ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

II Facoltà di Ingegneria

Corso di INGEGNERIA INFORMATICA

Laurea in SISTEMI DISTRIBUITI

Da Java a Objective-C: porting e dispositivi portatili

Candidato
Nompleggio Pietro Antonio

Relatore
Prof. Andrea Omicini

Correlatore
Stefano Mariani

Anno Accademico 2011/2012 - Sessione II

A Giulia, per avermi dato la forza di continuare,
a mia sorella, che mi ha sempre aiutato,
ai miei genitori, senza i quali tutto questo
non sarebbe stato possibile.

Indice

Introduzione **9**

1 Linguaggi di Programmazione ad Oggetti **13**

1 Ogni cosa è un oggetto . . . 13
2 Storia di Objective-C . . . 16
3 Storia di Java . . . 17

2 Differenze tra Java e Objective-C **19**

1 Package e Framework . . . 19
2 E' tutto un oggetto, o quasi . . . 21
2.1 Tipi Primitivi in Java . . . 21
2.2 Tipi Primitivi in Objective-C . . . 22
3 Caratteristica e struttura di una classe . . . 22
3.1 Definizione di una classe in Java . . . 23
3.2 Definizione di una classe in Objective-C . . . 24
4 Ereditarietà . . . 26
4.1 Ereditarietà in Java . . . 27
4.2 Ereditarietà in Objective-C . . . 27
5 Poliformismo . . . 28
5.1 Poliformismo in Java . . . 28
5.2 Poliformismo in Objective-C . . . 30
6 Definizione dei campi . . . 31
6.1 I campi in Java . . . 31
6.2 I campi in Objective-C . . . 33
7 Definizione dei metodi . . . 34
7.1 Definizione dei metodi in Java . . . 34

7.2 Definizione dei metodi in Objective-C 35
8 Definizione dei costruttori 36
8.1 Costruttori in Java 37
8.2 Costruttori in Objective-C 37
9 Inner Class 39
9.1 Inner Class in Java 39
9.2 Inner Class in Objective-C 39
10 Gestione della memoria 40
10.1 Gestione della memoria in Java 40
10.2 Gestione della memoria in Objective-C 42
11 Invocazione dei metodi 44
11.1 Invocazione dei metodi in Java 45
11.2 Invocazione dei metodi in Objective-C 45
12 Categories 46
13 Definizione di Array 48
13.1 Definzione di Array in Java 48
13.2 Definzione di Array in Objective-C 51
14 Delegate 53
15 Socket 56
15.1 Socket in Objective-C 57
15.2 Socket in Java 61
16 Thread 63
16.1 Thread in Java 64
16.2 Thread in Objective-C 66
17 Blocks in Objective-C 70

3 Porting ed esempi di programmazione 73
1 Porting 73
2 Tool di traduzione: j2Objc 75
3 Esempio del Contatore 77
3.1 Implementazione in Java 77
3.2 Mia implementazione in Objective-C 78
3.3 Traduzione del Contatore con il Tool 79
4 Esempio Array 80

4.1 Implementazione in Java 80
4.2 Mia implementazione in Objective-C 82
4.3 Traduzione con il tool dell'esempio array 84
5 Esempio Inner Class 86
5.1 Implementazione in Java 86
5.2 Implementazione in Objective-C 87
5.3 Traduzione dell'esempio InnerClass con il tool 91
6 Thread 92
7 Chat 93
7.1 Implementazione del Client Java 93
7.2 Implementazione del Client in Objective-C 95
7.3 Traduzione dell'esempio della Chat 98

Conclusioni **101**

Bibliografia **103**

Ringraziamenti **105**

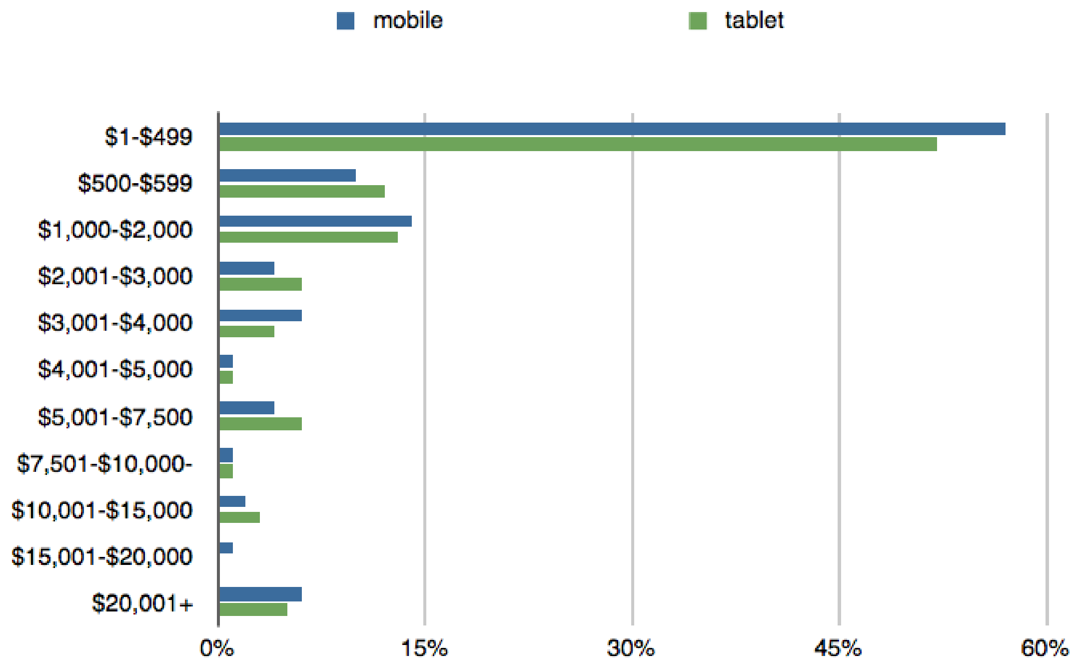
Introduzione

Nell'ultimo decennio il mondo dei dispotivi portatili è cambiato radicalmente, non si sarebbe mai potuto immaginare nel 2002 cosa oggi avremo avuto nelle nostre tasche: dispositivi in grado di connettersi a internet, mandare email, collegarsi ai più famosi social network, scattare fotografie, e paragonarsi e sostituirsi senza problemi a un personal computer. Ma cosa ci ha portato fino a qui? Cosa ha dettato questo cambiamento e chi?
La risposta più ovvia è che ciò sia avvenuto grazie ad Apple, Samsung, HTC, Blackberry, Nokia e i tanti altri distributori di dispostivi mobili, però se ci pensiamo, sono anni che i telefonini sono nelle nostre tasche ma non hanno mai suscitato tutto questo interesse e non sono mai stati così tecnologici. Qual'è stato il punto di svolta? Secondo la mia opinione il momento in cui tutto è cambiato è stato quando un uomo di nome Steve Jobs durante una conferenza del 2007 ha modificato per tutti il concetto di telefonino presentando il primo iPhone, un telefono rivuoluzionario, fu questo che disse durante quella storica conferenza:

> *"Today Apple Reinvent the Phone"*

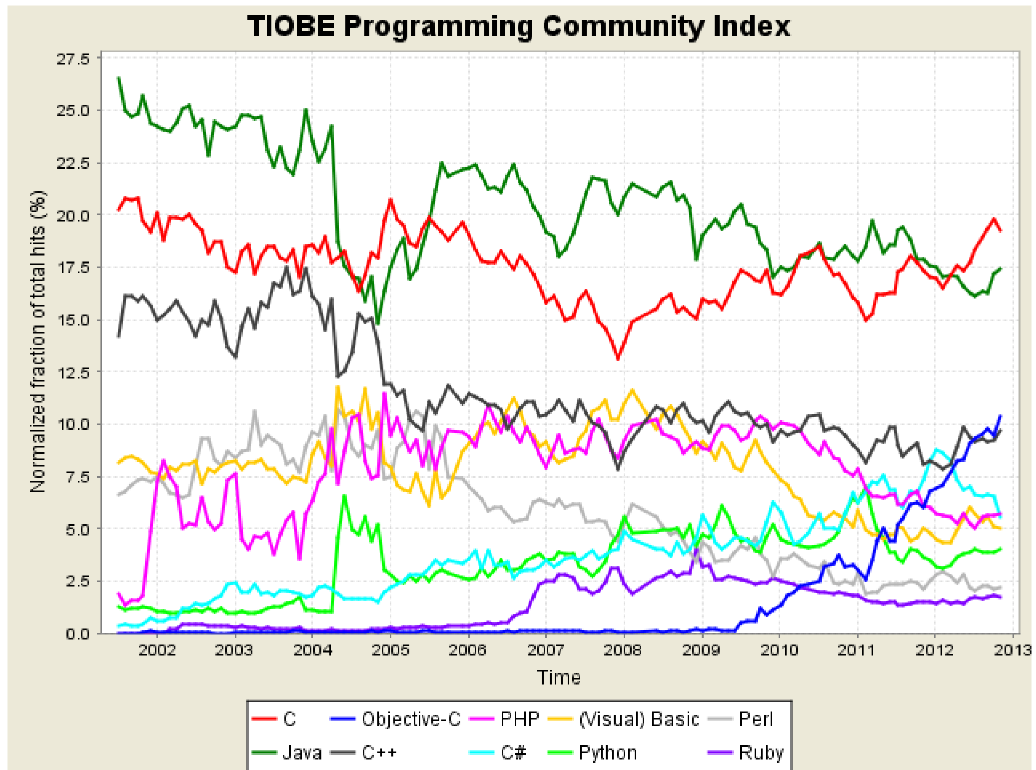
e un anno dopo, nell'estate del 2008 , aprì l'App Store, il negozio di applicazioni, un negozio virtuale in cui chiunque in possesso di un iPhone poteva acquistare applicazioni, il giorno dell'apertura il negozio conteneva 500 applicazioni, tre giorni dopo ne conteneva 800 , al giorno d'oggi possiamo contare circa 700.000 applicazioni presenti solo sull'App Store di Apple. Ovviamente questo suscitò l'interesse della concorrenza infatti BlackBerry e Nokia, e successivamente Windows e Google , costruirono i loro smartphone e aprirono il proprio market per i loro dispositivi portatili. Queste "App" sono diventate la chiave di successo dei telefonini, che oggi vengono chiamati smartphone,

cioè telefoni in grado di fare tutto, proprio per il fatto che esistono applicazioni in grado di fare tutto o quasi. L'Apple e in seguito tutti gli altri hanno fornito gli strumenti per poter realizzare le applicazioni per i dispostivi portatili agli sviluppatori e questo è diventato uno dei mercati più promettenti nell'ambito del lavoro. Qui di seguito è inserito un grafico che mostra quanto vengono pagati in media gli sviluppatori per dispositivi portatili al mese.



Quindi possiamo dedurre che può diventare un vero e proprio lavoro, ma quali linguaggi bisogna conoscere per poter sviluppare per questi dispositivi?

Per rispondere a queste domande vorrei mostrarvi un grafico, è un andamento preso dal sito web: "Tiobe.com", in cui è mostrata l'evoluzione nell'uso dei linguaggi di programmazione dal 2002 ad oggi.

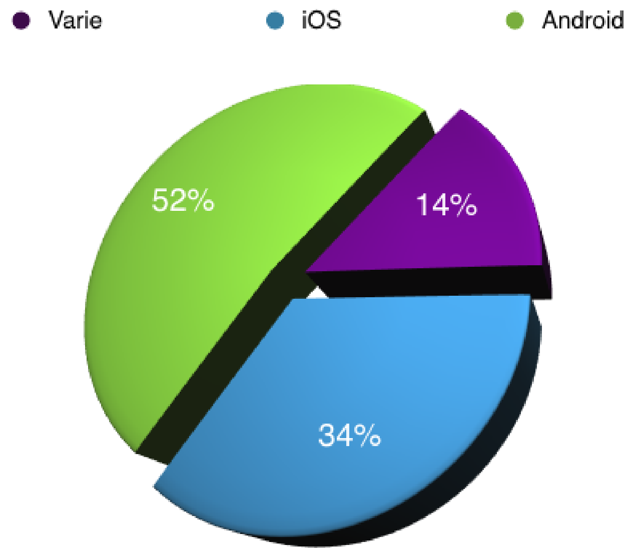


Il grafico mette in evidenza l'andamento dei linguaggi più usati per la creazione di sistemi software in generale. In particolare vorrei trattare due linguaggi : Java e Objective-C che potete vedere nel grafico rispettivamente con il colore verde e blu.

Come si può notare già dal 2002 Java (linea verde) era il linguaggio di programmazione più usato per la creazione di software, e invece l'Objective-C (linea blu) ha avuto una crescita esponenziale a partire dal 2009, questo farebbe pensare che l'Objective-C sia un linguaggio più giovane rispetto a Java, perchè fino al 2009 sembra che nessuno o quasi nessuno l'abbia mai usato, invece non è così, perché l'Objective-C è stato sviluppato prima di Java e veniva usato per sviluppare applicazioni Desktop, quindi questa crescita esponenziale è dovuta all'uscita dell'iPhone e successivamente l'iPad,che sfruttano nel loro sistema operativo l'Objective-C usato da Apple.

Quindi attualmente i due linguaggi più in uso nello sviluppo di software per dispositivi portatili sono da una parte Java, che domina una fetta del merca-

to con il sistema operativo Android, e dall'altro lato c 'è l'Objective-C, usato per scrivere applicazioni per iPhone e iPad, come possiamo vedere da una ricerca fatta a Giugno 2012, il mercato dei dispositivi mobili è prevalentemente diviso tra questi due attori:



Mettendo però da parte le questioni di marketing e facendo un'analisi ingegneristica dei due linguaggi possiamo dire che Java e Objective-C essendo entrambi linguaggi di programmazione ad oggetti avranno alcune caratteristiche simili, ma molto altri concetti invece saranno diversi tra loro in quanto non è possibile passare direttamente da un linguaggio ad un'altro in caso di porting, è per questo che l'obiettivo di questa tesi è quello di descrivere quali sono le loro differenze tra questi due linguaggi, in modo da capire come poter passare da Java a Objective-C e se esistono degli strumenti che ci possono aiutare in questa traduzione.

Capitolo 1

Linguaggi di Programmazione ad Oggetti

Come primo argomento vorrei introdurre il concetto di linguaggio di programmazione ad oggetti e inoltre uno dei primi linguaggi di programmazione, da cui Java e Objective-C discendono e da cui discendono la maggior parte dei linguaggi di programmazione ad Oggetti.

1 Ogni cosa è un oggetto

La programmazione orientata agli oggetti (OOP) ha avuto pieno sviluppo a partire dai primi anni '80, è un paradigma di programmazione, ossia un insieme di strumenti concettuali forniti da un linguaggio di programmazione per la stesura del codice sorgente di un programma. Essa permette di definire oggetti software in grado di interagire gli uni con gli altri attraverso lo scambio di messaggi. Uno dei primi linguaggi di programmazione a comparire sulla scena fu il Smalltalk, che fu lo spunto per la creazione di altri linguaggi come per esempio l'estensione di C in Objective-C negli anni '80, e successivamente anche in Java negli anni '90.
In questa tesi userò il paradigma ad oggetti per individuare le più importanti similitudini tra i due linguaggi. Possiamo ora elencare le caratteristiche essenziali definite da Alan Kay che progettò Smalltalk:

- *Everything is an Object.* Ogni cosa è un oggetto.

- *A Program is a bunch of object telling each other what to do by sending messages.* Un programma è un insieme di oggetti che interagiscono tra di lore mediate lo scambio reciproco di messaggi.

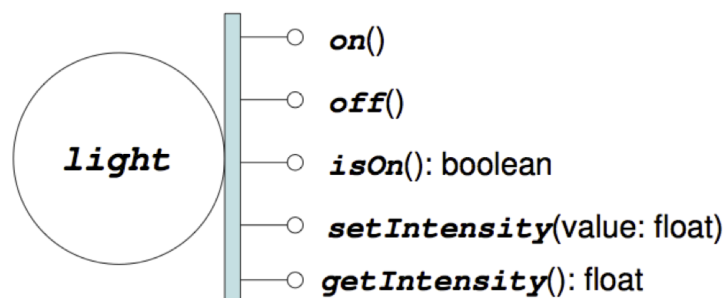
- *Every object has its own memory (state) composed by other objects.* Ogni oggetto ha la propria memoria (state), composta da altri oggetti.

- *Every object is an instante of a Class.* Ogni oggetto è istanza di una classe.

- *Every object has a Interface, which defines what messages it can receivem and so what the type of the object is.* Ogni oggetto ha una interfaccia che definisce la natura del messaggio che esso può ricevere, ovvero il tipo dell'oggetto.

Quindi nella OOP ogni entità astratta o concreta del problema viene modellata mediante l'astrazione di oggetto o insieme di oggetti. Ogni oggetto rappresenta una entità dotata di stato, in grado di svolgere un certo insieme di operazioni ,dette metodi , che costituiscono il comportamento del nostro oggetto. Questi metodi possono essere interpretati come servizi, che l'oggetto offre ai suoi possibili clienti. I clienti di un oggetto saranno altri oggetti che invocheranno i suoi servizi ossia i metodi, richiedendone l'esecuzione. L'insieme dei metodi costituisce l'interfaccia dell'oggetto verso i suoi clienti.

Facciamo un semplice esempio, proviamo a modellare l'oggetto lampadina che può essere caratterizzato dai seguenti metodi:



Quindi possiamo accendere e spegnere la lampadina con il metodi `on()` e `off()`, possiamo verificare se è accesa con il metodo `isOn()`, e possiamo dettare l'intensità o recuperare il valore dell'intensità con i metodi `setIntensity()` e `getIntensity()`. Quindi abbiamo fatto l'astrazione di una semplice entità come la lampadina e l'abbiamo modellata come oggetto, e quindi vediamo che ritorna il concetto di **Ogni cosa è un oggetto**.

E' importante anche sottolineare l'importanza dello stato che può possedere un'oggetto in quanto esso può cambiare dinamicamente ed è una caratteristica essenziale dell'astrazione di oggetto. Nel precedente esempio lo stato definisce se la lampadina è accesa o spenta, e quale sia l'intensità di luce. Lo stato di un oggetto è **nascosto** ai clienti dell'oggetto che ne percepiscono solamente l'interfaccia, perché per esempio per sapere se la lampadina è accesa è necessario invocare il metodo `isOn()`. Lo stato può cambiare dinamicamente sempre in virtù dei metodi invocati sull'oggetto, come per esempio il metodo `on()` cambia lo stato della lampadina da spenta ad accesa.

