Dal linguaggio C al linguaggio Java

(Prima parte)

Riccardo Silvestri

15-3-2009 1-3-2010

Queste dispense sono intese fornire una introduzione alla programmazione orientata agli oggetti e al linguaggio Java. Si assume che il lettore conosca il linguaggio C cosicchè le caratteristiche di Java che derivano direttamente dal C sono trattate il più rapidamente possibile. L'approccio adottato è pragmatico e concreto, nel senso che si preferisce introdurre una nuova astrazione sulla base di una esigenza concreta piuttosto che fare l'opposto (prima l'astrazione e poi le applicazioni concrete). Quindi, gli esempi e gli esercizi assumono un'importanza centrale per l'introduzione e la spiegazione di nuovi concetti. Infine è bene sottolineare che, allo scopo di rendere rapida l'acquisizione di una buona dimestichezza con la programmazione ad oggetti e con Java, alcuni aspetti non sono trattati con la dovizia di dettagli che meriterebbero. Questo è giustificato dalla convinzione che una volta acquisita una buona padronanza del linguaggio diventa poi più facile comprendere le sottigliezze non solo del linguaggio stesso ma anche della programmazione ad oggetti.

Sommario della prima parte

Che cos'è Java

Java è un linguaggio di programmazione orientato agli oggetti (*Object Oriented*) con una sintassi simile a quella dei linguaggi C e C++. È un linguaggio potente che evita però quelle complesse caratteristiche che rendono poco maneggevoli altri linguaggi orientati agli oggetti come il C++.

Breve storia di Java

Nel 1991 un team di ingegnieri della Sun Microsystems, guidato da Patrick Naughton e James Gosling, iniziò la progettazione di un linguaggio con l'obbiettivo di agevolare la programmazione di piccoli apparecchi elettronici. Siccome tali apparecchi non avevano grandi quantità di memoria e le CPU potevano essere le più diverse, era importante che il linguaggio producesse codice snello e che non fosse legato ad una particolare architettura hardware. Questi requisiti portarono il team ad ispirarsi ad un modello che era stato adottato da alcuni implementatori del linguaggio Pascal ai tempi dei primi personal computer. Infatti, l'inventore del Pascal, Niklaus Wirth era stato il primo a introdurre l'idea di un linguaggio portabile basato sulla generazione di codice intermedio per un computer ipotetico detto *macchina virtuale* (*virtual machine*). Gli ingegnieri del team adottarono questo modello ma basarono il nuovo linguaggio sul C++ piuttosto che sul Pascal (questo perché la loro formazione era radicata nel mondo UNIX). All'epoca, comunque, il nuovo linguaggio non era visto come un fine ma solamente come un mezzo per la programmazione di piccoli apparecchi elettronici. Inizialmente Gosling pensò di chiamarlo "Oak" (quercia) ispirato da una grande quercia che poteva ammirare dal suo studio. Ma, presto si accorsero che esisteva già un linguaggio con quel nome. Il nome Java fu poi suggerito durante una pausa in un coffee shop.

Intanto, agli inizi degli anni '90, il mercato degli apparecchi elettronici "intelligenti" non era ancora sufficientemente sviluppato e l'intero progetto rischiava il fallimento. Però, il World Wide Web e internet stavano crescendo a ritmi elevatissimi e il team si rese conto che la nascente tecnologia dei browser poteva essere notevolmente potenziata proprio da un linguaggio con le caratteristiche di Java (indipendente dall'architettura hardware, real-time, affidabile e sicuro). Nel 1995 alla conferenza SunWorld presentarono il browser HotJava scritto interamente in Java che poteva eseguire codice (Java) contenuto in pagine web (ciò che oggi è chiamato *applet*). Agli inizi del 1996 la Sun rilasciò la prima versione di Java e il linguaggio iniziò a suscitare un interesse crescente. La versione 1.0 di Java non era certo adeguata per lo sviluppo serio di applicazioni. Ma da allora il linguaggio, attraversando parecchie versioni, è stato via via arricchito e le sue librerie sono state enormemente potenziate ed ampliate fino alla più recente versione 6, rilasciata nel 2006. Attualmente Java è un linguaggio maturo ed è usato per sviluppare applicazioni su grande scala, per potenziare le funzionalità di server web, per fornire applicazioni per apparecchi elettronici di largo consumo come cellulari e PDA e per tanti altri scopi.

La macchina virtuale

Quando un programma scritto in Java è compilato, il compilatore lo converte in *bytecodes* che sono le istruzioni di un linguaggio macchina di una CPU virtuale chiamata appunto *Macchina Virtuale Java* (*Java Virtual Machine*), in breve *JVM*. La JVM non corrisponde a nessuna CPU reale anche se, in linea di principio, potrebbe essere realizzata direttamente in hardware. La JVM è sempre implementata sotto forma di un software che esegue le istruzioni bytecodes tramite un opportuno interprete. Ovvero la JVM traduce i bytecodes nelle istruzioni macchina della CPU del computer reale sul quale si vuole eseguire il programma Java. Quindi per poter eseguire un programma Java su un certo sistema hardware e software (ad esempio, un PC Pentium Intel con Mac OS X) è necessario vi sia installata una specifica JVM (capace di eseguire la traduzione per quel sistema).

A questo punto ci si potrebbe chiedere quali sono i vantaggi di questo passaggio per un linguaggio

intermedio (i bytecodes) rispetto ad un comune linguaggio interpretato. A prima vista questo sembra non avere altro effetto che rendere inutilmente più complicata l'esecuzione di un programma. In realtà, invece, fornisce parecchi vantaggi. Un primo vantaggio sta nel fatto che l'interprete per il linguaggio dei bytecodes è molto più efficiente di un interprete per un linguaggio ad alto livello come lo è Java stesso. Questo perché il linguaggio dei bytecodes è molto più vicino ad un linguaggio macchina di una CPU reale di quanto lo sia un qualsiasi linguaggio ad alto livello. Un secondo vantaggio proviene dall'indipendenza che il linguaggio intermedio (i bytecodes) introduce tra il linguaggio Java e le diverse architetture hardware. Ad esempio, il linguaggio Java può essere modificato o esteso senza necessariamente modificare il linguaggio dei bytecodes e le relative JVM (e questo è accaduto nel corso delle varie revisioni del linguaggio). Altri vantaggi, non meno importanti, riguardano la sicurezza.

Il linguaggio intermedio dei bytecodes e le relative JVM costituiscono solamente una parte del mosaico che rende il linguaggio Java uno dei linguaggi di programmazione più portabili. L'altra parte consiste nella cosidetta piattaforma Java.

La piattaforma

Per *piattaforma Java* (*Java platform*) si intende l'insieme di tutte quelle librerie predefinite che sono disponibili in ogni installazione Java e che quindi possono essere usate da un qualsiasi programma scritto in Java. Di solito con il termine piattaforma si intende l'insieme delle API (Application Programming Interface) che sono a diposizione di un programmatore su uno specifico sistema. Queste dipendono e sono invero definite dal sistema operativo (Linux, Mac OS X, Solaris, ecc.). Ora Java non è un sistema operativo però l'ampiezza e la profondità delle API messe a disposizione dalle librerie Java (la piattaforma Java appunto) sono confrontabili con quelle definite da un sistema operativo. Così un programmatore può scrivere interamente in Java delle applicazioni senza sacrificare quelle funzionalità avanzate che sarebbero normalmente disponibili solo ad applicazioni native scritte per uno specifico sistema operativo. Una applicazione scritta per la piattaforma Java può essere eseguita senza cambiamenti su un qualsiasi sistema operativo che supporta la piattaforma Java. Così lo stesso programma Java può essere eseguito su una grande varietà di sistemi operativi (Microsoft Windows, Mac OS X, Linux, Solaris). Questo è sintetizzato dal motto di Sun per Java: "write once, run anywhere".

Confronto con altri linguaggi

Prima di tutto, il confronto si impone con i due antenati diretti di Java: C e C++. Il linguaggio C è anche l'antenato del C++ e tutto quello che Java eredita dal C lo eredita direttamente dal C++. Inoltre Java riprende molte altre caratteristiche proprie del C++. Basti pensare che Java è un linguaggio orientato agli oggetti come lo è il C++ mentre il C è invece un linguaggio procedurale. Non è troppo impreciso dire che Java è una versione semplificata del C++. In effetti, due aspetti tra i più ostici del C++ sono stati risolti in Java. Il primo riguarda le complicanze della gestione diretta della memoria tramite i puntatori: la gestione automatica della memoria (*garbage collection*) di Java elimina tali problemi. Il secondo riguarda le difficoltà relative all'ereditarietà multipla del C++ che in Java è stata "risolta" sostituendola con un meccanismo più debole ma più affidabile.

Da quanto detto si potrebbe pensare che Java è una specie di sotto linguaggio del C++, ma in realtà non è così. Il linguaggio Java dispone di un supporto diretto per la programmazione concorrente che non ha una controparte nel C++.

Nel passato ci si poteva lamentare e a ragione della relativa lentezza di Java rispetto al C/C++. Ma nel corso degli anni gli interpreti Java sono stati notevolemente migliorati. Le attuali JVM sono altamente ottimizzate e includono una tecnologia chiamata *Just In Time compiler* (*JIT*) che compila al volo (cioè durante l'esecuzione del programma) parti di codice che sono eseguite ripetutamente. In alcuni casi il compilatore JIT può produrre un codice più efficiente di quello prodotto da un compilatore tradizionale perché può sfruttare le statistiche mantenute durante l'esecuzione del programma. Questa tecnologia insieme con il fatto che una parte strategica della piattaforma Java è direttamente implementata in codice nativo, fa sì che generalmente un programma Java non è significativamente più lento di un corrispondente programma scritto in C/C++.

Esistono altri linguaggi simili a Java. Sicuramente quello che più di tutti è simile a Java è il C#. Però questo linguaggio a differenza di Java, e anche del C e del C++, non è disponibile per sistemi operativi diversi da Windows.

Le basi procedurali di Java

Presentare ogni aspetto del linguaggio Java fino ad un adeguato livello di approfondimento, porta inevitabilmente a posticipare molti argomenti importanti e a frammentare e ritardare una visione d'assieme del linguaggio. E questo è tanto più vero per un linguaggio come Java che è molto più complesso di un linguaggio come il C. Per questa ragione, dapprima faremo un tour delle principali caratteristiche del linguaggio e poi ritorneremo su quelle che necessitano di un adeguato approfondimento. Diamo per scontata una buona conoscenza del linguaggio C e quindi non ci soffermeremo più dello stretto necessario sulle caratteristiche di Java che derivano direttamente da tale linguaggio.

