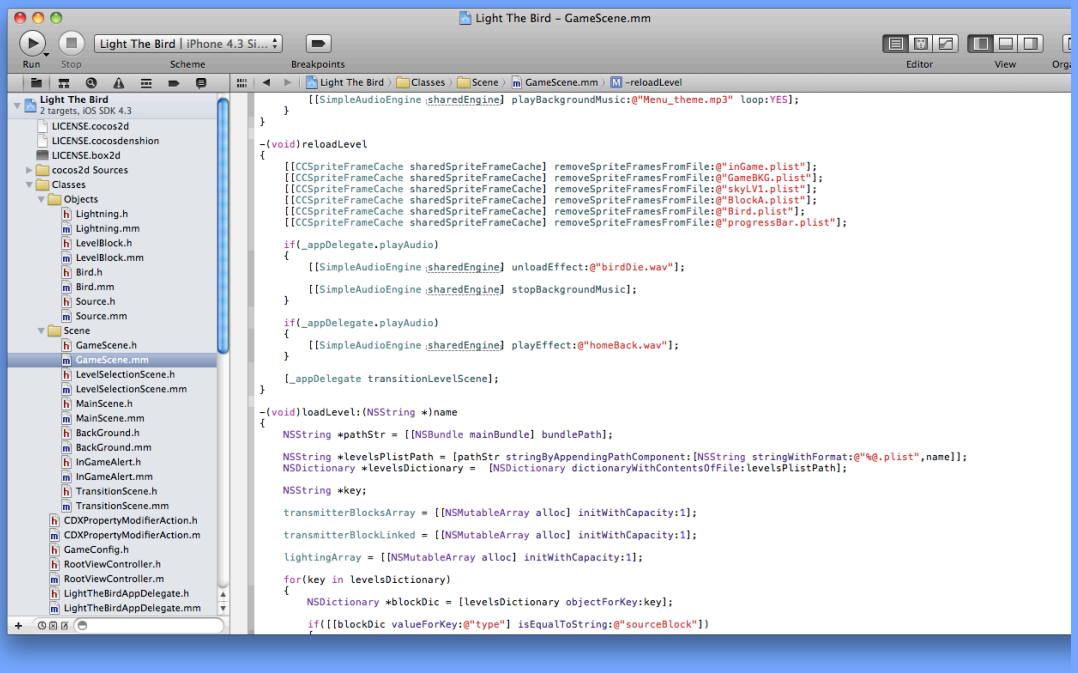


Diventare un Xcoder

Inizia a programmare il Mac usando Objective-C

AGGIORNATO ALLA VERSIONE 4.0 DI XCODE

a cura di Danilo Bottini (danilo)

di Bert Altenberg, Alex Clarke
e Philippe Moughin

Titolo originale
Become an Xcoder Copyright © 2008 by Bert Altenburg, Alex Clarke and Philippe Mougin.
Version 1.15

Autori :
Bert Altenburg, Alex Clarke and Philippe Mougin - **http://www.cocoalab.com**

Edizione Italiana
Xcodeitalia - http://www.xcodeitalia.com

Ringraziamenti

Per il lavoro di traduzione del testo originale, correzione bozze, revisione documento finale, per il prezioso tempo dedicato al progetto e per l'impegno profuso, si ringraziano :

Alessandro (alessandro)
Daniel (dr.got)
Davide (dave)
Fabrizio (fabrizioromeo)
Gino (lagmac)
Manuel (jekrom)
Marco (Ender)
Marco (iMArCo)
Riccardo (rickibrucelee)
Sandro (sandro.vandoni)
Simone (hrs)
Stefano (Stefano1983)

Indice generale

00: Prima di iniziare

Questo libro è stato scritto per te. Dal momento che è gratis, in cambio permettimi di spendere un paio di parole su come promuovere i Mac. Ogni utente Macintosh può aiutare a promuovere la propria piattaforma preferita con un piccolo sforzo. Vediamo come.

Più efficiente sei con il tuo Mac, più è facile che le altre persone considerino un Mac. Quindi, cerca di stare aggiornato visitando siti “Mac-oriented” e leggi riviste Mac. Certamente, impara Objective-C o AppleScript e mettiti ad usarlo in grande. Per i tuoi business, l'uso di AppleScript può farti risparmiare molti soldi e tempo. Guarda il libretto gratuito di Bert "AppleScript for Absolute Starter", disponibile su:

http://www.macscripter.net/books

Mostra al mondo che non tutti usano un PC rendendo così Macintosh più visibile.
Indossare semplici T-shirt Mac in pubblico è uno dei modi, ma ce ne sono altri per rendere Mac ancora più visibile da casa tua. Se si esegue Activity Monitor (nella cartella Utility, che si trova nella cartella Applicazioni sul vostro Mac), noterete che il vostro Mac utilizza tutta la sua potenza di elaborazione solo occasionalmente. Gli scienziati hanno iniziato diversi progetti di calcolo distribuito (DC) come ad esempio Folding@home o SETI@home, che sfruttano la potenza di calcolo non sfruttato, solitamente per il bene comune.

Scarica un piccolo programma gratuito chiamato DC client e fallo lavorare. Questi programmi sono eseguiti con il più basso livello di priorità. Se stai usando un programma sul tuo Mac e questo non ha bisogno di piena potenza di elaborazione, il DC client, immediatamente, ne prende una piccola parte. Quindi, non si dovrebbe notare che è in esecuzione. Questo come può aiutare Mac ?
Ebbene, la maggior parte dei progetti DC mantiene classifiche sui propri siti web delle unità di elaborazione. Se entri in un Mac team (riconoscerai i loro nomi in classifica), potrai aiutare il medesimo a scalare la classifica. Così, utenti che usano altre piattaforme vedranno quanto Mac sta facendo meglio. Ci sono svariate tipologie di DC client, per la matematica, la cura delle malattie e altro.

Per scegliere un progetto DC che ti piace visita:

http://distributedcomputing.info/projects.html

ATTENZIONE: potrebbe dare dipendenza!

Presta attenzione che la piattaforma Macintosh abbia il miglior software. Non solo creandoti i tuoi programmi.
Prendi l'abitudine di dare un commento agli sviluppatori dei programmi che usi, sia esso positivo o negativo.
Anche quando provi un software e non ti piace, spiega sempre il motivo allo sviluppatore.
Segnala eventuali errori (bug), malfunzionamenti e qualsiasi altra cosa che secondo te non funziona come dovrebbe, fornendo una descrizione la più accurata possibile.
Se stai segnalando un errore (bug), cerca di spiegare cosa esattamente stavi facendo nel momento in cui si è verificato.
Paga per il software che usi; se non ti va di pagarlo rivolgiti al software free e/o OpenSource.
Fino a che esisterà un mercato del software per Macintosh, gli sviluppatori continueranno a produrre software sempre migliore.
Un'ultima cosa ti chiediamo.
Per favore, contatta almeno 3 utenti Macintosh che potrebbero essere interessati a programmare per Mac, informali di questo libro e dove possono trovarlo. Oppure avvisali dei 4 punti precedenti.

OK, mentre scarichi un client DC in background, Iniziamo subito!

01: Un programma è una serie di istruzioni

Introduzione

Quando impari a guidare una macchina, hai bisogno d'imparare a gestire più cose contemporaneamente. Per questo a scuola guida ti spiegano tutto sul funzionamento di un'autovettura prima di provare a fartela guidare.
Nello stesso modo, anche programmare richiede di conoscere alcune cose prima d'iniziare, altrimenti il tuo programma non funzionerà, in termine tecnico "andrà in crash", così come in macchina se non sai qual è il pedale del freno inevitabilmente andrai a sbattere.
Affinché non ti senta disorientato, lasciamo l'ambiente di programmazione per un capitolo successivo. Inizieremo invece col farti familiarizzare con il linguaggio Objective-C, spiegando alcune regole matematiche basilari che sicuramente ti saranno familiari.

Alla scuola elementare hai imparato come fare i calcoli, compilando i puntini:

```
2 + 6 = ...
... = 3 * 4
```

(l'asterisco * è lo standard per rappresentare le moltiplicazione su una tastiera del computer)

Alle medie, i puntini sono andati fuori moda e le variabili chiamate x e y (e una nuova parola di fantasia "algebra") sono state aggiunte.

```
2 + 6 = x
y = 3 * 4
```

Variabili

Anche Objective-C usa le variabili. Le variabili non sono altro che nomi convenienti per riferirsi a specifici dati, come ad esempio numeri. Ecco uno *statement* di esempio in Objective-C, una linea di codice che assegna ad una variabile un particolare valore. [1]

```
//[1] x = 4;
```

Il punto e virgola

La variabile chiamata **x** ha preso il valore quattro.
Se presti attenzione alla fine della riga c'è un punto e virgola; questo è richiesto alla fine di ogni *statement*. Perché è richiesto che ci sia un punto e virgola?
Il codice dell'esempio [1] può sembrarti stupido, ma un computer non sa cosa farne. Un programma speciale, chiamato compilatore, è necessario per convertire il testo che tu scrivi in una sequenza di zero e uno comprensibili per il tuo Mac.
Leggere e capire del testo scritto da un umano è molto difficile per un compilatore, così tu devi fornirgli alcuni indizi, per esempio dove un particolare *statement* finisce. Questo è quello che fa il punto e virgola, indica al compilatore dove un determinato *statement* finisce.
Se ti dimentichi anche un singolo punto e virgola, il codice non può essere compilato, e trasformato in qualche cosa di comprensibile dal tuo Mac.
Non ti preoccupare troppo di questo adesso, perché come vedremo più avanti, il compilatore si lamenterà se non riuscirà a compilare il tuo codice e ti aiuterà a trovare che cos'è sbagliato.

Nominare le variabili

Mentre i nomi delle variabili non hanno un significato particolare per il compilatore, usare invece nomi descrittivi, rende un programma più semplice da leggere e capire per noi. Questo è un grande vantaggio se abbiamo bisogno di cercare un errore nel codice.

Gli errori nei programmi sono tradizionalmente chiamati bugs. Trovarli e correggerli è chiamato debugging.

Pertanto, nel codice dobbiamo evitare di utilizzare nomi di variabili non descrittivi come **x**. Per esempio, il nome di una variabile per la larghezza di un'immagine potrebbe essere `pictureWidth` [2].

```
//[2] pictureWidth = 8;
```

Dalla grande questione del dimenticare un punto e virgola per il compilatore, potrai capire come la programmazione sia un insieme di dettagli.
Uno di questi dettagli a cui prestare attenzione è infatti che il codice è *case-sensitive*.

Significa che egli capisce se usi lettere maiuscole o no. Il nome della variabile `pictureWidth` non è lo stesso di `pictureWIDTH` o di `PictureWidth`.
Seguendo le convenzioni generali, io creerò i nomi delle mie variabili fondendo alcune parole, la prima senza maiuscole e facendo iniziare le altre parole che formano la variabile con la lettera maiuscola, così come puoi vedere nell'esempio [2]. Questo stile viene solitamente chiamato *camelCase*. Seguendo questo schema ho ridotto gli errori di programmazione dovuti al *case-sensitive* in maniera drastica.

Nota che il nome di una variabile è sempre una parola unica (o un singolo carattere).

Mentre hai molta libertà nella scelta del nome, ci sono alcune regole alle quali un nome di variabile dev'essere conforme. Quest'elenco potrebbe essere noioso. La prima regola alla quale devi obbedire è che i nomi delle variabili non possono essere parole riservate di Objective-C (una parola che ha un significato speciale per Objective-C). Componendo il nome di una variabile con parole concatenate come pictureWidth sei sempre al sicuro. Mantenere il nome della variabile leggibile, l'uso delle maiuscole con i nomi delle variabili è raccomandato. Se ti attieni a questo schema, avrai un numero di errori (bug) inferiore nei tuoi programmi. Se vuoi continuare ad imparare ancora un paio di regole, finisci questo paragrafo. Oltre alle lettere, l'uso dei numeri è consentito, ma al nome di una variabile non è permesso d'iniziare con un numero. È consentito anche il carattere di underscore "_". Ora alcuni esempi di nome di variabile.

Permessi:
door8k
do8or

Non permessi:
door 8 *(contiene uno spazio)*
8door *(inizia con un numero)*

Sconsigliati:
Door8 *(inizia con lettera maiuscola)*

Usare le variabili nei calcoli

Adesso che sappiamo come assegnare ad una variabile un valore, possiamo effettuare calcoli.
Diamo un'occhiata al codice [3] per il calcolo della superficie di un'immagine.

```
//[3]
pictureWidth=8;
pictureHeight=6;
pictureSurfaceArea=pictureWidth*pictureHeight;
```

Sorprendentemente il compilatore non dice nulla sugli spazi (ad eccezione dei nomi delle variabili, delle parole chiave, etc..). Quindi per rendere il codice più leggibile possiamo usarli. [4]

```
//[4]
pictureWidth = 8;
pictureHeight = 6;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

Numeri Interi o a virgola mobile

Adesso, diamo uno sguardo all'esempio [5], in particolare ai primi due *statement*.

```
//[5]
pictureWidth = 8;     //[5.1]
pictureHeight = 4.5;  //[5.2]
pictureSurfaceArea = pictureWidth * pictureHeight;
```

I numeri in genere possono essere distinti in due tipi: numeri interi e numeri decimali, di cui puoi vederne un esempio negli *statement* [5.1] e [5.2].
Gli interi sono usati per i conteggi, tipo quando avremo da ripetere una serie d'istruzioni un numero specifico di volte (vedi capitolo 7). Conosci i numeri decimali o a virgola mobile, per esempio, nel baseball le medie delle battute.

Il codice dell'esempio[5] così come scritto non può funzionare.
Il problema è che il compilatore vuole sapere in anticipo quale nome di variabile vai ad utilizzare e a quale tipo di dati si riferiscono, per esempio interi o a virgola mobile. Dire al compilatore il nome della variabile e il tipo, viene chiamato “dichiarazione di una variabile”.

```
//[6]
int pictureWidth;                                          //[6.1]
float pictureHeight, pictureSurfaceArea;                   //[6.2]
pictureWidth = 8;
pictureHeight = 4.5;
pictureSurfaceArea = pictureWidth * pictureHeight;
```

Nella linea [6.1], `int` indica che la variabile `pictureWidth` è un intero. Nella linea successiva [6.2], dichiariamo due variabili in una volta, separando i nomi con una virgola e nello specifico, indica che entrambe le variabili sono di tipo `float`, ovvero numeri che contengono parti decimali.

In questo caso è un po' sciocco che `pictureWidth` sia di un tipo differente dalle altre due variabili.
Quello che non sai è che se moltiplichi un `int` con un `float` il risultato del calcolo è un `float`; ecco perché devi dichiarare la variabile `pictureSurfaceArea` come un `float` [6.2].
Perché il compilatore vuole conoscere quando una variabile rappresenta un intero o un numero con parte decimale?
Un programma ha bisogno di una parte della memoria del computer. Il compilatore riserva memoria (byte) per ogni variabile che incontra. Poiché differenti tipi di dati, in questo caso `int` e `float`, richiedono differenti quantità di memoria e diversa rappresentazione, il compilatore deve riservare la corretta quantità di memoria e usare la giusta rappresentazione.

E se dovessimo lavorare con numeri molto grandi o con un'alta precisione decimale?
Essi non riescono a stare nello spazio di pochi byte riservati dal compilatore, che fare?
Ci sono due risposte a questo: la prima, sia `int` che `float` hanno una controparte per memorizzare numeri grandi (o con alta precisione). In molti sistemi sono rispettivamente `long` e `double`. Ma anche questo può non bastare, il che ci porta alla seconda risposta: come programmatore è tuo dovere stare all'erta contro questi problemi. In ogni caso, non è un problema da discutere nel primo capitolo di un libro introduttivo.
Inoltre, i numeri sia interi che decimali possono essere anche negativi, come sarebbe il tuo conto corrente bancario se spendessi più soldi di quelli disponibili.
Se sai che il valore di una variabile non sarà mai negativo, puoi restringerne il range dei valori e adattarlo ai byte disponibili.

Nell'esempio [7] viene dichiarata la variabile `chocolateBarsInStock` di tipo `unsigned int`.

```
//[7]
unsigned int chocolateBarsInStock;
```

Questo significa che `chocolateBarsInStock` non potrà mai assumere un valore negativo, ma solo valori maggiori o uguali a zero. Il tipo `unsigned int` rappresenta quindi numeri interi maggiori o uguali a zero.

Dichiarare una variabile

È possibile dichiarare una variabile e assegnare un valore tutto in una volta [8].

```
//[8]
int x = 10;
float y = 3.5, z = 42;
```

Ti fa risparmiare qualche digitazione.

Tipi di dati

Come abbiamo appena visto, i dati memorizzati in una variabile possono essere di diverso tipo, per esempio `int` o `float`.
In Objective-C, semplici tipi di dati come questi sono anche conosciuti come dati scalari.
Nella tabella seguente una lista dei più comuni disponibili in Objective-C.

Nome	Tipo	Esempio
void	Vuoto	Nothing
int	Intero	...-1, 0, 1, 2...
unsigned	Intero senza segno	0, 1, 2...
float	Numero a virgola mobile	-0.333, 0.5, 1.223, 202.85556

Nome	Tipo	Esempio
double	Numero a virgola mobile e doppia precisione	5,25253E+25
char	Carattere	hello
BOOL	Booleano	0, 1; TRUE, FALSE; YES, NO.

Operazioni matematiche

Negli esempi precedenti abbiamo eseguito un'operazione di moltiplicazione.
Per eseguire le altre operazioni matematiche di base, devi usare i seguenti simboli, conosciuti ufficialmente come operatori.

```
+  per l'addizione
-  per la sottrazione
/  per la divisione
*  per la moltiplicazione
```

Usando gli operatori, siamo in grado di eseguire una vasta gamma di calcoli. Se si dà un'occhiata al codice dei programmatori professionisti in Objective-C, t'imbatterai in un paio di peculiarità. Invece di scrivere x = x + 1 il programmatore spesso ricorre a qualcosa di diverso come negli esempi [9] e [10].

```
//[9]
x++;

//[10]
++x;
```

In entrambi i casi questo significa: incrementa x di uno.
In alcune circostanze è importante se ++ è messo prima o dopo il nome della variabile.
Controlla nei seguenti esempi [11] e [12].

```
//[11]
x = 10;
y = 2 * (x++);

//[12]
```

```
x = 10;
y = 2 * (++x);
```

Nell'esempio [11], y è uguale a 20 e x uguale a 11. Al contrario nel [12] x è sempre uguale a 11, mentre y sarà uguale a 22.
Questo perché nel primo caso x viene incrementato di uno dopo la moltiplicazione, mentre nel secondo caso x viene incrementato, sempre di uno, prima della moltiplicazione.
Ecco spiegata la differenza, fondamentale, di mettere gli operatori d'incremento prima o dopo il nome della variabile.
Per chiarire ulteriormente il concetto, il codice dell'esempio [12] è equivalente a quello dell'esempio [13].

```
//[13]
x = 10;
x++;
y = 2 * x;
```

Quindi, il programmatore ha effettivamente fuso insieme due *statement* in uno. Personalmente, credo che questo renda un programma difficile da leggere. Se scegliete la via breve va bene ma attenzione che un errore potrebbe essere in agguato.

Parentesi

Se sei riuscito a passare la scuola media saranno per te cosa vecchia, ma le parentesi possono essere usate per determinare l'ordine nel quale le operazioni vengono eseguite. Normalmente * e / hanno la precedenza su + e -. Così 2*3+4 è uguale a 10. Usando le parentesi, puoi forzare l'addizione ad essere eseguita per prima: 2 * (3 + 4) uguale 14.

Divisione

L'operatore di divisione va trattato con particolare attenzione, perché è leggermente differente quando viene usato con interi o a virgola mobile. Dai un'occhiata ai seguenti esempi [14, 15].

```
//[14]
int x = 5, y = 12, ratio;
ratio =  y / x;
```

```
//[15]
float x = 5, y = 12, ratio;
ratio =  y / x;
```

Nel primo caso [14], il risultato è 2. Solo nel secondo caso [15], il risultato è quello che probabilmente ti aspetti: 2.4.

Booleani

Un Booleano è un semplice valore logico vero o falso, dove 1 e 0 stanno per true e false, spesso usati in maniera intercambiabile e possono essere considerati equivalenti:

True	False
1	0

Sono spesso utilizzati per valutare se svolgere alcune azioni a seconda dal valore booleano di qualche variabile o funzione.

Modulo

Un operatore con il quale probabilmente non avete familiarità è % (modulo). Esso non funziona come si potrebbe pensare ovvero il calcolo di una percentuale.

Il risultato dell'operatore % è il resto della divisione intera del primo operando con il secondo (se il valore del secondo operando è zero, il comportamento di % è indefinito). [16]

```
//[16]
int x = 13, y = 5, remainder;
remainder =  x % y;
```

Ora il risultato è che il resto (remainder) è uguale a 3, perché x è uguale a 2*y+3.

Ecco alcuni esempi di modulo:

```
21 % 7 è uguale a 0
22 % 7 è uguale a 1
23 % 7 è uguale a 2
```

```
24 % 7 è uguale a 3
27 % 7 è uguale a 6
30 % 2 è uguale a 0
31 % 2 è uguale a 1
32 % 2 è uguale a 0
33 % 2 è uguale a 1
34 % 2 è uguale a 0
50 % 9 è uguale a 5
60 % 29 è uguale a 2
```

L'operatore % può tornarti utile, ma ricorda che lavora solo con interi. Un uso molto comune è quello di determinare se un intero è dispari o pari.
Se è pari, il modulo di due è uguale a zero. Altrimenti sarà uguale ad un altro valore. [17]

```
//[17]
int unIntero;
//Qui ci sarebbe il codice che imposta il valore di unIntero
if ((unIntero % 2) == 0)
{
NSLog(@"unIntero è pari");
}
else
{
NSLog(@"unIntero è dispari");
}
```

02: No comment? Inaccettabile!

Introduzione

Utilizzando nomi di variabili autoesplicativi possiamo rendere il nostro codice molto più leggibile [1].

```
//[1]
float immagineLarg, immagineAltez, immagineArea;
immagineLarg = 8.0;
immagineAltez = 4.5;
immagineArea = immagineLarg * immagineAltez;
```

Per il momento il nostro codice d'esempio è breve, ma anche i programmi più semplici possono crescere velocemente a centinaia o migliaia di righe di codice.
Quando tornate sul codice, probabilmente dopo qualche settimana o mese, potrebbe essere difficile ricordarsi i motivi di alcune scelte di programmazione. Ecco a cosa servono i commenti.
I commenti sono utili a capire velocemente cosa fa una particolare porzione di codice e perché si trova in quel punto. Alcuni programmatori arrivano addirittura a cominciare a scrivere una classe come se fossero commenti, cosa che li aiuta ad organizzare i pensieri e a procedere senza intoppi.
Ti consigliamo quindi, di spendere un po' del tuo tempo per commentare il tuo codice. Ti possiamo assicurare che il tempo speso oggi sarà tutto tempo guadagnato domani.
Se condividi il codice con altri, i tuoi commenti li aiuteranno ad utilizzarlo più velocemente.

Fare un commento

Per creare un commento inizia la riga con due slash.