Un'altra caratteristica chiave è l'interazione via scambio di messaggi. Un cliente (un'altro oggetto) interagisce con un oggetto inviandogli un messaggio, con cui richiede l'esecuzione di un metodo. L'oggetto che in questo caso funge da servitore risponde all'invio del messaggio eseguendo il metodo corrispondente. Tale esecuzione può portare alla generazione di informazioni di ritorno, che vengono restituite al cliente. Il messaggio sarà caratterizzato da un nome ed eventualmente da parametri, che possono essere a loro volta oggetti stessi. Questa interpretazione definita in particolare per linguaggi altamente dinamici quali Smalltalk, conduce ad interpretare la computazione in generale che caratterizza un sistema OO come puro scambio di messaggi fra oggetti.

L'insieme dei metodi costituisce l'**interfaccia** dell'oggetto, ovvero l'insieme delle richieste che un cliente può fare all'oggetto, quindi l'insieme delle possibili interazioni. Inoltre un oggetto può possedere anche metodi cosiddetti privati che non fanno parte dell'interfaccia. Nell'ottica cliente/servitore l'interfaccia può essere concepita come **contratto** che l'oggetto si impegna a rispettare nei confronti dei suoi utilizzatori, in termini di operazioni e servizi

forniti a caratterizzazione dell'operazione/servizio fornito.

Un oggetto è composto da oggetti, ossia lo stato di un oggetto è mantenuto da strutture dati a loro volta oggetti definiti **campi** dell'oggetto. Tali campi non sono visibili dai clienti dell'oggetto, e solo l'oggetto può accedervi ed eseguire operazioni su di essi, tramite metodi sia pubblici sia privati.
Infine i linguaggi OO mettono a disposizione oggetti elementari che rappresentano valori di tipi di dati semplici come numeri (interi, reali), caratteri, stringhe ecc... . Inoltre Java e Objective-C adottano un approccio ibrido, per cui questi tipi di dato semplici sono forniti sia come oggetto, sia come tipi di dati primitivi.

Con il paradigma ad oggetto una applicazione è dunque inquadrabile come un sistema composto da un insieme (dinamico) di oggetti che interagiscono mediante lo scambio di messaggi (invocazione di metodi).

Il quarto principio dell'OOP dice che "Every object is an istance of a Class" quindi ogni oggetto è istanza di una **classe**, essa quindi può essere considerata un modello (un template) per creare oggetti. Tornando all'esempio di prima dell'oggetto lampadina, se allarghiamo il nostro esempio all'ambiente casa, possiamo definire altri tipi di oggetti come per esempio possono essere gli interruttori, che servono per accendere e spegnere le lampadine, oppure ci possono essere anche altri oggetti come la lavatrice, il frigorifero, il forno, tutti questi oggetti possono parlare tra di loro, per esempio il pulsante interagisce con la lampadina accendendola o spegnendola. Quindi ogni singolo oggetto, per esempio di questo ambiente casa ,nel paradigma object-oriented può essere catturato dalla nozione di classe, che identifica una categoria, una famiglia, un tipo di oggetto, quindi avremo la classe Lampadina, la classe Pulsante, quella Frigorifero ecc...

2 Storia di Objective-C

Brad Cox e Tim Love sono i principali fondatori dell'Objective-C, inizialmente chiamato "Object-Oriented Programming in C". L'obbiettivo era ag-

giungere capacità a Smalltalk che richiede un interprete C, senza progettare da zero un nuovo linguaggio. Cox pubblicò la prima descrizione formale di quello che era ormai diventato l'Objective-C nel 1986. Nel 1988 la NeXT Computer ha adottato l'Objective-C come linguaggio di sviluppo primario. NeXT ha arricchito l'Objective-C, creando una collezione di classi che sono diventate le fondamenta per le nuove applicazioni, e strumenti di sviluppo, e una significante porzione del loro sistema operativo, il NEXTSTEP. Il sistema operativo NEXTSTEP ha portato le OpenStep API che definiscono un consistente insieme di oggetti e interfacce che possono essere portate in più piattaforme. Insieme Objective-C e NEXTSTEP sono stati acclamati come ambienti di sviluppo innovativi, gli unici a fornire veramente la promessa di progettazione orientata agli oggetti. Ma per una serie di motivi, in particolare l'inerzia dell'industria ad usare il C++, NeXT e l'Objective-C rimasero poco più di una curiosità, un fulgido esempio di ciò che poteva essere realizzato, se solo fosse stato abbracciato da un segmento importante del settore. Questa soglia fu finalmente attraversata nel 1996 quando l'Apple acquistò NeXT Computer, il sistema operativo NEXTSTEP, e l'intera suite di strumenti di sviluppo in Objective-C. Apple ha fatto del NEXTSTEP una pietra miliare del loro nuovo sistema operativo Mac OS X. Il nuovo framework venne ribattezzato "Cocoa" e da allora è cresciuto e maturato in un ricco e potente set di strumenti, non solo per le applicazioni di personal computer, ma anche per dispostivi innovativi come l'iPhone e l'iPad.

3 Storia di Java

Java è stato creato da ricerche effettuate alla Stanford University agli inizi degli anni Novanta. Nel 1992 nasce il linguaggio Oak, prodotto da Sun Microsystem, tale nome venne successivamente cambiato in Java per problemi di copyright. Per facilitare il passaggio a Java ai programmatori vecchio stampo, e legati a linguaggi come il C++, la sintassi di base è stata mantenuta pressochè identica a quella del C++, e non sono state introdotte caratteristiche ritenute troppo complesse a livello di linguaggio, in modo da renderlo più leggero e veloce. In un primo momento Sun decise di destinare questo nuovo prodotto alla creazione di applicazioni complesse per piccoli dispositivi elet-

tronici, e fu solo nel 1993 che con l'esplosione di internet Java iniziò a farsi notare come strumento per iniziare a programmare per internet, e venne introdotto nei browser la Java Virtual Machine grazie alla quale le pagine web divennero interattive. Ed è per questo che troviamo Java in cima a quella classifica, perchè in tutti questi anni Java è stato usato non solo per piccoli dispositivi, o per sistemi entertainment ma anche per i browser, e per i giochi e per tantissime altre cose che usiamo durante il giorno. Il 27 gennaio 2010 Sun Microsystem venne acquisita da Oracle.

Capitolo 2

Differenze tra Java e Objective-C

Ora è necessario descrivere le principali differenze che i due linguaggi di programmazione hanno nell'implementazione delle principali strutture base e delle caratteristiche fondamentali in modo da avere una conoscenza base per poter programmare.

1 Package e Framework

I package in Java sono collezioni di classi, messe insieme in un unico "pacchetto", esse costituiscono delle vere e proprie librerie. Possiamo creare un package racchiudendovi ad esempio le classi che vogliamo utilizzare su diversi progetti, infatti la vera e propria potenza odierna di Java è il numero di classi già definite all'interno dei package Java che svolgono i più svariati compiti, ed è anche la parte che continua sempre più a crescere e ad aggiornarsi con le nuove versioni di Java. Il nucleo di Java contiene solo le parole chiave per la costruzione delle classi, i commenti, i normali costrutti come if, switch, while e for , tutto il resto è implementato nei package del linguaggio, comprese le normali primitive di input e output dove dobbiamo utilizzare il package `java.lang`. Per usare un package in una classe, prima della definizione della classe dobbiamo inserire import, ad esempio volendo usare il package lang dobbiamo inserire:

```
import java.lang
```

Questo package che ho appena introdotto è il più importante di Java, esso racchiude le classi fondamentali del linguaggio, tanto che non serve dichiararne l'import, pechè Java lo importa automaticamente. Per esempio alcune delle classi che troviamo in questo package sono:

- Boolean;
- Byte;
- Character;
- Double;
- Integer;
- Math;
- Object;
- String;
- Number;

E tantissimi altri, quindi come si può vedere questo package è il cuore di Java.

In Objective-C non abbiamo i package, ma abbiamo qualcosa di simile che viene chiamato framework, esse sono simili a delle librerie condivise, che possono essere dinamicamente caricate dentro un programma. Una libreria fondamentale nell'Objective-C paragonabile a `java.lang` è il Foundation Framework, che definisce un livello base di classi e di primitive che non vengono coperte dal solo Objective-C. Questo framework è stato sviluppato da NeXT Computer, acquisita poi da Apple, che ha in carico tutt'ora lo sviluppo e l'evoluzione di questo linguaggio in quanto fa parte del loro ambiente di sviluppo di tutti i loro prodotto. Nel Foundation Framework troviamo classi importanti come:

- NSString;
- NSArray;
- NSDictionary;
- NSSet;
- NSNumber;
- NSData;

E come in Java ci sono tantissime altre classi che offre il Foundation Framework ed anche qui è il nucleo dell'Objective-C e possiamo importarlo in una classe nel seguente modo:

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

2 E' tutto un oggetto, o quasi

Java e Objective-C sono due linguaggi di programmazione ad oggetti, ma oltre agli oggetti esistono altre entità che sono valori di tipo semplice definiti **primitivi**, ovvero numeri interi e reali, caratteri e boolean. In linguaggi come Smalltalk, tutto è un oggetto, compresi i numeri, valori booleani ecc... invece in Java e Objective-C funziona in modo leggermente diverso.

2.1 Tipi Primitivi in Java

In Java possiamo trovare sia tipi primitivi come: int, double, boolean, char, long e float sia oggetti come Stringhe e Array. Quindi per esempio l'istruzione per creare un int è la seguente:

```
int c = 10;
```

quindi la variabile c avrà il valore 10, invece per esempio se vogliamo creare un oggetto dobbiamo fare:

```
String str = "Ciao";
```

La creazione dinamica di un oggetto avviene specificando quale sia la classe che ne descrive struttura/comportamento in Java ciò avviene mediante l'operatore `new`:

```
Counter c = new Counter();
```

Quindi l'operatore `new` crea un nuovo oggetto della classe specificata, invocando il costruttore per la sua inizializzazione, l'operatore `new` restituisce un **riferimento** all'oggetto, ovvero ciò permette di riferire, identificare lo specifico oggetto per potervi poi interagire, per potergli inviare i messaggi, ovvero invocare metodi.

2.2 Tipi Primitivi in Objective-C

In Objective-C come in Java i tipi primitivi vengono dichiarati e inizializzati nello stesso modo:

```
int d = 25;
```

invece gli oggetti vengono dichiarati e inizializzati in modo diverso, per esempio una stringa viene creata nel seguente modo:

```
NSString *str = @"Ciao";
```

innanzi tutto vorrei aprire una piccola parentesi sul carattere `@`, che in Objective-C serve per dichiarare che questa stringa non è una stringa C, ma un oggetto NSString. Gli oggetti come si vede vengono creati con il prefisso `*` che sta a indicare che è un oggetto, però diversamente da Java non si usa l'operatore `new`, ma viene creato e inizializzato nel seguente modo:

```
Counter *c = [[Counter alloc] init];
```

le keyword `alloc` e `init` verrano spiegate successivamente nel paragrafo 6.2, invece per quanto riguarda le parentesi quadre, che vengono utilizzate per invocare i metodi, verranno descritte nel paragrafo 9.2.

3 Caratteristica e struttura di una classe

Una classe è caratterizzata da un nome e dalla descrizione dei suoi attributi, ovvero:

Campi, definiti data member, definiscono le strutture dati interne all'oggetto, quindi il suo stato;

Metodi, definiti anche funzioni membro, definiscono il comportamento dell'oggetto, le operazioni che è in grado di eseguire.

Fra i metodi ne esistono alcuni speciali, definiti **costruttori**, che vengono invocati solo all'atto della creazione dinamica di un oggetto per inizializzarlo opportunamente. Ogni oggetto creato dinamicamente come istanza della classe X avrà la struttura e il comportamento descritto dalla classe X. Il suo specifico stato dinamico poi varierà a seconda delle interazioni che avrà nel corso della sua esistenza.

3.1 Definizione di una classe in Java

In Java la sintassi per definire una classe è:

```
public class <NomeClasse> {
  <definizione dei campi>
  <definzione dei costruttori>
  <definzione dei metodi>
}
```

public, serve per dichiarare che la nostra classe sarà pubblica e quindi sarà visibile anche da altre classi esternamente;

class, definisce che stiamo dichiarando una classe.

Proviamo a fare l'esempio di una classe completa di un semplice oggetto Contatore:

```
public class Counter {
  private int count;    //campo

  public Counter () {   //costruttore
    count = 0;
  }

  public Counter(int v) {   //costruttore
    count = v;
  }

  public void setValue(int v) { //metodo
    count = v;
  }

  public void inc() {   //metodo
    count++;
  }

  public int getValue() {   //metodo
    return count;
  }

}
```

In Java esiste una precisa relazione fra classi e file, in particolare esiste una relazione 1 a 1 fra classi pubbliche e i file *.java. Ogni sorgente Java può contenere la definizione di una e una sola classe pubblica, quindi per esempio la classe Counter sarà descritta nel file sorgente Counter.java.

3.2 Definizione di una classe in Objective-C

In Objective-C la sintassi per definire una classe è molto diversa rispetto a quella di Java. Innazi tutto esiste, come in Java, una relazione fra classi e file, ma essendo una estensione del linguaggio C in Objective-C abbiamo due file per definire una classe, il file .h chiamato Header file, che conterrà le dichiarazioni di classe, tipo, metodi e costanti, e invece il file .m che sarà il Source file dove ci saranno le implementazioni della classe. Vediamo un esempio sempre con l'oggetto contatore:

`Counter.h`

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

@interface Counter : NSObject {
    int cont;
}

- (id)initWithCount:(int)value;
- (void)setCount:(int)value;
- (void)inc;
- (void)dec;
- (void)resetCount;
- (int)getCount;

@end
```

`Counter.m`

```
#import "Counter.h"

@implementation Counter
```

```
- (id)init
{
    self = [super init];
    if (self) {
        cont = 0;
    }
    return self;
}

-(id)initWithCount:(int)value {

    if (self = [super init]) {

        [self setCount:value];
    }

    return self;

}

- (void)setCount:(int)value {

    cont = value;
}

- (void)inc {

    cont++;
}

- (void)dec {

    cont--;
}

-(int)getCount {

    return cont;
}
```

```
@end
```

Analizziamo la sintassi per la dichiarazione di una classe:

`@interface`, è una parola chiave che permette di definire una classe;

`Counter: NSObject`, Counter rappresenta il nome che daremo alla nostra classe e i due punti (:) indicano che quello che segue sarà la nostra super classe, ossia quella da cui ereditiamo tutto. In Objective-C NSObject rappresenta la classe da cui discendono tutte le altre classi, questo concetto verrà spiegato meglio nel paragrafo successivo. Per fare un'analogia con il linguaggio Java, NSObject rappresenta la classe Object nel linguaggio Java;

`@end`, non è altro che un terminatore che indica al compilatore che la dichiarazione della classe è conclusa.

Quindi la realizzazione di una classe in Objective-C è molto più articolata rispetto a quella per Java e necessita di parole chiave differenti, come per esempio il prefisso `@`, seguito da `interface` e dal nome della classe, questa notazione rispetta pienamente le caratteristiche dell'OOP, perché come avevamo detto l'interfaccia è l'insieme dei metodi dell'oggetto, e dei campi definiti anche data member.

4 Ereditarietà

L'ereditarietà è uno dei concetti fondamentali nel paradigma di programmazione ad oggetti. Essa consiste in una relazione che il linguaggio di programmazione, o il programmatore stesso, stabilisce tra due classi. Per esempio se la classe B eredita dalla classe A, allora si dice che B è una sottoclasse di A e che A è una superclasse di B. In generale l'uso dell'ereditarietà dà luogo a una gerarchia di classi; nei linguaggi con ereditarietà singola, si ha un albero se esiste una superclasse radice unica di cui tutte le altre sono direttamente o indirettamente sottoclassi (come la classe Object nel caso di Java o la classe NSObject in Objective-C).

4.1 Ereditarietà in Java

In Java tutte le classi derivano, in maniera diretta o indiretta da una classe comune: Object. Come ho scritto nell'introduzione di questo paragrafo Java ha una ereditarietà singola, quindi può ereditare solamente da un'altra classe, per indicare che una classe eredita da un'altra classe si usa la parola chiave **extends**, proviamo ora ad estendere la classe Counter che ho scritto precedentemente dichiarando una nuova classe BCounter:

```
public class BCounter extends Counter {
  ...
}
```

in questo modo la classe BCounter potrà accedere alle variabili e ai metodi pubblici della classe padre tramite la keyword `super`:

```
super.inc();
```

quindi ho richiamato il metodo `inc` della classe padre Counter, senza aver implementato il metodo inc all'interno della classe, ma semplicemente tramite la keyword `super` chiameremo quello della classe padre, in questo modo la classe BCounter sarà uguale alla classe Counter, e in più avrà nuovi metodi e variabili applicati in essa, quindi sarà una estensione.

4.2 Ereditarietà in Objective-C

In Objective-C tutte le classi derivano, in maniera diretta o indiretta da una classe comune: NSObject. Come ho scritto precedentemente, anche l'Objective-C, come Java ha una ereditarietà singola, quindi può ereditare solamente da un'altra classe, e per indicare che una classe eredita da un'altra classe si usa il simbolo di punteggiatura : seguito dal nome della classe padre:

```
@interface BCounter : Counter {
    ...
}
```

anche qui abbiamo creato una classe BCounter, che avrà come classe padre la classe Counter, che di conseguenza come abbiamo visto nei paragrafi precedenti avrà come classe padre la classe NSObject. E come in Java potremo chiamare i metodi e le variabili pubbliche della classe padre Counter tramite

la keyword `super` racchiusa tra le parentesi quadre con il nome del metodo che vogliamo richiamare:

```
[super inc];
```

Quindi il concetto di ereditarietà tra Java e Objective-C è molto simile.

5 Poliformismo

Il poliformismo è un'altro concetto fondamentale della programmazione ad oggetti, ed il suo significato è relativo proprio alla definizione del termine ossia avere più forme, più aspetti. Per chiarire il concetto possiamo fare un semplice esempio di applicazione del poliformismo nella vita reale: quando facciamo riferimento ad un computer probabilmente useremo lo stesso termine (computer appunto) sia per identificare un computer desktop, un portatile o un netbook. Questo tipo di generalizzazione viene effettuata in quanto gli oggetto cui abbiamo fatto riferimento sostanzialmente effettuano le stesse operazioni, ma queste operazioni vengono fatte da oggetti con forme diverse (poliformismo).

5.1 Poliformismo in Java

Innanzi tutto recuperiamo le classi precedentemente create ossia BCounter e Counter, e vediamo come è possibile creare una nuova istanza di entrambe:

```
BCounter bC = new BCounter();
Counter c = new Counter();
```

però grazie al concetto di poliformismo posso scrivere questo:

```
Counter c = new BCounter();
```

perché come abbiamo detto in precedenza la classe BCounter estende la classe Counter, quindi essendo una estentensione ha tutto quello che ha Counter più nuove strutture se implementate. Non è però possibile fare il contrario ossia:

```
BCounter bC = new Counter();
```

perché nell'implementazione di BCounter ci possono essere variabili o metodi o altre strutture che sono state aggiunte per estendere la classe che Counter

non ha. Ma questo non è tutto, il concetto di poliformismo ha altri due aspetti fondamentali l'**overload** e l'**override**. L'overload si basa sulla scrittura di più metodi (o costruttori) identificati dallo stesso nome che però hanno in ingresso parametri di tipo e numero diverso, per esempio:

```
public int somma(int x, int y) {
    return x+y;
  }
public float somma(float x, float y) {
    return x-y;
  }
```

come si può vedere i due metodi hanno lo stesso nome, però il tipo dei parametri è diverso, quindi non ci dovremo assolutamente preoccupare di effettuare controlli sui due numeri in ingresso se sono di tipo float o int (come nell'esempio) e di conseguenza richiamare il metodo appropriato per effettuare la somma, perché verranno controllati automaticamente il tipo di parametri (e il numero) che sono stati passati in ingresso e verrà chiamato il metodo appropriato.

Con il termine override si intende una vera e propria riscrittura di un certo metodo di una classe che abbiamo ereditato. Dunque necessariamente l'override implica l'ereditarietà. Per esempio proviamo a fare nella classe BCounter l'override del metodo inc visto in precedenza nella classe padre Counter, in modo che invece di incrementare di 1 il metodo incrementi di 2:

```
public void inc() {
int i = super.getValue();
i = i + 2;
super.setValue(i);
}
```

in questo modo quindi abbiamo fatto l'overriding del metodo della classe padre, possiamo però anche estendere il metodo della classe padre in modo da eseguire tutte le istruzioni all'interno del metodo della classe padre ed aggiungerne di nuove tramite la keyword `super`, per esempio:

```
public void inc() {
  super.inc();
  int i = super.getValue();
```

```
            i = i + 2;
            super.setValue(i);
}
```

quindi andremo prima ad eseguire il metodo inc della classe padre, e poi eseguiremo il codice che è stato aggiunto.

5.2 Poliformismo in Objective-C

Recuperiamo come abbiamo fatto precedentemente in Java le classi Counter e BCounter e vediamo come creare una nuova istanza di entrambe:

```
BCounter *bC = [[BCounter alloc] init];
Counter *c = [[Counter alloc] init];
```

anche l'Objective-C gode del poliformismo, infatti posso scrivere:

```
Counter *c = [[BCounter alloc] init];
```

e non posso scrivere:

```
BCounter *c = [[Counter alloc] init];
```

quindi come possiamo vedere il concetto di poliformismo non cambia, ed è uguale a come viene realizzato in Java, e valgono le stesse considerazione fatte nel paragrafo precedente per Java. Proviamo però a vedere se anche le due caratteristiche principali del poliformismo come l'overloading e l'override sono supportate anche qui. Come abbiamo visto l'overloading ci permette di avere dei metodi o costruttori con nomi uguali, ma con tipo e numero di parametri in ingresso diversi, in modo che sia il compilatore ad associare la giusta esecuzione del metodo in base ai parametri assegnati, in Objective-C questo non è possibile, infatti se proviamo a scrivere:

```
- (int)somma:(int)number1 with:(int)number2;
- (int)somma:(float)number1 with:(float)number2;
```

riceveremo dal compilatore un errore: `Duplicate declaration of method somma...` quindi non possiamo scrivere metodi che hanno lo stesso nome, anche se i parametri in ingresso sono diversi. Proviamo invece a vedere se è possibile implementare l'override, e proviamo a fare anche qui l'override del metodo inc della classe padre Counter:

```
- (void)inc{
```

```
    int i = [super getCount];

    i = i +2;

    [super setCount:i];
}
```

Quindi l'implementazione è praticamente uguale a quella che abbiamo visto in Java, e vuol dire che abbiamo il concetto di override di un metodo, possiamo anche estendere il metodo della classe padre come abbiamo visto in Java:

```
- (void)inc{

    [super inc];

    int i = [super getCount];

    i = i +2;

    [super setCount:i];
}
```

anche in questo caso l'implementazione è uguale a quanto visto per Java, quindi in Objective-C abbiamo l'overriding ma non l'overloading di un metodo o di un costruttore.

6 Definizione dei campi

I campi costituiscono la struttura dati propria della classe, struttura che tipicamente vogliamo tenere privata, quindi non accessibile dall'esterno. Questo tipo di principio è chiamato principio dell'information hiding.

6.1 I campi in Java

```
private <TipoCampo> <NomeCampo>;
```

In questo modo definiamo un campo (privato all'oggetto, non visibile dall'esterno) del nome e tipo specificato. Esempio:

```
public class Counter {
  private int cont;
  ...
}
```

Per ogni oggetto istanza di una classe i campi costituiscono lo stato dell'oggetto, per esempio nel contatore il valore cont tiene traccia del conteggio. Esso si evolve man mano che l'oggetto interagisce con il mondo esterno, ovvero ne sono invocati metodi/operazioni che ne cambiano lo stato. Il campo può essere a sua volta costituito da un insieme di oggetti, quindi può essere a sua volta un oggetto. Lo stato è **nascosto** all'utilizzatore dell'oggetto. I campi (privati) non sono mai acceduti direttamente da chi interagisce con l'oggetto a cui è completamente nascosto il modo in cui tale stato è implementato. Per esempio per incrementare il contatore il cliente del contatore invoca il metodo inc, e non agisce direttamente lui sul valore font.

In Java è inoltre possibile definire campi pubblici, ovvero accessibili direttamente anche dai clienti dell'oggetto senza invocare i metodi, e si può fare usando il descrittore `public` invece di private. Questa non è una buona prassi ingegneristica in quanto violerebbe il principio dell'information hiding, rendendo visibili all'esterno aspetti di stato/implementazioni proprie dell'oggetto.

E' possibile definire anche campi statici, ovvero campi definiti a livello di classe, che tutti gli oggetti condividono, essi realizzano una forma di condivisione dello stato: ogni oggetto della classe vede il medesimo campo e ne può variare il contenuto. Possono essere implementate nel seguente modo:

```
private static <TipoCostante> <NomeCostante> = <Valore>;
public static <TipoCostante> <NomeCostante> = <Valore>;
```

Quindi ad esempio:

```
public class Mathlib {
  private static double PI = 3.1415;
}
```

6.2 I campi in Objective-C

In Objective-C i campi costituiscono come in Java la struttura dati propria della classe, però viene creata differentemente rispetto a Java e anche qui è più complicata e ci sono diverse tipologie di campi. Possiamo avere dei normali campi come quelli che abbiamo in Java, però non c'è bisogno di specificare il tipo private, perché inserendoli in `@interface` vengono automaticamente gestiti come private, questo è un esempio:

`Counter.h`

```
@interface Counter : NSObject {
    int cont;
}
```

è possibile mettere anche il descrittore `@private`, però la variabile sarà automaticamente privata anche senza tale descrittore. Esiste però un'altro modo per creare i campi in Objective-C ed è tramite il descrittore `@property`, vediamone subito un esempio:

```
@interface Counter : NSObject {
}
@property (nonatomic, assign) int cont;
```

Così facendo la variabile sarà pubblica, però non sarà necessario creare i metodi Setter e Getter che devono essere scritti in Java per modificare la variabile, ma verrano creati automaticamente. Invece per rendere la variabile private e avere comunque i benefici di creare automaticamente i metodi Getter e Setter il campo deve essere dichiarato all'interno del file .m in questo modo:

`Counter.m`

```
@interface Counter ()

@property (nonatomic, assign) int cont;
```

```
@end

@implementation Counter

@synthesize cont;
...
```

In questo modo, il campo `cont` soddisferà il principio dell'information hiding, e in questo caso sarà possibile utilizzarle i metodi Setter e Getter solamente all'interno della classe stessa, per poter accedervi anche esternamente bisognerà comunque creare i corrispettivi metodi di accesso.

7 Definizione dei metodi

I metodi costituiscono il comportamento degli oggetti della classe. Vediamoli nel dettaglio in Java e Objective-C

7.1 Definizione dei metodi in Java

In Java possiamo definire un metodo in questo modo:

```
public <ParamRitorno> <NomeMetodo> (<ParamList>) {
  ...
}
```

Ecco un esempio:

```
public class Counter {
  ...
  public void inc() {
    cont++;
  }
  ...
}
```

è possibile definire anche metodi **privati**, ovvero metodi intoccabili dall'esterno. La definizione è analoga:

```
private <ParamRitorno> <NomeMetodo> (<ParamList>) {
  ...
```

```
}
```

7.2 Definizione dei metodi in Objective-C

Anche i metodi in Objective-C vengono creati in maniera diversa rispetto a quelli in Java, si specifica un parametro di ritorno, il nome del metodo, e i parametri che vogliamo dare insieme alla richiesta. Vediamo nel dettaglio un esempio:

```
- (void)setCounterValue:(int)newValue;
```

- `-`, è l'identificatore del tipo di metodo che indica in questo caso che è un **metodo di istanza**, ovvero tutte le istanze della classe possono utilizzare il metodo richiamando il metodo utilizzando il nome dell'oggetto. E' possibile anche definire un metodo utilizzando come identificatore il `+` ed è un metodo che viene definito **metodo di classe**, ovvero verrà richiamato utilizzando il nome della classe;
- `void`, è il tipo di ritorno del metodo, in questo caso void sta a indicare che non abbiamo parametri di ritorno;
- `setCounterValue`, è il nome che identifica il metodo;
- `:(int)newValue`, indica che il metodo possiede un parametro in ingresso di tipo int che si chiama newValue, ed è possibile aggiungere parametri al metodo semplicemente accodandoli al primo con la sintassi :(tipoParam)nomeParam per esempio:

  ```
  - (void)SetCounter:(int)newValue withOtherParam:(int)
      newValue2;
  ```

Anche in Objective-C è possibile dichiarare un metodo come pubblico o privato, nel caso in cui si voglia dichiarare il metodo come pubblico va inserito nell'header, il file.h come abbiamo fatta precedentemente con l'esempio del contatore:

```
@interface Counter : NSObject {
    int cont;
}
```

```
- (id)initWithCount:(int)value;
- (void)setCount:(int)value;
- (void)inc;
- (int)getCount;

@end
```

In questo caso tutti i metodi sono dichiarati come pubblici, e quindi è possibile accedervi dall'esterno, se invece vogliamo dichiararli come privati, basta inserirli solo nel source file .m e non nel file.h. Anche qui ci sono due possibilità, per una maggiore leggibilità del codice è possibile inserirli come abbiamo spiegato precedentemente per le `property`, ecco un esempio:

```
#import "Counter.h"

@interface Counter ()

- (id)initWithCount:(int)value;
- (void)setCount:(int)value;
- (void)inc;
- (int)getCount;

@end

@implementation Counter

...
```

In questo caso i metodi oltre a essere implementati nel file .m vengono anche dichiarati all'interno dell'`@interface` posto dentro al file .m, questo porta a una maggiore pulizia e chiarezza del codice. Esiste anche una seconda possibilità in cui basta semplicemente implementarli dentro al file .m senza dichiararli all'interno di `@interface`, però come ho detto porta ad avere un codice non ordinato.

8 Definizione dei costruttori

Un costruttore è una sorta di "metodo speciale", invocato automaticamente all'atto della creazione dell'oggetto, per poterne inizializzare lo stato.

8.1 Costruttori in Java

Il costruttore viene invocato con l'operatore `new`, che alloca un nuovo spazio di memoria all'oggetto e quindi invoca il costruttore specificato. Tornando all'esempio del contatore possiamo avere due tipi di costruttori:

```
Counter c = new Counter();  //Viene creato e inizializzato un
    contatore con valore cont = 0
Counter c = new Counter(3); //Viene creato e inizializzato un
    contatore con un valore cont = 3
```

Se non si definiscono esplicitamente i costruttori, in una classe viene definito automaticamente il costruttore di default, che non ha parametri e ha corpo vuoto. Per la medesima classe, è possibile inizializzare un oggetto, e definire molteplici costruttori che rappresentano diversi modi di inizializzare un oggetto. I costruttori sono definiti come metodi con lo stesso nome della classe e senza parametri di ritorno. Anch'essi possono essere pubblici o privati.

```
public <NomeClasse> (<ParamList>) {...}
private <NomeClasse> (<ParamList>) {...}
```

Esempio:

```
  public class Counter {
    private int cont;

    public Counter () {
    }

    public Counter(int v) {
      count = v;
    }
    ...
  }
```

8.2 Costruttori in Objective-C

Come in Java se non si definiscono esplicitamente i costruttori di una classe viene automaticamente definito il costruttore di default, e può essere chiamato nel seguente modo:

```
Counter *c = [[Counter alloc] init];
```

Quindi per creare un oggetto manderemo un messaggio `alloc` alla classe, che in risposta creerà un oggetto in memoria e ci darà il puntatore di ritorno a quell'oggetto, infine salveremo il puntatore di ritorno nella variabile c. Però il solo messaggio `alloc` non basta, questa nuova istanza di classe non sarà attiva fino a che non manderemo anche il messaggio `init`, perché un oggetto deve essere allocato e inizializzato prima di essere usato, quindi per la creazione di un oggetto si combinano sempre questi due messaggi. Se vogliamo creare un costruttore personalizzato usando il costruttore init (che è il costruttore di default) possiamo aggiungere le modifiche al metodo in questo modo:

```
- (id)init
{
    self = [super init];
    if (self) {
        cont = 0;
    }
    return self;
}
```

Come si può vedere è stato modificato il costruttore di defualt init ed è stato fatto in modo che all'inizializzazione dell'oggetto il valore cont sia uguale a zero. Anche in Objective-C si possono creare molteplici costruttori che rappresentano modi diversi per inizializzare un oggetto, e si può fare in questo modo:

```
-(id)initWithCount:(int)value {

    if (self = [super init]) {

        [self setCount:value];
    }

    return self;

}
```

così facendo possiamo aggiungere quanti parametri in ingresso vogliamo e inizializzare un nuovo oggetto secondo i nostri scopi. Per chiamare poi un costruttore simile dobbiamo fare:

```
Counter *c = [Counter alloc] initWithCount:5];
```

9 Inner Class

Nei linguaggi di programmazione è possibile definire classi all'interno di classi, la visibilià di queste è limitata alla classe stessa. Si utilizza questa prassi quando occorre definire classi ausiliarie ad una specifica classe, la cui presenza deve essere ignorata dalle altre classi.

9.1 Inner Class in Java

Ci sono vari modi per definire classi interne, l'idioma classico prevede la definizione delle classi interne come `static private`:

```
public class A {
  <Campi>
  <Costruttori>
  <Metodi>

  static private class B {
  ...
  }
  ...
}
```

La classe B è visibile solo dalla classe A, quindi nella definizione dei campi e nel corpo dei metodi. E' possibile definire più classi nel medesimo sorgente Java, delle quali una sola deve essere dichiarata pubblica.

9.2 Inner Class in Objective-C

In Objective-C non c'è un vero e proprio concetto di inner class, è comunque possibile poter avere un comportamento simile nel seguente modo:

`ClassA.h`

```
@interface ClassA : NSObject {
}
@end

@interface ClassB: NSObject{
}
@end
```

ClassA.m

```
@implementation ClassA

@end

@implementation ClassB

@end
```

In questo modo posso creare una classe dentro un'altra classe, ClassB può essere vista solamente da ClassA, e non anche esternamente. Però questa procedura è sconsigliata in quanto è meglio che ogni classe sia dichiarata in file diversi, è possibile anche vedere una implementazione di questa procedura negli esempi nel capitolo 3.

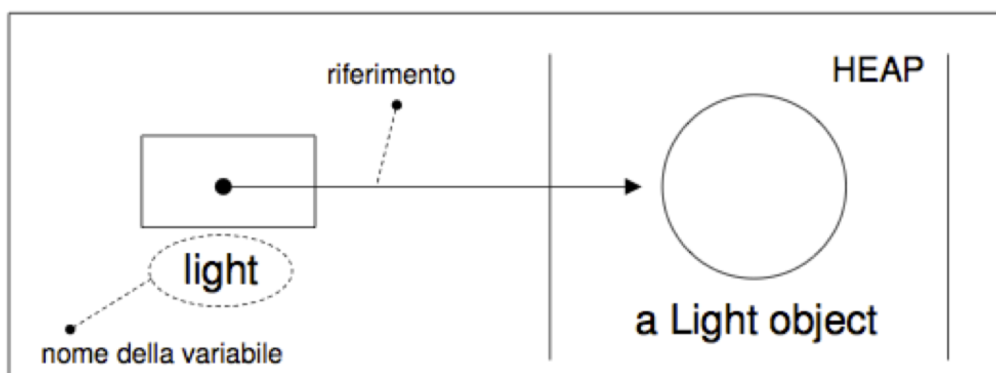
10 Gestione della memoria

Vediamo ora nel dettaglio come viene gestita la memoria nei due diversi linguaggi di programmazione.

10.1 Gestione della memoria in Java

Ogni oggetto creato occupa una certa quantità di memoria, necessaria per mantenere fisicamente i suoi dati. In Java tutti gli oggetti sono allocati nella memoria dinamica chiamata **heap**, che fa parte della JVM.

