Marco Coïsson

Introduzione al Linguaggio Objective-C

Coïsson Editore

Introduzione al linguaggio Objective-C

Piano dell'opera

Proseguendo nella tradizione inaugurata col primo volume di quest'opera, l'ormai famosa *Introduzione al Linguaggio C*, ho deciso di dare ancora una volta questo titolo altisonante all'introduzione di questo secondo volume, perché, diciamocelo, ma fa sentire importante! Scherzi a parte, è ora il caso di fare un breve riassunto di che cosa voglia essere questa *Introduzione al Linguaggio Objective-C* e a chi si rivolge.

Come già per il primo volume, questo secondo manuale è rivolto ad un pubblico di principianti, che non hanno esperienze di programmazione in Objective-C o che forse non hanno esperienze di programmazione in generale. Gli argomenti trattati saranno di base, mentre quelli più ostici o più avanzati saranno lasciati all'ottima letteratura presente in commercio o in rete, un sunto della quale potrete trovarlo nella bibliografia. In ogni caso, è necessario che abbiate almeno un'infarinatura di C; se non l'avete, vi consiglio caldamente di leggervi qualche testo di base, inclusa l'*Introduzione al Linguaggio C*, che potete reperire alla stessa pagina web dalla quale avete scaricato questo volume sull'Objective-C.

Anche in questo caso i primi due capitoli sono essenzialmente "filosofici": non entreremo nei dettagli del linguaggio, ma ci limiteremo a discutere brevemente su che cosa voglia dire programmare "ad oggetti", e sul perché questo sia talvolta più facile e talvolta più difficile che programmare "proceduralmente", come si fa col C e con molti altri linguaggi. Un breve e tutt'altro che omnicomprensivo confronto col C++ sarà doveroso nel capitolo 2, dal momento che il C++ è forse il linguaggio "ad oggetti" più largamente usato e che trova il maggior numero di estimatori.

Dal capitolo 3 inizieremo il lavoro sporco, partendo dai mattoni di base di qualunque linguaggio "ad oggetti", le *classi* e, per l'appunto, gli *oggetti*. Benché tutto quello che diremo sia in linea di principio trasportabile quasi senza modifiche su qualunque sistema operativo che abbia un compilatore Objective-C (come Linux, ad esempio), l'attenzione sarà qui rivolta quasi esclusivamente a MacOS X, di cui inizieremo ad usare alcuni degli strumenti che Apple ha messo a disposizione dei programmatori. Questo ci consentirà di scrivere del codice più semplice e più pulito, e di iniziare a ridurre le distanze con Cocoa, l'ambiente di sviluppo di applicazioni nativo di MacOS X che sarà oggetto del terzo volume di questa saga.

Nel capitolo 4 vedremo come e quando abbia senso creare delle sottoclassi. Nel capitolo 5 vedremo invece quando non sia necessario creare una sottoclasse e sia invece più opportuno usare strumenti alternativi come *categorie* e *protocolli* per estendere le funzionalità di una classe esistente. Non ci occuperemo, invece, di argomenti più ostici e non necessari (a mio parere) a chi si avvicina per la prima volta ad Objective-C senza avere particolari esperienze di programmazione in altri linguaggi, come l'introspezione, una gestione particolarmente avanzata della memoria, l'ambiente di runtime (che rende Objective-C un linguaggio estremamente versatile).

Tutti gli esempi presentati in questa *Introduzione al Linguaggio Objective-C* sono originali, autocontenuti e funzionanti. Sono stati tutti compilati con gcc version 3.3 20030304 (Apple Computer, Inc. build 1495) incluso in XCode 1.1 di MacOS X 10.3.2.

Che cosa faremo

Questo vuole essere un tutorial di base per l'apprendimento dei fondamenti dell'Objective-C. È adatto a chi sa poco o nulla di programmazione ma è interessato all'argomento. È richiesta una conoscenza di base del C; l'applicazione di quanto qui imparato, benché possibile virtualmente su ogni sistema operativo dotato di compilatore Objective-C, sarà comunque esplicitamente rivolta a MacOS X.

Che cosa non faremo

Non ci addentreremo (ancora) nella discussione delle classi contenute nelle librerie Cocoa di MacOS X, né degli aspetti più profondi e più difficili della gestione della memoria e dell'ambiente runtime di Objective-C.

1. Programmare "ad oggetti"

Linguaggi "procedurali" e linguaggi "ad oggetti"

La diatriba tra i linguaggi cosiddetti "procedurali" come il C e i linguaggi cosiddetti "ad oggetti" o "object oriented" come il C++ o l'Objective-C è vecchia grosso modo quanto i linguaggi stessi di programmazione. Infatti, non appena le risorse hardware sono state sufficientemente potenti da consentire lo sviluppo e l'esecuzione di programmi e sistemi operativi un po' più che banali, le due filosofie di pensiero si sono scontrate, alleate e sovrapposte. Non bisogna pensare che siano una l'antitesi dell'altra, perché sarebbe scorretto e riduttivo. Sono piuttosto l'una il complemento dell'altra, e difficilmente si può essere dei puristi da questo punto di vista. Tuttavia, i linguaggi "ad oggetti" hanno spesso un vantaggio rispetto a quelli procedurali: essi, infatti, consentono l'inserimento anche massiccio di codice "procedurale" al loro interno, mentre i linguaggi procedurali in genere non consentono l'uso di "oggetti". Ma ora cerchiamo di capire di che cosa stiamo parlando.

Il C, l'abbiamo visto nel primo volume, è un linguaggio procedurale. Ciò che volete fare col vostro programma è descritto in una o più funzioni che operano su dei dati, immagazzinati nelle *variabili* del programma; esse saranno variabili numeriche, array di caratteri, o strutture, ma in ogni caso i dati su cui opera il programma sono sempre identificabili con le sue variabili (anche un file è, alla fine della fiera, una variabile all'interno del programma). Invece, il *modo* con cui manipolate i dati è rappresentato dalle funzioni di cui è costituito il programma. Se i dati che avete a disposizione sono di vario tipo e richiedono, di conseguenza, trattamenti specifici per ogni tipologia, sarà compito del vostro programma distinguere un tipo di dati da un altro e chiamare l'opportuna funzione per il loro trattamento.

In un linguaggio "ad oggetti", invece, l'approccio è diverso. Il programma, in sé, non sa nulla di dati e di funzioni per manipolare gli stessi. Il programma sa solo che l'utente vuole poter maneggiare un certo numero di oggetti in un certo numero di maniere più o meno predefinite. Come questo avvenga non è compito del programma saperlo. È compito degli *oggetti*. Un oggetto è, se volete, un'estensione del concetto di *struttura* del C: è simultaneamente una collezione di dati (e quindi di variabili), ma anche di funzioni atte alla loro manipolazione. In un programma ad oggetti, il programma stesso ubbidisce alla richiesta dell'utente del tipo "prendi questi dati e fai questa operazione"; ma se i dati sono di tipo diverso, e se di conseguenza l'operazione da fare è diversa a seconda del tipo, non c'è bisogno di una verifica preliminare all'interno del programma, come avviene per i linguaggi procedurali; invece, ogni oggetto *sa* che cos'è e che cosa è in grado di fare; di conseguenza, ogni oggetto, quando gli verrà chiesto di eseguire la tale operazione sui suoi dati, saprà esattamente *come* implementarla correttamente, perché ha piena coscienza di sé stesso, del tipo di dati che sta maneggiando e di che cosa ci si aspetta da lui.

Messa così sembra una roba piuttosto complicata, e in effetti, da un certo punto di vista, lo è. Per chi viene da un linguaggio procedurale come il C (come è accaduto a me a suo tempo), il passaggio ad un modo di pensare "ad oggetti" può essere traumatico e faticoso. In realtà, bisogna solo trovare la guida giusta: finché non si trova il libro, il tutorial, il manuale ecc. che affronta il problema della programmazione ad oggetti nel modo in cui ci viene congeniale vederlo affrontato, triboleremo. Per questo non so se questa *Introduzione all'Objective-C* piacerà a tutti: perché la scriverò così come *a me* pare che sia chiaro l'argomento, ma potrebbe non essere così per tutti. Se non avete grosse esperienze di programmazione alle spalle, probabilmente non farete una grande fatica ad imparare l'Objective-C. Se invece avete parecchia esperienza con linguaggi procedurali e nessuna con linguaggi "ad oggetti", potreste fare più fatica. Non demordete: il salto è molto meno duro di quanto sembri inizialmente; e, se con linguaggi tipo il C++, una volta compiuto il salto, potebbe venirvi voglia di tornare indietro, vi garantisco che con l'Objective-C questa tentazione non vi verrà mai!

Quando programmare "ad oggetti"

La soluzione a tutti i mali, purtroppo, non esiste. Questo è vero anche quando si parla di linguaggi di programmazione. Benché si senta spesso dire che la programmazione "ad oggetti" è migliore di quella procedurale, è più moderna, è da preferirsi, ecc., questo non è sempre e non è necessariamente vero. Come al solito, *dipende*.

Se il programma che volete fare è piccolo, è più che altro una prova, o un semplice programmino che fa un compito ben preciso e limitato, sempre uguale, forse un linguaggio procedurale è meglio. Se invece state realizzando un programma più ampio, il cui comportamento è maggiormente variabile, in cui l'input dell'utente è meno scontato di quanto si potrebbe pensare, forse gli oggetti fanno per voi. Se volete effettuare conti su conti, simulazioni numeriche e cose simili, la programmazione procedurale vi sarà forse più congeniale. Se invece avete la necessità di manipolare vari soggetti più o meno indipendenti, come le figure geometriche di un disegno tecnico o i personaggi di un gioco di ruolo, gli oggetti potrebbero essere la vostra salvezza.

Non c'è una ricetta generale per stabilire se per realizzare un certo programma sia meglio ragionare "proceduralmente" o "ad oggetti". A mio avviso, per risolvere questo dilemma occorrono due cose: conoscere almeno un linguaggio procedurale ed uno ad oggetti, ed amarli entrambi. Se non ne conoscete almeno uno per tipo, il problema non si pone, perché non disponete degli strumenti necessari per affrontare il problema nelle due maniere alternative; se non ne amate almeno uno per tipo, la vostra scelta sarà sempre condizionata dai vostri gusti, che magari nascono da una non corretta scelta del linguaggio di programmazione con cui affrontare il problema. Non ho la pretesa di poter sempre risolvere la questione. Ma se sapete un po' di C, allora avete idea di che cosa può fare e come lavora un linguaggio procedurale. Se sapete un po' di C++ o di Objective-C, allora avete idea di che cosa può fare e come lavora un linguaggio ad oggetti.

Perché Objective-C

Molti vi diranno che il C++ è, tra i linguaggi "ad oggetti", il più potente, il più diffuso, il più bello, il più qui e il più là. Non contesto né discuto queste affermazioni. Ma allora, perché Objective-C? Per almeno 3 ragioni:

1. È un linguaggio ad oggetti, quindi perché no?
2. È il linguaggio che Apple ha scelto per lo sviluppo di applicazioni di alto livello su MacOS X. Può piacere o no, ma è un fatto. E siccome, alla fine del gioco, vogliamo arrivare a programmare su MacOS X, Objective-C è una scelta quanto meno sensata.
3. Non sarà potente e diffuso come il C++, ma secondo me lo supera, e di gran lunga, sotto molti punti di vista. Vedremo sommariamente il perché nel prossimo capitolo. Vi basti sapere che, pur mancando di alcune delle caratteristiche più avanzate del C++, l'Objective-C rinuncia ai difetti del C (che invece il C++ eredita tutti, e alcuni li peggiora), e guadagna nuove caratteristiche che lo rendono un linguaggio a mio modo di vedere meraviglioso.

Padroni di pensarla diversamente. Ma se siete nuovi della programmazione ad oggetti, credo che troverete l'apprendimento dell'Objective-C molto più facile e molto più entusiasmante dell'apprendimento del C++. Provare per credere!

2. Objective-C e C++

Objective-C è meglio di C++

E chi l'ha detto? Lo dico io! Ma non è del tutto vero. Objective-C ha senza dubbio una sintassi più semplice e più chiara rispetto al C++. Facciamo un esempio: vogliamo chiamare una funzione (un *metodo*, dovremmo dire) che calcola l'area di un trapezio. Vi ricordo la formula: l'area del trapezio è la somma delle basi, moltiplicata per l'altezza, diviso 2. In C++ (e in C), dovremmo chiamare una funzione con del codice fatto più o meno così:

```
area=AreaTrapezio(baseMagg,baseMin,altezza);
```

Lì per lì è chiaro. Il problema è che se vado a vedere il prototipo della funzione `AreaTrapezio()`, esso è definito come `double AreaTrapezio(double,double,double)`, che, a ben vedere, non ci dice molto; infatti, non ci dice in quale ordine dobbiamo inserire le tre grandezze da passare come argomento, ovvero la base maggiore, la base minore e l'altezza.

In Objective-C la faccenda è più semplice. La stessa funzione la chiameremmo con un'espressione di questo genere:

```
area=[self areaConBaseMagg:baseMagg baseMin:baseMin altezza:altezza];
```

A parte il fatto che difficilmente scriveremo mai una cosa di questo genere (sia in C++ che in Objective-C scriveremmo, semmai, un oggetto di classe `Trapezio` che abbia un metodo chiamato `Area`), il prototipo della funzione in Objective-C è del tipo

```
-(double)areaConBaseMagg:(double) baseMin:(double) altezza:(double);
```

Come vedete, il nome stesso della funzione contiene non solo una descrizione di che cosa fa la funzione stessa (`area`), ma anche una descrizione di che cosa sono i vari argomenti; infatti, i vari `ConBaseMagg:`, `baseMin:` e `altezza:` sono *parte integrante* del nome della funzione, e non possono essere omessi. Benché, a ben vedere, ci sia più roba da scrivere, un programma in Objective-C è più semplice da leggere, perché anche quando le variabili che si passano ad argomento non hanno nomi molto evocativi, i nomi delle funzioni sono in genere sufficientemente completi da permettere di capire che cosa facciano le funzioni stesse e che cosa siano i loro argomenti. Magari non vi sembra un grande miglioramento, ma quando rimetterete mano a del codice che avete scritto qualche mese prima o, peggio ancora, scritto da altri, ringrazierete che la sintassi dell'Objective-C sia così evocativa.

Un'altra caratteristica che rende l'Objective-C migliore del C++ è il fatto che è un linguaggio*strongly typed* ma può anche non esserlo. Abbiamo detto, del C, che è *strongly typed* perché una variabile può contenere solo valori dello stesso *tipo* della variabile stessa; se il valore è di un tipo diverso e se una qualche regola di promozione o di troncamento è applicabile, il valore viene promosso o troncato così da combaciare col tipo della variabile. Se questo non è possibile, il compilatore genera un errore. Nel C++ è grosso modo la stessa cosa. Nell'Objective-C anche. Solo che con l'Objective-C potete rinunciare a questo controllo preventivo dei valori e delle variabili ed assegnare alle variabili stesse *tipi* generici, che possono contenere di tutto. Naturalmente, durante la compilazione, qualunque cosa decidiate di assegnare a queste variabili andrà sempre bene. Dove sta il vantaggio? Sta nel fatto che potete trattare *con la stessa variabile e con la stessa porzione di codice* i casi più disparati, ovvero i casi in cui nella vostra variabile sono contenuti valori (in Objective-C saranno più propriamente degli *oggetti*) di tipo molto differente, che in C o in C++ richiederebbero variabili e porzioni di codice distinte per essere trattati; ancora meglio: nella vostra variabile potete inserire valori (o oggetti) di tipo (*classe*, diremo per gli oggetti) che *ancora non esisteva quando avete compilato il programma*, ma che sono stati creati durante l'esecuzione stessa del programma per effetto di un comando dell'utente. Una potenzialità immensa: il vostro programma è espandibile durante

la sua stessa esecuzione! Naturalmente questo introduce da un lato alcuni livelli di complicazione e richiede che il codice sia opportunamente progettato per trarre vantaggio da questa caratteristica senza risultare confuso, ma dall'altro, una volta presa l'abitudine, costituisce una semplificazione impressionante nel modo di programmare. Una vera manna dal cielo. Nei prossimi capitoli cercheremo di vedere perché.

