Array, cicli e schemi di programmi*

Luca Tesei

26 novembre 2004

Sommario

In queste note vengono introdotti gli array così come vengono trattati in Java e le principali operazioni su di essi. Di seguito vengono introdotti i tre cicli disponibili in Java: `while`, `do` e `for` con le loro caratteristiche ed esempi tipici di uso. Infine vengono definiti alcuni semplici schemi di programmi per risolvere problemi di ricerca lineare.

1 Array

Finora abbiamo usato all'interno dei programmi le variabili di tipo `int`, `boolean` o riferimento ad oggetto. Può essere utile in certe applicazioni raggruppare un certo numero di variabili dello stesso tipo in un'unica variabile. Nella programmazione classica questo tipo di struttura si chiama *array* (in italiano si può tradurre con vettore). In matematica un vettore **v** (ad esempio di numeri reali) di dimensione n si può indicare con una scrittura del tipo $(v_0, v_1, \ldots v_{n-1})$ e il generico elemento si indica con v_i dove i è un *indice* che può assumere valori nell'insieme $\{0, 1, \ldots, n\}$.

Nei linguaggi di programmazione di solito si indica il generico elemento del vettore **v** con la scrittura `v[i]` dove `v` è una variabile di tipo array e `i` è una variabile di tipo `int` che deve avere un valore compreso tra `0` ed `n - 1` se `n` è la dimensione dichiarata dell'array. Il generico elemento `v[i]` è proprio una variabile di tipo T dove T è il tipo indicato nella dichiarazione dell'array. Come tutte le variabili `v[i]` deve essere inizializzata, può essere assegnata con un valore di tipo T e può essere riferita all'interno di un'espressione. In quest'ultimo caso il valore semantico è il valore che è presente in quel momento nella posizione `i` dell'array.

*Note parzialmente tratte da Cay S. Horstmann, "Concetti di informatica e fondamenti di Java 2", Apogeo.

Nei linguaggi non ad oggetti come il Pascal o il C gli array sono variabili speciali che fanno parte dello stato allo stesso modo delle variabili "normali". In Java invece gli array sono *oggetti*. Come tutti gli oggetti devono essere creati e sono raggiungibili attraverso variabili che contengono i riferimenti.

Introduciamo la sintassi per la dichiarazione di array:

```
Decl ::= T[] Ide;
       | T Ide[];
       | T[] Ide = new T[Exp];
       | T Ide[] = new T[Exp];

T::= byte | short | int | long | float | double | char
   | boolean | Ide
Ide ::= <Identificatori>
Exp ::= <Espressioni>
```

Una dichiarazione del tipo `int[] a;` (oppure `int a[];`) crea, nel frame in testa alla pila di frame dell'attivazione corrente, un'associazione per il nome `a` con il valore $\overline{\omega}$[1]. La variabile `a` può quindi contenere un riferimento ad un oggetto array di interi.

Una dichiarazione del tipo `Foo[] fa;` crea una variabile che può contenere un riferimento ad un oggetto array di variabili riferimento. Ogni elemento di un oggetto array puntato da `fa` può quindi contenere un riferimento ad un oggetto della classe `Foo`.

Come per le altre variabili di frame, contestualmente alla dichiarazione si può anche creare un oggetto: `int[] a = new int[3];` crea la variabile `a` nel frame corrente e crea un oggetto array nello heap. Il riferimento al nuovo oggetto array viene assegnato alla variabile `a`. L'array creato nello heap contiene tre variabili intere inizializzate con il valore di default per le variabili istanza intere, cioè. L'effetto della dichiarazione è visualizzato in Figura 1.

La dichiarazione `Foo[] fa = new Foo[4];` crea la variabile `fa` nel frame corrente e crea un oggetto array nello heap. Il riferimento al nuovo oggetto viene assegnato alla variabile `fa`. L'array creato nello heap contiene quattro variabili riferimento inizializzate con `null`[2]. Queste variabili potranno contenere riferimenti ad oggetti della classe `Foo`. L'effetto della dichiarazione è visualizzato in Figura 2.

