1-Introduzione a Objective-C

Objective-C

Ojective-C è il linguaggio di programmaizone principale che si utilizza per sviluppare software per OS X e iOS, è un super-insime rigido del linguaggio C, che fornisce una programmazione object-oriented ed un'aggiunta di funzionalità dinamiche. Questo linguaggio permette di:

- compilare ogni programma scritto in C, mediante il compilatero di Objective-C
- permette di utilizzare le dichiarazioni del linguiaggio C
- permette di utilizzare codice C++, però è necessario istruire il compilatore correttamente per processare il codice C++

> Note
> Objective-C eredita la sintassi, tipi primitivi, ecc.. dal C ed aggiunge la sintassi per definire classi e metodi.

> Nota
> La sintassi che non è relativa all'object-oriented è la stessa del linguaggio C

Swift

Da iOS8, l'Apple ha introdotto un nuovo linguaggio di programmazione, chimato swift, con una sintassi coincisa ed espressiva. Swift introduce dei pattern di programmazione sicuri e non permette più la compatibilità con il linguaggio C; in oltre può lavorare affiancato ad Objective-C, che può essere utilizzato per estendere le applicazioni esistenti.

> Nota
> Swift è disponibile da XCode 6

```
var square = 0
var diceRoll = 0
while square < finalSquare {
        // roll the dice
        if ++diceRoll == 7 {
        diceRoll = 1
        }
        // move by the rolled amount
        square += diceRoll
        if square < board.count {
                square += board[square]
        }
}
println("Game over!")
```

La programmazione Object Oriented

La programmazione object- oriented, OOP, è un paradigma che mira a creare programmai che sono:

- altamente modulari
- ben strutturati
- semplici da mantenere
- semplici da espandere
- semplici da debuggare
- semplici da testare

> Nota
> Un'altro obbiettivo dell'OOP è di garantire anche la riutilizzabilità del codice

Un programma OOP è composto da un insieme graph) di oggetti che interagiscono tra loro, che rappresentato concetti relativi ad uno scenario specifico(domain). La rappresentazione di un oggetto è una struttura dati che è composta dall'aggregazione di:

- stato, ovvero i dati attuali su cui l'oggetto sta operando, ovvero gli attributi
- comportamento, ovvero le funzioni che permettono di poerare con l'oggetto e i suoi dati, ovvero i metodi

> Nota
> Un oggetto può interagire con un altro oggetto invocando uno dei suoi metodi.

> Nota
> Un oggetto ha delle respnsabilità uniche, quindi vine considerato come indipendente, seguendo il modelle *black-box*(scatola nera)

Classi e oggetti

Le classi sono dei progetti/prototipi/"stampi" per gli oggetti, gli oggetti invece sono delle istanze di una classe. Una classe definisce la struttura e il comportamento che tutte le istanze di quella classe condividono; deve anche essere specializzata per garantire alcuni comportamenti, quali ad esempio:

- rappresentare i dati
- controllare l'interfaccia utente
- eseguire interfacce di rete
- interagire con i database
- ecc...

I principi dell'OOP

Per essere eggettivo, l'OOP definisce tre principi che devono essere seguiti:

- Eredità
- Incapsulamento
- Polimorfismo

Eredità

L'eredità è un principio fondamentale dell'OOP, questa permette di definire nuove funzionalità di una classe *estendendola*. Il processo di estensione viene chiamato sottoclassificazione (subclassing).

Tutte le classi definiscono una gerarchia, che definisce quali funzionalità ha ogni istanza della classe.

> Nota
> La classe che viene estesa si chiama superclasse

> Nota
> Una classe eredità tutte le variabili e i metodi definiti dalla sua superclasse, quindi tutto ciò che può fare una superclasse può farlo anche una sua sottoclasse

> Nota
> L'eredità permette di creare una relazione secondo la semantica "is-a"

Incapsulamento

L'incapsulamento è un principio fondamentale dell'OOP. Incapsulamento significa che dati e metodi per operare sui dati sono incorporati all'interno di uno stesso componente (classe). In accordo con i principi di incapsulamento solo le interfacce possono essere accessibili e visibili all'esterno dell'oggetto, mentre le implementazioni devono essere nascoste.