Il primo programma

In Java un programma è composto da *classi*. Per ora, una classe può essere vista come una `struct` del C in cui però oltre ai *campi* è possibile definire anche delle funzioni che in Java sono chiamate *metodi*. Come in C la definizione di una `struct` può essere usata solamente allocando i corrispondenti elementi così in Java una classe per poter essere usata deve essere istanziata in *oggetti*. Tuttavia, in Java, come vedremo presto, una classe può essere usata anche senza che venga istanziata. Anzi i primi esempi riguarderanno proprio programmi che usano una classe senza istanziarla.

```
public class Primo {
    public static void main(String[] args) {
        System.out.println("Primo programma Java");
    }
}
```

L'effetto di questo programma è semplicemente quello di stampare a video la stringa `"Primo programma Java"`. In grassetto sono state evidenziate le parole chiave di Java. La parole chiave `class` inizia la definizione di una classe. Questa e poi seguita dal nome della classe, in questo caso `Primo`. Poi tra parentesi graffe è definito il corpo della classe, cioè tutti i suoi membri (campi e metodi). In questo caso, c'è un solo metodo ed è un metodo speciale perchè può essere visto come il corrispettivo in Java della funzione `main` del C. Infatti, l'esecuzione di un programma Java inizia sempre eseguendo il metodo `main` di una classe. Il metodo `main` come qualsiasi altro metodo è definito dichiarando una *intestazione* (*method header*) e un *corpo* (*method body*) racchiuso tra parentesi graffe. L'intestazione a sua volta comprende, nell'ordine, degli eventuali *modificatori* (*modifiers*), in questo caso `public` e `static`, il tipo del valore ritornato (`void`), il nome del metodo (`main`) e la lista dei *parametri* (`String[] args`). Il metodo main essendo speciale deve sempre avere l'intestazione che abbiamo visto. Vedremo in seguito il significato dei modificatori e dei parametri.

Il corpo del main, in questo caso, contiene solamente la *invocazione* di un metodo. Si noti che abbiamo usato il termine "invocazione" per indicare ciò che in C corrisponderebbe alla chiamata di una funzione. Infatti questo è il termine che si usa in Java. Il metodo invocato è `println()` che appartiene all'oggetto `out` che a sua volta è un campo della classe `System`. La classe `System` è una delle tantissime classi predefinite della piattaforma Java. Per adesso basti dire che l'effetto di `System.out.println("Primo programma Java")` è perfettamente simile a quello di `printf("Primo programma Java\n")` in C. Si noti anche che in Java si usa lo stesso operatore di selezione `"."` del C per accedere ai campi e ai metodi di una classe o di un oggetto.

In Java è richiesto che il file in cui è scritta una classe abbia lo stesso nome della classe. Se il nome della classe è `NomeClasse` allora il file deve chiamarsi `NomeClasse.java`. Così nel nostro caso il file che contiene la definizione della classe `Primo` deve chiamarsi `Primo.java`. Attenzione a rispettare le

maiuscule e minuscole perchè Java è un linguaggio sensibile a questa differenza in tutti i contesti: parole chiave, nomi di variabili, classi, metodi, file, ecc. Quindi tutti i file che contengono codice sorgente in Java devono avere l'estensione `.java` e il loro nome deve essere uguale al nome della classe definita nel file. Più precisamente, in un file `.java` può essere definita una sola classe pubblica (identificata appunto dal modificatore `public`), però può contenere anche la definizione di altre classi non pubbliche.

Tipi primitivi, stringhe, variabili e operatori

I tipi primitivi di Java sono simili a quelli del C ma con importanti differenze. La seguente tabella descrive i tipi primitivi di Java:

`boolean`	`true` o `false`
`char`	carattere 16-bits Unicode UTF-16 (senza segno)
`byte`	intero da 8 bits: da -128 a 127
`short`	intero da 16 bits: da -32768 a 32767
`int`	intero da 32 bits: da -2147483648 a 2147483647
`long`	intero da 64 bits: da -9223372036854775808 a 9223372036854775807
`float`	numero in virgola mobile da 32 bits (IEEE 754)
`double`	numero in virgola mobile da 64 bits (IEEE 754)

I tipi numerici `byte,short,int,long,float,double` sono molto simili a quelli del C. Il tipo `char` può essere visto come una estensione del corrispondente tipo del C e ne discuteremo fra poco. La dichiarazione delle variabili e la loro inizializzazione ricalca la sintassi del C. Ecco alcune dichiarazioni ed inizializzazioni:

```
byte interopiccolissimo = -2;
short interopiccolo = 50;
int interogrande = 120000;
long interograndissimo = 345000000000000;
```

Come in C il simbolo "`=`" rappresenta l'assegnamento e il "`;`" termina ogni istruzione (*statement*). Anche gli operatori sono gli stessi del C. Ad esempio, il seguente frammento di programma Java calcola gli interessi maturati in un investimento di 1000 euro per 5 anni al tasso annuo del 4%:

```
int capitaleIniziale = 1000;        //capitale iniziale
double tasso = 1.04, tassoComposto5;
    // il tasso composto per 5 anni e' il tasso annuale elevato alla quinta potenza
tassoComposto5 = tasso*tasso;
tassoComposto5 *= tassoComposto5;
tassoComposto5 *= tasso;
double capitaleFinale = capitaleIniziale*tassoComposto5;
double interessi = capitaleFinale - capitaleIniziale;
```

Come si intuisce da questo esempio gli operatori aritmetici `+,-,*,/,%` hanno lo stesso significato che hanno nel C, incluse le forme con assegnamento `+=,-=,*=,/=,%=` e gli operatori `++,--` di incremento e decremento. Anche i commenti si scrivono nello stesso modo: `//` per quelli su una singola linea e `/* ... */` per quelli che possono occupare più linee. Inoltre le conversioni automatiche tra i tipi numerici seguono regole simili a quelle del C.

Il tipo `char` è differente dall'omonimo del C. Infatti è in grado di rappresentare oltre ai tradizionali caratteri ASCII anche l'insieme molto più vasto dei caratteri *Unicode*. Le costanti carattere, come in C, sono racchiuse tra apici singoli. Ad esempio `'A','a','0','w','@'` rappresentano i corrispondenti caratteri. Inoltre, possono anche essere usate le *Unicode escape sequences*. Queste sono sequenze del tipo `\uxxxx` dove `xxxx` è un intero a 16 bits scritto in esadecimale. Ad esempio, `'\u0041'` è equivalente ad `'A'`, `'\u03C0'` è il carattere pi greco minuscolo. Per informazioni complete sui codici Unicode si può consultare il sito: http://www.unicode.org/. Oltre alle Unicode escape sequences che permettono di definire tutti i caratteri rappresentabili, è possibile usare anche delle sequenze di escape simili a quelle del C: `\b` (backspace), `\t` (tab), `\n` (line feed), `\f` (form feed), `\r` (carriage return), `\"` (double quote), `\'`

(single quote), and \\ (backslash).

Stringhe Oltre ai tipi primitivi Java fornisce un supporto speciale per le stringhe. Le stringhe sono rappresentate dalla classe predefinita `String` e, concettualmente, sono sequenze di `char`, cioè, sequenze di caratteri Unicode. Una costante stringa, come nel C, è rappresentata da una sequenza di caratteri tra doppi apici in cui possono essere anche usate sequenze di escape. Ad esempio, `"Java\u2122"` è la stringa `Java`™ (dove ™ è il singolo carattere che significa trademark). Sulle stringhe avremo modo di ritornarci molto spesso, per adesso mostriamo uno dei punti salienti del supporto di Java per le stringhe. L'operatore + quando è usato insieme a stringhe ne produce la concatenazione. Ad esempio,

```
String str1 = "Prima parte";
String str2 = "Seconda parte";
String messaggio = str1 + str2;
```

l'oggetto stringa `messaggio` sarà uguale a `"Prima parteSeconda parte"`, cioè la concatenazione degli oggetti stringa `str1` e `str2`. Inoltre, quando si concatena una stringa con un oggetto che non è una stringa quest'ultimo è convertito in una stringa (vedremo poi che ogni oggetto può essere convertito in una stringa):

```
String domanda = "Quanto sei alto?";
int altezza = 174;
Stringa risposta = domanda + " " + altezza + " cm";
```

nell'oggetto stringa `risposta` ci sarà la stringa `"Quanto sei alto? 174 cm"`. Tra i tantissimi metodi degli oggetti stringa, per il momento, mettiamo in evidenza solamente il metodo `length()` che ritorna la lunghezza della stringa e `charAt(int index)` che ritorna il carattere nella posizione `index` (le posizioni iniziano da `0`).

Funzioni matematiche Il linguaggio Java al pari del C non fornisce operatori o funzioni per il calcolo di funzioni matematiche di uso abbastanza comune come la radice quadrata o l'elevamento a potenza. Però la piattaforma Java contiene la classe `Math` che fornisce queste funzioni e molte altre. Ecco alcune delle funzioni e delle costanti messe a disposizione dalla classe `Math`:

```
double log10(double x)            // ritorna il logaritmo in base 10 di x
double sqrt(double x)             // ritorna la radice quadrata di x
double pow(double b, double x)    // ritorna b elevato alla potenza x
double random()       // ritorna un valore pseudo-casuale compreso tra 0.0 e 1.0
double E              // la costante e (base dei logaritmi naturali)
double PI             // la costante pi-greco
```

Consideriamo come esempio un programma che calcola una approssimazione del valore di pi-greco generando un gran numero di punti "casuali" nel quadrato di lato 2 e contando la frazione di questi che cadono nel cerchio di raggio unitario inscritto nel quadrato (si ricordi che l'area del cerchio di raggio unitario è proprio uguale a pi-greco). Il programma usa il metodo `Math.random()` per generare le coordinate dei punti "casuali". Inoltre, definisce un metodo statico per il calcolo della distanza di due punti (serve per determinare se un punto cade nel cerchio calcolando la sua distanza dal centro del cerchio).

```
public class PIgreco {
        // metodo statico per calcolare la distanza tra due punti
    public static double distanza(double x1, double y1, double x2, double y2) {
        double dx = Math.pow(x2 - x1, 2);
        double dy = Math.pow(y2 - y1, 2);
        return Math.sqrt(dx + dy);
    }
    public static void main(String[] args) {
        long numeroPunti = 100000;    // numero punti generati
        long puntiIn = 0;        /* mantiene il conteggio dei punti che
                                    cadono nel cerchio di raggio unitario */
        for (int i = 0 ; i < numeroPunti ; i++) {
            double x = 2*Math.random();    // genera le coordinate di un punto
            double y = 2*Math.random();    // "casuale" nel quadrato di lato 2
```

```
            if (distanza(x, y, 1.0, 1.0) <= 1.0)    // controlla se il punto
                puntiIn++;                          // cade nel cerchio unitario
        }
        System.out.println("Il valore di \u03C0 è "+Math.PI);
        double approxPI = (4*(double)puntiIn)/numeroPunti;
        System.out.println("Il valore approssimato è "+approxPI);
    }
}
```

