Capitolo 4

Strutture di controllo iterative

ANDREA GINI

Le strutture di controllo iterative permettono di impostare la ripetizione di un insieme di istruzioni per un determinato numero di volte. In Java esistono tre strutture di controllo di questo tipo: `while`, `for` e `do – while`. Sebbene siano equivalenti, ciascuna di queste tre forme ha un campo di applicazione privilegiato: nei prossimi paragrafi ciascuno di essi verrà approfondito in dettaglio.

Ciclo `while`

La struttura fondamentale del ciclo `while` è:

```
while(condizioneBooleana)
    istruzione;
```

come di consueto, l'istruzione può essere sostituita da un blocco:

```
while(condizioneBooleana) {
    istruzione_1;
    istruzione_2;
    ....
    istruzione_n;
}
```

L'effetto del `while` è quello di ripetere l'istruzione, o il blocco di istruzioni, fintanto che il valore della condizione booleana è vero. Il blocco all'interno del `while` deve contenere istruzioni che modifichino la condizione booleana; in caso contrario, il ciclo durerà all'infinito.

L'uso del `while` richiede alcune precauzioni: nel progetto di un ciclo `while`, infatti, è necessario considerare con attenzione le condizioni di ingresso e quelle di uscita. Se le condizioni di ingresso sono mal formulate, il computer in fase di esecuzione non entrerà mai in ciclo. Se d'altra parte esiste un errore di logica all'interno del ciclo, si corre il rischio che il computer in fase di esecuzione entri in loop, ossia resti intrappolato all'infinito dentro al ciclo. D'altra parte, ci sono situazioni in cui si desidera creare esplicitamente un ciclo infinito, ricorrendo a formulazioni del tipo:

```
while(true)
  istruzione;
```

Così facendo il ciclo proseguirà all'infinito, dal momento che la condizione booleana non diventerà mai falsa.

Ciclo do – while

Nel ciclo `while` standard, descritto nel paragrafo precedente, il blocco di istruzioni che costituisce il corpo del `while` può non essere mai eseguito, qualora le condizioni di ingresso non siano vere al momento di entrare in ciclo. Esistono casi in cui si vuole che tali istruzioni vengano eseguite comunque almeno una volta. Per questo, si può usare il costrutto `do - while`, la cui struttura è:

```
do
  istruzione;
while(condizioneBooleana)
```

o in alternativa:

```
do {
  istruzione_1;
  istruzione_2;
  ....
  istruzione_n;
}
while(condizioneBooleana)
```

se si desidera che venga eseguito un blocco di istruzioni.

Il costrutto `do - while` è presente in tutti i linguaggi di programmazione imperativi, a volte con il nome di `repeat - until`, come nel Pascal. Esso, in ogni caso, risulta poco utilizzato.

Ciclo for

Per ripetere un blocco di codice un numero prefissato di volte è possibile utilizzare un ciclo `while` di questo tipo:

```
int i = 0;
while(i < 10) {
    System.out.println(i);
    i++;
}
```

Il linguaggio Java prevede il costrutto `for`, che permette di gestire esplicitamente questo tipo di situazione, peraltro estremamente comune. La sintassi generale del ciclo `for` è:

```
for ( inizializzatore ; condizioneBooleana ; incremento )
    istruzione;
```

Anche in questo caso è possibile ricorrere al blocco qualora si desideri far ripetere un maggior numero di istruzioni:

```
for ( inizializzatore ; condizioneBooleana ; incremento ) {
    istruzione_1;
    istruzione_2;
    ....
    istruzione_n;
}
```

L'inizializzatore è un'istruzione che viene eseguita una sola volta prima di entrare nel ciclo. La condizione booleana è un'espressione che viene testata, come nel `while`, prima di ogni ciclo; infine, l'incremento è un'istruzione eseguita automaticamente al termine di ogni iterazione.

