Lezione 6

Istruzioni di controllo

Le operazioni di recupero e manipolazione dei dati raramente seguono un percorso lineare. Il più delle volte, i programmi compiono dei salti, ripetono più volte un blocco di istruzioni, oppure prendono strade diverse secondo le situazioni riscontrate. Per il controllo del flusso di esecuzione, Java rende disponibili tre insiemi di istruzioni di controllo. Chi proviene da un qualsiasi linguaggio C-like non faticherà nell'assimilare quanto riportato in questa lezione: Java, nelle istruzioni di controllo, ricalca scrupolosamente il rodato modello di C, salvo per alcune discordanze di entità minore.

6.1 - Istruzioni di selezione

Le istruzioni di selezione permettono di imboccare strade diverse nel caso in cui una certa condizione sia verificata oppure no.

La più importante istruzione di selezione è il costrutto ***if-else***:

```
if (condizione) istruzione1;
else istruzione2;
```

La condizione di selezione deve necessariamente essere espressa mediante un valore di natura booleana, che può essere letterale, contenuto in una variabile, calcolato da un'espressione oppure restituito da una funzione. Se (*if*) il valore riscontrato è *true*, allora l'interprete esegue *istruzione1*, altrimenti (*else*) considera *istruzione2*:

```
if (a == 5) System.out.println("a vale 5");
else System.out.println("a non vale 5");
```

Il ramo *else* è facoltativo e può essere omesso:

```
if (a == 5) System.out.println("a vale 5");
```

Le istruzioni collegate ai rami *if* e *else* possono essere semplici oppure complesse (cfr. Lezione 3). Per associare più istruzioni semplici ad uno dei due rami (o ad entrambi), è sufficiente far ricorso ad un blocco delimitato da parentesi graffe:

```
if (condizione) {
    istruzione;
    istruzione;
    istruzione;
    ...
} else {
    istruzione;
    istruzione;
    istruzione;
    ...
}
```

Naturalmente, è possibile annidare più costrutti *if-else*:

```

if (a == 5) System.out.println("a vale 5");
else {
    System.out.println("a non vale 5");
    if (a == 3) System.out.println("a vale 3");
    else System.out.println("a non vale neanche 3");
}

```

In casi come questo, bisogna solo fare attenzione nel collegare correttamente i rami *else* alle corrispondenti istruzioni *if*.

Un particolare caso di sequenze *if-else* annidate è costituito dal costrutto noto come ***if-else-if***.

```

if (condizione1) istruzione1;
else if (condizione2) istruzione2;
else if (condizione3) istruzione3;
else if (condizione4) istruzione4;
...
else istruzioneN;

```

Ad esempio:

```

if (a == 1) System.out.println("a vale 1");
else if (a == 2) System.out.println("a vale 2");
else if (a == 3) System.out.println("a vale 3");
else if (a == 4) System.out.println("a vale 4");

```

In presenza di una struttura di questo tipo, avviene che al massimo uno dei rami introdotti possa essere eseguito. La stessa cosa accadrebbe scrivendo:

```

if (a == 1) System.out.println("a vale 1");
if (a == 2) System.out.println("a vale 2");
if (a == 3) System.out.println("a vale 3");
if (a == 4) System.out.println("a vale 4");

```

Tuttavia, questa seconda forma è sconveniente sia da un punto di vista logico sia in termini di prestazioni. Le quattro valutazioni richieste, infatti, vengono sempre eseguite tutte, talvolta in maniera ridondante. Con la sequenza *if-else-if*, al contrario, la successione delle valutazioni si interrompe nel caso sia riscontrata una corrispondenza gestita. Se *a* vale 1, ad esempio, il costrutto *if-else-if* permette di non considerare gli inutili confronti con 2, 3 e 4.

La condizione associata ad un'istruzione *if*, come già detto, deve essere costituita da un valore di tipo *boolean*. Le condizioni di selezione più complesse vengono generalmente rese mediante delle espressioni che impiegano sia gli operatori di confronto sia gli operatori logici booleani esaminati nella lezione precedente:

```

if ((a >= 3 && a <= 6) || (a > 10 && a < 15)) {
    istruzione;
}

```

Questo frammento di codice, letteralmente, si traduce in: "se il valore di *a* è compreso tra 3 e 6 (estremi inclusi) oppure tra 10 e 15 (questa volta estremi esclusi) esegui istruzione". Con un po' di pratica e con qualche esercizio, diventa facile scrivere delle istruzioni di selezione

complesse, coerenti e raffinate.