Come si vede il `for`, l'`if` e vari operatori hanno la stessa sintassi e lo stesso significato che hanno nel linguaggio C (ritorneremo su di essi fra poco). L'esecuzione del programma produce il seguente risultato:

```
Il valore di π è 3.141592653589793
Il valore approssimato è 3.14652
```

Input & Output

Le librerie della piattaforma Java forniscono gli strumenti per programmare interfacce utente grafiche, *GUI* (*Graphical User Interface*), di tutti i generi da quelle più semplici a quelle più ricche e sofisticate. Però l'uso di tali strumenti richiede una conoscenza del linguaggio Java piuttosto approfondita. Almeno per il momento, dovremmo accontentarci dell'input/output forniti dalla cara e vecchia console. Per l'output abbiamo già incontrato `System.out.println()` che permette di stampare sullo "*standard output stream*" (cioè, la finestra della console). Per l'input, cioè, la lettura dallo "*standard input stream*", la situazione non è così semplice. L'analogo per l'input di `System.out` è `System.in` ma quest'ultimo oggetto (che per la cronaca è di tipo `InputStream`) permette di leggere dallo standard input solamente al livello dei bytes. Si può quindi intuire che se usassimo direttamente `System.in` per leggere, ad esempio, un numero o una stringa dovremmo fare parecchio lavoro per tradurre il flusso di bytes nel corrispondente dato (numero o stringa). Per fortuna, sempre la piattaforma Java, ci fornisce una classe che fa proprio questa traduzione. La classe si chiama `Scanner` e per usarla è sufficiente creare un oggetto di tipo `Scanner` che è "attaccato" al flusso di input:

```
Scanner in = new Scanner(System.in);
```

dell'operatore `new` e di come si costruisce un oggetto ne discuteremo in seguito. Per ora basti dire che questa istruzione crea un oggetto di tipo `Scanner` basato su `System.in` e pone il riferimento a tale oggetto nella variabile `in`. Gli oggetti di tipo `Scanner` hanno vari metodi che permettono di leggere il flusso di input come numeri, parole, linee, ecc. Ad esempio,

```
String linea = in.nextLine();
```

legge la prossima linea dal flusso di input (cioè la sequenza di caratteri fino al prossimo separatore di linea) e la pone in un oggetto stringa. Analogamente il metodo `next()` legge il prossimo token (sequenza di caratteri delimitata da whitespaces) e i metodi `nextInt()` e `nextDouble()` leggono, rispettivamente, il prossimo intero e il prossimo numero in virgola mobile (se presente).

Come esempio consideriamo un programma che calcola l'importo della rata per la restituzione di un prestito avendo fornito in input il capitale, il tasso annuo e il numero complessivo di rate mensili. La rata è calcolata applicando le formule:

$$RATA = CAPITALE*(TM*TC)/(TC - 1)$$
$$TM = (1 + TA/100)^{1/12} - 1$$
$$TC = (1 + TM)^{NR}$$

inoltre TA è il tasso annuo e NR è il numero di rate. Così $100*TM$ risulta essere il tasso su base mensile e $100*(TC - 1)$ è l'interesse composto relativo all'intero periodo di restituzione del prestito.

```
import java.util.*;

public class Rata {
        // metodo statico che calcola il tasso mensile a partire da quello annuo
```

```
    public static double tassoMensile(double ta) {
        return 100*(Math.pow(1 + ta/100, 1.0/12.0) - 1);
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);   // creazione dell'oggetto Scanner
        System.out.print("Capitale: ");
        int capitale = in.nextInt();           // legge l'importo del capitale
        System.out.print("Tasso annuo: ");
        double tassoAnnuo = in.nextDouble();   // legge il tasso annuo
        System.out.print("Numero rate mensili: ");
        int numeroRate = in.nextInt();         // legge il numero di rate
        double tassoMensile = tassoMensile(tassoAnnuo);
        System.out.println("Il tasso mensile è "+tassoMensile+"%");
        double tm = tassoMensile/100;                  // calcola l'importo della
        double tc = Math.pow(1 + tm, numeroRate);      // rata applicando la
        double rata = capitale*((tm*tc)/(tc - 1));     // formula
        System.out.println("L'importo della rata è "+rata);
    }
}
```

La linea `import java.util.*;` è necessaria perché la classe `Scanner` appartiene al package `java.util`. Tutte le volte che si usa una classe che non appartiene al package di base `java.lang` (`System`, `String` e `Math` appartengono a questo package) è necessario dichiarare il package di appartenenza tramite una direttiva `import`. Parleremo più dettagliatamente dei packages e della direttiva `import` in seguito. Una possibie esecuzione del programma produce il seguente risultato:

```
Capitale: 20000
Tasso annuo: 15
Numero rate mensili: 36
Il tasso mensile è 1.171491691985338%
L'importo della rata è 684.1151627734829
```

Controllo del flusso

Tutte le istruzioni di Java per il controllo del flusso in un programma sono riprese da quelle del C, con una sola eccezione che discuteremo più avanti. Quindi Java dispone delle istruzioni `if-else`, `while`, `do-while`, `for` e `switch-case` con la stessa sintassi del C. Vediamo subito alcuni semplici esempi. Il seguente programma prende in input tre numeri e li stampa ordinati in senso cresecente:

```
import java.util.*;

public class Ordine {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Digita tre numeri: ");
        double a = in.nextDouble(), b = in.nextDouble(), c = in.nextDouble();
        String risultato = "I tre numeri ordinati sono ";
        if (a <= b) {
            if (b <= c) risultato += a + " " + b + " " + c;
            else if (a <= c) risultato += a + " " + c + " " + b;
            else risultato += c + " " + a + " " + b;
        } else {
            if (a <= c) risultato += b + " " + a + " " + c;
            else if (b <= c) risultato += b + " " + c + " " + a;
            else risultato += c + " " + b + " " + a;
        }
        System.out.println(risultato);
    }
}
```

Come si intuisce da questo esempio anche tutti gli operatori relazionali `<,<=,>=,>,==,!=` sono uguali a quelli del C. Il prossimo programma prende in input una stringa e conta il numero di vocali presenti nella

stringa:

```
import java.util.*;

public class Vocali {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Digita una stringa: ");
        String str = in.nextLine();
        int n = str.length(), numVocali = 0;
        for (int i = 0 ; i < n ; i++) {
            char c = str.charAt(i);
            if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u')
                numVocali++;
        }
        System.out.println("Numero vocali: " + numVocali);
    }
}
```

Anche gli operatori logici `&&`, `||` e `!` sono gli stessi del linguaggio C. Il programma che segue prende in input due parole e calcola la lunghezza del più lungo prefisso comune alle due parole:

```
import java.util.*;

public class Prefisso {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Digita due parole: ");
        String word1 = in.next(), word2 = in.next();
        int n1 = word1.length(), n2 = word2.length();
        int p = 0;
        while (p < n1 && p < n2 && word1.charAt(p) == word2.charAt(p))
            p++;
        System.out.println("Lunghezza prefisso comune: " + p);
    }
}
```

Il prossimo programma prende in input una serie di interi positivi (terminata non appena viene immesso un numero negativo) e ne calcola la media:

```
import java.util.*;

public class Media {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        int n, sum = 0, count = 0;
        do {
            System.out.println("prossimo numero (-1 per terminare): ");
            n = in.nextInt();
            if (n >= 0) {
                sum += n;
                count++;
            }
        } while (n >= 0) ;
        double average = (count > 0 ? ((double)sum)/count : 0);
        System.out.println("La media è " + average);
    }
}
```

Come si vede da questo esempio anche l'operatore condizionale `( ? : )` e i cast sono presenti in Java. Il prossimo programma usa lo `switch-case`:

```
import java.util.*;

public class SwitchTest {
    public static void main(String[] args) {
```

```
        Scanner in = new Scanner(System.in);
        int n = in.nextInt();
        String msg;
        switch (n) {
            case 1:
                msg = "Hai digitato 1";
                break;
            case 2:
                msg = "Hai digitato 2";
                break;
            case 3:
                msg = "Hai digitato 3";
                break;
            default:
                msg = "Hai digitato qualcosa di diverso da 1,2,3";
        }
        System.out.println(msg);
    }
}
```

Ovviamente, ritroveremo tutti questi costrutti per il controllo del flusso molto spesso nel seguito usati in esempi ed esercizi. Inoltre, anche in Java è possibile scrivere metodi ricorsivi. Ecco un semplice programma che implementa un metodo ricorsivo per il calcolo del fattoriale:

```
import java.util.*;

public class Fattoriale {
    public static long fattoriale(int n) {    // metodo ricorsivo
        if (n <= 1) return 1;
        else return n*fattoriale(n - 1);
    }
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.print("Digita un intero: ");
        int n = in.nextInt();
        System.out.println("Il fattoriale di "+n+" è "+fattoriale(n));
    }
}
```

Esercizi

In alcuni esercizi può essere utile usare il metodo `print()` che è uguale a `println()` eccetto che non va a capo. Inoltre, si tenga presente che entrambi i metodi accettano come parametro anche un singolo `char`.

[Stringa_verticale] Scrivere un programma che legge una stringa (cioè una linea di testo) e la stampa in verticale. Ad esempio, se la stringa letta è `"gioco"` allora il programma stampa:

```
g
i
o
c
o
```

[Parole_verticali] Scrivere un programma che legge tre parole e le stampa in verticale l'una accanto all'altra. Ad esempio, se le parole sono `"gioco"`, `"OCA"` e `"casa"` allora il programma stampa:

```
gOc
iCa
oAs
c a
o
```

[Vocali] Scrivere un programma che legge una linea di testo e per ogni vocale stampa il numero di volte che appare nella linea di testo. Ad esempio, se la linea di testo è `"mi illumino di immenso"` allora il programma stampa:

```
a: 0      e: 1      i: 5      o: 2      u: 1
```

[Tre_più_grandi] Scrivere un programma che legge una serie di numeri interi positivi (la lettura si interrompe quando è letto un numero negativo) e stampa i tre numeri più grandi della serie. Ad esempio, se la serie di numeri è `2, 10, 8, 7, 1, 12, 2` allora il programma stampa:

```
I tre numeri più grandi sono: 12, 10, 8
```

[Cornice] Scrivere un programma che legge un intero `n` e stampa una cornice quadrata di lato `n` fatta di caratteri `'*'`. Ad esempio, se `n = 5`, il programma stampa:

```
*****
*   *
*   *
*   *
*****
```

[Triple_Pitagoriche] Una *tripla pitagorica* è una tripla di numeri interi a, b, c tali che $1 \le a \le b \le c$ e $a^2 + b^2 = c^2$. Ciò equivale a dire che a, b, c sono le misure dei lati di un triangolo rettangolo (da qui il nome). Scrivere un programmma che legge un intero M e stampa tutte le triple pitagoriche con $c \le M$.

