Introduzione a Swift: Optional

swift

Swift è il moderno linguaggio di programmazione sviluppato da Apple per *“coesistere”* con il tradizionale Objective-C.
Lo sviluppo del linguaggio è iniziato nel 2010 e la prima versione pubblica è stata presentata durante l’annuale *Worldwide Developers Conference* (WWDC) del 2014.
La motivazione che ha portato Apple alla decisione di sviluppare un linguaggio di programmazione completamente nuovo è essenzialmente la volontà di rendere disponibile alla community di sviluppatori uno strumento moderno che rispondesse alle mutate esigenze di sviluppo delle applicazioni mobile per *iOS*.
Intendiamoci, questo non significa che Objective-C sia morto, anzi, rimane vivo e vegeto.
Attualmente tutto quello che è possibile fare con Swift è realizzabile anche con Objective-C. Teniamo sempre presente che le fondamenta a basso livello di *Cocoa* e *Cocoa Touch* sono realizzati e mantenuti in Objective-C.
Quello che Apple ha fatto è stato prendere il meglio dell’esperienza dei linguaggi di programmazione e condensarla in Swift.
Dalla fine del 2015, Apple ha reso Swift open-source sotto licenza Apache 2.0 e ha portato il progetto sul proprio repository di GitHub.

funzionalità principali

Swift, oltre ad essere completamene orientato agli oggetti, è fortemente tipato e possiede al contempo un potente motore di *type-inference*.
Sin dalle origini supporta nativamente la *semantica di null*

per qualsiasi tipo.
Il programmatore può utilizzare le *closures* che in altri linguaggi sono note come *lambdas*.
La memoria è automaticamente gestita dal linguaggio mediante *reference-counting,* senza l'impiego della *garbage-collection*.
Il *multithreading* non è esplicito come in altri linguaggi: le computazioni *asincrone* sono demandate ad un gestore opaco con una modalità del tutto affine a quello che accade con GO.

interoperabilità

Sul fronte dell'interoperabilità, Swift è completamente integrabile con il codice scritto in Objective-C e C.
Per chi avesse necessità di integrare codice C++, Swift (alla versione attuale: 5.0), non supporta l'integrazione nativa con questo linguaggio, pertanto occorre necessariamente scrivere uno strato C di interfaccia verso il C++.

null semantic

Una delle caratteristiche di Swift più interessanti e potenti è sicuramente il supporto nativo alla *semantica di null*.
La semantica di null è un aspetto cruciale nel mondo dell'informatica: consente di determinare quando l'istanza di un tipo *non possiede un valore associato*.
A prima vista, qualcuno potrebbe pensare che si tratti di una cosa di poco conto: d'altronde quando mai una variabile non dovrebbe possedere un valore?
Sorprendentemente, o forse no, l'assenza di un valore è una proprietà estremamente comune nel nostro mondo fisico.
Facciamo un semplice esempio: supponiamo che abbiamo scritto un software che si connette a una miriade di sensori esterni: *temperatura*, *pressione*, *velocità del vento* e chi ne più ne metta.
Il nostro software deve semplicemente visualizzare i valori trasmessi dai vari sensori in una qualche GUI associata.
Che succede quando uno dei sensori si guasta e non trasmette più nulla alla nostra applicazione?

Che valore ci mettiamo nella variabile associata al sensore guasto?
Se il tipo fosse un intero con segno, ci mettiamo zero? Il minimo negativo degli interi?
E se fosse una stringa? Ci mettiamo “N/A”?
E se fosse un tipo ancora più complesso, magari una classe che contiene un centinaio di valori di tipi diversi?
C’è da dire che una qualche scappatoia potremmo anche trovarla, ma scegliere un valore tra quelli ammissibili per un tipo e dire: *“questo è il mio null”* non è una soluzione efficace anzi, è concettualmente errato.
Swift è stato progettato prevedendo che una variabile possa non avere un valore assciato e questo aspetto risulta pervasivo e coerente su tutto Cocoa.

i linguaggi storici

Se ci fermiamo a riflettere per un istante ai grandi linguaggi imperativi della tradizione informatica, come per esempio: C, C++ e Java, ci accorgiamo che per tutti loro la semantica di null è *deficitaria*.
Anzi, mi spingo ad affermare che per questi linguaggi, quando questa risulta possibile, lo è per *incidente*.
In che senso?
Nel senso che è solo per un caso *fortuito* che per questi linguaggi sia possibile, *a volte,* avere un’approssimazione del concetto di null.
Infatti, risulta possibile solo quando l’istanza di un tipo è allocata sullo *heap*.
In questo caso, poichè l’utente è in possesso del puntatore all’istanza di un tipo e dato che il puntatore può assumere il valore di null è possibile approssimare la semantica di null.