```
// Questo è un commento
```

Xcode evidenzia i commenti in verde. Se un commento è molto lungo e si dispone su più righe racchiudilo tra /* */

```
/* Questo è un commento
che si estende su due righe*/
```

OutComment

A breve ci occuperemo del debug di un programma e degli ottimi strumenti forniti da Xcode al riguardo.
Uno dei modi di farlo alla "vecchia maniera", invece, si chiama *Outcomment*. Racchiudendo una porzione di codice tra /* */ puoi disabilitare quel codice, per vedere se il resto funziona come ci si aspetta. Questo ti permette di rintracciare un *bug*.
Se, per esempio, la parte commentata contiene l'assegnazione di una particolare variabile, puoi commentare l'assegnazione e inserire una riga temporanea per assegnare un valore utile per testare il resto del tuo codice.

Perché commentare?

L'importanza dei commenti non sarà mai abbastanza ribadita.
Spesso è utile aggiungere una spiegazione in lingua umana (italiano, inglese...) di ciò che accade all'interno di una serie di righe di codice. Questo permette di non dover dedurre cosa il codice faccia e di capire se il problema può essere generato da quella porzione di codice.
Dovresti anche usare i commenti per spiegare cose che non si possono dedurre direttamente dal codice. Per esempio, se codifichi una funzione matematica, usando un modello specifico descritto in dettaglio in qualche libro, dovresti apporre un riferimento bibliografico al codice.
A volte è utile scrivere qualche commento prima di lavorare sul codice. Ti aiuterà a strutturare meglio i pensieri e programmare diventerà più semplice.
Il codice nel presente libro non sempre è commentato come dovrebbe, solo perché è già circondato da esaurienti spiegazioni.

03: Funzioni

Introduzione

Il più lungo blocco di codice che abbiamo visto finora aveva solo cinque istruzioni. Programmi costituiti da diverse migliaia di linee potrebbero sembrare un traguardo lontano ma, a causa della natura di Objective-C, dobbiamo discutere del modo in cui i programmi sono organizzati in questa fase iniziale.
Se un programma fosse costituito da una lunga, continua successione d'istruzioni, sarebbe difficile trovare e correggere i *bugs*. Inoltre, una particolare serie d'istruzioni potrebbe comparire in diversi punti del tuo programma. Se in quella sequenza ci fosse un *bug*, dovresti correggere il medesimo *bug* in diversi punti. Un incubo, perché è facile lasciarsene sfuggire uno o più.
Quindi è stato studiato un modo per organizzare il codice, rendendo più semplice correggere i bugs.
La soluzione a questo problema è di raggruppare le istruzioni in base alla loro funzione. Per esempio, potresti avere una sequenza d'istruzioni che permettono di calcolare l'area della superficie di un cerchio. Verificato che questa serie d'istruzioni è affidabile, non avrai mai più bisogno di controllare ancora quel codice per vedere se il bug si trova li. La sequenza d'istruzioni, chiamata funzione, ha un nome e puoi invocare quella sequenza d'istruzioni tramite quel nome perché il suo codice venga eseguito.
Il concetto dell'uso delle funzioni è così fondamentale, che c'è sempre almeno una funzione in ogni programma: la funzione `main()`.

La funzione `main()` è ciò che il compilatore cerca, così saprà dove dovrà cominciare l'esecuzione del codice a programma avviato.

La funzione main()

Diamo uno sguardo alla funzione `main()` più dettagliatamente. [1]

```
//[1]
main()
{                                                                    //[1.2]
    // Body of the main() function. Put your code here.
}                                                                    //[1.4]
```

L'istruzione [1.1] mostra il nome della funzione, ovvero "main", seguita dalle parentesi aperta e chiusa. Mentre "main" è una parola riservata e la funzione

main() dev'essere sempre presente, quando definisci le tue funzioni, puoi chiamarle più o meno in qualsiasi modo che ti piaccia.
Le parentesi sono li per una buona ragione, ma ne discuteremo più avanti in questo stesso capitolo. Nelle linee successive [1.2, 1.4], ci sono delle parentesi graffe. Dobbiamo mettere il nostro codice tra quelle parentesi graffe { }. Tutto ciò che si trova tra parentesi è chiamato "corpo della funzione".
Ho preso del codice dal primo capitolo e l'ho messo all'interno del corpo[2].

```
//[2]
main()
{
    // Variables are declared below
    float pictureWidth, pictureHeight, pictureSurfaceArea;
    // We initialize the variables (we give the variables a value)
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    // Here the actual calculation is performed
    pictureSurfaceArea = pictureWidth * pictureHeight;
}
```

La nostra prima funzione

Se continuassimo ad aggiungere codice nel corpo della funzione main(), finiremmo per trovarci con quel codice non strutturato e difficile da correggere, cosa che vogliamo evitare. Ora scriveremo un altro programma un po' più strutturato.
Oltre alla funzione obbligatoria main(), creeremo una funzione circleArea() [3].

```
//[3]
main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea;
    pictureWidth = 8.0;
    pictureHeight = 4.5;
    pictureSurfaceArea = pictureWidth * pictureHeight;
}

circleArea()                                                    // [3.9]
{

}
```

È stato facile, ma la nostra funzione personalizzata che inizia all'istruzione [3.9] non fa ancora niente. Notate che la specifica della funzione è al di fuori del corpo della funzione `main()`. In altre parole le due funzioni non sono annidate.
La nostra nuova funzione `circleArea()` dev'essere invocata dalla funzione `main()`. Vediamo come possiamo farlo [4].

```
//[4]
main()
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
    circleRadius, circleSurfaceArea;                                    // [4.4]
    pictureWidth  = 8.0;
    pictureHeight = 4.5;
    circleRadius  = 5.0;                                                // [4.7]
    pictureSurfaceArea =  pictureWidth * pictureHeight;
    // Here we call our function!
    circleSurfaceArea = circleArea(circleRadius);                       // [4.11]
}
```

Nota: il resto del codice non è visualizzato (vedi [3]).

Passare degli argomenti

Abbiamo aggiunto un paio di nomi di variabili del tipo `float` [4.4] e abbiamo inizializzato la variabile `circleRadius`, ovvero gli abbiamo dato un valore [4.7]. La riga di maggior interesse è [4.11], dove la funzione `circleArea()` viene invocata. Come puoi vedere, il nome della variabile `circleRadius` è stato messo fra le due parentesi. Esso è un argomento della funzione `circleArea()`. Il valore della variabile `circleRadius` sta per essere passato alla funzione `circleArea()`.
Quando la funzione `circleArea()` ha fatto il suo lavoro di eseguire il calcolo vero e proprio, deve restituire il risultato. Modifichiamo la funzione `circleArea()` di [3] per ottenere ciò che vogliamo [5].

```
//[5]
circleArea(float theRadius)                                             // [5.1]
{
    float theArea;                                                      // [5.3]
    theArea = 3.1416 * theRadius * theRadius;  // pi times r square  [5.4]
    return theArea;                                                     // [5.5]
```

```
}
```

Nella [5.1] abbiamo definito che per la funzione `circleArea()` un valore di tipo `float` è richiesto come input. Quando viene ricevuto, questo valore è immagazzinato nella variabile nominata `theRadius`. Usiamo una seconda variabile, `theArea`, per immagazzinare il risultato del calcolo [5.4], pertanto la dobbiamo dichiarare [5.3], allo stesso modo in cui abbiamo dichiarato le variabili nella funzione `main()` [4.4].
Avrai notato che la dichiarazione della variabile `theRadius` è stata collocata fra le parentesi [5.1].
La linea [5.5] restituisce il risultato alla parte del programma da cui la funzione è stata invocata. Come conseguenza, nella linea [4.11], la variabile `circleSurfaceArea` è inizializzata con quel valore.
La funzione nell'esempio [5] è completa, eccetto per una cosa. Non abbiamo specificato il tipo di dati che la funzione restituirà. Il compilatore richiede che lo facciamo, per cui non abbiamo altra scelta che obbedire e indicare che è di tipo `float` [6.1].

```
//[6]
float circleArea(float theRadius)                                    //[6.1]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}
```

Come indica la prima parola della linea [6.1], il dato restituito da questa funzione (in altre parole, il valore della variabile `theArea`) è di tipo `float`. Come programmatore, dovrai assicurarti che la variabile `circleSurfaceArea` nella funzione `main()` [4.4] sia anch'essa dello stesso tipo, in modo che il compilatore non abbia alcuna ragione di assillarci.
Non tutte le funzioni richiedono un argomento. Se non ce n'è alcuno, le parentesi () sono comunque obbligatorie, anche se sono vuote.

```
//[7]
int throwDice()
{
    int noOfEyes;
    // Code to generate a random value from 1 to 6
    return noOfEyes;
```

```
}
```

Restituire valori

Non tutte le funzioni restituiscono un valore. Se una funzione non restituisce nessun valore, essa è di tipo void. L'istruzione `return` è quindi facoltativa. Se la usi, la parola chiave `return` non dev'essere seguita da alcun valore o nome di variabile.[8]

```
//[8]
void beepXTimes(int x);
{
    // Code to beep x times
    return;
}
```

se una funzione ha più di un argomento, come la funzione `pictureSurfaceArea()` qui sotto, gli argomenti sono separati da una virgola.

```
//[9]
float pictureSurfaceArea(float theWidth, float theHeight)
{
    // Code to calculate surface area
}
```

La funzione `main()` dovrebbe restituire un intero, per convenzione, quindi si, anch'essa ha la sua istruzione `return`. Dovrebbe restituire 0 (zero, [10.9]), ad indicare che la funzione è stata eseguita senza problemi. Siccome la funzione `main()` restituisce un intero, dobbiamo scrivere "`int`" prima di `main()` [10.1]. Mettiamo tutto il codice che abbiamo in un unico listato.

```
//[10]
int main()                                                    // [10.1]
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
       circleRadius, circleSurfaceArea;
    pictureWidth = 8;
    pictureHeight = 4.5;
    circleRadius = 5.0;
    pictureSurfaceArea =  pictureWidth * pictureHeight;
```

```
    circleSurfaceArea = circleArea(circleRadius);                    // [10.8]
    return 0;                                                        // [10.9]
}

float circleArea(float theRadius)                                    // [10.12]
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;                        // [10.14]
    return theArea;
}
```

Far funzionare il tutto

Come puoi vedere [10], abbiamo una funzione `main()` [10.1] e un'altra funzione che abbiamo definito noi stessi [10.12]. Se compilassimo ora questo codice, il compilatore farebbe ancora lo schizzinoso. Nella linea [10.8] ci direbbe che non conosce alcuna funzione chiamata `circleArea()`. Perché?
Il compilatore inizia a leggere la funzione `main()` e incontra subito qualcosa che non conosce. Non cerca oltre e ti da quest'avvertimento. Per soddisfare il compilatore, dovremo aggiungere una dichiarazione della funzione prima dell'istruzione che contiene `int main()` [11.1]. Non è nulla di complicato, poiché è lo stesso codice di riga [10.12] con la differenza che finisce con un punto e virgola. Ora il compilatore non sarà sorpreso quando incontrerà la chiamata di questa funzione.

```
//[11]
float circleArea(float theRadius);  // function declaration

int main()                                                           // [11.1]
{
    // Main function code here...
}
```

Nota: il resto del codice non è visualizzato (vedi [10]).

Presto compileremo questo programma per davvero. Prima, però, un paio di aggiunte e le conclusioni.

Quando si scrivono dei programmi, è consigliabile prevedere il riutilizzo futuro del codice. Il nostro programma potrebbe avere una funzione `rectangleArea()`,

come mostrato sotto [12] e questa funzione potrebbe essere invocata nella nostra funzione `main()`. È utile anche se il codice che abbiamo messo in una funzione viene usato una sola volta. La funzione `main()` diventa più facile da leggere. Se devi fare il debug del tuo codice, sarà più facile scoprire dove potrebbe trovarsi il bug nel tuo programma. Potresti scoprire che si trova proprio in una funzione. Invece di dover leggere una lunga sequenza d'istruzioni, devi solo verificare le istruzioni della funzione, che sono facili da trovare, grazie alle parentesi graffe di apertura e di chiusura.

```
//[12]
float rectangleArea(float length, float width)                    // [12.1]
{
    return (length * width);                                      // [12.3]
}
```

Come puoi vedere, in un caso semplice come questo, è possibile scrivere una singola istruzione [12.3] che comprenda il calcolo e la restituzione del risultato. Ho usato la variabile superflua `theArea` *nella [10.14] solo per farti vedere come dichiarare una variabile in una funzione.*

Sebbene le funzioni che abbiamo definito noi stessi in questo capitolo sono piuttosto banali, è importante capire che potresti modificare una funzione senza alcun impatto sul codice che la invoca, fintanto che non cambierai la dichiarazione della funzione (in altre parole, la sua prima linea).
Per esempio potresti cambiare i nomi delle variabili in una funzione e la funzione funzionerebbe ancora (ciò non sconvolgerebbe il resto del programma). Qualcun altro potrebbe scrivere la funzione e tu potresti usarla senza sapere cosa succede dentro alla funzione. Tutto ciò che hai bisogno di sapere è come usare la funzione. Che significa sapere:

- il nome della funzione
- il numero, l'ordine e il tipo degli argomenti della funzione
- cosa restituisce la funzione (il valore dell'area della superficie del rettangolo) e il tipo del risultato

Nell'esempio [12], queste risposte sono rispettivamente:

- `rectangleArea`
- Due argomenti, entrambi `float`, dove il primo rappresenta la lunghezza e il secondo la larghezza.

- La funzione restituisce qualcosa e il risultato è di tipo `float` (come si può apprendere dalla prima parola dell'istruzione [12.1]).

Variabili protette

Il codice dentro alla funzione è protetto dal resto del programma e dalle altre funzioni.

Questo significa che il valore di una variabile dentro ad una funzione non può essere modificato da alcun'altra variabile di qualsiasi altra funzione, anche se ha lo stesso nome. Questa è una delle principali caratteristiche di Objective-C. Nel Capitolo 5, parleremo ancora di questo comportamento. Ma adesso, stiamo per avviare Xcode ed eseguire il programma qui sopra [10].

04: Mostrare a schermo qualcosa

Introduzione

Abbiamo fatto buoni progressi con il nostro programma, ma non abbiamo ancora discusso come mostrare a schermo i risultati della nostra elaborazione. Il linguaggio Objective-C non offre costrutti per farlo ma fortunatamente altri programmatori hanno scritto delle funzioni che ci vengono in aiuto e che possiamo richiamare all'occorrenza. Ci sono vari metodi per mostrare a schermo i risultati. In questo libro utilizzeremo una funzione fornita dall'ambiente Cocoa: NSLog(). Così facendo non ci dobbiamo preoccupare di dover scrivere una funzione personalizzata per mostrare a schermo i nostri risultati.

Lo scopo principale della funzione NSLog() è mostrare i messaggi di errore, non l'output di un'applicazione. Ciò nonostante è facile da usare e pertanto la adotteremo in questo libro per mostrare i nostri risultati. Quando avrai una maggior padronanza del linguaggio sarai in grado di usare tecniche più sofisticate.

Usare NSLog()

Andiamo a vedere come viene usata NSLog():

```
//[1]
int main()
{
    NSLog(@"Julia è la mia attrice preferita.");
    return 0;
}
```

Dopo aver eseguito il codice dell'esempio [1] dovrebbe apparire a schermo la scritta "Julia è la mia attrice preferita." Il testo compreso tra @" e " è chiamato stringa (e può avere zero o più caratteri).
Oltre alla stringa in se, la funzione NSLog() mostra varie informazioni addizionali, come la data corrente e il nome dell'applicazione. Per esempio, sul mio sistema l'output completo del programma [1] è:
2005-12-22 17:39:23.084 test[399] Julia è la mia attrice preferita.

Nota: nei successivi esempi verranno mostrate solo le parti interessanti della funzione main().

```
//[2]
NSLog(@"");                                    [2.1]
NSLog(@" ");                                   [2.2]
```

L'istruzione [2.1] contiene zero caratteri e rappresenta una stringa vuota (ovvero ha lunghezza zero). L'istruzione [2.2] invece non è una stringa vuota (anche se lo sembra): essa contiene un singolo spazio, perciò ha lunghezza pari ad uno.
Diverse sequenze di caratteri speciali hanno un significato particolare in una stringa. Questi caratteri speciali sono conosciuti come sequenze di escape.
Per esempio, per forzare l'ultima parola della nostra frase ad andare a capo dev'essere inserito un codice speciale nell'istruzione [3.1]. Il codice da usare è \n.

```
//[3]
NSLog(@"Julia è la mia attrice \npreferita.");      [3.1]
```

Ora l'output del nostro programma sarà questo (è mostrata solo la parte interessante):

```
Julia è la mia attrice
preferita.
```

La backslash in [3.1] è chiamato carattere di escape e indica alla NSLog() che il prossimo carattere non è un carattere da mostrare a schermo ma è un carattere con un significato speciale: in questo caso "n" significa "vai a capo".
Nel raro caso che tu voglia mostrare a schermo una backslash sembrerebbe ci sia un problema: se il carattere dopo la backslash ha un significato particolare com'è possibile stamparne uno? Bene, basta aggiungere un'altra backslash prima (o dopo, non fa differenza) quella principale. Questo dice alla funzione NSLog() che la (seconda) backslash è da stampare e che ogni significato speciale dev'essere ignorato. Ecco un esempio:

```
NSLog(@"Julia è la mia attrice preferita.\\n");          [4.1]
```

L'istruzione [4.1] darà questo output:

```
Julia è la mia attrice preferita.\n
```

Mostrare il valore delle variabili

Poco fa abbiamo mostrato a video solo alcune stringhe. Andiamo ora a visualizzare un valore ottenuto da alcuni calcoli.

```
//[5]
int x, integerToDisplay;
x = 1;
integerToDisplay = 5 + x;
NSLog(@”Il valore dell'intero è %d.”, integerToDisplay);
```

Nota che tra parentesi abbiamo una stringa, una virgola ed il nome di una variabile. Se notate, la stringa contiene qualcosa di strano: %d.
Come la backslash, anche il carattere % ha un significato particolare. Durante l'esecuzione, al posto di %d sarà inserito il valore dopo la virgola, nel nostro caso il valore della variabile integerToDisplay. L'output dell'esempio [5] sarà:

Il valore dell'intero è 6.

Per mostrare un float bisogna usare %f al posto di %d:

```
//[6]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@”Il valore float è %f.”, floatToDisplay);
```

Secondo le necessità è possibile indicare quante cifre significative mostrare. Per mostrare, ad esempio, due cifre significative inserisci .2 tra % ed f, così:

```
//[7]
float x, floatToDisplay;
x = 12345.09876;
floatToDisplay = x/3.1416;
NSLog(@”Il valore float è %.2f.”, floatToDisplay);
```

Infine potresti aver bisogno di creare una tabella di valori.
Ad esempio, immagina una tabella di conversione da gradi Fahrenheit a gradi Celsius. Se vuoi mostrare i valori in modo elegante i valori devono stare in due

colonne contenenti dati tutti con la stessa (fissa) larghezza. Puoi specificare questa larghezza con un valore intero tra % ed f (o % e d, a seconda dei casi). In ogni modo se il valore da mostrare è più largo della larghezza impostata, quest'ultimo valore sarà ignorato.

```
//[8]
int x = 123456;
NSLog(@"%2d", x);                                    [8.2]
NSLog(@"%4d", x);                                    [8.3]
NSLog(@"%6d", x);
NSLog(@"%8d", x);                                    [8.5]
```

L'esempio [8] ha il seguente output:

```
123456
123456
123456
    123456
```

Nelle prime due istruzioni [8.2, 8.3] abbiamo richiesto uno spazio troppo piccolo per il numero completo ma ad ogni modo lo spazio necessario viene preso lo stesso. Solo l'istruzione [8.5] specifica una larghezza più grande del valore, quindi ora vediamo che sono apparsi degli spazi vuoti addizionali.
È anche possibile combinare il numero di cifre significative dopo la virgola e la larghezza del valore:

```
//[9]
float x=1234.5678
NSLog(@"Riserva 10 caratteri e mostra 2 cifre significative.");
NSLog(@"%10.2f", x);
```

Mostrare il valore di più variabili contemporaneamente

Ovviamente è possibile mostrare più di un valore o un qualunque mix di valori [10.3] in ogni chiamata a NSLog(). Per fare ciò ti devi assicurare d'indicare i tipi delle variabili (int, float) correttamente usando %d e %f.