[Pi_greco] Scrivere un programma che letto un intero k stampa la somma dei primi k termini della serie

$$4 - 4/3 + 4/5 - 4/7 + 4/9 - 4/11 + ...$$

La serie converge al numero pi greco. Quanto deve essere grande k per ottenere le prime 8 cifre decimali corrette (3.14159265)?

[Cifre->lettere] Scrivere un programma che legge un intero `n` e stampa le cifre di `n` in lettere. Ad esempio, se `n = 2127`, il programma stampa: `due uno due sette`.

[Lettere->cifre] Scrivere un programma che esegue la trasformazione inversa di quella del programma precedente. Letta una linea di testo, se questa è composta di parole rappresentanti numeri da 1 a 9, stampa il numero corrispondente. Ad esempio, se legge `"due uno due sette"` allora stampa `2127`.

[Numeri_perfetti] Un *numero perfetto* è un numero intero che è uguale alla somma dei suoi divisori propri, ad esempio 6 è perfetto perché 6 = 1 + 2 + 3, mentre 8 non è perfetto dato che 1 + 2 + 4 non fa 8. Scrivere un programma che letto un intero M stampa tutti i numeri perfetti minori od uguali a M e le relative somme dei divisori. Ad esempio se M = 1000 il programma stampa:

```
6 = 1 + 2 + 3
28 = 1 + 2 + 4 + 7 + 14
496 = 1 + 2 + 4 + 8 + 16 + 31 + 62 + 124 + 248
```

[Sostituzione] Scrivere un programma che legge una linea di testo e se questa contiene la parola `"mille"` allora stampa la linea di testo in cui però tutte le occorrenze della parola `"mille"` sono sostituite con la parola `"cento"`. Ad esempio, se la linea di testo è `"mille non più mille"` allora il programma stampa: `"cento non più cento"`.

[Monete] Scrivere un programma che letto un numero intero rappresentante un importo in centesimi di euro stampa le monete da 50, 20, 10, 5, 2 e 1 centesimi di euro che servono per fare l'importo. Ad esempio, se l'importo è di 97 centesimi allora il programma stampa:

```
1 moneta da 50
2 monete da 20
1 moneta da 5
1 moneta da 2
```

[Fattori_primi] La scomposizione in fattori primi di un numero n è l'elenco dei fattori primi di n con le loro molteplicità. Si ricorda che un fattore primo di n è un divisore di n che è un numero primo (un numero è primo se non ha divisori propri). Scrivere un programma che legge un numero intero n e stampa la scomposizione in fattori primi di n. Il comportamento del programma per alcuni valori di n è il seguente:

```
n                       stampa del programma:
6                       2(1) 3(1)
7                       7(1)
362880                  2(7) 3(4) 5(1) 7(1)
1234567890              2(1) 3(2) 5(1) 3607(1) 3803(1)
```

[Palindrome] Una *palindroma* è una parola o stringa che rimane la stessa se letta da sinistra verso destra o al contrario. Scrivere un metodo ricorsivo che presa in input una stringa determina se è una palindroma. Scrivere anche un programma che mette alla prova il metodo.
[Frasi_palindrome] Una *frase palindroma* è una sequenza di parole che rimane uguale se la sequenza è letta da destra verso sinistra (senza rovesciare le parole). Ad esempio la frase `"libero è chi è libero"` è una frase palindroma. Scrivere un metodo ricorsivo che presa in input una stringa determina se è una frase palindroma. Può essere utile il metodo

```
String substring(int beginIndex, int endIndex)
```

della classe `String` che ritorna la sottostringa, della stringa su cui è invocato, che inizia all'indice `beginIndex` e comprende tutti caratteri fino all'indice `endIndex - 1`. Scrivere anche un programma che mette alla prova il metodo.
[Potenze] Scrivere un metodo `integerPow(double b, int k)` che in modo ricorsivo calcola `b` elevato alla potenza intera positiva `k`. Scrivere un programma che confronta il metodo scritto con il metodo `Math.pow()`.
[Monete_sbilanciate] Scrivere un metodo `boolean biasedCoin(float p)` che, sfruttando `Math.random()`, simula il lancio di una moneta sbilanciata: con probablità `p` ritorna `true` (testa) e con probabilità `1 - p` ritorna `false` (croce). Si assuma che `p` è sempre compreso tra `0.0` e `1.0`. Scrivere un programma che mette alla prova il metodo.

Struttura di un programma Java

Un programma Java consiste nella definizione di uno o più tipi. Finora non abbiamo spiegato cosa sono i tipi in Java eccezion fatta per i tipi primitivi. Abbiamo accennato alle classi ma queste non sono state ancora spiegate. Le classi rappresentano, come vedremo presto, il meccanismo più importante per definire tipi in Java ma ce ne sono anche altri. La piattaforma Java fornisce migliaia di tipi predefiniti (la maggior parte classi) che svolgono una funzione analoga a quella svolta dalla libreria standard per il linguaggio C. Tuttavia, non ve dubbio che la piattaforma Java è enormemente più ampia della libreria standard del C. Per organizzare sistemi così vasti e complessi è necessario un meccanismo per raggruppare tipi che sono tra loro legati e per evitare collisioni tra nomi di tipi. Per questo Java permette di raggruppare collezioni di tipi tramite i *packages*. Ogni package è identificato da un nome. Per dichiarare che un tipo appartiene ad un package occorre iniziare il file che contiene la definizione del tipo con una direttiva come quella qui riportata:

```
package nomePackage;
```

Se nel file non c'è una direttiva che specifica un package, allora i tipi definiti nel file appartengono ad un unico package di default senza nome. Ogni tipo ha un nome semplice che è il nome che gli è stato assegnato nella sua definizione e un nome completo che include il nome del package a cui appartiene. Ad esempio, la classe della piattaforma Java che rappresenta le stringhe ha nome `String` e siccome appartiene al package `java.lang` il suo nome completo è `java.lang.String`. La classe il cui nome è `Scanner` appartiene al package `java.util`, così il suo nome completo è `java.util.Scanner`. La direttiva `import` permette di usare il nome semplice di un tipo al posto del nome completo che spesso è molto lungo.

Il codice di un programma Java è scritto in uno o più file. Ogni file deve avere la seguente struttura:

```
<una direttiva opzionale che specifica il nome del package a cui i tipi
    definiti in questo file appartengono>
<zero o più direttive di import>
<una o più definizioni di tipi (classi o di altro genere)>
```

Questi elementi devono apparire nell'ordine dato.

Classi e oggetti

Finora abbiamo visto quegli aspetti di Java che non sono dissimili da quelli di un qualsiasi linguaggio di programmazione procedurale. Adesso iniziamo a entrare nel vivo del linguaggio Java, considerando quelle caratteristiche che lo rendono un linguaggio orientato agli oggetti.

Orientamento agli oggetti

Cosa significa dire che un linguaggio è "orientato agli oggetti"? Per rispondere a questa domanda conviene fare un passo indietro e ricordarsi qual'è l'obbiettivo di un linguaggio di programmazione. Il principale obbiettivo è rendere facile la vita dei programmatori in tutte quelle fasi dello sviluppo del software in cui il linguaggio di programmazione usato riveste un ruolo importante (ad esempio, nella progettazione, nella scrittura del codice, nel debugging, nel testing e nella manutenzione del codice). E come fa un linguaggio a tentare di raggiungere questo obbiettivo? Cercando di gestire al meglio la complessità che è intrinseca in un qualsiasi sistema software di dimensioni non banali. E c'è, essenzialmente, un solo modo per fronteggiare la complessità: cercare di decomporre l'intero in parti meno complesse. I diversi linguaggi di programmazione usano filosofie e meccanismi differenti per aiutare i programmatori ad attuare questa strategia.

I linguaggi procedurali, come ad esempio il C, offrono pochi mezzi: funzioni o procedure e la possibilità di costruire nuovi tipi aggregando altri tipi (ad esempio, tramite le `struct`). Una procedura permette di isolare una parte del sistema software dal resto. Così che il resto del sistema può disinteressarsi di come è fatta la procedura al suo interno e considerare solamente ciò che serve per poter usare la procedura (la sua interfaccia). Questo riduce la complessità riducendo il numero delle potenziali relazioni fra le varie parti del sistema. Sostanzialmente, è il principio dell'*information hiding* o *incapsulamento*: suddividere il sistema in parti in modo tale che le loro interazioni si possano definire in base a semplici interfacce che risultano indipendenti da come le parti sono implementate. Questo principio è così consolidato e naturale che ormai è quasi dato per scontato. Oltre al principio dell'incapsulamento ci sono altri aspetti di un linguaggio che possono aiutare a fronteggiare la complessità anche se non sono altrettanto importanti. La possibilità di costruire nuovi tipi tramite semplice aggregazione di altri tipi aiuta a ridurre la complessità tramite la diminuzione della distanza tra la natura delle informazioni reali e la loro rappresentazione nel sistema. Questo a sua volta migliora la leggibilità del codice e quindi anche la capacità di modificare ed estendere le funzionalità del sistema.

I linguaggi orientati agli oggetti come Java offrono mezzi più sofisticati per fronteggiare la complessità del software. Uno dei più importanti trae la sua forza dalla combinazione delle due caratteristiche sopra menzionate. Infatti una *classe* può essere vista, in prima approssimazione, come una `struct` che oltre ad avere campi ha anche delle funzioni che in Java si chiamano *metodi*. Così una classe è più efficace nell'isolare una parte del sistema software perché può comprendere sia procedure sia dati che sono intimamente connesse le une agli altri. Ad esempio, in un sistema software per la gestione dei dati del personale di una azienda, ci potrebbe essere una classe `Impiegato` che serve a rappresentare e manipolare i dati relativi agli impiegati. Questa classe conterrà, oltre ai soliti campi (`nome`, `cognome`, `data_di_nascita`, ecc.), anche dei metodi: un metodo `età()` che calcola l'età attuale, un metodo `stipendio()` che calcola la busta paga, un metodo `stampa()` per la stampa formattata dei dati dell'impiegato, ecc. Ogni istanza della classe `Impiegato`, che in Java si chiama *oggetto*, rappresenta uno specifico impiegato. Complessivamente i valori dei campi di un oggetto sono lo *stato* di quell'oggetto. Il comportamento di un oggetto, cioè il risultato dell'invocazione di un qualsiasi metodo relativamente a quell'oggetto, dipende dallo stato dell'oggetto. Così, se `rossi` è il riferimento all'oggetto che rappresenta l'impiegato Mario Rossi, l'invocazione del metodo `rossi.età()` ritorna proprio l'età di Mario Rossi, così come `verdi.età()` ritorna invece l'età di Giuseppe Verdi, se `verdi` è un riferimento all'oggetto che rappresenta l'impiegato Giuseppe Verdi.