```
Light light = new Light();
```



In Java **non** esiste modo per deallocare/distruggere esplicitamente un oggetto creato dinamicamente con una new. Il recupero della memoria relativa ad oggetti non più in uso viene fatto automaticamente dalla macchina (JVM), in modo trasparente ai programmi in esecuzione (e al programmatore). In particolare il componente della macchina virtuale responsabile di riciclare spazio di memoria (dinamica) prende il nome di **garbage collector** (dove garbage significa spazzatura). In generale un oggetto è ritenuto spazzatura dal momento in cui non è più accessibile da alcun riferimento all'interno del sistema. La gestione della memoria della JVM è tale da tener traccia del numero di riferimenti per ogni oggetto creato e quando tale numero diventa zero, l'oggetto viene deallocato e la memoria occupata liberata.
Ecco alcuni esempi:

```
Counter c1 = new  Counter();
Counter c2 = new Counter();
c2 = c1;
```

- Un altro esempio:

```
Counter c = new  Counter();
c = null;
```

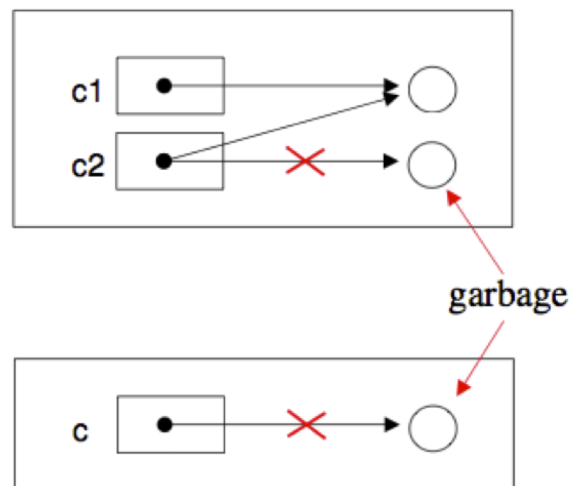


Figura 2.1: Gestione Memoria in Java.

I riferimenti sono il mezzo per interagire con gli oggetti e possono essere contenuti in variabili. Per ciò che concerne tali variabili distinguiamo la dichiarazione della variabile atta a contenere un riferimento ad un oggetto e la sua inizializzazione in cui alla variabile viene assegnato il riferimento ad un oggetto concreto. Esempio:

```
Counter c;  //Dichiarazione della variabile
c = new Counter();  //Inizializzazione della variabile a cui
        // viene assegnato il riferimento
        //dell'oggetto creato con l'istruzione new
```

La dichiarazione di una variabile non implica la creazione dell'oggetto ma solo l'allocazione dello spazio necessario per contenere il riferimento all'oggetto.

10.2 Gestione della memoria in Objective-C

Anche in Objective-C tutti gli oggetti sono allocati nella memoria chiamata **heap**, ma a differenza di Java esiste un modo per deallocare/distruggere esplicitamente un oggetto creato dinamicamente. Purtroppo questa funzione non è proprio un punto a favore per l'Objective-C ma per molti sviluppatori, soprattuto quelli alle prime armi è un grosso problema, perché sta allo sviluppaore decidere quando deallocare un'oggetto, mentre in Java questo compito è affidato al **garbage collector** che decide quando un oggetto non è più in uso. Si potrebbe anche pensare che non serva deallocare/disruggere un oggetto (se non serve più si abbandona e si continua a crearne altri) ma invece la gestione della memoria è un argomento molto importante, soprattuto quando si sviluppa per dispositivi mobili dove la quantità di memoria è molto limitata, rispetto alla programmazione per altre piattaforme.

Fortunatamente l'Apple qualche anno fa ha introdotto un nuovo modo di gestione della memoria, inserendo una sorta di **garbage collector**, anche se funziona in modo diverso, chiamato **ARC, Automatic Reference Counting**.

Innanzi tutto distinguiamo la dichiarazione della variabile e la sua inizializzazione in Objective-C, un oggetto viene dichiarato e inizializzato nel seguente modo:

```
Counter *c;  //Dichiarazione
```

```
c = [[Counter alloc] init];   //Allocazione e
    inizializzazione
```

Tramite l'allocazione viene allocato un pezzo di memoria nell'heap e successivamente viene chiamato il metodo init per poter inizializzare l'oggetto. Come già detto in precedenza sfortunatamente la memoria non è infinita, quindi è importante distruggere gli oggetti di cui non abbiamo più bisogno per liberare memoria, ma è anche importante non distruggere oggetti di cui abbiamo ancora bisogno. L'idea di proprietà di un oggetto ci può aiutare a determinare quando deve essere distrutto oppure no. Ecco un semplice esempio:

```
Counter *c = [[Counter alloc] init];

int i = [c getCount];
...
//continuiamo ad usare l'oggetto,
//e quando non ci serve più possiamo fare:

[c release]; //rilasciamo l'oggetto
```

In questo modo abbiamo rilasciato la variabile e liberato la memoria. Questa gestione della memoria però ha reso la vita difficile a molti sviluppatori, perché per esempio se lo sviluppatore dopo l'istruzione `release` prova a riutilizzare la variabile c, allora riceverà un errore sulla gestione della memoria e il programma andrà in crash. Perciò l'Apple ha deciso di introdurre l'**ARC** per cercare di migliorare la situazione. Con questa nuova funzione è il compilatore che prende in carico il problema di gestione della memoria, e sarà lui che inserirà in maniera silente e a noi nascosta gli opportuni `release` o `retain` quando lo riterrà opportuno, ossia quando nel codice non usiamo più quell'oggetto, per esempio:

```
- (void)myMethod {
  Counter *c = [[Counter alloc] init];

  [c inc];
```

```
  int i = [c getCount];

  //verrà inserito automaticamente qui

  int a = 3;
  int b = 5;
  int s = a + b;

} //qui il metodo è finito e in pratica l'ARC
//automaticamente inserirà senza mostrarlo
// all'utente [c release] dove ho indicato nel
//codice, perchè da quel momento in poi la
// variaibile c non viene più usato e quindi
//può essere deallocata.
```

Questa nuova funzione ha semplificato il lavoro agli sviluppatori che non si dovranno più preoccupare della gestione della memoria, ma potranno affidarsi all'ARC che prenderà in carico il lavoro sporco. Apple ha però lasciato la possibilità agli sviluppatori di decidere se poter usare l'ARC oppure no nel loro progetto, o addirittura è possibile anche solo disabilitare l'ARC solo per alcuni file aggiungendo questo flag: `-fno-objc-arc` al compilatore per lo specifico file, in questo modo è possibile continuare ad usare vecchie librerie di terze parti che non supportano e non supporteranno mai l'ARC.

11 Invocazione dei metodi

L'invocazione dei metodi è uno degli aspetti basilari nella programmazione ad oggetti, perché come abbiamo detto più volte, è il momento in cui c'è un reciproco scambio di messaggi tra oggetti, per esempio invio di una richiesta di invocazione di un metodo e invio di un parametro di ritorno quando è disponibile, vediamo nel dettaglio come i due linguaggi eseguono queste operazioni.

11.1 Invocazione dei metodi in Java

L'invocazione di metodi avviene utilizzando il riferimento, che funge da identificatore dell'oggetto su cui invocare il metodo (inviare il messaggio). In Java per invocare un metodo si utilizza la **notazione puntata**, la cui sintassi è:

```
<RiferimentoOggetto>.<NomeMetodo>(<ParametriAttuali>)
```

Esempio:

```
Counter c = new Counter();
//Invocazione metodo
c.setValue(10);
c.inc();
```

Inoltre i metodi in generale possono restituire un valore come risultato dell'operazione. Tale risultato può essere un valore primitivo o il riferimento ad un oggetto stesso. Esempio:

```
Counter c = new Counter();
c.inc();
int newVal = c.getValue();
```

Possiamo anche definire la proprietà di **overloading** (già descritte in precedenza), ossia avere più costruttore e più metodi con lo stesso nome. Esempio:

```
public void inc() {
  cont++:
}

public void inc (int delta) {
  cont+=delta;
}
```

Quindi i metodi devono essere distinguibili per il tipo di parametri forniti in ingresso. Nel caso ci sia ambiguità sarà il compilatore a segnalare il problema.

11.2 Invocazione dei metodi in Objective-C

Come in Java l'invocazione del metodo avviene utilizzando il riferimento che funge da identificatore dell'oggetto su cui invocare il metodo, però diversamente da Java per invocare un metodo la sintassi consigliata è questa:

```
[<Riferimento Oggetto> <NomeMetodo>:<Parametri>];
```

In pratica viene messo tra parentesi quadre il nome del metodo da invocare, all'inizio verrà inserito il riferimento all'oggetto di cui vogliamo chiamare il metodo e poi il nome del metodo e i parametri che diamo in ingresso se sono disponibili.
Esempio:

```
Counter *c = [[Counter alloc] init];

[c inc];

[c setValue:10];
```

Come in Java anche qui i metodi possono restituire un valore come risultato dell'operazione, un esempio:

```
Counter *c = [[Counter alloc] init];

int v = [c getValue];
```

12 Categories

In Objective-C possiamo trovare una funzione estranea a Java che sono le Categories. Con una Category è possibile dare funzioni aggiuntive a una classe già esistente nella libreria senza modificare nient'altro. Per esempio nel Foundation Framework, possiamo trovare la classe NSString, che ci permette di creare e gestire le stringhe, e ci sono moltissime funzioni già pronte per la gestione di queste stringhe, ma se uno sviluppatore ha bisogno di qualcosa di più specifico per il proprio progetto basta realizzare una Category per la classe NSString, ciò ci permette di estendere la classe senza fare nessuno sforzo. Ecco un esempio:

`NSString+MyApp.h`

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

@interface NSString (AmazingApp)

- (BOOL)pariodispari;

@end
```

NSString+MyApp.m

```
#import "NSString+AmazingApp.h"

@implementation NSString (AmazingApp)

- (BOOL)pariodispari{
    //YES è pari
    //NO è dispari
    if ([self length] % 2 == 0) {
        return YES;
    }

    return NO;

}
@end
```

In pratica con questa estensione della classe NSString potremo riconoscere quando il numero di caratteri di una stringa è pari o dispari e potremo usarla in qualsiasi altra classe come se fosse una funzione propria della classe NSString, semplicemente in questo modo:

```
#import "NSString+AmazingApp.h"

...

NSString *prova = @"Ciao";
BOOL result = [prova pariodispari];
```

In questo modo abbiamo esteso la nostra classe senza nessuno sforzo, in Java questo non è possibile, è possibile estendere le classi normali create dallo sviluppatore che fanno parte del progetto, ma non è possibile farlo con la semplicità che offrono le Categories di Objective-C e soprattutto non è possibile farlo con le classi che fanno parte della libreria di Java.

13 Definizione di Array

Un array è una struttura dati complessa usata in molti linguaggi di programmazione. Si può immaginare un array come una sorta di casellario, le cui caselle sono dette celle dell'array, ciascuna cella si comporta come una variabile tradizionale, e in Java tutte le celle sono variabili di uno stesso tipo preesistente, detto tipo base dell'array, invece in Objective-C un array può contenere anche oggetti di tipi diversi.
Ciascuna cella dell'array è identificata da un valore di **indice**. L'indice è generalmente numerico e i valori che gli indici possono assumere sono numeri interi contigui che partono da 0. Quindi la casella di indice 0 sarà la prima casella , quella di indice 1 la seconda e così via fino a N caselle.

13.1 Definzione di Array in Java

Un Array viene dichiarato in Java con una sintassi come la seguente:

```
int[] unArray;
```

Questa riga dichiara un array di interi: al tipo degli elementi (che in questo caso è un int) vengono fatte seguire le parentesi quadre che specificano appunto che è un array. Le parentesi possono anche essere collocate dopo l'identificatore (cioè dopo il nome della variabile):

```
int unArray[];
```

Queste due dichiarazioni comportano solo l'allocazione della variabile, ma non di un array vero e proprio, un array viene creato usando l'operatore new, che colloca l'array in memoria e restituisce il riferimento nella variabile unArray:

```
unArray = new int[15];
```

L'istruzione new int[15] crea un array di 45 celle ciascuna di tipo intero. Per accedere alla cella di un array, come in altri linguaggi si utilizza una sintassi

che consente di specificare l'indice di tale cella. Gli indici, come detto in precedenza, sono numeri interi consecutivi a partire da 0 (prima cella), quindi ad esempio per assegnare un valore alle prime due celle si usa la seguente istruzione:

```
unArray[0] = 1;
unArray[1] = 1;
```

In questo caso abbiamo assegnato alle prime due celle dell'array il valore 1. Ci sono molti altri modi per poter veder quanti e quali elementi ha un array, questo procedimento è chiamato iterazione, e serve per scorrere ogni elemento dell'array, uno dei modi per farlo è il seguente:

```
for (int i = 0; i< unArray.length; i++) {

  System.out.println(anArray[i]);
}
```

In Java è possibile creare array di qualsiasi altri tipo, inclusi ovviamente i tipi di riferimento (quindi le classe, le interfacce e perfino gli array stessi). Di seguito riporto il frammento di codice che dichiara e crea un array con tipo base String:

```
String altroArray[] = new String[50];
```

Il funzionamento dell'array è identico a quello descritto precedentemente, però le celle conterranno elementi String invece che int. Il problema nell'usare gli array è che non possono cambiare la propria dimensione, il numero di elementi contenuti viene stabilito al momento della creazione e rimane immutato. Per superare questa limitazione Java mette a disposizione la classe ArrayList, contenuta nel package java.util che permette di rappresentare sequenze di oggetti di lunghezza variabile, quindi le due differenze rispetto a un normale array sono la dimensione variabile durante l'esecuzione del programma e che gli elementi contenuti sono solo di tipo Object, quindi non

potrà contenere un elemento int, ma il suo corrispettivo oggetto ossia Integer. Inoltre cambiano anche le istruzioni per accedere e scrivere in questo tipo di Array. Un esempio per usarlo è il seguente:

```
ArrayList<String> myArr = new ArrayList<String>
```

Viene creato un ArrayList che può contenere elementi di tipo String e quindi in un array posso inserire solamente il tipo di oggetti che viene dichiarato nella creazione dell'array, e non posso inserire tipi di oggetti misti, per esempio un Integer, una String e così via, ma in questo caso solo String. Per poter aggiungere un elemento nell'array è possibile farlo con questa istruzione:

```
myArr.add(''Hello'');
```

Si può accedere a un elemento dell'array in questo modo:

```
String myString = myArr.get(0);
```

Quindi recupero con il comando `get()` l'elemento all'indice 0. Inoltre è anche possibile eliminare un determinato elemento nell'array tramite questo comando:

```
myArr.remove(<index>);
```

così facendo andremo a rimuovere un elemento alla specifica posizione index, ed è possibile anche rimuovere uno specifico elemento:

```
myArr.remove(<Object>);
```

in questo modo andremo a rimuovere la prima occorrenza di quello specifico oggetto all'interno dell'Array. Quindi un ArrayList è un elemento molto più dinamico e utile rispetto a un semplice Array.

13.2 Definzione di Array in Objective-C

Anche in Objective-C come in Java è possibile avere un array di soli int, viene dichiarato ed inizizializzato in questo modo:

```
NSInteger myIntegerArr[40];
```

Si può notare che la dichiarazione è molto simile a quella vista in Java, però leggermente differente in quanto in questo modo l'array viene già creato ed è già possibile accedere o modificare il contenuto delle celle. Per fare ciò è possibile ,come lo è in Java ,usare questa istruzione:

```
myInteger[0] = 10;
```

Quindi andremo ad attribuire il valore 10 alla cella di indice 0. E' possibile come in Java creare array di qualsiasi altro tipo, inclusi i tipi di riferimento. Di seguito riportiamo il frammento di codice che dichiara e crea un array di tipo String:

```
NSString *arr[10];
```

possiamo accedere e scrivere in una cella con questa istruzione:

```
arr[0] = @"Hello";
```

Però in Objective-C possiamo creare un'array in modo diverso e molto più utile di questo, e lo si fa usando le classi NSArray e NSMutableArray, queste due classi sono simili agli ArrayList di Java, l'unica differenza è che in un NSArray non è possibile cambiare la dimensione dell'array una volta creato, cosa che è possibile fare in un NSMutableArray e inoltre entrambi possono contenere solo oggetti e non tipi primitivi quali int o boolean. Bisogna però fare un'osservazione riguardo gli oggetti che è possibile mettere in questi array, in Java abbiamo visto che bisogna specificare il tipo di ArrayList ossia bisogna specificare che tipo di oggetti l'array può contenere, e non può contenere oggetti di tipo diversi, invece in Objective-C non è così, in un array possiamo inserire NSString, NSNumber ecc... senza avere nessun tipo di problema. Vediamo nel dettaglio come funzionano questi due tipi di Array. E' possibile creare e inizializzare un NSArray nel seguente modo:

```
NSArray *arr = [NSArray arrayWithObject:@"Uno",@"Due",@"Tre,
    @"Quattro",nil];
```

come è possibile vedere quando si crea un NSArray bisogna anche specificare gli elementi che deve contenere, perché come detto in precedenza è un Array immutabile, quindi una volta creato non è possibile eliminare o aggiungere nuove celle. E' da notare anche che l'ultimo elemento che abbiamo aggiunto all'array è `nil`, questo elemento serve per far capire al compilatore che non ci sono più elementi dopo il valore `nil` e che quindi l'ultimo elemento dell'array è la stringa con il valore Quattro. Ovviamente anche con questo tipo di array è possibile accedere alle celle esistenti (ovviamente non si può aggiungere, modificare o eliminare) ed è possibile farlo con la seguente istruzione:

```
NSString *myString = [arr objectAtIndex:0];
```

Quindi con il comando:

```
[arr objectAtIndex:<index>];
```

andremo a recuperare il valore dell'oggetto all'indice 0 e restituiremo il riferimento a una nuova variabile.
La classe che più si avvicina a un ArrayList in Java, è un NSMutableArray, con questo tipo di array abbiamo molta più libertà perché sarà dinamico, quindi potremo aggiungere modificare eliminare le celle del nostro array, inoltre sarà possibile interagire con esso in moltissimi modi, vediamo i modi principali con cui è possibile creare un nuovo array:

```
NSMutableArray *arr = [[NSMutableArray alloc]
   initWithCapacity:30];
```

Questo modo è quello che più ricorda la creazione di array con tipi primitivi, è possibile creare ed inizializzare un array con una capacità (in questo caso 30), però il nostro array sarà ancora vuoto, l'utilità di questa istruzione è quella di permettere al compilatore di creare già uno spazio atto a contenere questa grandezza di array, anche se poi potrebbe variare in futuro. Un'altro modo per poter creare un NSMutableArray è il seguente:

```
NSMutableArray *arr = [[NSMutableArray alloc] init];
```

In questo modo abbiamo creato ed inizializzato un nuovo array, è possibile anche creare un NSMutableArray come si fa già per un NSArray, quindi:

```
NSMutableArray *arr = [[NSMutableArray alloc] initWithObjects
   :@"Uno", @"Due", nil]];
```

Quindi così facendo abbiamo creato un array dinamico con già degli elementi iniziali, infine è possibile creare un NSMutableArray partendo da un array:

```
NSArray *arr = [NSArray arrayWithObject:@"Uno",@"Due",@"Tre,
    @"Quattro",nil];
NSMutableArray *arr2 = [[NSMutableArray alloc] initWithArray:
    arr];
```

In questo caso ho passato il riferimento di un NSArray per creare un NSMutableArray, che verrà creato e inizializzato con gli elementi dell'array `arr`. L'accesso a una determinata cella viene fatto nello stesso modo degli NSArray quindi:

```
[arr objectAtIndex:<index>];
```

però negli NSMutableArray abbiamo anche le funzioni per poter eliminare o modificare un elemento in una certa posizione, per esempio:

```
[arr setObject:<Oggetto> atIndexedSubscript:<index>];
```

in questo modo andremo a modificare un certo oggetto in una certa posizione, invece:

```
[arr removeObjectAtIndex:<index>];
```

andremo ad eliminare un oggetto in una certa posizione, oppure un'altro metodo di eliminazione molto utile è il seguente:

```
[arr removeObject:<Oggetto>];
```

In questo modo andremo ad eliminare tutte le occorrenze dell'oggetto all'interno dell'array, questo comando è simile a quello di Java con gli ArrayList, `remove()`, solo che in Java eliminiamo la prima occorrenza, mentre qui in Objective-C eliminiamo tutte le occorrenze di quell'oggetto.
Concludendo quindi possiamo equiparare gli ArrayList in Java, con l'NSMutableArray in Objective-C.