```
//[10]
int x = 8;
float pi = 3.1416;
NSLog(@"Il valore intero è %d, il valore float è %f.", x, pi);    [10.3]
```

Far corrispondere i simboli ai valori

Uno dei più comuni errori commessi dai principianti è sbagliare le indicazioni sul tipo di dato in NSLog() e altre funzioni. Se i tuoi risultati sono strani o semplicemente il programma crasha senza ragione apparente, prova a dare un'occhiata ai tipi di dato!
Per esempio, se sbagli il primo tipo, il secondo dato potrebbe non essere visualizzato correttamente! Ad esempio:

```
//[10b]
int x = 8;
float pi = 3.1416;
NSLog(@"Il valore intero è %f, mentre il valore float è %f.", x, pi);

// Versione corretta: NSLog(@"Il valore intero è %d, mentre  il valore float
è %f.", x, pi);
```

da il seguente output:

```
Il valore intero è 0.000000, mentre  il valore float è 0.000000.
```

Collegarsi al framework Foundation

Abbiamo ancora una questione da risolvere prima di eseguire il nostro primo programma. Come fa il nostro programma a conoscere la funzione NSLog()?
Bene, non la conosce, a meno che non glielo diciamo noi! Per farlo il nostro programma dovrà chiedere al compilatore d'importare una libreria apposita (che fortunatamente è inclusa in ogni mac).

Questa richiesta al compilatore si fa attraverso la seguente istruzione:

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

Questa dev'essere la prima istruzione del nostro programma. Ora mettiamo insieme tutto quello che abbiamo imparato in questo capito e scriviamo il codice seguente

```
//[11]
#import <Foundation/Foundation.h>                          //[11.1]
float circleArea(float theRadius);
float rectangleArea(float width, float height);
int main()
```

```
{
    float pictureWidth, pictureHeight, pictureSurfaceArea,
    circleRadius, circleSurfaceArea;
    pictureWidth  = 8.0;
    pictureHeight = 4.5;
    circleRadius  = 5.0;
    pictureSurfaceArea = rectangleArea(pictureWidth, pictureHeight);
    circleSurfaceArea = circleArea(circleRadius);
    NSLog(@"Area del cerchio: %10.2f.", circleSurfaceArea);
    NSLog(@"Area del rettangolo: %f. “, pictureSurfaceArea);
    return 0;
}

float circleArea(float theRadius)              // prima funzione
personalizzata
{
    float theArea;
    theArea = 3.1416 * theRadius * theRadius;
    return theArea;
}

float rectangleArea(float width, float height) // seconda funzione
personalizzata
{
    return width*height;
}
```

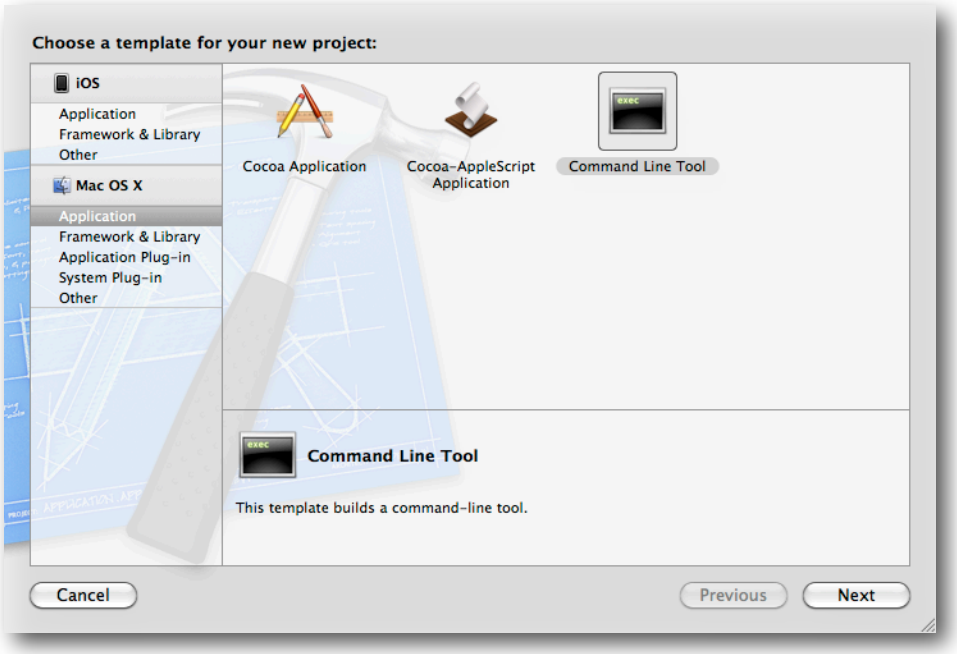
05: Compilare ed eseguire un programma

Introduzione

Il codice che abbiamo scritto fino ad ora non è altro che del testo che noi umani riusciamo a leggere. Pur non essendo intuitivo per noi, lo è ancora meno per il tuo Mac. Con esso non può farci nulla!
Serve un programma speciale, detto compilatore, per tradurre il codice di programmazione in un codice eseguibile dal Mac. Il compilatore è una parte dell'ambiente di programmazione Xcode. Dovresti aver installato Xcode usando il disco che ti è stato fornito con la tua copia di Mac OS X. In ogni caso verifica di aver l'ultima versione disponibile, che può essere scaricata alla sezione sviluppatori a http://developer.apple.com (richiesta registrazione gratuita).

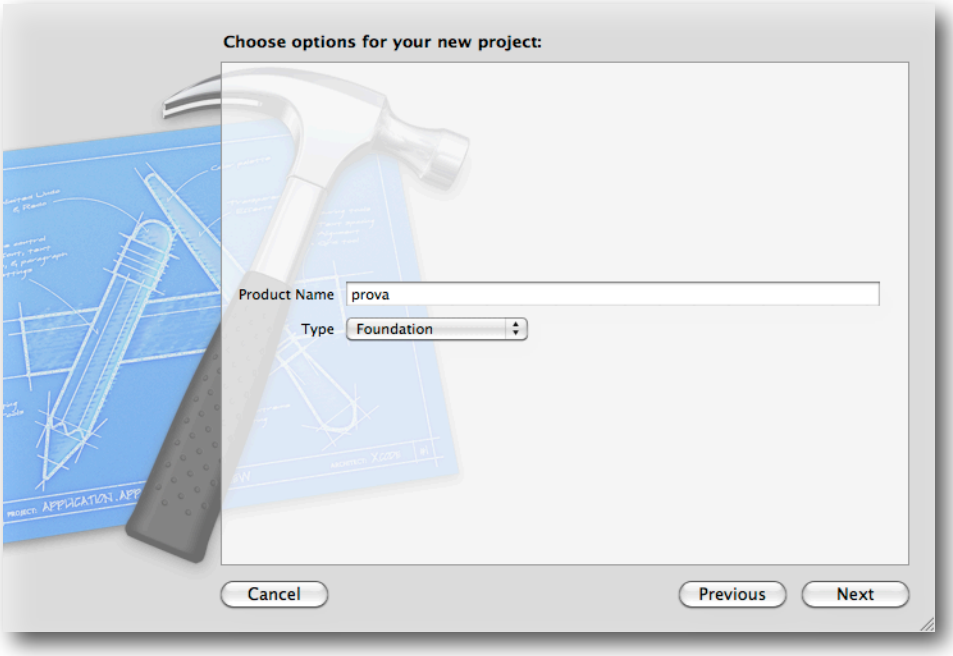
Creare un progetto

Avvia Xcode, che puoi trovare nella sottodirectory Applicazioni della directory Developer. Quando avvii il programma per la prima volta ti sarà presentata la finestra di Welcome dalla quale potrai lanciare i progetti più recenti, per il momento non la utilizziamo quindi scegli Cancel.
Ora seleziona "New>New Project" nel menu "File". Comparirà una finestra di dialogo con una lista di tutti i possibili tipi di progetto.



L'Assistente Xcode ti creerà il nuovo progetto.

La nostra intenzione è creare un programmino molto semplice in Objective-C, senza interfaccia grafica (GUI: Graphical User Interface), quindi seleziona la voce Command Line Tool nel gruppo Application della sezione Mac OS X.
Dai un nome alla tua applicazione, qualcosa tipo "prova" e seleziona Foundation nel menu Type, quindi premi il bottone Next e scegli una directory dove salvare il tuo progetto e clicca Create.

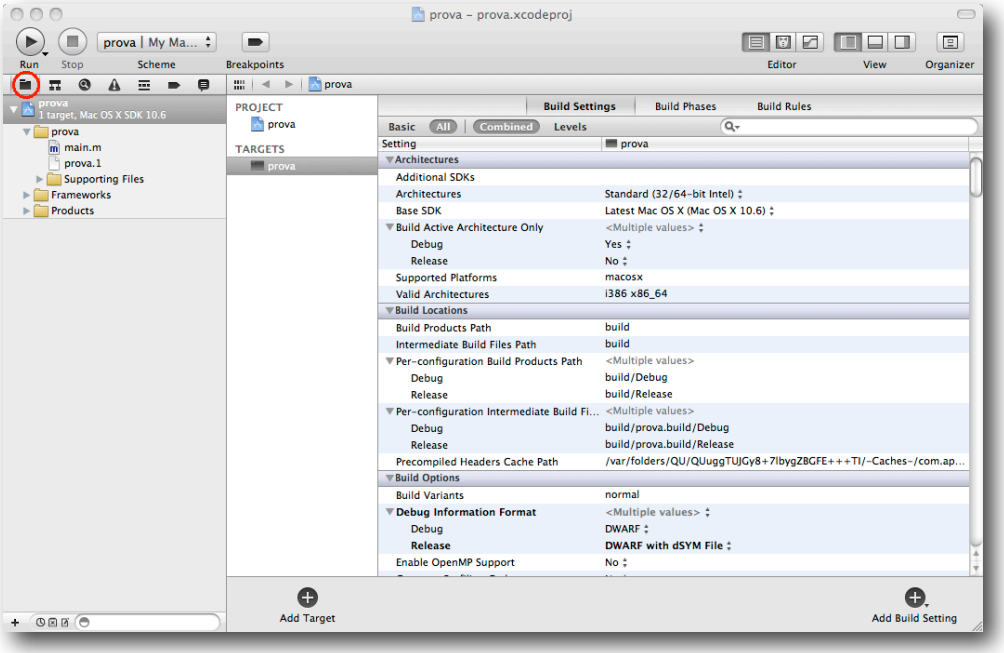


Impostare nome e directory per il nuovo progetto.

Il progetto che stiamo per creare potrà essere eseguito dal Terminale. Se vuoi essere in grado di eseguirlo evitandovi un po' di problemi assicurati di dare al tuo programma un nome di una sola parola. Inoltre è abitudine consolidata non iniziare con la maiuscola i nomi dei programmi da far eseguire da terminale. Al contrario i programmi dotati di GUI dovrebbero avere nomi che iniziano con lettera maiuscola.

Esplorando Xcode

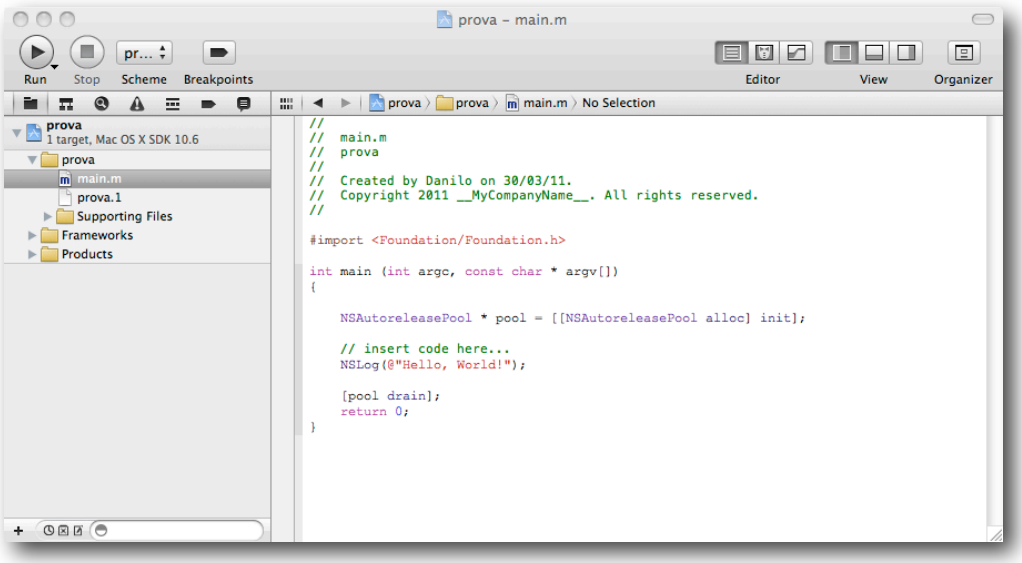
Dopo aver confermato la creazione del progetto ti si presenterà una finestra con molti elementi. La finestra è divisa in due sezioni.
Assicurati di aver selezionato la visualizzazione Groups & Files (vedi il cerchio rosso nell'immagine).



Con questa visualizzazione la sezione di sinistra elenca tutti i file di cui è composto il tuo progetto, permettendoti di accedere ad essi.

Attualmente non ce ne saranno molti ma tra poco, quando creeremo applicazioni multilingua con interfaccia grafica, potrai trovare qui parecchi file. I file sono raggruppati in cartelle, ma se cercherai queste cartelle nel tuo disco fisso non le troverai. Xcode suddivide i file del progetto in gruppi logici, per aiutarti a tenere organizzato il lavoro.

Nella sezione "Gruops & Files" apri il gruppo "prova" e naviga fino alla cartella prova. Al suo interno troverai un file chiamato "main.m" [1].
Ricordi che ogni programma deve contenere una funzione chiamata main() ?
Ecco, questo è il file che contiene quella funzione. Più avanti in questo capitolo andremo a modificarla per includere il codice del nostro programma. Se apri "main.m" selezionandolo da Group & Files, avrai una bella sorpresa.
Xcode ha già creato la funzione main() per te.



Xcode mostra la funzione main().

[1]

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

int main (int argc, const char * argv[])                    // [1.3]
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];    // [1.5]

  // insert code here...
  NSLog(@"Hello, World!");
  [pool drain];                                             //[1.9]
  return 0;
}
```

Ora guarda il programma e vedi cosa riesci a riconoscere. Se presti attenzione vedrai:

- La dichiarazione import necessaria per funzioni come NSLog(), che inizia con un cancelletto.
- La funzione `main()`.
- Le parentesi graffe che contengono il corpo del programma.
- Un commento, che ti invita ad inserire il tuo codice in quel punto.
- Una dichiarazione `NSLog()` per stampare una stringa sullo schermo.
- La dichiarazione `return 0;`.

Ci saranno anche alcune cose che non riconoscerai:

- Due strani argomenti tra parentesi alla funzione `main()` [1.3]
- Una dichiarazione che incomincia con `NSAutoreleasePool` [1.5]
- Un'altra dichiarazione con le parole `pool` e `drain` [1.9]

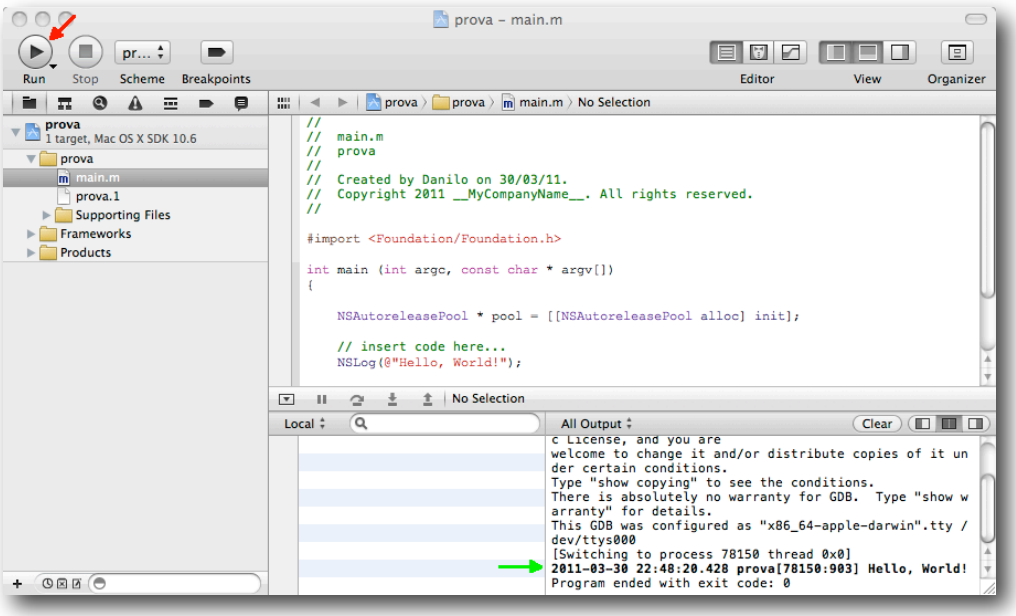
Personalmente non sono particolarmente felice quando l'autore di un libro si presenta, a me lettore, con del codice pieno di dichiarazioni e linee non familiari e mi promette che saranno spiegate successivamente. Questo è il motivo per cui abbiamo fatto tutto il percorso sul concetto di funzioni, così adesso non dovremo introdurre troppi nuovi concetti.

Sai già che le funzioni sono un modo per organizzare un programma, che ogni programma ha una funzione main() e come sono strutturate le funzioni. In ogni modo, devo ammettere che non è il momento di spiegarti tutto ciò che vedi nell'esempio [1].
Ti chiedo, quindi, d'ignorare quelle linee ([1.3, 1.5 e 1.9]). Ci sono altre cose del linguaggio Objective-C con cui devi familiarizzare prima di poter scrivere semplici programmi. La buona notizia è che hai già superato due grossi scogli e che i prossimi tre sono abbastanza semplici, prima di rituffarci nuovamente in qualcosa di complicato.

Se proprio non vuoi essere lasciato senza una spiegazione eccone una sommaria.
Gli argomenti della funzione main() sono necessari per eseguire il programma da terminale. Il tuo programma usa memoria. Memoria che altri programmi vorrebbero usare quando hai finito di far girare il programmino. Come programmatore è tuo dovere riservarti la memoria di cui hai bisogno. Altrettanto importante è liberare la memoria quando hai finito. Questo è ciò a cui serve la dichiarazione "pool".

Compilare ed eseguire

Eseguiamo ora il programma fornitoci da Apple [1] cliccando il pulsante "Run" (vedi la freccia rossa) per compilare ed eseguire l'applicazione.



Il bottone "Run".

Il risultato sarà che il programma verrà tradotto (compilato) in un linguaggio comprensibile dal Mac e quindi eseguito. Il risultato è stampato nella finestra "All Output", insieme ad altre informazioni aggiuntive. L'ultima frase dice che il programma è terminato, con un valore di ritorno pari a 0. Qui puoi vedere il valore di ritorno della funzione main(), come discusso nel capitolo 3 [7.9]. Quindi il nostro programma è giunto fino all'ultima riga e non è uscito prematuramente. Fin qui tutto OK!

Bugging

Torniamo ora all'esempio [1] e vediamo che succede se c'è un *bug* nel programma. Per esempio, modificando la dichiarazione NSLog() e dimenticandoci il punto e virgola finale.

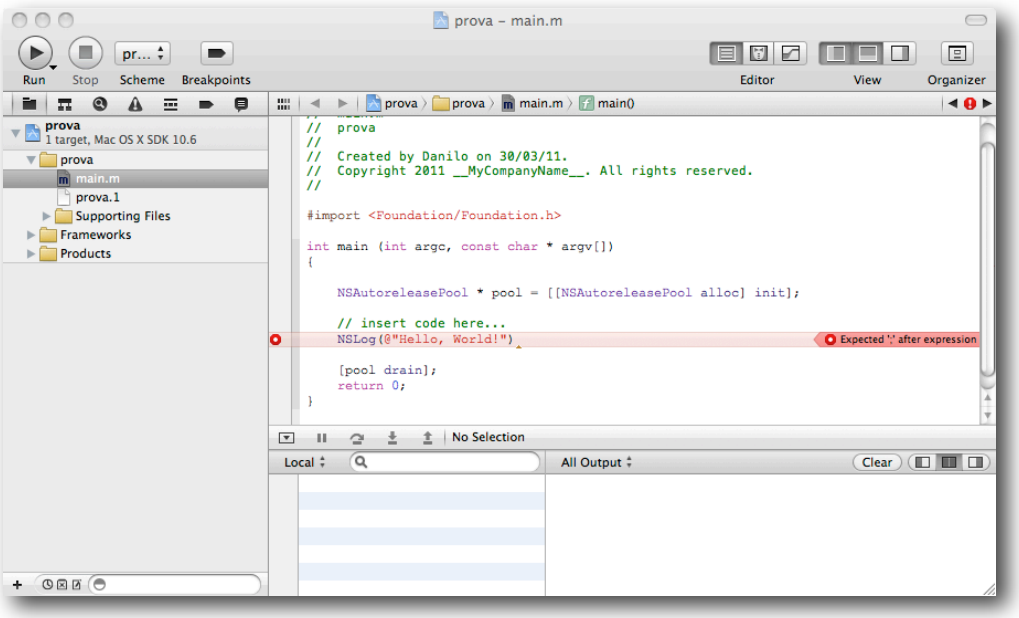
```
[2]
#import &lt;Foundation/Foundation.h&gt;

int main (int argc, const char * argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

  // insert code here...
  NSLog(@"Julia is my favourite actress")   //Whoops, forgot the semicolon!
```

```
  [pool drain];                                          //[2.9]
  return 0;
}
```

Nella finestra di editing viene segnalato l' errore che manca il ';'.
Il compilatore si accorge dell'errore mentre digiti il tuo programma (in-line parsing).



Xcode segnale un errore di compilazione

Se clicchi sul pallino a sinistra nella linea segnalata ti verrà proposta anche la correzione automatica (se il compilatore è in grado di proporla).

Il parsing è una delle prime cose che un compilatore fa, ovvero percorrere tutto il codice e controllare se può capire ogni singola linea. Il tuo dovere di programmatore è quello di aiutarlo a capire il significato di ciascuna parte.
Quindi per le dichiarazioni di import [2.1], dovrai inserire un cancelletto (#). Per indicare la fine di una linea [2.8] dovrai apporre un punto e virgola. Quando il compilatore raggiungerà la riga [2.9] noterà che manca un punto e virgola alla riga prima. La morale è che il compilatore cerca di darci dei *feedback* che abbiano senso, anche se non saranno un'accurata descrizione di quale sia il problema, nell'esatta posizione dell'errore.

Sistemiamo il programma aggiungendo il punto e virgola e compiliamo nuovamente per sincerarci che tutto funzioni.