L'incapsulamento permette di trattare gli oggetti come una black-box, che provvede alcune funzionalità ma nasconde i dettagli delle operazioni. Possono essere definiti 3 livelli di visibilità:

- `private`
- `public`
- `protected`

Polimorfismo

Il polimorfismo è il terzo principio OOP, sapendo che con l'ereditarietà tutte le sottoclassi ereditano lo stesso metodo della superclasse è possibile per le sottoclassi sovrascrivere questi metodi.

> Nota
> Si può dire che questo principio è l'abilità di adattare il comportamento di tipi concreti

I programmi Objective-C

I programmi Objective-C sono sivisi in 2 tipi principali di file:

- I file `.h`, che definiscono l'interfaccia. L'interfaccia definisce quali classi, strutture dati e metodi sono esposte nelle altre parti di codice
- I file `.m`, che implementano ciò che le corrispondendi interfacce definiscono
- I file `.mm`, che implementano i file che contengono codice C++

> Nota
> Si dice che l'implementazione è opaca, ovvero che sono le interfacce possono essere accedute dalle parti del codice

Definire una classe

Le classi sono definite all'interno dei file interfaccia `.h`, tra la coppia di tag `@interface` e `@end`. Tutto ciò che si trova all'interno del blocco dell'interfaccia definisce la struttura della classe e le sue sunzionalità.

Esempio

Definire una classe `MDPoi` che è utilizzata per memorizzare un punto d'interesse, con nome, latitudine e longitudine

```
#import <Foundation/Foundation.h>

@interface MDPoi : NSObject{
        NSString *_name;
        double _latitude;
        double _longitude;
}
- (void)setLatitude:(double)lat;
@end
```

Il codice sopra presenta:

- Un importazione di un header di un altra classe `#import <Foundation/Foundation.h>`
- La direttiva di dichiarazione di una classe `@interface MDPoi`
- L'estensione da una classe `: NSObject`
- L'istanziazione di variabili, che deve avvenire tra le parentesi graffe

```
{
        NSString *_name;
        double _latitude;
        double _longitude;
}
```

- La definizione di un metodo, che deve avvenire all'esterno delle parentesi graffe, `- (void)setLatitude:(double)lat;`

Implementare una classe

L'implemetazione delle classi avviene all'interno dei file `.m`

Esempio

Implementare la classe `MDPoi` definita nell'interfaccia `MDPoi.h`, tra la coppia di tag `@implementarion` e `@end`

```
#import "MDPoi.h"
@implementation MDPoi
- (void)setLatitude:(double)lat{
        _latitude = lat;
}
@end
```

Il codice sopra presenta:

- L'importazione dell'interfaccia del file header `#import "MDPoi.h"`
- La direttiva di implementazione di una classe `@implementation MDPoi`
- L'implementazione di un metodo dichiarato

```
- (void)setLatitude:(double)lat{
        _latitude = lat;
}
```

> Nota
> La mancata implementazione di un metodo causa 'solo' un errore a runtime (non a runtime genera solo un warning)

Keyword speciali

In Ojective-C, tutti gli oggetti sono allocati all'interno del heap, quindi l'accesso a quasti avviene sempre utilizzando un puntatore ad oggetto. Detto questo ci sono delle keyword speciali che bisogna sapere:

- `id` che indica un puntatore ad un oggetto di qualsiasi tipo (simile al `void*` del C)
- `nil` è il valore ti dun puntatore che sta puntando a nulla (`NULL` del C)
- `BOOL` è un tipo definito (`typedef` nel file `objc.h`) da Ojective-c per i valori booleani:
  - `YES` (1 o vero)
  - `NO` (0 o falso)
- `self` che indica un puntatore all'oggetto stesso (simile al `this` di java)

La comunicazione tra gli oggetti

In Ojective-C gli oggetti comunicano tra di loro mandandosi dei messaggi. L'invocazione di un metodo si basa sullo scambio di messaggi, questo metodo d'invocazione è diverso dalle classiche chiamate, perché il metodo che deve essere eseguito non è limitato ad una sezione specifica del codice, ma il destinatario del messaggio è risolto dinamicamente a runtime (o nella maggior parte dei casi a compile-time). Potrebbe accadere anche che il ricevente potrebbe non rispondere al messaggio, in questo caso verrà lanciata un eccezione.

