

# iOS Lab for Dev

Developing Apps for iPhone and iPad



## Lesson I

# The team

- Enrico Zeffiro  
*CTO @ H-umus*  
*ezeffiro@h-umus.it*
- Alessandro Benvenuti  
*Software developer @ H-umus*  
*abenvenuti@h-umus.it*
- Alberto De Bortoli  
*iOS developer @ H-umus*  
*adebortoli@h-umus.it*



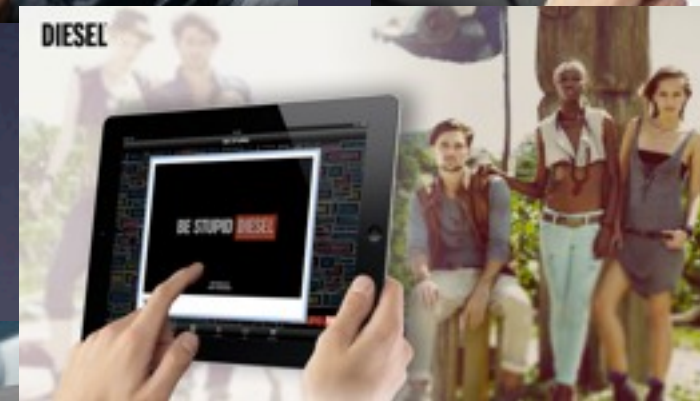
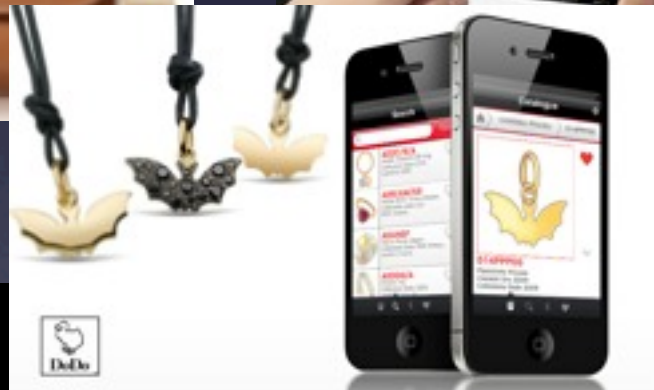
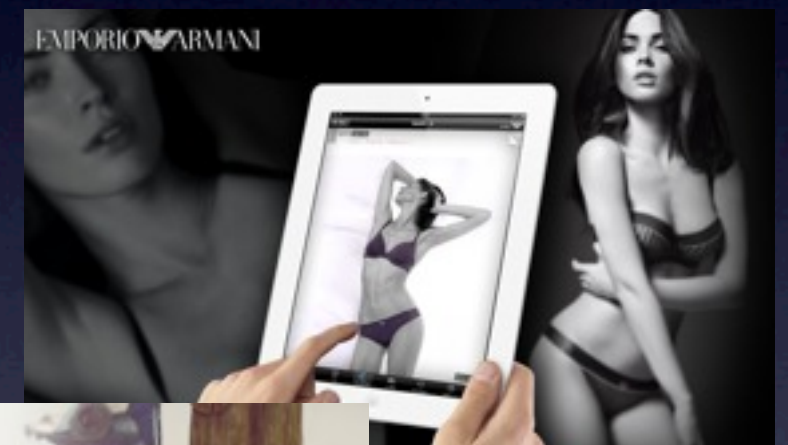
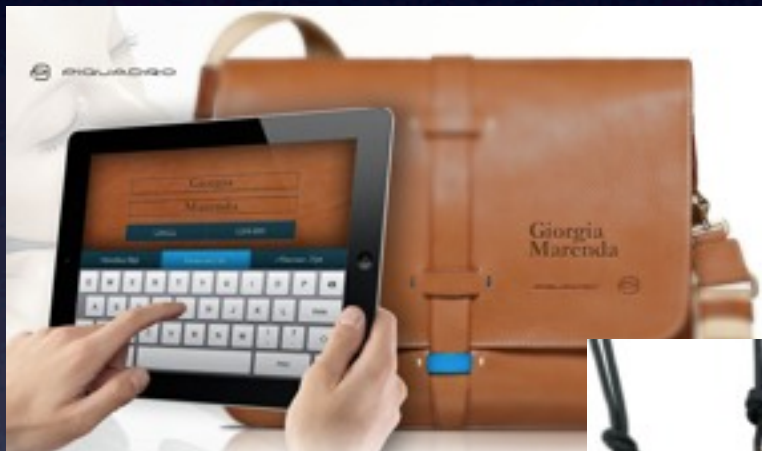


# H-humus

is a company growing in



- Cool iPad apps for the most important fashion brands out there



# Info

- Sabato 20/10, 27/10, 10/11, 17/11, 10.00 - 16.00
- Materiale: <https://github.com/albertodebortoli/iOSLab-DigitalAccademia-2012>