La nostra prima Applicazione

Prendiamo adesso il codice dal capitolo precedente e incolliamolo dentro il codice fornito da Apple [1], il risultato sarà il seguente [3].

```
[3]
#import <Foundation/Foundation.>

float circleArea(float theRadius);                                    // [3.3]
float rectangleArea(float width, float height);                       // [3.4]

int main (int argc, const char * argv[])                              // [3.6]
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  int pictureWidth;
  float pictureHeight, pictureSurfaceArea,
        circleRadius, circleSurfaceArea;
  pictureWidth = 8;
  pictureHeight = 4.5;


  circleRadius = 5.0;
  pictureSurfaceArea =  pictureWidth * pictureHeight;
  circleSurfaceArea = circleArea(circleRadius);
  NSLog(@"Area of picture: %f.  Area of circle: %10.2f.",
        pictureSurfaceArea, circleSurfaceArea);
  [pool drain];
  return 0;
}

float circleArea(float theRadius)                                      // [3.22]
{
  float theArea;
  theArea = 3.1416 * theRadius * theRadius;
  return theArea;
}

float rectangleArea(float width, float height)                        // [3.29]
```

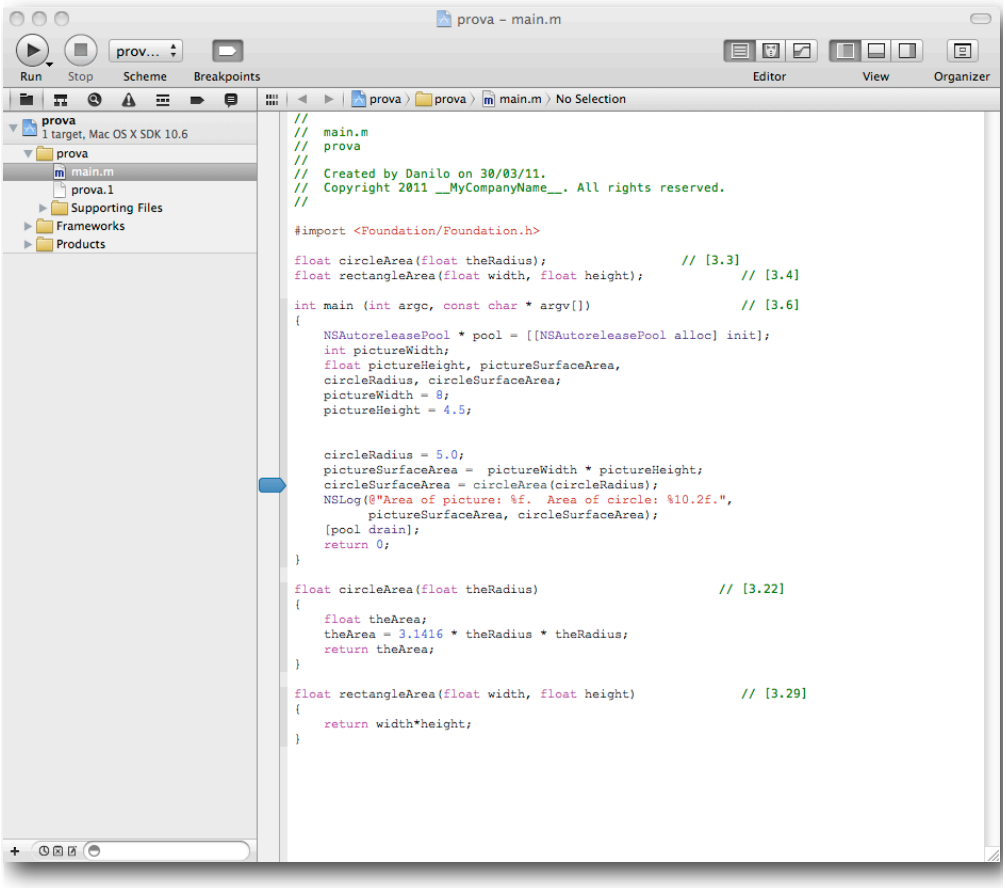
```
{
  return width*height;
}
```

Prenditi il tempo di capire la struttura del programma. Abbiamo le intestazioni delle funzioni [3.3, 3.4], prima della funzione *main()* [3.6] esattamente dove dovrebbero essere e le funzioni personalizzate *circleArea()* [3.22] e *rectangleArea()* [3.29], che si trovano al di fuori delle parentesi graffe della funzione *main()* [3.6].
Quando il codice viene eseguito otteniamo il seguente output:

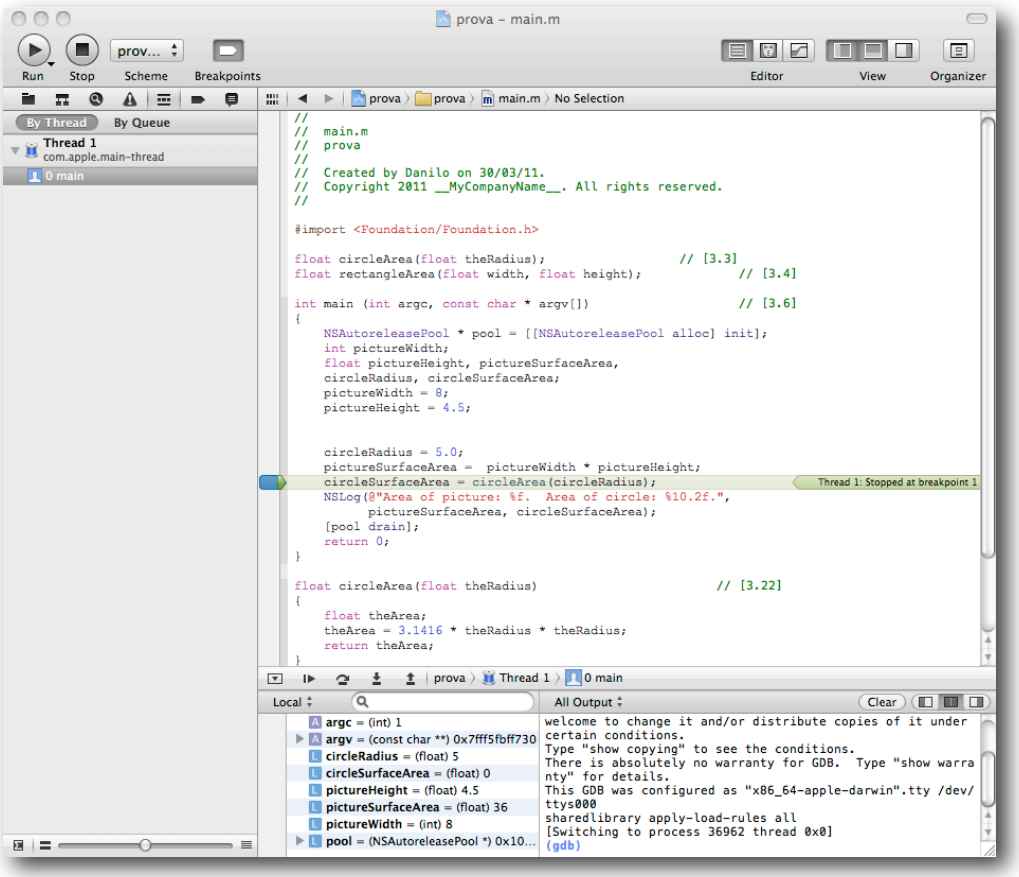
Area of picture: 36.000000. Area of circle: 78.54.

Debugging

Quando un programma si complica, diventa più difficile eliminare gli errori. Verrà quindi il momento che vorrai sapere cosa succede all'interno del programma mentre viene eseguito.
Xcode rende questo molto semplice. Clicca sul margine grigio prima della dichiarazione di cui vuoi conoscere il valore delle variabili; Xcode inserirà un punto d'interruzione (*breakpoint*), rappresentato dall'icona di una freccia blu.



Nota che vedrai il valore di quelle variabili, prima che quella riga di codice venga eseguita, quindi spesso avrai bisogno d'inserire il punto d'interruzione sulla riga di codice immediatamente successiva a quella in cui sei interessato.
Ora clicca sul tasto "Run" ed apparirà la seguente finestra.



Il debugger di Xcode ti permetterà di eseguire il programma riga per riga e di controllare le variabili.

Il programma continuerà fino a che non raggiungerà il primo punto d'interruzione (*breakpoint*).
Se controlli nel riquadro in basso a sinistra, vedrai i valori delle variabili. Per controllare l'esecuzione tramite il debugger usa i comandi della barra del debugger sopra il riquadro delle variabili.
Il debugger è uno strumento molto potente. Provalo per prenderci confidenza.

Conclusione

Abbiamo visto tutto quello che serve per scrivere, correggere ed eseguire semplici programmi per Mac OS X.
Se non sei intenzionato a creare programmi con interfacce grafiche, l'unica cosa che devi fare adesso è di approfondire la tua conoscenza di Objective-C, per

poter sviluppare programmi più sofisticati. Nei prossimi capitoli faremo esattamente questo. Dopo di che ci getteremo sulle applicazioni con interfaccia grafica. Avanti!

06: Espressioni condizionali

if()

A volte vorrai che il tuo codice faccia una serie di azioni solo se una particolare condizione è soddisfatta. A questo scopo, ci vengono messe a disposizione delle "parole chiave" speciali [1].

```
//[1]
// age è una variabile di tipo int che memorizza l'età dell'utente
if (age > 30)    // Il simbolo > (maggiore) significa "più grande di" //[1.2]
{
    NSLog(@"Hai più di 30 anni.");                                    //[1.4]
}
NSLog(@"Fine.");                                                      //[1.6]
```

La linea [1.2] mostra l'istruzione `if()`, conosciuta anche come un'istruzione condizionale. Riconoscerai le parentesi graffe, che contengono tutto il codice che vuoi venga eseguito, se l'espressione logica tra le parentesi () risulta vera. In questo caso, se la condizione `age > 30` sarà soddisfatta, allora la stringa [1.4] sarà visualizzata. Che la condizione sia soddisfatta o meno, la stringa alla linea [1.6] sarà visualizzata comunque, perchè esterna alle parentesi graffe della proposizione `if()`.

if() else()

Possiamo anche fornire un insieme alternativo d'istruzioni nel caso la condizione non sia soddisfatta, usando l'espressione if...else [2].

```
//[2]
// age è una variabile di tipo int che memorizza l'età dell'utente
if (age > 30)                                                         //[2.2]
{
    NSLog(@"Hai più di 30 anni.");                                    //[2.4]
}
else
{
    NSLog(@"Hai meno di 30 anni.");                                   //[2.7]
}
NSLog(@"Fine.");
```

La stringa nell'espressione [2.7] verrebbe visualizzata solo se la condizione non fosse soddisfatta, ovvero age più piccolo di 30 [2].

Confronti

Oltre al segno di maggiore (>) [1.2][2.2], sono a tua disposizione i seguenti operatori di confronto per i numeri:

==	uguale a
>	maggiore di
<	minore di
>=	maggiore o uguale a
<=	minore o uguale a
!=	non uguale a

Presta particolare attenzione all'operatore di uguaglianza == è composto da due segni di uguale. È facile dimenticarsene e usare semplicemente un segno di uguale. Purtroppo quest'ultimo è un operatore di assegnamento, e setterebbe la variabile ad un particolare valore. Questo è fonte di confusione, e di codice con errori, per i principianti. Ora di ad alta voce: non dimenticherò di usare due segni di uguale per i test di uguaglianza!
Gli operatori di confronto sono piuttosto utili quando vuoi ripetere una serie di espressioni diverse volte, argomento che tratteremo nel prossimo capitolo. Prima discuteremo qualche altro aspetto dell'espressione if che può capitare per le mani.

Esercizio

Diamo un'occhiata più da vicino all'eseguire dei confronti. Un'operazione di confronto ha solo due possibili risultati: vero oppure falso.

In Objective-C, vero e falso sono rappresentati come 1 o 0 rispettivamente. Esiste anche un tipo di dato speciale, chiamato BOOL che puoi usare per rappresentare questi valori. Per rappresentare il valore "vero" puoi scrivere 1 o YES. Per rappresentare il valore "falso" puoi scrivere 0 o NO.

```
//[3]
int x = 3;
BOOL y;
y = (x == 4);  // y sarà 0.
```

È possibile verificare più condizioni. Se più di una condizione dev'essere soddisfatta usa l'espressione logica AND, rappresentata da due &&.

Se almeno una delle condizioni dev'essere soddisfatta usa un'espressione logica OR rappresentata da due ||.

```
//[4]
if ( (age >= 18) && (age < 65) )
{
   NSLog(@"Hai da lavorare ancora una vita");
}
```

È anche possibile nidificare espressioni condizionali. È semplicemente questione di mettere un'espressione condizionale dentro le parentesi graffe di un'altra espressione condizionale. Prima la condizione più esterna viene valutata, se è soddisfatta, si passa a quella più interna, e così via:

```
//[5]
if (age >= 18)
{
    if (age < 65)
    {
        NSLog(@" Hai da lavorare ancora una vita");
    }
}
```

07: Ripetizione di espressioni (for a While)

Introduzione

In tutto il codice che abbiamo discusso finora, ogni espressione veniva eseguita solo una volta. Possiamo sempre ripetere del codice nelle funzioni invocandolo ripetutamente [1].

```
//[1]
NSLog(@"Julia è la mia attrice preferita.");
NSLog(@"Julia è la mia attrice preferita.");
NSLog(@"Julia è la mia attrice preferita.");
```

D'altra parte questo richiederebbe di ripetere la chiamata. A volte, vorrai eseguire una o più espressioni diverse volte. Come tutti i linguaggi di programmazione, Objective-C offre diverse possibilità per raggiungere questo scopo.

for()

Se conosci quante volte dev'essere ripetuta un'espressione (o un gruppo di espressioni), puoi specificarlo inserendo quel numero nell'espressione for dell'esempio [2].

```
//[2]
int x;
for (x = 1 ; x <= 10 ; x++)
{
    NSLog(@"Julia è la mia attrice preferita.");                    //[2.4]
}
NSLog(@"Il valore di x è %d", x);                                   //[2.6]
```

Nell'esempio [2], la stringa [2.4] verrebbe visualizzata 10 volte. Prima cosa, ad x è assegnato il valore 1. Il computer allora valuta la condizione con la formula che abbiamo messo: x <= 10. Questa condizione è soddisfatta (dal momento che x è uguale a 1), quindi l'espressione o espressioni tra parentesi graffe sono eseguite. Quindi, il valore di x è incrementato, in questo caso di uno, grazie alla espressione x++. Successivamente, il valore risultante di x, ora 2, è confrontato con 10. Siccome è ancora minore o uguale a 10, le espressioni tra parentesi graffe sono eseguite di nuovo. Appena x è 11, la condizione x <= 10 non è più

soddisfatta. L'ultima espressione [2.6] è stata inclusa per provarti che x è 11, non 10, dopo che il loop è finito.
A volte, avrai bisogno di fare step più larghi che solo un semplice incremento usando x++. Tutto quello che devi fare è sostituire con l'espressione che ti serve. L'esempio seguente [3] converte i gradi Fahrenheit in gradi Celsius.

```
//[3]
float celsius, tempInFahrenheit;
for (tempInFahrenheit = 0 ; tempInFahrenheit <= 200 ; tempInFahrenheit =
tempInFahrenheit + 20)
{
    celsius = (tempInFahrenheit - 32.0) * 5.0 / 9.0;
    NSLog(@"%10.2f -> %10.2f", tempInFahrenheit, celsius);
}
```

L'output di questo programma è:

```
0.00 -> -17.78
20.00 -> -6.67
40.00 ->  4.44
60.00 -> 15.56
80.00 -> 26.67
100.00 -> 37.78
120.00 -> 48.89
140.00 -> 60.00
160.00 -> 71.11
180.00 -> 82.22
200.00 -> 93.33
```

while()

Objective-C ha due altri modi per ripetere un set di espressioni:

```
while () { }
```

e

```
do {} while ()
```

La prima è in sostanza identica al loop for che abbiamo discusso sopra. Inizia eseguendo la valutazione di una condizione. Se il risultato della valutazione è falso, le espressioni nel loop non sono eseguite.

```
//[4]
int counter = 1;
while (counter <= 10)
{
    NSLog(@"Julia è la mia attrice preferita.\n");
    counter = counter + 1;
}
NSLog(@"Il  valore di counter è %d", counter);
```

In questo caso, il valore del contatore è 11, ne avrai bisogno più tardi nel tuo programma!
Con l'istruzione do {} while () , le espressioni tra parentesi graffe sono eseguite come minimo una volta.

```
//[5]
int counter = 1;
do
{
    NSLog(@"Julia è la mia attrice preferita.\n");
    counter = counter + 1;
}
while (counter <= 10);
NSLog(@"Il  valore di counter è %d", counter);
```

Il valore del contatore alla fine è 11.

Hai acquisito altre capacità di programmazione, quindi ora affrontiamo un argomento più difficile. Nel prossimo capitolo costruiremo il nostro primo programma con una Graphical User Interface (GUI).

08: Un programma con una GUI

Introduzione

Abbiamo aumentato le nostre conoscenze di Objective-C, ed ora siamo pronti per discutere su come creare un programma con una Graphical User Interface (GUI). Devo confessarvi una cosa. L' Objective-C è un'estensione del linguaggio di programmazione chiamato C. Tutto quello di cui abbiamo discusso precedentemente è principalmente C. In cosa differisce Objective-C dal C? Nella parte "objective"; Objective-C tratta di nozioni astratte conosciute come oggetti. Fino ad ora abbiamo principalmente trattato coi numeri. Come già sai, Objective-C nativamente supporta il concetto di numeri. Ciò ci permette di creare e manipolare numeri in memoria usando operatori matematici e funzioni matematiche. Questo è ottimo quando la tua applicazione tratta numeri (ad esempio una calcolatrice). Ma se la tua applicazione tratta cose tipo canzoni, playlist, artisti ecc...? O se la tua applicazione controlla il sistema del traffico aereo trattando con aerei, voli, aeroporti, etc.? Non sarebbe carino essere capaci di manipolare anche queste cose con Objective-C, facilmente come facciamo con i numeri? Questo è dove gli oggetti entrano in gioco. Con Objective-C, puoi definire il tipo di oggetto che ti interessa trattare e poi scrivere l'applicazione che lo manipola.

Oggetti in azione

Come esempio, prendiamo in considerazione come una finestra viene gestita da un programma scritto in Objective-C, quale è Safari. Prendiamo una finestra aperta di Safari sul tuo Mac.
In alto a sinistra, ci sono tre bottoni, di cui quello rosso è il bottone di chiusura. Cosa accade se chiudi la finestra cliccando il bottone rosso? Un messaggio viene spedito alla finestra e, in risposta, la finestra esegue il codice per chiudersi.

Un messaggio di chiusura è trasmesso alla finestra

La finestra è un oggetto. Tu puoi trasportarla in giro; anche i tre bottoni sono oggetti. Tu puoi cliccarli. Questo tipo di oggetti hanno una rappresentazione visuale sullo schermo, ma non è così per tutti. Per esempio, l'oggetto che rappresenta la connessione tra Safari e un dato sito internet non ha una rappresentazione sullo schermo.

Un oggetto (ad esempio la finestra) può contenere altri oggetti (ad esempio i bottoni)

Classi

Se ci pensi puoi avere quante finestre vuoi di Safari; allora i programmatori Apple:

- programmarono ogni finestra in anticipo, usando la loro massima potenza celebrale per prevedere quante finestre tu volevi aprire?
- Fecero un tipo di template e lasciarono che Safari potesse creare lo oggetto finestra al momento?

Sicuramente la seconda. Hanno creato il codice, nominato la classe che definisce cos'è una finestra e come deve apparire e comportarsi.

Quando crei una nuova finestra, è la classe che crea la finestra per te. Questa classe rappresenta il concetto di finestra, e ogni particolare della finestra è un'istanza di questo concetto (nella stessa maniera 76 è un'istanza del concetto di numero).

Variabili istanziate

La finestra che hai creato è presente in un certo punto dello schermo del tuo Mac. Se la minimizzi nel Dock, e la fai riapparire, essa si riporterà nella stessa posizione sullo schermo che aveva in precedenza. Come riesce questo lavoro?
La classe definisce variabili adeguate per ricordarsi la posizione della finestra sullo schermo. L'istanza della classe, ad esempio l'oggetto, contiene i valori di queste variabili. Così, ogni oggetto finestra contiene i valori di certe variabili, e gli oggetti finestra differenti conterranno differenti valori per queste variabili.

Metodi

La classe non solo ha creato l'oggetto finestra, ma dà anche l'accesso a una serie di azioni che può eseguire.
Una di queste azioni è “chiudi”. Quando tu clicchi il bottone di "chiusura" di una finestra, il bottone spedisce il messaggio chiudi a quella finestra oggetto. Le azioni che possono essere eseguite da un oggetto sono chiamati metodi, e come puoi ben vedere assomigliano molto alle funzioni, quindi non dovresti avere problemi ad imparare ad usarli, sempre se ci hai seguito.

Oggetti in memoria

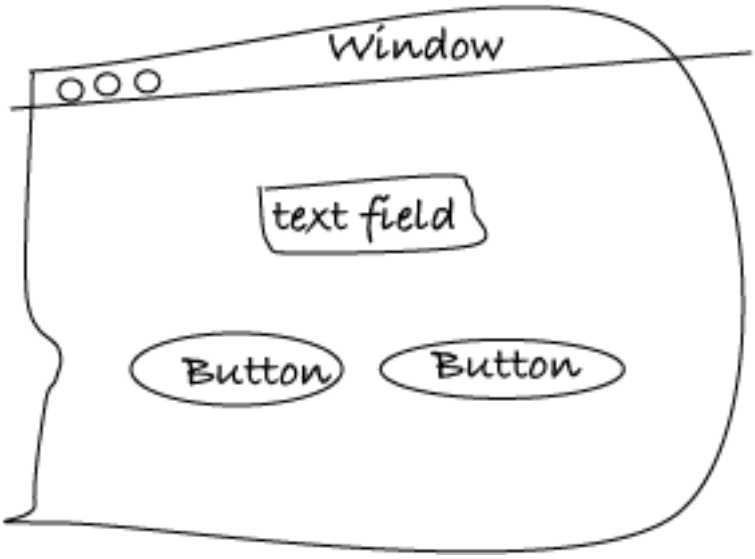
Quando la classe crea un oggetto finestra per te, riserva della memoria (RAM) per immagazzinare la posizione della finestra e tante altre informazioni, ma non fa una copia del codice per chiudere la finestra. Questo potrebbe sembrare un errore ma non è così. Il codice è lo stesso per ogni finestra e non ha bisogno di essere presente tante volta, ma una sola; questo perché ogni oggetto finestra ha accesso a questo codice nel momento in cui crei una nuova classe finestra. Il codice che vedrai in questo capitolo contiene alcune istruzioni per riservare dello spazio in memoria e rilasciarlo poi al sistema. Si tratta di un argomento avanzato che per il momento non tratteremo.

ù

Esercizio

La nostra applicazione

Il nostro obiettivo è di creare un'applicazione con due bottoni ed un campo di testo. Alla pressione di uno dei bottoni, il campo di testo verrà riempito con un valore diverso a seconda del bottone premuto. Pensa a una calcolatrice con due bottoni che però non può fare calcoli. Sicuramente, una volta che avrai imparato abbastanza potrai provare a creare una calcolatrice vera, ma procediamo a piccoli passi.



Uno schizzo dell'applicazione che vogliamo creare

Se uno dei bottoni della nostra applicazione viene premuto, manderà un messaggio. Il messaggio conterrà il nome del metodo da eseguire. Il messaggio è inviato a... a cosa? Nel caso della finestra, il messaggio di chiusura viene spedito all'oggetto finestra, che è un'istanza della classe finestra. Quindi ciò di cui abbiamo bisogno ora, è un oggetto che sia capace di ricevere un messaggio da ognuno dei due bottoni, e poter dire all'oggetto campo testo (text field) di visualizzare un valore.

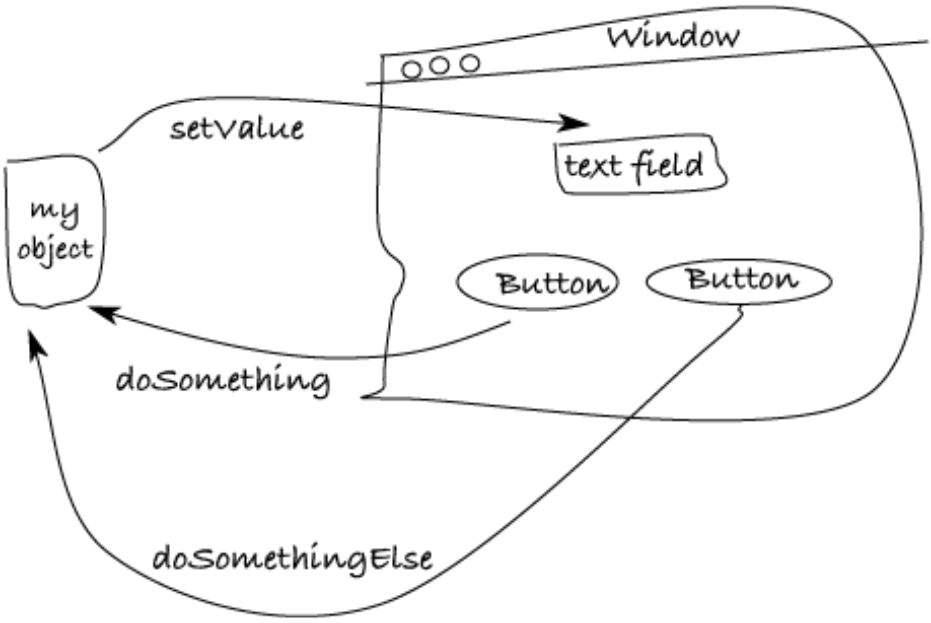
La nostra prima classe

Quindi, noi prima dobbiamo creare la nostra classe, e poi realizzare un'istanza di quest'ultima. Questo oggetto sarà il destinatario dei messaggi dai bottoni (fai

riferimento all'immagine nella pagina successiva). Così come un qualsiasi oggetto finestra, anche la nostra istanza è un oggetto, ma contrariamente ad un oggetto finestra non appare sullo schermo quando il programma viene eseguito, è solo qualcosa nella memoria del Mac.