C++ è meglio di Objective-C

E chi l'ha detto? Lo dicono in molti. Ma non è del tutto vero. Però non bisogna dimenticare che il C++ ha degli indubbi vantaggi rispetto all'Objective-C. Sono molti, ma cercherò di dare una breve rassegna. Innanzitutto il C++ supporta le *eredità multiple*, ovvero una classe può ereditare da più classi anziché solo da una come in Objective-C (che però supporta protocolli e categorie proprio per ovviare a questo limite). Poi il C++ supporta un *polimorfismo* molto più ampio rispetto a quello supportato dall'Objective-C. *Polimorfismo* vuol dire che più funzioni possono avere lo stesso nome ma avere un diverso insieme di argomenti. Questo in C standard è vietatissimo: ogni funzione deve avere un nome unico e ben definito, e non possono esistere due funzioni con lo stesso nome e un diverso insieme di argomenti. In C++ questo è consentito, anzi, è utile: ad esempio si potrebbe voler espandere la funzione `double sin(double x)` fornita con le librerie matematiche e definita nella header `math.h` in modo da poterla usare anche quando l'argomento non è un numero reale ma un opportunamente definito numero complesso, secondo un prototipo di questo tipo:

```
complex sin(complex x);
```

È evidente che il nome della funzione viene mantenuto identico per far sì che quando, all'interno del codice, si chiama la funzione `sin()`, non occorre preoccuparsi se l'argomento sia un numero reale o un numero complesso; la corretta funzione specifica per l'argomento passato verrà automaticamente eseguita; naturalmente è compito del programmatore scrivere la funzione `sin()` nel caso in cui l'argomento sia una variabile di tipo `complex`; anche quest'ultimo deve essere opportunamente definito dal programmatore. Questo è comodo e, una volta presa la mano, è anche abbastanza intuitivo. L'Objective-C supporta qualche cosa di simile grazie alla possibilità di assegnare tipi generici alle variabili (anche a quelle passate per argomento alle funzioni), ma non è esattamente la stessa cosa.

Forse (a mio modo di vedere) la caratteristica del C++ che più manca all'Objective-C è la possibilità di assegnare polimorfismo persino agli operatori. Ad esempio, se avete due numeri reali e li volete sommare e volete che il risultato sia assegnato ad una variabile reale, in C scrivereste qualche cosa del tipo:

```
a = b + c;
```

e il gioco sarebbe fatto. Le regole di promozione o di troncamento assicurano che l'operatore `+` agisca indifferentemente su variabili di tipo `int`, `short`, `float`, `double` ecc., e il risultato sia coerente con l'operazione fatta. Tuttavia, se avete definito un numero complesso a esempio come una struttura, siete costretti ad eseguire la somma di due numeri complessi ad esempio con una funzione del tipo:

```
int SommaComplessa(complex addendo1,complex addendo2, complex *risultato);
```

il che è abbastanza scomodo; soprattutto se volete sommare un numero reale ad un numero complesso, perché vi toccherebbe definire un'altra funzione specifica per quel caso; e via discorrendo. Non sarebbe molto più bello se poteste scrivere direttamente `a = b + c` con almeno una delle tre variabili definita come numero complesso? Questo in C non lo potete fare, ma in C++ sì. E in Objective-C no. Non è una cosa che serve spesso, ma può essere molto comoda. A mio parere, è l'unica vera limitazione che l'Objective-C ha rispetto al C++. Per il resto, più o meno tutte le caratteristiche del C++ sono disponibili (anche se in forma diversa e con nomi diversi) in Objective-C; il viceversa, però, non è sempre vero.

La quadratura del cerchio

E allora? È difficile dire se sia meglio C++, Objective-C o qualche altro linguaggio. Francamente, non credo nemmeno che sia importante stabilirlo. Ogni problema che necessita di un programma per essere risolto andrà affrontato con gli strumenti opportuni, che potranno chiamarsi C, C++, Objective-C, Perl, Python, BASIC, ecc. Più linguaggi conoscete, più strumenti avrete nella vostra cassetta degli attrezzi, meglio verrà il lavoro che dovete svolgere.

Se avrete voglia, qui parleremo di Objective-C e cercheremo di comprenderne i pregi, il tutto nell'ottica di avere delle basi sufficienti per affrontare il terzo volume della saga, in cui, finalmente, creeremo dei veri programmi per MacOS X, con tanto di interfaccia grafica in puro stile Aqua.

Ora non è più tempo di cincischiare. Allacciate le cinture e preparatevi al capitolo più ostico: quello su classi ed oggetti!

3. Classi ed oggetti

Il mondo delle idee e il mondo reale

Soddisfatto dal vostro eccellente lavoro (si veda l'Esempio 22 dell'*Introduzione al linguaggio C*), il Servizio Meteorologico Nazionale vi incarica di mettere mano al vostro programma, di aggiungere nuove funzioni, di renderlo più versatile e, soprattutto, di renderlo facilmente espandibile. Per fare questo, oltre ad offrirvi un sacco di soldi, vi dà anche la possibilità di scegliere in quale linguaggio scrivere il programma. Voi, dal momento che state leggendo queste pagine, decidete ovviamente di scriverlo in Objective-C, perché ha tutte le caratteristiche necessarie per soddisfare le richieste del committente.

Infatti, scrivendo il programma in Objective-C, potrete sfruttare appieno le potenzialità di un modo di lavorare "ad oggetti", in cui le vostre strutture dati sono intrinsecamente legate alle funzioni che dovranno manipolarle. Gioiosi, lieti e festanti, iniziate pertanto a pensare alla struttura ad oggetti e classi del vostro programma.

Ma che cosa sono, in effetti, *oggetti* e *classi*?

Pensate ad una sedia, ad esempio quella su cui siete seduti. È indiscutibilmente una sedia, è dotata di tutte le caratteristiche che una sedia deve avere. Eppure è diversa da quella che c'è nell'altra stanza. Anche quella è indiscutibilmente una sedia, anche quella è dotata di tutte le caratteristiche che una sedia deve avere; eppure, le due sedie sono diverse l'una dall'altra. Le due sedie sono *oggetti*. Il *concetto di sedia*, invece, ovvero *l'insieme di proprietà, caratteristiche e requisiti che una sedia vera e propria deve avere* è una *classe*.

In Objective-C, quindi, parleremo di classi quando vorremo descrivere un particolare modo di immagazzinare dati e manipolarli, parleremo di oggetti (appartenenti ad una certa classe) quando prenderemo in considerazione un ben preciso insieme di dati che verranno immagazzinati e manipolati con le modalità previste dalla classe a cui appartiene l'oggetto. Ecco quindi una nozione importante: *ogni oggetto è un "caso particolare" di una classe*; diremo che è un'*istanza* di una classe. Esistono le sedie (classe), che devono avere certi requisiti e fare certe cose. La sedia su cui sono seduto io (oggetto) è un caso particolare, un'istanza della classe "sedie".

Quando si progetta un programma ad oggetti, è pertanto necessario chiedersi innanzitutto su quali dati dovrà operare il programma, chiedersi come sia possibile suddividerli in oggetti, quindi definire le classi a cui questi oggetti apparterranno; solo quando si avranno chiare in mente queste idee sarà possibile iniziare a scrivere il programma.

Nel nostro caso, il Servizio Meteorologico Nazionale ci ha chiesto un programma che permetta di fare le seguenti cose: definire un certo numero di località; inserire, per ogni località, temperatura e umidità a mezzogiorno di ogni giorno; mantenere una media aggiornata delle temperature e umidità inserite. I nostri oggetti saranno pertanto le varie località, con le rispettive temperatura e umidità medie. La classe a cui apparterranno i nostri oggetti definirà un certo numero di variabili, in cui saranno memorizzati nome della località, temperatura media, umidità media, e quant'altro servirà, e un certo numero di funzioni o *metodi*, il cui compito sarà quello di gestire e manipolare queste variabili e permettere l'inserimento dei dati e la comunicazione dei risultati.

Aprite pertanto il vostro editor di testo preferito (SubEthaEdit, mi, ProjectBuilder, XCode, BBEdit Lite, TextEdit, o qualunque altro editor di solo testo vi piaccia), e digitate il seguente codice:

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

#define YES         1
#define NO          0

@interface localita : NSObject
```

```
{
    NSString    *nome;
    double      temperaturaMedia;
    double      umiditaMedia;
    int         numeroGiorni;
}

- (id)initWithName:(NSString *)unNome;
- (NSString *)nome;
- (void)nuoviValori;
- (void)mostraValori;
- (void)nuovaTemperatura:(double)unaTemperatura;
- (void)nuovaUmidita:(double)unaUmidita;

@end
```

Salvatelo col nome localita.h, quindi create un nuovo documento col vostro editor di testo preferito e digitate quanto segue:

```
#import "localita.h"

@implementation localita

- (id)initWithName:(NSString *)unNome
{
    [super init];
    nome=[[NSString alloc] initWithString:unNome];
    temperaturaMedia=0.0;
    umiditaMedia=0.0;
    numeroGiorni=0;
    return self;
}

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

- (NSString *)nome
{
    return nome;
}

- (void)nuoviValori
{
    double  t,u;

    printf("\n");
    printf("Citta': %s\n",[nome cString]);
    printf("Inserisci la temperatura di oggi a mezzigiorno (gradi C): ");
    scanf("%lg",&t);
```

```
    printf("Inserisci l'umidita' di oggi a mezzigiorno: ");
    scanf("%lg",&u);
    [self nuovaTemperatura:t];
    [self nuovaUmidita:u];
}

- (void)mostraValori
{
    printf("\n");
    printf("Citta': %s\n",[nome cString]);
      printf("%s: temperatura media: %g (gradi C) e umidita' media:
%g\n",[nome cString],temperaturaMedia,umiditaMedia);
       printf("Le medie sono state calcolate per un totale di %d
giorni\n\n",numeroGiorni);
}

- (void)nuovaTemperatura:(double)unaTemperatura
{
    temperaturaMedia=(temperaturaMedia*numeroGiorni+unaTemperatura)/(numero
Giorni+1);
    numeroGiorni++;
}

- (void)nuovaUmidita:(double)unaUmidita
{
    umiditaMedia=(umiditaMedia*(numeroGiorni-1)+unaUmidita)/numeroGiorni;
}

@end
```

Salvatelo col nome localita.m (l'estensione .m indica che il file in questione contiene codice scritto in Objective-C e non in C standard). Aprite ancora un nuovo documento col vostro editor di testo preferito e digitate ancora quanto segue:

```
#include <stdio.h>
#import "localita.h"
#import <Cocoa/Cocoa.h>

#define kNuovaCitta         1
#define kInserisciValori    2
#define kMostraRisultati    3
#define kEsci               0

BOOL        gDone=NO;
NSMutableArray      *citta;
NSAutoreleasePool   *ap;

void MenuPrincipale(void);
void NuovaCitta(void);
void InserisciValori(void);
void MostraRisultati(void);
void Esci(void);
```

```
int main(void)
{
    ap=[[NSAutoreleasePool alloc] init];
    citta=[[NSMutableArray alloc] initWithCapacity:1];
    do
    {
        MenuPrincipale();
    } while(gDone==NO);
    return 0;
}

void MenuPrincipale(void)
{
    int     scelta;

    printf("Menu principale:\n\n");
    printf("1. Nuova citta'\n");
    printf("2. Inserisci nuovi valori\n");
    printf("3. Mostra i risultati\n");
    printf("\n");
    printf("0. Esci\n");
    printf("\n");
    printf("Inserisci la tua scelta: ");
    scanf("%d",&scelta);

    switch(scelta)
    {
        case kNuovaCitta:
            NuovaCitta();
            break;
        case kInserisciValori:
            InserisciValori();
            break;
        case kMostraRisultati:
            MostraRisultati();
            break;
        case kEsci:
            Esci();
            break;
        default:
            printf("Inserisci un numero compreso tra 0 e 3!\n\n");
            break;
    }
}

void NuovaCitta(void)
{
    NSString    *nome;
    char        *nomeC;

    nomeC=calloc(40,sizeof(char));
    printf("\n");
```

```
    printf("Inserisci un nome per la nuova citta': ");
    scanf("%s",nomeC);
    nome=[[NSString alloc] initWithCString:nomeC];
    [citta addObject:[[localita alloc] initWithName:nome]];
    [nome release];
    free(nomeC);
}

void InserisciValori(void)
{
    NSEnumerator    *en;
    id              obj;

    en=[citta objectEnumerator];
    while(obj=[en nextObject])
        [obj nuoviValori];
}

void MostraRisultati(void)
{
    NSEnumerator    *en;
    id              obj;

    en=[citta objectEnumerator];
    while(obj=[en nextObject])
        [obj mostraValori];
}

void Esci(void)
{
    gDone=YES;
    [citta release];
    [ap release];
}
```

Salvatelo come main.m. Ora è giunto il momento di compilare il nostro programma. Siccome è la prima volta che facciamo una cosa del genere in Objective-C, vediamo la cosa nel dettaglio col nostro primo box di approfondimento:

Compilare ed eseguire programmi Objective-C

Aprite l'applicazione Terminal, la trovate nella cartella Utilities dentro la cartella Applicazioni. Vi comparirà una scritta del tipo:

`[computer ~] utente%`

oppure

`computer:~ utente$`

dove al posto di `computer` ci sarà il nome del vostro computer e al posto di `utente` ci sarà il nome dell'utente col quale avete fatto il login (probabilmente il vostro nome o cognome). Il segno "~" indica dove vi trovate nell'albero delle cartelle sull'hard disk, ovvero nella vostra home folder (/Users/vostronome/).

Andate nella cartella in cui avete salvato i file dell'Esempio1: assumendo che abbiate creato una cartella EsempiObjective-C all'interno della vostra home folder, digitate:

`cd EsempiObjective-C`

e premete invio oppure return. La scritta diventerà allora

```
[computer ~/EsempiObjective-C] utente%
```

oppure

```
computer:~/EsempiObjective-C utente$
```

Digitate

```
ls
```

per vedere che cosa c'è all'interno della cartella. I file localita.h, localita.m e main.m dovrebbero essere visibili.

Ora dobbiamo compilarli, ovvero trasformarli in qualche cosa di eseguibile dal computer. Però, rispetto a quando lavoravamo sul C nella prima puntata di questa trilogia, ora il nostro programma è fatto da tre file, di cui due di codice e uno di header. La procedura di compilazione, pertanto, sarà un po' più complicata.

Iniziamo a trasformare in file *oggetto* il file localita.m. Un file oggetto è la versione compilata *ma non ancora eseguibile* di un file di codice; compilata, perché il compilatore ha trasformato il listato (scritto in Objective-C nel nostro caso) in una sequenza di istruzioni comprensibili al microprocessore, ma non ancora eseguibile, perché ancora manca il *linking*, ovvero quell'operazione che consente al programma di poter utilizzare le librerie messe a disposizione dal sistema operativo. Virtualmente, ogni programma ha bisogno di essere *linkato* rispetto a qualche libreria. Il compilatore `gcc` esegue automaticamente l'operazione di linking, a meno che non gli venga detto esplicitamente di non farla. Noi ora gli diremo di non farla, perché il nostro programma è costituito da due file di codice, che *prima* dovremo compilare, e *solo successivamente* linkeremo trasformandoli nel programma vero e proprio eseguibile dall'utente.

Iniziamo pertanto digitando, sul terminale, il comando:

```
gcc -c localita.m
```

e premete invio o return alla fine. L'opzione `-c` indica proprio che volete compilare ma non linkare il file sorgente. Se non avete fatto errori, dopo pochi istanti il compilatore avrà finito. Digitate

```
ls
```

e vedrete che, oltre ai file precedenti, se n'è aggiunto un altro, localita.o, il file *oggetto* di localita.m; è da notare che con questa operazione abbiamo già lavorato anche su localita.h, in quanto si tratta del file di header di localita.m, e pertanto è stato automaticamente preso in considerazione.

Ora compiliamo il file main.m:

```
gcc -c main.m
```

e il file main.o verrà creato. Ora linkiamo il tutto:

```
gcc -o Esempio1 localita.o main.o -lobjc -framework Cocoa
```

Con questo comando abbiamo creato il file eseguibile Esempio1 (l'opzione `-o`), mettendo insieme i file localita.o e main.o creati prima, utilizzando la *libreria* Objective-C (l'opzione `-lobjc`) e il *framework* (che poi è una specie di libreria) Cocoa, cosa che rende questo programma linkabile ed eseguibile solo con MacOS X, e non con altri sistemi operativi per i quali sia disponibile l'Objective-C (l'opzione `-framework Cocoa`); la ragione di questa scelta è che Cocoa ci mette a disposizione alcune classi usabili con Objective-C che ci tornano utilissime per i nostri programmi.

Ora possiamo finalmente eseguire il programma digitando:

```
./Esempio1
```

e, come al solito, premendo invio o return alla fine.

Giocate un po' con il programma, inserendo dapprima due o tre nomi di città, poi inserendo qualche valore di temperatura ed umidità. Notate che potete aggiungere nuove città anche quando avete già inserito dei valori per quelle immesse in precedenza. Visualizzate temperature e umidità medie per tutte le vostre località. Poi, quando vi siete stufati e siete ansiosi di capire come funziona questo gioiellino, proseguite nella lettura.

Interfaccia di una classe

Iniziamo con l'esaminare il file localita.h. Come l'estensione suggerisce, è un file di header, quindi ci aspettiamo di trovare definizioni, costanti, le dichiarazioni delle variabili che saranno "globali" per i file di codice che faranno uso di questo file di header, prototipi di funzioni. Tuttavia, essendo questo file associato ad un file di codice che contiene una classe Objective-C, lo chiameremo più propriamente un file di *interfaccia*, o, più semplicemente, l'*interfaccia della classe*.

Essa incomincia subito con un comando nuovo, `#import <Cocoa/Cocoa.h>`: il comando `#import` non è molto diverso da `#include`, anzi, è meglio, perché assicura che il file di header specificato sia importato solo se necessario, ovvero solo se non è ancora stato importato in precedenza. Infatti, in questo programma le cose sono piuttosto semplici, ma in programmi più complessi, dove i file di header vengono letti e inclusi in molti file di codice, è facile cercare di includere più volte lo stesso file di header; il comando `#include`, in questi casi, genera degli errori in fase di compilazione. Il comando `#import`, invece, disponibile solo in Objective-C, risolve questo problema. Il file di header che viene importato è Cocoa.h: esso contiene tutte le definizioni e i prototipi necessari per usare all'interno del nostro programma le classi Cocoa, sviluppate da Apple per rendere più agevole la programmazione di MacOS X.

Dopo un paio di definizioni, la *direttiva* `@interface localita : NSObject` segnala che è iniziata la vera e propria interfaccia della classe. La classe di cui stiamo scrivendo l'interfaccia si chiama `localita`, e sarà utilizzata nel nostro programma per tenere traccia di tutte le proprietà di ogni località, ovvero nome, temperatura ed umidità medie, ecc. Accanto al nome della classe, dopo il due punti, è segnata la *classe genitrice*, ovvero la classe da cui `localita` *eredita*: nel nostro caso `NSObject` (si dice che localita è una *sottoclasse* di `NSObject`). Ci occuperemo nel capitolo 4 di ereditarietà e sottoclassi. Per ora basti sapere che, a meno di esigenze specifiche, una classe deve *sempre* ereditare da `NSObject` *o da una sua sottoclasse*. A differenza del C++, Objective-C non permette di definire più di una classe genitrice.

Tra parentesi graffe sono indicate tutte le variabili di cui la classe `localita` avrà bisogno: due variabili di tipo `double` memorizzeranno la temperatura media e l'umidità media, rispettivamente, della località il cui nome è memorizzato in una variabile di tipo puntatore a `NSString`: si tratta di una classe definita all'interno del framework Cocoa (ecco perché abbiamo importato Cocoa.h all'inizio) che permette di gestire stringhe di caratteri in modo molto più semplice e potente che non con gli strumenti standard messi a disposizione del C e dell'Objective-C. Tecnicamente, qui non è che ci serva più di tanto usare la classe `NSString`, avremmo potuto tranquillamente lavorare con un puntatore a `char` o un'array di `char`, ma l'uso della classe `NSString` era didattico! Che sia una classe definita all'interno del framework Cocoa è indicato anche dal suo nome: tutte le classi il cui nome inizia con NS (che sta per NextStep, il sistema operativo sviluppato dalla Next di Steve Jobs e da cui è derivato MacOS X) sono definite all'interno del framework Cocoa (che, di fatto, si compone di due altri framework, l'*Application Kit* e il *Foundation Kit*). Una regola generale è che in Objective-C un oggetto va *sempre* identificato come un *puntatore* alla classe di appartenenza; non è possibile, è esplicitamente vietato creare oggetti *staticamente* (come invece si può fare in C++). Quindi, la variabile (abituiamoci a dire l'*oggetto*) `nome` sarà di tipo `NSString *`. Infine, una variabile di tipo `int` terrà memoria di quanti giorni siano già stati conteggiati nel computo delle medie.

Così come sono state dichiarate, le variabili sono visibili (ovvero leggibli e scrivibili) solo all'interno degli oggetti appartenenti alla classe (e alle sue sottoclassi, come vedremo nel capitolo 4); nessuna porzione di codice esterno alla classe potrà accedere a queste variabili. È possibile variare queste impostazioni, come vedremo sempre nel capitolo 4.

Dopo la parentesi graffa di chiusura sono indicati i prototipi dei *metodi* della classe, ovvero di quelle funzioni che avranno il compito di manipolare i dati memorizzati nelle variabili della classe ed eventualmente di comunicare con l'esterno. La sintassi con cui si dichiarano i prototipi è diversa da quella del C: un segno "meno" posto all'inizio è obbligatorio ed indica che si tratta, per l'appunto, di un metodo che potrà essere chiamato *da ogni oggetto della classe*. Di per sé la cosa potrebbe sembrare ovvia, ma c'è una sottigliezza: esistono dei casi in cui avete

bisogno di chiamare un metodo di una classe *quando ancora l'oggetto appartenente a quella classe non esiste*; tipicamente questa situazione si realizza quando state per creare un oggetto appartenente ad una classe ma, prima di costruirlo, avete bisogno di riservare (allocare) un quantitativo sufficiente di memoria; è compito della classe, e non dell'oggetto (che non esiste ancora), effettuare queste operazioni, mediante opportuni metodi, detti *metodi di classe*, che sono identificati (nel prototipo) da un simbolo + anziché -. Potrebbe venirvi a questo punto il dubbio che anche nel nostro caso sia necessario implementare un metodo di classe che si occupi di gestire la memoria necessaria per creare gli oggetti di cui avremo bisogno (le varie località di cui terremo memoria dei dati meteorologici principali). È vero, dobbiamo fare tutto questo. Il fatto però di aver dichiarato la classe `localita` come sottoclasse di `NSObject` ci semplificherà enormemente la vita (come vedremo tra non molto), al punto che non abbiamo nemmeno bisogno di mettere, tra i prototipi, nessun metodo di classe.

A seguire il segno "meno" troviamo, tra parentesi tonde, il tipo restituito dal metodo, che può essere qualsiasi tipo di variabile valido in C ed Objective-C. Anche qui abbiamo il fatto che un metodo, `nome`, restituisce un oggetto puntatore a `NSString` (guarda caso il nome della località), mentre un altro, `initWithName:`, restituisce un oggetto di tipo `id`: in Objective-C, `id` è un tipo speciale, che identifica un oggetto appartenente a qualsivoglia classe, purché sia sottoclasse di `NSObject`. È un esempio dell'assegnazione dinamica di tipo di cui dissertavamo nei capitoli precedenti: in Objective-C possiamo riferirci a delle variabili (a degli oggetti) senza specificare in anticipo di che tipo saranno; si può usare un tipo generico, `id`, e poi, in fase di esecuzione, quel che sarà sarà. Benché nulla vieti, in questo esempio, di specificare che il metodo `initWithName:` restituisce un oggetto di tipo (classe) `localita` (perché così è), lasciamo, come è convenzione fare, il tipo a `id`, perché ci tornerà immensamente utile nel prossimo capitolo, quando creeremo una sottoclasse di `localita`. Tra l'altro, avete notato che quando parlo del metodo `initWithName:` includo sempre i due punti? Questo perché il metodo in questione richiede un argomento, e pertanto i due punti *fanno parte integrante del nome del metodo*. Ometterli costituirebbe un errore di sintassi.

Se un metodo, così come viene dichiarato nell'interfaccia della classe, non ha un tipo esplicitamente assegnato, si assume che restituisca `id`. La dichiarazione dell'interfaccia di una classe termina con la direttiva `@end`, senza nessun punto e virgola alla fine.

Riassumendo: in un file di interfaccia importiamo innanzitutto tutte le header che ci servono (tipicamente quelle riguardanti il framework Cocoa) perché le variabili e i metodi della classe siano ben definiti. Poi definiamo tutte le costanti che ci servono. Quindi specifichiamo da chi eredita la nostra classe, avendo cura che si tratti sempre di `NSObject` o di una sua sottoclasse (per comodità, come vedremo tra poco). Tra parentesi graffe dichiariamo quindi tutte le variabili di cui farà uso la nostra classe, variabili che sono utilizzabili solo dagli oggetti della classe e da nessun altro. Per sovrascrivere questo comportamento, dobbiamo aspettare il prossimo capitolo (dove capiremo anche quando e perché convenga farlo). Dopo la dichiarazione delle variabili, e al di fuori delle parentesi graffe, dichiariamo i metodi implementati dalla classe, preceduti dal segno "meno" e specificando tra parentesi tonde il tipo di ognuno di essi. Benché, formalmente, tutti i metodi siano uguali, alcuni di essi saranno da considerare "privati" ed altri *metodi accessori*, il cui significato e uso sarà discusso tra poco.

Siete un po' spaventati da tutte queste novità? Avete le idee confuse? Ci sono un sacco di cose che state accettando per fede perché non ne vedete, ora come ora, la necessità o l'utilità? Non preoccupatevi, è normale. Adesso che disquisiremo un po' dell'*implementazione* della classe `localita` un po' di cose inizieranno a diventarvi più chiare. Col prossimo capitolo, poi, altre cose si inquadreranno meglio e troveranno un posto nel ricco e variopinto mondo dell'Objective-C.

Implementazione di una classe

Se il file di interfaccia è una dichiarazione di intenti (la mia classe userà queste variabili e implementerà questi metodi), il file di implementazione è quello in cui si fa il lavoro sporco, quello in cui il codice Objective-C vero e proprio viene scritto. Nel nostro esempio, se localita.h era il file di interfaccia, è naturale pensare che localita.m sia il file di implementazione; e in effetti è così. Vediamo com'è fatto.

Innanzitutto si importa *sempre* il file di header che contiene l'interfaccia della classe. Se così non si facesse, sarebbe inutile creare il file di interfaccia. In effetti, tutto quanto è contenuto nel file di interfaccia potrebbe essere copiato e incollato all'inizio del file di implementazione, evitando così di avere a che fare con due file distinti. Tuttavia, tenere l'interfaccia e l'implementazione separate ha dei vantaggi: potete importare il file di interfaccia di una certa classe in tutti i file di implementazione di classi che lavorano con oggetti appartenenti a quella classe; e poi, potete usare i file di interfaccia come "documentazione": tutto ciò che c'è da sapere su questa classe (che cosa fa) senza aver bisogno di andare a guardare l'implementazione vera e propria (come lo fa).

L'implementazione vera e propria inizia con la direttiva `@implementation` seguita dal nome della classe. È necessario specificare quest'ultimo perché un file potrebbe contenere le implementazioni di più classi, anche se non è prassi comune farlo (anzi, è sconsigliato). Quindi, i metodi che la classe deve implementare sono riportati col loro prototipo, ma, al posto del punto e virgola finale, con la solita famosa coppia di parentesi graffe all'interno della quale si trovano le istruzioni che il programma eseguirà quando il metodo verrà chiamato. Come vedete, i metodi non sono poi molto diversi dalle funzioni. L'ordine con cui compaiono i metodi non è essenziale, purché compaiano tutti quelli che avete dichiarato nel file di interfaccia. In realtà ne possono comparire anche di più (con alcune regole, ovviamente), come vedremo tra un po'. Quando avete finito di scrivere il codice per tutti i metodi, la direttiva `@end`, sempre senza punto e virgola finale, chiude l'implementazione della classe.

Onde evitare di dimenticare pezzi per strada, io ho l'abitudine di implementare i metodi nello stesso ordine con cui li dichiaro, a cominciare dal metodo di *inizializzazione*. Ogni classe deve averne almeno uno, deve restituire un tipo `id` e il suo nome deve essere `init` o per lo meno deve iniziare con init (è una convenzione, non un obbligo, ma è meglio seguirla). Compito di ogni metodo di inizializzazione è affrontare nella maniera più semplice possibile quelle questioni barbose e complicate riguardanti l'allocazione della memoria che serve all'oggetto che sta per essere creato ed assegnare alle sue variabili dei valori iniziali.

Nel nostro caso, ogni volta che vogliamo monitorare temperatura e umidità medie di una certa località, iniziamo a dare un nome alla località stessa; il nostro programma, allora, chiamerà il metodo `initWithName:` della classe `localita` fornendogli, come argomento, un oggetto di tipo `NSString` contenente il nome della località scelta dall'utente. Il metodo di inizializzazione inizia in maniera standard (ovvero: *tutti* i metodi di inizializzazione *devono* iniziare così), con l'istruzione `[super init]`. Mamma mia! Qui succedono già un sacco di cose! Vediamole una per una.

Innanzitutto siamo di fronte ad un nuovo costrutto sintattico, ovvero il modo che Objective-C ha per "chiamare" i metodi di un oggetto: il nome dell'oggetto che si vuole usare compare per primo tra parentesi quadre, seguito, sempre tra le parentesi, dal nome del metodo da chiamare. Sostanzialmente, come prima cosa, il nostro metodo di inizializzazione sta dicendo all'oggetto`super` di eseguire il suo metodo denominato `init`. L'oggetto `super` è definito all'interno di ogni classe come l'oggetto che punta (ricordatevi che in Objective-C gli oggetti sono sempre puntatori!) alla *superclasse* o alla *classe genitrice* della classe di cui state scrivendo l'implementazione; poiché `localita` è una sottoclasse di `NSObject`, `super` punta ad un oggetto di classe `NSObject`. E qui sta la grande comodità di Cocoa: come si fa a riservare (allocare) memoria sufficiente per farci stare l'oggetto di classe `localita` che l'utente vuole creare? Quanta memoria serve? Come facciamo ad essere sicuri che ce ne sia abbastanza? Come facciamo a trovarla? E chi se ne frega? L'aver dichiarato la classe `localita` come sottoclasse di `NSObject` ci permette di risolvere tutti questi problemi in un colpo solo, semplicemente scrivendo `[super init]`; l'oggetto di classe `NSObject` a cui punta `super` penserà a tutto questo per noi. Comodo, invero! Non dimenticate mai di scrivere questa istruzione per prima. Se non lo fate, il vostro programma farà cose veramente molto strane, e sarà molto difficile capire perché.

Segue un'istruzione il cui compito è quello di memorizzare nella variabile `nome` il nome (ma guarda un po') della località scelta dall'utente. Poiché `nome` è un oggetto di tipo `NSString`, dobbiamo innanzitutto eseguire il *metodo di classe* `alloc` della classe `NSString`. `alloc` è un metodo implementato da `NSObject` e, automaticamente, da tutte le sue sottoclassi (come

localita e NSString) che si occupa di riservare memoria sufficiente alla creazione di un oggetto, prima, ovviamente, che questo venga creato (ecco perché è un metodo di classe). Essendo un metodo di classe, tra parentesi quadre non dobbiamo mettere il nome di un oggetto, ma il nome di una classe. [NSString alloc], allora, vuol dire che vogliamo creare un oggetto di tipo NSString. Il metodo alloc restituisce sempre un tipo id, quindi un oggetto, che nel caso specifico appartiene alla classe specificata (NSString nel nostro caso). Tale oggetto è allocato (abbiamo preparato la memoria necessaria col metodo alloc), ma non inizializzato (non abbiamo suddiviso la memoria tra le varie variabili che verranno utilizzate). La differenza è la stessa che c'è tra il destinare un certo appezzamento di terra alla costruzione di una villetta con giardino (allocazione) e il progetto effettivo della villetta e del giardino (inizializzazione). La villetta e il giardino veri e propri saranno l'oggetto, che potrà essere usato solo dopo che l'allocazione sarà stata fatta (il comune, ovvero il Sistema Operativo, ci ha dato la proprietà del terreno e ci ha dato la licenza edilizia) e l'inizializzazione sarà stata completata (il progetto della villetta e del giardino).