Benché appaia superfluo, è bene ricordare che l'operatore booleano NOT (!) può invertire la condizione specificata. In termini pratici, le due strutture sotto riportate sono equivalenti:

```
if (condizione) istruzione1;
else istruzione2;

if (!condizione) istruzione2;
else istruzione1;
```

Gli argomenti appena esaminati permettono di svelare il mistero legato agli operatori logici booleani short-circuit. Si consideri questo frammento:

```
int a = 0;
if (a != 0 & (5 / a) > 2) {
    ...
}
```

L'esecuzione del codice produce un errore di runtime, a causa di una tentata divisione per zero. Un caso analogo è già stato presentato nella lezione precedente, quando si discuteva sull'utilità dell'operatore ternario di valutazione condizionata (cfr. Paragrafo 5.9). Ricorrendo alla forma short-circuit dell'operatore AND, il problema viene evitato:

```
int a = 0;
if (a != 0 && (5 / a) > 2) {
    ...
}
```

Giacché il valore di `a` non è diverso da 0, l'operatore `&&` salta la valutazione successiva, scansando così la problematica divisione.

La seconda istruzione di selezione supportata da Java è ***switch***:

```
switch (valore) {
case costante1:
    istruzione;
    istruzione;
    ...
    break;
case costante2:
    istruzione;
    istruzione;
    ...
    break;
case costante3:
    istruzione;
    istruzione;
    ...
    break;
...
default:
    istruzione;
    istruzione;
```

```
...
break;
}
```

Il termine valore può essere un letterale, una variabile, un'espressione oppure il risultato di una funzione, purché sia sempre un numerico intero (un *byte*, un *char*, uno *short*, un *int* oppure un *long*). I valori in virgola mobile, i booleani e gli oggetti non sono ammessi. I termini *costante1*, *costante2*, *costante3* e così via, espressi di fianco alle diciture *case*, devono necessariamente essere dei valori letterali oppure delle variabili *final* inizializzate letteralmente.

In sostanza, il valore passato a *switch* viene confrontato con tutte le costanti che affiancano i diversi casi gestiti. Quando è riscontrata una uguaglianza, le istruzioni associate al caso vengono eseguite. Se nessuna corrispondenza è individuata, l'interprete esegue le istruzioni accodate alla clausola *default*, sempre che questa sia presente (l'uso di *default* è facoltativo, un po' come avviene con *else* nei costrutti *if*). Ad esempio:

```
switch (a) {
case 1:
    System.out.println("a vale 1");
    break;
case 2:
    System.out.println("a vale 2");
    break;
case 3:
    System.out.println("a vale 3");
    break;
default:
    System.out.println("a è diversa da 1, 2 e 3!");
    break;
}
```

Questo frammento è equivalente, nel risultato prodotto, al seguente codice:

```
if (a == 1) System.out.println("a vale 1");
else if (a == 2) System.out.println("a vale 2");
else if (a == 3) System.out.println("a vale 3");
else System.out.println("a è diversa da 1, 2 e 3!");
```

Più casi possono essere ricondotti ad un solo gruppo di istruzioni:

```
switch (a) {
case 1:
case 2:
    System.out.println("a è 1 o 2");
    break;
case 3:
case 4:
    System.out.println("a è 3 o 4");
    break;
default:
    System.out.println("a è diversa da 1, 2, 3 e 4!");
    break;
}
```

Eliminando la clausola *break* da uno dei casi gestiti, si fa in modo che venga eseguita anche la sequenza di istruzioni legata al confronto successivo:

```
switch (a) {
case 1:
    System.out.println("X");
case 2:
    System.out.println("Y");
    break;
}
```

Se *a* vale 2, questo frammento produce il responso:

Y

Quando *a* vale 1 ambo i gruppi sono eseguiti, nonostante il secondo confronto abbia esito negativo:

X
Y

6.2 - Istruzioni di iterazione

Le istruzioni di iterazione, dette altrimenti cicli, fanno in modo che un blocco di codice possa essere eseguito più volte.