14 Delegate

Il Delegate è uno strumento molto importante che offre l'Objective-C, con il quale è possibile far comunicare due o più classi tra di loro in maniera autonoma, proverò a spiegare il suo funzionamento con un esempio: proviamo

a creare un delegate nella classe Counter che abbiamo visto all'inizio di questo capitolo, in modo da notificare tutte le classi che sono registrate a questo delegate quando per esempio eseguiamo il metodo inc, vediamo come fare:

`Counter.h`

```
@class Counter;

@protocol CounterDelegate <NSObject>

- (void)incMethodCalledWithInt:(int)count;

@end

@interface Counter : NSObject {
    int cont;
}

@property (nonatomic, weak) id <CounterDelegate> delegate;

- (id)initWithCount:(int)value;
- (void)setCount:(int)value;
- (void)inc;
...

@end
```

in pratica tramite la keyword `@protocol` dichiaro che esiste un protocollo chiamato `CounterDelegate` che in questo caso ha il metodo `incMethodCalledWithInt` (ma se ne possono avere tanti altri), quindi ogni qual volta noi eseguiremo il metodo inc, verrà chiamato questo metodo, e tutte le classi che sono registrate a questo delegate verrano avvisate ed eseguiranno lo specifico metodo. Inoltre nell'interfaccia della nostra classe abbiamo creato l'oggetto relativo al `CounterDelegate` ossia `id <CounterDelegate> delegate` che si occuperà di spedire messaggi alle classi registrate, vediamo come diventerà il metodo inc:

`Counter.m`

```
- (void)inc {

    cont++;
    [self.delegate incMethodCalledWithInt:cont];

}
```

come si può vedere tramite l'istruzione `self.delegate` e il nome del metodo noi spediremo a tutte le classi registrate il nuovo valore del conteggio, proviamo ora a vedere come fare per registrarsi a questo delegate. Proviamo a creare una nuova classe in questo modo:

`NuovaClasse.h`

```
@interface NuovaClasse : NSObject <CounterDelegate> {

}

@end
```

come si può vedere la nostra nuova classe sarà predisposta per potersi registrare al `CounterDelegate`, grazie all'istruzione che abbiamo inserito all'interno dei simboli `<...>`, ma non abbiamo finito, perché dobbiamo anche dichiararlo esplicitamente nel nostro file .m

`NuovaClasse.m`

```
...
Counter *c = [[Counter alloc] init];
c.delegate = self;
...
```

in questo modo abbiamo creato una nuova istanza della classe Counter, e abbiamo registrato noi stessi ossia la classe NuovaClasse al delegate del Counter tramite l'istruzione `c.delegate = self`, infine il compilatore ci avviserà che non abbiamo finito, perché dobbiamo implementare il metodo che offre il delegate a cui ci siamo registrati quindi dovremo fare:

```
- (void)incMethodCalledWithInt:cont {

  //eseguiamo il codice che vogliamo
}
```

così facendo ogni qual volta verrà eseguito il metodo`inc` in qualunque altra classe, il `CounterDelegate` spedirà un messaggio a tutte le classi registrate, che eseguiranno il corrispettivo metodo, in questo caso `incMethodCalledWithInt`, questo strumento che offre l'Objective-C è molto utile agli sviluppatori perché per esempio gli permette di poter eseguire codice in altre classi ed aspettare una risposta a lavoro finito.

15 Socket

Una Socket è uno strumento che permette di trasmettere e ricevere in modo bidirezionale attraverso la rete. E' il punto in cui il codice applicativo di un processo accede al canale di comunicazione per mezzo di una porta, ottenendo una comunicazione tra processi che lavorano su due macchine fisicamente separate. Dal punto di vista del programmatore una socket è un particolare oggetto sul quale leggere e scrivere i dati da trasmettere o ricevere. Ogni lato è identificato da una combinazione di due elementi, l'indirizzo IP e la porta. Il primo identifica un computer e la seconda è connessa a un processo. Ci sono differenti tipi di socket, la differenza sta nel modo in cui vengono trasmessi i dati (protocolli), i più popolari sono TCP e UPD.

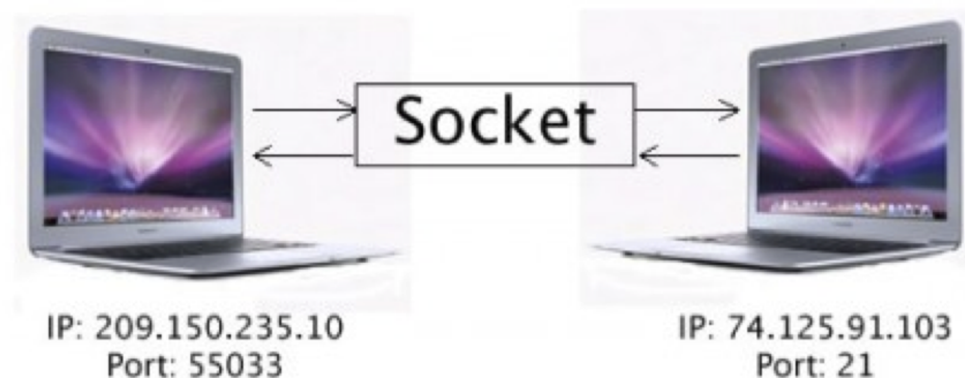


Figura 2.2: Socket.

15.1 Socket in Objective-C

Per stabilire una connessione socket nei dispositivi portatili che usano l'Objective-C quindi (iPhone, iPad, iPod Touch) usiamo gli stream. Uno stream può essere considerato un flusso di dati, quindi lo possiamo immaginare come un canale in cui vengono trasmessi i dati. Questi dati possono essere contenuti in oggetti differenti, come un file, un buffer C, o una connessione di rete. In Objective-c possiamo usare le CFStream API per stabilire una connessione socket, che creerà un oggetto stream come risultato il quale verrà usato per ricevere e spedire dati da un host remoto. Oltre al CFStream useremo anche la classe NSStream perché da sola non può collegarsi a un host remoto, ma ha appunto bisogno delle API CFStream per poterlo fare e si può trarre vantaggio dal fatto che queste due classi sono simili tra di loro, per poter fare un cast da un oggetto CFStream ad uno NSStream. Quindi per poter effettuare un semplice collegamento a un host remoto possiamo fare questo:

```
NSInputStream *inputStream;
NSOutputStream *outputStream;
CFReadStreamRef readStream;
CFWriteStreamRef writeStream;
CFStreamCreatePairWithSocketToHost(NULL,(CFStringRef)@"
    localhost", 80, &readStream, &writeStream);
inputStream = (NSInputStream *)readStream;
outputStream = (NSOutputStream *)writeStream;
```

Abbiamo creato un NSInputStream per leggere l'input e un NSOutputStream per scrivere in output dopo di che abbiamo creato il CFReadStreamRef e il CFWriteStreamRef che sono i corrispettivi oggetti della classe CFStream di cui ho parlato in precedenza, e di cui possiamo fare il cast. E' bastato poi usare il metodo `CFStreamCreatePairWithSocketToHost` fornendogli in ingresso un indirizzo IP e la porta dell'host per ricevere di ritorno il riferimento a un CFReadStreamRef e un CFWriteStreamRef tramite queste istruzioni `&readStream` e `&writeStream` in questo modo tramite la keyword `&` andremo ad inserire il riferimento di ritorno dal metodo dentro le nostre variabili che abbiamo creato. Infine assegneremo i riferimenti delle variabili CFStream, alle variabili NSStream grazie ad un casting, perché come ho detto in precedenza queste due classi sono simili tra di loro e sono state create per poter lavorare insieme.

Inoltre la classi che usano NSStream hanno bisogno di registrarsi all'NSStreamDelegate, come ho spiegato precedentemente un delegate è uno strumento che hanno le classi per poter scambiare messaggi tra di loro in modo autonomo, in questo caso registrandoci a questo delegate della classe NSStream potremo ricevere delle notifiche quando riceveremo messaggi dall'host remoto o in caso di errori di scrittura o di lettura. Quindi per farlo dobbiamo innanzi tutto scrivere nell'@interface dell'header file questo NSStreamDelegate tra <...>:

```
@interface MyClass : NSObject <NSStreamDelegate>
```

e poi nel source file subito il listato di codice che ho scritto sopra per il collegamento questo:

```
[inputStream setDelegate:self];
[outputStream setDelegate:self];
```

in modo da registrarsi alle notifiche dell'NSStreamDelegate. Dopo di che ci vogliono alcune istruzioni di default che ci permettono di poter effettuare la connessione:

```
[inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
   forMode:NSDefaultRunLoopMode];
[outputStream scheduleInRunLoop:[NSRunLoop currentRunLoop]
   forMode:NSDefaultRunLoopMode];
```

queste istruzioni ci permettono di poter eseguire altro codice durante la connessione, altrimenti il codice si bloccherebbe e non permetterebbe di eseguire nessun'altra istruzioni rimanendo in attesa di una risposta da parte dell'host remoto, questo ci permette di poter eseguire una interfaccia grafica o di fare altre operazioni senza mostrare all'utente blocchi durante l'esecuzione dell'applicazione, infine con queste due istruzioni:

```
[inputStream open];
[outputStream open];
```

apriamo i due flussi per poter ricevere e inviare dati all'host remoto.
Proviamo ora adesso a spedire un semplice messaggio di testo al nostro host remoto:

```
-(void)sendMessage {

NSString *msg = @"Hello World!";
```

```
NSData *data = [[NSData alloc] initWithData:[msg
    dataUsingEncoding:NSASCIIStringEncoding]];
[outputStream write:[data bytes] maxLength:[data length]];

}
```

così facendo abbiamo creato una stringa con il testo `Hello World!` e l'abbiamo codificata in un oggetto NSData, infine abbiamo spedito il messaggio con il metodo `[outputStream write....`
Se vogliamo invece ricevere i messaggi dall'host remoto dobbiamo innanzi tutto implementare il metodo di default che richiede la classe NSStreamDelegate ossia:

```
- (void)stream:(NSStream *)theStream handleEvent:
        (NSStreamEvent)streamEvent
```

come possiamo vedere in ingresso a questo metodo oltre ad un flusso NSStream riceveremo anche un `NSStreamEvent`, perchè dall'host remoto possiamo ricevere messaggi di tipo diverso, i principali sono:

- NSStreamEventOpenCompleted, la connessione è stata aperta;
- NSStreamEventHasBytesAvailable, fondamentale per ricevere i messaggi;
- NSStreamEventErrorOccured, ci permette di sapere quando avviene un problema durante la connessione;
- NSStreamEventEndEncountered, chiude lo stream quando il server si scollega.

E in base al tipo di messaggi in ingresso possiamo eseguire le istruzioni di conseguenza, in questo caso se vogliamo leggere un messaggio in ingresso dobbiamo verificare che il messaggio sia del tipo `NSStreamEventHasBytesAvailable`, vediamo nel dettaglio il listato:

```
- (void)stream:(NSStream *)theStream handleEvent:
        (NSStreamEvent)streamEvent {

  switch (streamEvent) {

    case NSStreamEventOpenCompleted:
      NSLog(@"Stream opened");
```

```
    break;

  case NSStreamEventHasBytesAvailable:
    if (theStream == inputStream) {

         uint8_t buffer[1024];
         int len;

         while ([inputStream hasBytesAvailable]) {
             len = [inputStream read:buffer
      maxLength:sizeof(buffer)];
             if (len > 0) {

                 NSString *output = [[NSString alloc]
      initWithBytes:buffer length:len
      encoding:NSASCIIStringEncoding];

                   if (nil != output) {
                     NSLog(@"Risposta Server: %@", output);
                   }
               }
           }
       }
      break;

  case NSStreamEventErrorOccurred:
    NSLog(@"Can not connect to the host!");
    break;

  case NSStreamEventEndEncountered:
    break;

  default:
    NSLog(@"Unknown event");
 }

}
```

Così facendo abbiamo letto un messaggio di tipo `NSStreamEventHasBytesAvailable` che ci è arrivato dall'host remoto, dove prepariamo un buffer di grandezza 1024, e leggiamo i byte che ci sono arrivati finchè ce ne sono, quando sono

finiti, stampiamo il risultato.

15.2 Socket in Java

Affrontiamo ora i socket nel linguaggio Java, creando anche qui una semplice connessione con un server, e uno scambio di messaggi. Il networking in Java viene gestito all'interno del package `java.net` con l'ausilio (per la gestione degli stream) delle classi definite all'interno del package `java.io` . Java implementa due tipi di protocollo a livello di trasporto, il TCP/IP e l'UDP. Quello che ha noi interessa in questo semplice esempio sono le classi riferite al TCP/IP che è il protocollo orientato alla conessione. Le classi che riguardano la gestione di questo protocollo nel nostro esempio sono:

```
java.net.Socket
```

Tramite questa classe possiamo gestire una conessione con l'oggetto Socket. Di particoalre importanza sono le due modalità di trasmissione dei dati tramite i socket, che in Java vengono gestiti come flussi di dati espressi concettualmente tramite gli oggetti di tipo `java.io` . InputStream e OutputStream, servono rispettivamente per leggere e scrivere dei dati attraverso i canali di comunicazione. I metodi necessari per utilizzare gli stream di scrittura e lettura della classe Socket sono:

```
public OutputStream getOutputStream()
public InputSteam getInputStream();
```

Vediamo la semplice implementazione simile a quella per Objective-C ossia effetuiamo una connessione all'host remoto tramite l'oggetto Socket e scambiamo dei messaggi:

```
public class Client {

private ObjectInputStream sInput;
private ObjectOutputStream sOutput;
private Socket socket;

public boolean start() {
socket = new Socket(\localhost", portnumber);

sInput = new ObjectInputStream(socket.getInputStream());
sOutput = new ObjectOutputStream(socket.getOutputStream());
```

```
//Creamo un thread per leggere i messaggi dal server
new ListenFromServer().start();
}

public void sendMessage(String msg) {

//Spedisco un messaggio
sOutput.writeObject(msg);

}

class ListenerFromServer extends Thread {

while(true) {
String msg = (String) sInput.readObject();

System.out.println(msg);

}

}

public static void main(String[] args) {

Client client = new Client();

client.start();

client.sendMessage(\Hello World!);

}
```

e così abbiamo creato una semplice connessione con un server, e ci siamo registrati per leggere e scrivere messaggi. Infine se vogliamo chiudere la connessione abbiamo dobbiamo usare questi comandi:

```
sInput.close();
sOutput.close();
socket.close();
```

Nel prossimo capitolo vedremo nel dettaglio come realizzare una chat che opera sia in Java che in Object C.

16 Thread

Nello sviluppo di software e non solo, si sente spesso parlare di operazioni concorrenti e parallele, in termini tecnici la concorrenza è una proprietà del software mentre le esecuzioni parallele sono una proprietà della macchina. Il parallelismo e la concorrenza sono due aspetti separati. Un programmatore, non può mai garantire che il codice che sviluppa possa essere eseguito su una macchina capace di eseguire operazioni in parallelo, ma può progettare il codice in modo che usi operazioni concorrenti. Prima di tutto dobbiamo definire alcuni termini:

- Task, è un semplice singolo pezzo di lavoro che deve essere eseguito;
- Thread, è un meccanismo fornito dal sistema operativo che permette che multiple operazioni vengano eseguite nello stesso momento all'interno di una singola applicazione;
- Process, è un pezzo di codice eseguibile, il quale può essere formato da thread multipli.

Figura 2.3: Socket.

Quindi come si può vedere dall'immagine un processo può contenere l'esecuzione di diversi thread, e ogni thread può eseguire più task però uno alla volta. In questo caso per esempio il Thread 2 esegue la lettura del file, mentre il Thread 1 visualizza l'interfaccia utente. Questo in generale è quello

che è sempre bene fare quando si realizzano applicazioni con una interfaccia grafica, cioè serve un Thread principale che si occupa di eseguire tutto il lavoro dell'interfaccia utente, e uno o più Thread secondari che si occupano di eseguire tutte le operazioni come leggere file, accedere alla rete e altre operazioni che se eseguite sul Thread principale andrebbero a bloccare l'interfaccia utente, e quindi l'applicazione non sarebbe reattiva ma subìrebbe grossi rallentamenti e addirittura blocchi totali. Quindi parliamo di applicazioni "multi-threading", dove il processo che eseguirà l'applicazione avrà più di un thread, assegnando ad ognuno compiti da eseguire parallelamente. D'altronde l'esecuzione parallela di più thread all'interno della stessa applicazione è vincolata all'architettura della macchina su cui gira. In altre parole, se la macchina ha un unico processore, in un determinato momento x, può essere in esecuzione un unico thread.

16.1 Thread in Java

La tecnologia Java, tramite la Java Virtual Machine, ci offre uno strato d'astrazione per poter gestire il multi-threading direttamente dal linguaggio. In Java i meccanismi della gestione dei thread, risiedono essenzialmente:

1. Nella classe Thread e l'interfaccia Runnable (package `java.lang`);
2. Nella classe Object (ovviamente package `java.lang`);
3. Nella JVM e nella keyword `syncrhonized`.