Il metodo alloc, dicevamo, restituisce un oggetto; non è che ce ne facciamo molto di questo oggetto, dal momento che, non essendo ancora inizializzato, non è utilizzabile. E qui viene una delle grandi comodità dell'Objective-C: la sua sintassi che io definisco "a scatole": la "scatola" [NSString alloc] è un oggetto ancora non inizializzato; devo inizializzarlo. Potrei assegnare tale oggetto ad una variabile (mioOggetto), e poi inizializzarlo in seguito ([mioOggetto initWithString:]), ma sarebbe uno spreco di variabili e di tempo. Molto meglio inscatolare l'oggetto [NSString alloc] nella scatola che conterrà l'istruzione di inizializzazione dell'oggetto stesso, ovvero [[NSString alloc] initWithString:]. Il metodo initWithString:, come tutti i medoti inizializzatori, restituisce un tipo id, quindi un oggetto. Esso è proprio l'oggetto nome, ovvero la NSString contenente il nome della località scelta dall'utente e che dovremo tenere in memoria.

Quando create un nuovo oggetto in un metodo di inizializzazione è bene che siate consapevoli che MacOS X richiede che seguiate alcune regole di base per non sprecare memoria (e per non rischiare di mandare in crash la vostra applicazione a causa di un non corretto uso della stessa). Queste regole, che a voler approfondire un po' la cosa possono diventare molto complicate, vanno scrupolosamente seguite; io stesso ho penato un po' per impararle e soprattutto per convincermi che fosse veramente importante seguirle. Devo infiniti ringraziamenti a Gian Luca Cannata e alla sua pazienza e competenza per avermi iniziato al mondo della gestione della memoria in Objective-C e avermi fatto capire là dove sbagliavo (pressocché ovunque). Dunque, benché la gestione avanzata della memoria non faccia parte degli scopi di questo manuale, qui vediamo giusto i fondamenti necessari per poter proseguire con tranquillità: quando creiamo un oggetto con la sintassi [[NomeClasse alloc] init] o con qualche variante del genere, stiamo sostanzialmente dicendo al sistema operativo di riservare spazio in memoria per l'oggetto che ci interessa e di tenerlo lì per il futuro, perché ci servirà. Nel nostro caso, l'oggetto nome verrà accuratamente conservato nella memoria del programma. Ma se, un bel giorno, non ne avremo più bisogno, il sistema operativo ci chiede di farglielo sapere. Il modo giusto per farlo è trasmettere all'oggetto in questione il *messaggio* (metodo) release, come vedremo fra poco.

Incidentalmente, stiamo apprezzando il fatto che qualunque oggetto sia in una maniera o nell'altra "figlio" di NSObject: i metodi alloc e release e vari altri sono implementati dalla classe NSObject, quindi da Cocoa, quindi da MacOS X, e li possiamo usare *con qualunque oggetto*, ivi compresi quelli che creiamo noi (come gli oggetti appartenenti alla classe localita), senza alcun bisogno di implementarli esplicitamente: infatti, quando di un oggetto si invoca un metodo che non è stato scritto esplicitamente nella sua classe, Objective-C risale alla classe genitrice e le chiede di eseguire il metodo in questione; se questa non sa che farsene, il metodo viene passato alla classe genitrice della classe genitrice dell'oggetto originale (la classe nonna?), e così via, fino a che qualcuno, fosse anche NSObject, non ha a disposizione il codice vero e proprio che implementa questo metodo. Naturalmente, se voi chiamate il metodo cicciopirillo e poi non lo implementate, è molto probabile che nessuno, nemmeno NSObject, lo implementi; in questo caso, la chiamata al metodo viene lasciata cadere, e il vostro programma genera un messaggio di avviso che viene memorizzato nel file console.log.

Nel metodo `initWithName:` ci sono poi semplici istruzioni di assegnazione a variabile: la località è appena stata creata, quindi le grandezze `temperaturaMedia`, `umiditaMedia` e `numeroGiorni` sono pari a zero.

Per finire, siccome tutti i metodi di inizializzazione *devono* restituire un oggetto (sé stessi), scriviamo `return self`; `self` non è nient'altro che un puntatore all'oggetto stesso. Ogni oggetto "sa" chi è, gli basta guardare nella variabile predefinita `self`.

Il secondo metodo che troviamo nel file localita.m è `dealloc`: come avrete notato, esso non è presente tra i prototipi nel file di interfaccia. Come mai? Il metodo `dealloc` è implementato dalla classe `NSObject` e quindi da tutte le sue sottoclassi (come `localita`). Se, per qualche ragione, una sottoclasse vuole *sovracrivere* il metodo `dealloc` di `NSObject` per estenderne le funzionalità, basta che lo inserisca nel proprio file di implementazione, senza bisogno di metterlo anche nel file di interfaccia; se, comunque, nel dubbio lo mettete anche lì, male non fa. E quali sarebbero queste ragioni per cui una sottoclasse di `NSObject` vorrebbe sovrascirivere il metodo `dealloc`? Una, soprattutto: se nel metodo di inizializzazione (`initWithName:` nel nostro caso) avete creato uno o più oggetti con una chiamata al metodo `alloc` di una classe e poi ad un qualche metodo di inizializzazione, allora sovrascrivere il metodo `dealloc` è *obbligatorio*; in esso, dovete liberare la memoria occupata da tutti gli oggetti che avete creato con una chiamata ai rispettivi metodi `release`.

È esattamente quello che facciamo qua: l'oggetto `nome` era stato creato con la chiamata `[[NSString alloc] initWithString:]` nel metodo `initWithName:`; ora lo rilasciamo con la chiamata `[nome release]`. Poi, siccome il metodo `dealloc` di `NSObject` si occupa di un sacco di cose utilissime come ad esempio liberare la memoria che il nostro oggetto di classe `localita` occupa (e che era stata allocata con la chiamata `[super init]` nel metodo di inizializzazione), non dimentichiamoci di invocare ancora il metodo `dealloc` della classe genitrice, mediante `[super dealloc]`; questa deve essere l'ultima cosa che facciamo nel metodo `dealloc` delle nostre classi.

Il metodo successivo, dal prototipo `(NSString *)nome`, è un cosiddetto *metodo accessorio* o *metodo di accesso*. Che cosa faccia è abbastanza evidente: quando chiamato, restituisce il contenuto della variabile `nome`, ovvero il nome della località. In realtà, nel nostro programma non useremo mai questo metodo, nel senso che nel file main.m, che si occuperà di organizzare tutto il lavoro, da nessuna parte il metodo `nome` verrà chiamato. E allora perché lo mettiamo? Per questioni di eleganza!

Come abbiamo già detto, le variabili di un oggetto non sono accessibili al codice esterno a quello di implementazione dell'oggetto stesso, a meno di introdurre speciali direttive che vedremo nel prossimo capitolo. In generale, introdurre queste direttive non è comunque una buona idea: proteggere le variabili di un oggetto dall'accesso esterno è infatti una strategia particolarmente importante. Se il codice esterno all'implementazione di un oggetto non sa di quali variabili abbia bisogno l'oggetto stesso, non può, né intenzionalmente né accidentalmente, modificarle, causando comportamenti strani o addirittura crash del programma. È importante che l'unico modo per modificare le variabili di cui fa uso un oggetto sia attraverso i metodi accessori, ovvero metodi che permettono di scrivere o di leggere il contenuto di una o più variabili dell'oggetto stesso. Può sembrare una complicazione, ma non lo è. Il nostro programma scrive nella variabile `nome` dell'oggetto di classe `localita` mediante il metodo `initWithName:`; noi, che abbiamo fatto tutto il programma, sappiamo che la variabile `nome` è un oggetto di classe `NSString`; avremmo potuto rendere questa variabile pubblica e lasciare che il programma, in un preciso punto del file main.m, scrivesse direttamente all'interno di questa variabile anziché chiamare il metodo `initWithName:`; solo che un bel giorno decidiamo di modificare la classe `localita`, perché ci viene più comodo memorizzare il nome della località in maniera diversa. Se lasciamo che il programma acceda alla variabile `nome` direttamente, dobbiamo modificarlo in tutti quei punti in cui la variabile `nome` viene letta o scritta, e potrebbero essere tanti. Se invece la variabile nome viene scritta *solo* col metodo di inizializzazione (o con un metodo di accesso in scrittura) e viene letta *solo* col metodo di

accesso in lettura, è tutto molto più facile: possiamo anche sbarazzarci della variabile `nome`, basta che lasciamo i due metodi di accesso: ci penseranno loro a memorizzare il nome in maniera opportuna, dal punto di vista del mondo esterno il nome della località verrà sempre letto e scritto nello stesso modo. Questo, se già fa comodo a noi, è di importanza fondamentale quando si lavora ad un progetto a molte mani: si definiscono le interfacce delle classi, soprattutto si specificano bene quali sono i metodi di accesso di una classe, così che tutti i programmatori sappiano come usare quella classe; chi si occuperà di implementarla effettivamente potrà fare quello che vuole, modificarla anche cento volte; ma se i metodi di accesso non cambiano, saranno fatti suoi, il lavoro degli altri programmatori sarà salvo. Su larga scala, questo torna molto utile anche a noi: tutte le classi del framework Cocoa (finora abbiamo accennato a `NSObject` ed `NSString`, tanto per fare degli esempi), possono essere modificate da Apple in qualunque momento con qualunque aggiornamento del sistema operativo. Questo però non vuol dire che tutti i programmi già realizzati smetteranno di funzionare: basta che i metodi di accesso siano sempre gli stessi! Grande cosa, la programmazione ad oggetti!

Un'ultima nota: il metodo `nome` ha lo stesso nome (eh! brutta cosa, i *nomi* uguali) della variabile `nome`: non c'è conflitto, lo *spazio dei nomi* dei metodi e delle variabili è separato; anzi, è prassi comune far sì che il metodo di accesso in lettura di una variabile abbia lo stesso nome della variabile stessa. È buona norma avere un metodo di accesso in lettura per lo meno di ogni variabile per la quale esiste un metodo di accesso in scrittura, fosse anche il metodo di inizializzazione; pertanto, anche se non lo useremo, inseriamo il metodo `nome` nell'interfaccia e nell'implementazione della nostra classe `localita`.

Il metodo successivo, `nuoviValori`, è, in senso lato, un metodo di accesso in scrittura; teoricamente, un metodo di accesso in scrittura accetta i valori da assegnare alle variabili di cui si occupa come argomenti; teoricamente. Non dico che questa non sia una buona prassi, ma non è sempre la scelta migliore. Nel nostro caso, e vedremo nel prossimo capitolo come la scelta fatta sia stata quanto mai opportuna, abbiamo bisogno di permettere all'utente di inserire, per ogni località, temperatura e umidità a mezzogiorno, al fine di calcolare le medie. Chi si deve occupare di questa operazione? Un buon candidato potrebbe essere il file main.m, il cuore del programma, colui che organizza tutto; ma se un giorno decidiamo che, oltre a temperatura e umidità, il nostro programma deve memorizzare anche la pressione atmosferica, siamo costretti a modificare la classe `localita` per tenere conto di queste modifiche *e anche* il file main.m, ovvero il cuore del nostro programma, perché dovrà chiedere all'utente di inserire un valore in più. Se invece affidiamo alla classe `localita` il compito di richiedere all'utente i valori da inserire, possiamo estendere alla pressione atmosferica le grandezze interessanti da monitorare, e il file main.m non avrà mai bisogno di saperlo: infatti, il solo metodo `nuoviValori` sarà coinvolto, oltre all'implementazione della classe `localita`, naturalmente. Questo fa molto "programmazione ad oggetti": gli oggetti di classe `localita` *sanno* di quali valori hanno bisogno per effettuare i loro conti; non è compito di main.m manipolare i dati memorizzati negli oggetti; compito di main.m è chiedere agli oggetti di manipolarseli loro, i loro dati, nella maniera che ritengono più opportuna.

`nuoviValori`, pertanto, non fa niente di particolarmente difficile: chiede all'utente di inserire temperatura e umidità a mezzogiorno per l'oggetto in questione (una ben precisa località); l'utente è informato di quale località si tratta grazie al fatto che un'istruzione `printf()` stampa a schermo la variabile nome; tuttavia, `printf()` non è in grado di gestire direttamente stringhe di testo appartenenti alla classe `NSString`; per fortuna, queste dispongono di un metodo, `cString`, che le converte in un formato comprensibile alla funzione `printf()`. Infine, una volta che i nuovi valori di temperatura e di umidità sono stati inseriti, i metodi `nuovaTemperatura:` e `nuovaUmidita:` vengono chiamati; essi sono ad esclusivo uso interno della classe `localita`, e hanno il compito di ricalcolare di volta in volta temperatura e umidità medie, rispettivamente, aggiornando le rispettive variabili di tipo `double` dichiarate nell'interfaccia della classe. Nessuna di queste variabili dispone di metodi diretti di accesso in scrittura, quindi non dispone nemmeno di metodi di accesso diretto in lettura. Il loro metodo di accesso "indiretto" in scrittura è proprio `nuoviValori`, che infatti è accompagnato da un

analogo metodo di accesso in lettura: `mostraValori`. Questo non fa altro che scrivere a schermo, mediante opportune funzioni `printf()`, il nome della località e temperatura media e umidità media a mezzogiorno relative agli ultimi `numeroGiorni` giorni.

Eseguire un programma

Con l'analisi dei file localita.h e localita.m ci siamo concentrati su un aspetto molto specifico, ma fondamentale, forse il più importante di tutti: *quali dati deve maneggiare il mio programma e come li maneggerà*? In C, la domanda è divisa in due parti: *quali dati* ha a che vedere con le variabili presumibilmente globali del programma; *come* ha a che vedere con le funzioni del programma che manipoleranno queste variabili. In Objective-C (in qualunque linguaggio ad oggetti, in realtà), la domanda è una sola, perché è la classe che creiamo che si occupa tanto della memorizzazione dei dati quanto della loro manipolazione. localita.h e localita.m mostrano come è fatto *ogni* oggetto di classe `localita`; il programma principale, che per noi è il file main.m, avrà invece il compito di mettere da qualche parte l'insieme degli oggetti di classe `localita`, che potranno essere uno o centomila (se fossero nessuno non avrebbe molto senso scrivere il programma), affinché ognuno di essi possa essere creato, distrutto e modificato a piacere dell'utente secondo quanto previsto dall'implementazione della classe `localita`.

Analizziamo allora il file main.m: esso inizia con l'includere la libreria stdio.h perché ci servirà per usare le funzioni `printf()` e `scanf()`; quindi importa localita.h perché gli oggetti con cui avrà a che fare apparterranno a questa classe; infine, importa Cocoa.h perché userà classi del framework Cocoa che ci torneranno particolarmente utili. Definisce quindi alcune costanti, che ci serviranno per il menu principale che consentirà all'utente di creare nuove città, inserire i valori di temperatura e umidità per le città esistenti, e vedere i valori medi, oppure uscire dal programma. Quindi, dichiara le variabili globali: `gDone`, di tipo `BOOL`, è una variabile che può contenere solo un'informazione di tipo vero/falso (nella forma del numero intero 1 per indicare *vero* oppure *sì* e del numero intero 0 per indicare *falso* oppure *no*). `BOOL` è un tipo di variabile che non esiste in C e che è stato aggiunto in Objective-C per comodità dei programmatori.

La seconda variabile globale è l'oggetto `citta`, un puntatore a `NSMutableArray`, una classe del framework Cocoa che consente di gestire in maniera estremamente versatile e potente delle array di oggetti; occupandoci del C nella prima puntata di questa trilogia abbiamo parlato di array, nella forma di array di numeri o di caratteri. Volendo, possiamo creare array di strutture; in Objective-C in accoppiata con Cocoa, possiamo creare array di oggetti! Per noi, `citta` sarà l'oggetto nel quale memorizzeremo tutti gli oggetti di classe `localita` che l'utente vorrà creare per mantenere le statistiche di temperatura e umidità media.