Esempio

```
[poi log];
[request serURL:_URL];
```

La struttura di un messaggio è `[oggetto bersaglio, nome del messaggio (metodo) : argomenti del metodo]`

I metodi

Prendendo l'esempio

```
- (NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)poi;
```

- Metodi d'istanza, relativi ad un particolare oggetto, deve iniziare con il simbolo del meno `-`, questi possono accedere alle variabili d'istanza
- Metodi di classe, relativi alla classe stessa, questi iniziano con il simbolo del più `+`, questi sono accessibili mediante il nome della classe (non sono necessari per un'instanza), non possono accedere alle variabili di istanza; un esempio di metodi di classe sono i metodì di allocazione e i metodi di servizio (utility)

La struttura di un metodo è:

- Tipo di ritorno tra le parentesi tonde `(NSArray *)`
- Prima parte del nome del metodo `pointsWithinRange:`
- Seconda parte del nome del metodo `fromPoi`

> Nota
> Il nome compelto del metodo è `pointsWithinRange:fromPoi:`

- Primo argomento `(double)distance`
- Secondo argomento `(MDPoi *)poi`

È importante che il nome dei metodi in Ojective-C devono "suonare" come frasi del linguaggio naturale, per questo motivo alcuni metodi hanno dei nomi molto lunghi, per riuscire a mantenere il codice leggibile.

Esempio

```
- (BOOL)tableView:(UITableView *)tableView
       canPerformAction:(SEL)action
       forRowAtIndexPath:(NSIndexPath *)indexPath
       withSender:(id)sender;
```

L'invocazione dei metodi

Suppondendo un metodo definito come segue

```
- (NSArray *)pointsWithinRange:(double)distance fromPoi:(MDPoi *)obj;
```

un esmpio di invocazione è

```
NSArray *list = [self pointsWithinRange:10.0f fromPoi:poi];
```

Le variabili d'istanza

Di default le variabili d'istanza sono protette `@proteceted`, oltre a questo è possibile dividere le variabili in blocchi:

- `@private`, le variabili contenute possono essere accedute solo dall'interno dall'interno della classe stessa
- `@protected`, le variabili possono essere accedute all'interno della classe stessa e all'interno delle sue sottoclassi
- `@public`, le variabili possono essere accedute da ogni parte del codice (non è buono nella pratica dell OOP)

> Nota
> Una buona pratica nell'OOP è di mettere tutte le variabili d'istanza private ed utilizzare dei metodi getter e setter per accedere ad esse

Esempio

`MDPoi.h`

```
#import <Foundation/Foundation.h>
@interface MDPoi : NSObject{
       @private // variabili d'istanza private
       NSString *_name;
       double _latitude;
       double _longitude;
}
- (double)latitude; // metodo getter
- (void)setLatitude:(double)lat; // metodo setter
- (double)longitude;
- (void)setLongitude:(double)lon;
@end
```

`MDPoi.m`

```
@implementation MDPoi
- (double) latitude{
        return _latitude;
}
- (void) setLatitude:(double)lat{
        _latitude = lat;
}
- (double) longitude{
        return _longitude;
}
- (void) setLongitude:(double)lon{
        _longitude = lon;
}
@end
```

Per accedere alle variabili di istrìanza con i getter e i setter:

```
// prende il valore
double lat = [poi latitude];

// imposta il valore
[poi setLatitude:12.0f];
```

Proprità (propertis)

Propertis sono un modo conveniente per impostare/leggere il valore di una variabile d'istanza utilizzando la nodazione punto (dot notation). Il vantaggio principale è di evitare l'utilizzo eccessivo di parentesi quadre all'interno del codice, rendendo così più leggibile la concatenazione di invocazione di metodi.

> Nota
> È molto importante fare attenzione ai caratteri maiuscoli e ai caratteri minuscoli, il compilatore capisce la dot-notation solo se il nome dei getter e dei setter è scritto correttamente

```
// prende il valore
double lat = poi.latitude;

// imposta il valore
poi.latitude = 12.0f;
```

Le proprietà generate dal compilatore

È possibile definire tutti i getter e tutti i setter a mano, però il compilatore sarà d'aiuto per fare ciò; la keyword `@property` istruisce il compilatore di generare le definizioni di getter e setter in automatico:

Esempio

```
- (double)latitude;
- (void)setLatitude:(double)lat;
```

Diventa

```
@property double latitude;
```

Se si vuole solo il metodo getter bisogna aggiungere la keyword `readonly`.

```
@property(readonly) double latitude;
```

Le properties possono essere definite senza un istanza di variabile:

`file.h`

```
@property (readonly) NSString *detail;
```

`file.m`

```
- (NSString *)detail{
        return [NSString stringWithFormat:@"%@ - (%f,%f)", _name, _latitude, _longitude];
```

```
}
```

`utilizzo`

```
NSString *str = poi.detail;
```

È possibile inoltre far generare l'implementazione (synthesize) le property utilizzando la direttiva `@synthesize`

> Nota
> L'utilizzo di `@synthesize` non impedisce l'implementazione delle direttive di property

Binding dinamico

Come detto in precedenza Objectiv-C si basa sullo scambio di messaggi per l'invocazione di un metodo, e la risoluzione del bersaglio del messaggi avviene a tempo di esecuzione. Specificando il tipo di un oggetto quando viene scritto il codice non aiuta il compilatore a svolgere qualsiasi tipo di controllo, semplicemente può aiutare ad evidenziare possibili bug.

> Nota
> Se un oggetto riceve un messaggio che a cuoi non può rispondere l'applicazione andrà in crash

> Nota
> È possibile forzare l'esecuzione di un metodo sfruttando i casting pointers, però bisogna fare molta atenzione

Quindi è importante essere sempre sicuri di mandare un messaggio a un oggetto che può rispondere.

Introspezione (introspection)

L'introspezione eprmette di scoprire a tempo di eseuzione se un oggetto può o non può rispondere ad un messaggio. Questo è particolarmente utile quando tutti possono prendere un oggetto di tipo `id`.
Come funziona? Ogni sottoclasse di `NSObject` ha i seguenti metodi:

- `isKindOfClass`, che può essere usato per controllare se il tipo dell'oggetto coincide con la classe o con una sua sottoclasse
- `isMemberOfClass`, che può essere suato per controllare se il tipo dell'oggetto coincide con quello della classe

> Nota
> `isKindOfClass`, `isMemberOfClass`, prendono un oggetto `Classe` come argomento; un oggetto classe può essere ottenuto utilizzando il metodo `class` che può essere invocato su ogni tipo di classe di Objective-C.

Esempio

```
if([obj isKindOfClass:[NSString class]]){
        NSString *str = (NSString *)obj;
        ...
}
```

- `respondsToSelector`, che può essere usato per controllare se il tipo dell'oggetto può rispondere a un particolare selettore

> Nota
> Un selettore (selector) è un tipo speciale che indica il nome di un metodo;
> `respondsToSelector` prende un selettore (`SEL`) come argomento e può essere ottenuto con la keyword `@selector`

Esempio

```
if([obj respondsToSelector:@selector(lowercaseString)]){
        NSString *str = [obj lowercaseString];
        ...
}
```

> Nota
> È possibile chiedere ad un oggetto di eseguire un selettore

```
SEL selector = @selector(lowercaseString);
if([obj respondsToSelector:selector]){
        NSString * str = [obj performSelector:selector];
        ...
}
```

Foundation framework

I foundation framework definiscono un layer di base di classi di Objective-C, fornendo un set di classi primitive molto utili ed introducendo alcuni paradigmi che definiscono funzionalità non coperte dal linguagigo stesso. In questo modo rende lo sviluppo software più semplice e permette un livello di indipendenza del sistema operativo per semplificare la portabilità.

NSObject

#todo_slide_52-64

Creazione di oggetti

La procedura per la creazione di un oggetto in Objective-C richiede 2 passi:

1. L'allocazione della memoria
2. L'instanziazione e l'inizializzazione dell'oggetto

Il metodo `alloc` chiede al sistema operativo di riservare un area di memoria per l'oggetto.

> Nota
> Il metodo `alloc` è ereticato da tutte le classi che estendono la classe `NSObject`

> Nota
> Il metodo `alloc` si occupa anche di impostare tulle le variabili di istanza a `nil`

Inizializzazione degli oggetti

Il metodo `alloc` ritorna un oggetto allocato con la variabile d'istanza impostata a `nil`. Le variabili saranno impostate successivamente utilizzando un apposito metodo `initializer`. Il nome di in inizializzatore inizia sempre con `init`.
Ogni classe ha un "inizializzatore designato" (`designated initializer`) che è responsabile di impostare tutte le variabili d'istanza in modo che l'oggetto di ritorno è in uno stato consistente.

