

Sorprende come Apple sia riuscita a mantenere segreto un progetto di queste proporzioni per gli anni necessari alla creazione di un compilatore. Il responsabile del progetto, Chris Lattner, nella sua homepage (<http://nondot.org/sabre/>) afferma infatti di avere iniziato il lavoro su Swift nel 2010, poco più di quattro anni fa.

Cominciamo a dare uno sguardo di insieme a Swift, iniziando con un esempio

```
let base = 3
let power = 10
var answer = 1
for _ in 1...power {
    answer *= base
}
```

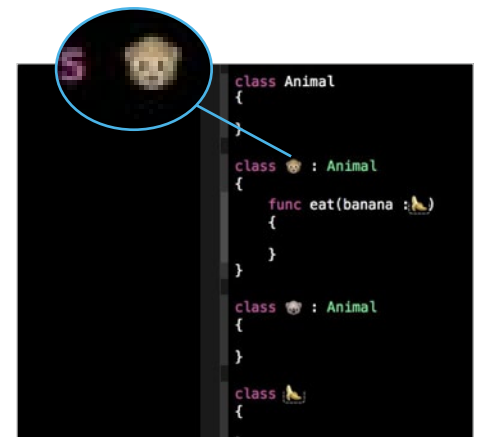


**Come il C
ma senza C**

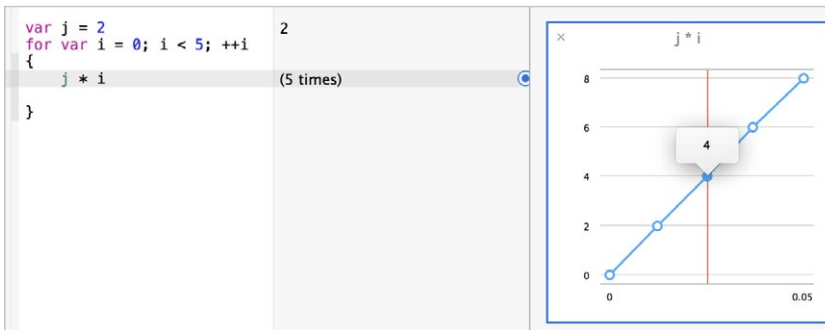
Il linguaggio si appoggia
al tradizione framework
Cocoa di OS X e iOS

La misura della novità sta in un tweet del giornalista americano John Gruber (@gruber) durante la conferenza: «Le persone con un badge Apple sono in silenzio, gli sviluppatori non impazziti. È una notizia che si tratta del futuro di tutto il mondo Apple». I più accreditati giornalisti writer di area Apple considerano di aspettare a preannunciare l'event-C e buttare via i vecchi programmi di programmazione.

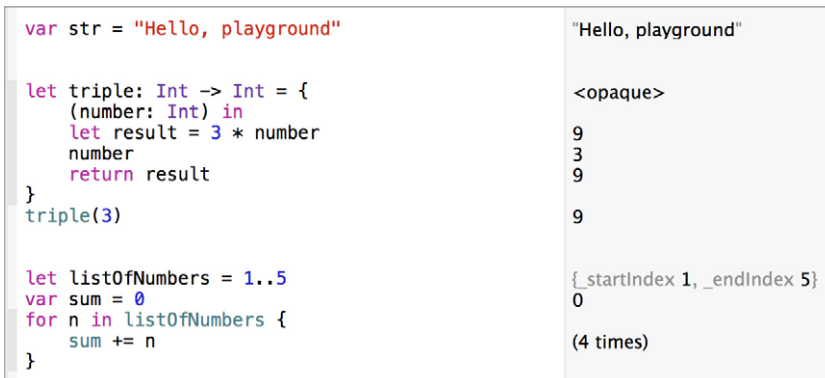
Il nostro prossimo progetto iOS o OS X potrà essere in Swift, ma per anni a venire il framework con cui interagiremo sarà scritto in Objective-C, così come nel linguaggio tradizionale saranno scritti per qualche tempo i



Il supporto completo per la codifica Unicode, consente l'uso di variabili in qualche altro alfabeto, o anche l'uso di simboli grafici come identificatori



I playground di XCode che arrivano insieme a Swift sono uno strumento didattico efficace in grado di mostrare l'andamento di un programma nel corso del tempo



Si possono visualizzare annotazioni al codice che ne rendono evidente il funzionamento.

```

}
println("\(base) to the power of \
(power) is \(answer)")
// prints "3 to the power of 10 is
59049"

```

Si tratta di un modo rozzo di calcolare la decima potenza di tre, ma queste poche righe si colgono in azione particolarità interessanti del linguaggio.

