol TargetFunctionality {
    func fooBar() {}
}

// Adapter used to pass the request on the Target to a request on the Adaptee
extension Adaptee: TargetFunctionality {
    func fooBar() {
        foo()
    }
}
```

Esempio di flusso di un adattatore unidirezionale: Client -> Target -> Adapter -> Adaptee

Gli adattatori possono anche essere bidirezionali e questi sono noti come **adattatori a due vie** . Un adattatore bidirezionale può essere utile quando due client diversi devono visualizzare un oggetto in modo diverso.

Facciata

A **Facade** fornisce un'interfaccia unificata e di alto livello per le interfacce del sottosistema. Ciò consente un accesso più semplice e sicuro alle strutture più generali di un sottosistema.

Di seguito è riportato un esempio di facciata utilizzato per impostare e recuperare oggetti in UserDefaults.

```
enum Defaults {
```

```
    static func set(_ object: Any, forKey defaultName: String) {
        let defaults: UserDefaults = UserDefaults.standard
        defaults.set(object, forKey:defaultName)
        defaults.synchronize()
    }

    static func object(forKey key: String) -> AnyObject! {
        let defaults: UserDefaults = UserDefaults.standard
        return defaults.object(forKey: key) as AnyObject!
    }

}
```

L'utilizzo potrebbe essere simile al seguente.

```
Defaults.set("Beyond all recognition.", forKey:"fooBar")
Defaults.object(forKey: "fooBar")
```

Le complessità di accesso all'istanza condivisa e la sincronizzazione di UserDefaults sono nascoste dal client e questa interfaccia è accessibile da qualsiasi punto del programma.

Leggi Modelli di design - Strutturali online:
https://riptutorial.com/it/swift/topic/9497/modelli-di-design---strutturali

Capitolo 44: NSRegularExpression in Swift

Osservazioni

Personaggi speciali

```
   *?+[(){}^$|\./
```

Examples

Estendere la stringa per fare una semplice corrispondenza di modelli

```
extension String {
    func matchesPattern(pattern: String) -> Bool {
        do {
            let regex = try NSRegularExpression(pattern: pattern,
                                                options: NSRegularExpressionOptions(rawValue:
0))
            let range: NSRange = NSMakeRange(0, self.characters.count)
            let matches = regex.matchesInString(self, options: NSMatchingOptions(), range:
range)
            return matches.count > 0
        } catch _ {
            return false
        }
    }
}

// very basic examples - check for specific strings
dump("Pinkman".matchesPattern("(White|Pinkman|Goodman|Schrader|Fring)"))

// using character groups to check for similar-sounding impressionist painters
dump("Monet".matchesPattern("(M[oa]net)"))
dump("Manet".matchesPattern("(M[oa]net)"))
dump("Money".matchesPattern("(M[oa]net)"))     // false

// check surname is in list
dump("Skyler White".matchesPattern("\\w+ (White|Pinkman|Goodman|Schrader|Fring)"))

// check if string looks like a UK stock ticker
dump("VOD.L".matchesPattern("[A-Z]{2,3}\\.L"))
dump("BP.L".matchesPattern("[A-Z]{2,3}\\.L"))

// check entire string is printable ASCII characters
dump("tab\tformatted text".matchesPattern("^[\u{0020}-\u{007e}]*$"))

// Unicode example: check if string contains a playing card suit
dump("♠".matchesPattern("[\u{2660}-\u{2667}]"))
dump("♡".matchesPattern("[\u{2660}-\u{2667}]"))
dump("".matchesPattern("[\u{2660}-\u{2667}]"))    // false

// NOTE: regex needs Unicode-escaped characters
dump("♣".matchesPattern("♣"))           // does NOT work
```

Di seguito è riportato un altro esempio che si basa su quanto sopra per fare qualcosa di utile, che non può essere facilmente realizzato con altri metodi e si presta bene a una soluzione regex.

```
// Pattern validation for a UK postcode.
// This simply checks that the format looks like a valid UK postcode and should not fail on
false positives.
private func isPostcodeValid(postcode: String) -> Bool {
    return postcode.matchesPattern("^[A-Z]{1,2}([0-9][A-Z]|[0-9]{1,2})\\s[0-9][A-Z]{2}")
}

// valid patterns (from
https://en.wikipedia.org/wiki/Postcodes_in_the_United_Kingdom#Validation)
// will return true
dump(isPostcodeValid("EC1A 1BB"))
dump(isPostcodeValid("W1A 0AX"))
dump(isPostcodeValid("M1 1AE"))
dump(isPostcodeValid("B33 8TH"))
dump(isPostcodeValid("CR2 6XH"))
dump(isPostcodeValid("DN55 1PT"))

// some invalid patterns
// will return false
dump(isPostcodeValid("EC12A 1BB"))
dump(isPostcodeValid("CRB1 6XH"))
dump(isPostcodeValid("CR 6XH"))
```

Uso di base

Ci sono diverse considerazioni quando si implementano le espressioni regolari in Swift.

```
let letters = "abcdefg"
let pattern = "[a,b,c]"
let regEx = try NSRegularExpression(pattern: pattern, options: [])
let nsString = letters as NSString
let matches = regEx.matches(in: letters, options: [], range: NSMakeRange(0, nsString.length))
let output = matches.map {nsString.substring(with: $0.range)}
//output = ["a", "b", "c"]
```

Per ottenere una lunghezza di intervallo accurata che supporti tutti i tipi di carattere, la stringa di input deve essere convertita in una NSString.

Per la sicurezza di corrispondenza contro un modello dovrebbe essere racchiuso in un blocco di cattura per gestire il fallimento

```
let numbers = "121314"
let pattern = "1[2,3]"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0,
nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
    output
} catch let error as NSError {
    print("Matching failed")
}
//output = ["12", "13"]
```

La funzionalità di espressione regolare viene spesso inserita in un'estensione o in un supporto per separare le preoccupazioni.

Sostituzione delle sottostringhe

I pattern possono essere usati per sostituire parte di una stringa di input.

L'esempio seguente sostituisce il simbolo del centesimo con il simbolo del dollaro.

```
var money = "¢¥€£$¥€£¢"
let pattern = "¢"
do {
    let regEx = try NSRegularExpression (pattern: pattern, options: [])
    let nsString = money as NSString
    let range = NSMakeRange(0, nsString.length)
    let correct$ = regEx.stringByReplacingMatches(in: money, options: .withTransparentBounds,
range: range, withTemplate: "$")
} catch let error as NSError {
    print("Matching failed")
}
//correct$ = "$¥€£$¥€£$"
```

Personaggi speciali

Per far corrispondere i caratteri speciali, utilizzare Double Backslash `\.` becomes `\\.`

Includere i personaggi che dovrete sfuggire

```
(){}[]/\+*$>.|^?
```

Nell'esempio seguente sono disponibili tre tipi di parentesi di apertura

```
let specials = "(){}[]"
let pattern = "(\\(|\\{|\\[)"
do {
    let regEx = try NSRegularExpression(pattern: pattern, options: [])
    let nsString = specials as NSString
    let matches = regEx.matches(in: specials, options: [], range: NSMakeRange(0,
nsString.length))
    let output = matches.map {nsString.substring(with: $0.range)}
} catch let error as NSError {
    print("Matching failed")
}
//output = ["(", "{", "["]
```

Validazione

Le espressioni regolari possono essere utilizzate per convalidare gli input contando il numero di corrispondenze.

```
var validDate = false

let numbers = "35/12/2016"
let usPattern =  "^(0[1-9]|1[012])[-/.](0[1-9]|[12][0-9]|3[01])[-/.](19|20)\\d\\d$"
let ukPattern = "^(0[1-9]|[12][0-9]|3[01])[-/](0[1-9]|1[012])[-/](19|20)\\d\\d$"
do {
    let regEx = try NSRegularExpression(pattern: ukPattern, options: [])
    let nsString = numbers as NSString
    let matches = regEx.matches(in: numbers, options: [], range: NSMakeRange(0,
nsString.length))

    if matches.count > 0 {
        validDate = true
```

```
    }

    validDate

} catch let error as NSError {
    print("Matching failed")
}
//output = false
```

NSRegularExpression per la convalida della posta

```
func isValidEmail(email: String) -> Bool {

    let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

    let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
    return emailTest.evaluate(with: email)
}
```

oppure potresti usare l'estensione String in questo modo:

```
extension String
{
    func isValidEmail() -> Bool {

        let emailRegEx = "[A-Z0-9a-z._%+-]+@[A-Za-z0-9.-]+\\.[A-Za-z]{2,4}"

        let emailTest = NSPredicate(format:"SELF MATCHES %@", emailRegEx)
        return emailTest.evaluate(with: self)
    }
}
```

Leggi NSRegularExpression in Swift online:
https://riptutorial.com/it/swift/topic/5763/nsregularexpression-in-swift

Capitolo 45: Numeri

Examples

Tipi di numeri e letterali

I tipi numerici incorporati di Swift sono:

- Intestato a livello di parola (dipendente dall'architettura) Int e UInt senza segno .
- Fixed-size interi con segno int8 , Int16 , Int32 , Int64 , e interi senza segno Uint8 , UInt16 , UInt32 , UInt64 .
- Tipi floating-point Float32 / Float , Float64 / Double e Float80 (solo x86).

letterali

Un tipo di letterale numerico è dedotto dal contesto:

```
let x = 42    // x is Int by default
let y = 42.0  // y is Double by default

let z: UInt = 42     // z is UInt
let w: Float = -1    // w is Float
let q = 100 as Int8  // q is Int8
```

I caratteri di sottolineatura (_) possono essere utilizzati per separare le cifre in valori letterali numerici. Gli zeri iniziali vengono ignorati.

I letterali in virgola mobile possono essere specificati usando le parti significante ed esponente (*«significand»* **e** *«exponent»* per decimale; **0x** *«significand»* **p** *«exponent»* per esadecimale).

Sintassi letterale intera

```
let decimal = 10               // ten
let decimal = -1000            // negative one thousand
let decimal = -1_000           // equivalent to -1000
let decimal = 42_42_42         // equivalent to 424242
let decimal = 0755             // equivalent to 755, NOT 493 as in some other languages
let decimal = 0123456789

let hexadecimal = 0x10         // equivalent to 16
let hexadecimal = 0x7FFFFFFF
let hexadecimal = 0xBadFace
let hexadecimal = 0x0123_4567_89ab_cdef

let octal = 0o10               // equivalent to 8
let octal = 0o755              // equivalent to 493
let octal = -0o0123_4567

let binary = -0b101010         // equivalent to -42
let binary = 0b111_101_101     // equivalent to 0o755
let binary = 0b1011_1010_1101  // equivalent to 0xB_A_D
```

Sintassi letterale in virgola mobile

```
let decimal = 0.0
let decimal = -42.0123456789
let decimal = 1_000.234_567_89
```

```
let decimal = 4.567e5               // equivalent to 4.567×10⁵, or 456_700.0
let decimal = -2E-4                 // equivalent to -2×10⁻⁴, or -0.0002
let decimal = 1e+0                  // equivalent to 1×10⁰, or 1.0

let hexadecimal = 0x1p0             // equivalent to 1×2⁰, or 1.0
let hexadecimal = 0x1p-2            // equivalent to 1×2⁻², or 0.25
let hexadecimal = 0xFEEDp+3         // equivalent to 65261×2³, or 522088.0
let hexadecimal = 0x1234.5P4        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x123.45P8        // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x12.345P12       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x1.2345P16       // equivalent to 0x12345, or 74565.0
let hexadecimal = 0x0.12345P20      // equivalent to 0x12345, or 74565.0
```

Converti un tipo numerico in un altro

```
func doSomething1(value: Double) { /* ... */ }
func doSomething2(value: UInt) { /* ... */ }

let x = 42               // x is an Int
doSomething1(Double(x))  // convert x to a Double
doSomething2(UInt(x))    // convert x to a UInt
```

Gli inizializzatori integer generano un **errore di runtime** se il valore trabocca o underflow:

```
Int8(-129.0) // fatal error: floating point value cannot be converted to Int8 because it is
less than Int8.min
Int8(-129)   // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(-128)   // ok
Int8(-2)     // ok
Int8(17)     // ok
Int8(127)    // ok
Int8(128)    // crash: EXC_BAD_INSTRUCTION / SIGILL
Int8(128.0)  // fatal error: floating point value cannot be converted to Int8 because it is
greater than Int8.max
```

Conversione da virgola **mobile a valori** interi **arrotonda i valori verso zero** :

```
Int(-2.2)  // -2
Int(-1.9)  // -1
Int(-0.1)  //  0
Int(1.0)   //  1
Int(1.2)   //  1
Int(1.9)   //  1
Int(2.0)   //  2
```

La conversione da intero a float potrebbe essere una **perdita** :

```
Int(Float(1_000_000_000_000_000_000))  // 999999984306749440
```

Converti numeri in / da stringhe

Utilizzare gli inizializzatori di stringhe per convertire i numeri in stringhe:

```
String(1635999)                          // returns "1635999"
String(1635999, radix: 10)               // returns "1635999"
String(1635999, radix: 2)                // returns "110001111011010011111"
```

```
String(1635999, radix: 16)                    // returns "18f69f"
String(1635999, radix: 16, uppercase: true)   // returns "18F69F"
String(1635999, radix: 17)                    // returns "129gf4"
String(1635999, radix: 36)                    // returns "z2cf"
```

Oppure usa l'interpolazione delle stringhe per casi semplici:

```
let x = 42, y = 9001
"Between \(x) and \(y)"  // equivalent to "Between 42 and 9001"
```

Utilizza gli inizializzatori di tipi numerici per convertire stringhe in numeri:

```
if let num = Int("42") { /* ... */ }                // num is 42
if let num = Int("Z2cF") { /* ... */ }              // returns nil (not a number)
if let num = Int("z2cf", radix: 36) { /* ... */ }   // num is 1635999
if let num = Int("Z2cF", radix: 36) { /* ... */ }   // num is 1635999
if let num = Int8("Z2cF", radix: 36) { /* ... */ }  // returns nil (too large for Int8)
```

Arrotondamento

il giro

Arrotonda il valore al numero intero più vicino con l'arrotondamento di x.5 (ma nota che -x.5 arrotonda per difetto).

```
round(3.000) // 3
round(3.001) // 3
round(3.499) // 3
round(3.500) // 4
round(3.999) // 4

round(-3.000) // -3
round(-3.001) // -3
round(-3.499) // -3
round(-3.500) // -4  *** careful here ***
round(-3.999) // -4
```

ceil

Arrotonda qualsiasi numero con un valore decimale fino al numero intero più grande successivo.

```
ceil(3.000) // 3
ceil(3.001) // 4
ceil(3.999) // 4

ceil(-3.000) // -3
ceil(-3.001) // -3
ceil(-3.999) // -3
```

pavimento

Arrotonda qualsiasi numero con un valore decimale fino al numero intero più piccolo successivo.

```
floor(3.000) // 3
floor(3.001) // 3
floor(3.999) // 3
```

```
floor(-3.000) // -3
floor(-3.001) // -4
floor(-3.999) // -4
```

Int

Converte un Double in un Int , lasciando cadere qualsiasi valore decimale.