Il `for` permette di riscrivere l'esempio precedente in una forma più compatta:

```
for ( int i = 0 ; i < 10 ; i++ )
    System.out.println(i);
```

Grazie al `for`, è possibile gestire in modo molto più sicuro tutte le circostanze in cui sia necessario far ripetere un gruppo di istruzioni per un numero prefissato di volte. Come vedremo nei prossimi paragrafi, tuttavia, è possibile usare questo costrutto in modo più sofisticato.

Uso del `for` con parametri multipli

È possibile fare in modo che il ciclo `for` lavori su più di un contatore, replicando l'inizializzatore e l'incremento e separandoli con la virgola. Naturalmente, questo richiede una maggiore attenzione nel progetto del ciclo, la cui complessità cresce enormemente insieme al numero dei parametri. Nel seguente esempio, la variabile `i` assume i valori tra 0 e 59 con incrementi unitari, mentre la variabile `j` parte da 1 e viene moltiplicata per 2 a ogni ciclo:

```
public class PotenzeDiDue {
    public static void main(String argv[]) {
```

```

for ( long i = 0, j = 1; i < 60; i++, j *= 2)
    System.out.println("due elevato alla " + i + " uguale a " + j);
}

```

Si noti che la variabile *j* non viene valutata all'interno della condizione booleana: in altre parole, la durata del ciclo viene stabilita unicamente mediante la variabile *i*. Pertanto è possibile riscrivere l'esempio precedente ricorrendo a un unico contatore tra i parametri del ciclo `for`, trasferendo l'istruzione di incremento della variabile *j* all'interno del corpo del ciclo:

```

public class PotenzeDiDue {
    public static void main(String argv[]) {
        long j = 1;
        for ( long i = 0 ; i < 60; i++) {
            System.out.println("due elevato alla " + i + " uguale a " + j);
            j *= 2;
        }
    }
}

```

La scelta tra le due forme va effettuata caso per caso, tenendo presente che spesso è preferibile privilegiare la chiarezza rispetto alla concisione (meglio scrivere più righe che correre il rischio di introdurre errori).

Omissione di parametri

È possibile lasciare in bianco uno o più parametri del `for`. Se la variabile che si usa come indice è già stata dichiarata, è possibile omettere l'inizializzazione:

```

int i = 0;
for ( ; i < 10 ; i++)
    System.out.println(i);

```

Se l'istruzione di incremento è presente all'interno del ciclo, si può lasciare in bianco il terzo parametro:

```

for ( int i = 0 ; i < 10 ; ) {
    System.out.println(i);
    i++;
}

```

Infine, se si lascia in bianco la condizione booleana, il ciclo verrà eseguito all'infinito:

```

for ( int i = 0 ; ; i++)
    System.out.println(i);

```


Uso di break

L'istruzione `break`, se inserita all'interno di un ciclo `while`, `do - while` o `for`, ha l'effetto di interrompere l'iterazione, e di ricominciare dall'istruzione immediatamente successiva. L'istruzione `break` consente di inserire condizioni di uscita supplementari, localizzate in punti diversi dall'inizio o dalla fine del ciclo.

Un uso tipico di `break` è all'interno di algoritmi di ricerca lineare, ossia algoritmi che scandiscono uno a uno gli elementi di un vettore fino a quando non trovano un particolare valore. In prima istanza, una valida soluzione al problema è un algoritmo del tipo:

```
boolean found = false;
for ( int i = 0; i < array.length ; i++ )
    if(array[i] == 101)
        found = true;
if(found)
    System.out.println("Il vettore contiene il valore 101");
```

Dato un array di interi, lo si scandisce dal primo all'ultimo elemento, controllando se uno di essi ha valore 101. Se viene trovato, la variabile booleana `found` assume il valore `true`; in caso contrario, al termine del ciclo esso conterrà il valore `false`. L'ultima istruzione stamperà una scritta se la variabile `found` è `true`. Se il vettore è molto grande (si immagini un vettore da 10 o 20 milioni di elementi) e il valore viene trovato dopo appena un centinaio di iterazioni, tutte le iterazioni successive risulteranno inutili. Inserendo un'istruzione `break`, si può fare in modo che il ciclo duri il minimo necessario per trovare l'elemento, con il vantaggio di ridurre il tempo di esecuzione.

```
boolean found = false;
for ( int i = 0; i < array.length ; i++ )
    if(array[i] == 101) {
        found = true;
        break; // il ciclo finisce qui
    }
if(found)
    System.out.println("Il vettore contiene il valore 101");
```