un esempio in java

```
package opt.test; //an user defined type
class Widget{ ... @override public String toString(){ ... }
} public class Main{ public static void main(String[] args){
Widget widget = null; //ok we can simulate null semantic with
```

```
classes if(widget != null){ System.out.println(widget); }else{
System.out.println("widget was null"); } int anInt = 21;
//ups.. how do we test if anInt is null..? }
}
```

Java, rispetto al C e al C++, risulta ancora più svantaggiato perchè non consente di allocare direttamente sullo *heap* i tipi primitivi.
In Java i tipi primitivi: *int*, *double*, *boolean*, *char,* etc sono allocabili esclusivamente sullo *stack*:

```
//it's ok to have primitive types on the stack: int
myStackInteger = 12;
double myStackDouble = 29.4; //but we cannot allocate an int
on the heap: int myHeapInteger = new int(12); //compile error!
```

Ne consegue che per i tipi primitivi, in Java, è impossibile la *non associazione* di un valore.
Lo stesso problema lo abbiamo anche con C e C++ quando un tipo è allocato sullo *stack*.
C e C++ consentono, a differenza di Java di allocare le *struct* e le *class* direttamente sullo *stack* come vediamo in questo esempio:

```
namespace opt_test{ class widget{
...
}; } int main(){ //we allocate a widget on the stack
opt_test::widget wdgt; //how do we test wdgt for null? sadly
we can not! //... }
```

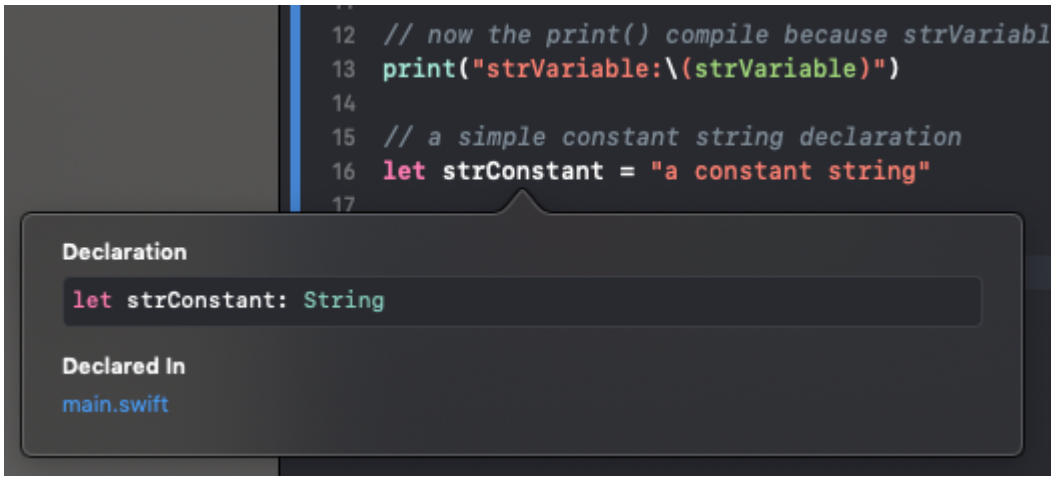
Per superare questa mancanza, lo standard 17 del C++ introduce std::optionalche è un tipo template che consente per i tipi allocati sullo *stack* di avere una semantica di null consistente.

variabili e costanti con swift

Prima di vedere nel dettaglio come Swift affronta la null semantic, diamo uno sguardo a come si dichiarano e si inizializzano le variabili e le costanti:

```
import Foundation // a simple variable string declaration without assigning a value var strVariable: String // the following print() does not compile because str is not initialized!
// print("strVariable:\(strVariable)") // now strVariable has a value
strVariable = "a string" // now the print() compile because strVariable has been initialized
print("strVariable:\(strVariable)") // a simple constant string declaration
let strConstant = "a constant string"
print("strConstant:\(strConstant)")
```