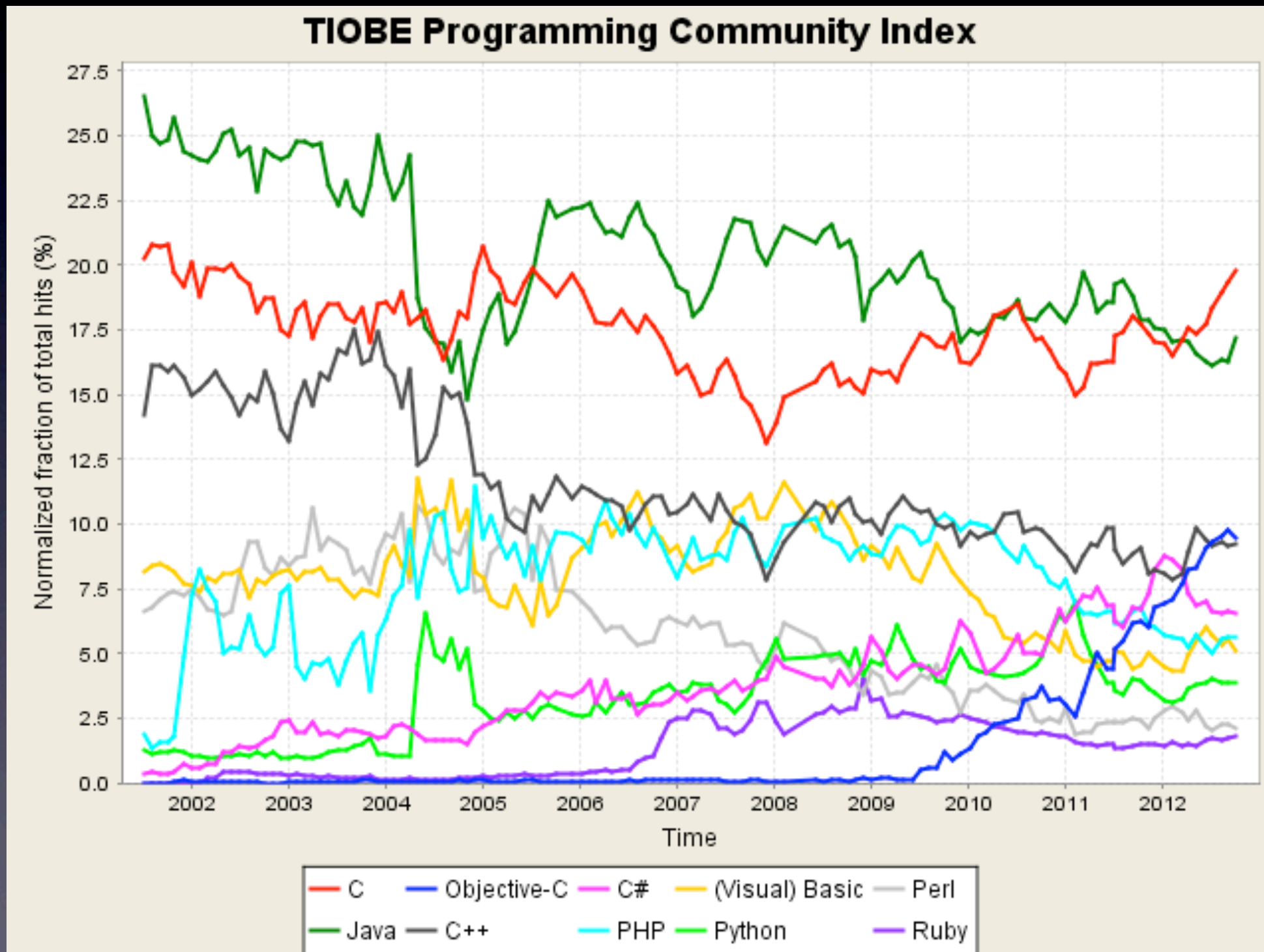


# Topics of the day

1. Objective-C basics
2. Cocoa UIViewControllers
3. Memory Management
4. Delegation
5. UITableViews

# Objective-C basics

# Objective-C basics





# Objective-C basics

A differenza del C++, in Objective-C i metodi non vengono “chiamati”, ma vengono inviati dei messaggi all’oggetto che espone il metodo.  
La sintassi del C++ prevede che un metodo venga chiamato nel seguente modo:

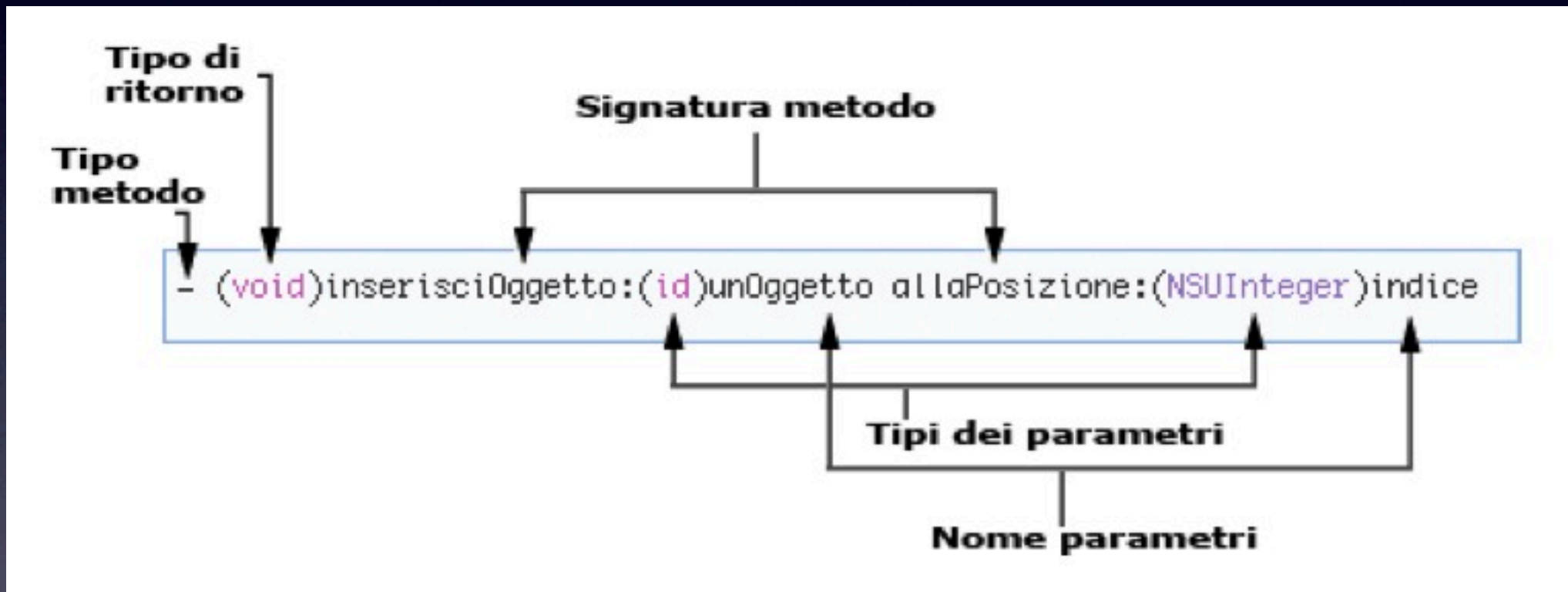
```
oggetto.metodo(parametro1, parametro2, ...)
```

La sintassi di Objective-C prevede che il messaggio venga inviato così:

```
[oggetto metodoConPrimoParametro:parametro1  
eSecondoParametro:parametro2]
```



# Objective-C basics



# Objective-C basics

Cocoa, il framework Apple mette a disposizione classi base:

- NSString
- NSNumber
- NSArray
- NSDictionary
- ...