```
Int(3.000) // 3
Int(3.001) // 3
Int(3.999) // 3

Int(-3.000) // -3
Int(-3.001) // -3
Int(-3.999) // -3
```

Gli appunti

- round , ceil e floor gestiscono l'architettura a 64 e 32 bit.

Generazione di numeri casuali

arc4random_uniform(someNumber: UInt32) -> UInt32

Questo ti dà numeri casuali nell'intervallo da 0 a someNumber - 1 .

Il valore massimo per UInt32 è UInt32 (ovvero, 2^32 - 1).

Esempi:

- Testa o croce

  ```
  let flip = arc4random_uniform(2) // 0 or 1
  ```

- Rotolo di dadi

  ```
  let roll = arc4random_uniform(6) + 1 // 1...6
  ```

- Random day in October

  ```
  let day = arc4random_uniform(31) + 1 // 1...31
  ```

- Anno casuale negli anni '90

  ```
  let year = 1990 + arc4random_uniform(10)
  ```

Forma generale:

```
let number = min + arc4random_uniform(max - min + 1)
```

dove number , max e min sono UInt32 .

Gli appunti

- C'è un leggero bias di modulo con arc4random quindi è preferito arc4random_uniform .
- Puoi lanciare un valore UInt32 su un Int ma UInt32 attenzione a non andare oltre il raggio

d'azione.

elevamento a potenza

In Swift, possiamo **esponenziare** Double s con il metodo built-in pow() :

```
pow(BASE, EXPONENT)
```

Nel codice sottostante, la base (5) è impostata sulla potenza dell'esponente (2):

```
let number = pow(5.0, 2.0) // Equals 25
```

Leggi Numeri online: https://riptutorial.com/it/swift/topic/454/numeri

Capitolo 46: Oggetti associati

Examples

Proprietà, in un'estensione del protocollo, ottenuta utilizzando l'oggetto associato.

In Swift, le estensioni del protocollo non possono avere proprietà reali.

Tuttavia, in pratica puoi usare la tecnica "oggetto associato". Il risultato è quasi esattamente come una proprietà "reale".

Ecco la tecnica esatta per aggiungere un "oggetto associato" a un'estensione di protocollo:

Fondamentalmente, si utilizza l'obiettivo-c "objc_getAssociatedObject" e _set calls.

Le chiamate di base sono:

```
get {
   return objc_getAssociatedObject(self, & _Handle) as! YourType
   }
set {
   objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
    }
```

Ecco un esempio completo. I due punti critici sono:

1. Nel protocollo, è necessario utilizzare ": class" per evitare il problema di mutazione.
2. Nell'estensione, è necessario utilizzare "where Self: UIViewController" (o qualsiasi altra classe appropriata) per fornire il tipo di conferma.

Quindi, per una proprietà di esempio "p":

```
import Foundation
import UIKit
import ObjectiveC          // don't forget this

var _Handle: UInt8 = 42    // it can be any value

protocol Able: class {
    var click:UIView? { get set }
    var x:CGFloat? { get set }
    // note that you >> do not << declare p here
}

extension Able where Self:UIViewController {

    var p:YourType { // YourType might be, say, an Enum
      get {
          return objc_getAssociatedObject(self, & _Handle) as! YourType
          // HOWEVER, SEE BELOW
          }
      set {
          objc_setAssociatedObject(self, & _Handle, newValue, .OBJC_ASSOCIATION_RETAIN)
          // often, you'll want to run some sort of "setter" here...
          __setter()
          }
    }

    func __setter() { something = p.blah() }
```

```
    func someOtherExtensionFunction() { p.blah() }
    // it's ok to use "p" inside other extension functions,
    // and you can use p anywhere in the conforming class
}
```

In qualsiasi classe conforme, ora hai "aggiunto" la proprietà "p":

Puoi usare "p" proprio come useresti qualsiasi proprietà ordinaria nella classe conforme. Esempio:

```
class Clock:UIViewController, Able {
    var u:Int = 0

    func blah() {
      u = ...
      ... = u
      // use "p" as you would any normal property
      p = ...
      ... = p
    }

    override func viewDidLoad() {
      super.viewDidLoad()
      pm = .none // "p" MUST be "initialized" somewhere in Clock
    }
}
```

Nota. DEVI inizializzare la pseudo proprietà.

Xcode ***non*** **imporrà** l'inizializzazione di "p" nella classe conforme.

È essenziale inizializzare "p", magari in viewDidLoad della classe di conferma.

Vale la pena ricordare che p è in realtà solo una ***proprietà calcolata*** . p è in realtà solo due funzioni, con zucchero sintattico. Non c'è p "variabile" da nessuna parte: il compilatore non "assegna un po 'di memoria per p" in alcun senso. Per questo motivo, non ha senso aspettarsi che Xcode imponga "l'inizializzazione p".

In effetti, per parlare in modo più accurato, è necessario ricordare di "usare p per la prima volta, come se si stesse inizializzando". (Anche in questo caso, molto probabilmente sarebbe nel tuo codice viewDidLoad.)

Per quanto riguarda il getter in quanto tale.

Notare che si **bloccherà** se il getter viene chiamato prima che sia impostato un valore per "p".

Per evitare ciò, considera il codice come:

```
    get {
        let g = objc_getAssociatedObject(self, &_Handle)
        if (g == nil) {
            objc_setAssociatedObject(self, &_Handle, _default initial value_,
.OBJC_ASSOCIATION)
            return _default initial value_
        }
        return objc_getAssociatedObject(self, &_Handle) as! YourType
        }
```

Ripetere. Xcode ***non*** **imporrà** l'inizializzazione di p nella classe conforme. È essenziale inizializzare p, ad esempio in viewDidLoad della classe conforme.

Rendere il codice più semplice ...

Potresti voler utilizzare queste due funzioni globali:

```
func _aoGet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ safeValue: Any!)->Any! {
    let g = objc_getAssociatedObject(ss, handlePointer)
    if (g == nil) {
        objc_setAssociatedObject(ss, handlePointer, safeValue, .OBJC_ASSOCIATION_RETAIN)
        return safeValue
    }
    return objc_getAssociatedObject(ss, handlePointer)
}

func _aoSet(_ ss: Any!, _ handlePointer: UnsafeRawPointer!, _ val: Any!) {
    objc_setAssociatedObject(ss, handlePointer, val, .OBJC_ASSOCIATION_RETAIN)
}
```

Si noti che non fanno nulla, tranne che per salvare la digitazione e rendere il codice più leggibile. (Sono essenzialmente macro o funzioni inline.)

Il tuo codice diventa quindi:

```
protocol PMable: class {
    var click:UILabel? { get set } // ordinary properties here
}

var _pHandle: UInt8 = 321

extension PMable where Self:UIViewController {

    var p:P {
        get {
            return _aoGet(self, &_pHandle, P() ) as! P
        }
        set {
            _aoSet(self, &_pHandle, newValue)
            __pmSetter()
        }
    }

    func __pmSetter() {
        click!.text = String(p)
    }

    func someFunction() {
        p.blah()
    }
}
```

(Nell'esempio su _aoGet, P è initalizable: invece di P () potresti usare "", 0, o qualsiasi valore predefinito.)

Leggi Oggetti associati online: https://riptutorial.com/it/swift/topic/1085/oggetti-associati

Capitolo 47: Operatori avanzati

Examples

Operatori personalizzati

Swift supporta la creazione di operatori personalizzati. I nuovi operatori sono dichiarati a livello globale usando la parola chiave operator .

La struttura dell'operatore è definita da tre parti: operando posizionamento, precedenza e associatività.

1. I modificatori prefix , infix e postfix sono utilizzati per avviare una dichiarazione personalizzata dell'operatore. I modificatori prefix e postfix dichiarano se l'operatore deve essere prima o dopo, rispettivamente, il valore su cui agisce. Tali operatori sono urnari, come 8 e 3++ ** , poiché possono agire solo su un obiettivo. L' infix dichiara un operatore binario che agisce sui due valori in cui è compreso, come 2+3 .

2. Gli operatori con **precedenza** più alta vengono calcolati per primi. La precedenza dell'operatore predefinito è appena superiore a ? ... : (un valore di 100 in Swift 2.x). La precedenza degli operatori Swift standard può essere trovata qui .

3. **Associatività** definisce l'ordine delle operazioni tra operatori della stessa priorità. Gli operatori associativi di sinistra sono calcolati da sinistra a destra (ordine di lettura, come la maggior parte degli operatori), mentre gli operatori associativi di destra calcolano da destra a sinistra.

3.0

A partire da Swift 3.0, si definiscono la precedenza e l'associatività in un **gruppo di precedenza** anziché l'operatore stesso, in modo che più operatori possano condividere facilmente la stessa precedenza senza fare riferimento ai numeri criptici. L'elenco dei gruppi di precedenza standard è mostrato di seguito .

Gli operatori restituiscono valori basati sul codice di calcolo. Questo codice agisce come una funzione normale, con parametri che specificano il tipo di input e la parola chiave return specifica il valore calcolato restituito dall'operatore.

Ecco la definizione di un semplice operatore esponenziale, poiché lo standard Swift non ha un operatore esponenziale.

```
import Foundation

infix operator ** { associativity left precedence 170 }

func ** (num: Double, power: Double) -> Double{
    return pow(num, power)
}
```

L' infix dice che l'operatore ** lavora tra due valori, come 9**2 . Poiché la funzione ha lasciato l'associatività, 3**3**2 viene calcolata come (3**3)**2 . La precedenza di 170 è superiore a tutte le operazioni Swift standard, il che significa che 3+2**4 calcola a 19 , nonostante l'associatività a sinistra di ** .

3.0

```
import Foundation

infix operator **: BitwiseShiftPrecedence

func ** (num: Double, power: Double) -> Double {
```

```
    return pow(num, power)
}
```

Invece di specificare esplicitamente la precedenza e l'associatività, su Swift 3.0 potremmo usare il gruppo di precedenza predefinito BitwiseShiftPrecedence che fornisce i valori corretti (come << , >>).

**: l'incremento e il decremento sono deprecati e verranno rimossi in Swift 3.

Sovraccarico + per dizionari

Poiché attualmente non esiste un modo semplice per combinare i dizionari in Swift, può essere utile sovraccaricare gli operatori + e += per aggiungere questa funzionalità utilizzando i generici .

```
// Combines two dictionaries together. If both dictionaries contain
// the same key, the value of the right hand side dictionary is used.
func +<K, V>(lhs: [K : V], rhs: [K : V]) -> [K : V] {
    var combined = lhs
    for (key, value) in rhs {
        combined[key] = value
    }
    return combined
}

// The mutable variant of the + overload, allowing a dictionary
// to be appended to 'in-place'.
func +=<K, V>(inout lhs: [K : V], rhs: [K : V]) {
    for (key, value) in rhs {
        lhs[key] = value
    }
}
```

3.0

A partire da Swift 3, inout deve essere posizionato prima del tipo di argomento.

```
func +=<K, V>(lhs: inout [K : V], rhs: [K : V]) { ... }
```

Esempio di utilizzo:

```
let firstDict = ["hello" : "world"]
let secondDict = ["world" : "hello"]
var thirdDict = firstDict + secondDict // ["hello": "world", "world": "hello"]

thirdDict += ["hello":"bar", "baz":"qux"] // ["hello": "bar", "baz": "qux", "world": "hello"]
```

Operatori commutativi

Aggiungiamo un operatore personalizzato per moltiplicare un CGSize

```
func *(lhs: CGFloat, rhs: CGSize) -> CGSize{
    let height = lhs*rhs.height
    let width = lhs*rhs.width
    return CGSize(width: width, height: height)
}
```

Ora questo funziona

```
let sizeA = CGSize(height:100, width:200)
let sizeB = 1.1 * sizeA         //=> (height: 110, width: 220)
```

Ma se proviamo a fare l'operazione al contrario, otteniamo un errore

```
let sizeC = sizeB * 20          // ERROR
```

Ma è abbastanza semplice aggiungere:

```
func *(lhs: CGSize, rhs: CGFloat) -> CGSize{
    return rhs*lhs
}
```

Ora l'operatore è commutativo.

```
let sizeA = CGSize(height:100, width:200)
let sizeB = sizeA * 1.1              //=> (height: 110, width: 220)
```

Operatori bit a bit

Swift Operatori bit a bit consentono di eseguire operazioni sulla forma binaria di numeri. È possibile specificare un valore letterale binario prefissando il numero con 0b , quindi per esempio 0b110 equivale al numero binario 110 (il numero decimale 6). Ogni 1 o 0 è un po 'nel numero.

Bitwise NOT ~ :

```
var number: UInt8 = 0b01101100
let newNumber = ~number
// newNumber is equal to 0b01101100
```

Qui, ogni bit viene cambiato nel suo opposto. Dichiarare il numero come esplicitamente UInt8 assicura che il numero sia positivo (in modo che non ci sia da trattare con i negativi nell'esempio) e che sia solo 8 bit. Se 0b01101100 fosse un UInt più grande, ci sarebbero degli 0 iniziali che verrebbero convertiti in 1 e diventerebbero significativi in caso di inversione:

```
var number: UInt16 = 0b01101100
// number equals 0b0000000001101100
// the 0s are not significant
let newNumber = ~number
// newNumber equals 0b1111111110010011
// the 1s are now significant
```

- 0 -> 1
- 1 -> 0

AND bit a bit & :

```
var number = 0b0110
let newNumber = number & 0b1010
// newNumber is equal to 0b0010
```

Qui, un dato bit sarà 1 se e solo se i numeri binari su entrambi i lati dell'operatore & contenessero un 1 in quella posizione di bit.

- 0 e 0 -> 0
- 0 & 1 -> 0
- 1 & 1 -> 1

Bitwise OR | :

```
var number = 0b0110
let newNumber = number | 0b1000
// newNumber is equal to 0b1110
```

Qui, un dato bit sarà 1 se e solo se il numero binario su almeno un lato del | l'operatore conteneva un 1 in quella posizione di bit.

- 0 | 0 -> 0
- 0 | 1 -> 1
- 1 | 1 -> 1

XOR bit a bit (OR esclusivo) ^ :

```
var number = 0b0110
let newNumber = number ^ 0b1010
// newNumber is equal to 0b1100
```

Qui, un dato bit sarà 1 se e solo se i bit in quella posizione dei due operandi sono diversi.

- 0 ^ 0 -> 0
- 0 ^ 1 -> 1
- 1 ^ 1 -> 0

Per tutte le operazioni binarie, l'ordine degli operandi non fa differenza sul risultato.

Operatori di overflow

L'overflow si riferisce a ciò che accade quando un'operazione determina un numero che può essere maggiore o minore della quantità di bit designata per quel numero.

A causa del modo in cui funziona l'aritmetica binaria, dopo che un numero diventa troppo grande per i suoi bit, il numero trabocca al numero più piccolo possibile (per la dimensione del bit) e quindi continua a contare da lì. Allo stesso modo, quando un numero diventa troppo piccolo, esso porta fino al numero più grande possibile (per la sua dimensione di bit) e continua il conto alla rovescia da lì.

Poiché questo comportamento non è spesso desiderato e può portare a seri problemi di sicurezza, gli operatori aritmetici Swift + , - e * generano errori quando un'operazione provoca un overflow o un underflow. Per consentire esplicitamente overflow e underflow, utilizza invece &+ , &- , e &* .