Ogni inizializzatore deve prima invocare l'inizializzatore designato della super classe e controllare che quest'ultimo non abbia ritornato un oggetto nullo (`nil` object). Un inizializzatore può prendere degli argomenti, solitamente quelli che servono per impostare in modo appropriato le variabili d'instanza.

> Nota
> È possibile avere più di un inizializzatore, però ognuno di essi devono comunque chiamare quello designato.

Esempio

`MDPoi.h`

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude;
```

`MDPoi.m`

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
        if(self = [super init]){
                _name = [name copy]; _latitude = latitude; _longitude = longitude;
        }
        return self;
}
```

Nell'implementazione della classe la chiamata all'inizializzatore designato della superclasse avviene `[super init]`

Creazione di un oggetto

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
```

> Nota
> Le versioni di iOS6 e precedenti, richiedono che un inizializzatore ritorni un `id`; da iOS7 in poi è stata introdotta una keyword speciale `instancetype` per dire che il metodo ritorna un oggetto che è dello stesso tipo dell'oggetto che ha mandato il messaggio

> Nota
> Ci sono molti altri metodi, diversi da alloc/init, che permettono di ottenere un oggeto.
> *Esempio*
> `NSString *str = [words objectAtIndex:0];`

Gestione della memoria

> Ricorda
> In Ojective-C tutti gli oggetti sono allcati nello heap, quindi per accedere ad essi bisogna usare un puntatore all'oggeto

Ci sono due aree in cui la memoria è gestita:

- stack, è una area di memoria dove i dati sono memorizzati secondo una polica lifo; lo stack è dove le variabili locali (variabili della funzione corrente) sono memorizzate.
- heap, è un area di memoria dove la memoria è allocata dinamicamente

Ogni volta che un oggetto è inizializzato, è necessario allocare un quantitativo di memoria appropriato nell'heap e ritornare un puntatore ad esso. Quando un oggetto non è più necessario, bisogna liberare la memoria allocata per quest'ultimo mediante la `free(pointer)`.

Inizialmente Objective-C non presentava nulla simile al garbage collector di Java, ma successivamente è stata implementata una tecnica chiamata "reference counting", che significa:

1. Tenere conto del numero di volte che un oggetto viene puntato
2. Quando si fa riferimento all'oggetto il contatore viene incrementato
3. Quando si ha terminato con l'oggetto il contatore viene decrementato
4. Quando il contatore va a 0 la memoria è liberata, viene invocatoil metodo `dealloc` (ereditato da `NSOject`)

> Nota
> Accedere a un oggetto liberato causa il crash dell'applicazione

Manual Reference Counting

`NSObject` definisce 2 metodi per incrementare e decrementare il reference count:

- `retain`(trattieni), incrementa il contatore di 1
- `release`(rilascia), decrementa il contatore di 1

Un oggetto quando deve essere utilizzato viene trattenuto finchè è necessario, impedento la sua distruzione. Quando non è più necessario viene rilasciato, quindi il reference count decresce e se raggiunge lo 0 la memoria viene liberata.

> Nota
> Un oggetto ottenuto mediante malloc/init ha il reference count pari a 1

> Nota
> È possibile ottenere il reference count corrente di un oggetto mediante il metodo `retainCount` ereditato da `NSObject`

Esempio

```
MDPoi *poi = [[MDPoi alloc] initWithName:@"MyPoi" latitude:45.2 longitude:10.12];
NSLog(@"retain count = %d", [poi retainCount]); // 1

[poi retain];
NSLog(@"retain count = %d", [poi retainCount]); // 2

[poi retain];
NSLog(@"retain count = %d", [poi retainCount]); // 3

[poi release];
NSLog(@"retain count = %d", [poi retainCount]); // 2

[poi release];
```

```
NSLog(@"retain count = %d", [poi retainCount]); // 1

[poi release]; // 0 e free
```

Il possesso di un oggetto (oject ownership)

Il modello della gestione della memoria si basa su il possesso di oggetto; un oggetto può avere 1 o più proprietari, finché l'oggetto ha almeno un proprietario continua ad esistere.

Le regole per il possesso di un oggetto:

1. Si possiede ogni oggetto creato (mediante `alloc/init`)
2. Si può prendere possesso di un oggetto utilizzando `retain`
3. Quando l'oggetto non è più necessario bisogna rinunciarci utilizzando `release`
4. Non si deve rinunciare ad un oggetto che non si possiede

> Nota
> Esiste una keyword sostituibile a `release` per indicare che l'oggetto verrà rilasciato successivamente (ovvero dopo che verrà ritornato)

Deallocazione della memoria

Quando un reference count raggiunge lo 0, il metodo `dealloc`, (se non implementato) ereditato da `NSObject` viene invocato.

> Nota importante
> Il metodo `dealloc` non deve mai essere chiamato direttamente

> Nota importante
> Il metodo `dealloc` deve essere reimplementato se nella classe sono utilizzati dei tipi non primitivi e l'ultima istruzione deve essere `[super dealloc]`

Esempio