Quando si avvia una applicazione Java c'è almeno un thread in esecuzione appositamente creato nella JVM per eseguire il codice dell'applicazione. Per avere più thread basta istanziarne altri dalla classe Thread. Nel prossimo esempio noteremo che quando si istanzia un oggetto Thread, bisogna passare al costruttore un'istanza di una classe che implementa l'interfaccia Runnable. In questo modo infatti, il nuovo thread, quando sarà fatto partire (mediante la chiamata del metodo `start()`), andrà ad eseguire il codice del metodo `run()` dell'istanza associata. L'interfaccia Runnable quindi, richiede l'implementazione del solo metodo `run()` che definsice il comportamento di un thread, e l'avvio di un thread si ottiene con la chiamata del metodo `start()`.

```
public class ThreadCreation implements Runnable {
  public ThreadCreation () {
    Thread ct = Thread.currentThread();
    ct.setName(\Thread Principale");
    Thread t = new Thread(this, \Thread figlio");
    System.out.println(\thread attuale: \ +ct);
    System.outprintln(\thread creato: \ + t);
    t.start();
    try {
      Thread.sleep(3000);
    }
    catch (InterruptedException e) {
      System.out.println(\principale interrotto");
    }
    System.out.println(\uscita Thread principale");
  }
  public void run() {
  try {
    for (int i = 5; i > 0; i--) {
      System.out.println(\"+i);
      Thread.sleep(1000);
    }
  } catch (InterruptedException e) {
    System.out.println(\Thread figlio interrotto");
  }
  System.out.println(\uscita Thread figlio");
  }
  public static void main(Strings args[ ]) {
  new ThreadCreation();
  }
}
```

L'output che avremo sarà il seguente:

```
Thread attuale: Thread[Thread principale,5,main]
Thread creato: Thread[Thread figlio,5,main]
5
4
3
uscita Thread principale
2
```

```
1
uscita Thread figlio
```

Quindi in pratica abbiamo il creato un secondo Thread chiamato Thread figlio, e poi abbiamo il thread principale, non appena il Thread figlio viene lanciato, il thread principale va in `sleep` per 3 secondi e il Thread figlio continua la sua operazione, dopo di che il thread principale finisce i 3 secondi di `sleep` e termina, invece il thread figlio continua la sua computazione fino alla fine. In questo modo abbiamo due thread che svolgono in maniera concorrente le proprie istruzioni. Questa è solo una semplice implementazione di Thread in Java, ci sono svariate possibilità come la priorità dei Thread e la sincronizzazione in maniera concorrente.

16.2 Thread in Objective-C

Il linguaggio Objective-C (come il C/C++), sfrutta le complesse librerie del sistema operativo per gestire il multi-threading, in particolare sfrutta il POSIX Threads API (pthreads) che è parte del sistema operativo. In Objective-C i due più usati e semplici modi per poter eseguire operazioni concorrentemente sono:

- `GCD`, grand central dispatch è una tecnologia sviluppata da Apple, per ottimizzare l'esecuzione delle applicazioni su sistemi multi core o su altri sistemi basati sul multiprocessing. Questa implementa un parallelismo a livello di thread. GCD lavora consentendo al programmatore di delimitare specifiche porzioni di codice che possono essere eseguite in parallelo definendole come blocchi. GCD per eseguire i blocchi utilizza i thread ma al programmatore la cosa è totalmente trasparente, è un modo semplice e leggero per eseguire operazioni concorrentemente. Non bisogna preoccuparsi di niente perchè è il sistema che che si prenderà carico di tutto. Ma è usato solamente per eseguire operazioni leggere e niente di complicato (i blocchi verranno descritti nel successivo paragrafo);
- `NSOperation` e `NSOperationQueue`, è un metodo più complicato di classi per la gestione dei Thread, queste classi però offrono una soluzione

completa per eseguire operazioni concorrentemente e permette di avere sincronizzazione fra le varie operazioni, oppure priorità alle code e tante altre funzioni.

Vediamo lo stesso esempio realizzato prima in Java, ma questa volta realizzato in Objective-C prima con il metodo CGD e successivamente con il metodo NSOperation e NSOperationQueue.

Metodo CGD:

```
dispatch_queue_t backgroundQueue = dispatch_queue_create("
   BackgroundQueue", NULL);
        NSLog(@"Thread Princiaple");

        dispatch_async(backgroundQueue, ^(void) {
            NSLog(@"Thread Figlio");
            for (int i = 5; i > 0; i--) {
                NSLog(@"%d",i);
                sleep(1);
            }
            NSLog(@"Uscita Thread Figlio");

        });
        sleep(3);
        NSLog(@"Uscita Thread Principale");
```

Quindi in maniera molto semplice abbiamo aggiunto al Thread princiaple un secondo Thread con pochissime linee di codice vediamole nel dettaglio:

- `dispatch_queue_t`, con questa istruzione vado a dichiarare una nuova coda che conterrà i thread che saranno pronti per essere eseguiti;
- `dispatch_queue_create`, dopo aver dichiarato la coda con l'istruzione sopra, vado a creare effettivamente la coda, passandogli in ingresso un nome, per poterla poi riconoscere successivamente e invece il secondo parametro in ingresso che abbiamo sta a indicare il tipo di coda che vogliamo creare, se vogliamo che sia seriale, che quindi esegua un'operazione alla volta, oppure concorrente in modo da eseguire più operazioni in parallelo, il valore `NULL` sta a indicare al compilatore di usare quella di default che è quella seriale;

- `dispatch_async`, infine con questa istruzione andremo a creare un blocco per una esecuzione asincrona contenente il codice del thread da eseguire, quindi la nostra operazione la aggiungeremo alla coda precedentemente creata.

Se invece proviamo ad usare il metodo NSOperationQueue è possibile realizzare in maniera molto simile ma con pochissime linee di codice la stessa soluzione:

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init];
        NSLog(@"Thread Princiaple");
        [queue addOperationWithBlock:^{

            NSLog(@"Thread Figlio");
            for (int i = 5; i > 0; i--) {
                NSLog(@"%d",i);
                sleep(1);
            }
            NSLog(@"Uscita Thread Figlio");

        }];

        sleep(3);
        NSLog(@"Uscita Thread Principale");
```

In pratica abbiamo creato una NSOperationQueue e abbiamo aggiunto un'operazione tramite una funzione Block, e il risultato di questi due tipi di implementazioni diverse è il medesimo, ossia:

```
Thread Principale
Thread Figlio
5
4
3
Uscita Thread Principale
2
1
Uscita Thread Figlio
```

Questi due tipi diversi di implementazioni sono molto semplici, ma usando l'NSOperationQueue possiamo realizzare implementazioni più articolate in modo che vengano eseguite classi diverse concorrentemente. Proviamo adesso ad usare l'NSOperation e l'NSOperationQueue insieme.
Innanzi tutto una NSOperationQueue è una coda di NSOperation che vengono eseguite concorrentemente, quindi se vogliamo che una certa classe sia eseguita concorrentemente deve essere una sottoclasse di NSOperation, proviamo a realizzare il precedente risultato in maniera leggermente diversa:

`MyClass.h`

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

@interface MyClass : NSOperation

@end

MyClass.m

#import "MyClass.h"

@implementation MyClass

- (void)main {

  @autoreleasepool {
    NSLog(@"Thread Figlio");
        for (int i = 5; i > 0; i--) {
            NSLog(@"%d",i);
            sleep(1);
        }
        NSLog(@"Uscita Thread Figlio");
  }
}

@end
```

Quindi abbiamo creato una classe che si chiama MyClass che è una sottoclasse di NSOperation, e l'operazione parte dal metodo `main` che è quello di

default per una NSOperation poi abbiamo inserito le nostre istruzioni all'interno della keyword `@autoreleasepool`, perché una NSOperation è anch'essa un oggetto, quindi bisogna sempre dichiarare il codice nel `main` all'interno di questa keyword che provvederà a proteggere l'operazione contro perdite di memoria che possiamo avere eseguendo il codice in parallelo, infine possiamo fare partire la nostra operazione in qualsiasi altra classe semplicemente facendo:

```
NSOperationQueue *queue = [[NSOperationQueue alloc] init
   ];
MyClass *myClass = [[MyClass alloc] init];
NSLog(@"Thread Principale");
[queue addOperation:myClass];
sleep(3);
NSLog(@"Uscita Thread Principale");
```

Appena aggiungiamo l'operazione alla coda, verrà eseguita automaticamente quando la coda lo ritiene opportuno, per esempio se c'è un limite di operazioni eseguibili concorrentemente e non ci sono posti liberi, la nostra operazione verrà messa in coda in attesa che le altre terminino l'esecuzione. Nel nostro esempio il risultato sarà il medesimo delle implementazioni precedenti, però in questo caso possiamo estendere le NSOperation ad altre classi in modo da avere più classi che operano concorrentemente.

17 Blocks in Objective-C

Un Blocco è un particolare tipo di esecuzione di istruzioni che non troviamo in Java, è molto utile perchè sono delle vere e proprie funzioni (metodi) che vengono dichiarati "on the fly" ed eseguite quando si arriva a quella istruzione di codice. Vediamo nel dettaglio di cosa si tratta, proviamo a prendere la prima implementazione della NSOperationQueue, ho usato questo tipo di soluzione:

```
[queue addOperationWithBlock:^{

            NSLog(@"Thread Figlio");
            for (int i = 5; i > 0; i--) {
                NSLog(@"%d",i);
```

```
            sleep(1);
        }
        NSLog(@"Uscita Thread Figlio");

    }];
```

quindi in pratica la classe NSOperationQueue oltre ai molti metodi di default ha anche un metodo che usa i Blocks, ossia addOperationWithBlock, e tutto quello che c'è tra le parentesi graffe è codice che viene eseguito "on the fly", cioè grazie a questo tipo di implementazione non devo creare una classe NSOperation come ho fatto nel secondo esempio, ma mi basta semplicemente usare un metodo così per poter eseguire quelle determinate istruzioni in poche righe di codice. Questa funzione è molto utile per gli sviluppatori e ci sono moltissime classi che offrono metodi per poter eseguire codice nei Blocks.

Possiamo anche creare un nostro metodo blocks in maniera molto semplice:

```
    int (^triple) (int)
    <parametrodiRitorno> (^<nomefunzione>) (<
       parametriIngresso)


int (^triple)(int) = ^(int number) {
  return number *3;
};
```

Per usare questo metodo basta semplicemente fare:

```
int result = triple(3);

//result = 9
```

Capitolo 3

Porting ed esempi di programmazione

In questo capitolo spiegherò nel dettaglio il significato di porting, e le difficoltà che si possono incontrare nell'effettuarlo. Successivamente introdurrò un tool creato per poter facilitare questo porting, il quale effettua una traduzione da Java a Objective-C , e andrò a descrivere alcuni esempi di programmazione in Java, provando a tradurli con il tool per verificare se la traduzione da Java a Objective-C che effettua è corretta. Questa operazione è necessaria per capire le differenze tra i due linguaggi di programmazione esposti nel capitolo precedente, non mostrerò il codice completo, ma solo le parti di codice per me fondamentali per la creazione dello specifico esempio, mostrando passo passo le differenze tra il codice originale Java, la mia implementazione in Objective-C e infine la traduzione con il tool.

1 Porting

Il Porting, è un processo usato per adattare l'esecuzione di un software in un ambiente diverso da quello originariamente pensato, per esempio differente CPU, differente sistema operativo, o differente linguaggio di programmazione. Un software può essere considerato "portable" se eseguirne il porting è semplice (poco costoso), rispetto a effetture un porting partendo da zero, cioè riscrivendo tutto, in tal caso il porting diventerebbe un'attività complessa e

costosa.

La portabilità dei programmi dipende essenzialmente dal linguaggio di programmazione usato. Alcuni linguaggi non si possono considerare portabili (ovvero non consentono la scrittura di programmi portabili) per il semplice motivo che per alcuni ambienti non esiste un interprete o compilatore per quel linguaggio; oppure perché i compilatori o interpreti disponibili in diversi ambienti presentano alcune differenze più o meno sottili relativamente alla sintassi che accettano o alla semantica che attribuiscono ad alcuni costrutti; cosicché un programma che funziona correttamente su una macchina potrebbe esibire dei malfunzionamenti più o meno gravi, o addirittura non compilare su un'altra.

I programmi scritti in C sono portabili, nel senso che possono essere compilati in qualsiasi ambiente, ottenendo programmi in linguaggio macchina ovviamente diversi ma che esibiscono un comportamento semanticamente equivalente. Cè però da osservare che nonostante il C e il C++ siano linguaggi portabili, la diversità dei compilatori dei due linguaggi di programmazione pone una serie di problemi alla portabilità. Il solo fatto di avere un programma in C oppure in C++ non garantisce la portabilità del medesimo e quindi spesso bisogna fare uno sforzo aggiuntivo per superare le differenze tra i due sistemi e compilatori.

Una forma più radicale di portabilità viene fornita dai linguaggi interpretati, ovviamente sotto la condizione che esistano interpreti per i diversi ambienti di interesse, e che questi interpreti esibiscano un comportamento equivalente (es. conforme ad uno standard). In questo caso il "porting" di un programma non richiede nessuna ricompilazione. E' per esempio il caso di Java, il software Java può essere eseguito in qualunque ambiente su cui sia installata la così detta macchina virtuale Java (più precisamente il JRE, Java Runtime Environment, prelevabile dal sito ufficiale della Oracle). Un'altro aspetto della portabilità di Java è quello che rende possibile l'inserimento di piccoli programmi Java (chiamati applet) all'interno di pagine Web destinate a essere visualizzate da diversi browser in esecuzione su macchine completamente

diverse.

Quindi il porting di Java grazie alla macchina virtuale non è un problema, e rende l'applicazione così detta "portable", ma non è sempre così semplice perchè non in tutti i dispositivi è presente la macchina virtuale Java, ma soprattuto non in tutti i dispositivi mobili, che sono l'argomento principale di questa tesi, è presente. Infatti nei dispositivi portatili Apple non esiste la macchina virtuale Java, ed è per questo che in questa tesi stiamo cercando di capire come fare per passare da un'app scritta in Java a un'app scritta in Objective-C. Questa mancanza di questo fondamentale strumento rende la vita difficile a moltissimi sviluppatori di App per dispositivi mobili che hanno sviluppato un'app per i dispositivi che usano Java, e che vorrebbero estendere il loro prodotto anche sui dispositivi mobili Apple. Quindi in questo caso non possiamo considerare il concetto di "portable" perchè come detto in precedenza per essere considerata tale, l'operazione di porting deve essere il meno costosa possibile e non deve essere riscritto tutto da zero.

2 Tool di traduzione: j2Objc

A volte però possono esistere vie alternative, ossia progetti di terze parti, che possono semplificare il lavoro di porting, questi vengono chiamati convertitori, in questo caso traduttori perchè passiamo da un linguaggio di programmazione (Java) ad un'altro (Objective-C). Dopo una lunga ricerca e dopo molte prove fatte, l'unico vero traduttore attualmente in circolazione, è il tool j2ObjC. J2ObjC è un progetto Open Source creato da Google, l'obbiettivo di questo tool è quello di tradurre il codice Java senza interfaccia grafica in Objective-C. E' reperibile al seguente indirizzo:

https://code.google.com/p/j2objc/

dove è possibile prelevare il tool, attualmente alla versione 0.5.5, ed è anche disponibile una piccola documentazione su come orientarsi, inoltre si può

trovare una tabella di mapping dove spiegano cosa è possibile tradurre con questo tool e come lo traducono, vediamola nel dettaglio:

Java	Objective-C
packages	class naming
classes	interfaces
interfaces	protocols plus constants
enum	enum design
instance variables	properties
method overloading	embedded parameter types
static variables and constants	static variables
inner classes	outer classes (class naming)
anonymous classes	outer classes (class naming)
arrays	array emulation
Object.clone, java.lang.Cloneable	clone support, NSCopying
synchronized	@synchronized
try/catch/finally	@try/@catch/@finally
java.lang.Object	NSObject (extended)
java.lang.String	NSString (extended)
java.lang.Number	NSNumber
java.lang.Throwable	NSException (extended)
java.lang.Class	native wrapper around Objective-C Class
boolean	BOOL
byte	char
char	unichar
double	double
float	float
int	int
long	long long int
short	short int
Java serialization	not implemented
Java reflection	subset to support test frameworks
JUnit tests	JUnit translation

Quindi come è possibile vedere secondo questa tabella con questo tool possiamo tradurre praticamente tutto il codice scritto in Java in maniera elementa-

re, vengono tradotti tutti i tipi primitivi, le classi, i metodi le interfacce ecc... Ma la domanda che adesso ci poniamo è questa, come vengono tradotti? Cioè la traduzione è corretta? Per poter rispondere a queste domande, proverò ad eseguire le traduzioni di alcuni esempi scritti in Java e poi tradotti da me in Objective-C per confrontarne le soluzioni.

3 Esempio del Contatore

Ho più volte descritto nel corso di questa tesi l'esempio del contatore, e lo farò anche in questo paragrafo perché ritengo sia un ottimo punto di partenza per testare le funzionalità del traduttore, in quanto l'esempio della classe contatore contiene dei semplici metodi che settano il valore di una variabile primitiva int. Per poter inizializzare la classe ci sono due tipi di costruttori, uno che inizializza il conteggio della variabile a 0, e l'altro che definisce la variabile con il valore definito dall'utente. Quindi questo esempio serve solo per dimostrare come poter usare variabili primitive, costruttori diversi e variabili `private` alle quali è possibile accedere tramite metodi pubblici creati per lo scopo.

3.1 Implementazione in Java

In questo listato descriviamo come è possibile creare una classe e dichiarare una variabile privata int che terrà il conteggio:

```
public class Counter {
    private int cont;
```

ovviamente ricordiamo che Java dichiara la classe in un solo file con estensione .java, nel quale possiamo avere solo una classe pubblica. Successivamente vediamo l'esempio non del costruttore di default (cioè quello che inizializza il contatore con valore 0) ma di quello che inizializza il contatore con un valore che gli daremo noi:

```
public Counter(int v) {
        cont = v;
    }
```

Infine vediamo l'implementazione di un semplice metodo per esempio una delle funzioni base del contatore, ossia aumentare il conteggio che faremo tramite il metodo inc:

```
public void inc(){
        cont++;
    }
```

3.2 Mia implementazione in Objective-C

Vediamo la corrispettiva implementazione in Objective-C:

```
Counter.h

@interface Counter : NSObject {
    int cont;
}
...
```

Come abbiamo spiegato nel precedente capitolo, l'Objective-C a differenza di Java, dichiara e implementa una classe in due file distinti, un file chiamato header file, con estensione .h e un file chiamato source file con estensione .m, e quindi nel file .h andremo solo a dichiarare il nome della classe e la nostra variabile int cont.