[1] Assumiamo che per le variabili di frame di qualunque tipo questo simbolo indichi un valore non inizializzato

[2] Il valore di default per le variabili istanza riferimento ad oggetti

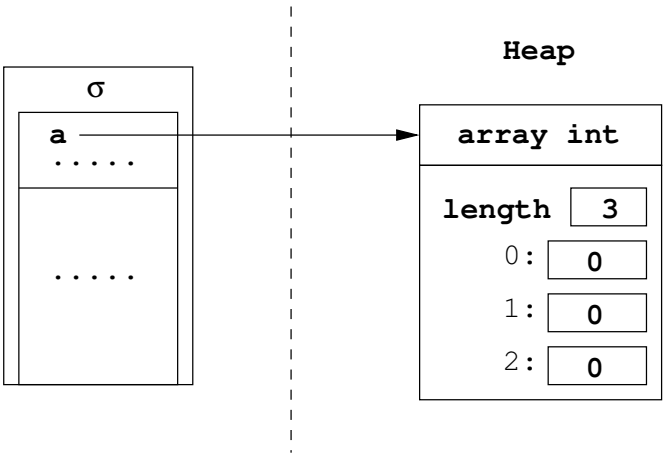

Figura 1: Effetto della dichiarazione `int[] a = new int[3];`

Come si vede dalla definizione della sintassi, nel linguaggio Java sono previsti due modi per la dichiarazione di array: è possibile porre le parentesi quadre vuote (`[]`) sia prima che dopo il nome dell'identificatore che rappresenta il nome della variabile riferimento ad array che si sta creando. Ad esempio le dichiarazioni possono essere scritte anche come segue:

```
int a[];
boolean b[];
// oppure con inizializzazione
int a1[] = new int[3];

Foo fa[];
// oppure con inizializzazione
Foo fa1[] = new Foo[4];
```

Ogni oggetto array ha una variabile istanza intera `length` che contiene la dimensione dell'array, vale a dire il numero di elementi dello stesso. Quindi è sempre possibile valutare l'espressione `a.length` per sapere la dimensione dell'array puntato da `a`. È importante notare a questo punto che *la dimensione di un array viene specificata quando viene creato e non può più essere modificata*[3]. Tuttavia gli array di Java hanno la possibilità, non presente ad esempio in C o in Pascal, di poter essere creati con una dimensione qualsiasi non nota a priori al compilatore (come deve essere per gli array in C o Pascal).

[3]La distribuzione di Java della Sun fornisce le classi `java.util.Vector` e `java.util.ArrayList` i cui oggetti sono array la cui dimensione può essere estesa dopo che questi sono stati creati. Consultare le API

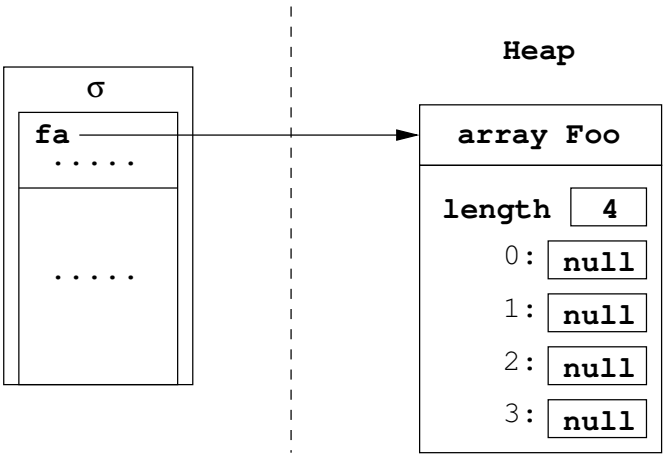

Figura 2: Effetto della dichiarazione `Foo[] fa = new Foo[4];`

Quando un array è stato creato si possono assegnare i valori del tipo giusto agli elementi dell'array. Per far questo il comando da utilizzare è ovviamente l'assegnamento, ma, a differenza dell'assegnamento di campi di oggetti normali(la referenziazione con il `.`), si usa la sintassi specifica: `a[0] = 3;` assegna il valore 3 al primo elemento dell'array puntato da `a`. È importante notare che gli elementi di un qualsiasi array sono numerati da 0 ad $n-1$ dove n è la lunghezza dell'array specificata nel campo `length`. Ogni tentativo di leggere/scrivere un elemento al di fuori di questo intervallo provoca un errore a run time (viene sollevata un'eccezione). All'interno delle parentesi quadre può essere inserita una qualunque espressione che abbia un valore intero compreso nell'intervallo $[0, N-1]$. Consideriamo il seguente frammento di codice:

```
...
int i=0;
a[i]=3;                                  (1)
a[i+1]=5;                                (2)
a[a.length - 1]= (i + 2) * a.length;  (3)
...
```