Infine, l'oggetto `ap`, di classe `NSAutoreleasePool`, è una roba un po' strana. Se permettete, ce ne occupiamo tra poco.

A seguire ci sono i prototipi delle funzioni che useremo nel file main.m: abbiamo omesso il prototipo della funzione `main()`, poiché è standard. Abbiamo poi la funzione `MenuPrincipale()`, che mostra all'utente il menu con tutte le opzioni tra cui può scegliere, e una funzione per ognuna delle operazioni che l'utente può eseguire, ovvero creare una nuova città, inserire i valori di temperatura e umidità per le città esistenti, visualizzare i valori medi, o uscire dal programma.

La funzione `main()` inizia con un'istruzione strana: `ap=[[NSAutoreleasePool alloc] init];` che sarà mai? Qui dovete avere un po' di fede. Quando realizzate un programma in Objective-C che fa uso di oggetti appartenenti a classi Cocoa, ovvero quando importate il file di header Cocoa.h e compliate linkando al framework Cocoa, MacOS X si aspetta che il vostro programma disponga di un *autorelease pool*, un costrutto necessario a Cocoa per gestire in maniera efficiente la memoria che verrà allocata e liberata tutte le volte che creerete, in maniera implicita od esplicita, oggetti temporanei, che non devono "sopravvivere" a lungo, magari perché vi servono solo rapidamente all'interno di un metodo per trasferire un po' di informazioni o fare un po' di conti. Se questo autorelease pool non è disponibile, la memoria destinata dal sistema al vostro programma si riempie presto di schifezze, rendendo molto

probabile che nel giro di poco tempo il programma vada in crash; inoltre, il programma vi riempie di avvisi che dicono pressappoco che, non essendoci un autorelease pool disponibile, la memoria sta diventando un ricettacolo di gente di malaffare. Per evitare di incorrere in questi inconvenienti, inseriamo questa istruzione all'inizio della funzione `main()` (e dichiariamo la variabile `ap` come puntatore a `NSAutoreleasePool` tra le variabili globali), così MacOS X è contento e non ci stressa.

Quindi dobbiamo "fare posto" per le città che l'utente vorrà creare; esse verranno immagazzinate nell'oggetto `citta` di classe `NSMutableArray`, ovvero un'array *mutevole*, ovvero i cui contenuti possono essere modificati (Cocoa mette a disponizione anche una classe `NSArray`, genitrice di `NSMutableArray`, a cui appartengono array i cui contenuti non possono essere modificati una volta che l'array è stata creata). Iniziamo a chiamare il metodo di classe `alloc` con `[NSMutableArray alloc]` per riservare in memoria un "appezzamento" adatto ad accogliere un oggetto di classe `NSMutableArray`; quindi, con l'oggetto restituito, "inscatoliamo" il tutto con la chiamata ad uno dei vari metodi di inizializzazione della classe `NSMutableArray`: ho scelto il metodo `initWithCapacity:`, che inizia a riservare memoria sufficiente per archiviare tanti oggetti quanti sono indicati nell'argomento (uno, nel nostro esempio), riservandosi poi di recuperare automaticamente memoria sufficiente qualora il numero di oggetti da memorizzare fosse maggiore. L'oggetto allocato e opportunamente inizializzato può ora essere assegnato alla variabile `citta`, che ora è un'*array mutevole* a tutti gli effetti, pronta ad archiviare al suo interno le varie località che l'utente vorrà creare.

Subito dopo, un ciclo `do-while` si occupa di continuare a mostrare il menu principale all'utente finché questi non sceglie di uscire dal programma: la variabile globale `gDone`, che può far uscire da questo loop e subito dopo dalla funzione `main()` causando l'arresto del programma, viene impostata a `YES` solo se l'utente sceglie di uscire.

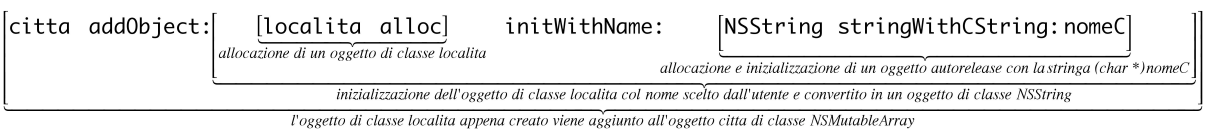
La funzione `MenuPrincipale()` è molto semplice: una serie di funzioni `printf()` scrive a schermo le varie opzioni a disposizione dell'utente, e una funzione `scanf()` accetta l'input numerico da tastiera; un blocco `switch`, sfruttando le costanti definite all'inizio, verifica che la scelta dell'utente sia tra quelle effettivamente disponibili e dirotta il flusso del programma alla funzione opportuna; in caso di scelta errara da parte dell'utente (un numero fuori dai limiti consentiti), un messaggio d'errore invita a riprovare, il flusso del programma torna al ciclo `do-while` della funzione `main()` ed essendo la variabile `gDone` ancora fissata a `NO` la funzione `MenuPrincipale()` viene chiamata nuovamente.

Quando l'utente sceglie di creare una nuova città (operazione che andrebbe fatta subito, visto che non ha molto senso inserire temperature e umidità per località che non esistono) viene chiamata la funzione `NuovaCitta()`. Visto che siamo in vena di cose nuove, ho deciso di introdurre una piccola aggiunta a quanto avevamo imparato con la nostra *Introduzione al linguaggio C*; all'epoca, trattavamo le stringhe di caratteri come array di tipo `char`; adesso, anche in analogia col modo con cui trattiamo tutti gli oggetti, definiamo una stringa di caratteri come un puntatore a `char`. Esattamente come nel caso degli oggetti, dobbiamo in qualche maniera allocare memoria per questa stringa; non trattandosi di un oggetto non possiamo chiamare il suo metodo `alloc`, ma possiamo usare una delle funzioni standard del C, `calloc()`, il cui secondo argomento indica di che tipo sono le variabili per cui voglio creare posto in memoria, e il primo argomento indica quante sono; in altre parole, con `calloc(40,sizeof(char))` stiamo riservando spazio in memoria per 40 caratteri, ovvero per una stringa, `nomeC`, lunga fino a 40 caratteri. L'equivalente del metodo di inizializzazione per questa stringa è l'assegnazione che viene fatta all'interno della funzione `scanf()`.

Fin qui è tutto abbastanza pacifico. Ora avviene la prima delle cose spettacolari: finalmente usiamo la classe `localita`! Sfruttando il fatto che essa eredita il metodo `alloc` da `NSObject`, chiamiamo `[localita alloc]` per allocare la memoria necessaria, poi l'oggetto (di tipo `id`) restituito da questa chiamata lo passiamo subito (inscatolandolo) al metodo di inizializzazione della nostra classe, ovvero `initWithName:`. Esso vuole come argomento un oggetto di classe `NSString`, e non un semplice puntatore a `char`, ed ecco perché l'istruzione precedente alloca memoria per un oggetto di classe `NSString` inizializzandolo con un puntatore

a `char` (`nomeC`) ed assegnando l'oggetto risultante a `nome`. Quindi, `[[localita alloc] initWithName:nome]` crea un oggetto di classe `localita` con il nome scelto dall'utente. Esso va immagazzinato da qualche parte; lo mettiamo nell'array `citta` che abbiamo creato appositamente, procedendo con l'inscatolamento: la classe `NSMutableArray` offre un metodo, `addObject:`, che aggiunge l'oggetto messo ad argomento (di classe `id`, quindi qualunque) nell'array: il nostro oggetto è `[[localita alloc] initWithName:nome]`, quindi lo mettiamo ad argomento di `[citta addObject:]`; ora `citta` contiene un oggetto nuovo, di classe `localita`, inizializzato col nome scelto dall'utente. Quindi liberiamo la memoria occupata dall'oggetto `nome`, visto che non ci serve più, e quella occupata da `nomeC`, visto che anche lei ormai è diventata inutile.

In realtà avremmo potuto fare ancora meglio: l'oggetto `nome` di classe `NSString` della funzione `NuovaCitta()` non serve a nulla: è soltanto un posto in cui memorizzare temporaneamente il nome della città, convertito da `char *` a `NSString`, prima di memorizzarlo definitivamente nell'oggetto di classe `localita`. Infatti l'oggetto `nome` non ci servirà al di fuori della funzione `NuovaCitta()`, essendo in effetti una variabile locale per essa. Possiamo quindi sbarazzarci dell'oggetto `nome`: `[[localita alloc] initWithName:]` vuole un oggetto `NSString` come argomento, proprio `[[NSString alloc] initWithCString:nomeC]`; dobbiamo però stare attenti; se scrivessimo `[[localita alloc] initWithName:[[NSString alloc] initWithCString:nomeC]]` ed assegnassimo il tutto ad un nuovo elemento dell'array `citta`: `[citta addObject: [[localita alloc] initWithName:[[NSString alloc] initWithCString:nomeC]]]` commetteremmo un errore sottile: la sintassi `[[NSString alloc] initWithCString:]` crea un oggetto (che non assegnamo a nessuna variabile) al quale non saremmo poi in grado di mandare un messaggio di `release`: un errore di gestione della memoria, un *memory leak*. Ma Cocoa ci mette a disposizione un'alternativa, un metodo per generare un oggetto "temporaneo" senza che dobbiamo preoccuparci di mandargli un release quando non ci occorre più: usiamo pertanto la sintassi `[NSString stringWithCString:]`. Come vedete facciamo uso di un metodo di classe (infatti è chiamato nello stesso blocco di parentesi quadre in cui compare il nome della classe) che fa la stessa cosa dell'uso combinato dei metodi `alloc` e `initWithCString:`, ma genera un cosiddetto *oggetto autorelease*. Ricordate quando all'inizio della funzione `main()` abbiamo creato un autorelease pool? Beh, serve per gestire queste cose qua. Un oggetto autorelease è un oggetto la cui memoria verrà automaticamente liberata dal sistema operativo al momento opportuno, sicuramente quando l'esecuzione del programma sarà uscita dal metodo o dalla funzione in cui è stato creato l'oggetto autorelease. Ecco quindi una regola d'oro: se un oggetto vi serve in più punti (funzioni o metodi) del programma, allocatelo con `alloc` e inizializzatelo con un metodo `init`, e quando non vi serve più rilasciatelo con `release`; se un oggetto ha un uso temporaneo, limitato ad una sola funzione o metodo, potete usare la sintassi che fa uso di `alloc`, `init` e `release`, oppure usare un metodo autorelease (un metodo di classe che non sia `alloc`), così potete inscatolare tutto in un'unica espressione. Potenza dell'Objective-C e della sua sintassi così diversa dal C++! Con una sola riga di istruzioni possiamo creare un oggetto (autorelease) `NSString` a partire da un puntatore a `char`, usarlo come inizializzatore di un oggetto di classe `localita` creato all'uopo e aggiungere tale oggetto, una volta inizializzato, ad un oggetto di classe `NSMutableArray`! Da principio, quando non siete ancora abituati, tutto questo "inscatolare" fa girare un po' la testa. Ma poi ci si fa l'abitudine, e ci si chiede com'è stato possibile vivere senza la sintassi dell'Objective-C per tutto il tempo che si è usato il C per programmare! Vediamo di riassumere tutto con uno schemino:



All'interno della funzione `InserisciValori()` succedono altre cose spettacolari. Compito di questa funzione è, per ogni oggetto (quindi per ogni località) memorizzato nell'array

citta, chiamare il suo metodo nuoviValori, implementato nel file localita.m. Ci sono più modi per scorrere i vari oggetti presenti in un'array, il più ovvio dei quali è realizzare un ciclo for la cui variabile di controllo fa da indice per le varie caselle dell'array. Questo si può fare anche con oggetti di classe NSArray e NSMutableArray (anche se in modo un po' diverso rispetto alle array standard del C), ma esse ci mettono a disposizione un modo molto più elegante, mediante l'utilizzo di un oggetto di classe NSEnumerator (un "enumeratore", ammesso che esista questa parola in italiano). Notate la bellezza: il metodo objectEnumerator implementato dalla classe NSArray (e quindi automaticamente dalla sua classe figlia o sottoclasse NSMutableArray) restituisce un oggetto di classe NSEnumerator, già allocato e inizializzato (autorelease), che assegnamo alla variabile en. La classe NSEnumerator, a sua volta, implementa un metodo, nextObject, che restituisce il prossimo oggetto della lista. Che vuol dire? Applicato ad un'array significa che l'oggetto en contiene un elenco di tutti gli oggetti memorizzati all'interno dell'array stessa, e tutte le volte che chiamo il metodo nextObject dell'oggetto di classe NSEnumerator mi viene restituito uno degli oggetti presenti nell'array; uno per volta, finché non li esaurisco; a questo punto, [en nextObject] restituisce nil, la condizione nel ciclo while diventa falsa e si esce dal ciclo. L'oggetto restituito da [en nextObject] va memorizzato da qualche parte, ad esempio nella variabile che ho chiamato obj; non devo preoccuparmi di chiamare il metodo release di obj, perché [en nextObject] restituisce un oggetto autorelease (mica ho usato i metodi alloc ed init per crearlo, no?). Ora, io so che tutti gli oggetti contenuti nell'array citta sono di classe localita, quindi potrei dichiarare obj come puntatore alla classe localita, ma preferisco sfruttare le proprietà di assegnazione dinamica di tipo dell'Objective-C (una cosa che in C e in C++ non posso fare!), lasciando obj dichiarato come di tipo generico id, ovvero un oggetto qualunque. Questo perché sono previdente: in una NSMutableArray come citta posso inserire oggetti appartenenti a classi diverse (ecco un indubbio vantaggio rispetto alle array del C, che invece immagazzinano solo variabili dello stesso tipo); sfogliando gli oggetti uno per uno mediante l'uso della classe NSEnumerator e assegnandoli ad obj, se lo costringessi ad essere un puntatore a localita otterrei un errore (in fase di esecuzione del programma) se uno degli oggetti non fosse di classe localita. Al momento non è possibile, ma in futuro? Se il Servizio Meteorologico Nazionale mi chiedesse di estendere il mio programma e io avessi bisogno di memorizzare in citta anche oggetti di classe diversa da localita? Perché cacciarsi nei guai e dover modificare la funzione InserisciValori()? Lasciamo obj di tipo generico id, e limitiamoci a chiamare il suo metodo nuoviValori. Se obj è di classe localita, il metodo nuoviValori chiederà una temperatura e un'umidità e calcolerà i valori medi. Se obj fosse di un'altra classe, sarà sufficiente che questa implementi un metodo chiamato nuoviValori e il programma continuerà a funzionare! Ogni classe è responsabile di gestire i propri dati, ogni oggetto è responsabile di sé stesso; main.m non ha bisogno di saperlo. Vedremo nel prossimo capitolo una prima applicazione di questa importante novità.

Quando l'utente sceglie invece di visualizzare le temperature e le umidità medie per le varie località inserite, la funzione MostraRisultati() viene chiamata. Ormai siete degli esperti: il meccanismo è lo stesso della InserisciValori(). Un oggetto di classe NSEnumerator, accoppiato ad un ciclo while, permette di chiamare il metodo mostraValori di ogni località memorizzata nell'oggetto citta. main.m non ha bisogno di sapere quali valori siano memorizzati in ogni oggetto né come vadano trattati; è l'oggetto a saperlo, è lui che pensa ad accettare i nuovi valori in ingresso e a mostrare i risultati finali in uscita.

Infine, la funzione Esci() imposta a YES la variabile gDone, causando l'uscita dal loop do-while della funzione main() all'iterazione successiva. Gli oggetti citta e ap, che erano stati creati chiamando il metodo alloc e un metodo di tipo init, vengono ora liberati col metodo release (ap va rilasciato per ultimo). Il programma è pronto per uscire.

4. Sottoclassi

«Congratulazioni! Il Suo programma è piaciuto tantissimo ai nostri Responsabili di Progetto. Attualmente lo stiamo utilizzando a pieno regime per monitorare temperatura e umidità medie di oltre 20 località sparse per il Paese. Gradiremmo che ora Lei lo espandesse così da includere una nuova categoria, "città con aeroporto", che oltre alle grandezze già monitorate per le città normali calcoli anche la visibilità media presso l'aeroporto. Certi della Sua proficua collaborazione, porgiamo cordiali saluti». Questa è la lettera che vi è appena arrivata dal Servizio Meteorologico Nazionale. Siete molto contenti, ma anche molto nervosi: vi siete cacciati in un bel guaio! Quest'aggiunta delle "città con aeroporto", che sono come le città normali ma hanno un pezzo in più, vi costringerà a riscrivere buona parte del codice, e il tempo disponibile è poco. Accidenti! Ma… un momento… avete usato Objective-C! Ah, ma allora è diverso! È un giochetto da ragazzi: basta creare una sottoclasse di `localita` e modificare leggermente main.m.

Rinfrancati, aprite il vostro editor di testo preferito e digitate quanto segue:

```
#import <Cocoa/Cocoa.h>
#import "localita.h"

@interface aeroporto : localita
{
    double      visibilitaMedia;
}

- (void)nuovaVisibilita:(double)unaVisibilita;

@end
```

Salvatelo come aeroporto.h (magari in una nuova cartella denominata Esempio2). Quindi, dopo aver copiato in questa cartella anche i file localita.h e localita.m dell'Esempio1, tornate al vostro editor di testi preferito e digitate quanto segue:

```
#import "aeroporto.h"

@implementation aeroporto

- (id)initWithName:(NSString *)unNome
{
    [super initWithName:unNome];
    visibilitaMedia=0.0;
    return self;
}

- (void)nuoviValori
{
    double      v;

    [super nuoviValori];
    printf("Inserisci la visibilita' di oggi a mezzigiorno (m): ");
    scanf("%lg",&v);
    [self nuovaVisibilita:v];
}
```

```
- (void)mostraValori
{
    [super mostraValori];
      printf("Presso l'aeroporto la visibilita' media e' stata di %g
m\n",visibilitaMedia);
}

- (void)nuovaVisibilita:(double)unaVisibilita
{
               visibilitaMedia=(visibilitaMedia*(numeroGiorni-
1)+unaVisibilita)/numeroGiorni;
}

@end
```

Salvate il file come aeroporto.m, quindi copiate nella cartella Esempio2 il file main.m dell'Esempio1 e, sempre usando il vostro editor di testo preferito, modificatelo come segue (le modifiche rispetto alla versione usata per l'Esempio1 sono riportate in grassetto):

```
#include <stdio.h>
#import "localita.h"
#import  "aeroporto.h"
#import <Cocoa/Cocoa.h>

#define kNuovaCitta         1
#define  kNuovoAeroporto        2
#define  kInserisciValori       3
#define  kMostraRisultati       4
#define kEsci               0

BOOL        gDone=NO;
NSMutableArray      *citta;
NSAutoreleasePool   *ap;

void MenuPrincipale(void);
void NuovaCitta(void);
void  NuovoAeroporto(void);
void InserisciValori(void);
void MostraRisultati(void);
void Esci(void);

int main(void)
{
    ap=[[NSAutoreleasePool alloc] init];
    citta=[[NSMutableArray alloc] initWithCapacity:1];
    do
    {
        MenuPrincipale();
    } while(gDone==NO);
    return 0;
}
```

```
void MenuPrincipale(void)
{
    int     scelta;

    printf("Menu principale:\n\n");
    printf("1. Nuova citta'\n");
      printf("2. Nuovo aeroporto\n");
      printf("3. Inserisci nuovi valori\n");
      printf("4. Mostra i risultati\n");
    printf("\n");
    printf("0. Esci\n");
    printf("\n");
    printf("Inserisci la tua scelta: ");
    scanf("%d",&scelta);

    switch(scelta)
    {
        case kNuovaCitta:
            NuovaCitta();
            break;
          case kNuovoAeroporto:
              NuovoAeroporto();
             break;
        case kInserisciValori:
            InserisciValori();
            break;
        case kMostraRisultati:
            MostraRisultati();
            break;
        case kEsci:
            Esci();
            break;
        default:
                printf("Inserisci un numero compreso tra 0 e 4!\n\n");
            break;
    }
}

void NuovaCitta(void)
{
    NSString    *nome;
    char        *nomeC;

    nomeC=calloc(40,sizeof(char));
    printf("\n");
    printf("Inserisci un nome per la nuova citta': ");
    scanf("%s",nomeC);
    nome=[[NSString alloc] initWithCString:nomeC];
    [citta addObject:[[localita alloc] initWithName:nome]];
    [nome release];
    free(nomeC);
}
```

```
void  NuovoAeroporto(void)
{
     NSString     *nome;
     char         *nomeC;

       nomeC=calloc(40,sizeof(char));
      printf("\n");
      printf("Inserisci un nome per il nuovo aeroporto: ");
      scanf("%s",nomeC);
       nome=[[NSString alloc] initWithCString:nomeC];
       [citta addObject:[[aeroporto alloc] initWithName:nome]];
      [nome release];
      free(nomeC);
}

void InserisciValori(void)
{
    NSEnumerator    *en;
    id              obj;

    en=[citta objectEnumerator];
    while(obj=[en nextObject])
        [obj nuoviValori];
}

void MostraRisultati(void)
{
    NSEnumerator    *en;
    id              obj;

    en=[citta objectEnumerator];
    while(obj=[en nextObject])
        [obj mostraValori];
}

void Esci(void)
{
    gDone=YES;
    [citta release];
    [ap release];
}
```

Salvate il file come main.m, quindi compilate localita.m, aeroporto.m e main.m e infine linkate i tre file oggetto (tutti e tre!) nel programma Esempio2:

```
gcc -c localita.m
gcc -c aeroporto.m
gcc -c main.m
gcc -o Esempio2 localita.o aeroporto.o main.o -lobjc -framework Cocoa
```

Questa è l'ultima volta che scriviamo come si fa, chiaro? Ora eseguite l'Esempio2 e giocateci un po'. Quando avete sperimentato quello che succede creando un po' di città e un po' di aeroporti (che sta per "città con aeroporto"), tornate qui e ne discutiamo.

Cominciamo dal file aeroporto.h. Come suggerisce la dichiarazione dell'interfaccia della classe, `aeroporto` è una *sottoclasse* di `localita`; ovvero, `localita` è la *classe genitrice* o la *superclasse* di `aeroporto`. Perché questo giochetto funzioni, è necessario importare il file di interfaccia della classe genitrice, mediante l'istruzione `#import "localita.h"`. Nel file localita.h l'aver importato Cocoa.h aveva automaticamente reso disponibile il file di interfaccia della classe `NSObject`.

Dire che `aeroporto` è una sottoclasse di `localita` vuol dire che ogni `aeroporto` è una `localita`, ma non viceversa. Ovvero: un `aeroporto` è una `localita` con un nome e un sistema per monitorare temperatura e umidità medie; in aggiunta, un `aeroporto` può anche monitorare la visibilità media. Ecco un primo grande vantaggio di una sottoclasse: una "città con aeroporto" è, in fin dei conti, una città; quindi, perché duplicare il codice già esistente per la gestione delle città? Allora creiamo una sottoclasse di `localita` e aggiungiamo quello che serve per gestire le caratteristiche aggiuntive e peculiari di un aeroporto. Avremmo potuto modificare direttamente la classe `localita`, ma non sarebbe stato molto elegante: avremmo dovuto modificare il metodo di inizializzazione, prevedendo un modo per specificare se si trattasse di una città o di una città con aeroporto, quindi in una variabile della classe avremmo dovuto memorizzare questa informazione, e nei metodi `nuoviValori` e `mostraValori` un blocco `if` avrebbe selezionato se richiedere o visualizzare oppure no le informazioni riguardanti la visibilità a seconda che l'oggetto fosse identificato come città con aeroporto o città. Scomodo e molto poco elegante. Molto meglio una sottoclasse; come vedete, non è stato necessario modificare né localita.h né localita.m.

L'implementazione della classe `aeroporto` avviene nel file aeroporto.m; qui *sovrascriviamo* il metodo di inizializzazione perché vogliamo che la variabile `visibilitaMedia` sia inizializzata a zero; tuttavia, poiché un `aeroporto` è una `localita`, è inutile riscrivere il codice già scritto: chiamiamo `[super initWithName:]`, dove `super` indica proprio la classe `localita`. Naturalmente, anziché chiamare il metodo `[super init]` come facevamo in localita.m, qui chiamiamo `[super initWithName:]`, perché è `initWithName:` il metodo di inizializzazione della classe `localita`. Come sempre, il metodo di inizializzazione deve restituire `self`.

Non è necessario sovrascrivere i metodi che non necessitano di essere modificati: il metodo `dealloc`, ad esempio, è immutato, perché `aeroporto` non necessita di nessun nuovo oggetto. Quando alla classe `aeroporto` verrà richiesto dal sistema operativo di eseguire il suo metodo `dealloc`, essa, non avendolo, dirotterà automaticamente la richiesta alla classe genitrice, ovvero a `localita`, che eseguirà felicemente il suo metodo `dealloc`.

È invece necessario sovrascrivere il metodo `nuoviValori`, perché un `aeroporto` è qualcosa di diverso da una città normale. Siccome le proprietà di temperatura media e umidità media si gestiscono alla stessa maniera, chiamiamo `[super nuoviValori]`, che richiede all'utente le solite informazioni su temperatura e umidità a mezzogiorno, aggiornando poi i valori medi. Poi, chiediamo all'utente di inserire anche la visibilità presso l'aeroporto, calcolando la visibilità media chiamando il metodo `nuovaVisibilita:` della classe `aeroporto`. Questa è una regola quasi generale: quando create una sottoclasse e ne sovrascrivete un metodo, facilmente questo chiamerà il metodo omonimo della classe genitrice, e poi lo espanderà con nuove linee di codice specifiche per le operazioni in più che la sottoclasse deve compiere.

Un discorso analogo vale per il metodo `mostraValori`: esso chiama `[super mostraValori]` per la parte in comune con le città senza aeroporto; quindi visualizza il valore della variabile `visibilitaMedia`.

Una cosa molto più interessante succede invece nel metodo `nuovaVisibilita:`: qui la visiblità media viene calcolata come si fa con l'umidità nel metodo `nuovaUmidita:` nel file

localita.m. Niente di strano, direte voi. E unvece una cosa interessante c'è: il metodo `nuovaVisibilita:` usa la variabile `numeroGiorni`, dichiarata per la classe `localita` ma non per la classe `aeroporto`. Probabilmente non sarete sorpresi nemmeno adesso, e fate bene: `aeroporto` è una sottoclasse di `localita`, tutto ciò che ha una `localita` lo ha anche un `aeroporto`; se la variabile `numeroGiorni` è definita per ogni oggetto di classe `localita`, lo è anche per ogni oggetto di classe `aeroporto`. Questo è vero, ed è il comportamento standard. Tuttavia non è l'unico comportamento possibile.

Se andiamo ad esaminare il file di interfaccia della classe `localita`, localita.h, vediamo che le variabili sono dichiarate semplicemente all'interno delle parentesi graffe che seguono la direttiva`@interface`. Tuttavia, come accennavamo nel capitolo precedente, è possibile assegnare alle variabili di una classe delle direttive diverse. In particolare, esse possono essere dichiarate come pubbliche, private o protette.

Quando dichiarate le variabili nel file di interfaccia di una classe potete scrivere, ad esempio:

```
@interface localita : NSObject
{
    @public
    NSString  *nome;

    @private
    double    temperaturaMedia;
    double    umiditaMedia;

    @protected
    int  numeroGiorni;
}
```

È chiaro che la variabile `nome` è dichiarata come *pubblica*, `temperaturaMedia` e `umiditaMedia` sono *private* e `numeroGiorni` è *protetta*. Ma che differenza c'è?

Partiamo da `numeroGiorni`: essendo una variabile protetta (è la direttiva standard, quella che viene assegnata ad una variabile di una classe se non specificate altrimenti), tutti gli oggetti della classe `localita` potranno leggere e scrivere direttamente il contenuto della variabile `numeroGiorni`; inoltre, anche tutti gli oggetti delle sottoclassi di `localita` (come `aeroporto`) potranno leggere e scrivere direttamente la variabile. È quello che succede nel metodo `nuovaVisibilita:`: in assenza di ulteriori precisazioni, `numeroGiorni` è stata dichiarata come protetta, la sottoclasse `aeroporto` può leggerla tranquillamente per aggiornare il valore di `visibilitaMedia`.

Quando una variabile è dichiarata come privata, invece, essa è accessibile solamente agli oggetti della classe in questione; nessun altro può leggere o scrivere quella variabile, nemmeno gli oggetti appartenenti alle sottoclassi. Nell'ipotetica interfaccia per la classe `localita` scritta sopra, le variabili `temperaturaMedia` e `umiditaMedia` sono dichiarate come private; questo vuol dire che gli oggetti di classe `localita` possono leggere e scrivere queste due variabili, ma gli oggetti di classe `aeroporto`, pur avendo `localita` come classe genitrice, non potrebbero né leggere né scrivere le variabili `temperaturaMedia` e `umiditaMedia`. Dichiarare privata una variabile è un modo per proteggerla da modifiche accidentali. Inoltre, è un modo per svincolare il codice dall'implementazione di una classe. La classe `localita` dispone di metodi opportuni per inserire nuovi valori di umidità e di temperatura e per visualizzare le medie. Se, per ipotesi, un oggetto di classe `aeroporto` accedesse direttamente alle variabili `temperaturaMedia` e `umiditaMedia`, e in un secondo momento voi decideste di modificare il modo con cui la classe `localita` gestisce queste informazioni, eliminando queste variabili o modificandone il nome, la classe `aeroporto` non sarebbe più valida; se invece `temperaturaMedia` e `umiditaMedia` sono private e la classe `aeroporto` accede ad esse solo tramite i metodi opportunamente previsti, è

sufficiente che voi continuiate a tenerli nella classe localita, e la classe aeroporto continuerà a funzionare, anche se decidete di eliminare le variabili temperaturaMedia e umiditaMedia. Noi non ci siamo preoccupati di questo perché il nostro programma è piccolo e facile da mantenere, ma in generale è saggio dichiarare come private tutte le variabili di una classe che si presuppone non siano di diretto interesse delle sottoclassi.

È invece meglio evitare di dichiarare una variabile come pubblica. Dichiararla tale, infatti, la rende visibile a tutto il programma, dove con *tutto* intendo proprio *tutto*. È molto rischioso: per sbaglio potreste modificare il valore di una variabile pubblica in un punto del codice in cui pensate di accedere ad una certa variabile locale e invece finite per alterare il contenuto di una variabile di un oggetto che non c'entra niente. E poi, se il vostro codice si affida alla presenza di questa variabile dichiarata come pubblica e un giorno decidete di eliminarla, dovrete ricontrollare tutto. Insomma, se potete evitare di usare le variabili pubbliche è meglio.

Quindi, se volete migliorare l'interfaccia della classe localita, modificatela come segue:

```
@interface localita : NSObject
{
    @private
    NSString  *nome;
    double    temperaturaMedia;
    double    umiditaMedia;