# Objective-C basics

La creazione di oggetti avviene inviando prima il metodo per allocare l'area di memoria, poi per inizializzare l'oggetto.

```
[[Classe alloc] init]
```

## Esempi:

```
NSString *aString = [[NSString alloc] initWithString:@"Testo"]
```

```
NSArray *aArray = [[NSArray alloc] initWithObject:aString];
```

Documentazione Apple su Objective-C:

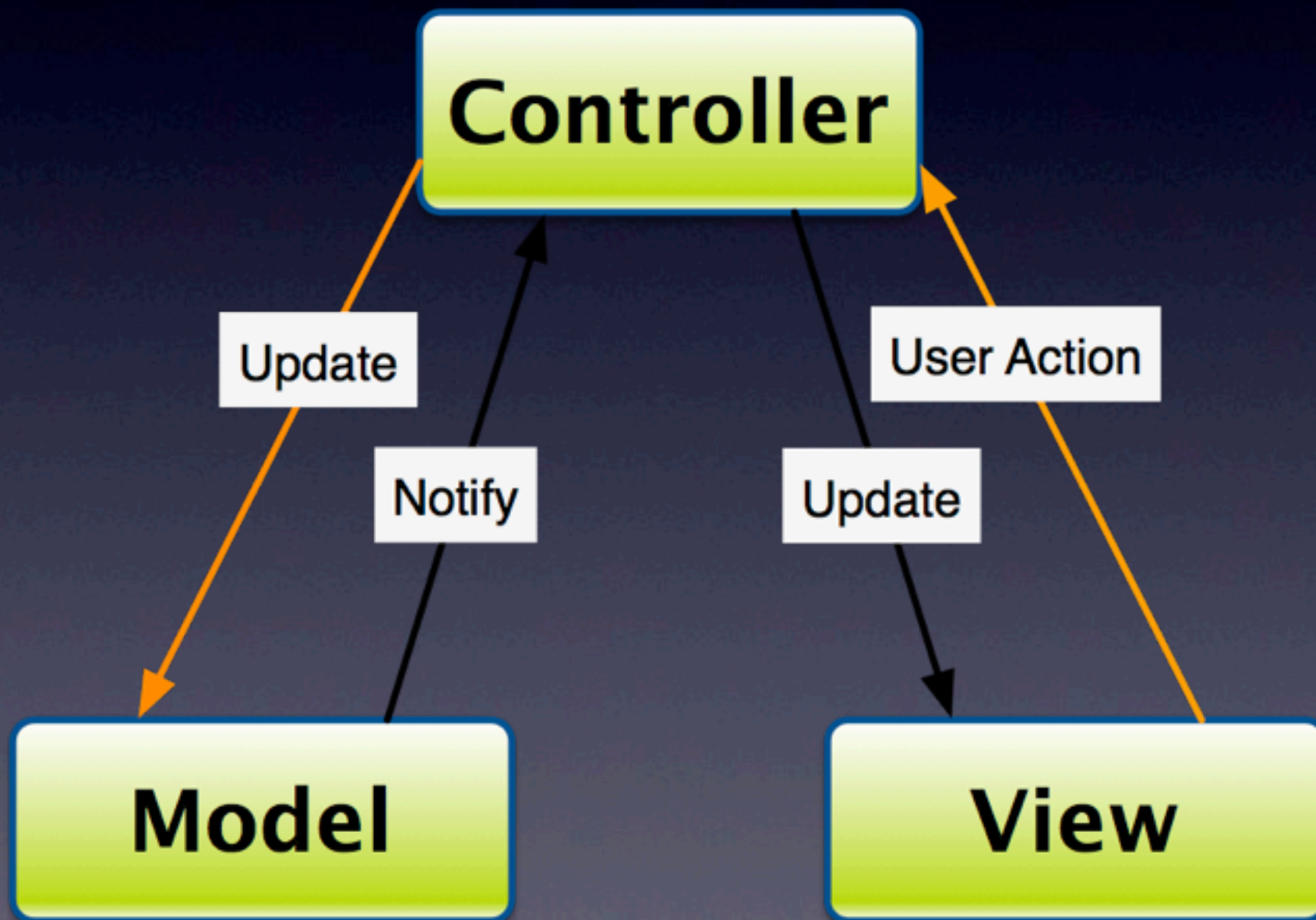
<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf>