In Figura 3 è mostrata la situazione dopo l'esecuzione del frammento di codice. Come si vede si dichiara una variabile `i` che viene inizializzata a 0. L'assegnamento `(1)` inserisce il valore 3 nella posizione `i` dell'array puntato da `a`. Dato che il valore dell'espressione `i` è 0, sarà l'elemento in posizione 0 dell'array ad essere modificato. L'assegnamento `(2)` assegna il valore 5 alla posizione `i + 1` dell'array puntato da `a`. Il valore dell'espressione `i + 1` nello stato corrente è 1, quindi sarà l'elemento in posizione 1 dell'oggetto array ad

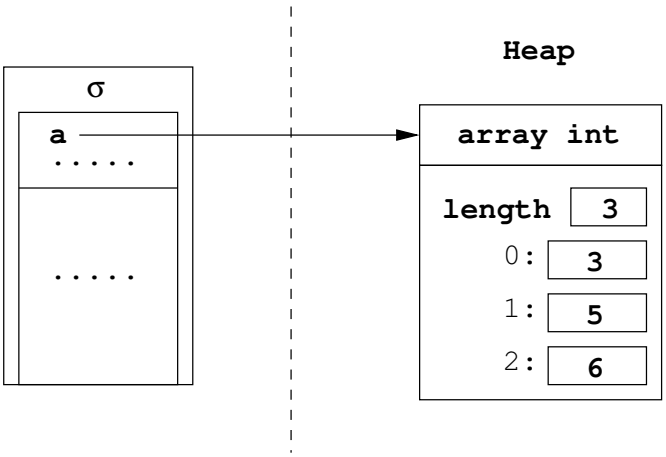

Figura 3: Effetto degli assegnamenti `(1)`, `(2)` e `(3)`

essere modificato. L'assegnamento `(3)` assegna il valore 6 all'elemento in posizione 2 dell'array (cioè l'ultimo). Questo perché il valore dell'espressione `a.length - 1` nello stato corrente è 2 e l'espressione a destra dell'"'=" vale 6 nello stato corrente. È importante notare a questo punto la differenza fra il campo `length` dell'oggetto array e gli elementi dell'array stesso. Il campo `length` viene considerato come un campo qualsiasi di un oggetto e viene quindi raggiunto tramite l'operatore ".". Invece gli elementi dell'oggetto array vengono raggiunti con l'operatore "`[ E ]`" dove `E` è un'espressione che deve avere un valore intero.

Se un array non è di tipo `int` o uno degli altri tipi base, allora i suoi elementi sono puntatori ad oggetti di una certa classe. Consideriamo l'array `fa` dichiarato sopra e supponiamo che la classe `Foo` sia definita come segue:

```
public class Foo {
   public int z;
   public boolean b;
   public void condInc(int y) {
      if (this.b) this.z = this.z + y;
   }
}
```

Possiamo usare gli elementi dell'array `fa` per mantenere dei riferimenti ad oggetti della classe `Foo`. Consideriamo il seguente frammento di codice:

```
...
fa[0] = new Foo;  (1)
```

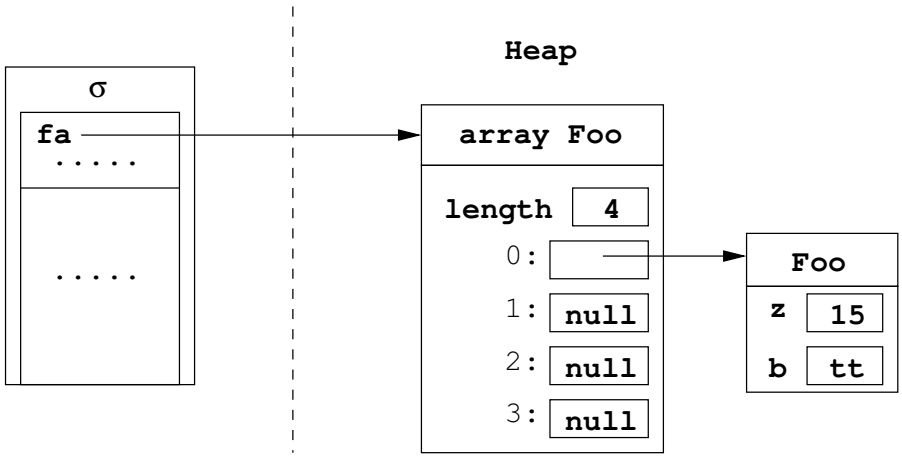

Figura 4: Effetto degli assegnamenti `(1)`, `(2)` e `(3)`

```
fa[0].z = 15;      (2)
fa[0].b = true;    (3)
fa[1] = fa[0];     (4)
fa[2] = new Foo;   (5)
...
```