I cicli ***while*** ripetono le istruzioni a loro associate fin quando una peculiare condizione booleana risulta vera:

```
while (condizione) istruzione;
```

Ad esempio:

```
int a = 0;
while (a < 5) a++;
```

Il ciclo *while* introdotto in questo frammento esegue l'incremento di *a* per cinque volte consecutive. Il ragionamento è semplice. Alla prima verifica, *a* è minore di 5, pertanto il suo valore viene aumentato di una unità. La condizione di confronto, a questo punto, viene valutata nuovamente. Giacché *a* è ancora minore di 5, l'incremento è ripetuto. Da qui in poi, si procede osservando lo stesso ragionamento, fin quando *a* non diventa pari a 5 e la condizione impiegata, di conseguenza, risulta falsa.

Un blocco delimitato da parentesi graffe permette di associare più istruzioni ad un solo ciclo ***while***:

```
int a = 0;
while (a < 3) {
    System.out.println(a);
    a++;
}
```

Questo codice produce il seguente output:

```
0
1
2
```

Il costrutto **do-while** si comporta in maniera simile al ciclo *while*:

```
do istruzione;
while (condizione);
```

La sola differenza tra i due cicli risiede nel fatto che l'istruzione specificata viene eseguita almeno una volta, anche nel caso in cui la condizione di iterazione sia falsa sin da principio. Il valore booleano che determina la necessità di ulteriori passaggi, infatti, viene valutato dopo ogni iterazione, e non prima, come invece avviene con *while*.

```
int a = 5;
do a--;
while (a < 0);
```

Al termine dell'esecuzione di questo frammento, *a* vale 4. Benché il valore di *a* sia maggiore di 0 sin da principio, l'operazione di decremento viene ugualmente eseguita. Questo non accade con un semplice *while*, dove la condizione è valutata prima di ogni altra cosa. Anche nel caso dei cicli *do-while*, i blocchi possono essere liberamente impiegati:

```
do {
    istruzione;
    istruzione;
    ...
} while (condizione);
```

Il ciclo **for** è un costrutto di iterazione ricco di funzionalità:

```
for (inizializzazione; condizione; iterazione) istruzione;
```

L'istruzione abbinata al ciclo può essere semplice, come nel modello appena visto, oppure complessa, come nel seguente schema:

```
for (inizializzazione; condizione; iterazione) {
    istruzione;
    istruzione;
    ...
}
```

Non appena il flusso di esecuzione imbocca un ciclo *for*, l'interprete esegue l'espressione di inizializzazione abbinata all'istruzione. In questa parte del codice, di solito, si imposta una variabile di controllo, che regola l'andamento del ciclo lavorando come contatore. L'inizializzazione è eseguita una sola volta, prima dell'effettiva entrata nel ciclo, e non viene ripetuta durante le eventuali iterazioni successive. A questo punto, l'interprete valuta la

condizione. Le condizioni impiegate in un ciclo *for* sono generalmente delle espressioni booleane di confronto. Se la condizione risulta vera, il corpo del ciclo (ossia il gruppo delle istruzioni abbinate al comando *for*) viene ordinatamente eseguito. Prima di valutare nuovamente la condizione, in modo da poter decretare l'esigenza di ulteriori passaggi, l'interprete esegue l'espressione di iterazione, che generalmente effettua una variazione sul contatore. Quindi si procede reiterando lo schema: se la condizione è vera, vengono eseguiti prima il corpo e poi l'espressione di iterazione. La procedura viene ripetuta fin quando la condizione non risulta falsa.

Il più classico esempio di ciclo *for* si basa sull'impiego di un contatore intero, incrementato di una unità ad ogni iterazione:

```
int i;
for (i = 0; i < 3; i++) System.out.println("Ciclo " + i);
```

Questo frammento di codice produce l'output:

```
Ciclo 0
Ciclo 1
Ciclo 2
```

In pratica, un ciclo *for* può sempre essere approssimativamente espanso in un costrutto *while*:

```
int i = 0;
while (i < 3) {
    System.out.println("Ciclo " + i);
    i++;
}
```

L'istruzione *for*, ad ogni modo, non è ridondante, poiché permette delle operazioni esclusive e proficue. Ad esempio, è possibile dichiarare la variabile di controllo direttamente nella porzione di codice dedicata alla sua inizializzazione:

```
for (int i = 0; i < 3; i++) {
    ...
}
```