Tutto questo, oltre ad ampliare le possibilità di applicazione del principio dell'information hiding, permette anche di migliorare la leggibilità e sopratutto la riusabilità del codice. La riusabilità è un aspetto

di grandissima importanza per rendere l'attività della programmazione più proficua ed efficiente. Quando infatti la strategia della programmazione orientata agli oggetti è applicata alla realizzazione di librerie software (ad esempio, manipolazione di stringhe, gestione di collezioni di elementi, accesso a file, ecc.) mostra tutta la sua forza realizzando strumenti di uso generale che possono essere usati e riusati in tantissime situazioni. Il successo e la continua crescita della piattaforma Java ne è una solida prova.

Nel linguaggio Java, al pari degli altri linguaggi orientati agli oggetti, il meccanismo base delle classi e degli oggetti è coadiuvato da altri meccanismi. Tra i più importanti c'è il meccanismo dell'*ereditarietà* che consente di estendere in modo naturale una classe (cioè, modificare o aggiungere funzionalità) per definire altre classi. Questo meccanismo a sua volta permette il *polimorfismo* che è molto utile per trattare in modo uniforme le funzionalità di oggetti appartenenti a classi differenti. E poi ci sono la *genericità* e l'*overloading*.

Classi e oggetti

Per il momento vedremo solamente la versione base della definizione di una classe. Poi, mano a mano, avremo modo di introdurre tutte le altre caratteristiche. Lo schema semplificato della definizione di una classe pubblica può essere descritto così:

```
public class NomeClasse {
    dichiarazioni di campi
    dichiarazioni di costruttori
    dichiarazioni di metodi
}
```

Il modificatore di accesso (*access modifier*) `public` indica proprio che la classe è pubblica, cioè è visibile e quindi accessibile da qualsiasi altra parte del programma. Non c'è nessun vincolo sull'ordine con cui sono elencate le dichiarazioni all'interno del corpo della classe. Di solito però sono disposte in quell'ordine.

Campi Con il termine *campo* si intende una variabile che appartiene ad una classe e la dichiarazione di un campo è simile alle dichiarazioni di variabili che abbiamo già incontrato. Però può essere preceduta da dei modificatori, tra questi quelli che vedremo subito sono i modificatori di accesso `public` e `private`. Ad esempio,

```
public double valore;
private int status;
int codice;
```

la prima dichiarazione riguarda una variabile `valore` di tipo `double` che è pubblica, cioè qualsiasi parte del programma, anche al di fuori della classe e del package della classe, può accedere al campo `valore`, cioè può leggerlo o scriverlo. Mentre la variabile `status` essendo dichiarata privata è accessibile solamente dall'interno della classe in cui è definita. La variabile `codice`, non avendo alcun modificatore di accesso specificato, è accessibile solamente dall'interno del package a cui appartiene la classe. Di solito i campi di una classe sono dichiarati privati per evitare che dall'esterno della classe si possa modificarne i valori senza che questo sia controllato dalla classe. Quindi l'uso del modificatore `private` aiuta l'applicazione del principio dell'incapsulamento.

Metodi Un *metodo* è una funzione che appartiene ad una classe. La dichiarazione di un metodo rispetta il seguente schema semplificato che comprende una intestazione e un corpo:

```
modificatori tipo-ritornato nomeMetodo(lista-parametri) {
    corpo-del-metodo
}
```

L'*intestazione* del metodo è formata da uno o più modificatori, il nome *tipo-ritornato* del tipo del valore ritornato, il nome del metodo e la lista dei parametri. Se il metodo non ritorna alcun valore allora il *tipo-ritornato* è `void`, come nel C. La lista dei parametri può essere vuota ed è simile alla lista dei parametri di una funzione del C. Inoltre, il passaggio dei parametri è, come nel C, per valore. La *signature* (firma)

del metodo è il nome del metodo e la lista dei parametri (intesa come lista dei tipi dei parametri). In Java, a differenza del C, possono essere definiti diversi metodi, appartenenti alla stessa classe, con lo stesso nome, purché abbiano differenti signature. Questa caratteristica è chiamata *overloading* e ne vedremo presto degli esempi. Per quanto riguarda i modificatori, per adesso, ci limitiamo a considerare solamente i modificatori di accesso `public` e `private`. Il loro significato è del tutto simile a quello che abbiamo già visto per i campi. Ecco due semplici esempi:

```
public double square(double x) {
    return x*x;
}
private int compute(int a, int b) {
    int c = (a + b)*(a - b);
    return (c > 3 ? c : 0);
}
```

Il primo metodo `square` dichiarato con il modificatore `public` è un metodo pubblico che può essere invocato da qualsiasi parte del programma. Mentre il metodo `compute`, essendo dichiarato con il modificatore `private`, è privato e può essere invocato solamente dall'interno della classe in cui è definito. I metodi la cui dichiarazione non specifica alcun modificatore di accesso possono essere invocati solamente dall'interno del package a cui appartiene la classe. Generalmente i metodi pubblici di una classe rappresentano l'interfaccia della classe, cioè le funzionalità e i servizi offerti dagli oggetti della classe a tutto il resto del programma. Invece i metodi privati svolgono funzioni di utilità all'interno della classe, cioè sono di aiuto alle elaborazioni dei metodi pubblici ma non servono all'esterno della classe.

Costruttori Un *costruttore* è un tipo speciale di metodo che è invocato solamente quando un nuovo oggetto della classe è creato. Il compito di un costruttore è di svolgere tutte quelle elaborazioni e inizializzazioni (dei campi) che sono necessarie affinché l'oggetto appena creato risulti valido. Un esempio di costruttore lo abbiamo già visto per la classe `Scanner`. Anche la sintassi dei costruttori è speciale:

```
modificatori NomeClasse(lista-parametri) {
    corpo-del-costruttore
}
```

Come si vede non c'è un tipo del valore ritornato perchè questo è sempre il tipo dell'oggetto (cioè la classe). Inoltre, il nome del costruttore deve sempre coincidere con il nome della classe. Generalmente l'unico modificatore usato è `public` (solo in rare circostanze è differente). Grazie all'overloading ci possono essere più costruttori per la stessa classe. C'è sempre un costruttore di default (senza parametri) che è invocato solamente quando non c'è ne uno definito che può essere usato. Questo implica che una classe può anche non avere costruttori (definiti).
Come si è detto, un costruttore è invocato solamente quando un nuovo oggetto della classe viene creato. Questo avviene in congiunzione con l'operatore `new`. Ad esempio, per creare un nuovo oggetto della classe *NomeClasse* si può scrivere:

```
NomeClasse nuovoOggetto = new NomeClasse();
```

L'espressione `new NomeClasse()` crea una istanza della classe `NomeClasse` e ritorna il riferimento all'oggetto creato. È importante osservare che la variabile `nuovoOggetto` non conterrà un oggetto di tipo `NomeClasse` ma un riferimento ad un oggetto di quel tipo. Si può immaginare un riferimento ad un oggetto come un puntatore a quell'oggetto (come nel C). Questo punto sarà approfondito più avanti.
A differenza del C, in Java non dobbiamo preoccuparci di rilasciare la memoria allocata per un oggetto. Infatti, ci penserà il *garbage collector* che automaticamente e costantemente durante l'esecuzione del programma rilascia la memoria degli oggetti che non sono più usati dal programma.

Campi e metodi statici Oltre ai campi e i metodi che abbiamo appena visto una classe può avere campi e metodi *statici*. La differenza sta nel fatto che i campi e i metodi (non statici) appartengono agli oggetti della classe mentre i campi e i metodi statici appartengono alla classe. Questo significa che i primi hanno valore e comportamento che dipende dallo specifico oggetto a cui appartengono mentre i

secondi non dipendono da nessun oggetto della classe. I campi e i metodi statici possono essere visti come campi e metodi condivisi da tutti gli oggetti della classe. Ad esempio, un campo statico potrebbe mantenere un valore costante che è uguale per tutti gli oggetti della classe. Esempi di campi statici sono i campi `out` e `in` della classe `System` o il campo `PI` della classe `Math`. Un metodo statico potrebbe essere un metodo che combina in qualche modo due oggetti della classe creandone un terzo oppure un metodo che non ha bisogno dello stato di un oggetto specifico per essere calcolato. Esempi di metodi statici sono tutti i metodi della classe `Math`, come `sqrt()`, `pow()`, ecc. Per dichiarare un campo o un metodo statico si usa il modificatore `static`.

Un esempio Consideriamo come esempio una semplice classe che rappresenta studenti. Ogni oggetto della classe ha tre campi `matricola`, `nome` e `cognome`. Inoltre ha un costruttore e alcuni metodi pubblici. La classe ha anche un campo statico `matricolaCorrente` che serve a mantenere l'ultima matricola usata e un metodo statico privato che produce una nuova matricola. La classe ha anche un metodo statico pubblico che permette di cambiare la matricola di uno studente.

```
public class Studente {
        // dichiarazione e inizializzazione di un campo statico
    private static long matricolaCorrente = 1000000;
        // metodo publico statico
    public static void cambiaMatricola(Studente s) {
       s.matricola = nuovaMatricola();
    }

    private static long nuovaMatricola() {     // metodo privato statico
        matricolaCorrente++;
        return matricolaCorrente;
    }

    private long matricola;          // dichiarazione di campi (non statici)
    private String nome, cognome;

    public Studente(String nome, String cognome) {    // costruttore
        matricola = nuovaMatricola();
        this.nome = nome;
        this.cognome = cognome;
    }

    public String getNome() { return nome; }         // metodi pubblici
    public String getCognome() { return cognome; }
    public long getMatricola() { return matricola; }
    public void stampa() {
        System.out.println("Matricola: " + matricola);
        System.out.println("Cognome: " + cognome + "  Nome: " + nome);
    }
}
```

Si osservi che il metodo statico `cambiaMatricola()` può accedere al campo privato dell'oggetto `Studente` perché appartiene alla stessa classe. Nel costruttore è usata la parola chiave `this` che rappresenta il riferimento all'oggetto stesso. Qui `this` è usato per potersi riferire ai campi `nome` e `cognome` dell'oggetto che altrimenti sarebbero stati mascherati dagli omonimi argomenti del costruttore.
La suddetta classe è usata nel seguente programma.

```
public class Main {
    public static void main(String[] args) {
            // crea due oggetti di tipo Studente
        Studente stu1 = new Studente("Mario", "Rossi");
        Studente stu2 = new Studente("Maria", "Verdi");
        stu1.stampa();          // stampa i dati dei due studenti
        stu2.stampa();
            // cambia la matricola del primo studente
        Studente.cambiaMatricola(stu1);
            // e la stampa
        System.out.println("Nuova matricola: " + stu1.getMatricola());
```

```
    }
}
```

Si noti che i metodi (non statici), come ad esempio `stampa()`, possono essere invocati solamente in relazione ad uno specifico oggetto, in questo caso gli oggetti `stu1` e `stu2` di tipo `Studente`. Mentre i metodi statici, come `cambiaMatricola()`, possono essere invocati solamente in relazione alla classe, proprio perché non appartengono ad alcun oggetto ma appartengono invece alla classe.