L'assegnamento `(1)` crea un oggetto della classe `Foo` e inserisce il suo riferimento nell'elemento 0 dell'array puntato da `fa`. I due successivi assegnamenti inizializzano i valori dei campi dell'oggetto appena creato. È da notare come i campi vengano raggiunti a partire dal riferimento contenuto in `fa[0]` e a cui correttamente viene applicato l'operatore ".". A sua volta il riferimento `fa[0]` viene ottenuto dal riferimento `fa` all'array e dall'operatore `[]`. In Figura 4 è mostrato lo stato dopo l'esecuzione dell'assegnamento `(3)`. L'assegnamento `(4)` opera a livello di riferimenti, nel senso che assegna all'elemento 1 dell'array `fa` il valore dell'espressione `fa[0]`. Questa espressione, se valutata nello stato corrente, restituisce il valore dell'elemento 0 dell'array `fa`, vale a dire il riferimento all'oggetto creato in `(1)`. L'effetto sarà quindi quello di avere due riferimenti a tale oggetto: uno è quello che già avevamo e l'altro si troverà nell'elemento 1 dell'array. L'assegnamento `(5)` ha l'effetto di creare un nuovo oggetto della classe `Foo` e di inserire il suo riferimento nell'elemento 2 dell'array. Il risultato è mostrato in Figura 5.

Vediamo a questo punto come si chiamano i metodi degli oggetti della classe `Foo` tramite i riferimenti che abbiamo nell'array. Consideriamo i seguenti comandi a partire dallo stato rappresentato in Figura 5:

```
fa[2].z = 5;              (6)
fa[1].condInc(fa[2].z);   (7)
```

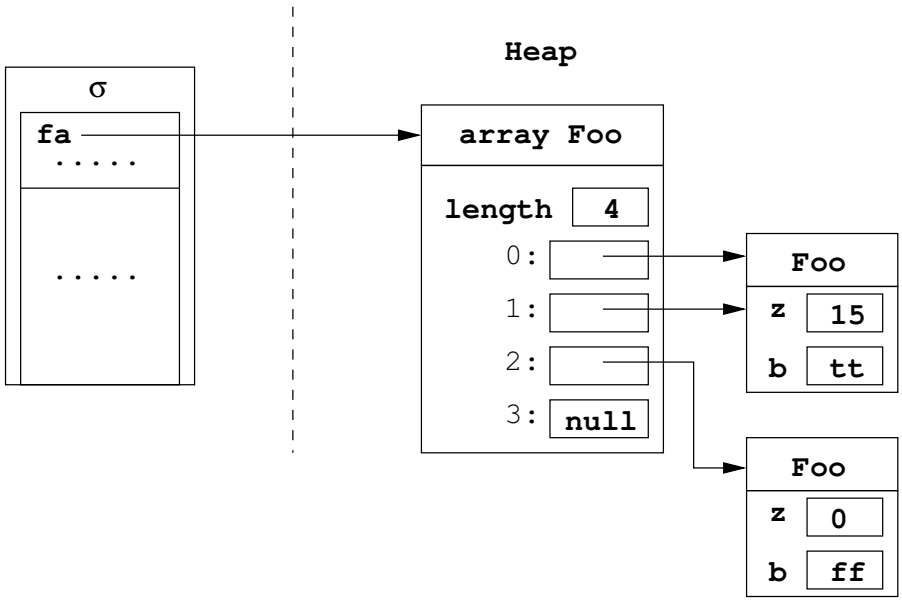

Figura 5: Effetto degli assegnamenti `(4)` e `(5)`

L'assegnamento `(6)` assegna il campo `z` del secondo oggetto creato con il valore 5. Il comando `(7)` è la chiamata del metodo `condInc` dell'oggetto riferito dall'elemento 1 dell'array. Tale oggetto, come risulta dallo stato, è il primo oggetto che è stato creato e che è riferito anche dal primo elemento dell'array. Il metodo incrementa il campo `z` dell'oggetto con il valore passato nella chiamata solo se il campo `b` dell'oggetto ha valore vero. Il valore passato al metodo è il valore dell'espressione `fa[2].z`, cioè il valore del campo `z` del secondo oggetto creato. Tale valore è 5. Il risultato è mostrato in Figura 6.