Quando la nostra istanza riceve un messaggio mandato da uno dei due bottoni viene eseguito il metodo appropriato. Il codice del metodo è memorizzato nella classe (non nell'istanza stessa). Durante l'esecuzione, questo metodo imposterà il testo nell'oggetto campo testo.

Come fa il metodo nella nostra classe a sapere come impostare il testo nel campo testo? Non lo sa; ma il campo testo sa cosa fare con il suo contenuto. Così noi spediamo un messaggio al campo testo, chiedendogli di farlo. Quale tipo di messaggio dovrebbe essere? Sicuramente, abbiamo bisogno di specificare il nome del contenitore (nell'esempio l'oggetto campo testo nella nostra finestra). Poi avremo bisogno di dire, nel messaggio, cosa vogliamo che il contenitore faccia. Per farlo useremo il nome del metodo che il campo testo deve eseguire al ricevimento del messaggio. Infine abbiamo bisogno di comunicare all'oggetto campo testo quale valore mostrare (relativamente al bottone cliccato). Quindi il messaggio spedito esprime non solo il nome dell'oggetto e del metodo, ma anche un argomento (valore) da usare nel metodo dell'oggetto campo testo.



Uno schizzo dei messaggi scambiati tra gli oggetti della nostra applicazione

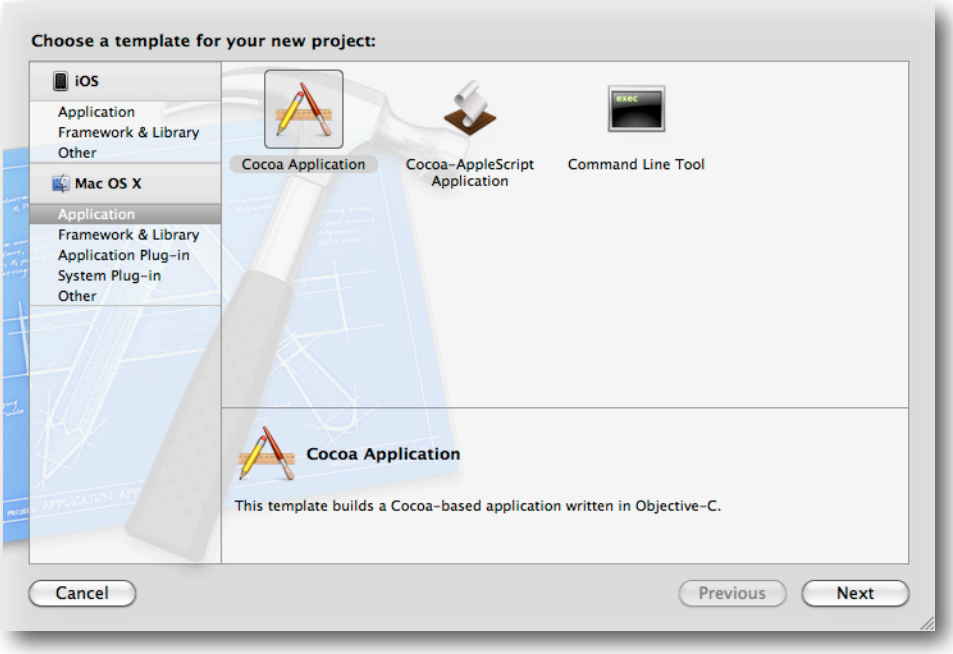
Qui c'è il formato generale di come si spediscono messaggi in Objective-C, con [1.1] o senza [1.2] argomento:

```
//[1]
[oggetto_ricevente messaggio];                          //[1.1]
[oggetto_ricevente messaggio:argomento_del_messaggio];  //[1.2]
```

Come puoi vedere in ognuna di queste dichiarazioni, il tutto è messo tra le parentesi quadre ed il classico punto e virgola è presente come tocco finale. Tra le parentesi, l'oggetto ricevente è menzionato prima, seguito dal nome del suo rispettivo metodo. Se il metodo richiedesse uno o più valori, questi troverebbero posto dopo il segno : alla fine del messaggio. [1.2]

Creiamo il progetto

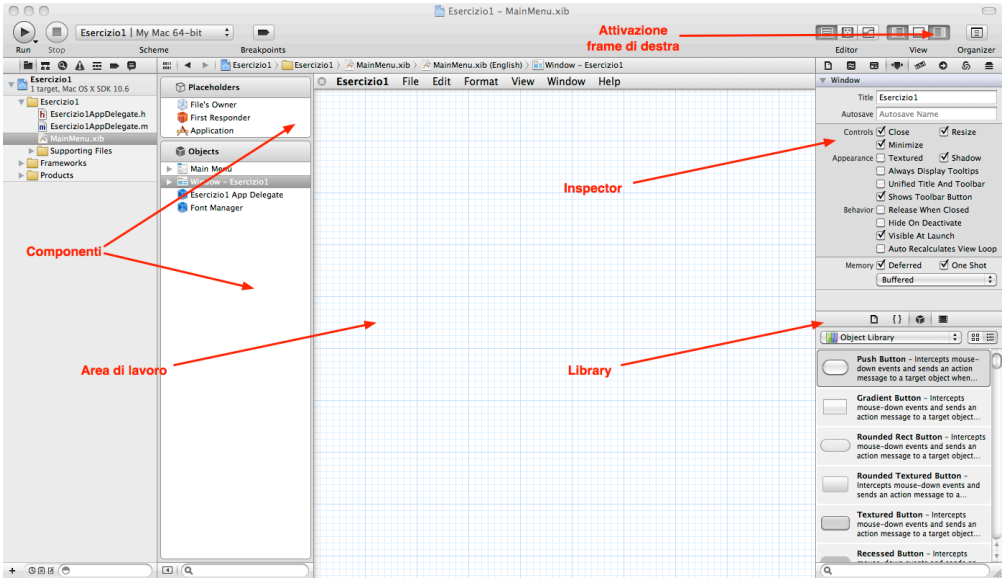
Vediamo adesso come, nella pratica, si può fare tutto quello che abbiamo appena spiegato. Avvia Xcode per creare un nuovo progetto e selezionia Cocoa Application



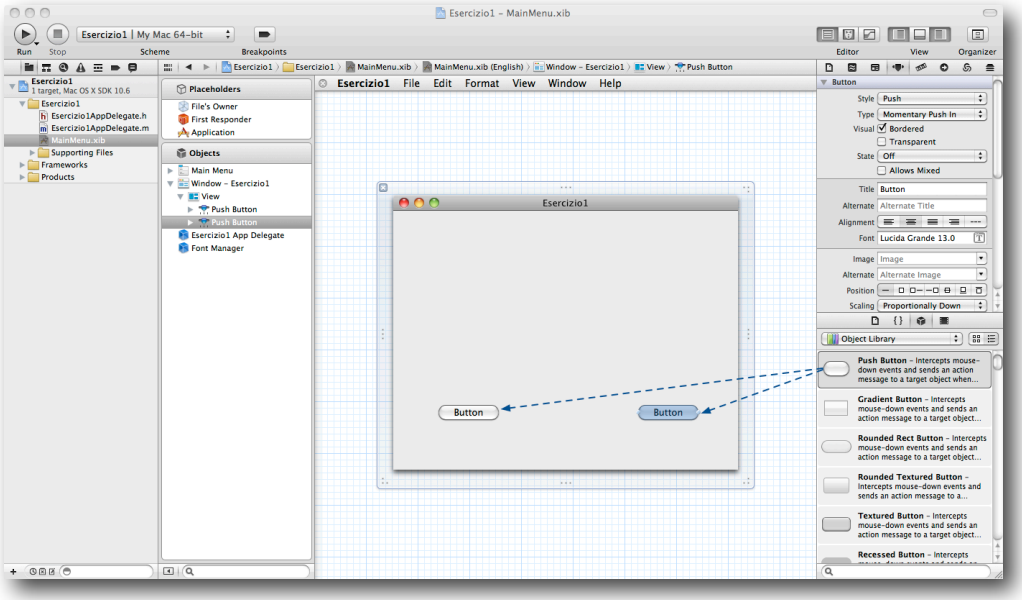
Dai quindi un nome al progetto (convenzionalmente il nome della tua applicazione dovrebbe iniziare con una maiuscola, es: Esercizio1) nel campo

Product Name e lascia le altre opzioni come sono e salva. Creato il progetto, seleziona MainMenu.xib, che si trova sotto Groups & Files, nella cartellina con il nome del progetto (Esercizio1).

Creiamo la GUI



Selezionando un file xib si entra nell'editor dell'interfaccia grafica.
Oltre alla sezione Groups & Files (già presente) puoi individuare una sezione principale di Area di lavoro ed una colonna alla sua sinistra dove sono elencate le componenti dell'interfaccia sia in termini di Oggetti che di PlaceHolders (per ora accetta questi nomi, man mano che ne avremo bisogno analizzeremo i diversi elementi).
Attivando il frame di destra con il bottone in alto a destra, si apre una colonna con due sezioni, l'inspector per operare sulle proprietà di un oggetto selezionato e la Library per creare nuovi oggetti di interfaccia.
Scorri la lista degli oggetti disponibili fino a trovare l'oggetto "Push Button" e trascinalo due volte nella finestra della applicazione.



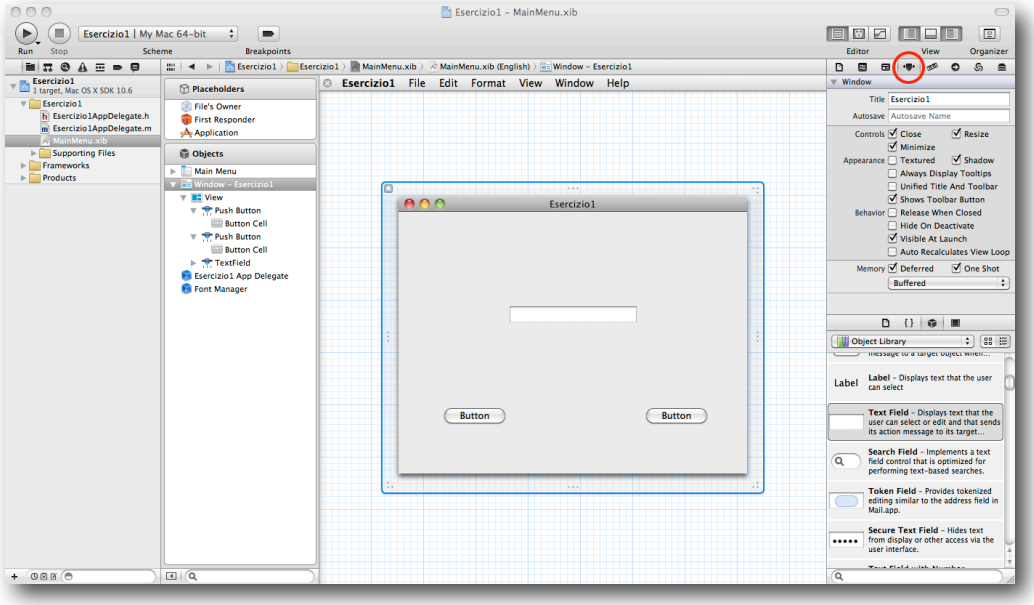
Ora cerca l'oggetto chiamato "Label" e allo stesso modo dei pulsanti trascina anche lui nella finestra.

L'azione di trascinare un bottone dalla Library alla finestra dell'applicazione, crea un nuovo oggetto bottone e lo mette nella tua finestra. Lo stesso accade per il campo testo ed ogni altro oggetto che potresti trascinare dalla Library alla finestra dell'applicazione.
Nota che se selezioni e tieni il cursore fermo sopra un'icona nella Library, comparirà una finestra con una breve descrizione ed un nome, tipo NSButton o NSTextView. Questi sono i nomi delle classi fornite da Apple. In seguito, in questo capitolo, vedremo come possiamo trovare i metodi delle classi, di cui abbiamo bisogno per mettere in azione il nostro programma.
Sistema a piacimento gli oggetti trascinati sulla finestra "Esercizio1". Cambia il testo degli oggetti bottone, prima uno poi l'altro, cliccando due volte sopra di essi.

Ti consiglio di esplorare la Library, dopo che abbiamo finito questo progetto, per prendere familiarità su come aggiungere altri oggetti alla finestra.

Esploriamo Interface Builder

Per cambiare le proprietà di un oggetto, seleziona l'oggetto e guarda la sezione Inspector.
Per esempio, seleziona la finestra "Esercizio1" (puoi verificare che è selezionata nella sezione Objects) e apri l'Inspector degli Attributi con il selettore evidenziato in rosso in figura.



A questo punto puoi spuntare Textured, e questo dà alla tua applicazione un look metallico. Ti accorgerai di poter personalizzare la applicazione senza bisogno di una riga di codice!

Classe background

Come dicevamo prima, ora dobbiamo creare una classe. Ma prima di farlo andiamo un po' più a fondo su come lavorano le classi. Per salvarti da inutili sforzi di programmazione, sarebbe bello se potessi costruire la tua classe su una base già esistente, invece che partire da zero. Se tu per esempio, vuoi creare una finestra con speciali proprietà (capabilities), hai bisogno di aggiungere solamente il codice di queste proprietà, non di riscrivere codice per tutto, tipo come minimizzare o chiudere una finestra. Costruendo su quello che altri programmatori hanno fatto, puoi inserire tutti questi nuovi comportamenti

(proprietà) liberamente. E questo è ciò che Objective-C fa a differenza del semplice C.
Come avviene questo? Bene, esiste una classe finestra (NSWindow), che se utilizzata per creare un altra classe, quest'ultima ne eredita tutte le proprietà e metodi. Supponiamo che aggiungi alcune caratteristiche alla tua classe finestra. Cosa accade se la tua finestra speciale riceve un messaggio "chiudi"? Nella realtà tu non hai né scritto , né copiato alcun codice per fare ciò. Semplice, se la classe della finestra specializzata (quella creata da te) non contiene il codice per un particolare metodo, il messaggio è trasferito automaticamente alla classe superiore dalla quale eredita; ecco perché la classe dalla quale la nostra eredita viene chiamata superclasse. E se necessario, questo andrà avanti finché il metodo viene trovato o si raggiunge il massimo della gerarchia di eredità, che nel caso di Objective-C è la classe NSObject, da cui derivano tutte le altre classi.

Se il metodo non può essere trovato, hai spedito un messaggio che non può essere gestito. È come richiedere ad un'officina di cambiare le gomme del tuo slittino. Anche il capo dell'officina non potrà mai aiutarti. In questo caso Objective-C segnalerà un errore.

Custom classes

E se volessimo implementare delle caratteristiche particolari per un metodo ereditato dalla superclasse? In questo caso dobbiamo sovrascrivere il metodo della superclasse. Per esempio, puoi scrivere del codice che, cliccando il tasto di chiusura, muove la finestra fuori dalla tua vista prima che si chiuda. La tua classe speciale per la finestra potrebbe usare lo stesso metodo nominato per chiudere una finestra come lo definisce Apple. Così quando la tua speciale finestra riceve un messaggio di chiusura, il metodo eseguito è il tuo, e non quello Apple. Quindi, ora la finestra si muoverà fuori lato, prima di chiudersi realmente.
In realtà il metodo per chiudere una finestra, è già stato scritto da Apple. All'interno del nostro metodo di chiusura, noi possiamo anche invocare il metodo di chiusura implementandolo dalla nostra superclasse, sebbene ciò richieda una leggera differenza nella chiamata, per essere sicuri che il nostro metodo di chiusura non sia ricorsivo. [2]

```
//[2]
// Codice per muovere la finestra fuori dalla vista.
[super close]; // Usa il metodo close della superclasse.
```

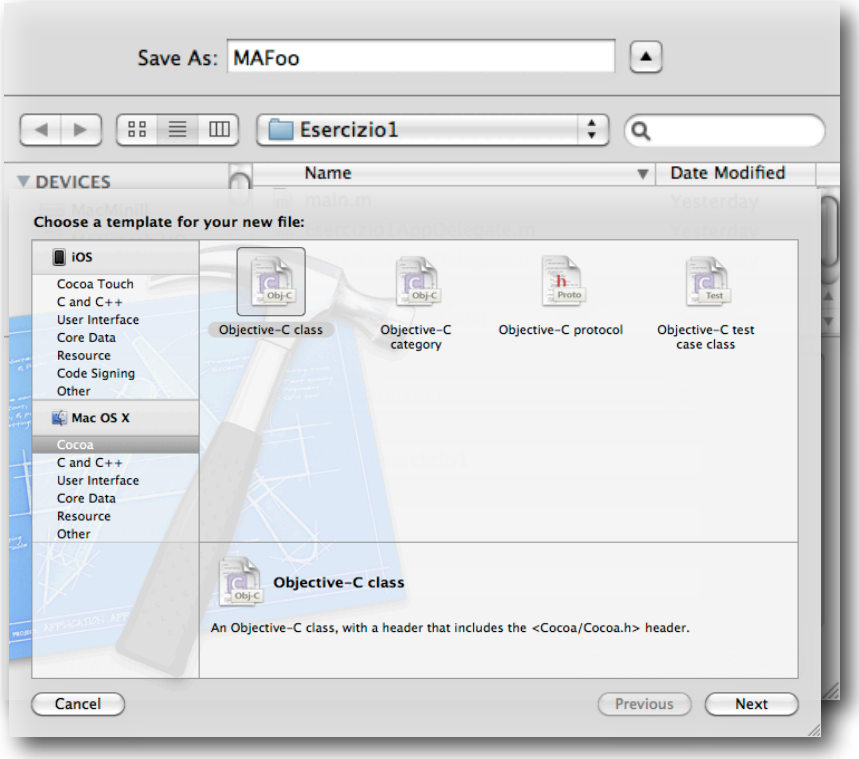
Questo è un argomento troppo avanzato per un testo introduttivo, e non ci aspettiamo di spiegarla con queste poche righe di codice.

Una classe per controllare tutte le altre

La madre di tutte le classi è la classe NSObject. Sostanzialmente tutte le classi che creerai o userai sono sottoclassi di NSObject, direttamente o indirettamente. Per esempio la classe NSWindow è una sottoclasse della NSResponder, la quale a sua volta è sottoclasse di NSObject. La classe NSObject definisce i metodi comuni di tutti gli oggetti (ad esempio generare una descrizione testuale di un oggetto, richiedere ad un oggetto se è pronto a ricevere un dato messaggio etc...)
Prima che ti annoi troppo con la teoria vediamo come creare una classe.

Creiamo la nostra classe

Torniamo ad Xcode e selezioniamo New file dal File menu. Scegliamo un classe Objective-C dalla lista, poi clicchiamo Next.
Seleziona NSObject come superclasse clicca Next e nella finestra di salvataggio chiamiamola "MAFoo", quindi clicca su Save.



Creiamo la MAFoo Class

Le prime due lettere di MAFoo stanno per My Application. Puoi inventarti il nome della classe come più ti piace. Una volta iniziato a scrivere la nostra applicazione, mi raccomando di scegliere due o tre lettere come prefisso che userai per tutte le classi onde evitare confusione con il nome delle classi esistenti. Comunque, non usare NS, potrebbe causare confusione. NS viene usato per le classi Apple e sta per NextStep, società fondata da Steve Jobs, poi comprata da Apple, mamma del linguaggio OBJ-C.

La CocoaDev wiki contiene un lista di altri prefissi da evitare. Tu puoi controllarli quando devi sceglierli:
http://www.cocoadev.com/index.pl?ChooseYourOwnPrefix

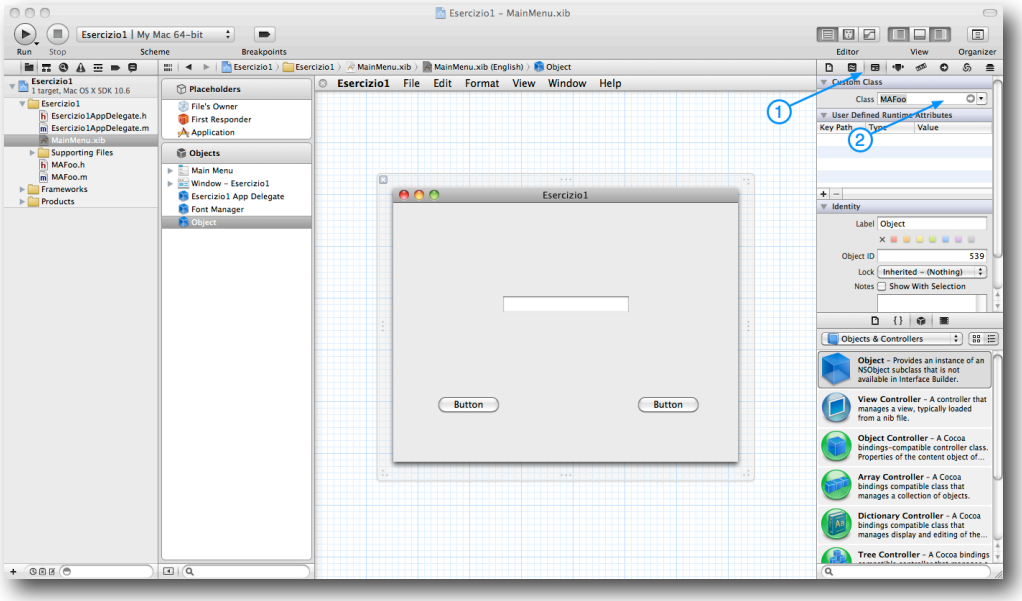
Quando crei un nuova classe dovresti dargli un nome che renda bene l'idea di che classe si tratta. Per l'istanza, abbiamo già visto che in Cocoa la classe usata per rappresentare le finestre è NSWindows, Un altro esempio è la classe che viene usata per rappresentare i colori, che è nominata NSColor. Nel nostro caso, la classe MAFoo che stiamo creando è giusto per illustrare come gli oggetti comunicano tra loro in un'applicazione.
Ecco perché abbiamo dato un nome generico e non uno specifico.

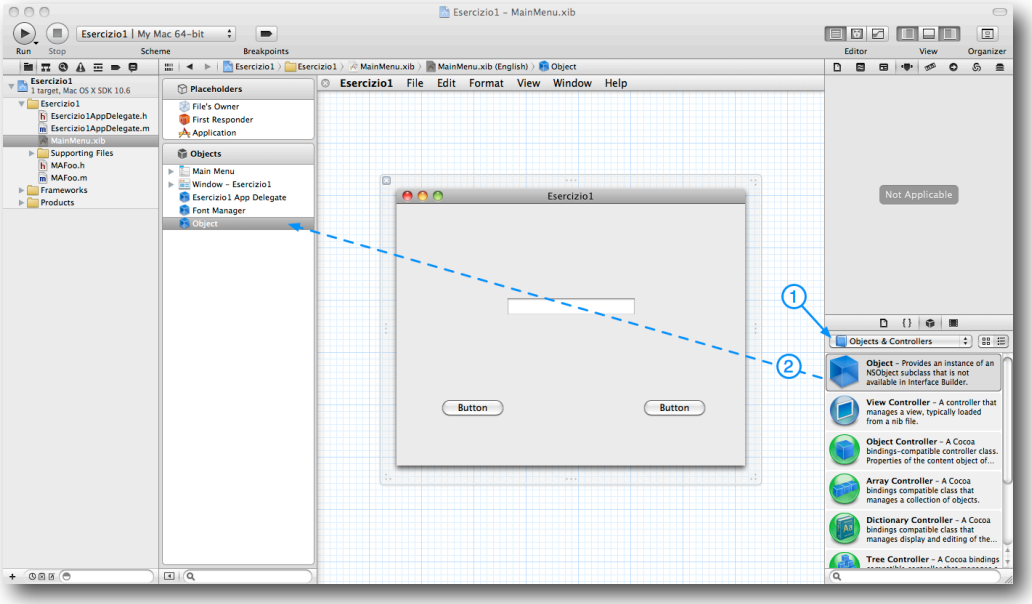
Creare un'istanza in Interface Builder

Torna alla creazione dell'interfaccia selezionando MainMenu.xib, vai nella Library e, nell'area superiore, scegli Objects & Controllers (1). Trascina quindi un oggetto (il cubo blu) dalla Library nella sezione Objects (2).