```
var almostTooLarge = Int.max
almostTooLarge + 1 // not allowed
almostTooLarge &+ 1 // allowed, but result will be the value of Int.min
```

Precedenza degli operatori Swift standard

Gli operatori che vincolano più strettamente (precedenza più alta) sono elencati per primi.

operatori	Gruppo di precedenza (≥3,0)	Precedenza	Associatività
.		∞	sinistra
? , ! , ++ , -- , [] , () , {}	(Postfix)		
! , ~ , + , - , ++ , --	(prefisso)		

operatori	Gruppo di precedenza (≥3,0)	Precedenza	Associatività
~> (rapido ≤2.3)		255	sinistra
<< , >>	BitwiseShiftPrecedence	160	nessuna
* , / , % , & , &*	MultiplicationPrecedence	150	sinistra
+ , - , | , ^ , &+ , &-	AdditionPrecedence	140	sinistra
... , ..<	RangeFormationPrecedence	135	nessuna
is , as , as? , as!	CastingPrecedence	132	sinistra
??	NilCoalescingPrecedence	131	destra
< , <= , > , >= , == != , === !== , ~=	ComparisonPrecedence	130	nessuna
&&	LogicalConjunctionPrecedence	120	sinistra
||	LogicalDisjunctionPrecedence	110	sinistra
	DefaultPrecedence *		nessuna
? ... :	TernaryPrecedence	100	destra
= , += , -= , *= , /= , %= , <<= , >>= , &= , |= , ^=	AssignmentPrecedence	90	giusto, incarico
->	FunctionArrowPrecedence		destra

3.0

- Il gruppo di precedenza DefaultPrecedence è superiore a TernaryPrecedence , ma non è ordinato con il resto degli operatori. Oltre a questo gruppo, il resto delle precedenze sono lineari.

- Questa tabella può essere trovata anche sul riferimento API di Apple
- La definizione attuale dei gruppi di precedenza può essere trovata nel codice sorgente su GitHub

Leggi Operatori avanzati online: https://riptutorial.com/it/swift/topic/1048/operatori-avanzati

Capitolo 48: OPTIONSET

Examples

Protocollo OptionSet

OptionSetType è un protocollo progettato per rappresentare i tipi di maschera di bit in cui i singoli bit rappresentano i membri dell'insieme. Un insieme di logiche e / o funzioni impone la sintassi corretta:

```
struct Features : OptionSet {
  let rawValue : Int
  static let none = Features(rawValue: 0)
  static let feature0 = Features(rawValue: 1 << 0)
  static let feature1 = Features(rawValue: 1 << 1)
  static let feature2 = Features(rawValue: 1 << 2)
  static let feature3 = Features(rawValue: 1 << 3)
  static let feature4 = Features(rawValue: 1 << 4)
  static let feature5 = Features(rawValue: 1 << 5)
  static let all: Features = [feature0, feature1, feature2, feature3, feature4, feature5]
}

Features.feature1.rawValue //2
Features.all.rawValue //63

var options: Features = [.feature1, .feature2, .feature3]

options.contains(.feature1) //true
options.contains(.feature4) //false

options.insert(.feature4)
options.contains(.feature4) //true


var otherOptions : Features = [.feature1, .feature5]

options.contains(.feature5) //false

options.formUnion(otherOptions)
options.contains(.feature5) //true

options.remove(.feature5)
options.contains(.feature5) //false
```

Leggi OPTIONSET online: https://riptutorial.com/it/swift/topic/1242/optionset

Capitolo 49: Opzionali

introduzione

"Un valore opzionale contiene un valore o contiene nil per indicare che manca un valore"

Estratto da: Apple Inc. "The Swift Programming Language (Swift 3.1 Edition)." IBooks. https://itun.es/us/k5SW7.l

I casi d'uso facoltativi di base includono: per una costante (let), l'uso di un facoltativo all'interno di un ciclo (if-let), lo srotolamento sicuro di un valore facoltativo all'interno di un metodo (guard-let) e come parte di cicli di commutazione (caso-let), per impostazione predefinita su un valore se nil, utilizzando l'operatore di coalesce (??)

Sintassi

- var optionalName: optionalType? // dichiara un tipo facoltativo, il valore predefinito è nullo
- var optionalName: optionalType? = valore // dichiara un opzionale con un valore
- var optionalName: optionalType! // dichiara un facoltativo facoltativo da scartare
- opzionale! // forza scartare un opzionale

Osservazioni

Per ulteriori informazioni sugli optionals, vedere The Swift Programming Language .

Examples

Tipi di Optionals

Gli optionals sono un tipo enum generico che funge da wrapper. Questo wrapper consente a una variabile di avere uno o due stati: il valore del tipo definito dall'utente o nil , che rappresenta l'assenza di un valore.

Questa capacità è particolarmente importante in Swift perché uno degli obiettivi di design dichiarati della lingua è quello di funzionare bene con i framework di Apple. Molti (molti) framework di Apple utilizzano nil causa della sua facilità d'uso e significato per i pattern di programmazione e il design delle API all'interno di Objective-C.

In Swift, affinché una variabile abbia un valore nil , deve essere facoltativa. Gli optionals possono essere creati aggiungendo a ! o un ? al tipo di variabile. Ad esempio, per rendere un Int opzionale, è possibile utilizzare

```
var numberOne: Int! = nil
var numberTwo: Int? = nil
```

? le opzioni devono essere esplicitamente scartate e dovrebbero essere utilizzate se non si è certi che la variabile abbia o meno un valore quando si accede ad essa. Ad esempio, quando si trasforma una stringa in un Int , il risultato è un Int? opzionale Int? , perché nil verrà restituito se la stringa non è un numero valido

```
let str1 = "42"
let num1: Int? = Int(str1) // 42

let str2 = "Hello, World!"
let num2: Int? = Int(str2) // nil
```

! le opzioni vengono scartate automaticamente e dovrebbero essere utilizzate *solo* quando si è *certi* che la variabile avrà un valore quando ci si accede. Ad esempio, un UIButton! globale

UIButton! variabile inizializzata in viewDidLoad()

```
//myButton will not be accessed until viewDidLoad is called,
//so a ! optional can be used here
var myButton: UIButton!

override func viewDidLoad(){
    self.myButton = UIButton(frame: self.view.frame)
    self.myButton.backgroundColor = UIColor.redColor()
    self.view.addSubview(self.myButton)
}
```

Scartare un facoltativo

Per accedere al valore di un Opzionale, deve essere scartato.

È possibile *scartare condizionatamente* un Opzionale usando l'associazione opzionale e *forzare scartare* un Opzionale usando il ! operatore.

Lo srotolamento condizionato richiede in modo efficace "Questa variabile ha un valore?" mentre force unwrapping dice "Questa variabile ha un valore!".

Se imposti di scartare una variabile nil , il tuo programma genererà un risultato *inaspettato nullo mentre scarterai* un'eccezione *opzionale* e si bloccherà, quindi devi valutare attentamente se usi ! è appropriato.

```
var text: String? = nil
var unwrapped: String = text! //crashes with "unexpectedly found nil while unwrapping an
Optional value"
```

Per lo srotolamento sicuro, è possibile utilizzare un'istruzione if-let , che non genererà un'eccezione o un arresto anomalo se il valore spostato è nil :

```
var number: Int?
if let unwrappedNumber = number {       // Has `number` been assigned a value?
    print("number: \(unwrappedNumber)") // Will not enter this line
} else {
    print("number was not assigned a value")
}
```

O una dichiarazione di guardia :

```
var number: Int?
guard let unwrappedNumber = number else {
    return
}
print("number: \(unwrappedNumber)")
```

Si noti che l'ambito della variabile unwrappedNumber trova all'interno unwrappedNumber if-let e all'esterno del blocco di guard .

Puoi concatenare la scartocciatura di molti optionals, questo è utile soprattutto nel caso in cui il tuo codice richieda più di una variabile per funzionare correttamente:

```
var firstName:String?
var lastName:String?

if let fn = firstName, let ln = lastName {
    print("\(fn) + \(ln)")//pay attention that the condition will be true only if both
```

```
optionals are not nil.
}
```

Si noti che tutte le variabili devono essere scartate per passare correttamente il test, altrimenti non si avrebbe modo di determinare quali variabili sono state scartate e quali no.

Puoi concatenare le dichiarazioni condizionali usando i tuoi optionals immediatamente dopo averli scartati. Questo significa che non ci sono nested if - else statements!

```
var firstName:String? = "Bob"
var myBool:Bool? = false

if let fn = firstName, fn == "Bob", let bool = myBool, !bool {
    print("firstName is bob and myBool was false!")
}
```

Nil Coalescing Operator

Puoi utilizzare l' *operatore* a *coalescenza nil* per scartare un valore se non è zero, altrimenti fornire un valore diverso:

```
func fallbackIfNil(str: String?) -> String {
    return str ?? "Fallback String"
}
print(fallbackIfNil("Hi")) // Prints "Hi"
print(fallbackIfNil(nil)) // Prints "Fallback String"
```

Questo operatore è in grado di cortocircuitare , nel senso che se l'operando di sinistra non è zero, l'operando di destra non verrà valutato:

```
func someExpensiveComputation() -> String { ... }

var foo : String? = "a string"
let str = foo ?? someExpensiveComputation()
```

In questo esempio, poichè foo è non-nil, someExpensiveComputation() non saranno chiamati.

Puoi anche concatenare più istruzioni coalescenti nil:

```
var foo : String?
var bar : String?

let baz = foo ?? bar ?? "fallback string"
```

In questo esempio baz verrà assegnato al valore unwrapped di foo se non è zero, altrimenti verrà assegnato il valore unwrapped della bar se non è zero, altrimenti verrà assegnato il valore di fallback.

Concatenamento opzionale

È possibile utilizzare il concatenamento opzionale per chiamare un metodo , accedere a una proprietà o pedice facoltativo. Questo viene fatto posizionando un ? tra la variabile opzionale data e il membro dato (metodo, proprietà o pedice).

```
struct Foo {
    func doSomething() {
        print("Hello World!")
    }
```

```
}

var foo : Foo? = Foo()

foo?.doSomething() // prints "Hello World!" as foo is non-nil
```

Se foo contiene un valore, doSomething() verrà chiamato su di esso. Se foo è nil , allora non accadrà nulla di male: il codice fallirà semplicemente in silenzio e continuerà ad essere eseguito.

```
var foo : Foo? = nil

foo?.doSomething() // will not be called as foo is nil
```

(Questo è un comportamento simile all'invio di messaggi a `nil` in Objective-C)

La ragione per cui il concatenamento facoltativo è chiamato così è perché "l'opzionalità" sarà propagata attraverso i membri che chiami / acceda. Ciò significa che i valori di ritorno di tutti i membri utilizzati con il concatenamento facoltativo saranno facoltativi, indipendentemente dal fatto che siano stati digitati come facoltativi o meno.

```
struct Foo {
    var bar : Int
    func doSomething() { ... }
}

let foo : Foo? = Foo(bar: 5)
print(foo?.bar) // Optional(5)
```

Qui foo?.bar sta restituendo un Int? anche se la bar non è opzionale, poichè foo stesso è facoltativo.

Mentre l'opzionalità viene propagata, i metodi che restituiscono Void restituiranno Void? quando viene chiamato con concatenamento opzionale. Questo può essere utile per determinare se il metodo è stato chiamato o meno (e quindi se l'opzione ha un valore).

```
let foo : Foo? = Foo()

if foo?.doSomething() != nil {
    print("foo is non-nil, and doSomething() was called")
} else {
    print("foo is nil, therefore doSomething() wasn't called")
}
```

Qui stiamo confrontando il Void? restituire il valore con nil per determinare se il metodo è stato chiamato (e quindi se foo è non-zero).

Panoramica - Perché Optionals?

Spesso durante la programmazione è necessario fare una distinzione tra una variabile che ha un valore e una che non lo fa. Per i tipi di riferimento, come C Pointers, è possibile utilizzare un valore speciale come null per indicare che la variabile non ha alcun valore. Per tipi intrinseci, come un numero intero, è più difficile. È possibile utilizzare un valore nominale, ad esempio -1, ma ciò dipende dall'interpretazione del valore. Elimina anche quel valore "speciale" dal normale utilizzo.

Per risolvere questo problema, Swift consente a qualsiasi variabile di essere dichiarata come facoltativa. Questo è indicato dall'uso di un? o ! dopo il tipo (Vedi Tipi di optionals)

Per esempio,

```
var possiblyInt: Int?
```

dichiara una variabile che può o non può contenere un valore intero.

Il valore speciale nil indica che al momento non è assegnato alcun valore a questa variabile.

```
possiblyInt = 5      // PossiblyInt is now 5
possiblyInt = nil    // PossiblyInt is now unassigned
```

nil può anche essere usato per testare un valore assegnato:

```
if possiblyInt != nil {
    print("possiblyInt has the value \(possiblyInt!)")
}
```

Nota l'uso di ! *nell'istruzione* print per *scartare* il valore opzionale.

Come esempio di un uso comune di optionals, si consideri una funzione che restituisce un intero da una stringa contenente cifre; È possibile che la stringa contenga caratteri non numerici o addirittura vuota.

In che modo una funzione che restituisce un semplice Int indica un errore? Non può farlo restituendo un valore specifico in quanto ciò impedirebbe che il valore venga analizzato dalla stringa.

```
var someInt
someInt = parseInt("not an integer") // How would this function indicate failure?
```

In Swift, tuttavia, quella funzione può semplicemente restituire un Int *opzionale* . Quindi il fallimento è indicato dal valore di ritorno di nil .

```
var someInt?
someInt = parseInt("not an integer")  // This function returns nil if parsing fails
if someInt == nil {
    print("That isn't a valid integer")
}
```

Leggi Opzionali online: https://riptutorial.com/it/swift/topic/247/opzionali

Capitolo 50: Prestazione

Examples

Prestazioni di allocazione

In Swift, la gestione della memoria viene eseguita automaticamente utilizzando il conteggio dei riferimenti automatico. (Vedi Gestione della memoria) L'allocazione è il processo di riservare un punto in memoria per un oggetto, e in Swift comprendere le prestazioni di tale richiede una certa comprensione **dell'heap** e dello **stack** . L'heap è una locazione di memoria in cui vengono collocati la maggior parte degli oggetti e si può pensare a un magazzino. La pila, d'altra parte, è una pila di chiamate di funzioni che hanno portato all'esecuzione corrente. (Quindi, una traccia dello stack è una sorta di stampa delle funzioni sullo stack delle chiamate).

Assegnare e deallocare dallo stack è un'operazione molto efficiente, tuttavia l'allocazione dell'heap di confronto è costosa. Quando progetti per prestazioni, dovresti tenerlo a mente.

Classi:

```
class MyClass {

    let myProperty: String

}
```

Le classi in Swift sono tipi di riferimento e quindi accadono diverse cose. Innanzitutto, l'oggetto effettivo verrà assegnato all'heap. Quindi, qualsiasi riferimento a quell'oggetto deve essere aggiunto allo stack. Ciò rende le classi un oggetto più costoso per l'allocazione.

le strutture:

```
struct MyStruct {

    let myProperty: Int

}
```

Poiché le strutture sono tipi di valore e quindi copiate quando vengono passate, vengono allocate nello stack. Questo rende le strutture più efficienti delle classi, tuttavia, se hai bisogno di una nozione di identità e / o di semantica di riferimento, una struttura non può fornirti quelle cose.

Avviso sulle strutture con stringhe e proprietà che sono classi

Mentre le struct sono in genere cheeper rispetto alle classi, dovresti fare attenzione alle strutture con proprietà che sono classi:

```
struct MyStruct {

    let myProperty: MyClass

}
```

Qui, a causa del conteggio dei riferimenti e di altri fattori, la performance è ora più simile a una classe. Inoltre, se più di una proprietà nella struttura è una classe, l'impatto sulle prestazioni potrebbe essere anche più negativo rispetto a se la struttura fosse invece una classe.

Inoltre, mentre le stringhe sono strutture, memorizzano internamente i loro caratteri nello heap, quindi sono più costosi della maggior parte delle strutture.

Leggi Prestazione online: https://riptutorial.com/it/swift/topic/4067/prestazione

Capitolo 51: Programmazione funzionale in Swift

Examples

Estrarre un elenco di nomi da un elenco di Person (s)

Dato una struct Person

```
struct Person {
    let name: String
    let birthYear: Int?
}
```

e una schiera di Person(s)

```
let persons = [
    Person(name: "Walter White", birthYear: 1959),
    Person(name: "Jesse Pinkman", birthYear: 1984),
    Person(name: "Skyler White", birthYear: 1970),
    Person(name: "Saul Goodman", birthYear: nil)
]
```

possiamo recuperare un array di String contenente la proprietà name di ogni Person.