2 Cicli

In questa sezione impareremo a scrivere dei programmi che eseguono ripetutamente uno o più enunciati (comandi). I costrutti che si comportano in questo modo sono detti cicli e in Java ci sono tre tipi di cicli: il `while` il `do` e il `for`. Vedremo che in realtà sono molto simili l'uno all'altro; in particolare il ciclo `while` è quello più generale di tutti mentre gli altri due sono due casi particolari, che occorrono frequentemente nei programmi, del ciclo `while`.

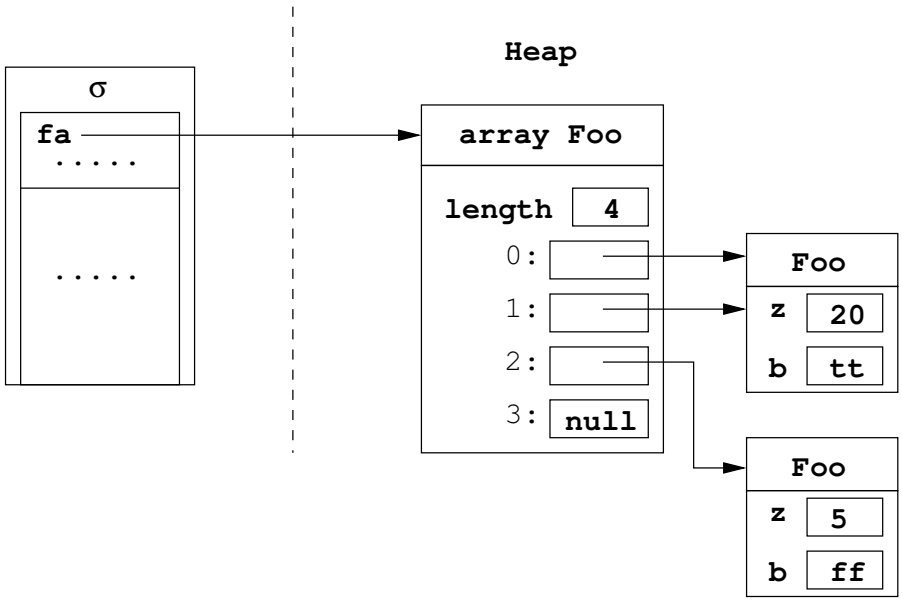

Figura 6: Effetto degli assegnamenti `(6)` e `(7)`

Anno	Saldo
0	10.000,00
1	10.500,00
2	11.025,00
3	11.576,25
4	12.155,06
5	12.762,82

Tabella 1: Crescita negli anni di un investimento.

2.1 Ciclo `while`

Consideriamo un investimento che parte con un capitale iniziale a cui ogni anno sono aggiunti gli interessi ottenuti secondo un tasso annuale fisso. Per fissare dei valori supponiamo che il capitale iniziale sia di 10.000 euro e il tasso annuale fisso sia del 5%. La crescita negli anni dell'investimento è raffigurata nella tabella 1: ogni anno l'interesse viene calcolato sul capitale dell'anno precedente più gli interessi maturati (capitalizzazione).

Vogliamo fare un programma che calcoli quanti anni occorrono affinché il saldo diventi almeno 20.000 euro. Come possiamo fare? Dobbiamo fare in modo che la macchina astratta Java capitalizzi gli interessi di ogni anno e conti gli anni *fino a quando* il saldo non supera i 20.000 euro. Questo è il

tipico problema la cui soluzione si ottiene tramite un ciclo `while`. Estendiamo la sintassi dei comandi con questo nuovo costrutto:

```
Com ::= while (BoolExpr) Block_Or_Com;
      | ...
```

Dopo la parola riservata `while` va inserita una espressione di tipo `boolean` fra parentesi (esattamente come per il costrutto `if`) seguita da un comando singolo o un blocco (anche qui come il costrutto `if`). L'espressione fra parentesi viene spesso chiamata *guardia* del `while` e il blocco (o comando) che segue viene spesso chiamato *corpo* del `while`. Il significato di un costrutto di questo tipo, ad esempio `while (E) C;`, è il seguente:

1. Viene valutata l'espressione `E`
2. Se il valore è `false` il comando `while` non fa niente (in particolare `C` non viene eseguito) e l'esecuzione prosegue con il comando successivo
3. Se il valore è `true` allora viene eseguito il comando `C` (o il blocco, se è un blocco) dopodiché l'esecuzione non passa al comando successivo, ma si ritorna ad eseguire l'intero `while` dal punto 1.