La prima classe

Consideriamo una classe, che chiameremo `CharRect`, i cui oggetti rappresentano rettangoli di caratteri che possono essere visualizzati sulla console. Inizialmente la classe sarà molto spartana e permetterà di rappresentare solamente rettangoli riempiti con il carattere `'*'`. Sarà via via raffinata ed ampliata esemplificando nel contempo nuove caratteristiche del linguaggio Java e anche alcune tecniche di progettazione.

La prima versione della classe permette di costruire un nuovo rettangolo fornendo la posizione del suo carattere in alto a sinistra, la larghezza (numero di colonne) e l'altezza (numero di righe). La posizione è data relativamente ad un ipotetico sistema di riferimento che numera le righe dall'alto verso il basso partendo da 0 e le colonne da sinistra verso destra partendo sempre da 0. La classe ha un solo metodo il quale stampa il rettangolo. Ecco una definizione di questa classe:

```
import static java.lang.System.*;

public class CharRect {
    private int left, top;    // posizione del primo crattere in alto a sinistra
    private int width, height;    // dimensioni del rettangolo
        // costruttore
    public CharRect(int l, int t, int w, int h) {
        left = l;
        top = t;
        width = w;
        height = h;
    }

    public void draw() {              // stampa il rettangolo
        for (int i = 0 ; i < top ; i++) out.println();
        for (int r = 0 ; r < height ; r++) {
            int right = left + width;
            for (int c = 0 ; c < right ; c++)
                out.print(c < left ? ' ' : '*');
            out.println();
        }
    }
}
```

La direttiva `import static` permette di "importare" i campi e i metodi statici di una classe. Ovviamente, la classe va scritta in un file di nome `CharRect.java`. I campi sono tutti privati perchè fanno parte dell'implementazione della classe e quindi non dovrebbero essere visibili dall'esterno. Mentre, il costruttore e il metodo `draw()` devono essere pubblici per poter essere invocati liberamente dall'esterno. Il costruttore inizializza i campi che definiscono l'oggetto rettangolo con i valori che saranno forniti al momento della creazione. Quando, come in questo caso, è definito un metodo con almeno un parametro, e non è esplicitamente definito il costruttore senza parametri, il costruttore di default non può essere invocato. Vale a dire, non si può scrivere `new CharRect()`.

Vediamo subito come questa classe può essere usata. Per fare ciò occorre una classe che implementa un metodo `main()`. Definiamo quindi una classe che chiameremo `Test` (in un file di nome `Test.java`):

```
public class Test {
    public static void main(String[] args) {
        CharRect rectA = new CharRect(3, 0, 10, 5);
        CharRect rectB = new CharRect(6, 1, 12, 3);
```

```
        CharRect rectC = new CharRect(10, 1, 4, 4);
        rectA.draw();
        rectB.draw();
        rectC.draw();
        rectA.draw();
    }
}
```

Il risultato dell'esecuzione di questo programma è il seguente:

```
**********
**********
**********
**********
**********

   ************
   ************
   ************

       ****
       ****
       ****
       ****
**********
**********
**********
**********
**********
```

Un miglioramento che possiamo facilmente apportare alla nostra classe è di aggiungere un metodo che permetta di cambiare il carattere usato. Basterà aggiungere un campo e il seguente metodo alla classe `CharRect`:

```
private char fillChar = '*';    // carattere di riempimento
public void setChar(char c) { fillChar = c; }
```

Dobbiamo poi sostituire `'*'` con `fillChar` nel metodo `draw()`. Inoltre, per evidenziare che il carattere `'*'` è il carattere di default ed è una costante condivisa da tutti gli oggetti, conviene introdurre un nuovo campo statico:

```
private static final char DEF_FILLCHAR = '*';
```

Il modificatore `final` indica che la variabile è una costante e il suo valore non può essere modificato. `final` può essere usato in tutte le dichiarazioni di variabli, non solo in quelle statiche. L'inizializzazione della variabile `fillChar` diventa:

```
private char fillChar = DEF_FILLCHAR;
```

Ora vogliamo aggiungere un metodo che stampa il rettangolo a strisce verticali alternate con due caratteri, come il rettangolo qui sotto:

```
*o*o*o*o
*o*o*o*o
*o*o*o*o
*o*o*o*o
```

Chiaramente, occorre definire un'altro carattere. Così aggiungiamo i seguenti campi:

```
private static final char DEF_FILLCHAR2 = 'o';
private char fillChar2 = DEF_FILLCHAR2;
```

Inoltre dobbiamo aggiungere un nuovo metodo che esegue il nuovo tipo di stampa:

```
public void drawVStripes()
```

Per l'implementazione conviene introdurre un metodo ausiliario che stampa una linea del rettangolo e che può essere usato in entrambi i metodi di stampa. La nuova versione della classe è la seguente:

```
import static java.lang.System.*;

public class CharRect {
    private static final char DEF_FILLCHAR = '*';
    private static final char DEF_FILLCHAR2 = 'o';

    private int left, top;
    private int width, height;
    private char fillChar = DEF_FILLCHAR, fillChar2 = DEF_FILLCHAR2;

    public CharRect(int l, int t, int w, int h) {
        left = l;
        top = t;
        width = w;
        height = h;
    }

    public void setChar(char c) { fillChar = c; }

    public void setChar(char c, char c2) { fillChar = c; fillChar2 = c2; }

    public void draw() {
        for (int i = 0 ; i < top ; i++) out.println();
        for (int r = 0 ; r < height ; r++)
            drawLine(fillChar, fillChar);
    }

    public void drawVStripes() {
        for (int i = 0 ; i < top ; i++) out.println();
        for (int r = 0 ; r < height ; r++)
            drawLine(fillChar, fillChar2);
    }
        // metodo ausiliario (privato)
    private void drawLine(char ch1, char ch2) {
        int right = left + width;
        for (int k = 0 ; k < right ; k++) {
            char ch = ' ';
            if (k >= left)
                ch = ((k - left) % 2 == 0 ? ch1 : ch2);
            out.print(ch);
        }
        out.println();
    }
}
```

Il metodo `drawLine()` è privato perché è utile per implementare i metodi pubblici `draw()` e `drawVStripes()` ma non deve essere accessibile dall'esterno della classe. Si osservi che, grazie all'overloading, i due metodi `setChar()` hanno lo stesso nome. Un programma che mette alla prova la nuova versione è il seguente:

```
public class Test {
    public static void main(String[] args) {
        CharRect rectA = new CharRect(3, 0, 10, 5);
        CharRect rectB = new CharRect(6, 1, 12, 3);
        CharRect rectC = new CharRect(10, 1, 4, 4);
        rectA.draw();
        rectB.drawVStripes();
        rectC.draw();
        rectA.drawVStripes();
        rectB.setChar('#', '!');
        rectB.drawVStripes();
    }
}
```

Ed ecco il risultato:

```
**********
**********
**********
**********
**********

   *o*o*o*o*o*o
   *o*o*o*o*o*o
   *o*o*o*o*o*o

       ****
       ****
       ****
       ****
*o*o*o*o*o
*o*o*o*o*o
*o*o*o*o*o
*o*o*o*o*o
*o*o*o*o*o

   #!#!#!#!#!#!
   #!#!#!#!#!#!
   #!#!#!#!#!#!
```

Questa è ancora una versione rudimentale della classe `CharRect`, più avanti vedremo delle versioni molto più versatili e potenti.

Ed ora alcune considerazioni circa lo stile di programmazione che sono importanti perché se applicate con costanza e coerenza migliorano la leggibilità del codice. Il nome di una classe di solito è un sostantivo singolare che si riferisce direttamente all'oggetto della classe. Inoltre è consuetudine che i nomi delle classi inizino con una maiuscola. Questo per meglio dstinguerli dagli altri identificatori (nomi di metodi e variabili) che dovrebbero sempre iniziare con una minuscola. I nomi delle costanti invece, come nel C, dovrebbero contenere solo maiuscole. I nomi dei metodi che semplicemente modificano i valori dei campi della classe dovrebbero iniziare con `set` (come il metodo `setChar()`). Mentre quelli che ritornano il valore di un campo dovrebbero iniziare con `get` (se ci fosse un simile metodo nella nostra classe si chiamerebbe `getChar()`). Inutile, forse, aggiungere quanto sia importante per la leggibilità, sopratutto per un linguaggio complesso come Java, una corretta e coerente indentazione del codice. Per un lettore umano, può essere persino più importante della correttezza sintattica.

Esercizi

[Metodi_che_accedono] Aggiungere alla classe `CharRect` dei metodi per leggere i campi `fillChar` e `fillChar2` e inoltre aggiungere un metodo per modificare la posizione del rettangolo.
[Strisce_orizzontali] Aggiungere alla classe `CharRect` un metodo `drawHStripes()` che stampa il rettangolo a strisce orizzontali come nell'esempio qui sotto:

```
******
oooooo
******
oooooo
```

L'implementazione può sfruttare il metodo `drawLine()`?
[Scacchiera] Aggiungere alla classe `CharRect` un metodo `drawChessboard()` che stampa il rettangolo a mo' di scacchiera, come nell'esempio qui sotto:

```
*o*o*o*o
o*o*o*o*
*o*o*o*o
```

Si può modifcare il metodo `drawLine()` in modo tale che risulti utile anche per stampare i rettangoli a

scacchiera?