```
let names = persons.map { $0.name }
// ["Walter White", "Jesse Pinkman", "Skyler White", "Saul Goodman"]
```

attraversamento

```
let numbers = [3, 1, 4, 1, 5]
// non-functional
for (index, element) in numbers.enumerate() {
    print(index, element)
}

// functional
numbers.enumerate().map { (index, element) in
    print((index, element))
}
```

Proiezione

L'applicazione di una funzione a una raccolta / flusso e la creazione di una nuova raccolta / flusso è detta **proiezione** .

```
/// Projection
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
```

```
        "id": 654356453,
        "title": "Bad Boys",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [4.0],
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": [5.0],
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]


var videoAndTitlePairs = [[String: AnyObject]]()
newReleases.map { e in
    videoAndTitlePairs.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videoAndTitlePairs)
```

filtraggio

Crea un flusso selezionando gli elementi da un flusso che passa una determinata condizione è chiamato **filtro**

```
var newReleases = [
    [
        "id": 70111470,
        "title": "Die Hard",
        "boxart": "http://cdn-0.nflximg.com/images/2891/DieHard.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 654356453,
        "title": "Bad Boys",
        "boxart": "http://cdn-0.nflximg.com/images/2891/BadBoys.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ],
    [
        "id": 65432445,
        "title": "The Chamber",
        "boxart": "http://cdn-0.nflximg.com/images/2891/TheChamber.jpg",
```

```
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 4.0,
        "bookmark": []
    ],
    [
        "id": 675465,
        "title": "Fracture",
        "boxart": "http://cdn-0.nflximg.com/images/2891/Fracture.jpg",
        "uri": "http://api.netflix.com/catalog/titles/movies/70111470",
        "rating": 5.0,
        "bookmark": [[ "id": 432534, "time": 65876586 ]]
    ]
]

var videos1 = [[String: AnyObject]]()
/**
 *  Filtering using map
 */
newReleases.map { e in
    if e["rating"] as! Float == 5.0 {
        videos1.append(["id": e["id"] as! Int, "title": e["title"] as! String])
    }
}

print(videos1)

var videos2 = [[String: AnyObject]]()
/**
 *  Filtering using filter and chaining
 */
newReleases
    .filter{ e in
        e["rating"] as! Float == 5.0
    }
    .map { e in
    videos2.append(["id": e["id"] as! Int, "title": e["title"] as! String])
}

print(videos2)
```

Usare il filtro con le strutture

Spesso è possibile filtrare strutture e altri tipi di dati complessi. La ricerca di una serie di strutture per voci che contengono un particolare valore è un'attività molto comune e facilmente ottenibile in Swift utilizzando le funzionalità di programmazione funzionale. Inoltre, il codice è estremamente succinto.

```
struct Painter {
    enum Type { case Impressionist, Expressionist, Surrealist, Abstract, Pop }
    var firstName: String
    var lastName: String
    var type: Type
}

let painters = [
    Painter(firstName: "Claude", lastName: "Monet", type: .Impressionist),
    Painter(firstName: "Edgar", lastName: "Degas", type: .Impressionist),
    Painter(firstName: "Egon", lastName: "Schiele", type: .Expressionist),
    Painter(firstName: "George", lastName: "Grosz", type: .Expressionist),
    Painter(firstName: "Mark", lastName: "Rothko", type: .Abstract),
```

```
    Painter(firstName: "Jackson", lastName: "Pollock", type: .Abstract),
    Painter(firstName: "Pablo", lastName: "Picasso", type: .Surrealist),
    Painter(firstName: "Andy", lastName: "Warhol", type: .Pop)
]

// list the expressionists
dump(painters.filter({$0.type == .Expressionist}))

// count the expressionists
dump(painters.filter({$0.type == .Expressionist}).count)
// prints "2"

// combine filter and map for more complex operations, for example listing all
// non-impressionist and non-expressionists by surname
dump(painters.filter({$0.type != .Impressionist && $0.type != .Expressionist})
    .map({$0.lastName}).joinWithSeparator(", "))
// prints "Rothko, Pollock, Picasso, Warhol"
```

Leggi Programmazione funzionale in Swift online:
https://riptutorial.com/it/swift/topic/2948/programmazione-funzionale-in-swift

Capitolo 52: protocolli

introduzione

I protocolli sono un modo per specificare come utilizzare un oggetto. Descrivono un insieme di proprietà e metodi che una classe, struttura o enum dovrebbero fornire, sebbene i protocolli non pongano restrizioni all'implementazione.

Osservazioni

Un protocollo Swift è una raccolta di requisiti che devono essere implementati dai tipi conformi. Il protocollo può quindi essere utilizzato nella maggior parte dei casi in cui è previsto un tipo, ad esempio matrici e requisiti generici.

I membri del protocollo condividono sempre lo stesso qualificatore di accesso dell'intero protocollo e non possono essere specificati separatamente. Sebbene un protocollo possa limitare l'accesso con requisiti getter o setter, come negli esempi sopra.

Per ulteriori informazioni sui protocolli, vedere The Swift Programming Language .

I protocolli Objective-C sono simili ai protocolli Swift.

I protocolli sono anche paragonabili alle interfacce Java .

Examples

Nozioni di base sul protocollo

Informazioni sui protocolli

Un protocollo specifica inizializzatori, proprietà, funzioni, pedici e tipi associati richiesti di un tipo di oggetto Swift (classe, struct o enum) conforme al protocollo. In alcune lingue idee simili per le specifiche dei requisiti degli oggetti successivi sono conosciute come "interfacce".

Un Protocollo dichiarato e definito è un Tipo, in sé e per sé, con una firma dei suoi requisiti dichiarati, in qualche modo simile al modo in cui le Funzioni Swift sono un Tipo basato sulla loro firma di parametri e ritorni.

Le specifiche del protocollo Swift possono essere facoltative, richieste esplicitamente e / o fornite implementazioni predefinite tramite una funzione nota come Extension Protocol. Un tipo di oggetto Swift (classe, struct o enum) che desidera conformarsi a un protocollo che è arricchito con estensioni per tutti i suoi requisiti specificati, deve solo indicare il suo desiderio di conformarsi per essere in piena conformità. La funzionalità di implementazione predefinita delle estensioni del protocollo può essere sufficiente per soddisfare tutti gli obblighi di conformità a un protocollo.

I protocolli possono essere ereditati da altri protocolli. Questo, in congiunzione con Protocol Extensions, significa che i protocolli possono e devono essere considerati come una caratteristica significativa di Swift.

I protocolli e le estensioni sono importanti per realizzare gli obiettivi e gli approcci più ampi di Swift alla flessibilità della progettazione del programma e ai processi di sviluppo. Lo scopo principale dichiarato della capacità di protocollo e estensione di Swift è l'agevolazione della progettazione compositiva nell'architettura e nello sviluppo del programma. Si parla di programmazione orientata al protocollo. I vecchi croccanti considerano questo superiore l'attenzione al design OOP.

I protocolli definiscono interfacce che possono essere implementate da qualsiasi struct , classe o enum :

```
protocol MyProtocol {
    init(value: Int)                        // required initializer
    func doSomething() -> Bool              // instance method
    var message: String { get }             // instance read-only property
    var value: Int { get set }              // read-write instance property
    subscript(index: Int) -> Int { get }    // instance subscript
    static func instructions() -> String    // static method
    static var max: Int { get }             // static read-only property
    static var total: Int { get set }       // read-write static property
}
```

Le proprietà definite nei protocolli devono essere annotate come { get } o { get set } . { get } significa che la proprietà deve essere disponibile e quindi può essere implementata come *qualsiasi* tipo di proprietà. { get set } significa che la proprietà deve essere impostabile e ricevibile.

Una struttura, una classe o una enum possono essere **conformi a** un protocollo:

```
struct MyStruct : MyProtocol {
    // Implement the protocol's requirements here
}
class MyClass : MyProtocol {
    // Implement the protocol's requirements here
}
enum MyEnum : MyProtocol {
    case caseA, caseB, caseC
    // Implement the protocol's requirements here
}
```

Un protocollo può anche definire **un'implementazione predefinita** per uno qualsiasi dei suoi requisiti attraverso un'estensione :

```
extension MyProtocol {

    // default implementation of doSomething() -> Bool
    // conforming types will use this implementation if they don't define their own
    func doSomething() -> Bool {
        print("do something!")
        return true
    }
}
```

Un protocollo può essere **usato come un tipo** , a condizione che non abbia i requisiti del tipo associatedtype :

```
func doStuff(object: MyProtocol) {
    // All of MyProtocol's requirements are available on the object
    print(object.message)
    print(object.doSomething())
}

let items : [MyProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

Puoi anche definire un tipo astratto conforme a **più** protocolli:

3.0

Con Swift 3 o superiore, questo viene fatto separando la lista dei protocolli con una e commerciale (&):

```
func doStuff(object: MyProtocol & AnotherProtocol) {
    // ...
}

let items : [MyProtocol & AnotherProtocol] = [MyStruct(), MyClass(), MyEnum.caseA]
```

3.0

Le versioni precedenti hanno il protocol<...> sintassi protocol<...> dove i protocolli sono un elenco separato da virgole tra parentesi angolari <> .

```
protocol AnotherProtocol {
    func doSomethingElse()
}

func doStuff(object: protocol<MyProtocol, AnotherProtocol>) {

    // All of MyProtocol & AnotherProtocol's requirements are available on the object
    print(object.message)
    object.doSomethingElse()
}

// MyStruct, MyClass & MyEnum must now conform to both MyProtocol & AnotherProtocol
let items : [protocol<MyProtocol, AnotherProtocol>] = [MyStruct(), MyClass(), MyEnum.caseA]
```

I tipi esistenti possono essere **estesi** per conformarsi a un protocollo:

```
extension String : MyProtocol {
    // Implement any requirements which String doesn't already satisfy
}
```

Requisiti del tipo associato

I protocolli possono definire i **requisiti di tipo associati** utilizzando la parola chiave associatedtype :

```
protocol Container {
    associatedtype Element
    var count: Int { get }
    subscript(index: Int) -> Element { get set }
}
```

I protocolli con i requisiti di tipo associati **possono essere utilizzati solo come vincoli generici** :

```
// These are NOT allowed, because Container has associated type requirements:
func displayValues(container: Container) { ... }
class MyClass { let container: Container }
// > error: protocol 'Container' can only be used as a generic constraint
// > because it has Self or associated type requirements

// These are allowed:
func displayValues<T: Container>(container: T) { ... }
class MyClass<T: Container> { let container: T }
```

Un tipo conforme al protocollo può soddisfare implicitamente un requisito di tipo associatedtype , fornendo un determinato tipo in cui il protocollo si aspetta che venga visualizzato il tipo associatedtype :

```
struct ContainerOfOne<T>: Container {
    let count = 1          // satisfy the count requirement
    var value: T

    // satisfy the subscript associatedtype requirement implicitly,
    // by defining the subscript assignment/return type as T
    // therefore Swift will infer that T == Element
    subscript(index: Int) -> T {
        get {
            precondition(index == 0)
            return value
        }
        set {
            precondition(index == 0)
            value = newValue
        }
    }
}

let container = ContainerOfOne(value: "Hello")
```

(Si noti che per aggiungere chiarezza a questo esempio, il tipo di segnaposto generico è denominato `T` - un nome più adatto sarebbe `Element` , che ombreggia l' `associatedtype Element` del protocollo. Il compilatore dedurrà comunque che l' `Element` segnaposto generico viene utilizzato per soddisfare il tipo `associatedtype Element` Requisito `associatedtype Element` .)

Un tipo associatedtype può anche essere soddisfatto esplicitamente attraverso l'uso di un typealias :

```
struct ContainerOfOne<T>: Container {

    typealias Element = T
    subscript(index: Int) -> Element { ... }

    // ...
}
```

Lo stesso vale per le estensioni:

```
// Expose an 8-bit integer as a collection of boolean values (one for each bit).
extension UInt8: Container {

    // as noted above, this typealias can be inferred
    typealias Element = Bool

    var count: Int { return 8 }
    subscript(index: Int) -> Bool {
        get {
            precondition(0 <= index && index < 8)
            return self & 1 << UInt8(index) != 0
        }
        set {
            precondition(0 <= index && index < 8)
            if newValue {
                self |= 1 << UInt8(index)
            } else {
                self &= ~(1 << UInt8(index))
            }
        }
    }
}
```

Se il tipo conforme soddisfa già il requisito, non è necessaria alcuna implementazione:

```
extension Array: Container {}  // Array satisfies all requirements, including Element
```

Modello delegato

Un *delegato* è un modello di progettazione comune utilizzato nei framework Cocoa e CocoaTouch, in cui una classe delega la responsabilità di implementare alcune funzionalità a un'altra. Ciò segue un principio di separazione delle preoccupazioni, in cui la classe framework implementa funzionalità generiche mentre un'istanza delegato separata implementa il caso d'uso specifico.

Un altro modo per esaminare il modello delegato è in termini di comunicazione dell'oggetto. Objects spesso bisogno di parlare tra loro e per farlo un oggetto deve essere conforme a un protocol per diventare un delegato di un altro oggetto. Una volta che questa configurazione è stata eseguita, l'altro oggetto torna ai suoi delegati quando accadono cose interessanti.

Ad esempio, una vista nell'interfaccia utente per visualizzare un elenco di dati dovrebbe essere responsabile solo della logica di come vengono visualizzati i dati, non per decidere quali dati devono essere visualizzati.

Facciamo un esempio più concreto. se hai due classi, un genitore e un bambino:

```
class Parent { }
class Child { }
```

E vuoi informare il genitore di un cambiamento dal bambino.

In Swift, i delegati vengono implementati usando una dichiarazione di protocol e quindi dichiareremo un protocol che il delegate implementerà. Qui delegato è l'oggetto parent .

```
protocol ChildDelegate: class {
    func childDidSomething()
}
```

Il bambino deve dichiarare una proprietà per memorizzare il riferimento al delegato:

```
class Child {
    weak var delegate: ChildDelegate?
}
```

Si noti che la variabile delegate è facoltativa e il protocollo ChildDelegate è contrassegnato per essere implementato solo per tipo di classe (senza che la variabile delegate non possa essere dichiarata come riferimento weak evitando qualsiasi ciclo di conservazione. Ciò significa che se la variabile delegate non è più riferito altrove, sarà rilasciato). Questo è così la classe genitore registra solo il delegato quando è necessario e disponibile.