L'istruzione continue

L'istruzione `continue` ha un effetto simile a quello di `break`, ma invece di dirottare l'esecuzione al termine del ciclo, la riporta all'inizio e passa all'iterazione successiva. Così, se si desidera realizzare un ciclo che effettui il prodotto di tutti i valori di un vettore, trascurando gli elementi con valore zero, è sufficiente scrivere:

```
long prodotto = 0;
for ( int i = 0; i < array.length ; i++ ) {
```

```
if(array[i] == 0)
    continue; // salta direttamente all'iterazione successiva

    prodotto = prodotto * array[i];
}
```

Uso di break e continue in cicli nidificati

Quale effetto hanno le istruzioni `break` e `continue` in un sistema di cicli nidificati? Esse hanno effetto solo sul ciclo corrente. Se si colloca un'istruzione `break` in un ciclo esterno, essa provocherà la fine di entrambi i cicli e la prosecuzione del programma, mentre se si pone il `break` in un ciclo interno, provocherà l'interruzione del solo ciclo interno e la prosecuzione di quello esterno:

```
for ( int i = 0; i < 11; i++) { // ciclo esterno
    ...
    for ( int j = 0; j < 11; j++) { // ciclo interno
        ...
        break; // interrompe solo il ciclo interno
    }
    // dopo il break si riparte da qui
    ...
}
```

Un discorso del tutto simile vale per `continue`:

```
for ( int i = 0; i < 11; i++) { // ciclo esterno
    ...
    for ( int j = 0; j < 11; j++) { // ciclo interno
        ...
        continue; // va all'iterazione successiva del ciclo interno
        ...
    }
    ...
}
```

Uso di break e continue con label

Le istruzioni `break` e `continue` possono essere utilizzate in modo più avanzato grazie alle `label`. Una `label` è un identificatore seguito da un carattere `:` e un'istruzione, o un blocco di istruzioni tra parentesi graffe:

```
nome : istruzione;
```

L'istruzione, che in questa circostanza prende il nome di *labeled statement* (istruzione etichettata), può essere una qualsiasi istruzione Java, compreso un altro ciclo.

L'istruzione `break` può specificare attraverso una *label* quale istruzione si intende interrompere. Come si è già visto, l'effetto standard del `break` è quello di interrompere il ciclo corrente. Il `break` con etichetta permette di riproporre una variazione dell'esempio precedente in cui il `break`, sebbene posto nel ciclo più interno, provoca l'interruzione del ciclo più esterno. Per ottenere questo effetto, è necessario etichettare il ciclo più esterno con un nome (cosa che viene fatta nella prima riga) e specificare, dopo il `break`, l'etichetta assegnata al ciclo più esterno:

```
cicloEsterno :
for ( int i = 0; i < 11; i++) { // ciclo esterno
  ...
  for ( int j = 0; j < 11; j++) { // ciclo interno
    ...
    break cicloEsterno; // interrompe il ciclo esterno
  }
  ...
}
// dopo il break si riparte da qui
```

Anche l'istruzione `continue` prevede la variante con *label*; in questo caso, l'etichetta permette di indicare quale ciclo si intende proseguire:

```
cicloEsterno :
for ( int i = 0; i < 11; i++) { // ciclo esterno
  // dopo il continue si prosegue da qui
  ...
  for ( int j = 0; j < 11; j++) { // ciclo interno
    ...
    continue cicloEsterno; // riprende il ciclo esterno
  }
  ...
}
```

È inutile sottolineare come l'uso di queste istruzioni tenda a complicare enormemente i programmi laddove, nella maggior parte dei casi, è possibile formulare un costrutto equivalente. Le *labeled statement* sono un costrutto ereditato dal C, e non capita spesso di incontrarlo nei programmi Java.