    @protected
    int  numeroGiorni;
}
```

Le cose interessanti, comunque, non sono di certo finite. Passiamo al file main.m. Le prime modifiche, ovvie, riguardano le costanti (abbiamo aggiunto kNuovoAeroporto) e la funzioneMenuPrincipale(), dove abbiamo aggiunto il comando per inserire un nuovo aeroporto, cosa che verrà fatta nella funzione NuovoAeroporto().

La funzione NuovoAeroporto() è in tutto e per tutto uguale a NuovaCitta(), con la sola differenza che all'oggetto citta, di classe NSMutableArray, si aggiunge un oggetto di classe aeroporto anziché localita. Ecco qui una prima bella cosa: gli oggetti di classe NSMutableArray, come citta, possono contenere oggetti appartenenti a classi diverse. Nel nostro programma, infatti, memorizziamo all'interno di citta sia oggetti di classe localita, sia oggetti di classe aeroporto. Il fatto che siano l'una sottoclasse dell'altra non vi tragga in inganno: in un oggetto di classe NSMutableArray potete archiviare oggetti appartenenti alle classi più disparate, anche se queste non sono imparentate tra di loro (a ben vedere, praticamente sempre le classi sono imparentate tra di loro, perché praticamente sempre derivano tutte da NSObject).

Il vero miracolo, tuttavia, avviene nelle funzioni InserisciValori() e MostraValori(). Ora capirete il perché di tutte quelle belle parole sull'assegnazione dinamica di tipo spese nel capitolo 2. Infatti, pur avendo dichiarato una nuova classe, pur avendo aggiunto un nuovo "tipo" di oggetti (gli aeroporti), non abbiamo bisogno di modificare il codice che riguarda l'inserimento e la visualizzazione dei dati. Il file main.m, infatti, non è tenuto a sapere i dettagli di localita o di aeroporto, non gliene deve fregare niente; main.m sa che quando l'utente vuole inserire dei nuovi dati bisogna chiamare il metodo nuoviValori di ognuno degli oggetti contenuti in citta; quando l'utente vuole visualizzare i risultati, bisogna chiamare il metodo mostraValori di ognuno degli oggetti contenuti in citta. Non importa che cosa siano questi oggetti, non importa a che classe appartengano, non importa come i dati siano memorizzati e manipolati dagli oggetti in questione; non importa se le loro classi sono state "pensate" fin dall'inizio o sono state aggiunte in seguito. È irrilevante. citta può immagazzinare oggetti di qualunque classe. Le variabili obj di InserisciValori() e MostraRisultati() sono di tipo id, ovvero puntatori ad oggetti generici, quindi possono essere qualunque cosa, localita, aeroporto, o altre classi ancora che non abbiamo ancora

scritto. Se ognuno degli oggetti immagazzinati in `citta` risponde ai metodi `nuoviValori` e `mostraValori`, main.m continuerà a funzionare senza bisogno di modifiche, anche se alcuni oggetti fossero di un'altra classe ancora in cui memorizziamo il numero di scarpe di persone selezionate in base a chissà quale criterio. Notevole, vero? E se uno o più degli oggetti immagazzinati in `citta` non dovesse rispondere a questi metodi? Il programma si bloccherebbe, come se fosse scritto in C o in C++? No: MacOS X scriverebbe nel file console.log un avvertimento, segnalando il problema, ma il programma continuerebbe a funzionare lo stesso (naturalmente nessuna operazione verrebbe eseguita per quegli oggetti di cui mancano i metodi `nuoviValori` e `mostraValori`).

Se non volete correre il rischio che, nella foga di creare nuove classi per estendere il comportamento del vostro programma, qualche oggetto non implementi i metodi `nuoviValori` e `mostraValori`, allora non perdetevi il prossimo capitolo, dove parleremo di protocolli.

5. Categorie e Protocolli

Categorie

Nel capitolo precedente abbiamo visto che creare una sottoclasse è un buon modo per *specializzare* una classe, ovvero per aggiungere delle funzionalità che la rendono più specifica; infatti, abbiamo creato `aeroporto` come sottoclasse di `localita`, intendendo dire che ogni `aeroporto` è anche una `localita` (ne ha le stesse proprietà), ma non è vero il viceversa: non tutte le `localita` sono degli `aeroporti`.

Questa tecnica di creare delle sottoclassi qualora si voglia specializzare una classe esistente è valida e raccomandabile, ma non è sempre furbo applicarla. In alcune circostanze, ad esempio, si preferirebbe semplicemente poter *estendere* una classe, aggiungendovi funzionalità, senza per questo specializzarla ulteriormente, senza bisogno, insomma, di creare una sottoclasse. Questo è tanto più vero quanto più si pensa non alle classi che creiamo noi nei nostri programmi, quanto a quelle che il sistema operativo ci mette a disposizione tramite ad esempio il framework Cocoa. Potremmo infatti aver bisogno di espandere una classe fornitaci da Cocoa per implementare un qualche cosa che ci serve ma che non era inizialmente previsto. Anziché reinventare tutto da zero con una nuova classe o dover creare una sottoclasse di quella che vorremo estendere, incorrendo in possibili problemi o difficoltà, Objective-C ci dà un potente ed elegante strumento: le *zeppole farcite!* Come dite? Questo paragrafo si intitola *categorie*? Mi sa che avete ragione: Objective-C ci dà le categorie, per estendere le funzionalità di una classe esistente, fosse anche una di quelle messe a disposizione da Cocoa.

L'occasione di imparare qualche cosa sulle categorie ci viene data dal solito Servizio Meteorologico Nazionale, ormai il nostro miglior cliente. Le alte sfere hanno deciso che, in aggiunta a tutto ciò che già fa il nostro programma, occorre ora definire un nuovo concetto, una *regione*; giornalmente, regione per regione, l'utente deve poter inserire il numero di millimetri di pioggia caduti. A differenza dell'umidità e della temperatura (e della visibilità, per gli aeroporti), qui non interessa un dato medio, ma interessa il dato totale mensile: sapere nel mese corrente quanti millimetri di pioggia sono caduti. Decidiamo che non è il caso di creare una sottoclasse di `localita`, perché sarebbe controproducente: essa erediterebbe sì metodi utili come `initWithName:`, ma anche i metodi inutili che calcolano i valori medi dei dati inseriti giornalmente; inoltre, non vi sarebbe traccia del meccanismo di controllo delle mensilità, dal momento che il dato totale riguardante le precipitazioni di pioggia va azzerato ad ogni inizio di un nuovo mese. Non risolviamo il problema nemmeno usando le categorie, che invece ci serviranno ad altro. Semplicemente, creiamo una nuova classe, `regione`, dotata delle caratteristiche che ci interessano. Apriamo pertanto il nostro editor di testo preferito e inseriamo il seguente codice:

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

@interface regione : NSObject
{
    NSCalendarDate      *dataCreazione;
    int                 mmDiPioggia;
    NSString            *nome;
}

- (id)initWithName:(NSString *)unNome;
- (NSString *)nome;
- (void)nuoviValori;
- (void)mostraValori;
```

```
@end

@interface NSCalendarDate (regione)
- (BOOL)isThisMonthsDate;
@end
```

Salviamolo come regione.h. Quindi digitiamo quest'altro codice:

```
#import "regione.h"

@implementation regione

- (id)initWithName:(NSString *)unNome
{
    [super init];
    nome=[[NSString alloc] initWithString:unNome];
    dataCreazione=[NSCalendarDate calendarDate];
    [dataCreazione retain];
    mmDiPioggia=0;
    return self;
}

- (void)dealloc
{
    [nome release];
    [dataCreazione release];
    [super dealloc];
}

- (NSString *)nome
{
    return nome;
}

- (void)nuoviValori
{
    int     pioggia;

    printf("\n");
    printf("Regione: %s\n",[nome cString]);
    printf("Inserisci i mm di pioggia caduti oggi: ");
    scanf("%d",&pioggia);
    if([dataCreazione isThisMonthsDate])
        mmDiPioggia+=pioggia;
    else
    {
        [dataCreazione release];
        dataCreazione=[NSCalendarDate calendarDate];
        [dataCreazione retain];
        mmDiPioggia=pioggia;
    }
}
```

```
- (void)mostraValori
{
    printf("\n");
    printf("Regione: %s\n",[nome cString]);
          printf("Sono caduti %d mm di pioggia nel mese
corrente\n\n",mmDiPioggia);
}

@end

@implementation NSCalendarDate (regione)
- (BOOL)isThisMonthsDate
{
        return ([self monthOfYear]==[[NSCalendarDate calendarDate]
monthOfYear]);
}
@end
```

Salviamolo come regione.m. Nella stessa cartella in cui abbiamo salvato i file precedenti (e che potremmo chiamare Esempio3), copiate anche i file localita.h, localita.m, aeroporto.h, aeroporto.m e main.m. Quest'ultimo modificatelo come segue (le parti modificate sono in grassetto):

```
#include <stdio.h>
#import "localita.h"
#import "aeroporto.h"
#import  "regione.h"
#import <Cocoa/Cocoa.h>

#define kNuovaCitta         1
#define kNuovoAeroporto     2
#define kNuovaRegione       3
#define kInserisciValori    4
#define kMostraRisultati    5
#define kEsci               0

BOOL        gDone=NO;
NSMutableArray      *citta;
NSAutoreleasePool   *ap;

void MenuPrincipale(void);
void NuovaCitta(void);
void NuovoAeroporto(void);
void  NuovaRegione(void);
void InserisciValori(void);
void MostraRisultati(void);
void Esci(void);

int main(void)
{
    ap=[[NSAutoreleasePool alloc] init];
    citta=[[NSMutableArray alloc] initWithCapacity:1];
```

```
    do
    {
        MenuPrincipale();
    } while(gDone==NO);
    return 0;
}

void MenuPrincipale(void)
{
    int     scelta;

    printf("Menu principale:\n\n");
    printf("1. Nuova citta'\n");
    printf("2. Nuovo aeroporto\n");
      printf("3. Nuova regione\n");
      printf("4. Inserisci nuovi valori\n");
      printf("5. Mostra i risultati\n");
    printf("\n");
    printf("0. Esci\n");
    printf("\n");
    printf("Inserisci la tua scelta: ");
    scanf("%d",&scelta);

    switch(scelta)
    {
        case kNuovaCitta:
            NuovaCitta();
            break;
        case kNuovoAeroporto:
            NuovoAeroporto();
            break;
          case kNuovaRegione:
              NuovaRegione();
             break;
        case kInserisciValori:
            InserisciValori();
            break;
        case kMostraRisultati:
            MostraRisultati();
            break;
        case kEsci:
            Esci();
            break;
        default:
                printf("Inserisci un numero compreso tra 0 e 5!\n\n");
            break;
    }
}

void NuovaCitta(void)
{
    NSString    *nome;
```

```
    char        *nomeC;

    nomeC=calloc(40,sizeof(char));
    printf("\n");
    printf("Inserisci un nome per la nuova citta': ");
    scanf("%s",nomeC);
    nome=[[NSString alloc] initWithCString:nomeC];
    [citta addObject:[[localita alloc] initWithName:nome]];
    [nome release];
    free(nomeC);
}

void NuovoAeroporto(void)
{
    NSString    *nome;
    char        *nomeC;

    nomeC=calloc(40,sizeof(char));
    printf("\n");
    printf("Inserisci un nome per il nuovo aeroporto: ");
    scanf("%s",nomeC);
    nome=[[NSString alloc] initWithCString:nomeC];
    [citta addObject:[[aeroporto alloc] initWithName:nome]];
    [nome release];
    free(nomeC);
}

void  NuovaRegione(void)
{
     NSString     *nome;
     char         *nomeC;

       nomeC=calloc(40,sizeof(char));
      printf("\n");
      printf("Inserisci un nome per la nuova regione: ");
      scanf("%s",nomeC);
       nome=[[NSString alloc] initWithCString:nomeC];
       [citta addObject:[[regione alloc] initWithName:nome]];
      [nome release];
      free(nomeC);
}

void InserisciValori(void)
{
    NSEnumerator    *en;
    id              obj;

    en=[citta objectEnumerator];
    while(obj=[en nextObject])
        [obj nuoviValori];
}
```

```
void MostraRisultati(void)
{
    NSEnumerator    *en;
    id              obj;

    en=[citta objectEnumerator];
    while(obj=[en nextObject])
        [obj mostraValori];
}

void Esci(void)
{
    gDone=YES;
    [citta release];
    [ap release];
}
```

Compilate il tutto, linkate il programma chiamandolo Esempio3, quindi eseguitelo e divertitevi a creare nuove località, aeroporti e regioni. Osservate il diverso comportamento di queste ultime rispetto a quanto visto finora. Riservatevi di giocare col programma al cambio di mese, inserendo alcuni dati l'ultimo giorno di un mese e poi, senza uscire dal programma, inserendone degli altri il primo giorno del mese successivo, e ammirate il risultato: il conteggio del numero totale di millimetri di piogga caduti verrà azzerato.

Ora mettetevi comodi e iniziamo a vedere come funziona il nostro giochetto. Partiamo dal file regione.h. Non contiene grosse novità: un oggetto di classe `NSString` che contiene il nome della regione, una variabile intera che contiene i millimetri di pioggia caduti nell'ultimo mese, un oggetto di classe `NSCalendarDate` che ha il compito di memorizzare la data di creazione dell'oggetto, così da permettere un controllo del cambio di mese ed eventualmente azzerare la variabile `mmDiPioggia`. I metodi dichiarati sono i soliti. C'è però, al fondo, dopo la direttiva `@end`, una nuova dichiarazione di interfaccia, applicata alla classe `NSCalendarDate`. Tutto ciò sarebbe illegale, dal momento che la classe `NSCalendarDate` esiste già ed è dichiarata in Cocoa.h; se non fosse che noi, tra parentesi tonde, specifichiamo che stiamo non già ridefinendo la classe `NSCalendarDate`, ma piuttosto estendendola, mediante una *categoria*, che per comodità chiamiamo `regione`, dal momento che questa estensione ci viene comoda all'interno della classe `regione`. Tale estensione a `NSCalendarDate` sarà visibile (e utilizzabile) solo ed esclusivamente all'interno delle porzioni di codice in cui è dichiarata (quindi in regione.h e, di conseguenza, in regione.m): state tranquilli: non influenza il funzionamento di altri programmi o del sistema operativo!

Per capire come funzionano queste categorie, andiamo a vedere il contenuto del file regione.m. Il metodo `initWithName:` non contiene particolari novità, se non che l'oggetto `dataCreazione` viene creato col metodo `calendarDate`. Qui occorre un approfondimento. Non stiamo usando la solita sintassi che richiede di chiamare il metodo `alloc` della classe e poi un metodo `init`; scegliamo invece di usare un metodo che restituisce un oggetto autorelease; siccome però questo oggetto ci servirà in futuro, non vogliamo che il sistema operativo liberi la memoria occupata da `dataCreazione` appena usciti dal metodo `initWithName:`. È per questo motivo che chiamiamo il metodo `retain` (ereditato anch'esso da `NSObject`) di `dataCreazione`. Questo impedirà che la memoria occupata dall'oggetto venga rilasciata anzitempo. Sarà compito nostro dire al sistema operativo che non ci serve più l'oggetto `dataCreazione` chiamandone il metodo `release` al momento opportuno.

Il metodo `calendarDate` assicura che l'oggetto `dataCreazione` contenga, in una qualche forma, tutte le grandezze necessarie per localizzare nel tempo in maniera univoca e perfettamente definita l'istante in cui il metodo stesso è stato chiamato. Poiché il metodo è stato chiamato alla creazione dell'oggetto `dataCreazione`, questo conterrà data e ora esatte della creazione di sé stesso e, di conseguenza, dell'oggetto di classe `regione` a cui appartiene.

Niente nuove buone nuove per i metodi `dealloc` e `nome`. Naturalmente chiamiamo `[dataCreazione release]` per compensare la chiamata a `retain` che avevamo fatto sullo stesso oggetto, per assicurarne a lungo la vita essendo esso un oggetto autorelease, per il modo in cui era stato creato. Qualche cosa di interessante succede invece in `nuoviValori`. Innanzitutto si chiede all'utente di inserire i millimetri di pioggia caduti quel giorno. Quindi, l'oggetto `dataCreazione` riceve l'ordine di eseguire il proprio metodo `isThisMonthsDate` (che vuol dire: è una data di questo mese?). Se andate a vedere la documentazione di Apple relativa alla classe `NSCalendarDate`, scoprirete che non esiste alcun metodo `isThisMonthsDate`. E in effetti è quello che abbiamo deciso di aggiungere noi alla classe `NSCalendarDate` mediante una categoria. In regione.h l'interfaccia per questa categoria è definita. In regione.m, al fondo, troveremo l'implementazione del metodo `isThisMonthsDate` applicato alla classe `NSCalendarDate`. Lo scopo di questo metodo, l'avrete capito, è quello di restituire un valore logico vero se la data contenuta in `dataCreazione` appartiene allo stesso mese corrente, un valore logico falso in caso contrario. Vedremo tra poco come questo avvenga. Qui succede che se il giorno in cui inserite i nuovi valori di millimetri di pioggia appartiene allo stesso mese in cui avete creato l'oggetto `dataCreazione`, la variabile `mmDiPioggia` viene semplicemente aggiornata aggiungendo il nuovo valore appena inserito dall'utente. In caso contrario, è necessario far ripartire il conteggio. L'oggetto `dataCreazione` viene rilasciato in memoria (questo ne provoca la distruzione) quindi viene ricreato con la data odierna (è iniziato un nuovo mese) e la variabile `mmDiPioggia` viene impostata al valore appena inserito dall'utente, il primo per il nuovo mese. Naturalmente, l'effetto di questo giochino sarà visibile se voi lanciate il programma ad esempio il 30 aprile, inserite un po' di dati, poi, *senza uscire dal programma*, aspettate il giorno successivo, 1° maggio, e inserite altri dati ancora; mentre per gli oggetti di classe `localita` e `aeroporto` non succederà nulla, gli oggetti di classe `regione` vedranno ricreato il loro oggetto `dataCreazione` e la variabile `mmDiPioggia` sarà azzerata e portata al valore inserito per primo il 1° maggio. `mostraValori`, dal canto suo, non fa nulla di nuovo.

Al fondo del file regione.m trovate poi l'implementazione della categoria `regione` applicata alla classe `NSCalendarDate`. Benché non sia possibile aggiungere nessuna variabile ad una classe mediante una categoria, potete aggiungere dei metodi, che qui vengono implementati. Nel nostro caso si tratta del metodo `isThisMonthsDate`. Vediamo nel dettaglio che cosa fa: l'espressione `[self monthOfYear]` contiene il primo trucco: `self` *non si riferisce ad un oggetto di classe* `regione`, perché la categoria si applica alla classe `NSCalendarDate`; `self` si riferisce all'oggetto di classe `NSCalendarDate` di cui è stato invocato il metodo `isThisMonthsDate`: nel nostro caso, si tratta dell'oggetto `dataCreazione`, il cui metodo `isThisMonthsDate` è stato chiamato all'interno del metodo `nuoviValori`. Ogni oggetto di classe `NSCalendarDate` dispone di un metodo `monthOfYear`, che restituisce un numero da 1 a 12 indicante il mese dell'anno memorizzato nell'oggetto stesso. Nel nostro caso, indica il mese dell'anno in cui `dataCreazione` è stato creato. A destra dell'operatore di uguaglianza `==`, l'espressione `[[NSCalendarDate calendarDate] monthOfYear]` crea un nuovo oggetto di classe `NSCalendarDate` inizializzandolo alla data corrente; quindi ne estrae l'informazione relativa al mese dell'anno; l'operatore `==` restituirà pertanto un valore logico vero nel caso in cui il mese dell'anno memorizzato in `dataCreazione` coincida con il mese corrente, falso in caso contrario. L'esito del test viene restituito come valore di uscita del metodo `isThisMonthsDate`.