# Cocoa UITableViewController

# MVC

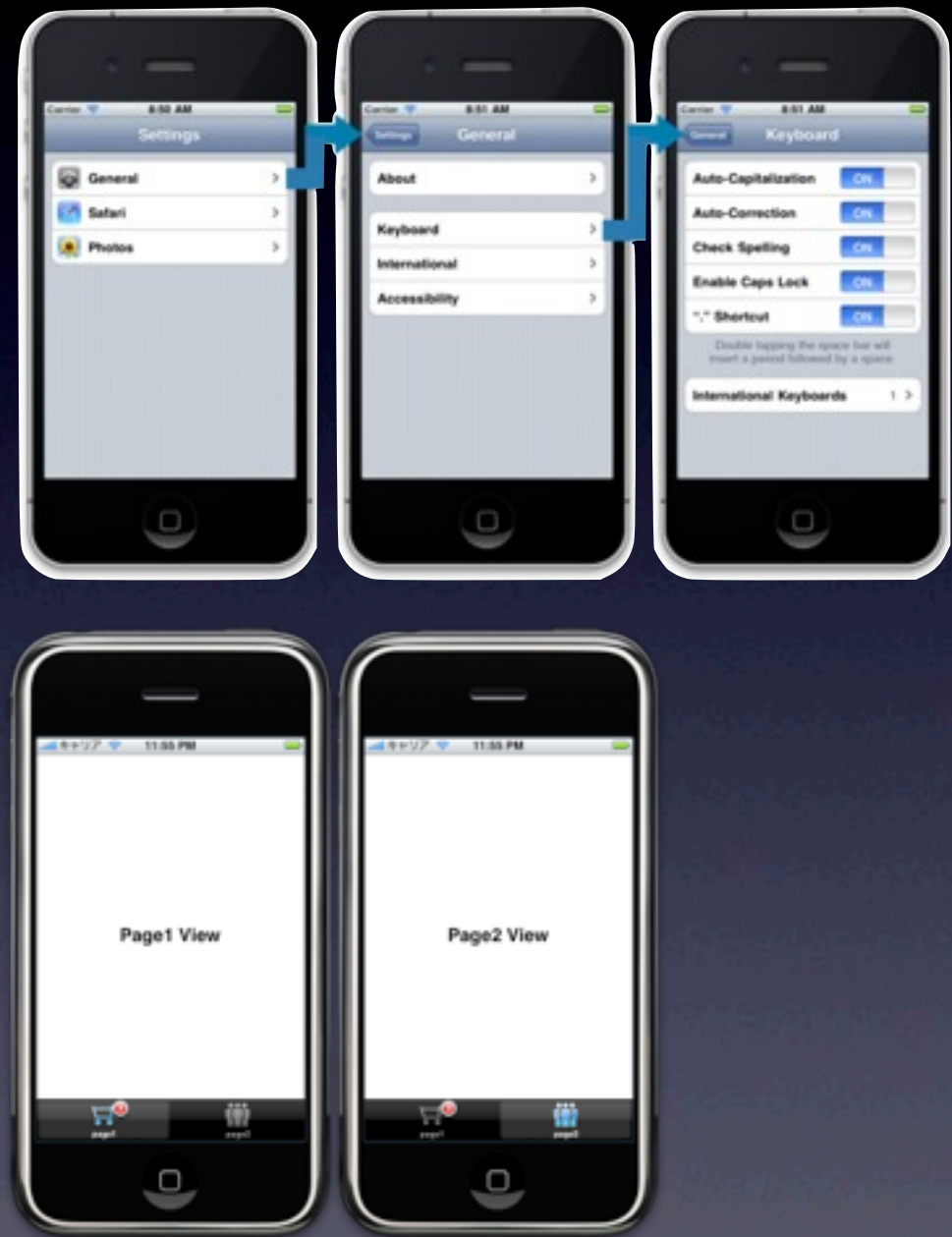
(Model-View-Controller)



# Cocoa UIViewControllers

## ViewController comuni

1. UIViewController (presenta una vista)
2. UINavigationController (presenta dei view controller sfruttando un meccanismo a pila)
3. UITabBarController (mostra dei view controller organizzandoli per sezioni)



[http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UIViewController\\_Class/Reference/Reference.html#//apple\\_ref/occ/cl/UIViewController](http://developer.apple.com/library/ios/#DOCUMENTATION/UIKit/Reference/UIViewController_Class/Reference/Reference.html#//apple_ref/occ/cl/UIViewController)



# Cocoa UIViewControllers

## **Metodi tipici di UIViewController sui quali fare override**

- 1.- `(void)viewDidLoad;`
- 2.- `(void)viewWillAppear:(BOOL)animated;`
- 3.- `(void)viewDidAppear:(BOOL)animated;`
- 4.- `(void)viewWillDisappear:(BOOL)animated;`
- 5.- `(void)viewDidDisappear:(BOOL)animated;`

Tali metodi vengono chiamati nell'ordine indicato durante il ciclo di vita del view controller;

- `(void)viewDidUnload:(BOOL)animated;`

Viene chiamato in caso di memory warnings.

cfr. esercizio allegato

# Memory Management

# Objective-C

## Memory Management

Nel creare un'applicazione è importantissimo ricordare che:

- la memoria del nostro hardware è limitata e non va sprecata!
- ogni oggetto occupa un'area di memoria ed era nostra responsabilità preoccuparci di liberarla, nel momento nel quale l'oggetto cessava d'esistere.
- in Cocoa la gestione della memoria veniva effettuata manualmente per evitare problemi di performance (a differenza di Java ad esempio).
- in Cocoa Touch non è presente nessun meccanismo di Garbage Collection
- dall'introduzione di ARC (Automatic Reference Counting) la gestione della memoria è automatizzata.



# Objective-C

## Memory Management

Nel gestire la memoria possiamo andare incontro a 2 tipo di problemi:

- **Viene liberata o sovrascritta un'area di memoria ancora utilizzata.**

Può portare a crash o alla corruzione di dati.

- **Non viene liberata un'area di memoria quando non più necessaria**

Causa memory leaks che portano rallentamenti dell'app e appesantimento.

# Objective-C

## Memory Management

Ogni oggetto ha la responsabilità di assicurarsi che gli oggetti ai quali riferisce rimangano in vita e, nel momento nel quale smette di averne bisogno, che ne rilasci il possesso.

La regola aurea dice che quando si crea un oggetto attraverso i metodi **alloc**, **new**, **copy** e **mutableCopy** (i quali aumentano il retainCount di 1) l'oggetto va poi rilasciato (operazione che diminuisce il retainCount di 1).

# Objective-C

## Memory Management

Gli strumenti che Objective-C mette a disposizione per gestire correttamente la memoria sono i metodi **retain**, **release**.