Successivamente vediamo l'implementazione del metodo costruttore con un parametro in ingresso:

```
Counter.m

@implementation Counter

-(id)initWithCount:(int)value {

    if (self = [super init]) {

        [self setCount:value];
    }
```

```
    return self;

}
```

in cui dopo la dichiarazione `@implementation`, c'è l'implementazione del costruttore per poter inizializzare un contatore con un valore scelto da noi. In pratica qui andremo a richiamare `self = [super init]` che ci serve per poter inizializzare il nostro oggetto dalla nostra super classe che in questo caso è NSObject (come abbiamo spiegato nel capitolo precedente), e poi andremo a settare la variabile count con il parametro in ingresso value, e infine ritornerà la keyword `self` che sta a indicare appunto noi stessi ossia il riferimento della classe in cui stiamo eseguendo il codice quindi in questo caso Counter. Infine vediamo anche qui l'implementazione del metodo per incrementare il conteggio:

```
- (void)inc {

    cont++;
}
```

l'implementazione del metodo è molto simile a quella di Java, la sola differenza sta nel fatto che Objective-C usa istruzioni diverse per la creazione di un metodo, in questo caso void perché non avremo nessun parametro di ritorno.

3.3 Traduzione del Contatore con il Tool

Vediamo alcuni frammenti della traduzione. Innanzi tutto dalla traduzione del file Counter.java riceveremo in output due file, Counter.h e Counter.m, questa cosa è positiva vuol dire che il traduttore ha riconosciuto la sintassi di Java, e la adattata creando una sintassi per l'Objective-C, andiamo a vedere nel dettaglio il file Counter.h:

```
#import "JreEmulation.h"

@interface Counter : NSObject {
 @public
  int cont_;
}
```

```
@property (nonatomic, assign) int cont;

- (id)init;
- (id)initWithInt:(int)v;
- (void)setValueWithInt:(int)v;
- (void)inc;
- (void)dec;
- (void)resetCont;
- (int)getValue;
@end
```

la prima cosa che c'è subito da notare in questa traduzione, è la prima riga, ossia importa il file JreEmulation.h, di cui per adesso non si capisce l'utilità ma la spiegherò nei prossimi esempi, continuando a leggere il codice il traduttore ha fatto un ottimo lavoro, ha mappato ogni singolo costruttore e metodo che avevamo in Java seguendo la sintassi dell'Objective-C, vediamo il costruttore che richiede un parametro in ingresso:

```
- (id)initWithInt:(int)v {
  if ((self = [super init])) {
    cont_ = v;
  }
  return self;
}
```

anche qui la traduzione è corretta, a parte il file JreEmulation.h che ne spiegheremo l'utilità a breve, la traduzione è avvenuta con successo, senza nessun errore.

4 Esempio Array

In questo esempio mostreremo come poter usare un array, gestito da un'altra classe, quindi avremo due classi, di cui una manderà le richieste di aggiungere o eliminare o riportare quanti elementi ha l'array, e l'altra appunto che eseguirà e risponderà alle richieste che gli arriveranno.

4.1 Implementazione in Java

Partiamo con l'implementazione della classe che gestirà l'array, in Java:

```java
import java.util.*;

public class MyArray {
    ArrayList<Integer> myArray = new ArrayList<Integer>();
    ...
```

Come si può vedere questa è la solita implementazione di una classe, vorrei solo sottolineare che abbiamo usato un ArrayList in questo caso di tipo Integer, perché l'ArrayList è la struttura più dinamica e utile di un array che possiamo avere in Java. Continuamo descrivendo i rispettivi metodi per poter aggiungere, eliminare o avere il conteggio di quanti elementi ci sono nell'array, però guardiamo nel dettaglio solamente un metodo:

```java
public boolean addElementInArray(Integer i) {

        boolean result = false;

        for (Integer a:myArray) {

            if (a.intValue() == i.intValue()) {

                result = true;
                break;
            }

        }

        if (!result) {

            myArray.add(i);
            return true;
        }

        return false;
    }
```

in questo metodo andiamo ad aggiungere un elemento nell'array, quindi innanzi tutto vediamo che il metodo è dichiarato come public, quindi potrà essere usato anche da classi esterne, dopo di che avremo un parametro Integer in ingresso che sarà appunto il nostro elemento che vorremo aggiungere nell'array, ma prima di aggiungerlo controlliamo se è già disponibile un ele-

mento uguale facendo una iterazione di tutto l'array, se non troviamo niente di uguale allora aggiungiamo l'elemento all'array e ritornerà `true` se è stato aggiunto, altrimenti `false` in caso contrario.
Infine per poter usare la classe dovremo richiamarla in questo modo in qualsiasi altra classe:

```
MyArray myArray = new MyArray();
```

e per esempio per usare il metodo per poter aggiungere un elemento dovremo usare questa istruzione:

```
boolean result = myArray.addElementInArray(i);
```

dove i sarà un Integer, e la variabile di ritorno verrà inserita in result per sapere se l'aggiunta è andata a buon fine oppure no.

4.2 Mia implementazione in Objective-C

Partiamo con la dichiarazione della classe nell'header file:

`MyArray.h`

```
@interface MyArrayClass : NSObject

@property (nonatomic, strong) NSMutableArray *myArray;
```

come al solito abbiamo la dichiarazione della classe e delle variabili nel nostro header file.h, ma anche qui vorrei sottolineare una cosa, ossia quale tipo di array ho usato in Objective-C, come ho spiegato nel capitolo precedente ciò che più si avvicina in Java a un ArrayList è proprio un NSMutableArray, quindi andremo ad usare questo tipo di array, e lo dichiareremo tramite la `@property` nell'header file perché vogliamo accedervi anche al di fuori di questa classe, quindi non sarà una variabile private ma public.

Vediamo ora il metodo per poter aggiungere un elemento nell'array:

`MyArray.m`

```
@implementation MyArrayClass
...
- (BOOL)addNumberInArray:(NSNumber *)number {

    BOOL found = NO;

    for (NSNumber *myNumber in self.myArray) {

        if ([myNumber intValue] == [number intValue]) {
            found = YES;
            break;
        }

    }

    if (!found) {
        [self.myArray addObject:number];

        return YES;

    } else {
        NSLog(@"Il numero %d esiste già.",[number intValue]);

        return NO;
    }

    return NO;

}
```

Come possiamo vedere è molto molto simile a quella vista precedentemente in Java, per poter dichiarare questo metodo come pubblico dobbiamo dichiarare questo:

```
- (BOOL)addNumberInArray:(NSNumber *)number
```

nel file .h fuori dalla dichiarazione di `@interface` e prima dell'`@end`.

Per poter poi usare questa classe dovremo scrivere:

```
MyArrayClass *myArrayClass = [[MyArrayClass alloc] init];
```

e poi potremo usare questo per aggiungere un elemento nell'array:

```
BOOL result = [myArrayClass addNumberInArray:<NSNumber>];
```

Il risultato dell'implementazione in Java e in Objective-C è il medesimo, quindi questa mia traduzione è corretta.

4.3 Traduzione con il tool dell'esempio array

Proviamo ora a tradurre questo esempio per gestire gli array, quindi traducendo la classe MyArray.java. Vediamo come si è comportato in questo caso il traduttore descrivendo alcuni frammenti di codice, partiamo dall'header file:

`MyArray.h`

```
#import "JreEmulation.h"

@interface MyArray : NSObject {
 @public
  JavaUtilArrayList *myArray_;
}

@property (nonatomic, retain) JavaUtilArrayList *myArray;

- (id)init;
- (BOOL)addElementInArrayWithJavaLangInteger:(JavaLangInteger
    *)i;
- (BOOL)removeElementInArrayWithJavaLangInteger:(
   JavaLangInteger *)i;
- (int)countElementInArray;
- (BOOL)addRapidElementWithJavaLangInteger:(JavaLangInteger
   *)i;
- (BOOL)removeRapidElementWithJavaLangInteger:(
   JavaLangInteger *)i;
@end
```

possiamo vedere anche qui l'import del file JreEmulation.h, se proseguiamo la lettura del codice vediamo un elemento che non ci aspettavamo ossia è stato dichiarato un nuovo tipo di array, di tipo `JavaUtilArrayList`, infatti come ho spiegato più volte nel corso di questa tesi, secondo la mia conoscenza dell'Objctive-C, l'oggetto che più si avvicina a un `ArrayList` in Java è un

NSMutableArray in Objective-C, qui invece abbiamo un nuovo tipo di Array, che non ha nulla a che fare con l'Objective-C e con il famoso Foundation Framework di cui ho parlato nei capitoli precedenti. Il problema di questo tool è che quando si trova ad aver a che fare con tipi non primitivi come può essere un `int` o un `boolean`, ma con oggetti più complessi come può essere un Array, gli sviluppatori del tool hanno deciso di creare una libreria di emulazione, chiamata appunto `JreEmulation`, la quale nel caso degli array crea un nuovo tipo chiamato appunto `JavaUtilArrayList`, che secondo la loro documentazione è un adattamento di un array in C. Quindi qualora io volessi esportare questa soluzione in altri progetti in Objective-C, non posso, perché dovrei esportare anche tutta la loro libreria di emulazione, ma vediamo anche un frammento del file .m, in particolare la traduzione del metodo che aggiunge un elemento nell'array:

```
- (BOOL)addElementInArrayWithJavaLangInteger:(JavaLangInteger
    *)i {
  BOOL result = NO;
  {
    id<JavaLangIterable> array__ = (id<JavaLangIterable>)
       myArray_;
    if (!array__) {
      @throw [[[JavaLangNullPointerException alloc] init]
         autorelease];
    }
    id<JavaUtilIterator> iter__ = [array__ iterator];
    while ([iter__ hasNext]) {
      JavaLangInteger * a = (JavaLangInteger *) [iter__ next
         ];
      if ([((JavaLangInteger *) NIL_CHK(a)) intValue] == [((
         JavaLangInteger *) NIL_CHK(i)) intValue]) {
        result = YES;
        break;
      }
    }
  }
  if (!result) {
    [((JavaUtilArrayList *) NIL_CHK(myArray_)) addWithId:i];
    return YES;
  }
```

```
  return NO;
}
```

anche qui la soluzione non ha nulla a che vedere con quella che ho fatto io in Objective-C, vengono creati anche qui nuovi tipi come per esempio `JavaUtilIterator` o `JavaLangIterable`, ma questo non è quello che serve a noi, perchè non è un vero e proprio porting da Java a Objective-C, ma è un adattamento, un emulazione di Java adattata all'Objective-C. Però l'unica cosa positiva è che la sintassi dello scheletro del metodo è corretta, ossia la struttura è fedele a quella dell'Objective-C ma tutto il resto no.

5 Esempio Inner Class

In questo esempio mostreremo le differenze nell'uso di Inner Class in Java e di come implementare la mancanza di questo concetto in Objective-C, queste due classi gestiscono un array di numeri interi da 0 a 15, e scambiando messaggi tra di loro arriveranno a una soluzione finale ossia quella di stampare in console solamente i numeri pari.

5.1 Implementazione in Java

Innanzi tutto in java ricordiamo che ogni file .java può avere una sola classe pubblica, quindi il nostro file per esempio si chiamerà InnerClassExample.java e la classe pubblica dovrà chiamarsi InnerClassExample, dopo di che creeremo un nuovo array con una grandezza SIZE che sarà una variabile int statica, vediamo l'implementazione:

```
public class InnerClassExample {

    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];
    ...
```

Proseguiamo vedendo come viene creata una inner class che è l'argomento principale di questo esempio, in Java:

```
private class InnerEvenIterator {

        private int next = 0;
```

```
        public boolean hasNext() {

            return (next <= SIZE - 1);
        }

        public int getNext() {

            int retValue = arrayOfInts[next];

            next += 2;
            return retValue;
        }
    }
```

potremo poi accedere ovunque nella nostra classe a questa nuova classe privata in questo modo:

```
InnerEvenIterator iterator = this.new InnerEvenIterator();
```

e sarà possibile accedere ai suoi metodi con questa istruzione:

```
iterator.hasNext()
```

questa classe come dice il nome è privata, quindi potrà essere vista solamente all'interno della classe principale InnerClassExample, e di conseguenza anche i due metodi pubblici `hasNext()` e `getNext()` potranno essere usati solo all'interno della classe principale, invece per esempio la variabile `next` potrà essere vista solo all'interno della classe private InnerEvenIterator, invece è possibile accedere al nostro array `arrayOfInts` creato nella classe principale anche dentro alla classe private come è visibile nell'esempio, quindi in pratica l'InnerClass è molto utile per poter dichiarare come dice il nome stesso classi all'interno di classi.

5.2 Implementazione in Objective-C

Iniziamo descrivendo l'implementazione della classe nell'header file:

`InnerClassExample.h`

```
#define SIZE 15

@interface InnerClassExample : NSObject {

    NSInteger arrayOfInts[SIZE];
    int next;
}
```

quindi anche in questo caso come nell'esempio precedente possiamo vedere come viene differentemente gestita una classe in Objective-C, verrà dichiarata nel file .h e definiamo una variabile globale SIZE con il valore 15, simile a quelle static in Java.

Purtroppo come ho già spiegato non è possibile realizzare un'Inner Class in Objective-C, però ci sono due possibili alternative per poter ovviare a questo problema. La prima alternativa è quella di eliminare il concetto di Inner Class (dove è possibile) ed unire le due classi in un unica classe, quindi in pratica inserire il metodo `hasNext` e `getNext` all'interno dell'implementazione della classe InnerClassExample come possiamo vedere qui di seguito:

```
@implementation InnerClassExample
...

- (BOOL)hasNext {

    return (next <= SIZE - 1);

}

- (int)getNext {

    int retValue = arrayOfInts[next];
    next += 2;

    return retValue;
}

...
```

La seconda alternativa, più simile al concetto di InnerClass di Java, sta nella possibilità in Objective-C di poter creare nuove `@interface` all'interno dello

stesso file, quindi in pratica andremo a fare nel seguente modo:

`InnerClassExample.h`

```
@interface InnerClassExample : NSObject {

    NSInteger arrayOfInts[SIZE];
}

...

@end

@interface InnerEvenIterator : NSObject {

  int next;
  InnerClassExample2 *mainClass;
}

- (BOOL)hasNext;
- (int)getNext;

@end
```

e poi dovremo fare la stessa cosa nel source file, ossia implementare questa nuova classe, quindi avremo:

```
@implementation InnerClassExample
.....

@end

@implementation InnerEvenIterator
....
@end
```

così facendo avremo creato una classe dentro un'altra classe, la quale anche in questo caso come in Java non sarà visibile dall'esterno, l'unica differenza con Java è che in questa nuova classe, non potrò accedere all'array dichiarato nella classe principale, ma saranno due classi completamente distinte e separate, l'unica cosa che le accomuna è che sono state create nello stesso file.h e file.m, quindi in pratica per poter accedere alle rispettive risorse che forniscono le

due classi per esempio la classe principale dovrà poter accedere ai metodi `hasNext` e `getNext`, invece la classe InnerEvenIterator dovrà poter accedere all'array per poter dire quale sarà il prossimo elemento e se ce ne sarà uno, per fare ciò dovremo creare un nuovo costruttore nella classe privata, in modo che sia inizializzato con il riferimento della classe InnerClassExample, vediamo:

```
-(id)initWithInnerClassExample:(InnerClassExample *)newClass
   {

    if (self = [super init]) {

        mainClass = newClass; //mainClass è di tipo
           InnerClassExample

    }

    return self;

}
```

e nella classe principale dovremo fare questo per poter creare l'oggetto della classe privata:

```
InnerEvenIterator *innerClass = [[InnerEvenIterator alloc]
   initWithInnerClassExample:self];
```

così facendo abbiamo creato un nuovo oggetto della classe privata, e gli abbiamo passato l'istruzione self, che sta a indicare se stesso, quindi in pratica gli abbiamo passato il nostro riferimento della classe principale alla classe privata, in modo da lavorare con gli stessi riferimenti, e così potremmo accedere ai metodi della classe privata in questo modo:

```
[innerClass hasNext]
```

sia in Java che in Objective-C siamo riusciti ad arrivare alla stessa soluzione, ossia stampare dall'array di interi da 0 a 15 solo i numeri pari quindi in tutti i casi avremo questo risultato:

```
0
2
4
6
```

```
8
10
12
14
```

quindi in pratica in questo caso, sta allo sviluppatore decidere cosa è meglio fare, se integrare le classi in un'unica classe, oppure creare due classi all'interno dello stesso file come abbiamo visto nel secondo esempio che è quello che più si avvicina all'idea di Inner Class di Java, e questa implementazione è molto più complicata e richiede molte più linee di codice rispetto all'implementazione in Java.

5.3 Traduzione dell'esempio InnerClass con il tool

Questo esempio serviva solo per mostrare la mancanza dell'Inner Class in Objective-C, e di come è possibile colmare questa mancanza, quindi siccome abbiamo capito che questo tool non converte come dovrebbe gli array e tutti gli altri tipi di oggetti, ma ne emula secondo una loro libreria le funzionalità, vorrei solo mostrarvi nel dettaglio se è riuscito a tradurre il concetto di Inner Class che manca in Objective-C. Vediamo alcuni frammenti della traduzione:

`InnerClassExample.h`

```
@interface InnerClassExample : NSObject {
 @public
  IOSIntArray *arrayOfInts_;
}
...
@end

@interface InnerClassExample_InnerEvenIterator : NSObject {
 @public
  InnerClassExample *this$0_;
  int next_;
}

@property (nonatomic, retain) InnerClassExample *this$0;
```

```
....

@end
```

tralasciando ovviamente anche qui la non corretta traduzione dell'array, possiamo però vedere che la mancanza dell'Inner Class in Objective-C è stata gestita molto bene, hanno creato una nuova interfaccia e anche la sua implementazione che non ho mostrato è corretta, quindi l'unica cosa corretta è la struttura.