[Scacchiera_su_misura] Aggiungere alla classe `CharRect` un metodo `drawChessboard(int cellSize)` che stampa il rettangolo a mo' di scacchiera con le celle di dimensione `cellSize`, come negli esempi qui sotto riportati (il rettangolo ha larghezza 17 e altezza 9):

```
cellSize = 1          cellSize = 3          cellSize = 7
*o*o*o*o*o*o*o*o*     ***ooo***ooo***oo     *******ooooooo***
o*o*o*o*o*o*o*o*o     ***ooo***ooo***oo     *******ooooooo***
*o*o*o*o*o*o*o*o*     ***ooo***ooo***oo     *******ooooooo***
o*o*o*o*o*o*o*o*o     ooo***ooo***ooo**     *******ooooooo***
*o*o*o*o*o*o*o*o*     ooo***ooo***ooo**     *******ooooooo***
o*o*o*o*o*o*o*o*o     ooo***ooo***ooo**     *******ooooooo***
*o*o*o*o*o*o*o*o*     ***ooo***ooo***oo     *******ooooooo***
o*o*o*o*o*o*o*o*o     ***ooo***ooo***oo     ooooooo*******ooo
*o*o*o*o*o*o*o*o*     ***ooo***ooo***oo     ooooooo*******ooo
```

Si può ulteriormente modificare (invero generalizzare) il metodo `drawLine()` in modo tale che risulti utile anche per stampare i rettangoli a scacchiera su misura? Si osservi che una volta implementato il metodo `drawChessboard(int cellSize)` l'implementazione del metodo dell'esercizio precedente si risolve nella semplice invocazione di `drawChessboard(1)`.

[Metti_alla_prova] Scrivere un programma che mette alla prova le eventuali modifiche apportate alla classe `CharRect` in uno o più degli esercizi precedenti. Questo è in realtà un meta-esercizio perché può essere applicato a tutti gli esercizi che seguono.

[Piramidi] Definire una nuova classe `CharPyramid` che permette di stampare triangoli (simili al profilo di una piramide) e di altezza variabile, come negli esempi qui sotto:

```
height = 1    height = 2   height = 3   height = 5
                                             *
                                            ***
                                  *        *****
                    *            ***      *******
 *                 ***          *****    *********
```

La classe, similmente alla classe `CharRect`, avrà dei campi che determinano la posizione della piramide (le coordinate dello spigolo in alto a sinistra del più piccolo rettangolo che racchiude la piramide) e un campo che ne determina l'altezza. Inoltre avrà un opportuno costruttore e un metodo per la stampa.

[Piramidi_capovolte] Aggiungere un metodo `drawUpsideDown()` alla classe dell'esercizo precedente che stampa la piramide capovolta, come nell'esempio qui sotto:

```
*********
 *******
  *****
   ***
    *
```

[Date] Definire una nuova classe `Data` per rappresentare appunto date. I campi dovrebbero essere `giorno`, `mese` e `anno` (tutti e tre numeri interi). La classe deve avere un costruttore `Data(int g, int m, int a)` e tre metodi per stampare la data nei tre formati:

```
18/2/2009       18022009       18 febbraio 2009
```

[Date+] Aggiungere alla classe `Data` dei metodi per leggere i valori dei campi `giorno`, `mese` e `anno`. Aggiungere anche un metodo `incrementa(int gg)` che consente di modificare la data incrementandola di `gg` giorni. Ad esempio, se la data dell'oggetto è 18/2/2009 ed è eseguita l'invocazione `incrementa(13)` allora la data diventa 3/3/2009.

[Differenza_di_date] Aggiungere alla classe `Data` un metodo `int diff(Data d)` che ritorna il numero di giorni che intercorrono tra la data dell'oggetto e la data `d`. Ad esempio, se la data dell'oggetto è 18/2/2009 allora l'invocazione di `diff(d)` con `d` che rappresenta la data 3/3/2009 ritorna 13, mentre se `d` rappresenta la data 5/2/2009 allora ritorna -13.

[Date_in_stringhe] Aggiungere alla classe `Data` un costruttore `Data(String d)` che costruisce l'oggetto interpretando la data contenuta nella stringa `d` assumendo che possa essere in uno qualsiasi dei

tre formati dell'esercizio [Date].

[Data_di_nascita] Aggiungere alla classe `Studente` un campo di tipo `Data` che contiene la data di nascita dello studente. Modificare il costruttore in modo che prenda come argomento anche la data di nascita espressa tramite una stringa. Inoltre, modificare il metodo `stampa()` in modo che stampi anche la data di nascita dello studente.

[Razionali] Definire una nuova classe `Razionale` per rappresentare numeri razionali. I campi dovrebbero essere `numeratore` e `denominatore` entrambi di tipo `long`. La classe deve avere due costruttori `Razionale(long num, long den)` e `Razionale(double d)`. Il secondo cotruttore dovrà fare una conversione determinando il numeratore e il denominatore dal numero in virgola mobile `d`. Questa conversione potrebbe non essere possibile se `d` è troppo vicino a zero (ma non è zero) o è troppo grande. Aggiungere anche un metodo che stampa il numero razionale.

[Razionali+] Aggiungere alla classe `Razionale` il metodo `void add(Razionale r)` che addiziona il razionale `r` modificando così il valore dell'oggetto (su cui è invocato il metodo). Aggiungere anche un metodo statico `static Razionale add(Razionale r1, Razionale r2)` che ritorna un nuovo oggetto della classe `Razionale` il cui valore è pari alla somma del valore di `r1` e il valore di `r2`.

Tipi, riferimenti e variabili

È arrivato il momento di chiarire alcuni punti riguardo ai tipi. Quando una classe è definita il suo nome diventa automaticamente il nome di un tipo. Questo è simile a ciò che accade in C con le `struct`. Ma c'è una importante differenza. Consideriamo una semplice classe che rappresenta punti in uno spazio bidimensionale:

```
public class Point {
    public double x, y;        //campi pubblici (coordinate del punto)
    public Point(double x, double y) {   //costruttore
        this.x = x;
        this.y = y;
    }
}
```

Dopo questa definizione, il nome della classe diventa anche il nome di un tipo e così possiamo dichiarare variabili di tipo `Point`:

```
Point p;
```

Se `Point` fosse una `struct` del C, la variabile `p` avrebbe come valore una `struct` di tipo `Point`. In Java, invece, la variabile `p` ha come possibile valore il *riferimento* ad un oggetto della classe `Point`. Il riferimento ad un oggetto può essere pensato come il puntatore o l'indirizzo di memoria dell'oggetto. Però in Java i riferimenti sono del tutto opachi al programmatore, non c'è qualcosa di simile all'aritmetica dei puntatori del C. Ovvero, in Java non c'è alcun modo per manipolare (direttamente) i riferimenti.

Riassumendo, il nome di una classe determina un tipo detto *tipo classe* (*class type*) i cui valori sono riferimenti a oggetti della classe. I tipi classe fanno parte della più ampia famiglia dei *tipi riferimento* (*reference types*). Quindi, i tipi classe sono tipi riferimento ma, come vedremo presto, anche altri tipi, diversi dai tipi classe, sono tipi riferimento.

Tipi primitivi e tipi riferimento Un tipo primitivo, come ad esempio `int`, ha come valori proprio i valori di quel tipo non i riferimenti a valori (oggetti) di quel tipo. Il seguente frammento di codice mostra le implicazioni di questa differenza per gli assegnamenti:

```
int a = 12;
int b;
b = a;          /* ora ci sono due copie del valore intero 12, una nella variabile a e
                   l'altra nella variabile b */
Point p = new Point(0.0, 0.0);
Point q = p;    /* ora c'è un solo oggetto della classe Point e il riferimento ad
                   esso è contenuto sia nella variabile p che in q. La variabile q non
                   contiene una copia dell'oggetto ma solamente un riferimento ad esso */
q.x = 1.0;      // ora l'oggetto Point ha coordinate (1.0, 0.0)
```

```
p.y = 2.0;      // ora l'oggetto Point ha coordinate (1.0, 2.0)
```

Siccome il passaggio dei parametri è per valore, un metodo che ha come argomento un tipo primitivo non può modificare il valore della variabile che è stata usata per fornire quell'argomento nell'invocazione del metodo. Mentre, se l'argomento è un tipo riferimento allora il metodo può modificare il valore dell'oggetto il cui riferimento è stato passato nell'invocazione del metodo.

Anche per quanto riguarda il confronto, i tipi primitivi e i tipi riferimento differiscono. Il seguente frammento di codice mostra questa differenza:

```
int a = 12;
int b = 12;
if (a == b)     // è VERO perché le variabili a e b hanno lo stesso valore
    ....
Point p = new Point(0.0, 0.0);
Point q = new Point(0.0, 0.0);
if (p == q)     /* è FALSO perché le variabili p e q hanno valori differenti. Anche
                   se gli oggetti a cui si riferiscono hanno lo stesso valore, p e q
                   si riferiscono a oggetti DISTINTI */
    ....
if (p.x == q.x && p.y == q.y)    // questo confronto, invece, è VERO
```

Inizializzazioni e valori di default I campi, cioè le variabili dichiarate nel corpo di una classe, sono automaticamente inizializzati con valori di default (vedi la tabella qui sotto). Invece le variabili locali dichiarate nel corpo di un metodo non sono automaticamente inizializzate. Questo significa che una variabile dichiarata in un metodo ha un valore indefinito finché non gli viene assegnato esplicitamente un valore. Ecco la tabella dei valori di default:

Tipo	Valore di default
boolean	false
char	'\u0000'
byte	0
short	0
int	0
long	0
float	0.0f
double	0.0d
tipo riferimento	null

Il valore `null` (che è una parola chiave di Java) significa assenza di riferimento o riferimento indefinito.

Errori in compilazione e in esecuzione Come si è già detto le variabili locali (dichiarate nel corpo dei metodi) non sono automaticamente inizializzate. Così se all'interno di un metodo si scrive:

```
Point p;
double x = p.x;
```

Questo immediatamente produce una segnalazione di errore da parte del compilatore che avverte che la variabile `p` non è stata inizializzata prima di essere usata. Se invece, sempre all'interno di un metodo, si scrive:

```
Point p = null;
double x = p.x;
```

Il compilatore è soddisfatto (la variabile `p` è stata inizializzata) ma sarà generato un errore durante l'esecuzione del metodo perchè si è tentato di accedere ad un oggetto inesistente (il riferimento è `null`).

In Java gli errori che si producono durante l'esecuzione di un programma si chiamano *eccezioni* (*Exceptions*). Ovviamente questi possono essere prodotti da una grande varietà di cause, non solo dal tentativo di accedere ad un oggetto inesistente. Java fornisce dei meccanismi per gestire tali errori che vedremo più avanti.

Classi nidificate

Le classi possono anche essere definite all'interno di altre classi. Vale a dire che nel corpo di una classe oltre a campi, costruttori e metodi possono anche essere definite delle classi (*nested classes*). Queste possono essere statiche o non statiche. Una classe nidificata statica è direttamente connessa alla classe che la contiene mentre classi nidificate non statiche (dette *inner classes*, cioè *classi interne*) sono direttamente connesse con gli oggetti della classe che le contiene, più precisamente gli oggetti delle classi interne sono direttamente associati agli oggetti della classe che ne contiene la definizione. Considereremo solamente le classi nidificate statiche perchè sono la forma più semplice e più utile di classi nidificate.