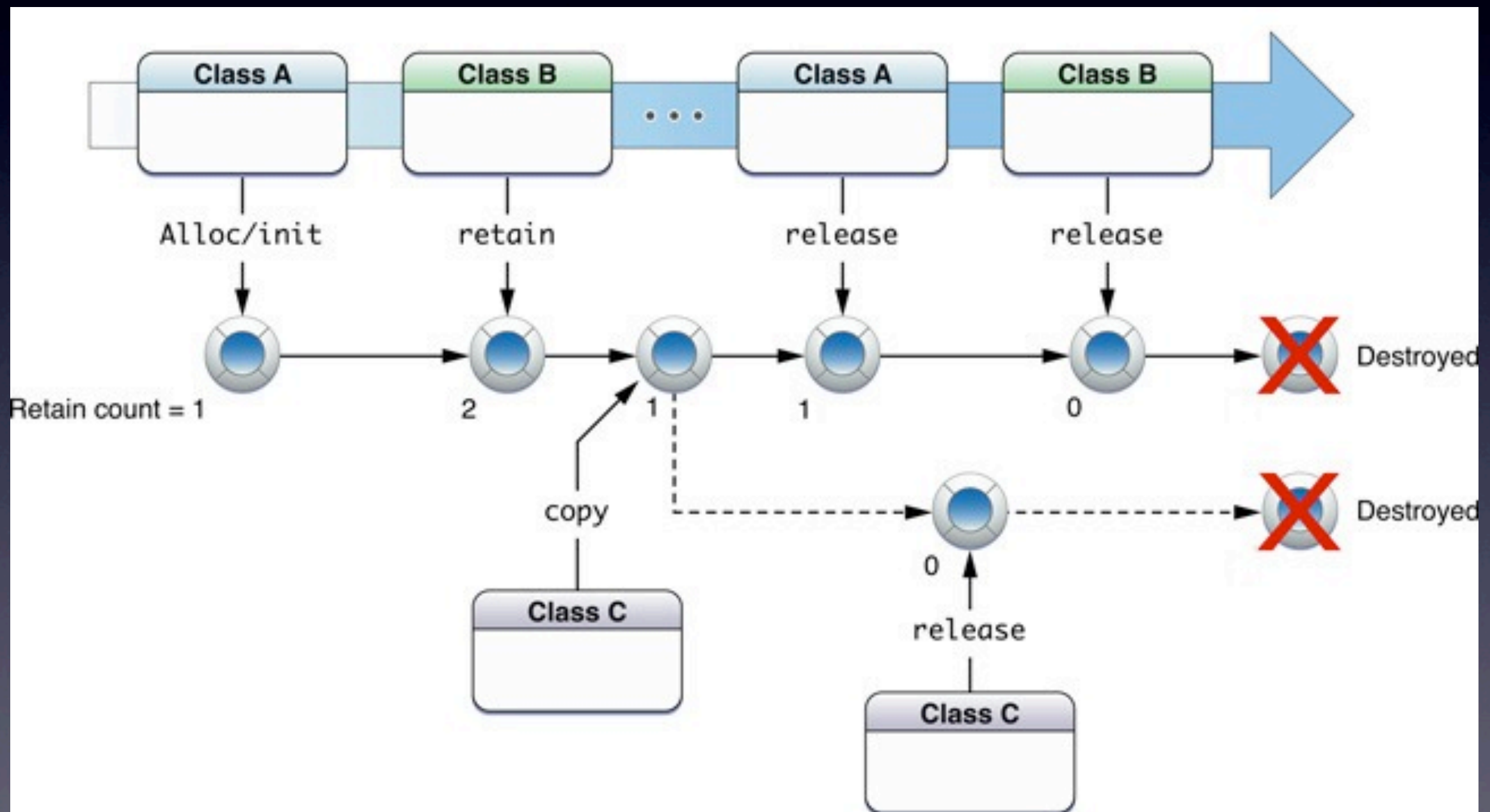
**Retain:** aumenta il retainCount di l

**Release:** diminuisce il retainCount di l

In teoria meno spesso ci si ritrova ad utilizzare i metodi retain e release nel proprio codice meglio è: vuol dire che si è demandata la gestione della memoria a pochi punti specifici del proprio codice e si è ridotto il rischio d'errore.



# Objective-C Memory Management



# Objective-C

## Memory Management

### **Autorelease e AutoreleasePool**

Quando su un oggetto viene richiamato il metodo autorelease l'oggetto stesso viene aggiunto ad un elenco di oggetti ai quali viene promesso di essere rilasciati quando finirà il runloop corrente (ovvero in un momento futuro del quale il programmatore non dovrà preoccuparsi).

Questo fornisce la possibilità di cedere il possesso dell'oggetto in un tempo futuro prima del quale potrà essere trattenuto da altre entità.

# Objective-C

## Memory Management

Esistono due modi di dichiarare i membri di classe: **strong** e **weak**. Un membro strong verrà dichiarato con la parola chiave **retain** e nel proprio setter automaticamente aumenterà il retainCount di 1. Un membro weak verrà dichiarato con la parola chiave **assign** e nel setter non aumenterà il retainCount.

L'uso di referenze strong può creare problemi di cicli di retain (ad.es: l'oggetto "documento" riferisce le pagine che lo compongono ed ogni pagina fa lo stesso con il proprio documento. Nessuno potrà mai essere rilasciato). Per questo esistono le referenze weak.

Ovviamente in un grafo di oggetti dev'esserci qualche referenza forte, per mantenere in vita il grafo, altrimenti tutti gli oggetti che lo compongono verrebbero rilasciati subito.

**Il pattern d'uso è che un genitore ha un riferimento strong verso i propri membri, mentre i membri (se è necessario che conoscano il genitore) hanno un riferimento weak verso di esso.**



# Objective-C

## Memory Management

Esempio di setter di property strong:

```
- (void)setProperty:(NSString *)aValue
{
    if (aValue != self.property) {
        [property release];
        [aValue retain];
        property = aValue;
    }
}
```

Esempio di setter di property weak:

```
- (void)setProperty:(NSString *)aValue
{
    if (aValue != self.property) {
        property = aValue;
    }
}
```

# Objective-C

## Memory Management

### **Regole Auree**

1. ogni qual volta in cui mi ritrovo a fare retain di un oggetto devo assicurarmi di farne un release (o autorelease) per assicurarmi che la memoria venga liberata
2. per ogni membro dichiarato strong (retain) nel dealloc della classe devo preoccuparmi di rilasciarlo
3. solo quando un oggetto viene creato attraverso i metodi alloc, new, copy, mutableCopy, dev'essere rilasciato.
4. nel cambiare valore ad un membro è preferibile utilizzare il setter per assicurarsi un corretto ciclo della gestione della memoria
5. tutte gli oggetti presenti nello xib per i quali viene creato un outlet vanno rilasciati nel dealloc (e settati a nil nel viewDidLoadUnload)
6. non bisogna creare referenze cicliche di tipo strong

ARC



# ARC 1/5

## (Automatic Reference Counting)

**\*ARC è una killer feature. Punto.\***

La gestione della memoria può essere ardua.