Inoltre, per contrassegnare il nostro delegato come weak , dobbiamo limitare il nostro protocollo ChildDelegate ai tipi di riferimento aggiungendo la parola chiave della class nella dichiarazione del protocollo.

In questo esempio, quando il bambino fa qualcosa e ha bisogno di notificare il suo genitore, il bambino chiamerà:

```
delegate?.childDidSomething()
```

Se il delegato è stato definito, al delegato verrà notificato che il bambino ha fatto qualcosa.

La classe genitore dovrà estendere il protocollo ChildDelegate per essere in grado di rispondere alle sue azioni. Questo può essere fatto direttamente sulla classe genitore:

```
class Parent: ChildDelegate {
    ...

    func childDidSomething() {
        print("Yay!")
    }
}
```

O utilizzando un'estensione:

```
extension Parent: ChildDelegate {
    func childDidSomething() {
        print("Yay!")
    }
}
```

Il genitore deve anche dire al bambino che è il delegato del bambino:

```
// In the parent
let child = Child()
child.delegate = self
```

Di default un protocol Swift non consente l'implementazione di una funzione opzionale. Questi possono essere specificati solo se il tuo protocollo è contrassegnato con l'attributo @objc e il modificatore optional .

Ad esempio, UITableView implementa il comportamento generico di una vista tabella in iOS, ma l'utente deve implementare due classi delegate denominate UITableViewDelegate e UITableViewDataSource che implementano l'aspetto e il comportamento delle celle specifiche.

```
@objc public protocol UITableViewDelegate : NSObjectProtocol, UIScrollViewDelegate {

        // Display customization
        optional public func tableView(tableView: UITableView, willDisplayCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath)
        optional public func tableView(tableView: UITableView, willDisplayHeaderView view:
UIView, forSection section: Int)
        optional public func tableView(tableView: UITableView, willDisplayFooterView view:
UIView, forSection section: Int)
        optional public func tableView(tableView: UITableView, didEndDisplayingCell cell:
UITableViewCell, forRowAtIndexPath indexPath: NSIndexPath)
        ...
}
```

È possibile implementare questo protocollo modificando la definizione della classe, ad esempio:

```
class MyViewController : UIViewController, UITableViewDelegate
```

Tutti i metodi non contrassegnati come optional nella definizione del protocollo (UITableViewDelegate in questo caso) devono essere implementati.

Estensione del protocollo per una classe conforme specifica

È possibile scrivere l' **implementazione del protocollo predefinita** per una classe specifica.

```
protocol MyProtocol {
    func doSomething()
}
```

```
extension MyProtocol where Self: UIViewController {
    func doSomething() {
        print("UIViewController default protocol implementation")
    }
}

class MyViewController: UIViewController, MyProtocol { }

let vc = MyViewController()
vc.doSomething() // Prints "UIViewController default protocol implementation"
```

Utilizzo del protocollo RawRepresentable (Extensible Enum)

```
// RawRepresentable has an associatedType RawValue.
// For this struct, we will make the compiler infer the type
// by implementing the rawValue variable with a type of String
//
// Compiler infers RawValue = String without needing typealias
//
struct NotificationName: RawRepresentable {
    let rawValue: String

    static let dataFinished = NotificationNames(rawValue: "DataFinishedNotification")
}
```

Questa struttura può essere estesa altrove per aggiungere casi

```
extension NotificationName {
    static let documentationLaunched = NotificationNames(rawValue:
"DocumentationLaunchedNotification")
}
```

E un'interfaccia può progettare attorno a qualsiasi tipo di RawRepresentable o in particolare alla tua struttura di enum

```
func post(notification notification: NotificationName) -> Void {
    // use notification.rawValue
}
```

Al sito di chiamata, è possibile utilizzare la sintassi del sintassi del punto per il NotificationName typesafe

```
post(notification: .dataFinished)
```

Utilizzo della funzione RawRepresentable generica

```
// RawRepresentable has an associate type, so the
// method that wants to accept any type conforming to
// RawRepresentable needs to be generic
func observe<T: RawRepresentable>(object: T) -> Void {
    // object.rawValue
}
```

Protocolli di sola classe

Un protocollo può specificare che solo una classe può implementarla utilizzando la parola chiave

class nel suo elenco di eredità. Questa parola chiave deve apparire prima di qualsiasi altro protocollo ereditato in questo elenco.

```
protocol ClassOnlyProtocol: class, SomeOtherProtocol {
    // Protocol requirements
}
```

Se un tipo non di classe tenta di implementare ClassOnlyProtocol , verrà generato un errore del compilatore.

```
struct MyStruct: ClassOnlyProtocol {
    // error: Non-class type 'MyStruct' cannot conform to class protocol 'ClassOnlyProtocol'
}
```

Altri protocolli possono ereditare da ClassOnlyProtocol , ma avranno lo stesso requisito di sola classe.

```
protocol MyProtocol: ClassOnlyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}

class MySecondClass: MyProtocol {
    // ClassOnlyProtocol Requirements
    // MyProtocol Requirements
}
```

Semantica di riferimento dei protocolli di sola classe

L'uso di un protocollo di solo classe consente la semantica di riferimento quando il tipo conforme è sconosciuto.

```
protocol Foo : class {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {

    // this assignment requires reference semantics,
    // as foo is a let constant in this scope.
    foo.bar = "new value"
}
```

In questo esempio, dato che Foo è un protocollo di sola classe, l'assegnazione alla bar è valida poiché il compilatore sa che foo è un tipo di classe e quindi ha semantica di riferimento.

Se Foo non era un protocollo di sola classe, veniva restituito un errore del compilatore, poiché il tipo conforme poteva essere un tipo di valore , che richiederebbe un'annotazione var per poter essere mutato.

```
protocol Foo {
    var bar : String { get set }
}

func takesAFoo(foo:Foo) {
    foo.bar = "new value" // error: Cannot assign to property: 'foo' is a 'let' constant
}
```

```
func takesAFoo(foo:Foo) {
```

```
    var foo = foo // mutable copy of foo
    foo.bar = "new value" // no error - satisfies both reference and value semantics
}
```

Variabili deboli del tipo di protocollo

Quando si applica il modificatore weak a una variabile del tipo di protocollo, tale tipo di protocollo deve essere solo di classe, in quanto weak può essere applicato solo ai tipi di riferimento.

```
weak var weakReference : ClassOnlyProtocol?
```

Implementazione del protocollo Hashable

I tipi utilizzati in Sets e Dictionaries(key) devono essere conformi al protocollo Hashable che eredita dal protocollo Equatable .

Hashable tipo personalizzato conforme al protocollo Hashable

- Una proprietà calcolata hashValue
- Definire uno degli operatori di uguaglianza, ovvero == o != .

L'esempio seguente implementa il protocollo Hashable per una struct personalizzata:

```
struct Cell {
    var row: Int
    var col: Int

    init(_ row: Int, _ col: Int) {
        self.row = row
        self.col = col
    }
}

extension Cell: Hashable {

    // Satisfy Hashable requirement
    var hashValue: Int {
        get {
            return row.hashValue^col.hashValue
        }
    }

    // Satisfy Equatable requirement
    static func ==(lhs: Cell, rhs: Cell) -> Bool {
        return lhs.col == rhs.col && lhs.row == rhs.row
    }

}

// Now we can make Cell as key of dictonary
var dict = [Cell : String]()

dict[Cell(0, 0)] = "0, 0"
dict[Cell(1, 0)] = "1, 0"
dict[Cell(0, 1)] = "0, 1"

// Also we can create Set of Cells
var set = Set<Cell>()
```

```
set.insert(Cell(0, 0))
set.insert(Cell(1, 0))
```

Nota : non è necessario che valori diversi nel tipo personalizzato abbiano valori hash diversi, le collisioni sono accettabili. Se i valori di hash sono uguali, verrà utilizzato l'operatore di uguaglianza per determinare l'uguaglianza reale.

Leggi protocolli online: https://riptutorial.com/it/swift/topic/241/protocolli

Capitolo 53: Puntatori di buffer (non sicuri)

introduzione

"Un puntatore del buffer viene utilizzato per l'accesso a basso livello a una regione di memoria. Ad esempio, è possibile utilizzare un puntatore del buffer per l'elaborazione efficiente e la comunicazione dei dati tra app e servizi. "

Estratto da: Apple Inc. "Uso di Swift con Cocoa e Objective-C (Swift 3.1 Edition)." IBooks. https://itun.es/us/utTW7.l

Sei responsabile della gestione del ciclo di vita di qualsiasi memoria con cui lavori con i puntatori del buffer, per evitare perdite o comportamenti non definiti.

Osservazioni

Concetti strettamente allineati **richiesti** per completare la comprensione dei BufferPointer (non sicuri).

- MemoryLayout (*il layout di memoria di un tipo, descrivendo le sue dimensioni, passo e allineamento*).
- Non gestito (*Un tipo per propagare un riferimento a un oggetto non gestito* .)
- UnsafeBufferPointer (*un'interfaccia di raccolta non proprietaria di un buffer di elementi memorizzati in modo contiguo nella memoria*).
- UnsafeBufferPointerIterator (*un iteratore per gli elementi nel buffer a cui fa riferimento un'istanza UnsafeBufferPointer o UnsafeMutableBufferPointer*).
- UnsafeMutableBufferPointer (*un'interfaccia di raccolta non proprietaria di un buffer di elementi mutabili memorizzati in modo contiguo nella memoria*) .
- UnsafeMutablePointer (*puntatore per l'accesso e la manipolazione dei dati di un tipo specifico*) .
- UnsafeMutableRawBufferPointer (*Un'interfaccia di raccolta nono mutabile per i byte in una regione di memoria.*)
- UnsafeMutableRawBufferPointer.Iterator (*un iteratore sui byte visualizzati da un puntatore del buffer raw*) .
- UnsafeMutableRawPointer (*Un puntatore raw per accedere e manipolare dati non tipizzati*) .
- UnsafePointer (*puntatore per accedere ai dati di un tipo specifico*) .
- UnsafeRawBufferPointer (*un'interfaccia di raccolta nonown ai byte in una regione di memoria*) .
- UnsafeRawBufferPointer.Iterator (*un iteratore sui byte visualizzati da un puntatore del buffer raw*) .
- UnsafeRawPointer (*Un puntatore raw per accedere a dati non tipizzati*) .

  (Fonte, Swiftdoc.org)

Examples

UnsafeMutablePointer

```
struct UnsafeMutablePointer<Pointee>
```

Un puntatore per accedere e manipolare i dati di un tipo specifico.

Si utilizzano istanze del tipo UnsafeMutablePointer per accedere ai dati di un tipo specifico in memoria. Il tipo di dati a cui un puntatore può accedere è il tipo Pointee del puntatore. UnsafeMutablePointer non fornisce alcuna garanzia automatica di gestione della memoria o allineamento. Sei responsabile della gestione del ciclo di vita di qualsiasi memoria con cui lavori con puntatori non sicuri per evitare perdite o comportamenti non definiti.

La memoria gestita manualmente può essere non tipizzata o associata a un tipo specifico. Si utilizza il tipo UnsafeMutablePointer per accedere e gestire la memoria che è stata associata a un tipo specifico. (Fonte)

```
import Foundation

let arr = [1,5,7,8]

let pointer = UnsafeMutablePointer<[Int]>.allocate(capacity: 4)
pointer.initialize(to: arr)

let x = pointer.pointee[3]

print(x)

pointer.deinitialize()
pointer.deallocate(capacity: 4)

class A {
  var x: String?

  convenience init (_ x: String) {
    self.init()
    self.x = x
  }

  func description() -> String {
    return x ?? ""
  }
}


let arr2 = [A("OK"), A("OK 2")]
let pointer2 = UnsafeMutablePointer<[A]>.allocate(capacity: 2)
pointer2.initialize(to: arr2)

pointer2.pointee
let y = pointer2.pointee[1]

print(y)

pointer2.deinitialize()
pointer2.deallocate(capacity: 2)
```

Convertito a Swift 3.0 dalla fonte originale

Caso di utilizzo pratico per puntatori di buffer

Decostruire l'uso di un puntatore non sicuro nel metodo della libreria Swift;

```
public init?(validatingUTF8 cString: UnsafePointer<CChar>)
```

Scopo:

Crea una nuova stringa copiando e convalidando i dati UTF-8 con terminazione nullo a cui fa riferimento il puntatore specificato.

Questo inizializzatore non tenta di riparare sequenze di unità di codice UTF-8 mal formate. Se ne viene trovato uno, il risultato dell'inizializzatore è nil . L'esempio seguente chiama questo inizializzatore con puntatori al contenuto di due diversi array CChar --- il primo con sequenze di unità di codice UTF-8 ben formate e il secondo con una sequenza malformata alla fine.

Source , Apple Inc., file di intestazione Swift 3 (per l'accesso all'intestazione: In Playground, Cmd + Fare clic sulla parola Swift) nella riga di codice:

```
import Swift
```

```
let validUTF8: [CChar] = [67, 97, 102, -61, -87, 0]
     validUTF8.withUnsafeBufferPointer { ptr in
         let s = String(validatingUTF8: ptr.baseAddress!)
         print(s as Any)
     }
     // Prints "Optional(Café)"

     let invalidUTF8: [CChar] = [67, 97, 102, -61, 0]
     invalidUTF8.withUnsafeBufferPointer { ptr in
      let s = String(validatingUTF8: ptr.baseAddress!)
      print(s as Any)
     }
// Prints "nil"
```

(Fonte, Apple Inc., Esempio di file Swift Header)

Leggi Puntatori di buffer (non sicuri) online:
https://riptutorial.com/it/swift/topic/9140/puntatori-di-buffer--non-sicuri-

Capitolo 54: Riflessione

Sintassi

- Specchio (che riflette: istanza) // Inizializza uno specchio con il soggetto da riflettere
- mirror.displayStyle // Visualizza lo stile utilizzato per i campi da gioco Xcode
- mirror.description // Rappresentazione testuale di questa istanza, vedere CustomStringConvertible
- mirror.subjectType // Restituisce il tipo del soggetto che viene riflesso
- mirror.superclassMirror // Restituisce il mirror della super-classe del soggetto che viene riflesso

Osservazioni

1. ***Revisione generale:***

Uno Mirror è una struct utilizzata nell'introspezione di un oggetto in Swift. La sua proprietà più importante è l'array per bambini. Un possibile caso d'uso è serializzare una struttura per Core Data . Questo viene fatto convertendo una struct in un NSManagedObject .

2. ***Uso di base per i commenti dello specchio:***

La proprietà children di un Mirror è una matrice di oggetti figlio dall'oggetto che l'istanza Mirror sta riflettendo. Un oggetto child ha due proprietà label e value . Per esempio un bambino potrebbe essere una proprietà con il title del nome e il valore di Game of Thrones: A Song of Ice and Fire .