Il problema del nostro esempio può essere risolto *in generale* con questo codice:

```
int years = 0;
while (balance < targetBalance) {
  years++;
  double interest = balance * rate / 100;
  balance = balance + interest;
}
```

All'inizio controlliamo se `balance` (il saldo attuale) è minore del `targetBalance` (il saldo che vogliamo raggiungere nell'investimento: nel nostro esempio il saldo attuale, all'inizio, è di 10.000 euro e il `targetBalance` è di 20.000 euro). Se questo è falso, cioè se già il saldo attuale ha raggiunto l'obiettivo, il `while` non fa niente e, alla sua terminazione, `years` contiene il numero di anni necessari per arrivare al saldo voluto (cioè 0).

Se invece, come è più plausibile, la condizione è vera allora viene eseguito il corpo del `while` che semplicemente capitalizza gli interessi di un anno e incrementa di uno `years`. Alla fine dell'esecuzione del blocco si ripassa a controllare la condizione. Se ancora non è stato raggiunto l'obiettivo allora il blocco viene eseguito di nuovo capitalizzando gli interessi dell'anno successivo

(calcolati sul capitale più gli interessi dell'anno precedente). Di nuovo, alla fine del blocco, si ripassa a controllare se l'obiettivo è stato raggiunto. Il ciclo va avanti fino a quando la condizione non diventa falsa, cioè fino a quando l'obiettivo è stato raggiunto. All'uscita dal ciclo la variabile `years` conterrà un valore intero che indica il numero di anni necessari per arrivare all'obiettivo.

Vediamo tutto il programma che risolve questo problema

```
/** Una classe per controllare la crescita di un investimento
    che accumula interessi a un tasso annuale fisso */
public class Investimento  {
  /** Costruisce un Investimento con un saldo iniziale e un
      tasso di interesse fisso
      @param aBalance saldo iniziale
      @parem aRate il tasso di interesse annuale fisso */
  public Investimento(double aBalance, double aRate) {
    balance = aBalance;
    rate = aRate;
    years = 0;
  // Anni necessari inizialmente posti a zero
  }

  /** Continua ad accumulare interessi finch&eacute; il
      saldo non raggiunge un valore desiderato
      @param targetBalance il saldo desiderato */
  public void waitForBalance(double targetBalance) {
    while ( balance < targetBalance ) {
      years++;
      double interessi = balance * rate / 100;
      balance += interessi;
    }
  }

  /** Restituisce il saldo attuale dell'Investimento
      @return il saldo attuale */
  public double getBalance() {
    return balance;
  }

  /** Restituisce il numero di anni per i quali
      l'investimento ha accumulato interessi
```

```
     @return il numero di anni trascorsi dall'inizio
             dell'investimento
   */
  public int getYears() {
    return years;
  }
  // Variabili istanza
  private int years;

  private double balance;

  private double rate;
}
```

L'inconveniente maggiore che può capitare usando dei cicli è che il ciclo non termini mai! Quando ciò accade si dice che "il programma è andato in ciclo" e di solito sembra che il pc si sia bloccato. In realtà sta lavorando ed eseguendo sempre lo stesso blocco di un `while` (o di un altro ciclo) senza che la condizione divenga mai falsa. Se il corpo contiene stampe allora si vedranno scorrere velocissime delle righe sullo standard output e se ci sono altri comandi "pesanti" all'interno del corpo, come ad esempio scritture di file o interazioni con il sistema operativo, è possibile addirittura che l'intero pc si blocchi! In genere il sistema operativo mette a disposizione un modo per terminare forzatamente un programma (in ambiente Linux/Unix premendo Ctrl-c nella console da cui abbiamo lanciato la virtual machine; in ambiente Windows aprendo il Task Manager e terminando il processo corrispondende al nostro programma).

È importante cercare in tutti i modi di prevenire queste situazioni. Ci possono certo essere dei casi in cui si vuole che un programma non termini mai (es. gestore di un bancomat, controllore di una centrale nucleare, ecc.), e allora sarà opportuno scrivere dei cicli che non terminano mai. Ma nella maggior parte dei casi si suppone che un ciclo debba terminare.