Istanziare un nuovo oggetto

Ora, con l'oggetto appena inserito selezionato, nell'Inspector clicca la tab Identity, quindi scegli MAFoo dal menu a comparsa Class.





Settiamo l'identità della nostra istanza oggetto

Bene abbiamo appena creato un'istanza della nostra classe MAFoo in Interface Builder. Questo ci permetterà di far dialogare codice ed interfaccia grafica.

Creiamo le connessioni

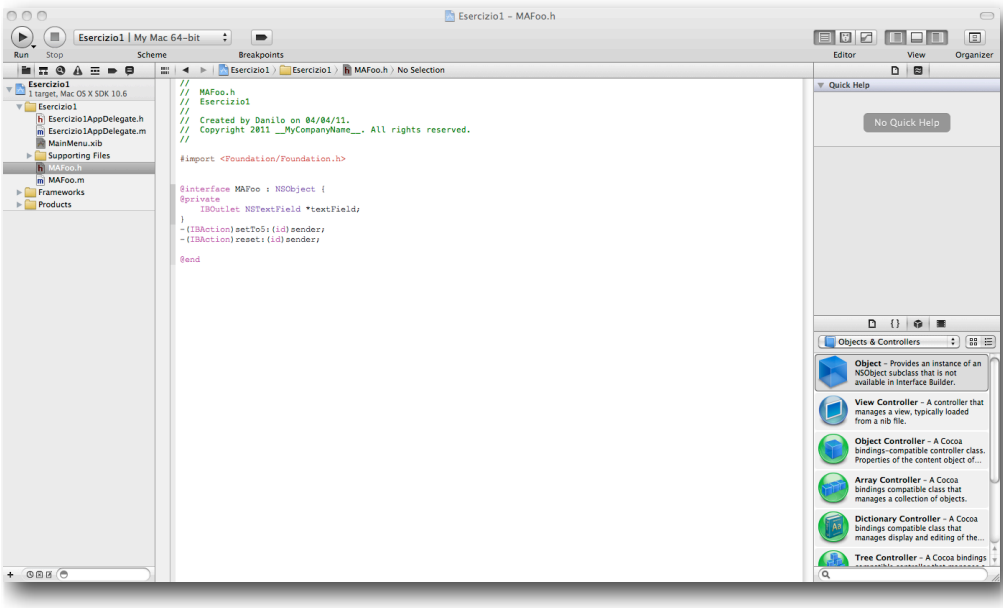
Il nostro prossimo passo è creare le connessioni tra i bottoni (da cui sono spediti i messaggi) e il nostro oggetto MAFoo. In aggiunta, creeremo anche una connessione tra l'oggetto MAFoo e il campo testo, questo perché un messaggio sarà spedito all'oggetto campo testo. Un oggetto non ha modo di spedire un messaggio ad un altro oggetto se non ha un riferimento a quest'ultimo. Facendo una connessione tra un bottone e il nostro oggetto MAFoo, noi facciamo si che il bottone abbia un riferimento al nostro oggetto MAFoo. Usando questo riferimento, il bottone riuscirà a spedire messaggi all'oggetto MAFoo. Analogamente, stabilendo una connessione tra l'oggetto e il campo testo renderemo possibile inviare messaggi a quest'ultimo.
Consideriamo cosa deve fare l'applicazione. Ognuno dei bottoni può spedire, quando cliccato, un messaggio corrispondente ad una particolare azione.
Questo messaggio contiene il nome del metodo della classe MAFoo che dev'essere eseguito. Il messaggio viene spedito all'istanza della classe MAFoo che abbiamo appena creato, l'oggetto MAFoo. (Ricorda: l'oggetto istanziato non contiene il codice per realizzare l'azione, ma la classe sì). Quindi, questo messaggio spedito all'oggetto MAFoo fa eseguire un metodo della classe MAFoo per fare qualcosa: in questo caso, spedisce il messaggio al l'oggetto campo testo. Come ogni messaggio, questo consiste nel nome di un metodo (che l'oggetto campo testo dovrà eseguire). In questo caso , il metodo dell'oggetto campo testo ha l'obiettivo di mostrare un valore, e questo valore dev'essere spedito come parte di un messaggio (chiamato "argomento", ricordate?), assieme al nome del metodo per mostrarlo sul campo testo.
La nostra classe ha bisogno quindi di due azioni (metodi), i quali saranno mandati in esecuzione dai due oggetti bottoni. La nostra classe ha bisogno anche di un outlet, una variabile per ricordare a quale oggetto (nell'esempio l'oggetto campo testo) sarà spedito il messaggio.
Seleziona il file MAFoo.h. Dopo la parola @private scrivi

```
    IBOutlet NSTextField *textField;
```

in questo modo hai dichiarato la variabile textField che sarà usata come riferimento al campo testo, IBOutlet scritto all'inizio fa si che essa sia visibile anche nella gestione dell'interfaccia grafica.
Dopo la '}' e prima di @end scrivi

```
-(IBAction)setTo5:(id)sender;
-(IBAction)reset:(id)sender;
```

in questo modo hai dichiarato i due metodi che saranno associati ai due bottoni. Nel nostro esempio useremo il metodo setTo5 per mostrare 5 nel campo testo ed il metodo reset per mostrare 0.



Analizziamo brevemente il file MAFoo.h.

```
//[3]
/* MAFoo */
#import <cocoa/cocoa.h>                    // [3.2]

@interface MAFoo : NSObject                // [3.4]
{
    IBOutlet NSTextField textField;                // [3.6]
}
- (IBAction) setTo5:(id)sender;           // [3.8]
- (IBAction) reset:(id)sender;            // [3.10]
@end
```

Torniamo per un attimo al Capitolo 4, dove si discuteva delle funzioni. Ricordi lo header di una funzione [11.1]? Era un tipo di avviso per il compilatore per dire cosa poteva attendersi. Uno dei due file che abbiamo appena creato si chiama appunto MAFoo.h, ed è chiamato "file d'intestazione della classe" e contiene informazioni circa la nostra classe. Esaminandolo riconosciamo la linea

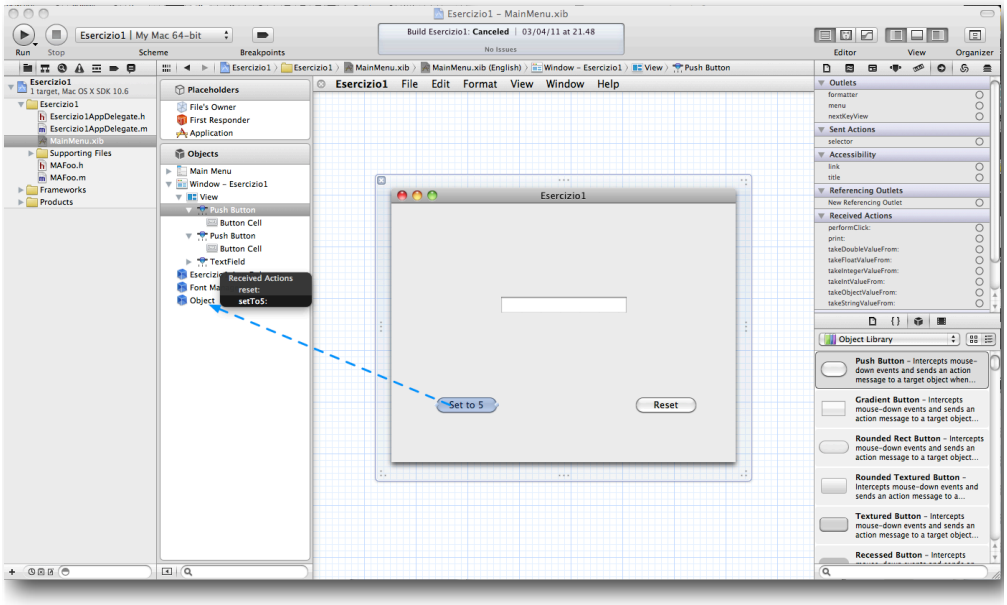
[3.4] contenente NSObject, a significare che la nostra classe eredita dalla superclasse NSObject.
Abbiamo dichiarato un Outlet [3.6] al campo testo (la variabile che, all'interno del programma, farà da riferimento per l'oggetto dell'interfaccia). "IB" invece sta per Interface Builder, il componente di XCode usato per creare graficamente l'interfaccia utente. IBAction [3.8, 3.10] invece è l'equivalente di void, in quanto i nostri messaggi non devono ritornare nessun valore. Abbiamo dichiarato due Interface Builder Action. Questi sono due metodi della nostra classe. I metodi possono essere paragonati alle funzioni, di cui abbiamo già parlato, ma con alcune differenze che vedremo fra poco.

Prima di stabilire le connessioni tra gli oggetti diamo dei nomi significativi ai nostri due bottoni. Poichè il primo chiederà all'istaza MAFoo di visualizzare il numero 5 nel campo testo, lo chiamiamo "Set to 5" (per cambiare nome al bottone: doppio click sul suo nome nell'oggetto e inserisci il nuovo nome). Allo stesso modo chiamiamo il secondo "Reset". Nota che questo passaggio di dare un nome particolare ad un bottone non è necessario per far funzionare correttamente la nostra applicazione. E' solo che vogliamo che l'interfaccia utente sia il più descrittiva possibile per l'utente finale.

Ora siamo pronti per creare le connessioni tra :

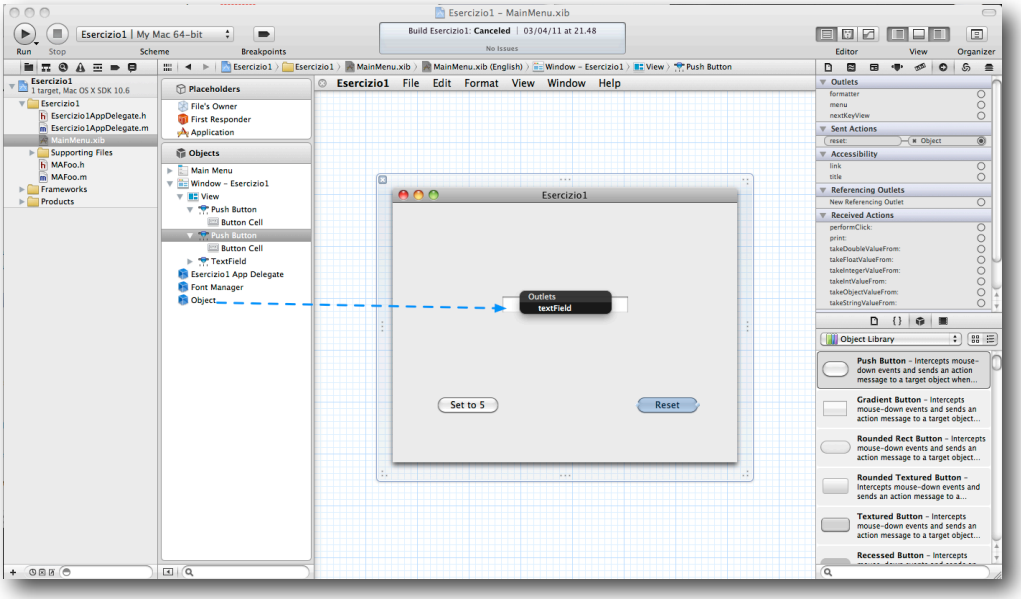
- il bottone "Reset" e l'istanza MAFoo
- il bottone "Set to 5" e l'instanza MAFoo
- l'istanza MAFoo e il campo testo.

Per creare le connessioni per le Actions fai click con il destro sul bottone Set to 5 e trascina il mouse sull'oggetto Object; lascia il pulsante del mouse e seleziona setTo5: nel popUp che ti sarà presentato.



Ripeti lo stesso procedimento per l'altro bottone.

Analogamente per connettere l'Outlet clicca con il bottone destro sull'oggetto Object e trascina il mouse sul campo di testo, rilascia e scegli textField.



Ma che cosa è successo? Presto detto; creando le connessioni tra gli oggetti e l'istanza della tua classe MAFoo hai generato una parte codice che permetterà di far funzionare la tua applicazione, il tutto senza scrivere una sola riga.
Nella realtà il codice generato sarà a te del tutto, o comunque in parte invisibile, ma fidati che c'è e tra un pò te ne darò la dimostrazione.

Ora apri il file, MAFoo.m, chiamato "file d'implementazione della classe".

```
@implementation MAFoo

- (id)init
{
    self = [super init];
    if (self) {
        // Initialization code here.
    }

    return self;
}

- (void)dealloc
{
    [super dealloc];
}

@end
```

Prima di tutto viene importato il file di intestazione MAFoo.h [4.2]. Poi trovi due metodi (funzioni) predefiniti

```
- (id)init
- (void)dealloc
```

con il loro corpo di funzione che per il momento puoi ignorare
Nella nostra applicazione, quando un bottone è premuto, spedisce un messaggio al tuo oggetto MAFoo, richiedendo l'esecuzione di uno dei metodi setTo5 o reset che abbiamo dichiarato nel file .h ma non abbiamo ancora implementato ossia scritto il codice che poi verrà eseguito.
Tenendo sempre a mente che il nostro scopo è quello di visualizzare un valore all'interno del oggetto campo testo, modifica il file MAFoo.m come in [5] per mandere un messaggio, nel nostro caso `setIntValue` al campo testo, che nel nostro caso è l'outlet textField [5.6, 5.11].

```
//[5]
#import "MAFoo.h"


@implementation MAFoo

- (id)init
{
    self = [super init];
    if (self) {
        // Initialization code here.
    }

    return self;
}

- (IBAction)reset:(id)sender
{
    [textField setIntValue:0];       // [5.6]
}

- (IBAction)setTo5:(id)sender
{
    [textField setIntValue:5];       // [5.11]
}

- (void)dealloc
{
    [super dealloc];
}

@end
```

Come puoi vedere, mandiamo un messaggio all'oggetto referenziato dall'outlet textField. Poichè abbiamo collegato questo outlet al campo di testo effettivo il nostro messaggio sarà mandato all'oggetto corretto. Il messaggio è il nome di un metodo, setIntValue:, insieme ad un valore intero. Il metodo setIntValue: è in grado di visualizzare un valore intero nel campo testo, nel prossimo capitolo vedremo come trovarne altri..

Pronti a ballare?

Giunto a questo punto sei pronto per testare la tua prima applicazione. Premi quindi "Run" e attendi un istante che Xcode crei l'applicazione e la visualizzi. Adesso puoi divertirti a premere i bottoni e vedere il risultato.

La tua applicazione in funzione

Complimenti hai appena creato la tua prima applicazione con Interfaccia Grafica.

09: Troviamo i metodi

Introduzione

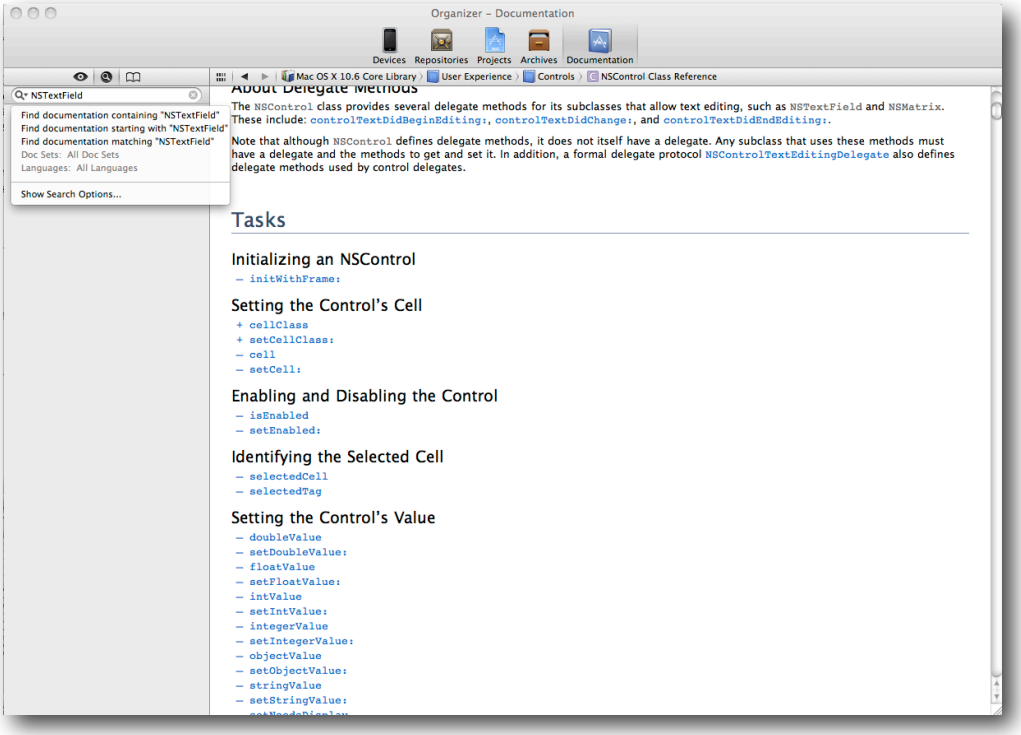
Nel capitolo precedente abbiamo imparato cosa sono i metodi. Abbiamo scritto (il corpo di) due metodi, ma ne abbiamo anche utilizzato uno fornito da Apple , setIntValue:, metodo usato per visualizzare un intero in un oggetto textfield. Come possiamo fare a scoprire tutto sui metodi a nostra disposizione?

Ricorda, per ogni metodo creato da Apple che usi non devi scrivere nessun codice. Inoltre, è più probabile che sia senza errori. Quindi, vale sempre la pena spendere un pochino di tempo per verificare se sono presenti dei metodi adeguati per il tuo programma.

Esercizio

Seleziona il file MainMenu.xib (accedendo così alla componente Interface Builder di XCode), se scegli un oggetto dalla Library e ci lasci sopra il puntatore per un secondo si apre una piccola finestra descrittiva con in più un nome. Se metti il cursore sull'icona bottone, vedrai "NSButton". Se lo tieni sopra il campo di testo , vedrai "NSTextField". Ognuno di questi nomi è il nome di una classe. Diamo un'occhiata alla classe "NSTextField" per vedere quali metodi sono disponibili.

Vai in Xcode e nel menu seleziona Help->Documentation and API Reference. Inserisci "NSTextField" nel campo di ricerca e scegli l'opzione di ricerca *Find documentation containing* e poi, nei risultati della ricerca, scegli la voce NSTextField Class Reference.



La prima cosa che devi notare è che questa classe eredita da una serie di altre classi. L'ultimo della lista è il padre di tutto, l'NSObject. Un po' sotto vi è la sezione *Tasks* nella quale sono elencati i metodi raggruppati per tipologia di compiti e che sono poi descritti nella sezione successiva *Instance Methods*.

Ecco da dove cominciare la nostra ricerca. Un rapido sguardo ai metodi ci dirà che qui non troveremo il metodo di cui abbiamo bisogno per la visualizzazione di un valore nel campo di testo dell'oggetto. Perché, a causa del principio di eredità, dobbiamo visitare la superclasse madre di NSTextField, che è NSControl (e se non riusciamo, dobbiamo visitare la sua superclasse NSView, etc). Poichè la documentazione è in HTML, tutto quello che dobbiamo fare è cliccare sulla parola NSControl (scritta nella lista della gerarchia Inherits from). Questa ci fornirà le informazioni della classe NSControl:

NSControl
Eredita da NSView : NSResponder : NSObject

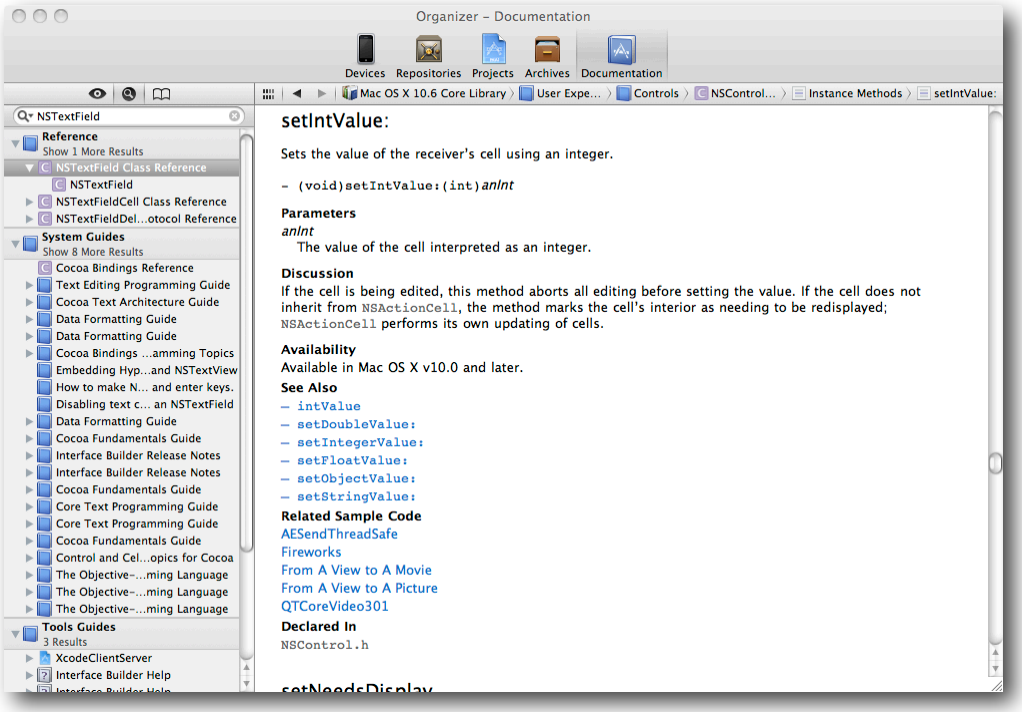
Come puoi vedere siamo saliti di una classe. Nella lista dei metodi si nota un gruppo:

Setting the control's value

Questo è ciò che vogliamo, impostare un valore. Al suo interno troviamo:

- *setIntValue*:

Potrebbe essere lui? Controlliamo la descrizione di questo metodo cliccando il link setIntValue:



Nella nostra applicazione, il nostro oggetto NSTextField è il ricevitore e noi dobbiamo passargli un intero. Possiamo anche vederlo dalla dichiarazione del metodo:

```
-(void)setIntValue:(int)anInt
```

In Objective-C, il segno meno indica l'inizio della dichiarazione di un metodo di istanza (al contrario della dichiarazione di un metodo di classe, del quale parleremo dopo). Void indica quando non viene restituito nulla al chiamante del metodo. Così, quando inviamo il messaggio `setIntValue:` al textField, il nostro

oggetto MAFoo non riceverà nessun valore indietro dall'oggetto campo di testo. Questo va bene. Dopo i due punti, (int) indica che la variabile anInt deve essere un intero. Nel nostro esempio noi mandiamo un valore di 5 o 0, che sono interi e questo va bene. A volte è un po' difficile trovare qual è il metodo da usare. Lo vedrai meglio quando prenderai più dimestichezza con la documentazione, quindi continua ad allenarti.
Che cosa succede se vuoi leggere il valore del nostro campo di testo dell'oggetto textField? Ricordi il fatto che le funzioni racchiudono tutte le variabili al loro interno? Lo stesso vale per i metodi. Spesso, tuttavia, gli oggetti hanno una coppia di metodi, chiamati "Accessors", uno per la lettura e uno per l'impostazione del valore. Conosciamo già l'ultimo, quello è il metodo `setIntValue:`. Il primo sarà invece:

```
//[1]
-(int) intValue
```

Come puoi vedere, questo metodo ritorna un intero. Quindi, se vogliamo leggere un valore intero associato al nostro oggetto textfield, dobbiamo inviare un messaggio come questo:

```
//[2]
int resultReceived = [textField intValue];
```

Di nuovo, nelle funzioni (e metodi), tutti i nomi delle variabili sono schermati. Questo è fantastico per i nomi delle variabili, perché non devi temere che l'impostazione di una variabile in una parte del tuo programma interesserà una variabile con lo stesso nome nella tua funzione. Tuttavia i nomi delle funzioni devono essere unici nel programma. Objective-C va un passo oltre nella schermatura: i nomi dei metodi devono essere univoci all'interno di una sola classe, ma diverse classi possono avere metodi con lo stesso nome. Questa è una caratteristica ottima per i grandi programmi, in quanto i programmatori possono scrivere classi indipendenti l'una dall'altra, senza dover temere conflitti nel nome dei metodi.