- I problemi portano a crash
- i crash causano stress
- lo stress causa il rifiuto dell'app dall'App Store
- il rifiuto dell'app dallo store causa stress

\*\*\* non ci piace lo stress \*\*\*

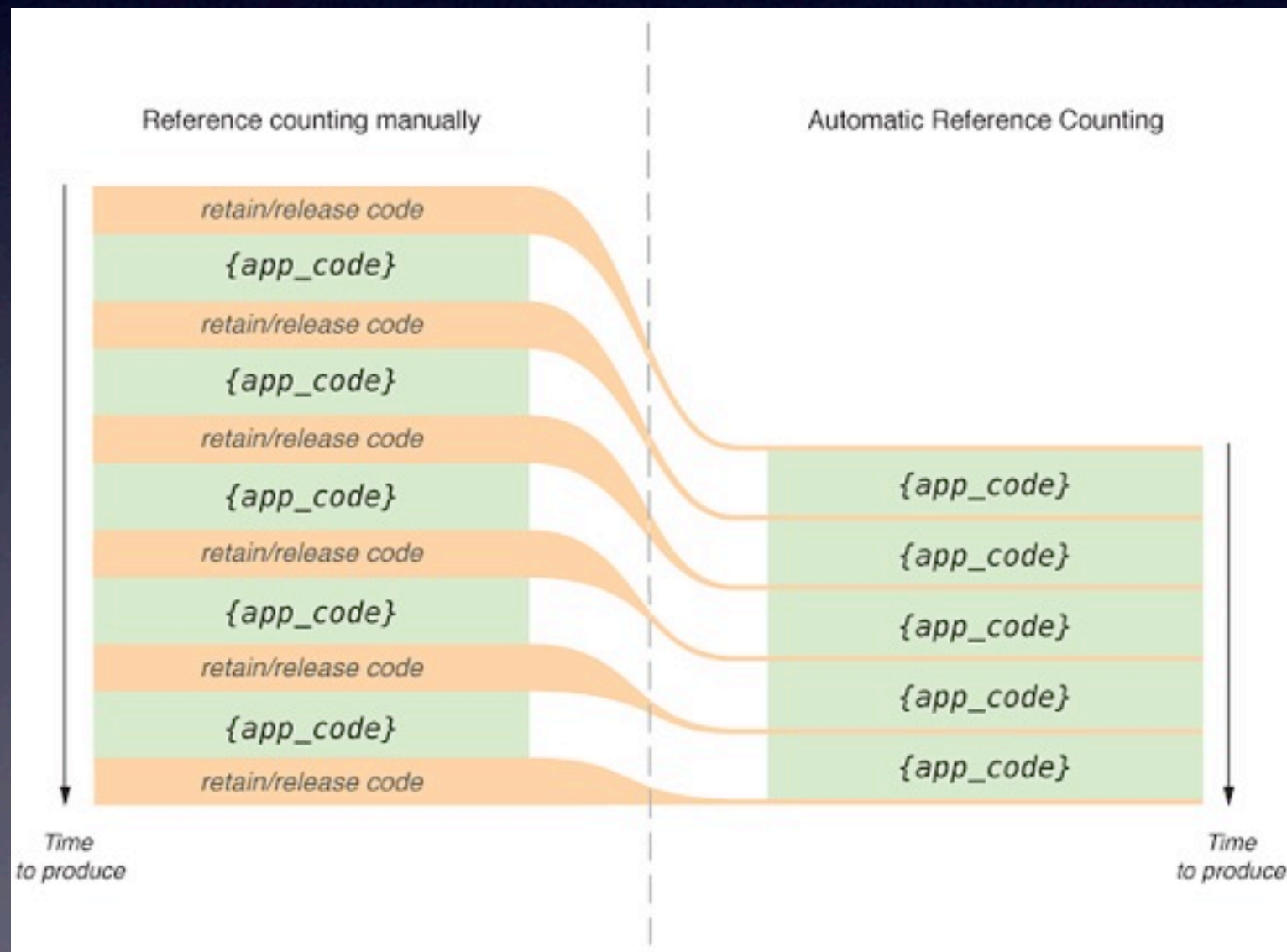
Programmare con le regole MRC (Manual Reference Counting, retain/release) dovrebbe essere banale ma... ci sono molti dettagli... autorelease pools...

**ARC formalizza le convenzioni e automatizza le regole.**

# ARC 2/5

## Cosa fa ARC:

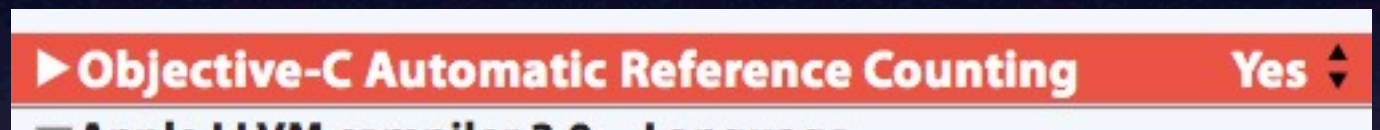
A compile time, il compilatore aggiunge retain/release/autorelease al posto del programmatore. Tali operazioni sono decidibili (matematicamente).



# ARC 3/5

## Cosa **NON** è ARC:

- No new runtime memory model
- No automation for malloc/free, etc.
- No garbage collector
- No heap scans
- No whole app pauses
- No non-deterministic releases



## Cosa è ARC:

- Direttiva di compilazione (Project settings in Xcode)
- retain -> strong
- assign -> unsafe\_unretained
- assign -> weak (per gli oggetti, set to nil automaticamente, no dangling pointers)