Benché l'applicazione possa sembrare banale, l'uso di una categoria è in realtà una grande trovata. Pensate a che cosa avremmo dovuto fare se non avessimo avuto la possibilità di estendere la classe `NSCalendarDate` per verificare che i dati che stiamo inserendo sulla piovosità della regione non siano più vecchi di un mese. Avremmo dovuto creare un metodo della classe `regione` destinato a fare un test logico di questo tipo: `[dataCreazione monthOfYear] == [[NSCalendarDate calendarDate] monthOfYear]`. E che sarà mai, direte voi? In effetti non è nulla di sconcertante. Però non è elegante: da un punto di vista logico, il confronto fra due date è una cosa che compete all'oggetto che ha il compito di immagazzinare e gestire le date stesse; è compito di un oggetto della classe `NSCalendarDate`

stabilire se il mese memorizzato in qualche maniera al suo interno sia o meno il mese corrente, non è compito di un oggetto della classe `regione`. È chiaro che potete fare questa verifica anche all'interno della classe `regione`, ma sarebbe una violazione delle regole della programmazione ad oggetti: ogni oggetto è responsabile di sé stesso; non c'è nessun motivo per cui un oggetto di classe `regione` debba farsi i fatti di un oggetto di classe `NSCalendarDate`. Lo so, sembra un po' una questione di lana caprina. Bisogna farci l'abitudine. Finché i programmi sono piccoli e semplici va tutto bene; ma se il vostro programma cresce di dimensioni, allora… e poi, una categoria è una cosa che potete trasportare in un qualunque altro programma semplicemente con un copia-incolla, proprio perché è un'estensione di una classe che *non dipende* da come avete implementato la classe che invece utilizzerà quella categoria. In altre parole: quando farete un nuovo programma che si occuperà di valutare se le uova che avete in frigo sono scadute o no, potrete fare un copia-incolla della categoria definita in questo Esempio3 e potrete usare senza timore il metodo `isThisMonthsDate`, perché tanto la categoria quanto il metodo ignorano completamente il contesto in cui verranno usati, ovvero non dipendono da come sono fatte la classe `regione` o la classe `uova` che userete nel programma di gestione del vostro frigorifero.

Le cose difficili sono finite: non ci resta che dare un'occhiata a main.m, notando che sono state semplicemente aggiunte le opzioni per la creazione di una nuova regione. Come già visto nel capitolo precedente, l'uso della classe `NSMutableArray` per l'oggetto `citta` e l'uso del tipo oggetto generico `id` nelle funzioni `InserisciValori()` e `MostraRisultati()` consente di manipolare indifferentemente oggetti di classe `localita`, `aeroporto` (sottoclasse di `localita`) o `regione`, *purché supportino tutte e tre i metodi* `initWithName:`, `nuoviValori` e `mostraValori`.

Protocolli formali

Finora abbiamo ampliato il nostro programma aggiungengo nuove classi o creando delle sottoclassi. Il file main.m ha richiesto solo piccole modifiche, dal momento che le due funzioni principali, `InserisciValori()` e `MostraRisultati()`, sono in grado di maneggiare oggetti appartenenti a classi qualunque, purché questi rispondano ai metodi `nuoviValori` e `mostraValori`. L'idea è buona, ma presenta un rischio: il nostro programma potrebbe ampliarsi a tal punto che, quando ci venisse richiesto di aggiungere una nuova classe, potremmo facilmente perdere il conto di quali e quanti metodi essa debba implementare affinché il file main.m non debba perdere in generalità. Naturalmente, possiamo tenere un appunto da qualche parte scrivendo che è obbligatorio che ogni nuova classe che venga aggiunta implementi i metodi `nuoviValori` e `mostraValori`; tuttavia Objective-C ci offre un modo per fare questo in maniera chiara e elegante, ovvero usando un *protocollo formale* o, più rapidamente, un *protocollo*. Esso è semplicemente un elenco di metodi privi di implementazione (è solo un'interfaccia, se volete); ogni classe dichiarata *conforme* a quel protocollo è *obbligata* ad implementare *tutti* i metodi dichiarati nel protocollo stesso. Vediamo come l'uso di un protocollo possa contribuire a rendere più robusto il codice del nostro programma.

Aprite il vostro editor di testo preferito e digitate il seguente codice:

```
@protocol luogoGeografico
- (id)initWithName:(NSString *)unNome;
- (NSString *)nome;
- (void)nuoviValori;
- (void)mostraValori;
@end
```

Salvatelo come luogoGeografico.h in una nuova cartella, chiamata Esempio4. Quindi digitate quanto segue (al solito, in grassetto ci sono le modifiche rispetto all'esempio precedente):

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

```
#import  "luogoGeografico.h"

#define YES         1
#define NO          0

@interface  localita  :  NSObject  <luogoGeografico>
{
    NSString    *nome;
    double      temperaturaMedia;
    double      umiditaMedia;
    int         numeroGiorni;
}

- (void)nuovaTemperatura:(double)unaTemperatura;
- (void)nuovaUmidita:(double)unaUmidita;

@end
```

Salvatelo come localita.h. Notate che rispetto alla versione dello stesso file dell'Esempio3 sono state tolte delle righe! Digitate quindi:

```
#import "localita.h"
#import  "luogoGeografico.h"

@implementation localita

- (id)initWithName:(NSString *)unNome
{
    [super init];
    nome=[[NSString alloc] initWithString:unNome];
    temperaturaMedia=0.0;
    umiditaMedia=0.0;
    numeroGiorni=0;
    return self;
}

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

- (NSString *)nome
{
    return nome;
}

- (void)nuoviValori
{
    double  t,u;

    printf("\n");
```

```
    printf("Citta': %s\n",[nome cString]);
    printf("Inserisci la temperatura di oggi a mezzigiorno (gradi C): ");
    scanf("%lg",&t);
    printf("Inserisci l'umidita' di oggi a mezzigiorno: ");
    scanf("%lg",&u);
    [self nuovaTemperatura:t];
    [self nuovaUmidita:u];
}

- (void)mostraValori
{
    printf("\n");
    printf("Citta': %s\n",[nome cString]);
      printf("%s: temperatura media: %g (gradi C) e umidita' media:
%g\n",[nome cString],temperaturaMedia,umiditaMedia);
       printf("Le medie sono state calcolate per un totale di %d
giorni\n\n",numeroGiorni);
}

- (void)nuovaTemperatura:(double)unaTemperatura
{
    temperaturaMedia=(temperaturaMedia*numeroGiorni+unaTemperatura)/(numero
Giorni+1);
    numeroGiorni++;
}

- (void)nuovaUmidita:(double)unaUmidita
{
    umiditaMedia=(umiditaMedia*(numeroGiorni-1)+unaUmidita)/numeroGiorni;
}

@end
```

Salvatelo come localita.m. Copiate così come sono i file aeroporto.h e aeroporto.m, quindi digitate quanto segue (anche qui mancano delle righe rispetto alla versione precedente!):

```
#import <Cocoa/Cocoa.h>
#import  "luogoGeografico.h"

@interface  regione  :  NSObject  <luogoGeografico>
{
    NSCalendarDate      *dataCreazione;
    int                 mmDiPioggia;
    NSString            *nome;
}

@end

@interface NSCalendarDate (regione)
- (BOOL)isThisMonthsDate;
@end
```

che naturalmente salverete come regione.h. Ora è la volta di quanto segue:

```
#import "regione.h"
#import  "luogoGeografico.h"

@implementation regione

- (id)initWithName:(NSString *)unNome
{
    [super init];
    nome=[[NSString alloc] initWithString:unNome];
    dataCreazione=[NSCalendarDate calendarDate];
    [dataCreazione retain];
    mmDiPioggia=0;
    return self;
}

- (void)dealloc
{
    [nome release];
    [dataCreazione release];
    [super dealloc];
}

- (NSString *)nome
{
    return nome;
}

- (void)nuoviValori
{
    int     pioggia;

    printf("\n");
    printf("Regione: %s\n",[nome cString]);
    printf("Inserisci i mm di pioggia caduti oggi: ");
    scanf("%d",&pioggia);
    if([dataCreazione isThisMonthsDate])
        mmDiPioggia+=pioggia;
    else
    {
        [dataCreazione release];
        dataCreazione=[NSCalendarDate calendarDate];
        [dataCreazione retain];
        mmDiPioggia=pioggia;
    }
}

- (void)mostraValori
{
    printf("\n");
    printf("Regione: %s\n",[nome cString]);
           printf("Sono   caduti   %d   mm   di   pioggia   nel   mese
```

```
corrente\n\n",mmDiPioggia);
}

@end

@implementation NSCalendarDate (regione)
- (BOOL)isThisMonthsDate
{
        return ([self monthOfYear]==[[NSCalendarDate calendarDate]
monthOfYear]);
}
@end
```

Salvatelo come regione.m. Quindi copiate così com'è il file main.m. Ricompilate tutti i file con estensione .m e linkateli in Esempio4. Eseguitelo e giocateci. Come dite? È identico a Esempio3? Meno male! Vuol dire che abbiamo fatto le cose per bene! Sì, ma, esattamente: che cosa abbiamo fatto?

Partiamo dal file più importante: luogoGeografico.h. Abbiamo dichiarato un protocollo mediante la direttiva `@protocol` e gli abbiamo dato un nome (`luogoGeografico`). Seguono quattro dichiarazioni di metodi: tutte le classi che vorranno essere conformi a (o *adottare)* questo protocollo dovranno implementare questi quattro metodi. Oltre ai già discussi `inserisciValori` e `mostraValori`, abbiamo aggiunto `initWithName:`, perché è pratico avere un metodo comune di inizializzazione per tutti gli oggetti che manipoleremo nel nostro programma, e, di conseguenza, abbiamo aggiunto anche `nome`. Come avrete notato, un protocollo consta solo di un elenco (di un'interfaccia), non c'è l'implementazione dei metodi dichiarati. Questo perché il protocollo non specifica *come* i metodi debbano essere implementati o *che cosa* debbano fare; si limita a specificare che essi *devono* essere implementati.

La classe `localita` è la prima che analizziamo che fa uso del protocollo `luogoGeografico`. Dobbiamo pertanto importare il file luogoGeografico.h e dichiarare, nella linea con la direttiva `@interface`, che la classe `localita` adotta il protocollo `luogoGeografico` indicandolo tra parentesi angolari <>. Nulla vieta di adottare più di un protocollo: i loro nomi andranno inseriti, separati da una virgola, tra le medesime parentesi angolari. È da notare che abbiamo tolto dalla dichiarazione dell'interfaccia della classe `localita` i metodi già dichiarati nel protocollo. Infatti, `localita` *deve* implementarli per il fatto stesso che è stata dichiarata conforme al protocollo `luogoGeografico`, non c'è bisogno di elencare nel suo file di interfaccia i metodi richiesti dal protocollo stesso.

Il file localita.m non subisce modifiche, se non l'aggiunta della riga con cui si importa il file luogoGeografico.h. È fondamentale ricordare che bisogna importare il file di header del protocollo *dopo* il file di interfaccia della classe che si sta implementando; infatti, in un file di implementazione, la *prima* cosa che deve essere importata è il file di interfaccia della classe!

Molto interessante è la ragione per cui la classe `aeroporto`, che ricordo essere sottoclasse di `localita`, apparentemente *non* adotta il protocollo `luogoGeografico`. Ho detto *apparentemente*, perché in realtà il protocollo è ovviamente adottato dalla classe genitrice `localita`. Se avessimo richiesto esplicitamente l'adozione del protocollo `luogoGeografico` da parte di `aeroporto` inserendo l'istruzione `<luogoGeografico>` al punto opportuno nel file di interfaccia della classe, saremmo stati costretti ad implementare *all'interno* della classe `aeroporto` i metodi richiesti dal protocollo (pena un errore in fase di compilazione), senza poter sfruttare i metodi già implementati dalla classe genitrice. Quindi: se una classe implementa un protocollo, le sue sottoclassi *non* sono tenute ad implementarlo, in quanto i metodi richiesti sono comunque già implementati nella classe genitrice. Questo è molto comodo quando si sottoclassa una classe messa a disposizione dal framework Cocoa (ovvero dal sistema operativo): molto spesso le classi Cocoa sono conformi a uno o più protocolli; tuttavia, il programmatore, sottoclassandole, non deve preoccuparsi di implementare i protocolli stessi (meno male!). La classe `regione` subisce modifiche analoghe a `localita`. Per finire, il file

main.m non viene modificato. Ripetiamolo: l'Esempio4 non fa nulla di diverso dall'Esempio3. Ma il giorno in cui dovremo aggiungere una classe al programma per gestire tipologie di dati diverse da quelle usate finora, basterà richiedere che la classe sia conforme al protocollo `luogoGeografico` e saremo *sicuri* che il programma non necessiterà di modifiche, specialmente al file main.m, in quanto saranno rispettati i criteri di generalità necessari per il suo funzionamento. In un programma così piccolo tutto ciò può essere superfluo, ma se il vostro programma cresce di dimensioni…

Protocolli informali

La rigorosità del sistema dei protocolli è molto utile, ma in alcuni casi può essere restrittiva: potrebbe essere necessario, ad esempio, elencare una serie di metodi che una classe *può* implementare, ma non necessariamente *deve*. Addirittura, la classe potrebbe scegliere di implementare solo alcuni di questi metodi e non tutti. Anche questa possibilità è stata prevista dai geniali ideatori di Objective-C, e si chiama *protocollo informale*. Il nome, in realtà, è fuorviante, in quanto si tratta in effetti di una categoria dotata di interfaccia, ma non necessariamente di implementazione. Non ci soffermeremo qui sui protocolli informali, né forniremo esempi; li citiamo solo perché essi sono talvolta usati dalle classi di Cocoa. Se desiderate creare protocolli informali nei vostri programmi, non dovete fare altro che scriverne le interfacce come se fossero delle categorie, dichiarare la classe conforme a quelle categorie, e poi implementarne solo i metodi che vi fanno comodo.

Bibliografia

Mi perdonerete se presento solo bibliografia in inglese, ma non sono a conoscenza di testi in italiano. Inoltre, presento solo le risorse su cui ho studiato, per serietà. Ecco quindi il poco che conosco (e che mi sento di consigliarvi):

Apple fornisce un buon testo (in pdf) su Objective-C, direttamente nella documentazione inclusa con i DeveloperTools (o XCode o XTools, chiamateli come volete). Il file si chiama ObjC.pdf, e, sul mio Mac, si trova a questo percorso:
/Developer/Documentation/Cocoa/Conceptual/ObjectiveC/ObjC.pdf

Una guida di riferimento, assolutamente non adatta per iniziare a studiare Objective-C ma molto utile quando si vogliano rintracciare in fretta i dettagli (e le sottigliezze) di un qualche costrutto del linguaggio, è
Andrew M. Duncan, *Objective-C Pocket Reference*, O'Reilly (2002)

Per informazioni sulle classi Cocoa che abbiamo usato nei vari esempi, invece, le fonti migliori sono i file di documentazione inclusi sempre con la vostra installazione dei DeveloperTools, quindi sul vostro hard disk in /Developer/Documentation/Cocoa/Reference. Se non avete voglia di navigare tra centinaia di file html, vi consiglio un paio di programmini freeware particolarmente adatti per sfogliare le varie pagine di documentazione sulle classi Cocoa. Si tratta di **AppKiDo** (http://homepage.mac.com/aglee/downloads/) e **CocoaBrowser** (http://homepage2.nifty.com/hoshi-takanori/cocoa-browser/). Vi suggerisco di andarvi a vedere i dettagli delle classi `NSString`, `NSMutableArray`, `NSEnumerator` e `NSCalendarDate` che abbiamo usato nei nostri esempi.