La non terminazione di un ciclo è spesso dovuta a un errore logico del programmatore. Ad esempio quando si dimentica di incrementare la variabile che viene controllata nella condizione del ciclo:

```
int years = 0;
while (years < 20) {
  // manca years++; !!!
  double interest = balance * rate / 100;
  balance = balance + interest;
}
```

In questo caso si voleva calcolare il valore dell'investimento dopo 20 anni, ma la variabile `years` usata nel controllo per l'uscita del ciclo non è stata incrementata nel corpo e continua ad avere valore 0 tutte le volte. Quindi la condizione `years < 20` è sempre vera e il ciclo non esce mai.

Un altro errore tipico è quello di incrementare invece di decrementare la variabili usata per il controllo:

```
int years = 20;
while (years > 0) {
  years++; // Errore!!! doveva essere years--;
  double interest = balance * rate / 100;
  balance = balance + interest;
}
```

Nel progetto di un ciclo una parte importante è determinare la guardia del ciclo stesso in modo che esso termini e che venga eseguito il numero giusto di volte. Spesso le guardie utilizzano gli operatori `>` o `<` oppure `<=` o `>=`. La decisione di quale operatore usare richiede un piccolo ragionamento sul ciclo per non commettere i cosiddetti "errori per scarto di uno", cioè errori dovuti all'esecuzione del ciclo una volta in più o una in meno del dovuto.

Prendiamo ad esempio un programma che voglia calcolare in quanti anni un certo capitale iniziale raddoppia:

```
int years = 0;
double balance = initialBalance; // capitale iniziale
while (balance < initialBalance * 2) {
  years++;
  double interest = balance * rate / 100;
  balance = balance + interest;
}
```

Le domande a cui bisogna rispondere con cognizione di causa sono:

1. Il conteggio degli anni, nella variabile `years`, deve cominciare da 0 o da 1?

2. La guardia giusta è `balance < initialBalance * 2` oppure `balance <= initialBalance * 2`?

Per rispondere adeguatamente un buon metodo è quello di considerare un caso tipico con valori semplici e provare ad eseguire i passi del ciclo a mano per vedere se l'inizializzazione e la guardia si comportano come si vuole. Prendiamo ad esempio un saldo iniziale di 100 euro e un interesse del

50% (è bene scegliere i valori che semplificano al massimo i calcoli e che ci permettano di concentrarci sulla logica del ciclo). Dopo un anno il saldo sarà 150 e dopo due anni 225. Quindi l'investimento raddoppia dopo due anni, in questo caso e il corpo del ciclo deve essere eseguito due volte per arrivare al risultato giusto. Siccome il corpo del ciclo incrementa `years` tutte le volte allora significa che essa deve essere inizializzata con il valore 0.

Notiamo inoltre che `balance`, quando viene valutata la guardia del ciclo, contiene sempre il valore del saldo *dopo* il termine dell'anno considerato: all'inizio contiene il valore del saldo *dopo* 0 anni di investimento, non dopo 1 anno. Dopo ogni esecuzione del corpo del ciclo contiene il valore dell'investimento *dopo* l'anno indicato dal valore della variabile `years`. E, infine, all'uscita del ciclo contiene il valore del saldo *dopo* l'ultimo anno del numero di anni indicati da `years` (questa asserzione viene detta *invariante* del ciclo perché rimane sempre vera immediatamente prima, durante e dopo il ciclo stesso).

Passiamo ora all'operatore relazionale: `<` o `<=`? Bisogna considerare il caso in cui alla fine di un certo anno il saldo sia *esattamente* uguale al doppio del saldo iniziale. Naturalmente questo è un caso che può verificarsi. Ragioniamo sulla logica del programma: se dopo un certo anno il saldo è *esattamente* il valore doppio del saldo iniziale il programma non deve eseguire un'altra volta il ciclo! Il ciclo si deve interrompere. Siccome si interrompe quando la condizione diventa falsa dobbiamo fare in modo che sia falsa anche in questo caso particolare. Quindi l'operatore corretto da inserire è `<`, non `<=` (se usassi questo il corpo verrebbe eseguito una volta in più in quei casi in cui si raggiunge esattamente il doppio dell'investimento iniziale in un certo anno).