La prima cosa che salta all'occhio è la parola chiave `let` per l'assegnazione, una keyword che è capace di dare qualche brivido ai reduci dei primi Basic. Come vedremo è la spia di una funzione avanzata. Notiamo l'assenza dei punti e virgola come separatori fra le istruzioni. I punti e virgola sono opzionali, ma ridondanti, come in JavaScript, e si possono omettere del tutto, salvo nel caso che si accumulino più istruzioni sulla stessa riga. Un'altra reminiscenza del linguaggio dei browser è l'istruzione `var` nella terza riga. La compresenza di `let` e `var` per la definizione di variabili, discende da un aspetto fondamentale del linguaggio, l'immutabilità di alcune variabili. `let` si usa per creare variabili che possono essere assegnate una volta sola, mentre `var` si usa per variabili che possono mutare di valore nel corso

dell'esecuzione. Si tratta di un impegno che il compilatore ci farà rispettare, avvisandoci se trovasse un'istruzione come `power = 5` perché è un errore tentare di alterare il valore di una costante. Il ciclo che segue mostra un altro paio di dettagli interessanti. L'indice del ciclo è un underscore, questo esprime che non ci interessa usare il valore dell'indice all'interno del loop, vogliamo solo eseguire il ciclo un certo numero di volte. Specificando un underscore, usiamo una variabile anonima. In secondo luogo, `for _ in` specifica un ciclo in un range di valori. L'espressione `1...power` denota un intervallo di valori e specifica anche che l'intervallo è chiuso a destra, perché abbiamo scritto tre puntini, non due.

Il linguaggio è compatibile unicode: possiamo usare caratteri internazionali nel nome degli identificatori, come in questo esempio

```

let π = 3.14159
let 你好 = "你好世界"

```

Come tutti i linguaggi moderni, non mancano collezioni di vario tipo. Ecco, per esempio, come si crea e inizializza un dizionario

```

var airports = ["TYO": "Tokyo",
"DUB": "Dublin"]
println("Il dizionario degli
aeroporti contiene \(airports.
count) elementi.")
// stampa " Il dizionario degli
aeroporti contiene 2 elementi."

```

Un dizionario, ovviamente, è mutabile, quindi possiamo aggiungere un elemento

```

airports["LHR"] = "London"
// ora gli elementi sono tre

```

E possiamo cambiare il valore associato a una chiave

```

airports["LHR"] = "London Heathrow"

```

Nell'esempio successivo mostriamo il modo naturale per gestire una variabile non assegnata in Swift, un `if` sull'assegnazione:

```

if let airportName =
airports["DUB"] {
    println("Il nome dell'aeroporto
con codice DUB è \(airportName).")
} else {
    println("Non abbiamo il codice
DUB nell'elenco.")
}

```

Esaminiamo il codice Swift che serve per aggiornare un elemento nel dizionario ricercando con chiave e conservando il valore precedente, se ne esiste uno.

```

if let oldValue = airports.
updateValue("Dublin International",
forKey: "DUB") {
    println("Il valore precedente
di DUB era \(oldValue).")
}

```

Si nota subito la somiglianza con questo frammento di Objective-C

```

[defaultAttributes setValue:font
forKey:NSFontAttributeName];

```

La somiglianza non è casuale: il framework alle spalle di Swift è quello di Objective-C e la sintassi del nuovo linguaggio esalta la somiglianza. È quello che intendeva comunicare la frase «Swift è Objective-C senza il C».

Consideriamo il codice che cambia il valore della stringa descrittiva dell'aeroporto di Dublino:

```
airports.updateValue("Dublin
International", forKey: "DUB")
```

Come abbiamo visto, *airports*, è un dizionario. Il metodo `updateValue` del dizionario è quello che aggiorna il valore associato a una chiave, se questo esiste. Il primo parametro è la stringa da associare, il secondo è la stringa "DUB" preceduta dal token `forKey:` che consente di specificare un nome per un parametro, in modo da rendere più chiara l'intenzione del codice rispetto all'uso di parametri posizionali. Avevamo già visto questa funzionalità in altri linguaggi, da Python a C#. Ecco un esempio di codice in cui usiamo un parametro per nome in una funzione che unisce due stringhe con uno spazio o una stringa specificata dal chiamante

```
func join(s1: String, s2: String,
joiner: String = " ") -> String {
    return s1 + joiner + s2
}
```

si noti che il parametro *joiner* ha un valore di default, quindi se non è specificato prenderà il valore di uno spazio. Possiamo anche dare un valore al parametro, chiamandolo per nome, cioè specificando *joiner: " "*. Non sarebbe indispensabile per togliere ambiguità, dato che il valore è passato nella terza posizione, quella corretta. Quando ci sono più parametri opzionali, invece, specificare il nome può diventare importante.

```
join("hello", "world", joiner: "
")
// restituisce "hello, world"
```