```
- (void) dealloc{
        [_name release];
        [super dealloc];
}
```

Gestione della memoria con le propertis

Tipicamente

- I getters ritornano un oggetto che andà un tempo di vita abbastanza lungo da essere trattenuto (simile a `autorelease`)
- I setters trattengono l'oggetto che deve essere impostato

Il problema però si presenta quando le properties sono `@synthesized`, quindi ci sono 3 possibili opsioni per dire come i setters dovrebbero essere implementati:

- `@property (retain) NSString *name:`

```
- (void) setName:(NSString *)name{
        [_name release];
        _name = [name retain];
}
```

- `@property (copy) NSString *name;`

```
- (void) setName:(NSString *)name{
        [_name release];
        _name = [name copy];
}
```

- `@property (assign) NSString *name;`

```
- (void) setName:(NSString *)name{
        _name = name;
}
```

> Nota
> Una buona pratica è quella di non utilizzare metodi accessori nell'inizializzazione, perché i metodi richiedono di operare in base allo stato dell'oggetto, ma quando questo viene inizializzato non si ha nulla di garantito sullo stato.

Esempio di errore

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
        if(self = [super init]){
                self.name = name;
                self.latitude = latitude;
                self.longitude = longitude;
        }
        return self;
}
```

Esempio corretto

```
- (id) initWithName:(NSString *)name latitude:(double)latitude longitude:(double)longitude{
        if(self = [super init]){
                self.name = [name copy];
                self.latitude = latitude;
                self.longitude = longitude;
        }
        return self;
}
```

Lo stesso ragionamento vale per la fase di deallocazione, perché dato che l'oggetto deve essere deallocato non è ne sicuro ne appropriato mandare un messaggio.

Esempio corretto

```
- (void) dealloc{
        [_name release];
        [super dealloc];
}
```

Automatic Reference Counting

Da iOS 4, è stata introdotta la Automatic Reference Counting (ARC), che delega il compito di contare le referenze al compilatore. Quindi non è più necessario invocare esplicitamente `retain`, `release` e `autorelease`, non è più necessario implementare un `dealloc` personalizzato, e se non vengono utilizzati dei tipi non primitivi non bisogna scrivere alla fine del metodo `[super dealloc]` perché ARC lo fa automaticamente.

Esempi

`MRC`

```
- (NSArray *) getAllPois{
        NSMutableArray *array = [[NSMutableArray alloc] init];
        ...
        [array autorelease];
        return array;
}
```

`ARC`

```
- (NSArray *) getAllPois{
        NSMutableArray *array = [[NSMutableArray alloc] init];
        ...
        return array;
}
```

ARC e le properties

L'ARC introduce il concetto di "weak references". Questa non estende il lifetime di un oggetto a cui si sta puntando che automaticamente diventa `nil` se nessuna "strong reference" lo sta puntando. Per risolvere questo problema sono state introdotte due keyword `weak` e `strong`

Esempio

```
        @property (strong) MyClass *myObject; // corrisponde a @property (retain) MyClass
*myObject;
```

```
        @property (weak) MyClass *myObject; // corrisponde a @property (assign) MyClass
*myObject;
```

Protocolli

Objective-C provvede un modo per definire un insieme di metodi simili alle interfacce di Java, questi sono i protocolli.
I protocolli definiscono il comportamento di una classe (i metodi che devono essere presenti), indipendentemente dal tipo di classe.