In questo caso, la variabile *i* appartiene all'ambito di visibilità circoscritto dal corpo del ciclo, come se fosse stata dichiarata al suo interno. In pratica:

```
for (int i = 0; i < 3; i++) {
    ...
    System.out.println(i); // Ok, corretto!
    ...
}
int h = i + 5; // Errore, i qui non esiste!
```

In un medesimo ciclo *for* possono essere impiegate più espressioni di inizializzazione ed iterazione. In tal caso, una virgola deve essere usata come separatore. Il seguente esempio fa uso di due contatori (che devono necessariamente essere del medesimo tipo):

```
for (int i = 0, j = 6; i < j; i++, j--) {
    System.out.println("i: " + i);
    System.out.println("j: " + j);
}
```

L'output prodotto è:

```
i: 0
j: 6
i: 1
j: 5
i: 2
j: 4
```

Le espressioni di inizializzazione ed iterazione, infine, sono di utilizzo facoltativo:

```
int i = 0;
for ( ; i < 3 ; ) {
    System.out.println(i);
    i++;
}
```

Un approccio come questo è ridondante, giacché il costrutto *for* è stato usato come fosse un ciclo *while*. Tipicamente, infatti, le espressioni di inizializzazione ed iterazione non vengono mai omesse simultaneamente, ma una alla volta.

Anche l'impiego della condizione è facoltativo:

```
for ( ; ; ) System.out.println("ah ah ah!");
```

Si ottiene, così, un loop infinito: sulla console continuerà ad apparire un'agghiacciante risata, fin quando il programma non verrà interrotto manualmente (di norma, la pressione simultanea dei tasti CTRL e "c" termina bruscamente i programmi che occupano la riga di comando). Un altro tipo di loop infinito può essere ottenuto con il costrutto *while*:

```
while (true) {
    ...
}
```

I loop infiniti sono utili in diverse occasioni, anche se poche volte sono effettivamente senza termine: le istruzioni di salto, esaminate nel paragrafo successivo, vengono spesso impiegate per fornire una via di uscita alternativa. Generalmente, si incappa in loop realmente infiniti più a causa di errori di programmazione che non per manifesta volontà.

6.3 - Istruzioni di salto

Le istruzioni di salto permettono di spostarsi da una parte all'altra di un programma in maniera non sequenziale e a volte disordinata. Per questo motivo, godono generalmente di cattiva fama: il disordine è tanto un nemico della leggibilità quanto una spina nel fianco dei compilatori che cercano di ottimizzare i codici processati. Alcune sconsiderate forme di salto, come quelle di estrazione BASIC, favoriscono l'ingarbugliamento dei listati e fomentano

l'insorgere del cosiddetto *codice a spaghetti*. I linguaggi C-like, al contrario, fanno affidamento sulla coerenza e sull'organizzazione gerarchica dei blocchi di istruzioni, e proprio per questo si prestano bene per progetti di ragguardevole dimensione. Questo non significa, ad ogni modo, che C, C++, C# e Java non supportino alcune istruzioni di salto. I linguaggi C-like, infatti, implementano dei salti ragionati e ragionevoli. Java, in particolar modo, dispone di numerose strutture di questo tipo. Due di queste sono immediatamente esaminate in questo paragrafo, mentre altre verranno presentate successivamente, all'interno di ambiti più consoni alla loro trattazione.

L'istruzione **break**, in Java, ha triplice valenza. Il suo primo impiego è già stato esaminato quando si è parlato dei costrutti *switch*. Il secondo riguarda i cicli. Con *break* è possibile abbandonare un ciclo prima che la condizione di iterazione specificata risulti falsa:

```
System.out.println("Prima del ciclo");
for (int i = 0; i < 10; i++) {
    System.out.println(i);
    if (i > 2) break;
}
System.out.println("Dopo il ciclo");
```

Il conteggio da 4 a 9, che in assenza della clausola *break* sarebbe normalmente eseguito, viene saltato. L'output prodotto è:

```
Prima del ciclo
0
1
2
3
Dopo il ciclo
```