Swift permette di dare un nome esterno al parametro. Possiamo cioè mantenere il nome *joiner* nel corpo della funzione e consentire *withJoiner* nell'invocazione della funzione, come segue

```
func join(string s1: String,
toString s2: String,
withJoiner joiner: String = "
") -> String {
    return s1 + joiner + s2
}
```

```
join(string: "hello", toString:
"world", withJoiner: "-")
// restituisce "hello-world"
```

```
join(string: "hello", toString:
```

USERHASLOST IN SWIFT

```
func userHasLost() -> Bool {
    if !gameboardFull() {
// Gameboard must be full for the user
to lose
        return false
    }
}
```

```
func tileBelowHasSameValue(loc:
(Int, Int), value: Int) -> Bool {
    let (x, y) = loc
    if y == dimension-1 {
        return false
    }
    switch gameboard[x, y+1] {
    case let .Tile(v):
        return v == value
    default:
        return false
    }
}
```

```
func
tileToRightHasSameValue(loc: (Int,
Int), value: Int) -> Bool {
    let (x, y) = loc
    if x == dimension-1 {
        return false
    }
    switch gameboard[x+1, y] {
    case let .Tile(v):
        return v == value
    default:
        return false
    }
}
```

```
// Run through all the tiles
and check for possible moves
for i in 0..dimension {
    for j in 0..dimension {
        switch gameboard[i, j] {
        case .Empty:
            assert(false, "Gameboard
reported itself as full, but we
still found an empty tile. This is
a logic error.")
            case let .Tile(v):
                if
tileBelowHasSameValue((i, j), v)
|| tileToRightHasSameValue((i, j),
v) {
                    return false
                }
            }
        }
    }
}
return true
}
```

USERHASLOST IN OBJC

```
- (BOOL)userHasLost {
    for (NSInteger i=0; i<[self.gameState
count]; i++) {
        if (((F3HTileModel *) self.
gameState[i]).empty) {
// Gameboard must be full for the user to lose
            return NO;
        }
    }
    // This is a stupid algorithm, but
given how small the game board is it
should work just fine
    // Every tile compares its value to
that of the tiles to the right and below
(if possible)
    for (NSInteger i=0; i<self.dimension;
i++) {
        for (NSInteger j=0; j<self.
dimension; j++) {
            F3HTileModel *tile =
[self tileForIndexPath:[NSIndexPath
indexPathForRow:i inSection:j]];
            if (j != (self.dimension - 1)
&& tile.value ==
[self tileForIndexPath:[NSIndexPath
indexPathForRow:i inSection:j+1]].value) {
                return NO;
            }
            if (i != (self.dimension - 1)
&& tile.value ==
[self tileForIndexPath:[NSIndexPath
indexPathForRow:i+1 inSection:j]].value) {
                return NO;
            }
        }
    }
    return YES;
}
```

Cosa cambia e cosa resta uguale nel passaggio da Objective-C a Swift si può intuire confrontando l'aspetto di queste due porzioni di codice dalla versione in Objective-C e quella in Swift del gioco 2048. La funzione è quella che decide se l'utente ha perso.



```
"world")
// restituisce "hello world"
```

QUALCOSA DI SPECIALE

Swift ha le caratteristiche di un linguaggio moderno. Per esempio la sintassi snella, che sta a metà fra linguaggi tradizionali e i linguaggi di scripting, grazie punti e virgola opzionali (come in JavaScript) e grazie a un meccanismo automatico di inferenza dei tipi, che permette di omettere quasi ovunque una dichiarazione specifica nella definizione delle variabili. Swift ha i *generics* e strutture dati ricche e espressive. Insomma, una discreta eredità da linguaggi funzionali come OcaML. Ci sono variabili immutabili, funzioni e altre caratteristiche dei linguaggi funzionali che hanno dato eccellente prova di sé nella programmazione concorrente. Al di là di trovare somiglianze, però, domandiamoci dove stanno quelle due o tre caratteristiche che danno personalità a un linguaggio.

Al primo posto, secondo noi, sta Arc, *Automatic Reference Counting*, l'infrastruttura che usa da qualche tempo Objective-C per tenere traccia delle allocazioni di memoria e rilasciare automaticamente le variabili quando hanno finito il loro ciclo di vita. Si tratta di una caratteristica importante, su cui il team che ha sviluppato Objective-C ha puntato quando ancora nessuno aveva ancora sentito parlare di Swift. Si tratta di una scelta orgogliosa e coraggiosa, ora che tutti scelgono la strada della garbage collection, e che mette Swift su un piano diverso dagli altri linguaggi. L'allocazione di memoria precisa e sicura senza il carico della garbage collection significa che Swift è adatto per applicazioni in tempo reale. Un'altra caratteristica che emerge, sono le variabili *optional*. Swift consente di esprimere in modo chiaro al compilatore e a chi legge il codice dopo di noi, che alcune variabili possono essere eventualmente non inizializzate, anche dove vengono usate.