> Nota
> I protocolli definiscono i contratti dei messaggi

> Nota
> L'utilizzo di protocolli permettono di massimizzare la riutalizzabilità del codice, minimizzando le interdipendenze tra le parti del codice

Definire un protocollo

Un protocollo è definito nel file interfaccia tra la direttiva `@protocol` e il corrispondente `@end`. Nella definizione è possibile dichiarare se alcuni metodi sono obbligatori (default), `@required` o opzionali, `@optional`

```
@protocol MyProtocol
        // definition of methods and properties
        - (void)method1;
        - (NSString *)method2;
        @optional
        - (void) optionalMethod;
        @required
        - (NSArray *)method3:(NSString *)str;
@end
```

Conformità ai protocolli

Le classi che implementano i metodi, definiti da un protocollo, che sono richiesti (`@required`) sono detti conformi al protocollo ("conform to the protocol").

> Nota
> Se una classe deve diventare conforme ad un protocollo deve implementare i metodi richiesti definiti nel protocollo

Conformandosi ad un protocollo una classe è in grado di essere usata fornendo un comportamento concreto per una parte del programma, che dipende da questo comportamento, ma che non dipende dalla particoare implementazione di quest'ultimo.

Esempi

- `UITableViewDataSource` è un protocollo che definisce metodi da cui dipende la `UITableView`, la cui implementazione è indipendente dai dati da mostrare.
- `UITableView` usa un `UITableViewDataSource` per "andare a prendere" i dati che saranno mostrati.

> Nota
> Una classe conforme alla `UITableViewDataSource` sarà passata alla `UITableView` durante la fase di esecuzione.

Definizione di una classe conforme a un protocollo

Una classe conforme ad un protocollo deve avere:

1. Il protocollo dichiarato tra le parentesi angolate
2. L'implementazione dei metodi richiesti dal protocollo
3. Se un metodo è dichiarato opzionale è importante controllare a tempo di esecuzione se l'oggetto target lo ha implementato prima di mandargli un messaggio

```
if([target respondsToSelector:@selector(optionalMethod)]){
        // ...
        [target optionalMethod];
        // ...
}
```

4.

Esempio

`file.h`

```
@protocol MyProtocol
        - (void)method1;
        - (NSString *)method2;
        @optional
        - (void)optionalMethod;
        @required
        - (NSArray *)method3:(NSString *)str;
@end

@interface MyClass : NSObject<MyProtocol>{ // 1
        //...
}
@end
```

`file.m`

```
@implementation MyClass
- (void)method1{ // 2
        //...
}
- (NSString *)method2{ // 2
        //...
}
- (NSArray *)method3:(NSString *)str{ // 2
        //...
}

@end
```

Categorie (categories)

Alcune volte, quando si vorrebbe estendere una classe esistente aggiungendo alcuni comportamenti utili solo in alcune situazioni, si protrebbe pensare all'utilizzo di sottoclassi, ma questo potrebbe essere una soluzione difficile da apprendere.
Objective-C fonrisce una soluzione per gestire queste situazioni in modo semplice e pulito mediante le categories. Queste aggiungono metodi a classi esistenti senza utilizzare delle sottoclassi.

> *Esempio di utilizzo*
> Se fosse necessario aggiungere dei metodi ad una classe esistente, così da aggiungere delle funzionalità per rendere più semplice qualcosa nell'applicazione.

Definire una category che migliora una classe

Per deginire una category per una classe bisogna:

1. "Ridefinire" la classe con la direttiva `@interface`
2. Specificare il nome della category tra le parentesi
3. Dichiarare il nuovo metodo
4. Fornire un implementazione per la category nel file `.m` utilizzando la direttiva `@implementation`

> Nota
> C'è una convezione per nominare i file:
>
> - `<BaseClass>+<Category>.h`
> - `<BaseClass>+<Category>.m`

Esempio

`NSString+MyCategory.h`

```
@interface NSString (MyCategory)
        - (int) countOccurrences:(char)c;
@end
```

`NSString+MyCategory.m`

```
@implementation NSString (MyCategory)
- (int) countOccurrences:(char)c{
        // ...
}
@end
```

`utilizzo`

```
#import "NSString+MyCategory.h"
// ...
int occurA = [self.name countOccurrences:'a'];
// ...
```

Un trucco con le categories

Sarebbe bello avere dei metodi che si utilizzano nella propria classe privatamente senza esporli, per fare ciò bisogna definire una category anonima nel file di implementazione e dichiarare i propri metodi prevati al suo interno.

Esempio

`file.m`

```
@interface MDPoi ()
        @property (strong) NSString *name;
        - (void) privateMethod;
@end
@implementation MDPoi
        @synthesize name = _name;
        - (void) privateMethod{
                // ...
        }
@end
```