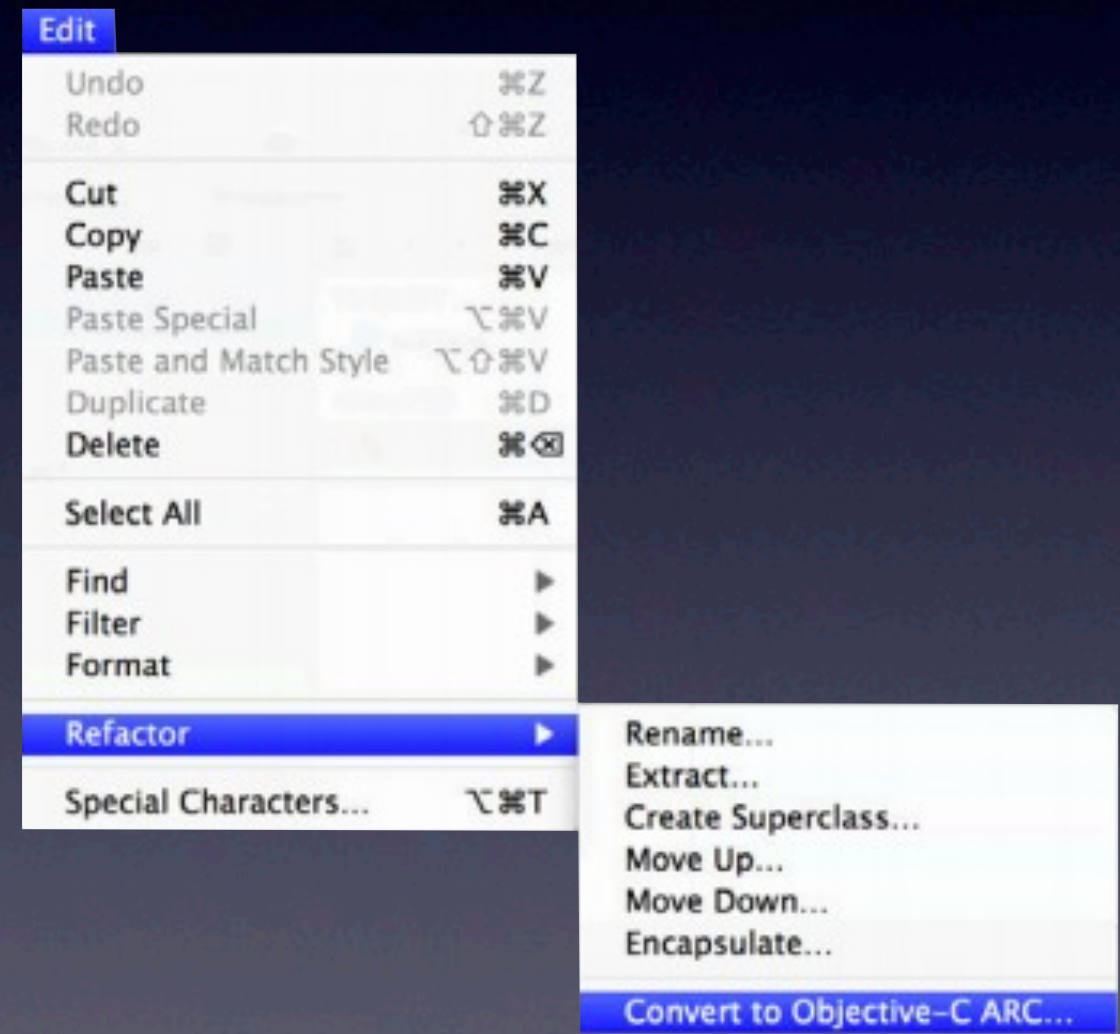





# ARC 4/5

## Migrare codice esistente ad ARC

- Rimuovere tutte le chiamate a retain, release, autorelease;
- Rimpiazzare NSAutoreleasePool con @autoreleasepool;
- @property(assign) diventano @property(weak) per puntatori a oggetti.



# ARC 5/5

	<b>ARC (automatic)</b>	<b>MRC (manual)</b>	<b>Garbage Collector</b>
<b>Pro</b>	<ul style="list-style-type: none"><li>• no memory management,</li><li>• gestione memoria determinata a compile time dal compilatore</li></ul>	<ul style="list-style-type: none"><li>• gestione memoria determinata a compile time</li></ul>	<ul style="list-style-type: none"><li>• no memory management</li></ul>
<b>Cons</b>		<ul style="list-style-type: none"><li>• manuale, soggetta a errori del programmatore</li></ul>	<ul style="list-style-type: none"><li>• demone a runtime (gestione memoria eseguita a runtime)</li></ul>

# Delegation



# Delegation

I protocolli sono un meccanismo per definire dei metodi che una classe deve implementare.

L'interfaccia, in senso stretto, di una classe espone il suo repertorio di metodi pubblici; le entità esterne alla classe interagiscono con essa rispettando questo contratto.

Tanto quanto una classe espone metodi, così essa può anche rendere noto il fatto di invocare dei metodi facenti parte di un protocollo. Tali metodi devono essere implementati altrove e l'oggetto che *promette* di implementarli prende il nome di oggetto delegato.

La dichiarazione di un protocollo ha una sintassi simile alla seguente:

```
@protocol MioProtocollo
- (void)primoMetodoDaImplementare:(id)oggetto;
- (void)secondoMetodoDaImplementare:(id)oggetto;
@end
```

# Delegation

La sintassi per dichiarare l'adozione di un protocollo nell'interfaccia e la successiva implementazione dei metodi è la seguente:

```
@interface MiaClasseDelegata <MioProtocollo> {  
    // variabili di istanza ...  
}  
  
@end  
  
@implementation MiaClasseDelegata  
- (void)primoMetodoDaImplementare:(id)parametro { ... }  
- (void)secondoMetodoDaImplementare:(id)parametro { ... }  
@end
```

L'oggetto delegante che utilizza un protocollo per comunicare con entità esterne ad essa presenta un campo dati con la seguente sintassi:

```
id <MioProtocollo> delegate;
```

# Delegation

Tale variabile di istanza viene settata con il riferimento all'oggetto delegato per rendere noto che sarà esso ad occuparsi dell'implementazione dei metodi del protocollo.

Quando l'oggetto delegante invoca i metodi che sono stati delegati, ciò viene fatto sull'oggetto delegato che ha tipo statico id in quanto non vi sono vincoli sul tipo dinamico della classe che ne fornisce l'implementazione.