Examples

Uso di base per Mirror

Creare la classe per essere il soggetto dello specchio

```
class Project {
    var title: String = ""
    var id: Int = 0
    var platform: String = ""
    var version: Int = 0
    var info: String?
}
```

Creare un'istanza che sarà effettivamente l'oggetto dello specchio. Anche qui è possibile aggiungere valori alle proprietà della classe Project.

```
let sampleProject = Project()
sampleProject.title = "MirrorMirror"
sampleProject.id = 199
sampleProject.platform = "iOS"
sampleProject.version = 2
sampleProject.info = "test app for Reflection"
```

Il codice seguente mostra la creazione dell'istanza Mirror. La proprietà child del mirror è AnyForwardCollection<Child> dove Child è una tupla di tipografia per la proprietà e il valore dell'oggetto. Child aveva label: String e value: Any .

```
let projectMirror = Mirror(reflecting: sampleProject)
let properties = projectMirror.children
```

```
print(properties.count)        //5
print(properties.first?.label) //Optional("title")
print(properties.first!.value) //MirrorMirror
print()

for property in properties {
    print("\(property.label!):\(property.value)")
}
```

Uscita in Playground o Console in Xcode per il ciclo for sopra.

```
title:MirrorMirror
id:199
platform:iOS
version:2
info:Optional("test app for Reflection")
```

Testato in Playground su Xcode 8 beta 2

Ottenere il tipo e i nomi delle proprietà per una classe senza dover istanziarla

Usando la classe Swift Mirror funziona se si desidera estrarre *nome* , *valore* e *tipo* (Swift 3: type(of: value) , Swift 2: value.dynamicType) di proprietà per **un'istanza** di una determinata classe.

Se la classe eredita da NSObject , è possibile utilizzare il metodo class_copyPropertyList insieme a property_getAttributes per trovare il *nome* e i *tipi* di proprietà per una classe, **senza avere un'istanza di essa** . Ho creato un progetto su Github per questo, ma ecco il codice:

```
func getTypesOfProperties(in clazz: NSObject.Type) -> Dictionary<String, Any>? {
    var count = UInt32()
    guard let properties = class_copyPropertyList(clazz, &count) else { return nil }
    var types: Dictionary<String, Any> = [:]
    for i in 0..<Int(count) {
        guard let property: objc_property_t = properties[i], let name = getNameOf(property:
property) else { continue }
        let type = getTypeOf(property: property)
        types[name] = type
    }
    free(properties)
    return types
}

func getTypeOf(property: objc_property_t) -> Any {
    guard let attributesAsNSString: NSString = NSString(utf8String:
property_getAttributes(property)) else { return Any.self }
    let attributes = attributesAsNSString as String
    let slices = attributes.components(separatedBy: "\"")
    guard slices.count > 1 else { return getPrimitiveDataType(withAttributes: attributes) }
    let objectClassName = slices[1]
    let objectClass = NSClassFromString(objectClassName) as! NSObject.Type
    return objectClass
}

   func getPrimitiveDataType(withAttributes attributes: String) -> Any {
        guard let letter = attributes.substring(from: 1, to: 2), let type =
primitiveDataTypes[letter] else { return Any.self }
        return type
    }
```

Dove primitiveDataTypes è un dizionario che associa una lettera nella stringa di attributo a un tipo di valore:

```
let primitiveDataTypes: Dictionary<String, Any> = [
    "c" : Int8.self,
    "s" : Int16.self,
    "i" : Int32.self,
    "q" : Int.self, //also: Int64, NSInteger, only true on 64 bit platforms
    "S" : UInt16.self,
    "I" : UInt32.self,
    "Q" : UInt.self, //also UInt64, only true on 64 bit platforms
    "B" : Bool.self,
    "d" : Double.self,
    "f" : Float.self,
    "{" : Decimal.self
]

   func getNameOf(property: objc_property_t) -> String? {
        guard let name: NSString = NSString(utf8String: property_getName(property)) else {
return nil }
        return name as String
    }
```

Può estrarre il NSObject.Type di tutte le proprietà che il tipo di classe eredita da NSObject come NSDate (Swift3: Date), NSString (Swift3: String ?) E NSNumber , tuttavia è memorizzato nel tipo Any (come puoi vedere come tipo del valore del dizionario restituito dal metodo). Ciò è dovuto alle limitazioni dei value types di value types come Int, Int32, Bool. Poiché questi tipi non ereditano da NSObject, chiamando .self su es. Int - Int.self non restituisce NSObject.Type, ma il tipo Any . Quindi il metodo restituisce Dictionary<String, Any>? e non il Dictionary<String, NSObject.Type>? .

Puoi usare questo metodo in questo modo:

```
class Book: NSObject {
    let title: String
    let author: String?
    let numberOfPages: Int
    let released: Date
    let isPocket: Bool

    init(title: String, author: String?, numberOfPages: Int, released: Date, isPocket: Bool) {
        self.title = title
        self.author = author
        self.numberOfPages = numberOfPages
        self.released = released
        self.isPocket = isPocket
    }
}

guard let types = getTypesOfProperties(in: Book.self) else { return }
for (name, type) in types {
    print("'\(name)' has type '\(type)'")
}
// Prints:
// 'title' has type 'NSString'
// 'numberOfPages' has type 'Int'
// 'author' has type 'NSString'
// 'released' has type 'NSDate'
// 'isPocket' has type 'Bool'
```

Puoi anche provare a lanciare Any to NSObject.Type , che avrà successo per tutte le proprietà che ereditano da NSObject , quindi puoi controllare il tipo usando l'operatore standard == :

```
func checkPropertiesOfBook() {
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if let objectType = type as? NSObject.Type {
            if objectType == NSDate.self {
                print("Property named '\(name)' has type 'NSDate'")
            } else if objectType == NSString.self {
                print("Property named '\(name)' has type 'NSString'")
            }
        }
    }
}
```

Se dichiari questo operatore == personalizzato:

```
func ==(rhs: Any, lhs: Any) -> Bool {
    let rhsType: String = "\(rhs)"
    let lhsType: String = "\(lhs)"
    let same = rhsType == lhsType
    return same
}
```

È quindi possibile controllare anche il tipo di value types di value types come questo:

```
func checkPropertiesOfBook() {
    guard let types = getTypesOfProperties(in: Book.self) else { return }
    for (name, type) in types {
        if type == Int.self {
            print("Property named '\(name)' has type 'Int'")
        } else if type == Bool.self {
            print("Property named '\(name)' has type 'Bool'")
        }
    }
}
```

LIMITAZIONI Questa soluzione non funziona quando i value types sono opzionali. Se hai dichiarato una proprietà nella sottoclasse NSObject come questa: var myOptionalInt: Int? , il codice sopra non troverà quella proprietà perché il metodo class_copyPropertyList non contiene tipi di valore opzionali.

Leggi Riflessione online: https://riptutorial.com/it/swift/topic/1201/riflessione

Capitolo 55: RxSwift

Examples

Nozioni di base su RxSwift

FRP, o Functional Reactive Programming, ha alcuni termini di base che è necessario conoscere.

Ogni pezzo di dati può essere rappresentato come Observable , che è un flusso di dati asincrono. La potenza di FRP è rappresentata da eventi sincroni e asincroni come flussi, Observable e fornendo la stessa interfaccia per lavorare con esso.

Solitamente Observable contiene diversi (o nessuno) eventi che .Next la data - .Next Eventi successivi, e quindi può essere terminata con successo (.Success) o con un errore (.Error).

Diamo un'occhiata al seguente diagramma di marmo:

```
--(1)--(2)--(3)|-->
```

In questo esempio c'è un flusso di valori Int . Con il passare del tempo, tre eventi. .Next sono verificati e quindi il flusso è stato interrotto correttamente.

```
--X->
```

Il diagramma sopra mostra un caso in cui non sono stati emessi dati e. .Error evento di .Error termina l' Observable .

Prima di andare avanti, ci sono alcune risorse utili:

1. RxSwift . Guarda gli esempi, leggi i documenti e inizia.
2. RxSwift Slack room ha alcuni canali per la risoluzione dei problemi educativi.
3. Gioca con RxMarbles per sapere cosa fa l'operatore e quale è il più utile nel tuo caso.
4. Dai un'occhiata a questo esempio , esplora il codice da solo.

Creare osservabili

RxSwift offre molti modi per creare un Observable , diamo un'occhiata:

```
import RxSwift

let intObservale = Observable.just(123) // Observable<Int>
let stringObservale = Observable.just("RxSwift") // Observable<String>
let doubleObservale = Observable.just(3.14) // Observable<Double>
```

Quindi, gli osservabili sono creati. Essi contengono solo un valore e quindi termina con successo. Tuttavia, non è successo nulla dopo che è stato creato. Perché?

Ci sono due passaggi per lavorare con Observable s: **osservi** qualcosa per *creare* uno stream e poi ti **iscrivi** allo stream o lo **leghi** a qualcosa per *interagire* con esso.

```
Observable.just(12).subscribe {
    print($0)
}
```

La console stamperà:

```
.Next(12)
.Completed()
```

E se mi interessa solo lavorare con i dati, che si svolgono in .Next Eventi successivi, vorrei utilizzare l'operatore subscribeNext :

```
Observable.just(12).subscribeNext {
    print($0) // prints "12" now
}
```

Se voglio creare un osservabile di molti valori, utilizzo diversi operatori:

```
Observable.of(1,2,3,4,5).subscribeNext {
    print($0)
}
// 1
// 2
// 3
// 4
// 5

// I can represent existing data types as Observables also:
[1,2,3,4,5].asObservable().subscribeNext {
    print($0)
}
// result is the same as before.
```

E infine, forse voglio un Observable che funzioni. Ad esempio, è conveniente racchiudere un'operazione di rete in Observable<SomeResultType> . Diamo un'occhiata a fare uno può ottenere questo:

```
Observable.create { observer in    // create an Observable ...
    MyNetworkService.doSomeWorkWithCompletion { (result, error) in
        if let e = error {
            observer.onError(e)    // ..that either holds an error
        } else {
            observer.onNext(result) // ..or emits the data
            observer.onCompleted()  // ..and terminates successfully.
        }
    }
    return NopDisposable.instance // here you can manually free any resources
                                  //in case if this observable is being disposed.
}
```

smaltimento

Dopo aver creato la sottoscrizione, è importante gestirne la corretta deallocazione.

I documenti ce l'hanno detto

> Se una sequenza termina in un tempo limitato, non chiamare dispose o non usare addDisposableTo (disposeBag) non causerà alcuna perdita permanente di risorse. Tuttavia, tali risorse verranno utilizzate fino al completamento della sequenza, terminando la produzione di elementi o restituendo un errore.

Esistono due modi per deallocare le risorse.

1. Utilizzo di disposeBag s e addDisposableTo operator.
2. Usando takeUntil operatore takeUntil .

Nel primo caso si passa manualmente l'abbonamento all'oggetto DisposeBag , che cancella correttamente tutta la memoria acquisita.

```
let bag = DisposeBag()
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(bag)
```

In realtà non è necessario creare DisposeBag in ogni classe che si crea, basta dare un'occhiata al progetto *RxSwift Community* denominato NSObject + Rx . Utilizzando il framework il codice sopra può essere riscritto come segue:

```
Observable.just(1).subscribeNext {
    print($0)
}.addDisposableTo(rx_disposeBag)
```

Nel secondo caso, se il tempo di sottoscrizione coincide con la durata dell'oggetto self , è possibile implementare lo smaltimento utilizzando takeUntil(rx_deallocated) :

```
let _ = sequence
    .takeUntil(rx_deallocated)
    .subscribe {
        print($0)
    }
```

Attacchi

```
Observable.combineLatest(firstName.rx_text, lastName.rx_text) { $0 + " " + $1 }
.map { "Greetings, \($0)" }
.bindTo(greetingLabel.rx_text)
```

Usando l'operatore combineLatest ogni volta che un oggetto viene emesso da uno dei due Observables , combinare l'ultimo oggetto emesso da ciascun Observable . In questo modo combiniamo il risultato dei due UITextField creando un nuovo messaggio con il testo "Greetings, \($0)" usando l'interpolazione della stringa per legare in seguito al testo di un UILabel .

Possiamo legare i dati a qualsiasi UITableView e UICollectionView in un modo molto semplice:

```
viewModel
.rows
.bindTo(resultsTableView.rx_itemsWithCellIdentifier("WikipediaSearchCell", cellType:
WikipediaSearchCell.self)) { (_, viewModel, cell) in
    cell.title = viewModel.title
    cell.url = viewModel.url
}
.addDisposableTo(disposeBag)
```

Questo è un wrapper Rx attorno al metodo di origine dati cellForRowAtIndexPath . E anche Rx si prende cura dell'implementazione di numberOfRowsAtIndexPath , che è un metodo obbligatorio in senso tradizionale, ma non è necessario implementarlo qui, è curato.

RxCocoa e ControlEvents

RxSwift fornisce non solo i modi per controllare i dati, ma anche per rappresentare le azioni dell'utente in modo reattivo.

RxCocoa contiene tutto ciò di cui hai bisogno. Trasporta la maggior parte delle proprietà dei componenti dell'interfaccia utente in s Observable , ma non proprio. Ci sono alcuni Observable aggiornati chiamati ControlEvent (che rappresentano eventi) e ControlProperties (che rappresentano proprietà, sorpresa!). Queste cose Observable flussi Observable sotto il cofano, ma hanno anche alcune sfumature:

- Non fallisce mai, quindi nessun errore.
- Complete sequenza sul controllo che viene deallocato.
- Fornisce eventi sul thread principale (MainScheduler.instance).

Fondamentalmente, puoi lavorare con loro come al solito:

```
button.rx_tap.subscribeNext { _ in   // control event
    print("User tapped the button!")
}.addDisposableTo(bag)

textField.rx_text.subscribeNext { text in // control property
    print("The textfield contains: \(text)")
}.addDisposableTo(bag)
// notice that ControlProperty generates .Next event on subscription
// In this case, the log will display
// "The textfield contains: "
// at the very start of the app.
```

Questo è molto importante da usare: finché usi Rx, dimenticati del materiale @IBAction , tutto ciò di cui hai bisogno puoi collegarlo e configurarlo contemporaneamente. Ad esempio, il metodo viewDidLoad del tuo controller di visualizzazione è un buon candidato per descrivere come funzionano i componenti dell'interfaccia utente.

Ok, un altro esempio: supponiamo di avere un campo di testo, un pulsante e un'etichetta. Vogliamo **validare** il **testo** nel campo di testo quando si **tocca** il pulsante, e **visualizzare** i risultati in etichetta. Sì, sembra un altro compito di email di conferma, eh?

Prima di tutto, prendiamo il button.rx_tap ControlEvent:

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

Qui le parentesi vuote mostrano i tocchi dell'utente. Successivamente, prendiamo ciò che è scritto nel campo testo con withLatestFrom operatore withLatestFrom (withLatestFrom un'occhiata qui , immaginate che il flusso superiore rappresenti i tocchi utente, quello in basso rappresenta il testo nel campo di testo).

```
button.rx_tap.withLatestFrom(textField.rx_text)

----("")--------------------("123")--->
//  ^ tap   ^ i wrote 123    ^ tap
```

Bello, abbiamo un flusso di stringhe da convalidare, emesse solo quando abbiamo bisogno di convalidare.