6 Thread

L'esempio dei Thread è stato dettagliatamente spiegato nel capitolo precedente sia in Java che in Objective-C, quindi vorrei solamente mostrare come viene tradotto questo esempio con il tool e vediamo se anche in questo caso useranno la loro libreria di emulazione, partiamo dall'header file:

`MyThread.h`

```
#import "JreEmulation.h"
#import "java/lang/Runnable.h"

@interface MyThread : NSObject < JavaLangRunnable > {
}

- (id)init;
- (void)run;
@end
```

anche in questo caso, hanno emulato la classe `Runnable` di Java per poter usare i Thread, quindi questa implementazione a mio avviso non ha nulla a che vedere con l'Objective-C, anche qui l'unica cosa corretta è la struttura della classe, vediamo anche un piccolo frammento dell'implementazione:

`MyThread.m`

```
...
```

```
JavaLangThread *t = [[[JavaLangThread alloc]
    initWithJavaLangRunnable:self withNSString:@"Thread figlio
    "] autorelease];
NSLog(@"%@", [NSString stringWithFormat:@"thread attuale: %@"
    , ct]);
NSLog(@"%@", [NSString stringWithFormat:@"thread creato: %@",
     t]);
[((JavaLangThread *) NIL_CHK(t)) start];
...
```

sono stati creati nuovi tipi di classi come `JavaLangThread` che non sono classi che fanno parte delle librerie dell'Objective-C.

7 Chat

In questo esempio ho realizzato una Chat che usa un Server scritto in Java sul quale non mi soffermerò, ma mi soffermerò sull'implementazione del Client Java e del Client Objective-C, quindi in questa chat potranno comunicare dispositivi scritti in linguaggi diversi, che scambiano messaggi con lo stesso server.

7.1 Implementazione del Client Java

Ora mostrerò solo le istruzioni principali per spiegare il funzionamento di questo Client.

Andremo a collegarci al nostro Server usando la classe Socket:

```
Socket socket = new Socket(server, port);
```

qui andremo a creare un oggetto Socket, che ci permetterà di scambiare messaggi con il Server, e per crearlo dovremo passare due parametri, l'indirizzo del server e la porta. Dopo di che andremo ad aprire i due canali di comunicazione per poter inviare messaggi e ricevere messaggi:

```
DataInputStream inStream = new DataInputStream(socket.
    getInputStream());
DataOutputStream outStream = new DataOutputStream(socket.
    getOutputStream());
```

dopo di che creeremo un Thread che rimarrà in ascolto per tutta l'esistenza del Client, per ricevere messaggi in ingresso:

```
class ListenFromServer extends Thread {

    public void run() {
      while(true) {
        try {
          String msg = inStream.readUTF();
            System.out.println(msg);
            System.out.print("> ");
        }
        catch(IOException e) {
          display("Server has close the connection: " + e);
                    break;
      }
    }
  }
    }
```

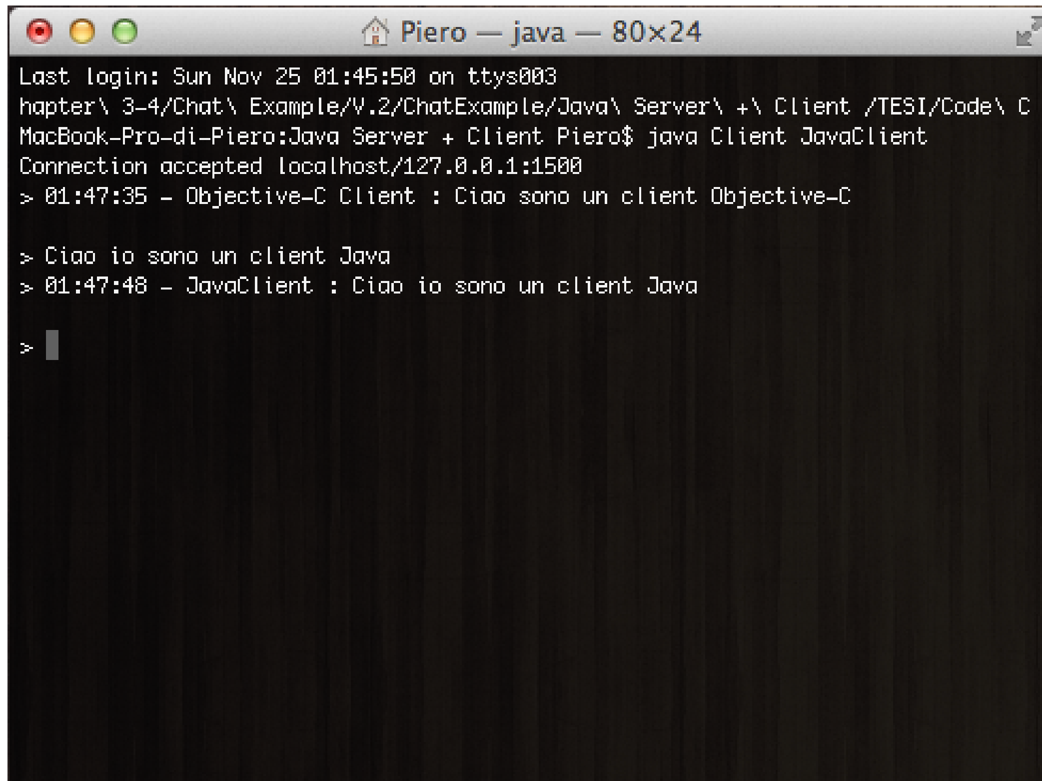
come possiamo vedere la parte che più ci interessa è questa:

```
String msg = inStream.readUTF();
```

è proprio con questa istruzione che andremo a leggere i messaggi che arrivano dall'esterno, vediamo invece ora come fare per scrivere messaggi:

```
outStream.writeUTF(msg);
```

in questo modo sarà possibile spedire messaggi, come è possibile vedere nell'immagine sottostante per poter comunicare con il server e di conseguenza con gli altri Client.



7.2 Implementazione del Client in Objective-C

Partiamo anche qui come abbiamo fatto per la versione Java mostrando solo le istruzioni principali di questo esempio per connetterci con il nostro Server remoto:

```
CFReadStreamRef readStream;
CFWriteStreamRef writeStream;
CFStreamCreatePairWithSocketToHost(NULL, (CFStringRef)@"
    localhost", 1500, &readStream, &writeStream);

NSInputStream *inputStream = (__bridge NSInputStream *)
    readStream;
NSOutputStream *outputStream = (__bridge NSOutputStream *)
    writeStream;
[self.inputStream setDelegate:self];
[self.outputStream setDelegate:self];
[self.inputStream scheduleInRunLoop:[NSRunLoop currentRunLoop
    ] forMode:NSDefaultRunLoopMode];
```

```
[self.outputStream scheduleInRunLoop:[NSRunLoop
    currentRunLoop] forMode:NSDefaultRunLoopMode];
[self.inputStream open];
[self.outputStream open];
```

questo listato di codice appena scritto è stato spiegato dettagliatamente nel capitolo precedente al paragrafo Socket, poi per poter ricevere i messaggi dall'host remoto dobbiamo implementare di default questo metodo, perché ci siamo registrati all'NSStreamDelegate, anche questo passaggio è stato spiegato nel capitolo precedente:

```
- (void)stream:(NSStream *)theStream handleEvent:(
    NSStreamEvent)streamEvent;
```

dove per la lettura aspetteremo lo streamEvent: `NSStreamEventHasBytesAvailable` che ci indica che è arrivato un messaggio a andremo a leggere dei byte nel buffer, in questo modo:

```
uint8_t buffer[1024];
        int len;

        while ([self.inputStream hasBytesAvailable]) {
          len = [self.inputStream read:buffer maxLength:
              sizeof(buffer)];
          if (len > 0) {

            NSString *output = [[NSString alloc]
                initWithBytes:buffer length:len encoding:
                NSUTF8StringEncoding];

            if (nil != output) {
            NSLog(@"server said: %@", output);
            }
            }
```

andremo a leggere tutto il buffer e comporre il nostro messaggio in questo caso la nostra NSString, invece per poter spedire messaggi useremo queste istruzioni:

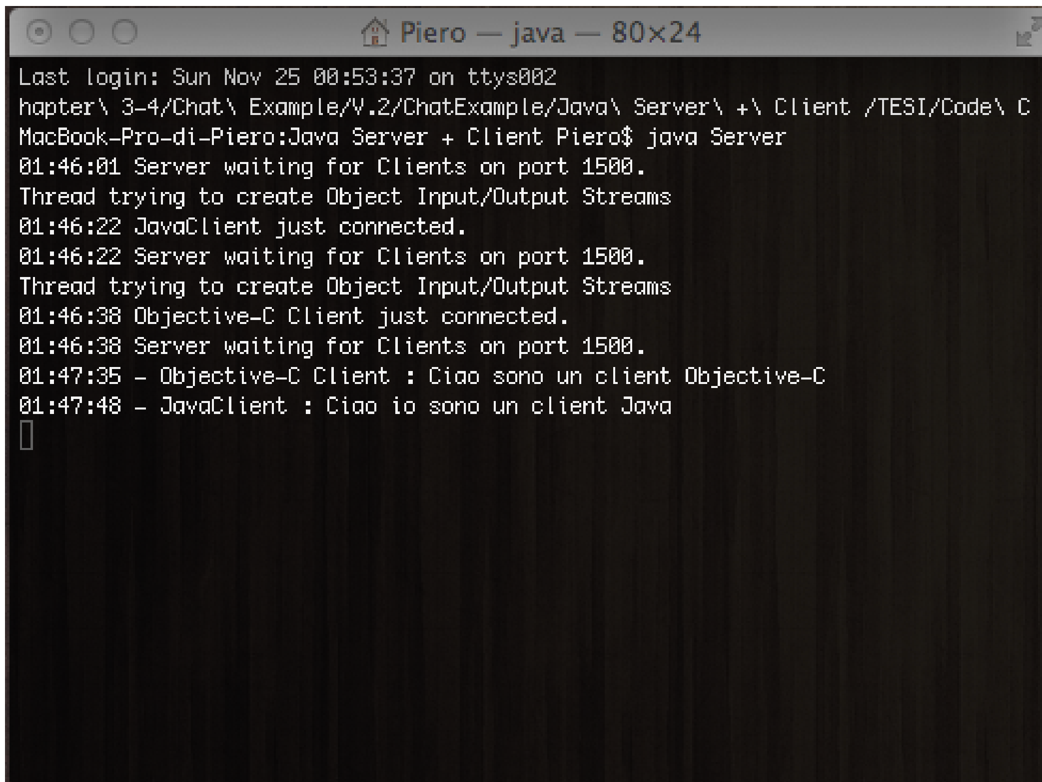
```
NSData *data = [self convertToJavaUTF8:messageToSend];

int dataLenght = [data length];
```

```
[self.outputStream write:(const uint8_t *)[data bytes]
    maxLength:dataLenght];
```

dove in pratica trasformeremo la nostra stringa di testo in un oggetto NSData usando la codifica UTF8, che è una codifica di caratteri Unicode in sequenza

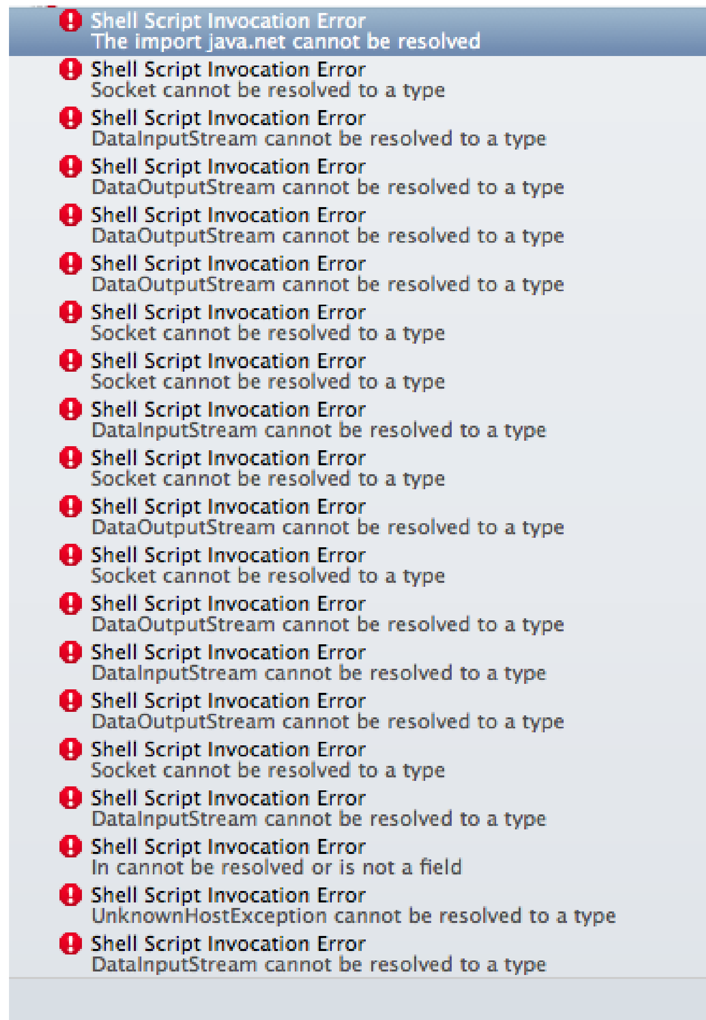
di lunghezza variabile di byte, e ci permette di poter spedire messaggi attraverso linguaggi di programmazione diversi, come in questo caso facciamo tra Java e Objective-C.



Infine come è possibile vedere nelle tre figure in questa sezione, cè uno scambio di messaggi tra un Client Java avviato tramite linea di comando, e un Client Objective-C avviato tramite il simulatore dell'iPhone offerto dall'ambiente di sviluppo Xcode dell'Apple, e infine la console del Server che riceverà i messaggi da entrambi i client e li spedirà ai suoi destinatari.

7.3 Traduzione dell'esempio della Chat

Ho provato infine a tradurre l'esempio dell'implementazione della Chat, ma il risultato della traduzione è il seguente:



ossia una lista di errori che non permettono la traduzione di questo esempio, perché il tool per il momento non supporta il package `java.net`, e quindi l'unica soluzione è tradurla manualmente.

Conclusioni

Come ho cercato di dimostrare in questa tesi Java e Objective-C sono i due linguaggi di programmazione più usati al momento per sviluppare applicazioni nel settore dei dispositivi mobil. Uno sviluppatore che opera in questo settore, per espandere il suo mercato, deve realizzare applicazioni per entrambi i linguaggi. Quindi se ha realizzato una applicazione in Java deve poterla sviluppare anche in Objective-C e viceversa, ed è per questo che ho iniziato descrivendo le principali caratteristiche che hanno in comune questi due linguaggi, perché essendo entrambi linguaggi di programmazione ad oggetti avranno sicuramente i concetti di base simili.

Molti altri concetti invece non coincidono, ed è per questo che nel secondo capitolo ho cercato di mettere in evidenza queste differenze. Tutto ciò è stato fatto per poter arrivare ad una soluzione comune, mostrando passo per passo le principali classi che uno sviluppatore deve conoscere in Java, e le corrispettive implementazioni in Objective-C.

Ho descritto tutte queste classi nei due linguaggi in quanto un'applicazione scritta in Java non può essere eseguita anche sui dispositivi che usano l'Objective-C, perché l'Apple per ora non ha intenzione di introdurre nei propri dispositivi la JVM, Java Virtual Machine, quindi uno sviluppatore è costretto ad effettuare un porting dell'applicazione per poter passare da un linguaggio di programmazione ad un'altro. Come abbiamo visto nel terzo capitolo esiste uno strumento creato da Google che si propone esplicitamente come traduttore e promette di tradurre il codice Java in Objective-C.

All'inizio dello sviluppo di questa tesi, ho effettuato molte prove con questo

tool e molte cose non funzionavano bene o non erano supportate, la versione attuale (0.5.5) con cui ho effettuato gli ultimi test ha apportato molti miglioramenti e compatibilità con diversi package Java che prima non erano presenti, e considerando il fatto che sono ancora tra la fase alpha e la fase beta della produzione di questo tool penso che gli sviluppatori di Google stiano facendo un buon lavoro nello sviluppo.

Però come ho mostrato nel terzo capitolo dopo aver provato ad eseguire la traduzione di diversi esempi ho potuto constatare che questo tool non è veramente un traduttore, ma è un emulatore. Esso infatti non permette una vera e propria traduzione del codice in Objective-C ma vengono create nuove classi che usano una nuova libreria creata per lo scopo. Quindi i problemi secondo me più evidenti, sono che il codice che viene tradotto non essendo scritto usando le classi dell'Objective-C non può essere esportato in altri progetti senza esportare anche le librerie di emulazione, e che se lo sviluppatore decide di apportare modifiche al codice tradotto non può farlo, in quanto non esiste una documentazione su come usare queste classi.
Concludendo, secondo la mia opinione attualmente l'unica soluzione che ha uno sviluppatore per poter passare da Java a Objective-C è quella di tradurre il codice manualmente senza usare traduttori di terze parti che potrebbero portare solo a rallentamenti nel porting e anche problemi di compatibilità.

Bibliografia

[1] Andrea Roli (2010/2011), *"Corso di Fondamenti di Informatica B"*, Programmazione Orientata agli oggetti introduzione;

[2] Horstmann, *"Concetti di informatica e fondamenti di Java 2"*;

[3] Ken Arnlod, *"The Java Programming Language"*;

[4] Apple, *"Apple Developer Documentation"*, Primer in Objective-C;

[5] G. Kochan, *"Programming in Objective-C'* Developer's Library;

[6] Google, *"j2Objc Wiki Website'* http://code.google.com/p/j2objc/wiki/Motivation;

Ringraziamenti

Dopo tutti questi anni di Ingengeria le persone da ringraziare sarebbero infinite e ricordamele tutte sarebbe veramente difficile, cercherò di fare del mio meglio.

Grazie a Giulia perché ha sempre creduto in me, e senza di lei forse non sarei qui oggi. Grazie a mia sorella per i suoi "ma si tanto è facile!". Grazie ai miei genitori che mi hanno sempre sostenuto e non mi hanno mai fatto pressioni. Grazie a Richard, Ste, Busca, Campo e Occhiuto, per tutte quelle ore passate davanti alle calcolatrici.

Grazie a Zack per questo primo anno di Code Strategy (o Nerd Pinguin o come la vorremo chiamare!) dove ci siamo tolti piccole soddisfazioni e spero ce ne toglieremo tante altre!

Grazie a tutti i miei amici che, "Ma sei ancora li?", grazie a Trave, Bongi, Cleo, Porto, Riccio, Marz e Savo e ai bellissimi momenti passati con loro che mi hanno fatto distrarre un po'.

Grazie al Prof. Omicini per la Sua disponibilità e continua simpatia.

Grazie al Dott. Stefano Mariani, che mi ha sempre aiutato in questi anni, e in questa tesi, e mi vuoi spiegare come hai fatto ad essere già lì ?!!? Scherzo continua così!

Grazie a Tea per i suoi salti carpiati sulle pareti e il suo affetto.