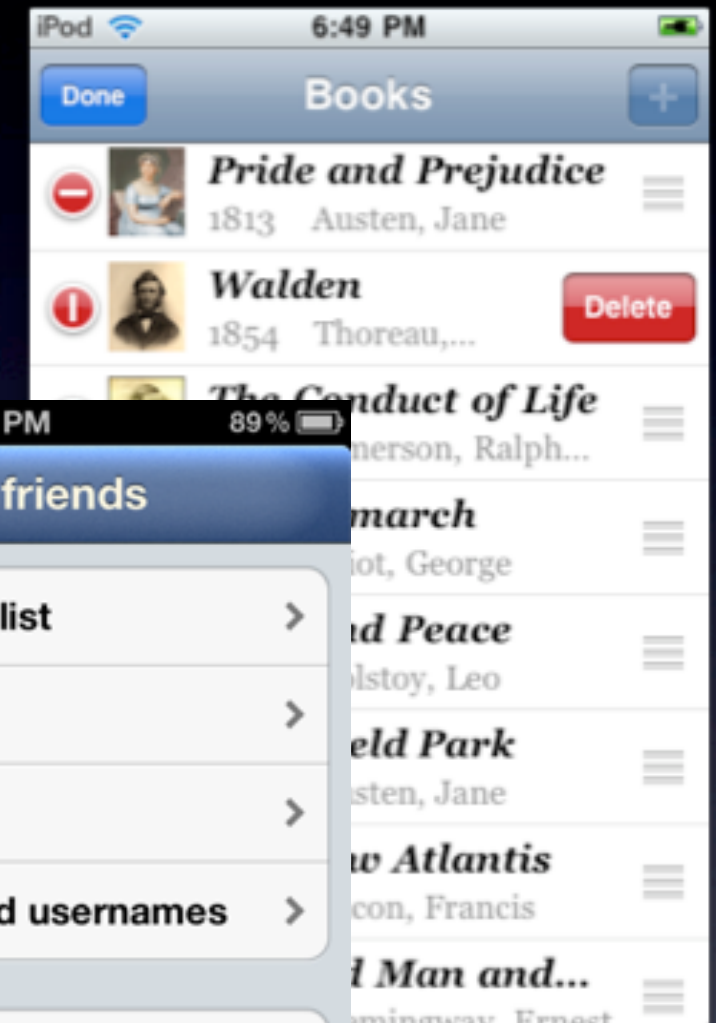
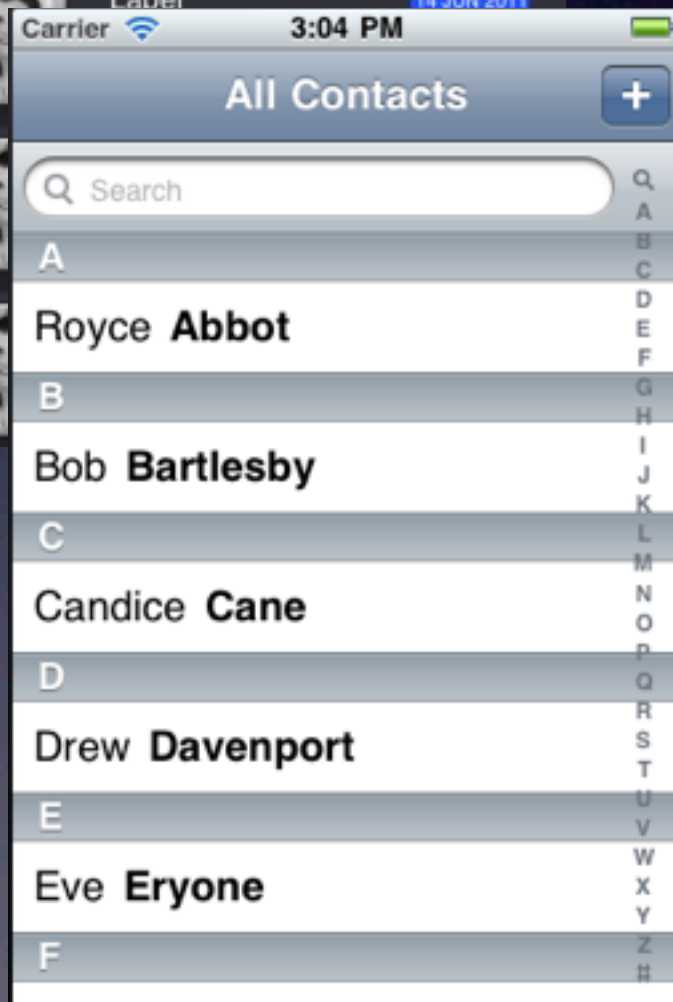
```
[delegate primoMetodoDaImplementare:parametro];
```

Due principali svantaggi della delegazione sono la complicazione del codice nella lettura e l'eccessiva lunghezza che ne deriva dall'utilizzo di tale pattern. D'altra parte il delegation pattern è un pattern fondamentale dell'ingegneria del software che permette di esprimere molta potenza linguistica.



# UITableView

# UITableViews



# UITableViewDelegate

## Configuring Rows for the Table View

- 1 - tableView:heightForRowAtIndexPath:
- 2 - tableView:indentationLevelForRowAtIndexPath:
- 3 - tableView:willDisplayCell:forRowAtIndexPath:

## Managing Accessory Views

- 1 - tableView:accessoryButtonTappedForRowWithIndexPath:
- 2 - tableView:accessoryTypeForRowWithIndexPath: Deprecated in iOS 3.0

## Managing Selections

- 1 - tableView:willSelectRowAtIndexPath:
- 2 - tableView:didSelectRowAtIndexPath:
- 3 - tableView:willDeselectRowAtIndexPath:
- 4 - tableView:didDeselectRowAtIndexPath:

## Modifying the Header and Footer of Sections

- 1 - tableView:viewForHeaderInSection:
- 2 - tableView:viewForFooterInSection:
- 3 - tableView:heightForHeaderInSection:
- 4 - tableView:heightForFooterInSection:
- ...

- [http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDelegate\\_Protocol/Reference/Reference.html](http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDelegate_Protocol/Reference/Reference.html)



# UITableViewDataSource

## Configuring a Table View

- 1 - tableView:cellForRowAtIndexPath: required method
- 2 - numberOfSectionsInTableView:
- 3 - tableView:numberOfRowsInSection: required method
- 4 - sectionIndexTitlesForTableView:
- 5 - tableView:sectionForSectionIndexTitle:atIndex:
- 6 - tableView:titleForHeaderInSection:
- 7 - tableView:titleForFooterInSection:

## Inserting or Deleting Table Rows

- 1 - tableView:commitEditingStyle:forRowAtIndexPath:
- 2 - tableView:canEditRowAtIndexPath:

## Reordering Table Rows

- 1 - tableView:canMoveRowAtIndexPath:
- 2 - tableView:moveRowAtIndexPath:toIndexPath:
- ...

[http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDataSource\\_Protocol/Reference/Reference.html](http://developer.apple.com/library/ios/#documentation/uikit/reference/UITableViewDataSource_Protocol/Reference/Reference.html)