La prima cosa che notiamo è che Swift non ci obbliga a scrivere il codice all'interno di una funzione o di un metodo.
Questo risulta molto comodo quando vogliamo sperimentare qualcosa o quando dobbiamo scrivere un semplice *script*.
Ma veniamo al codice vero e proprio.
Notiamo che in Swift possiamo dichiarare una variabile con: var, le costanti invece, si indicano con: let.
Il tipo a differenza dei classici linguaggi *C-like* può essere indicato dopo il nome della variabile o costante preceduto da un *colon*.
Un'altra differenza, è che possiamo omettere il *semicolon* a fine riga, questo ovviamente se la riga contiene un solo *statement*, altrimenti dobbiamo usare il *semicolon* per separare gli *statement*.
Se ci fate caso, quando abbiamo dichiarato strConstantnon abbiamo indicato il tipo.
Questo perchè il tipo può essere automaticamente inferito da Swift quando è determinato dall'assegnamento contestuale alla dichiarazione.
Per conoscere il tipo inferito, con Xcode è sufficiente tenere premuto il tasto *option* e cliccare sull'elemento:



Il popup che si apre, ci mostra il tipo associato alla costante strConstant.
Le variabili e le costanti in questo esempio sono della stessa *“classe”* di quelle definibili sullo *stack* con il C, il C++ e il Java: non ammettono cioè, *l’assenza di un valore*.

optional in swift

Come indichiamo quindi, in Swift, la volontà di ammettere l’assenza di un valore per una variabile o una costante?
Usiamo *Optional*.
Prima di definire formalmente cosa sia Optional, vediamo come si usa:

```
//a string
var str : String = "a string" //an optional string
var optStr : String? = "an optional string"
```

Dichiarare una variabile *opzionale*, in Swift è estremamente semplice: si mette un punto interrogativo dopo il tipo.
Il tipo in questo caso non sarà più una *stringa* bensì, una *stringa opzionale*.
In questo caso stiamo dicendo che optStrpuò trovarsi in due stati distinti:

1. optStr == nil, nessun valore associato
2. optStrpossiede un valore associato

Come testiamo la presenza o meno di un valore per un tipo

opzionale?
Esistono almeno 2 modi:

```
var optStr: String? = "an optional string" //classic C, C++ and Java idiom to test if an object is null if optStr != nil{
print("optStr:\(optStr!)")
}else{ print("optStr: was nil!")
} //Swift recommended way if let unwrappedStr = optStr{
print("optStr:\(unwrappedStr)")
}else{ print("optStr: was nil!")
}
```

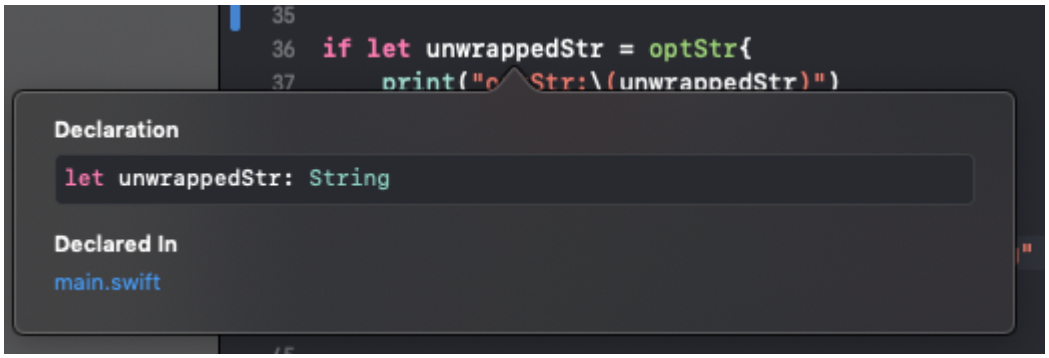
MODO CLASSICO

Il primo modo è il classico modo che abbiamo sempre usato nei linguaggi che ben conosciamo: testiamo cioè la variabile optStr, direttamente verso la parola chiave: nil.
A questo punto se non era nil, effettuiamo l'*unwrapping* di optStrmediante l'operatore postfisso: !.

Alla SWIFT

Il secondo modo è quello preferibile e raccomandato in Swift: introduciamo una costante temporanea: unwrappedStrche viene inizializzata con il *valore* di optStr*se questo esiste*.
Nel caso che optStrnon possieda un valore associato, il controllo passa al ramo else in cui unwrappedStrnon è più accedibile dal programma.
Il vantaggio rispetto alla prima modalità è che il programma è contemporaneamente più leggibile, pulito e sopratutto non cè modo di rischiare di effettuare l'*unwrapping* su di un *Optional* che non abbia un valore.
Effettuare l'*unwrapping* di un *Optional* che non ha un valore associato, giustamente, provoca il *crash* del programma.
Una piccola considerazione prima di andare avanti: qual'è il tipo di unwrappedStr?
Il costrutto if let identifier = optionaleffettua l'unwrapping di optional se questo ha un valore e quindi è lecito aspettarsi che identifier abbia il tipo di optional senza la nozione di *Optional*.