Generalmente, una classe nidificata statica è usata per definire un tipo che ha senso ed è utile nel contesto della classe di definizione ma che potrebbe non avere senso o non essere utile al di fuori della classe di definizione. Essenzialmente si tratta di un meccanismo che può risultare utile per migliorare la struttura logica della definizione di una classe. La sintassi (un po' semplificata) della definizione di una classe nidificata statica è semplice e diretta:

```
public class NomeEnclosingClass {
    modificatori static class NomeNestedClass {
        <corpo-nested-class>
    }
}
```

I modificatori possono essere tutti quelli che si possono usare nella definizione di una classe, in particolare possono essere quelli di accesso `public` e `private`. Anche il corpo può contenere qualsiasi cosa può contenere una classe, comprese definizioni di classi ulteriormente nidificate. Dall'interno della classe che ne contiene la definizione (*`NomeEnclosingClass`*) la classe nidificata è accessibile tramite il suo semplice nome (*`NomeNestedClass`*). Invece, dall'esterno (sempreché non sia privata) è accessibile solamente specificando anche il nome della classe che la contiene (*`NomeEnclosingClass.NomeNestedClass`*).

Consideriamo un esempio. Aggiungiamo alla classe `Studente` i dati riguardanti un eventuale trasferimento da un altro ateneo. Questi dati (per semplicità ci limtiamo all'ateneo e al corso di laurea di provenienza) possono essere raggruppati in una classe che chiamiamo `Provenienza`. Siccome tali dati hanno poco senso se considerati disgiuntamente dalla classe `Studente`, è naturale rappresentarli tramite una classe nidificata nella classe `Studente`. La seguente definizione non riporta tutti i membri della classe `Studente`:

```
public class Studente {
    public static class Provenienza {    // classe nidificata (statica)
        public final String ateneo;
        public final String corso;

        private Provenienza(String ateneo, String corso) {
            this.ateneo = ateneo;
            this.corso = corso;
        }
    }

    private String nome, cognome;
    private Provenienza prov = null;

    public Studente(String nome, String cognome) {
        this.nome = nome;
        this.cognome = cognome;
    }

    public void setProvenienza(String ateneo, String corso) {
        prov = new Provenienza(ateneo, corso);
    }
```

```
    public Provenienza getProvenienza() { return prov; }
}
```

Si osservi che i due campi della classe `Provenienza` sono stati resi pubblici perché essendo costanti (dichiarati `final`) non possono essere modificati. I due campi sono inizializzati nel costruttore. In generale, un campo dichiarato `final` deve essere inizializzato o direttamente nella dichiarazione o nel costruttore. Il costruttore della classe `Provenienza` è privato perché non ha senso che la classe possa essere istanziata al di fuori del contesto della classe `Studente`. Per questo c'è il metodo `setProvenienza()` che crea una nuova istanza della classe e la collega tramite il campo `prov` all'oggetto `Studente`. Impostando così i dati relativi alla provenienza dello studente.

Per vedere come la classe nidificata `Provenienza` può essere usata dall'esterno della classe `Studente`, consideriamo un semplice programma che la usa:

```
public class Test {
     public static void main(String[] args) {
        Studente stu = new Studente("Mario", "Rossi");
        stu.setProvenienza("Universita' di Bologna", "Laurea in Chimica");
        Studente.Provenienza prov = stu.getProvenienza();
        if (prov != null)
            System.out.println("Ateneo: "+prov.ateneo+"    Corso: "+prov.corso);
        else System.out.println("Nessuna provenienza");
    }
}
```

Si noti come dall'esterno della classe `Studente` l'accesso alla classe nidificata `Provenienza` può avvenire solamente specificando `Studente.Provenienza`. Un'altra importante osservazione da fare riguarda il metodo `getProvenienza()`. Questo metodo ritorna il riferimento ad un oggetto contenuto in un campo privato. In generale, tali metodi ritornano un riferimento ad una copia dell'oggetto, non il riferimento all'oggetto originale, perché altrimenti c'è la possibilità che l'oggetto originale possa essere modificato al di fuori del controllo della classe. In questo caso però l'oggetto di tipo `Provenienza` non è modificabile (è *immutabile*) quindi si può rendere pubblico il suo riferimento senza rischi. È la stessa ragione per cui si possono tranquillamente rendere pubblici i riferimenti agli oggetti di tipo `String` perché sono anch'essi immutabili.

Esercizi

[Errori] Nel seguente programma ci sono tre errori trovarli e spiegarli.

```
class Pair {
    private final int val;
    private String str;

    public Pair(int v) {
        val = v;
    }

    public Pair(String s) {
        str = s;
    }

    public void set(String s) { str = s; }
    public int getVal() { return val; }
    public String getStr() { return str; }
}

public class Test {
    public static void main(String[] args) {
        Pair p = new Pair();
        Pair p2 = new Pair(13);
        System.out.println(p2.getStr());
        Pair pp;
```

```
        pp.set("A");
    }
}
```

[Immutabilità] Si consideri il seguente frammento di codice Java:

```
String s = "prima";
String s2 = s;
s2 += "dopo";
Sytem.out.println(s);
```

Spiegare perché se viene eseguito stamperà `"prima"` invece di `"primadopo"`.

[Liste_di_interi] Definire una classe `IntList` per rappresentare liste di interi, cioè, una istanza della classe rappresenta una lista di interi. La classe deve avere un costruttore (senza parametri) che costruisce la lista vuota e deve implementare i seguenti metodi.

- `void addHead(int x)` aggiunge `x` in testa alla lista.
- `void addTail(int x)` aggiunge `x` in coda alla lista.
- `int find(int x)` ritorna la posizione (a partire da `0`) della prima occorrenza di `x` nella lista, se `x` non è presente ritorna `-1`.
- `boolean remove(int x)` rimuove dalla lista la prima occorrenza di `x` e ritorna `true`, se `x` non è presente ritorna `false`.
- `int length()` ritorna la lunghezza della lista.
- `print()` stampa (nello standard output) la lista.

Suggerimento: Definire una classe nidificata statica e privata della classe `IntList` per rappresentare gli elementi della lista. Ogni istanza di tale classe nidificata conterrà il riferimento al prossimo elemento della lista.

[Date_vicine] Scrivere un programma che legge in input una sequenza di date, una per linea, e poi stampa la coppia di date (o le coppie) che sono più vicine (cioè, la differenza in giorni tra le due date è la più piccola). La sequenza di date termina quando si legge una linea vuota.

Suggerimento: Definire ed usare una opportuna classe che rappresenta liste di date (cioè, gli elementi della lista sono di tipo `Data`, si vedano gli esercizi [Date], [Date+] e [Differenza_di_date]).

[Code_di_stringhe] Definire una classe `StrQueue` per rappresentare code di stringhe. La classe deve avere un costruttore (senza parametri) che costruisce la coda vuota e deve implementare i seguenti metodi.

- `void enqueue(String s)` accoda la stringa `s` alla coda.
- `String first()` ritorna la stringa in testa alla coda, se la coda è vuota ritorna `null`.
- `String dequeue()` estrae e ritorna la stringa in testa alla coda, se la coda è vuota ritorna `null`.
- `boolean isEmpty()` ritorna `true` se la coda è vuota e `false` altrimenti.

Suggerimento: Definire una classe nidificata statica e privata della classe `StrQueue` per rappresentare gli elementi della coda. Ogni istanza di tale classe nidificata conterrà il riferimento al prossimo elemento della coda.

[Pile_di_stringhe] Definire una classe `StrStack` per rappresentare pile di stringhe. La classe deve avere un costruttore (senza parametri) che costruisce la pila vuota e deve implementare i seguenti metodi.

- `void push(String s)` aggiunge la stringa `s` in cima alla pila.
- `String top()` ritorna la stringa in cima alla pila, se la pila è vuota ritorna `null`.
- `String pop()` estrae e ritorna la stringa in cima alla pila, se la pila è vuota ritorna `null`.
- `boolean isEmpty()` ritorna `true` se la pila è vuota e `false` altrimenti.

Suggerimento: Definire una classe nidificata statica e privata della classe `StrStack` per rappresentare gli elementi della pila. Ogni istanza di tale classe nidificata conterrà il riferimento al prossimo elemento della pila.

[Pile_di_interi&stringhe] Definire una classe `IntStrStack` per rappresentare pile di interi e stringhe. Cioè, una istanza della classe gestisce una pila i cui elementi possono essere sia interi (di tipo `long`) che stringhe. La classe deve avere un costruttore che costruisce la pila vuota e deve implementare i seguenti metodi.

- `void push(long v)` aggiunge l'intero `v` in cima alla pila.
- `void push(String s)` aggiunge la stringa `s` in cima alla pila.

- `boolean isTopInt()` ritorna `true` se l'elemento in cima alla pila è un intero, altrimenti ritorna `false`.
- `boolean isTopStr()` ritorna `true` se l'elemento in cima alla pila è una stringa, altrimenti ritorna `false`.
- `long topInt()` ritorna l'intero in cima alla pila, se in cima alla pila non c'è un intero ritorna `0`.
- `String topStr()` ritorna la stringa in cima alla pila, se in cima alla pila non c'è una stringa ritorna `null`.
- `void pop()` rimuove l'elemento in cima alla pila.
- `boolean isEmpty()` ritorna `true` se la pila è vuota e `false` altrimenti.

Suggerimento: Definire una classe nidificata statica e privata della classe `IntStrStack` per rappresentare gli elementi della pila. Ogni istanza di tale classe nidificata conterrà il riferimento al prossimo elemento della pila e potrà contenere o un intero o una stringa.

[Espressioni] Scrivere un programma che prende in input una linea di testo che contiene una espressione aritmetica e stampa il valore dell'espresssione. Si può assumere che l'espressione in input è completamente parentesizzata (cioè ogni operazione è fra parentesi e ogni parentesi contiene direttamente una sola operazione), tutti gli elementi sono separati da spazi ed è sintatticamente corretta. Ecco alcuni esempi di espressioni:

espressione	*valore*
`( ( ( 10 * 2 ) + 100 ) - 90 )`	`30`
`( 12 - ( 4 + ( ( 123 - 12 ) * ( 45 * 2 ) ) ) )`	`-9982`

Suggerimento: Per leggere gli elementi dell'espressione si può usare la classe `Scanner`. Sfruttare la classe `IntStrStack` dell'esercizio precedente per fare il parsing e la valutazione dell'espressione.