C'è di più. Il fatto che differenti metodi in classi differenti possano avere lo stesso nome è chiamato in gergo polimorfismo, ed è una delle cose che rendono la programmazione orientata agli oggetti così speciale. Esso consente di scrivere porzioni di codice senza dover conoscere in anticipo quali sono le classi di oggetti che stai manipolando. E' necessario solo che, durante l'esecuzione , l'oggetto attuale capisca il messaggio inviato.

Approfittando del polimorfismo, è possibile scrivere applicazioni che siano progettate per essere flessibili ed estendibili. Ad esempio nell'applicazione GUI che abbiamo creato, possiamo rimpiazzare il campo di testo con un oggetti di una classe differente che abbia l'abilità di capire il messaggio di `setIntValue:`, la nostra applicazione continuerà a funzionare senza la necessità di modificare il nostro codice, o addirittura di ricompilare. Saremo anche in grado di variare l'oggetto durante l'esecuzione senza rompere nulla. Qui sta il potere della programmazione orientata agli oggetti.

10: awakeFromNib

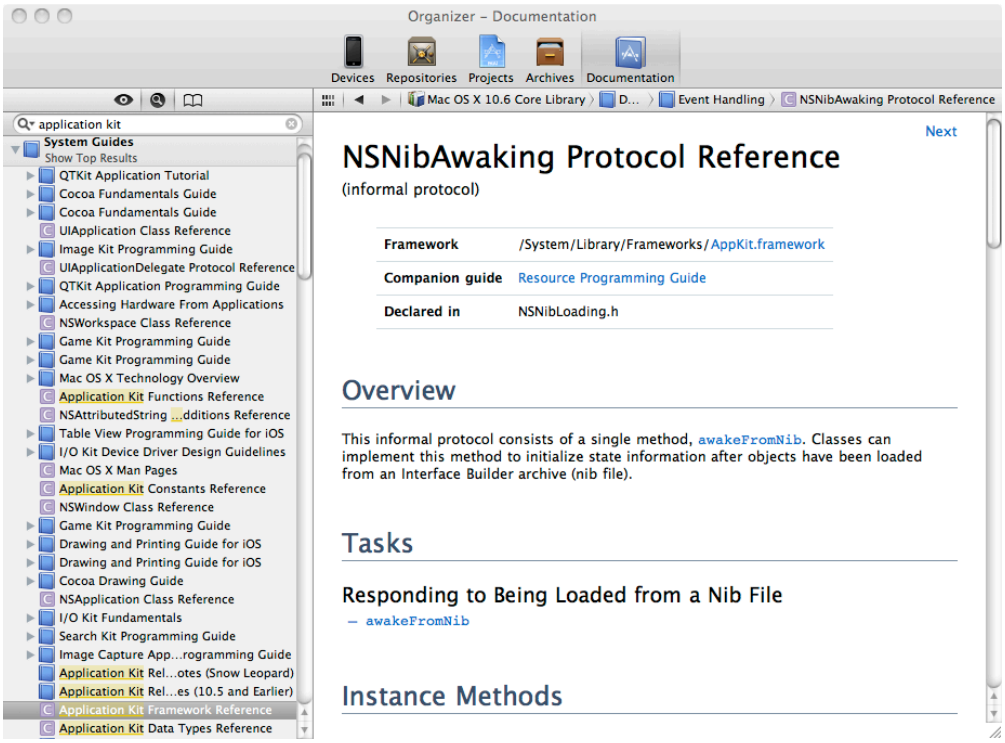
Introduzione

Apple ha fatto un sacco di lavoro per te, rendendo più facile la creazione dei tuoi programmi. In una tua piccola applicazione, non devi preoccuparti di disegnare una finestra e un bottone su uno schermo, e molte altre cose.
La maggior parte del lavoro è resa disponibile attraverso due frameworks. Il Foundation Kit framework che abbiamo importato nell'esempio [12] del capitolo 4, che fornisce la maggior parte dei servizi non associati alle interfacce grafiche. L'altro framework, chiamato Application Kit, si occupa di mostrare gli oggetti sullo schermo e di gestire i meccanismi di interazione con l'utente. Entrambi i framework sono ben documentati.

Torniamo alla nostra applicazione GUI. Supponiamo di voler mostrare immediatamente un particolare valore nel nostro campo di testo, quando l'applicazione è avviata e la finestra è visualizzata.

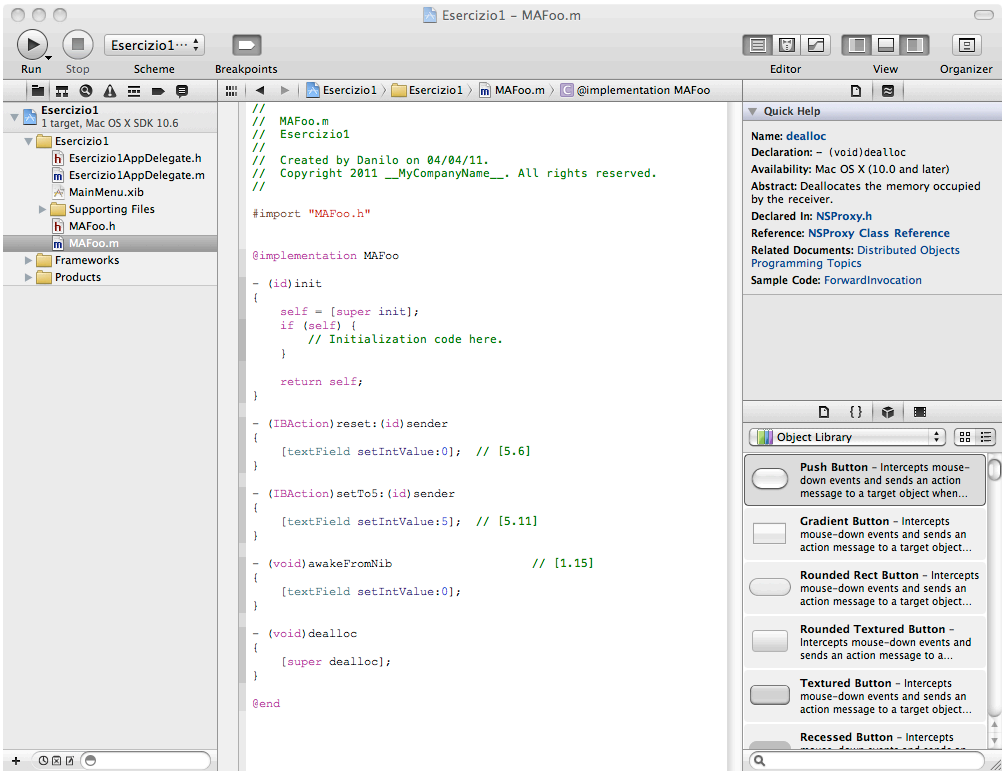
Esercizio

Tutte le informazioni per la finestra sono memorizzate in un file xib (nelle versioni precedenti si chiamava nib che sta per NeXT Interface Builder, non approfondiamo in questa sede la differenza tra i due tipi di file). Questa è un buon indice del fatto che il metodo di cui abbiamo bisogno possa essere parte di Application Kit. Vediamo come ottenere informazioni su questo framework.
In Xcode, vai nel menu Help e seleziona Documentation & API Reference. Nella finestra della documentazione scrivi Application Kit nel campo di ricerca e premi Return.
Xcode ti fornisce risultati multipli. Nel gruppo System Guides vi è un documento denominato Application Kit Framework Reference se non è nell'elenco visualizzato clicca su *show more results*. All'interno troverete un elenco dei servizi forniti dal framework. Sotto la voce Protocols c'è un link chiamato NSNibAwaking. e se fai fa clic su di esso, otterrai la documentazione per la classe NSNibAwaking.



Se implementiamo l'unico metodo fornito da questo protocollo, esso verrà chiamato quando il nostro oggetto è caricato dal suo file xib. Così siamo in grado di utilizzarlo per raggiungere il nostro obiettivo: la visualizzazione di un valore nel campo di testo al momento dell'avvio.
Non voglio dire che sia sempre semplice trovare il metodo corretto. Spesso, si richiede un po' di navigazione e l'uso creativo di parole chiave per le ricerche, per trovare quello desiderato. Per questo motivo, è molto importante familiarizzare con la documentazione di entrambi i framework, così da poter sapere quali classi e metodi sono disponibili. Potrebbe non essere necessario in quel momento, ma ti aiuteranno a capire come ottenere dal tuo programma ciò che vuoi.

Ok, adesso abbiamo trovato il nostro metodo, tutto quello che dobbiamo fare è di aggiungerlo all'implementazione nel file MAFoo.m [1.15].



Se ora esegui la tua applicazione quando la finestra si apre, il metodo `awakeFromNib` è chiamato automaticamente. Come risultato il campo di testo visualizzerà zero.

11: Puntatori

Attenzione!

Questo capitolo contiene concetti avanzati e si occupa di concetti fondamentali del C che potrebbero intimidire i principianti. Se non capisci tutto adesso, non ti preoccupare. Grazie al cielo, in generale, anche se capire come funzionano i puntatori è utile, non è essenziale per iniziare a programmare in Objective-C

Introduzione

Quando dichiari una variabile il tuo Mac associa a questa variabile dello spazio in memoria dove immagazzinare il valore. Per esempio, esaminiamo la seguente istruzione:

```
//[1]
int x = 4;
```

Quando viene eseguita, il tuo Mac trova dello spazio nella sua memoria, che non sia ancora usato, e annota che questo spazio è dove la variabile x scrive il suo valore (certamente avremmo potuto e dovuto usare un nome più descrittivo per la nostra variabile). Guarda nuovamente l'istruzione [1]. Indicando il tipo della variabile (qui `int`) fai sapere al tuo computer quanto spazio in memoria è necessario per contenere il valore di x. Se il valore fosse stato long long o double, ci sarebbe voluta più memoria.

L'istruzione di assegnamento `x = 4` memorizza il numero 4 nello spazio a lei riservato. Certamente il tuo computer ricorda dove il valore della variabile chiamata x è memorizzato, o, in altre parole, qual è l'indirizzo di x. In questo modo ogni volta che usi x in un programma, il tuo computer può guardare nel posto giusto (al giusto indirizzo) e trovare il valore attuale di x.

Un puntatore è semplicemente una variabile che contiene l'indirizzo di un'altra variabile.

Referenziare variabili

Data una variabile, tu puoi prendere il suo indirizzo scrivendo & prima della variabile. Per esempio, per prendere l'indirizzo di x tu scriverai &x.

Quando il computer valuta l'espressione x egli ritorna il valore della variabile x (nel nostro esempio, ritornerà 4). Di contro quando il computer valuta l'espressione &x ritornerà l'indirizzo della variabile x, non il valore memorizzato. L'indirizzo è un numero che denota uno specifico posto nella memoria del computer (come un numero di stanza denota una specifica camera in un hotel).

Usare i puntatori

Dichiara un puntatore in questo modo:

```
//[2]
int *y;
```

Questa istruzione definisce una variabile chiamata y che conterrà l'indirizzo di una variabile di tipo int. Di nuovo: non conterrà una variabile int, ma l'indirizzo della variabile. Per memorizzare nella variabile y l'indirizzo della variabile x devi fare:

```
//[3]
y = &x;
```

Adesso y "punta a" l'indirizzo di x. Usando y, perciò, è possibile rintracciare x, vediamo come.
Dato un puntatore, tu puoi accedere alla variabile che punta scrivendo un asterisco prima del puntatore. Per esempio valutando l'espressione:

```
*y
```

ritornerà 4. Questo è equivalente a valutare l'espressione x. Eseguire l'istruzione:

```
*y = 5
```

è equivalente a eseguire l'istruzione:

```
x = 5
```

I puntatori sono utili perché spesso non vuoi riferirti al valore di una variabile, ma all'indirizzo di essa. Per esempio, tu potresti voler programmare una funzione che aggiunge 1 a una variabile. Bene, non puoi farlo in questo modo?

```
//[4]
void increment(int x)
{
    x = x + 1;
}
```

Al momento no. Se tu chiami questa funzione da un programma, non ottieni il risultato che ti aspetti:

```
//[5]
int myValue = 6;
increment(myValue);
NSLog(@"%d:\n", myValue);
```

Questo codice mostra 6 sul monitor. Perché? Non incrementa myValue chiamando la funzione di incremento? No, al momento no. Come vedi, la funzione in [4] prende il valore di myValue (nell'esempio il numero 6), lo incrementa di 1 e... fondamentalmente, lo butta. Le funzioni lavorano solo con i valori che gli passi, non sulle variabili che li contengono. Anche se modifichi x (come puoi vedere in [4]), tu modifichi solo il valore che la funzione riceve. Ogni modifica verrà persa quando la funzione sarà completata. Inoltre, x, non è necessariamente una variabile: se si chiama incremento(5); che cosa ti aspetti di incrementare? Se vuoi scrivere una versione di una funzione di incremento che funzioni, per esempio che accetti una variabile come suo argomento e ne incrementi permanentemente il valore, hai bisogno di passare l'indirizzo di una variabile. In questo modo, tu modifichi cosa è memorizzato nella variabile, non solo il valore corrente. Quindi usi un puntatore come argomento:

```
//[6]
void increment(int *y)
{
    *y = *y + 1;
}
```

Puoi chiamare la funzione così:

[7]

```
int myValue = 6;
increment(&myValue); // passiamo l'indirizzo
// ora myValue è uguale a  7
```

12: Stringhe

Introduzione

Sinora abbiamo visto diverse tipologie base di dati, quali int, long, float, double, BOOL. Inoltre nello scorso capitolo abbiamo avuto modo di introdurre i puntatori. Quando abbiamo accennato alle stringhe, lo abbiamo fatto solo in relazione alla funzione NSLog(). Questa funzione, come abbiamo avuto modo di vedere, ci permette di stampare una stringa a video e di inserire il valore di una variabile in corrispondenza delle parole chiave inizianti con l'operatore %, ad esempio %d.

```
//[1]
float piValue = 3.1416;
NSLog(@"Qui ci sono alcuni esempi di stringhe stampate a video.\n");
NSLog(@"Pi approssimativamente %10.4f.\n", piValue);
NSLog(@"Il numero di facce di un dado é %d.\n", 6);
```

Non abbiamo trattato prima le stringhe come tipi di dati per una buona ragione: diversamente da interi e numeri a virgola mobile le stringhe sono veri oggetti, create usando la classe NSString o la classe NSMutableString. Studiamo queste classi, partendo da NSString.

NSString

Ancora puntatori

```
//[2]
NSString *favoriteComputer;                                    //[2.1]
favoriteComputer = @"Mac!";
NSLog(favoriteComputer);
```

Probabilmente comprenderai la seconda riga, mentre la prima [2.1] necessiterà di qualche spiegazione. Ti ricordi che, quando dichiariamo una variabile puntatore, dobbiamo dire a che tipo di variabile punta? Come nello statement del capitolo 11 [3].

```
//[3]
int *y;
```

Dobbiamo dire al compilatore che il puntatore y contiene l'indirizzo ad una locazione di memoria dove si trova un intero.
In [2.1] diciamo al compilatore che il puntatore favoriteComputer contiene l'indirizzo ad una locazione di memoria dove si trova un oggetto di tipo NSString. Per le stringhe utilizziamo dei puntatori perché in Objective-C gli oggetti non sono mai manipolati direttamente ma solo attraverso i loro puntatori.
Non preoccuparti troppo di capire questo concetto - non è fondamentale (al momento. ndT). L'importante è di riferirsi ad un'istanza di NSString o NSMutableString (o a qualunque oggetto) usando l'operatore *.

Il simbolo @

Bene, perché ci ritroviamo ovunque questo simbolo @? Il fatto é che Objective-C é un estensione del linguaggio C, che ha i suoi modi per trattare con le stringhe. Per differenziare le nuove tipologie di stringhe, che ricordiamo essere oggetti a tutti gli effetti, Objective-C usa la notazione @.

Un nuovo tipo di stringhe

In che maniera Objective-C migliora le stringhe del linguaggio C? Le stringhe in C sono in formato ASCII, mentre in Objective-C sono in formato Unicode. Le stringhe Unicode possono contenere caratteri di qualunque linguaggio, dall'alfabeto latino al cinese.

Esercizio

Naturalmente é possibile dichiarare e inizializzare una stringa in un unico momento [4].

```
//[4]
NSString *favoriteActress = @"Julia";
```

Il puntatore favoriteActress punta alla memoria dove risiede l'oggetto che rappresenta la stringa "Julia".
Dopo aver inizializzato la variabile, es. favoriteComputer, puoi dare alla variabile un altro valore, ma non puoi cambiare la stringa stessa [5.7], perché é un'istanza della classe NSString. Dettagli al riguardo a breve.

```
//[5]
#import <Foundation/Foundation.h>
int main (int argc, const char *argv[])
{
  NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
  NSString *favoriteComputer;
  favoriteComputer = @"iBook";                              // [5.7]
  favoriteComputer = @"MacBook Pro Intel";
  NSLog(favoriteComputer);
  [pool release];
  return 0;
}
```

Quando viene eseguito questo programma stampa:

MacBook Pro Intel

NSMutableString

Una stringa della classe NSString é detta immutabile, perché non può essere modificata.
A che serve una stringa che non può essere modificata? Stringhe di questo genere sono più facili da gestire per il sistema operativo, quindi il programma può essere più veloce. Ti accorgerai, scrivendo programmi in Objective-C, che spesso non avrai bisogno di modificare le tue stringhe.
Di tanto in tanto, certo, potresti necessitare di stringhe che debbano essere modificate. Per questo motivo esiste una classe diversa la quale crea oggetti stringa modificabili. La classe é NSMutableString e ne parleremo più avanti in questo capitolo.

Esercizio

Prima di tutto assicuriamoci che tu abbia capito che le stringhe sono oggetti. Visto che sono oggetti possiamo mandare loro dei messaggi. Per esempio possiamo mandare il messaggio `length` [6].

```
//[6]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
```

```
NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int theLength;
    NSString * foo;
    foo = @"Julia!";
    theLength  = [foo length];                              // [6.10]
    NSLog(@"The length is %d.", theLength);
    [pool release];
    return 0;
}
```

Quando eseguiamo questo programma viene stampato:

The length is 6.

I programmatori spesso usano nomi quali foo e bar per le variabili quando spiegano le cose. In realtà non sono dei buoni nomi, perché non sono descrittivi, esattamente come x. In questo caso li usiamo in modo che non ti troverai spiazzato incontrandoli in una discussione in internet.

Alla riga [6.10] mandiamo all'oggetto foo, il messaggio `length`. Il metodo `length` é definito nella classe NSString nel seguente modo:

```
- (unsigned int)length
```

Restituisce il numero di caratteri Unicode nel ricevente.

Puoi anche modificare i caratteri della stringa in maiuscoli [7]. A questo scopo manda alla stringa il messaggio appropriato, uppercaseString, che dovresti essere in grado di rintracciare da solo nella documentazione (controlla i metodi disponibili per la classe NSString).
Alla ricezione di questo messaggio l'oggetto stringa crea e restituisce un nuovo oggetto stringa uguale a se stesso, ma con ciascun carattere convertito nel suo corrispettivo maiuscolo.

```
//[7]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSString *foo, *bar;
    foo = @"Julia!";
    bar  = [foo uppercaseString];
```

```
   NSLog(@"%@ é stato convertito in %@.", foo, bar);
    [pool release];
    return 0;
}
```

Questo programma, quando eseguito, stampa:

```
Julia! é stato convertito in JULIA!
```

A volte potresti voler modificare il contenuto di una stringa esistente invece di crearne una nuova. In questo caso dovrai usare la classe NSMutableString per istanziare la tua stringa. La classe NSMutableString offre diversi metodi per modificare il contenuto delle stringhe. Per esempio il metodo `appendString`: il quale aggiunge la stringa fornita come argomento alla fine del ricevente.

```
//[8]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableString *foo;                                          // [8.7]
    foo = [@"Julia!" mutableCopy];                                 // [8.8]
    [foo appendString:@" Io sono felice"];
    NSLog(@"Qui c'é il risultato: %@.",  foo);
    [pool release];
    return 0;
}
```

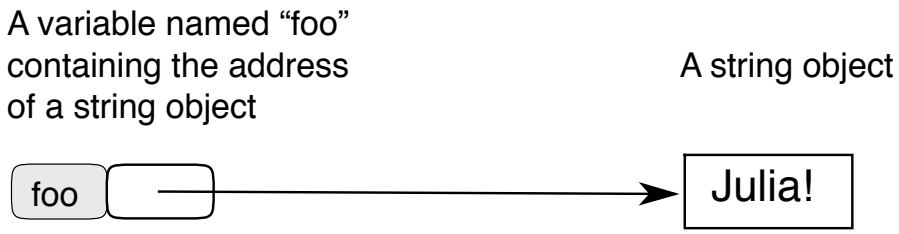
Quando eseguito il programma stampa:

```
Qui c'é il risultato: Julia! Io sono felice.
```

Alla linea [8.8], il metodo mutableCopy (che é fornito dalla classe NSString) crea e restituisce una stringa modificabile con lo stesso contenuto del ricevente. Quindi, dopo l'esecuzione della linea [8.8], foo punta ad un oggetto stringa mutabile che contiene la stringa "Julia!".