Qualsiasi Observable ha operatori familiari come la map o il filter , prenderemo la map per convalidare il testo. Crea tu stesso la funzione di validateEmail mail, usa qualsiasi regex che desideri.

```
button.rx_tap                                  // ControlEvent<Void>
        .withLatestFrom(textField.rx_text)     // Observable<String>
        .map(validateEmail)                    // Observable<Bool>
        .map { (isCorrect) in
            return isCorrect ? "Email is correct" : "Input the correct one, please"
        }                                      // Observable<String>
        .bindTo(label.rx_text)
        .addDisposableTo(bag)
```

Fatto! Se hai bisogno di più logica personalizzata (come mostrare le visualizzazioni di errore in caso di errore, effettuare una transizione su un altro schermo in caso di successo ...), iscriviti al flusso Bool finale e scrivilo lì.

Leggi RxSwift online: https://riptutorial.com/it/swift/topic/4890/rxswift

Capitolo 56: Structs

Examples

Nozioni di base di strutture

```
struct Repository {
    let identifier: Int
    let name: String
    var description: String?
}
```

Definisce una struttura di Repository con tre proprietà memorizzate, un identifier numero intero, un name stringa e una description stringa facoltativa. L' identifier e il name sono costanti, in quanto sono stati dichiarati usando let -keyword. Una volta impostati durante l'inizializzazione, non possono essere modificati. La descrizione è una variabile. La modifica aggiorna il valore della struttura.

I tipi di struttura ricevono automaticamente un inizializzatore membro se non definiscono alcuno dei propri inizializzatori personalizzati. La struttura riceve un inizializzatore membro, anche se ha memorizzato le proprietà che non hanno valori predefiniti.

Repository contiene tre proprietà memorizzate di cui solo la description ha un valore predefinito (nil). Inoltre non definisce alcun inizializzatore, quindi riceve gratuitamente un inizializzatore membro:

```
let newRepository = Repository(identifier: 0, name: "New Repository", description: "Brand New Repository")
```

Le strutture sono tipi di valore

A differenza delle classi, che vengono passate per riferimento, le strutture passano attraverso la copia:

```
first = "Hello"
second = first
first += " World!"
// first == "Hello World!"
// second == "Hello"
```

String è una struttura, quindi è copiata sul compito.

Anche le strutture non possono essere confrontate con l'operatore di identità:

```
window0 === window1 // works because a window is a class instance
"hello" === "hello" // error: binary operator '===' cannot be applied to two 'String' operands
```

Ogni istanza di due strutture è considerata identica se si confronta uguale.

Collettivamente, questi tratti che differenziano le strutture dalle classi sono ciò che rende i tipi di valore delle strutture.

Mutare una Struct

Un metodo di una struttura che modifica il valore della struttura stessa deve essere preceduto dalla parola chiave mutating

```
struct Counter {
    private var value = 0

    mutating func next() {
        value += 1
    }
}
```

Quando puoi usare i metodi di muting

I metodi mutating sono disponibili solo sui valori della struttura all'interno delle variabili.

```
var counter = Counter()
counter.next()
```

Quando NON puoi usare metodi di muting

D'altra parte, i metodi di mutating NON sono disponibili sui valori di struct all'interno di costanti

```
let counter = Counter()
counter.next()
//  error: cannot use mutating member on immutable value: 'counter' is a 'let' constant
```

Le strutture non possono ereditare

A differenza delle classi, le strutture non possono ereditare:

```
class MyView: NSView { }  // works

struct MyInt: Int { } // error: inheritance from non-protocol type 'Int'
```

Le strutture, tuttavia, possono adottare protocolli:

```
struct Vector: Hashable { ... }  // works
```

Accesso ai membri di struct

In Swift, le strutture usano una semplice "sintassi del punto" per accedere ai loro membri.

Per esempio:

```
struct DeliveryRange {
  var range: Double
  let center: Location
}
let storeLocation = Location(latitude: 44.9871,
                             longitude: -93.2758)
var pizzaRange = DeliveryRange(range: 200,
                               center: storeLocation)
```

Puoi accedere (stampare) la gamma in questo modo:

```
    print(pizzaRange.range) // 200
```

Puoi anche accedere ai membri dei membri usando la sintassi del punto:

```
print(pizzaRange.center.latitude) // 44.9871
```

Simile a come puoi leggere i valori con la sintassi del punto, puoi anche assegnarli.

```
pizzaRange.range = 250
```

Leggi Structs online: https://riptutorial.com/it/swift/topic/255/structs

Capitolo 57: Swift HTTP server di Kitura

introduzione

Server Swift con Kitura

Kitura è una struttura web scritta in swift creata per i servizi web. È molto facile da configurare per le richieste HTTP. Per l'ambiente, ha bisogno di OS X con XCode installato o di Linux che esegue swift 3.0.

Examples

Ciao domanda mondiale

Configurazione

Per prima cosa, crea un file chiamato Package.swift. Questo è il file che dice al compilatore veloce dove si trovano le librerie. In questo esempio ciao mondo, stiamo usando GitHub repos. Abbiamo bisogno di Kitura e HeliumLogger . Inserisci il seguente codice all'interno di Package.swift. Ha specificato il nome del progetto come *kitura-helloworld* e anche gli URL delle dipendenze.

```
import PackageDescription
let package = Package(
    name: "kitura-helloworld",
        dependencies: [
            .Package(url: "https://github.com/IBM-Swift/HeliumLogger.git", majorVersion: 1,
minor: 6),
            .Package(url: "https://github.com/IBM-Swift/Kitura.git", majorVersion: 1, minor:
6) ] )
```

Quindi, crea una cartella chiamata Sorgenti. Dentro, crea un file chiamato main.swift. Questo è il file che implementiamo tutta la logica per questa applicazione. Inserire il seguente codice in questo file principale.

Importa librerie e attiva la registrazione

```
import Kitura
import Foundation
import HeliumLogger

HeliumLogger.use()
```

Aggiungere un router. Il router specifica un percorso, un tipo, ecc. Della richiesta HTTP. Qui stiamo aggiungendo un gestore di richieste GET che stampa *Hello world* , quindi una richiesta di post che legge il testo in chiaro dalla richiesta e quindi lo rimanda indietro.

```
let router = Router()

router.get("/get") {
    request, response, next in
    response.send("Hello, World!")
    next()
}

router.post("/post") {
    request, response, next in
    var string: String?
    do{
```

```
        string = try request.readString()

    } catch let error {
        string = error.localizedDescription
    }
    response.send("Value \(string!) received.")
    next()
}
```

Specificare una porta per eseguire il servizio

```
let port = 8080
```

Collegare insieme il router e la porta e aggiungerli come servizio HTTP

```
Kitura.addHTTPServer(onPort: port, with: router)
Kitura.run()
```

Eseguire

Passare alla cartella principale con il file Package.swift e la cartella Risorse. Esegui il seguente comando. Il compilatore Swift scaricherà automaticamente le risorse menzionate in Package.swift nella cartella Pacchetti, quindi compila queste risorse con main.swift

```
swift build
```

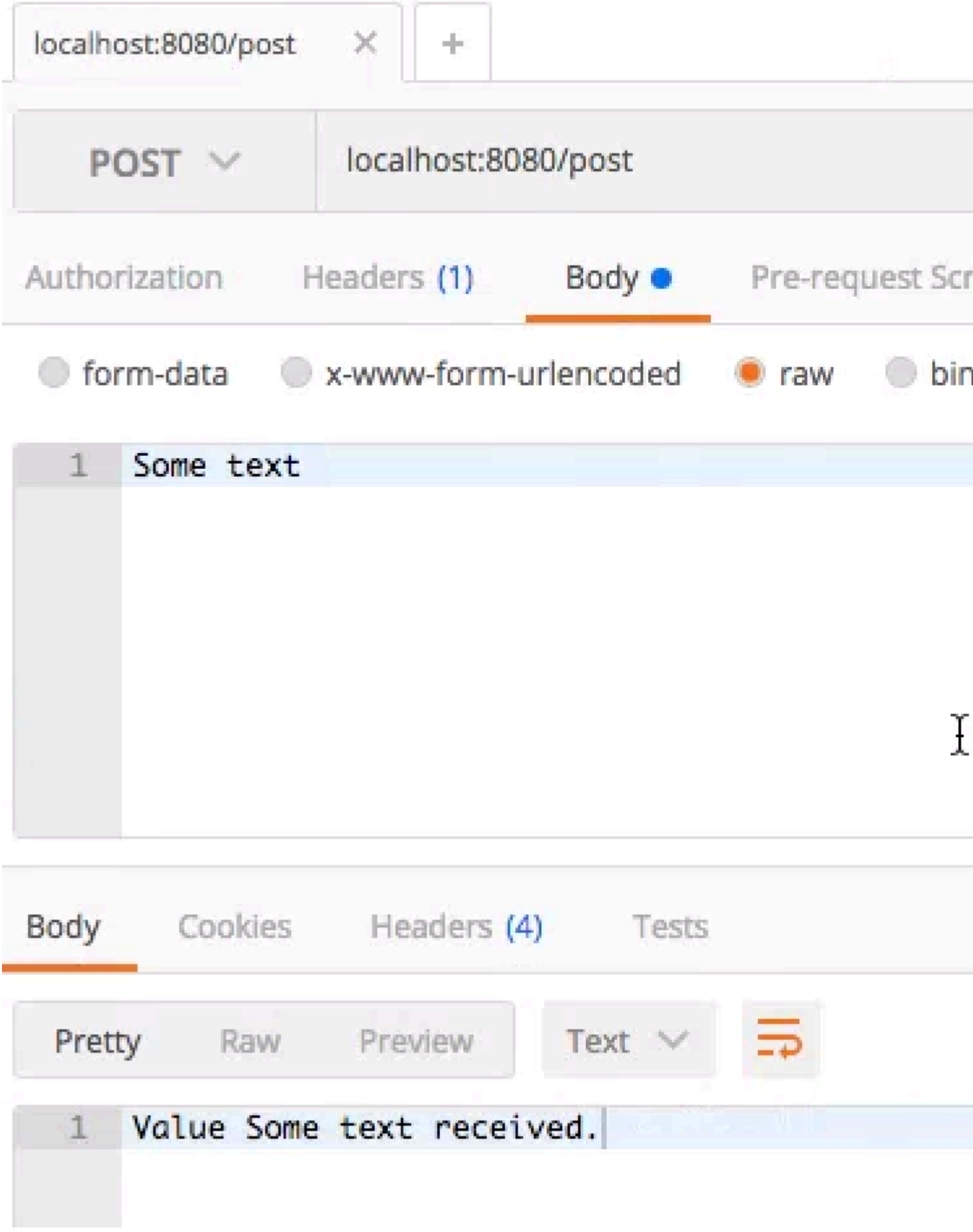
Al termine della costruzione, l'eseguibile verrà posizionato in questa posizione. Fare doppio clic su questo file eseguibile per avviare il server.

```
.build/debug/kitura-helloworld
```

Convalidare

Aprire un browser, digitare localhost:8080/get come url e premere invio. Dovrebbe uscire la pagina del mondo Hello.

Aprire un'app di richiesta HTTP, inviare testo normale a localhost:8080/post . La stringa di risposta mostrerà il testo inserito correttamente.



Leggi Swift HTTP server di Kitura online: https://riptutorial.com/it/swift/topic/10690/swift-http-server-di-kitura

Capitolo 58: Swift Package Manager

Examples

Creazione e utilizzo di un semplice pacchetto Swift

Per creare un pacchetto Swift, apri un terminale quindi crea una cartella vuota:

```
mkdir AwesomeProject
cd AwesomeProject
```

E inizia un repository Git:

```
git init
```

Quindi crea il pacchetto stesso. Si potrebbe creare manualmente la struttura del pacchetto, ma c'è un modo semplice usando il comando CLI.

Se vuoi fare un eseguibile:

```
swift package init --type executable
```

Saranno generati diversi file. Tra questi, *main.swift* sarà il punto di ingresso per la tua applicazione.

Se vuoi creare una libreria:

```
swift package init --type library
```

Il file *AwesomeProject.swift* generato verrà utilizzato come file principale per questa libreria.

In entrambi i casi è possibile aggiungere altri file Swift nella cartella *Sorgenti* (si applicano le normali regole per il controllo degli accessi).

Il file *Package.swift* stesso verrà automaticamente popolato con questo contenuto:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject"
)
```

Il controllo della versione del pacchetto avviene con i tag Git:

```
git tag '1.0.0'
```

Una volta trasferito su un repository Git remoto o locale, il pacchetto sarà disponibile per altri progetti.

Il tuo pacchetto è ora pronto per essere compilato:

```
swift build
```

Il progetto compilato sarà disponibile nella cartella *.build / debug* .

Il tuo pacchetto può anche risolvere dipendenze da altri pacchetti. Ad esempio, se si desidera includere "SomeOtherPackage" nel proprio progetto, modificare il file *Package.swift* per includere la dipendenza:

```
import PackageDescription

let package = Package(
    name: "AwesomeProject",
    targets: [],
    dependencies: [
        .Package(url: "https://github.com/someUser/SomeOtherPackage.git",
                 majorVersion: 1),
    ]
)
```

Quindi ricostruisci il progetto: lo Swift Package Manager risolverà automaticamente, scaricherà e creerà le dipendenze.

Leggi Swift Package Manager online: https://riptutorial.com/it/swift/topic/5144/swift-package-manager

Capitolo 59: Typealias

Examples

tipografie per chiusure con parametri

```
typealias SuccessHandler = (NSURLSessionDataTask, AnyObject?) -> Void
```

Questo blocco di codice crea un alias di tipo denominato SuccessHandler , proprio nello stesso modo var string = "" crea una variabile con la string nome.

Ora ogni volta che usi SuccessHandler , ad esempio:

```
func example(_ handler: SuccessHandler) {}
```

Stai scrivendo intensamente:

```
func example(_ handler: (NSURLSessionDataTask, AnyObject?) -> Void) {}
```

tipografie per chiusure vuote

```
typealias Handler = () -> Void
typealias Handler = () -> ()
```

Questo blocco crea un alias di tipo che funziona come una funzione Void to Void (non accetta parametri e non restituisce nulla).

Ecco un esempio di utilizzo:

```
var func: Handler?

func = {}
```

tipalità per altri tipi

```
typealias Number = NSNumber
```

Puoi anche usare un alias di tipo per dare un tipo ad un altro nome per renderlo più facile da ricordare o per rendere il tuo codice più elegante.

tipografie per tuple

```
typealias PersonTuple = (name: String, age: Int, address: String)
```

E questo può essere usato come:

```
func getPerson(for name: String) -> PersonTuple {
    //fetch from db, etc
    return ("name", 45, "address")
}
```

Leggi Typealias online: https://riptutorial.com/it/swift/topic/7552/typealias

Capitolo 60: Variabili e proprietà

Osservazioni

Proprietà : associato a un tipo

Variabili : non associate a un tipo

Consulta l' iBook della lingua di programmazione rapida per maggiori informazioni.

Examples

Creare una variabile

Dichiarare una nuova variabile con var , seguita da un nome, tipo e valore:

```
var num: Int = 10
```

Le variabili possono avere i loro valori modificati:

```
num = 20 // num now equals 20
```

A meno che non siano definiti con let :

```
let num: Int = 10 // num cannot change
```

Swift deduce il tipo di variabile, quindi non devi sempre dichiarare il tipo di variabile:

```
let ten = 10 // num is an Int
let pi = 3.14 // pi is a Double
let floatPi: Float = 3.14 // floatPi is a Float
```

I nomi delle variabili non sono limitati a lettere e numeri, ma possono anche contenere la maggior parte degli altri caratteri Unicode, sebbene vi siano alcune restrizioni

> I nomi di costanti e variabili non possono contenere caratteri di spazi bianchi, simboli matematici, frecce, punti di codice Unicode di uso privato (o non validi) o caratteri di disegno di linee e riquadri. Né possono iniziare con un numero
>
> **Fonte** developer.apple.com

```
var π: Double = 3.14159
var 🍎🍎: String = "Apples"
```

Nozioni di base sulla proprietà

Le proprietà possono essere aggiunte a una classe o una struttura (anche tecnicamente enum , vedi esempio "Proprietà calcolate"). Questi aggiungono valori che si associano alle istanze di classi / structs:

```
class Dog {
    var name = ""
}
```

Nel caso precedente, le istanze di Dog hanno una proprietà denominata name di tipo String . La proprietà è accessibile e modificata in caso di Dog :

```
let myDog = Dog()
myDog.name = "Doggy" // myDog's name is now "Doggy"
```

Questi tipi di proprietà sono considerati **proprietà memorizzate** , poiché memorizzano qualcosa su un oggetto e ne influenzano la memoria.

Proprietà memorizzate pigre

Le proprietà memorizzate pigro hanno valori che non vengono calcolati fino al primo accesso. Ciò è utile per il salvataggio della memoria quando il calcolo della variabile è dispendioso dal punto di vista computazionale. Dichiari una proprietà lazy con lazy :

```
lazy var veryExpensiveVariable = expensiveMethod()
```

Spesso è assegnato a un valore di ritorno di una chiusura:

```
lazy var veryExpensiveString = { () -> String in
    var str = expensiveStrFetch()
    str.expensiveManipulation(integer: arc4random_uniform(5))
    return str
}()
```

Le proprietà memorizzate pigro devono essere dichiarate con var .

Proprietà calcolate

Diversamente dalle proprietà memorizzate, le proprietà **calcolate** vengono create con un getter e un setter, eseguendo il codice necessario quando si accede e si imposta. Le proprietà calcolate devono definire un tipo:

```
var pi = 3.14

class Circle {
    var radius = 0.0
    var circumference: Double {
        get {
            return pi * radius * 2
        }
        set {
            radius = newValue / pi / 2
        }
    }
}

let circle = Circle()
circle.radius = 1
print(circle.circumference) // Prints "6.28"
circle.circumference = 14
print(circle.radius) // Prints "2.229..."
```

Una proprietà calcolata di sola lettura è ancora dichiarata con una var :

```
var circumference: Double {
    get {
        return pi * radius * 2
    }
}
```

Le proprietà calcolate di sola lettura possono essere abbreviate per escludere get :

```
var circumference: Double {
    return pi * radius * 2
}
```

Variabili locali e globali

Le variabili locali sono definite all'interno di una funzione, metodo o chiusura:

```
func printSomething() {
    let localString = "I'm local!"
    print(localString)
}

func printSomethingAgain() {
    print(localString) // error
}
```

Le variabili globali sono definite al di fuori di una funzione, metodo o chiusura e non sono definite all'interno di un tipo (si pensi al di fuori di tutte le parentesi). Possono essere utilizzati ovunque:

```
let globalString = "I'm global!"
print(globalString)

func useGlobalString() {
    print(globalString) // works!
}

for i in 0..<2 {
    print(globalString) // works!
}

class GlobalStringUser {
    var computeGlobalString {
        return globalString // works!
    }
}
```

Le variabili globali sono definite pigramente (vedi esempio "Proprietà Lazy").

Tipo Proprietà

Le proprietà del tipo sono proprietà sul tipo stesso, non sull'istanza. Possono essere sia proprietà memorizzate che calcolate. Si dichiara una proprietà di tipo con static :

```
struct Dog {
    static var noise = "Bark!"
}

print(Dog.noise) // Prints "Bark!"
```

In una classe, è possibile utilizzare la parola chiave della class anziché static per renderla sovrascrivibile. Tuttavia, puoi applicarlo solo su proprietà calcolate:

```
class Animal {
    class var noise: String {
        return "Animal noise!"
    }
```

```
}
class Pig: Animal {
    override class var noise: String {
        return "Oink oink!"
    }
}
```

Questo è usato spesso con il modello singleton .

Osservatori di proprietà

Gli osservatori di proprietà rispondono alle modifiche al valore di una proprietà.

```
var myProperty = 5 {
    willSet {
        print("Will set to \(newValue). It was previously \(myProperty)")
    }
    didSet {
        print("Did set to \(myProperty). It was previously \(oldValue)")
    }
}
myProperty = 6
// prints: Will set to 6, It was previously 5
// prints: Did set to 6. It was previously 5
```

- willSet viene chiamato **prima che** myProperty sia impostato. Il nuovo valore è disponibile come newValue e il vecchio valore è ancora disponibile come myProperty .
- didSet viene chiamato **dopo che** myProperty è impostato. Il vecchio valore è disponibile come oldValue e il nuovo valore è ora disponibile come myProperty .

  Nota: didSet e willSet non verranno richiamati nei seguenti casi:

  - Assegnare un valore iniziale
  - Modifica della variabile all'interno del proprio didSet o willSet

- I nomi dei parametri per oldValue e newValue di didSet e willSet possono anche essere dichiarati per aumentare la leggibilità:

```
var myFontSize = 10 {
    willSet(newFontSize) {
        print("Will set font to \(newFontSize), it was \(myFontSize)")
    }
    didSet(oldFontSize) {
        print("Did set font to \(myFontSize), it was \(oldFontSize)")
    }
}
```

Attenzione: mentre è supportato per dichiarare i nomi dei parametri setter, si dovrebbe essere prudenti a non mescolare i nomi:

- willSet(oldValue) e didSet(newValue) sono completamente legali, ma confonderanno considerevolmente i lettori del tuo codice.

Leggi Variabili e proprietà online: https://riptutorial.com/it/swift/topic/536/variabili-e-proprieta

Titoli di coda

S. No	Capitoli	Contributors
1	Iniziare con Swift Language	Ahmad F, Anas, andy, Cailean Wilkinson, Claw, Community, esthepiking, Ferenc Kiss, Jim, jtbandes, Luca Angeletti, Luca Angioloni, Moritz, nmnsud, Seyyed Parsa Neshaei, sudo, Sunil Prajapati, Tanner, user3581248
2	Accesso a Swift	Adam Bardon, D4ttatraya, DanHabib, jglasse, Moritz, paper1111, RamenChef
3	Algoritmi con Swift	Austin Conlon, Bohdan Savych, Hady Nourallah, SteBra, Stephen Leppik, Tommie C.
4	Archi e personaggi	Akshit Soota, Andrea Antonioni, antonio081014, AstroCB, Caleb Kleveter, Carpsen90, egor.zhdan, Feldur, Franck Dernoncourt, Govind Rai, Greg, Guilherme Torres Castro, Hamish, HariKrishnan.P, HeMet, JAL, Jason Sturges, Jojodmo, jtbandes, kabiroberai, Kirit Modi, Kyle KIM, Lope, LopSae, Luca Angeletti , LukeSideWalker, Magisch, Mahmoud Adam, Matt, Matthew Seaman, Max Desiatov, maxkonovalov, Moritz, Nate Cook, Nikolai Ruhe, Panda, Patrick, pixatlazaki, QoP, sdasdadas, Shanmugaraja G, shim, solidcell, Sunil Sharma, Suragch, taylor swift, The_Curry_Man, ThrowingSpoon, user3480295, Victor Sigler, Vinupriya Arivazhagan, WMios
5	Array	BaSha, Ben Trengrove, D4ttatraya, DarkDust, Hamish, jtbandes, Kevin, Luca Angeletti, Moritz, Moriya, nathan, pableiros, Palle , Saagar Jha, Stephen Leppik, ThrowingSpoon, tomahh, toofani, vacawama, Vladimir Nul
6	blocchi	Matt
7	booleani	Andreas, jtbandes, Kevin, pableiros
8	chiusure	ctietze, Duncan C, Hamish, Jojodmo, jtbandes, LopSae, Matthew Seaman, Moritz, Timothy Rascher, Tom Magnusson
9	Classi	Dalija Prasnikar, esthepiking, FelixSFD, jtbandes, Luca Angeletti, Matt, Ryan H., tktsubota, Tommie C., Zack
10	Completamento dell'handler	Maysam, Moritz
11	Concorrenza	Adda_25, Ahmad F, FelixSFD, JAL, LukeSideWalker, M_G, Matthew Seaman, Palle, Rob, Santa Claus
12	Condizionali	AK1, atxe, Brduca, Community, Dalija Prasnikar, DarkDust, Hamish, jtbandes, ThaNerd, Thomas Gerot, tktsubota, toofani, torinpitchers
13	Controllo di accesso	4444, Asdrubal, FelixSFD
14	Convenzioni di stile	Grimxn, Moritz, Palle, Ryan H.
15	Crittografia AES	Matt, Stephen Leppik, zaph
16	Derivazione chiave	BUZZE, zaph

	PBKDF2	
17	Design Patterns - Creazionale	Ahmad F, AMAN77, Brduca, Dalija Prasnikar, Ian Rahman, Moritz, SeanRobinson159, SimpleBeat, Sơn Đỗ Đình Thy, Stephen Leppik, Thorax, Tommie C.
18	Digitare Casting	Anand Nimje, andyvn22, godisgood4, LopSae, Nick Podratz
19	dizionari	dasdom, Diogo Antunes, egor.zhdan, iOSDevCenter, Jason Bourne, Kirit Modi, Koushik, Magisch, Moritz, RamenChef, Saagar Jha, sasquatch, Suneet Tipirneni, That lazy iOS Guy , ThrowingSpoon
20	Enums	Alex Popov, Anh Pham, Avi, Caleb Kleveter, Diogo Antunes, Fantattitude, fredpi, Hamish, Jason Sturges, Jojodmo, jtbandes, juanjo, Justin Whitney, Matt, Matthew Seaman, Nathan Kellert, Nick Podratz, Nikolai Ruhe, SeanRobinson159, shannoga, user3480295
21	estensioni	Brduca, David, Esqarrouth, Jojodmo, jtbandes, Luca Angeletti, Moritz, rigdonmr
22	Funzione come cittadini di prima classe in Swift	Kumar Vivek Mitra
23	funzioni	Ajith R Nayak, Andy Ibanez, Caleb Kleveter, jtbandes, Kote, Luca Angeletti, Matt Le Fleur, Nikita Kurtin, noor, ntoonio, Saagar Jha, SKOOP, Stephen Schaub, ThrowingSpoon, tktsubota, ZGski
24	Funzioni di Swift Advance	DarkDust, Sagar Thummar
25	Genera UIImmagine delle iniziali dalla stringa	RubberDucky4444
26	Generics	Andrey Gordeev, DarkDust, FelixSFD, Glenn R. Fisher, Hamish, Jojodmo, Kent Liau, Luca D'Alberti, Suneet Tipirneni, Ven, xoudini
27	Gestione degli errori	Anil Varghese, cpimhoff, egor.zhdan, Jason Bourne, jtbandes, Mehul Sojitra, Moritz, Tom Magnusson
28	Gestione della memoria	Accepted Answer, Daniel Firsht, jtbandes, Marc Gravell, Moritz, Palle, Tricertops
29	Hashing crittografico	zaph
30	Il Defer Statement	Palle
31	Imposta	Community, Dalija Prasnikar, Luca Angeletti, Moritz, Steve Moser
32	Iniezione di dipendenza	Bear with me, JPetric
33	inizializzatori	Brduca, FelixSFD, rashfmnb, Santa Claus, Vinupriya Arivazhagan
34	Interruttore	Ajwhiteway, AK1, Duncan C, elprl, Harshal Bhavsar, joan, Josh Brown, Luca Angeletti, Moritz, Santa Claus, ThrowingSpoon

35	Introduzione alla programmazione orientata ai protocolli	Alessandro Orrù, Fred Faust, kabiroberai, Krzysztof Romanowski
36	Lavorare con C e Objective-C	4444, Accepted Answer, jtbandes, Mark
37	Le tuple	Accepted Answer, BaSha, Caleb Kleveter, JAL, Jason Sturges, Jojodmo, kabiroberai, LopSae, Luca Angeletti, Moritz, Nathan Kellert, Rick Pasveer, Ronald Martin, tktsubota
38	Lettura e scrittura JSON	Cyril Ivar Garcia, Ethan Kay, Glenn R. Fisher, Ian Rahman, infl3x, Jack C, Jason Sturges, jtbandes, Leo Dabus, lostAtSeaJoshua, Luca D'Alberti, maxkonovalov, Moritz, nstefan, Steffen D. Sommer, Stephen Leppik, toofani
39	Loops	Caleb Kleveter, D31, Efraim Weiss, Fred Faust, Hamish, Idan, Irfan, Jeff Lewis, Luca Angeletti, Moritz, Mr. Xcoder, Saagar Jha, Santa Claus, WMios, xoudini
40	Markup della documentazione	Abdul Yasin, Martin Delille, Moritz, Rashwan L
41	Memorizzazione nella cache dello spazio su disco	Viktor Gardart
42	Metodo Swizzling	JAL, Noam, Umberto Raimondi
43	Modelli di design - Strutturali	Ian Rahman
44	NSRegularExpression in Swift	Echelon, Hady Nourallah, ThrowingSpoon
45	Numeri	Arsen, jtbandes, Suragch, WMios, ZGski
46	Oggetti associati	Fattie, JAL
47	Operatori avanzati	avismara, egor.zhdan, Fluidity, Hamish, Intentss, JAL, jtbandes, kennytm, Matthew Seaman, orccrusher99, tharkay
48	OPTIONSET	4444, Alessandro
49	Opzionali	Anand Nimje, Andrey Gordeev, Arnaud, Caleb Kleveter, Hamish, Ian Rahman, iwillnot, Jason Sturges, Jojodmo, juanjo, Kevin, Michaël Azevedo, Moritz, Nathan Kellert, Paulw11, shannoga, SKOOP, Tanner, tktsubota, Tommie C.
50	Prestazione	Matthew Seaman
51	Programmazione funzionale in Swift	Echelon, Luca Angeletti, Luke, Matthew Seaman, Shijing Lv
52	protocolli	Accepted Answer, Ash Furrow, Cory Wilhite, Dalija Prasnikar, esthepiking, Hamish, iBelieve, Igor Bidiniuc, Jason Sturges, Jojodmo, jtbandes, Luca D'Alberti, Matt, matt.baranowski, Matthew Seaman, Oleg Danu, Rahul, SeanRobinson159, SKOOP, Tim Vermeulen, tktsubota, Undo, Victor Sigler
53	Puntatori di buffer	Tommie C.

	(non sicuri)	
54	Riflessione	Asdrubal, LopSae, Sajjon
55	RxSwift	Alexander Olferuk, FelixSFD, imagngames, Moritz, Victor Sigler
56	Structs	Accepted Answer, AK1, Diogo Antunes, fredpi, Josh Brown, Kevin, Luca Angeletti, Marcus Rossel, Moritz, pbush25, Rob Napier, SamG
57	Swift HTTP server di Kitura	Fangming Ning
58	Swift Package Manager	Moritz
59	Typealias	Bartłomiej Semańczyk, Caleb Kleveter, D4ttatraya, Moritz
60	Variabili e proprietà	Christopher Oezbek, FelixSFD, Jojodmo, Luke, Santa Claus, tktsubota