Se ci facciamo dire da Xcode il tipo, vediamo che la nostra intuizione era corretta:



Nil-Coalescing Operator

Supponiamo di avere definito tre stringhe opzionali:

```
var optStr1: String? = "first optional string"
var optStr2: String? = "second optional string"
var optStr3: String? = nil
```

Vogliamo adesso definire una stringa str *non opzionale* che contenga, *se esiste*, uno dei valori delle tre stringhe opzionali partendo da optStr1e proseguendo con le altre.
Se ci accorgiamo che nessuna delle tre stringhe opzionali contiene un valore, allora vogliamo impostare strcon un valore di default: *“default string”*.
Il modo più ovvio di farlo sarebbe il seguente:

```
var optStr1: String? = "first optional string"
var optStr2: String? = "second optional string"
var optStr3: String? = nil var str : String if optStr1 != nil
{ str = optStr1!
} else if optStr2 != nil { str = optStr2!
} else if optStr3 != nil { str = optStr3!
} else { str = "default string"
}
```

Questo codice è perfettamente valido, tuttavia è parecchio verboso e ci costinge ad usare molte volte il copia/incolla, pratica che introduce spesso errori subdoli.
Molto meglio usare il *nil-coaleshing operator*: ??

```
var optStr1: String? = "first optional string"
var optStr2: String? = "second optional string"
var optStr3: String? = nil var str = optStr1 ?? optStr2 ?? optStr3 ?? "default string"
```

Il codice è decisamente più pulito e facilmente comprensibile.
Il *nil-coaleshing operator* si può applicare a catena su un numero arbitrario di *Optional* ed effettua automaticamente l'*unwrapping* non appena trova un *Optional* che abbia un valore associato.
Un altro vantaggio di questo approccio è che in questo caso abbiamo potuto sfruttare la *type inference* di Swift senza dover dichiarare il tipo di str.
A questo proposito, vi propongo un piccolo quesito: se str fosse stata dichiarata in questo modo:

```
var optStr1: String? = nil
var optStr2: String? = nil
var optStr3: String? = "third optional string" var str = optStr1 ?? optStr2 ?? optStr3
```

Quale sarebbe stato in questo caso il tipo di str?

optional chaining

Supponiamo adesso di avere definito un tipo in questo modo:

```
struct A{ var field : B? struct B{ var field : C? struct C{ var field = "end of chain!" } }
}
```

Il tipo Aè un tipo strutturato, contiene al suo interno la definizione di un tipo B che a sua volta contiene la definizione di un tipo C.
Ad ogni livello di profondità troviamo un membro fieldche è un *Optional* del sottotipo definito a quel livello.
Il tipo C, poichè è una foglia, ha il membro field di tipo String.
Supponiamo di instanziare una variabile optA di tipo A?in questo modo:

```
var optA : A? = A()
optA!.field = A.B()
optA!.field!.field = A.B.C()
```

Adesso, vorremmo introdurre come prima, una variabile strche contenga il valore di A.B.C.fieldse riusciamo a raggiungerlo, altrimenti al solito, vorremmo impostare: *“default string”*.
Vediamo come possiamo farlo alla maniera *classica*.

```
struct A{ var field : B? struct B{ var field : C? struct C{ var field = "end of chain!" } }
} var optA : A? = A()
optA!.field = A.B()
optA!.field!.field = A.B.C() var str: String if optA != nil { if optA!.field != nil { if optA!.field!.field != nil { str = optA!.field!.field!.field } else { str = "default string" } } else { str = "default string" }
} else { str = "default string"
}
```

MODO CLASSICO

Questa volta il codice ci è venuto parecchio brutto ed è pure noioso e faticoso da scrivere.
Per nostra fortuna il tipo strutturato aveva solo tre livelli, però ci accorgiamo che se la catena fosse stata più lunga sarebbe stata un’agonia dover ripetere tutte le volte quegli else tutti uguali.
Fortunatamente Swift ci viene in soccorso con l’*optional chaining* con il quale possiamo scrivere la stessa cosa in questo modo:

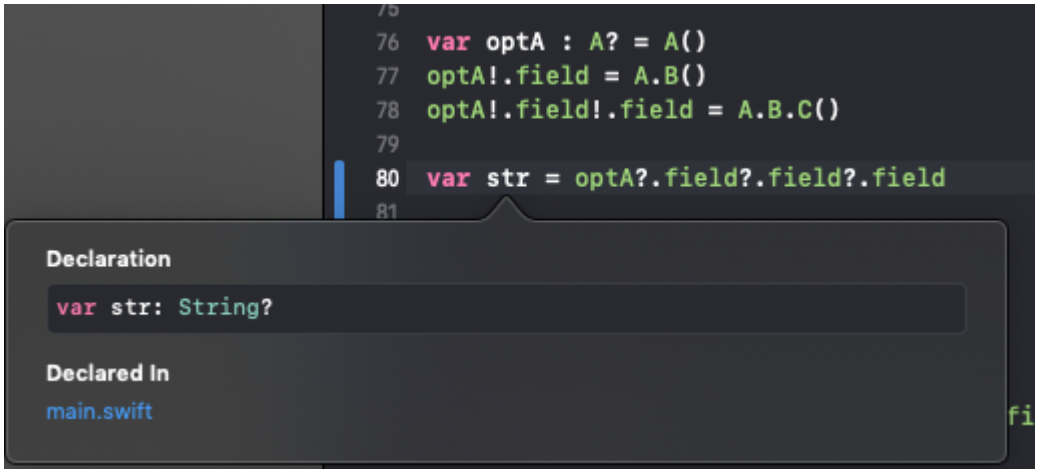
```
struct A{ var field : B? struct B{ var field : C? struct C{ var field = "end of chain!" } }
} var optA : A? = A()
optA!.field = A.B()
optA!.field!.field = A.B.C() //optional chaining + nil-coaleshing operator var str = optA?.field?.field?.field ?? "default string"
```

Come possiamo vedere, l’*optional chaining* ci permette di navigare agevolmente tutti i livelli *senza dover testare ogni*

livello.
Non appena l'*optional chaining* incontra un *Optional* == nil, la valutazione si interrompe immediatamente e viene restituito un *Optional settato a nil del tipo finale della catena*.
Quindi se a un certo punto, un field era nil, viene restituita una String? settata a nil.
Siccome abbiamo usato il *nil-coaleshing operator,* questo ci restituisce la nostra *"default string".*
Vi faccio il solito quesito di prima: quale sarebbe stato il tipo di str se avessimo omesso il *nil-coaleshing operator* finale?

```
//optional chaining, without final nil-coaleshing operator var str = optA?.field?.field?.field
```

Questa volta vediamo cosa ci dice Xcode:



Eh si, è proprio una String?e non poteva essere altrimenti, se l'*optional chaining* incontra un field == nilnon può far altro che valutare il tipo del field finale (di C) e restituire un *Optional* di quel tipo settato a nil.
Questo tipo è quindi quello che *battezza* str.

formalizziamo optional

Ora che ci siamo fatti un'idea di cosa sia *Optional* e di come si usa in Swift, proviamo a formalizzare cosa *è*.

- *Optional* è un tipo *generico* (*template*) come tanti altri

- *Optional* è un’*enumerazione* che possiede due stati: *none* e *some(<T>)*

Il tipo *Optional* in Swift è definito come segue:

enum Optional<T> { case none case some(<T>)
}

Quindi un enum, nè più nè meno, ma con un sacco di sintassi aggiuntiva e operatori specializzati.
Ovviamente, essendo un tipo così importante in Swift, Apple ha deciso di dedicargli un bel po’ di cura per facilitarne l’uso.
Ecco come potremmo dichiarare la variabile optStr usando la sintassi concisa ed esplicita:

enum Optional<T> { case none case some(<T>)
} var optStr: String? var optStr: Optional<String> = .none
var optStr: String? = “hello” var optStr: Optional<String> = .some(“hello”)
var optStr: String? = nil var optStr: Optional<String> = .none

CONCLUSIONI

Siamo alla fine, ci sarebbero altri aspetti di *Optional* da coprire, ma direi che i concetti base li abbiamo affrontati.
Spero di avervi trasmesso l’importanza che *Optional* riveste in Swift.
Chiunque voglia cimentarsi nella realizzazione di App per *iOS* o *macOS* con Swift deve avere ben chiara quale sia la semantica di *Optional* e del perchè sia così importante e pervasiva.

Grazie a tutti e alla prossima!

.

.

https://italiancoders.it/introduzione-a-swift-optional/