Questo tipo di salto può essere indistintamente applicato ai tre tipi di iterazione supportati da Java. La funzionalità torna utile in due occasioni: quando si riscontra l'inutilità delle iterazioni successive e quando si desidera uscire da un loop altrimenti infinito. Si osservi che, all'interno di un gruppo di cicli annidati, *break* causa l'uscita solamente da quello cui si riferisce, e non da quelli di ordine superiore:

```
for (int i = 0; i < 2; i++) {
    System.out.println("i:" + i);
    for (int j = 0; j < 10; j++) {
        if (j > 1) break;
        System.out.println("  j:" + j);
    }
}
```

L'output generato è:

```
i: 0
  j: 0
  j: 1
i: 1
  j: 0
  j: 1
```

Il terzo significato di *break* è legato all'impiego delle *etichette*. Le etichette sono elementi sintattici costituiti da un identificatore e da un carattere di due punti. Servono per assegnare un nome ad un'istruzione semplice o ad un blocco:

```
etichetta1: {  
    ...  
    etichetta2: {  
        ...  
    }  
    ...  
}
```

All'interno di una struttura del genere, l'istruzione

```
break etichetta;
```

fa uscire il flusso di esecuzione dal blocco specificato. Si consideri il seguente esempio:

```
boolean salta = true;  
blocco: {  
    System.out.println("mi leggi");  
    if (salta) break blocco;  
    System.out.println("non mi leggi");  
}
```

La stringa "non mi leggi" non viene stampata sul canale di output, poiché l'esecuzione del blocco è interrotta prima del comando corrispondente. Si osservi come il salto sia in realtà condizionato da un'istruzione *if*. Se si fosse fatto altrimenti, il compilatore avrebbe prodotto un errore. L'affermazione può essere verificata:

```
blocco: {  
    System.out.println("mi leggi");  
    break blocco;  
    System.out.println("non mi leggi");  
}
```

L'errore che si riceve è:

```
unreachable statement  
    System.out.println("non mi leggi");  
    ^
```

In pratica, il compilatore si accorge dell'incoerenza: se le istruzioni successive devono essere saltate incondizionatamente, che senso ha immetterle nel codice?

Quando si annidano più blocchi, è possibile specificare esplicitamente da quale di questi si intende uscire, ricorrendo all'etichetta associata:

```
boolean salta = true;  
uno: {  
    System.out.println("blocco uno");  
    due: {
```

```

        System.out.println("blocco due");
        if (salta) break uno;
        System.out.println("non mi leggi");
    }
    System.out.println("non mi leggi");
}
System.out.println("fuori da ogni blocco");

```

Il responso prodotto è:

```

blocco uno
blocco due
fuori da ogni blocco

```

Questa peculiarità può essere sfruttata quando, all'interno di una serie di cicli annidati, si vuole uscire fuori da un ciclo di ordine superiore rispetto quello in cui viene chiamata l'istruzione *break*:

```

ciclifor:
for (int i = 0; i < 2; i++) {
    System.out.println("i:" + i);
    for (int j = 0; j < 10; j++) {
        if (j > 1) break ciclifor;
        System.out.println("  j:" + j);
    }
}

```

Il riscontro ottenuto è mostrato di seguito:

```

i:0
  j:0
  j:1

```

Letteralmente *break* vuol dire interrompi. Interrompere l'esecuzione di un blocco che non sta venendo eseguito è un'operazione priva di senso. Di conseguenza, l'istruzione *break* può far riferimento esclusivamente ad uno dei blocchi annidati che la contiene. Non è logico specificare l'etichetta di un blocco estraneo all'istruzione. Il seguente codice, infatti, genera un errore di compilazione:

```

blocco1: {
    System.out.println("Blocco 1");
}

boolean salta = true;
blocco2: {
    System.out.println("Blocco 2");
    if (salta) break blocco1;
}

```

Il messaggio di errore riscontrabile è:

```

undefined label: blocco1

```

```
if (salta) break blocco1;
    ^
```

Per il compilatore, l'etichetta *blocco1* non è definita. Questo avviene perché le etichette hanno un ambito di visibilità analogo a quello delle variabili: sono note solo all'interno dei blocchi che identificano.