Ancora puntatori!

Precedentemente in questo stesso capitolo abbiamo affermato che, in Objective-C, gli oggetti non sono mai manipolati direttamente, ma sempre attraverso puntatori. Questo é il motivo, per esempio, per cui usiamo la notazione puntatore alla linea [8.7]. In realtà, quando usiamo la parola "oggetto" in Objective-C, quello che intendiamo in realtà é "il puntatore all'oggetto". Ma siccome usiamo sempre gli oggetti attraverso i puntatori, usiamo la parola "oggetto" per brevità. Che gli oggetti siano sempre utilizzati attraverso i puntatori ha un'importante ripercussione che devi assolutamente capire: molte variabili possono riferirsi allo stesso oggetto contemporaneamente. Per esempio, dopo l'esecuzione della linea [8.7], la variabile foo si riferisce ad un oggetto rappresentante la stringa "Julia!", qualcosa che possiamo rappresentare nel seguente modo:

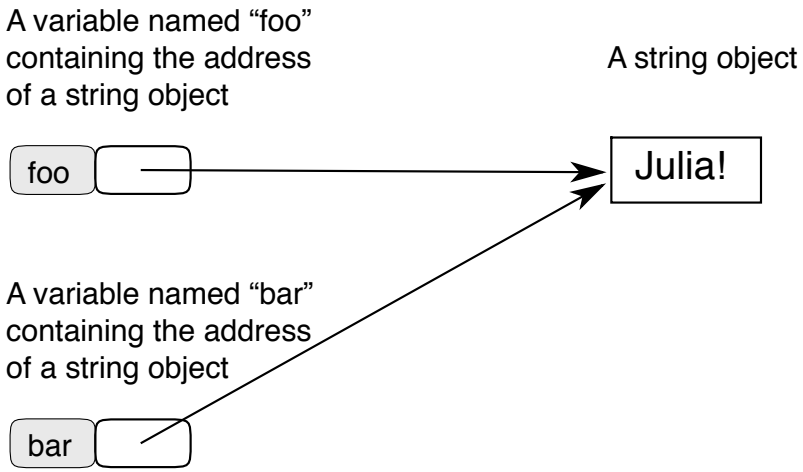


Gli oggetti sono sempre manipolati attraverso i puntatori

Adesso supponiamo di assegnare il valore di foo alla variabile bar in questo modo

```
bar = foo;
```

Il risultato di questa operazione é che sia foo che bar adesso puntano allo stesso oggetto:



Più variabili possono riferirsi allo stesso oggetto

In una situazione di questo tipo, mandare un messaggio ad un oggetto usando foo come ricevente ([foo faiqualcosa];) ha lo stesso effetto che mandare il messaggio tramite bar ([bar faiqualcosa];) come mostrato in questo esempio:

```
[9]
#import <Foundation/Foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableString *foo = [@"Julia!" mutableCopy];
    NSMutableString *bar = foo;
    NSLog(@"foo points to the string: %@.",  foo);
    NSLog(@"bar points to the string: %@.",  bar);
    NSLog(@"\n");
    [foo appendString:@" I am happy"];
    NSLog(@"foo points to the string: %@.",  foo);
    NSLog(@"bar points to the string: %@.",  bar);
    [pool release];
    return 0;
}
```

Eseguendo questo programma si ottiene:

foo points to the string: Julia!

bar points to the string: Julia!

foo points to the string: Julia! I am happy
bar points to the string: Julia! I am happy

Avere riferimenti allo stesso oggetto da posti diversi contemporaneamente é una caratteristica essenziale dei linguaggi orientati agli oggetti. In realtà abbiamo usato questa peculiarità nei capitoli precedenti, ad esempio, nel capitolo 8, abbiamo puntato all'oggetto MAFoo da due diversi oggetti pulsante.

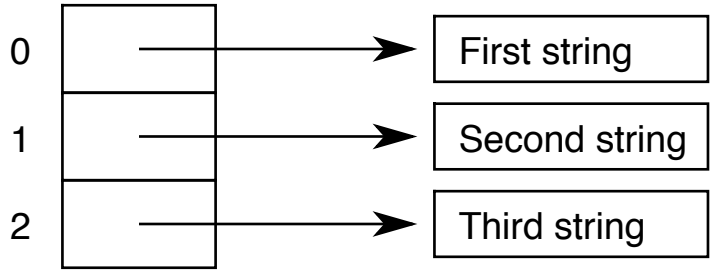
13: Arrays

Introduzione

A volte avrai la necessità di gestire insiemi di dati. Per esempio, potresti avere bisogno di gestire un elenco di stringhe. Sarebbe piuttosto scomodo usare una variabile per ognuna di quelle stringhe. Esiste in questo caso una soluzione migliore: ***l'array***.

Un array è una lista ordinata di oggetti (o, più precisamente, una lista di puntatori di oggetti). Puoi aggiungere un oggetto all'array, rimuoverlo o chiedere all'array di farti sapere quale oggetto è memorizzato ad un dato indice (cioè ad una data posizione) inoltre puoi chiedere all'array quanti oggetti contiene in quel momento.

Quando conti degli elementi, solitamente parti da 1. Negli arrays invece, il primo elemento è etichettato come zero, il secondo come 1 e così via.



Esempio: Un array contenente tre stringhe

Più avanti in questo capitolo ti daremo degli esempi di codice dove potrai vedere gli effetti della numerazione con partenza da zero.

Gli Arrays sono composti da due classes: NSArray e NSMutableArray. Come con le stringhe, esiste una versione "mutable" ed una "immutable". In questo capitolo considereremo la versione "mutable".

Questi arrays sono specifici dei linguaggio Objective-C e Cocoa. Esiste un' altro tipo di array nel linguaggio C (che è anche parte del Objective-C), ma non vogliamo trattarlo in questa sede. Questo è giusto per ricordare, tuttavia, che potresti leggere

qualcosa a riguardo degli array nel linguaggio C, e tranquillizzarti che non hanno niente a che fare con NSArrays o NSMutableArrays.

Una "class method"

Un metodo per creare un array è eseguire un'espressione come questa:

```
[NSMutableArray array];
```

Quando viene eseguito, questo codice crea e restituisce un array vuoto. Ma .. aspetta un minuto...questo codice sembra strano, non è vero? Effettivamente, in questo caso noi abbiamo usato il nome NSMutableArray class per specificare la ricezione di un messaggio. Ma finora noi abbiamo solo mandato messaggi alle istanze, non alle classi, vero?
Bene, abbiamo appena imparato qualcosa di nuovo: il fatto è che, in Objective-C, noi possiamo anche spedire messaggi alle class (e il motivo è che le classi sono anche oggetti), istanze di quello che noi chiamiamo meta-classi, ma non approfondiremo questa cosa in questo articolo introduttivo).

Noterai che questo oggetto è autorilasciato in automatico quando viene creato; cioè e allegato al NSAutoreleasePool e impostato per essere cancellato dal metodo della classe che lo ha creato. Chiamare la class method equivale a:

```
NSMutableArray *array = [[[NSMutableArray alloc] init] autorelease];
```

Nel caso in cui vogliamo che l'array sopravviva di più rispetto al tempo di autorelease, dovrai trasmettere all' instance un messaggio di `-retain`.

Nella documentazione Cocoa, i metodi che inviamo alle classes sono identificati da un simbolo "+", invece del simbolo "-" che vediamo solitamente prima del nome dei method (Capitolo 8 [4.5]). Per esempio, nella documentazione vediamo questa descrizione per il metodo array:

array
`+ (id)array`

Crea e restituisce un array. questo metodo è usato in mutable subclasses di NSArray. Vedi anche: `+ arrayWithObject:`, `+ arrayWithObjects:`

Esercizio:

Torniamo al codice. Il seguente programma crea un array vuoto, memorizza tre stringhe al suo interno, e poi stampa il numero di elementi di cui è composto l'array.

```
//[1]
#import <foundation/foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableArray *myArray = [NSMutableArray array];
    [myArray addObject:@"prima stringa"];
    [myArray addObject:@"seconda stringa"];
    [myArray addObject:@"terza stringa"];
    int count = [myArray count];
    NSLog(@"Ci sono %d elementi nel mio array", count);
    [pool release];
    return 0;
}
```

Quando eseguito il programma stampa:

```
Ci sono 3 elementi nel mio array
```

il seguente programma è lo stesso del precedente eccetto che stampa la stringa memorizzata all'indice 0 nell'array. Per ottenere questa stringa, esso usa il metodo objectAtIndex: [2.13].

```
//[2]
#import <foundation/foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableArray *myArray = [NSMutableArray array];
    [myArray addObject:@"prima stringa"];
    [myArray addObject:@"seconda stringa"];
    [myArray addObject:@"terza stringa"];
```

```
    NSString *element = [myArray objectAtIndex:0];                // [2.13]
    NSLog(@"L'elemento con indice 0 nell'array è: %@", element);
    [pool release];
    return 0;
}
```

Quando viene eseguito il programma stampa:

```
l'elemento con indice 0 nell'array è: prima stringa
```

Spesso dovrai percorrere l'array per poter lavorare con ogni elemento. Per fare questo è possibile usare strutture a ciclo come nel programma seguente che stampa ogni elemento dell'array ed il suo indice:

```
//[3]
#import <foundation/foundation.h>

int main (int argc, const char * argv[])
{
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    NSMutableArray *myArray = [NSMutableArray array];
    [myArray addObject:@"prima stringa"];
    [myArray addObject:@"seconda stringa"];
    [myArray addObject:@"terza stringa"];
    int i;
    int count;
    for (i = 0, count = [myArray count]; i &lt; count; i = i + 1)
    {
       NSString *element = [myArray objectAtIndex:i];
       NSLog(@"L'elemento con indice %d nell'array è: %@", i, element);
    }
    [pool release];
    return 0;
}
```

Quando eseguito il programma stampa:

```
L'elemento con indice 0 nell'array è: prima stringa
L'elemento con indice 1 nell'array è: seconda stringa
L'elemento con indice 2 nell'array è: terza stringa
```

Nota che gli array non sono limitati all'uso delle stringhe, infatti possono contenere qualsiasi oggetto tu desideri.

Le classi NSArray e NSMutableArray forniscono molti altri metodi, e ti incoraggiamo a dare uno sguardo alla documentazione per queste classes allo scopo di imparare altro sugli arrays. Alla fine di questo capitolo parleremo dei metodi che permettono di rimpiazzare un oggetto ad un dato indice con un altro oggetto. Questo metodo è chiamato replaceObjectAtIndex:withObject:.
Finora ci siamo occupati di metodi che trattano solo un argomento. Questo è diverso, ha 2 argomenti. Possiamo osservare questa peculiarità dalla doppia presenza dell'operatore due punti (:). Nei methods in Objective-C possiamo avere qualsiasi numero di argomenti.
Di seguito un esempio di utilizzo di questo metodo:

```
//[4]
[myArray replaceObjectAtIndex:1 withObject:@"Hello"];
```

Dopo l'esecuzione di questo metodo, l'oggetto all'indice 1 è la stringa @"Hello". Naturalmente, questo metodo deve essere invocato per valori di indice validi. Cioè ci deve già essere un oggetto memorizzato all'indice che diamo al metodo affinché il metodo possa sostituirlo nell'array dall'oggetto che noi forniamo.

Conclusioni

Come puoi vedere, i method names in objective-C sono simili a frasi con un buco al loro interno (preceduti da un "due punti"). Quando tu invochi un metodo tu completi il buco con il valore corrente creando cosi una "sentence" di senso compiuto. Questo modo di indicare i nomi dei metodi e le loro invocazioni proviene dallo Smalltalk ed è uno dei maggiori punti di forza del linguaggio Objective-C in quanto rende il codice piu espressivo. Quando tu crei un tuo metodo personale dovresti sforzarti ad assegnargli un nome comprensibile. Questo renderà il codice objective-C più leggibile e questa è una cosa molto importante nel semplificare la manutenzione del tuo programma.

14: Gestione della memoria

Introduzione

Se ricordi, nei capitoli precedenti ho evitato di spiegare alcuni statement negli esempi. Questi statement parlavano di memoria. Il tuo programma non è il solo programma sul tuo Mac, e la RAM è una merce preziosa. Quindi se il tuo programma non ha bisogno più di una parte di memoria, questa andrebbe restituita nuovamente al sistema. Quando la mamma ti ha detto che dovevi essere educato e vivere in armonia con la comunità, ti stava insegnando come programmare! Anche se il tuo programma è l'unico in esecuzione, la memoria non liberata potrebbe eventualmente relegare il tuo programma in un angolo e il computer potrebbe rallentare.

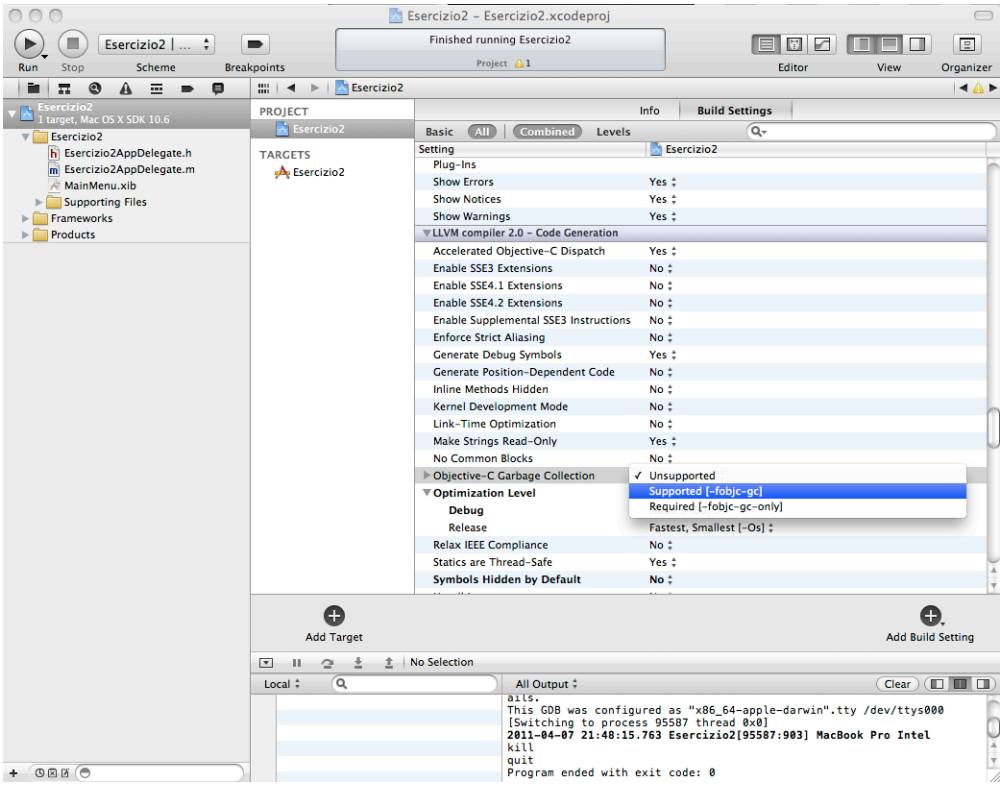
Garbage Collection

Le tecniche di gestione di memoria usate da Cocoa e introdotte successivamente in questo capitolo sono comunemente conosciute come "Reference Counting". Puoi trovare una spiegazione completa di questo sistema di gestione della memoria di Cocoa in alcuni libri avanzati o articoli (vedi Capitolo 15).

Mac OS X 10.5 Leopard introduce una nuova forma di gestione della memoria per Objective-C 2.0, conosciuta come Cocoa Garbage Collection. Garbage collection gestisce la memoria automaticamente, eliminando il bisogno di esplicitare il rilascio o l'autorelease di oggetti Cocoa. La magia del garbage collection funziona su tutti gli oggetti Cocoa che ereditano da NSObject o NSProxy e permette ai programmatori di scrivere semplicemente meno codice rispetto alle versioni precedenti di Objective-C. Non c'è molto altro da dire su di esso, in pratica. Dimenticate tutto ciò che avete imparato in questo capitolo!

Abilitare il garbage collection

Il garbage collection necessita di essere abilitato, in un nuovo progetto Xcode è spento di default. Per attivarlo seleziona il progetto e accedi alle property.



Abilita la proprietà Objective-C Garbage Collection. Nota che ogni framework che tu colleghi al tuo progetto deve essere anch'esso sotto Garbage collector.

Reference Counting: Il ciclo di vita di un oggetto

Se sei interessato alle tecniche di gestione della memoria pre-Leopard, leggi quanto segue.

Quando il tuo programma crea un oggetto, questo occupa spazio in memoria e tu devi liberare quello spazio quando il tuo oggetto non è più usato. Ovvero, quando il tuo oggetto non è più usato, devi distruggerlo. Ad ogni modo, determinare quando un oggetto ha finito di essere usato potrebbe non essere cosÏ semplice.

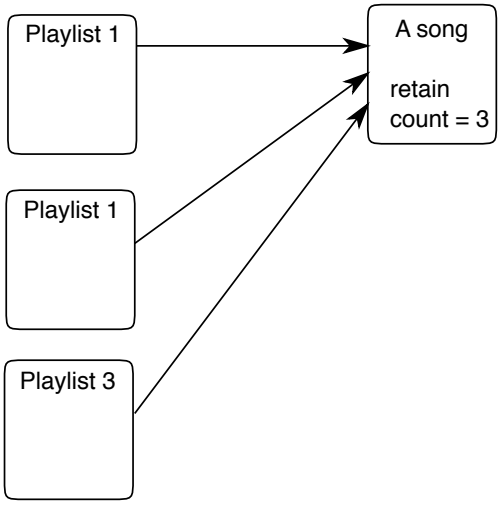
Per esempio, durante l'esecuzione del programma, il tuo oggetto può essere referenziato da molti altri oggetti, e quindi non deve essere distrutto perché c'è una possibilità che possa essere usato da altri oggetti (cercare di utilizzare un

oggetto che sia stato distrutto può causare un crash nel tuo programma o un comportamento imprevedibile).

Il contatore retain

Per aiutarti a distruggere gli oggetti quando non sono più necessari, Cocoa associa un contatore a ogni oggetto, che rappresenta quello che è chiamato il "retain count" di un oggetto. Nel tuo programma, quando memorizzi un riferimento a un oggetto, devi farlo sapere all'oggetto incrementando il retain count di uno. Quando rimuovete un riferimento ad un oggetto, il retain deve essere decrementato di uno. Quando il retain count di un oggetto diventa uguale a zero, l'oggetto sa che non è più referenziato da nessuno e può distruggersi tranquillamente. L'oggetto che si distrugge libera la memoria associata.

Per esempio, supponiamo che la tua applicazione sia un jukebox digitale e tu hai oggetti rappresentanti canzoni e playlist. Supponi che la data canzone oggetto sia referenziata da tre playlist oggetto. Se non è referenziata altrove la tua canzone oggetto avrà un retain count di tre.



Un oggetto sa quante volte è referenziato, grazie al proprio retain count.

Retain e Release

Per aumentare il retain count di un oggetto, non devi far altro che inviare all'oggetto un messaggio di retain.

```
[anObject retain];
```

Per diminuire il retain count di un oggetto, non devi far altro che inviare all'oggetto un messaggio di release.

```
[anObject release];
```

Autorelease

Cocoa offre anche un meccanismo chiamato "autorelease pool" che ti consente di inviare un messaggio di rilascio ritardato ad un oggetto, non immediatamente, ma in un secondo tempo. Per usarlo, devi registrare l'oggetto con quello che viene chiamato autorelease pool, inviando un messaggio di autorelease.

```
[anObject autorelease];
```

L'autorelease pool si prenderà cura di inviare un messaggio di rilascio ritardato al tuo oggetto. Le istruzioni che si occupano dell'autorelease pool che abbiamo visto prima nel nostro programma sono istruzioni che diamo al sistema per un corretto set-up del meccanismo dell'autorelease pool.

15: Appendice A

Il modesto obiettivo di questo libro era quello di insegnare le basi di Objective-C nell'ambiente Xcode. Se sei passato attraverso il libro due volte e hai provato gli esempi con le tue variazioni, sei pronto per imparare come scrivere la "killer application" che stai cercando di creare. Questo libro ti ha dato una conoscenza sufficiente per risolvere i tuoi problemi in tempi breve. Come hai fatto in questo capitolo, sei pronto per utilizzare altre risorse, e quelle che ti menzioniamo qui sotto dovrebbero avere la tua attenzione. Un importante consiglio prima di iniziare a scrivere il tuo codice: non partire subito, controlla i framework, perché Apple potrebbe già avere fatto il lavoro per te, o fornirti le classi che con poco lavoro ti porteranno a quello che ti serve. Inoltre, qualcun altro potrebbe aver già fatto tutto quello che ti serve, e reso disponibile il codice sorgente. Quindi spendi un pochino del tuo tempo guardando attraverso la documentazione e cercando su internet. La tua prima visita deve essere al sito degli sviluppatori Apple:

http://developer.apple.com

Ti consigliamo vivamente anche questi:

http://osx.hyperjeff.net/reference/CocoaArticles.php
http://www.cocoadev.com
http://www.cocoadevcentral.com
http://www.cocoabuilder.com
http://www.stepwise.com

I siti sopra hanno un numero molto grande di link ad altri siti o altre fonti di informazioni. Puoi anche iscriverti alla cocoa-dev mailing list a http://lists.apple.com/mailman/listinfo/cocoa-dev. Questo è il posto dove puoi scrivere le tue domande. Gli altri membri faranno del loro meglio per aiutarti. In cambio, sii educato e prima verifica se sia possibile trovare la risposta alla tua domanda negli archivi (http://www.cocoabuilder.com). Per alcuni consigli su come postare domande sulla mailing list guarda "How To Ask Questions The Smart Way" al http://www.catb.org/~esr/faqs/smart-questions.html
Ci sono diversi ottimi libri sullo sviluppo su Cocoa. Programming in Objective-C, di Stephen Kochan é rivolto ai principianti.

Altri libri assumono che tu abbia almeno una qualche conoscenza acquisita da questo libro. Noi personalmente abbiamo trovato utili Cocoa Programming for Mac OS X di Aaron Hillegass of the Big Nerd Ranch, il quale ha insegnato Xcode per una vita. Vi segnaliamo inoltre Cocoa with Objective-C di James Duncan Davidson e Apple, pubblicato da O'Reilly.

Per finire, alcuni consigli.
Prima di rilasciare il tuo programma, assicurati che sia esente da bug, ma anche che sia visibilmente gradevole e che segua le linee guida di Apple sulla realizzazione dell'interfaccia grafica.Una volta che hai fatto questo, non essere timido nel rendere il tuo programma disponibile! I commenti dagli altri ti aiuteranno a perfezionarlo, espanderne le potenzialità e a mantenerne alta la qualità.

Speriamo che questo libro ti sia piaciuto e che abbia, in un qualche modo, alimentato ancora di più il tuo interesse verso quella che è la programmazione per Mac OSX.