Facciamo un esempio

```
var optionalString: String? =
"Hello"
optionalString == nil
```

Le variabili *optional* permettono l'uso dell'*if* nell'assegnazione, un modo espressivo di subordinare il codice al successo nel leggere il valore di una variabile, che permette eventualmente di specificare cosa fare se la variabile è *nil*

```
var optionalName: String? =
"Michele Costabile"
var greeting = "Hello!"
if let name = optionalName {
    greeting = "Hello, \(name)"
}
```

Un altro elemento sintattico caratteristico è l'*optional chaining*, l'indicazione della possibilità di trovare un puntatore non inizializzato in una catena di dereferenziazioni. Per spiegarci, ecco un esempio

```
if let cardNumber = john.
bankAccount?.creditCard?.number {
    println("il numero di
    carta di credito di John è \
    (cardNumber).")
} else {
    println("John non ha un conto
    o, se lo ha, non ha una carta di
    credito.")
}
```

Richiediamo il numero di carta di credito associato al conto corrente di un utente, senza prenderci impegni sul fatto che il conto esista e, se esiste, che ci sia associata una carta di credito. Si tratta di una caratteristica che ci piace un sacco, perché mette al primo posto l'espressione dell'intenzione del codice, mettendo fra le righe i necessari controlli di congruenza, che sono inseriti automaticamente dal compilatore. In questo modo, possiamo scrivere codice che non rischia di generare un errore per un puntatore nullo, e conservare l'espressività che viene dal mettere al primo posto, nella riga uno, l'intenzione di navigare da un oggetto utente, attraverso un conto bancario e una carta di credito, fino al numero di carta. Un plus che arriva insieme a Swift sono i *Playground*, una visualizzazione offerta da XCode del codice in azione, che permette, per esempio, di seguire l'evoluzione di una variabile nel corso di un ciclo di elaborazione, navigando avanti e indietro nell'esecuzione del programma, sia riga per riga, sia nella timeline. Si tratta di uno strumento didattico eccellente, ottimo per insegnare a programmare o per vedere in pratica l'azione di costrutti poco familiari.

EVOLUZIONE FUTURA

Non c'è ragione di aver paura che Objective-C scompaia improvvisamente: il framework applicativo di Apple resta interamente in Objective-C, così

come enormi quantità di codice pubblico in rete. Il vecchio linguaggio resta ancora l'unico strumento per importare codice C++ in un progetto e avere accesso a tutto le sorgenti libere, general purpose, che si trovano sul web. In una parola, il codice e le competenze Objective-C resteranno importanti per parecchi anni a venire e il linguaggio non scomparirà senza preavviso, come i frame in acciaio spazzolato. Per contro, è indiscutibile che i giorni di Objective-C siano contati. Swift non nasce per affiancarsi al vecchio linguaggio, ma per sostituirlo e, prima o poi, lo farà. Quando sarà finito il ciclo di beta test, sarà opportuno iniziare a sperimentare con componenti nel nuovo linguaggio anche in progetti esistenti, dato che la convivenza è garantita.

Chi si accosta alla programmazione iOS adesso, troverà opportuno prendere le cose dalla parte del nuovo. Swift è più coeso e coerente di Objective-C. Purtroppo, non sarà possibile ignorare totalmente il linguaggio in cui è scritto il corpo del sistema e delle sue librerie applicative, ma è più intuitivo prendere questa strada che fare il giro nel verso opposto.

Swift non assomiglia molto a Objective-C, ma le sue interfacce con il framework sono comunque progettate in modo da calarsi nel modo più naturale su Cocoa, quindi le similitudini sono ben visibili. Per fare un esempio, ecco un frammento Objective-C

```
UITableView *myTableView
= [[UITableView alloc]
initWithFrame:CGRectZero
style:UITableViewStyleGrouped];
```

ed ecco come si mappa in Swift:

```
let myTableView: UITableView =
UITableView(frame: CGRectZero,
style: .Grouped)
```

L'architettura di Cocoa è stata un invitato importante al tavolo di progetto di Swift. Secondo alcuni questo è un difetto architetturale, perché la progettazione dovrebbe partire da principi primi e cercare l'innovazione, come è avvenuto con il linguaggio Go.

Noi la vediamo diversamente: il framework ha avuto la libertà di evolvere e maturare in modo autonomo dal linguaggio e la sua organizzazione può portare ordine, più che limitare la creatività. •