L'istruzione ***continue***, all'interno di un ciclo, conclude in anticipo l'iterazione corrente:

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) continue;
    System.out.println(i);
}
```

Questo ciclo stampa i numeri dispari compresi tra 1 e 10 (1, 3, 5, 7 e 9). L'esempio fa ben comprendere l'utilizzo di *continue*: ogni volta che la variabile *i* assume il valore di un numero pari (il modulo per 2 dei numeri pari è 0), l'iterazione corrente viene interrotta. Le istruzioni successive al salto, quindi, non sono eseguite. A differenza di *break*, che causa la totale uscita da un ciclo, *continue* fa in modo che i cicli riprendano ad essere eseguiti dal passo successivo, come se nulla fosse accaduto. L'istruzione può essere indistintamente applicata ai tre tipi di cicli supportati da Java.

In presenza di cicli annidati, *continue* può far ricorso ad un'etichetta. In questo modo è possibile specificare quale tra i cicli in esecuzione debba essere interrotto e successivamente ripreso:

```
cicloesterno:
for (int i = 0; i < 3; i++) {
    System.out.println("i: " + i);
    for (int j = 0; j < 10; j++) {
        if (j > 1) continue cicloesterno;
        System.out.println("  j: " + j);
    }
    System.out.println("non mi leggi");
}
```

L'impiego di questa forma di *continue* è piuttosto rara, poiché Java dispone di migliori strumenti per la costruzione di cicli complessi. All'occorrenza, ad ogni modo, è bene ricordarsi della disponibilità di questa struttura.

L'istruzione goto non c'è!

Mentre lavoravo alla stesura del mio primo libro su Java, mi è stata rivolta una domanda degna di nota da parte di uno dei tester. Ne riporto il testo:

"Vedo, con un po' di dispiacere devo dire, che Java non dispone di un'istruzione *goto* (come C++); dapprima mi sono chiesto se Carlo si fosse dimenticato di menzionare questo tipo di istruzione, allora ho stilato un breve codice Java servendomi delle stesse regole del C++. Come in parte mi aspettavo, non è stato compilato... Perché in Java l'istruzione *goto* non è supportata?"

Enrico Pelliccioni"

Per spiegare come mai *goto* non faccia parte del cuore di Java, trascrivo e traduco un breve frammento esplicativo tratto dal sito Web di Java (<http://java.sun.com/>):

"Java has no *goto* statement. Studies illustrated that *goto* is (mis)used more often than not simply "because it's there". Eliminating *goto* led to a simplification of the language. There are no rules about the effects of a *goto* into the middle of a *for* statement, for example. Studies on approximately 100,000 lines of C code determined that roughly 90 percent of the *goto* statements were used purely to obtain the effect of breaking out of nested loops. As mentioned above, multi-level break and continue remove most of the need for *goto* statements."

"Java non dispone dell'istruzione *goto*. Gli studi hanno dimostrato come l'istruzione *goto* venga spesso (ab)usata semplicemente "perché c'è". L'esclusione di *goto* porta ad una netta semplificazione del linguaggio. Ad esempio, non servono delle norme che descrivano l'effetto di un'istruzione *goto* all'interno di un ciclo *for*. Alcuni studi svolti su circa cento mila righe di codice C hanno dimostrato come l'istruzione *goto* sia utilizzata più o meno il 90% delle volte per causare l'uscita da una serie di cicli annidati. Come già detto, le versioni multi-livello di *break* e *continue* (allude all'uso delle etichette, deliziosa caratteristica di Java, *n.d. Carlo*) eliminano la necessità delle istruzioni *goto*."

Asserzioni

Le *asserzioni* sono una novità sintattica di Java 2 versione 1.4. Sono uno strumento molto utile durante le fasi di sperimentazione e risoluzione dei problemi di un software, prima della sua divulgazione e del suo impiego in casi reali. Le asserzioni controllano una condizione booleana, propagando un errore quando la stessa non è verificata. L'istruzione alla base delle asserzioni è *assert*:

```
assert a == 2;
```

Se *a* è diverso da 2, questa asserzione causerà l'arresto del programma. Sulla console verrà mostrato un messaggio di errore. Per un debug più efficiente e rapido, è possibile personalizzare il messaggio d'errore collegato alla condizione:

```
assert a == 2 : "a è diverso da 2";
```

La compilazione di un codice che utilizza la clausola *assert* necessita di un'opzione aggiuntiva:

```
javac -source 1.4 ...
```

A questo punto, durante l'esecuzione del programma, è possibile abilitare le asserzioni, in modo da poterle sfruttare per il debug:

```
java -enableassertions ...
```

o anche

```
java -ea ...
```