2.2 Il ciclo `do`

Talvolta si vuole eseguire il corpo di un ciclo almeno una volta prima di controllare la guardia. Il ciclo `do` serve esattamente a questo:

```
Com ::= do Com_Or_Block while (BoolExpr);
        | ...
```

Un ciclo `do C while (E);` funziona in questo modo:

1. Viene eseguito il comando (o il blocco) `C`
2. Viene valutata l'espressione `E`
3. Se il valore è `false` allora il ciclo esce e l'esecuzione passa all'istruzione successiva

4. Se il valore è `true` allora si riparte dal punto 1.

Facciamo un esempio di uso: supponiamo di voler ottenere in input un valore intero con il vincolo che non sia negativo. Possiamo pensare di continuare a chiederlo fino a quando l'utente non ne inserisce uno positivo:

```
double value;
do {
  String input = JOptionPane.showInputDialog(
                  ''Inserisci un numero positivo'');
  value = Double.parseDouble(input);
} while (value <= 0);
```

All'inzio viene richiesto il numero e il valore immesso viene trasformato in `double`. Poi si passa a controllare la condizione: se il valore immesso non è positivo allora la condizione risulta vera e il ciclo riparte richiedendo di nuovo il valore. Andrà avanti così fino a quando il valore immesso non sarà positivo (e quindi la guardia sarà falsa).

In realtà possiamo ottenere lo stesso comportamento anche con un normale ciclo `while` avendo l'accortezza di usare una variabile `boolean` di supporto:

```
boolean done = false; // indica se e' stato inserito un valore
                      // corretto
while(!done) {
  String input = JOptionPane.showInputDialog(
                  ''Inserisci un numero positivo'');
  value = Double.parseDouble(input);
  if (value > 0) done = true;
}
```

Il ciclo uscirà quando la guardia sarà falsa cioè quando `done` avrà valore `true` e l'`if` assegna a `done` valore `true` solo quando è stato inserito un valore positivo.

In Java non è previsto l'uso del `goto`: non è possibile cioè etichettare con un nome una certa istruzione e poi saltare a quella istruzione da un qualsiasi altro punto del programma. In molti altri linguaggi questo tipo di costrutto è permesso e in linguaggi di basso livello tipo l'Assembly è l'unico modo di creare dei cicli. Tuttavia usando il `goto` potrebbero essere scritti programmi che hanno una struttura non lineare e non riconducibile a nessun ciclo strutturato. Alcuni programmatori non si preoccupano di questo fatto perché secondo loro con il `goto` si riesce a scrivere codice più compatto. Tuttavia è sconsigliabile l'uso di un costrutto del genere in un linguaggio strutturato e di altro livello e il Java, in particolare, non lo ammette.

2.3 Il ciclo `for`

Il ciclo `for` è un'abbreviazione di un ciclo `while` che viene spessissimo usato sempre in una certa forma. La sintassi del `for` è la seguente:

```
Com ::= for (Com_or_Decl; Exp; Com) Com;

Com_or_Decl ::= Com | Decl
```

Il comando `for (C1; E; C2) C;` consiste nell'eseguire il comando (o la dichiarazione) `C1` (detto anche inizializzazione) in un nuovo blocco e, a partire dal nuovo stato ottenuto, eseguire il ciclo `while (E) { C; C2; }`. Nella pratica il comando `for` viene usato quasi sempre in questa forma:

```
for (i = inizio; i <= fine; i++) {
  ...
}
```

In alcuni casi la variabile `i` viene dichiarata nell'inizializzazione:

```
for (int i = inizio; i <= fine; i++) {
  ...
}
```

Ad esempio usiamo questo ciclo per determinare il valore di un nostro investimento dopo `n` anni:

```
for (int i = 1; i <= n; i++) {
  double interest = balance * rate / 100;
  balance += interest;
}
```

Innanzitutto facciamo un'osservazione sintattica. Il comando `i++` non è seguito dal punto e virgola. Questo è lecito, ma non è in accordo con la sintassi solita che conosciamo. In realtà in Java in quel punto non è richiesto il punto e virgola poichè c'è la parentesi tonda che chiude il contesto. Inoltre, nel caso in cui la variabile `i` venga dichiarata nell'inizializzazione, la sua visibilità è circoscritta al ciclo `for`: in pratica ogni ciclo `for` viene eseguito in un nuovo blocco (creato appositamente dal compilatore) che finisce nello stesso punto dove finisce il `for`. Il blocco del corpo, se presente, rappresenta un ulteriore blocco annidato.

Abbiamo detto che un ciclo `for` non è nient'altro che un'abbreviazione per una forma particolare di `while`: si può sempre scrivere un ciclo `for` come ciclo `while` con questo schema:

```
{
i = inizio;
while (i <= fine) {
...
i++;
}
}
```

Nel nostro esempio otterremmo:

```
{
int i = 1;
while (i <= n) {
  double interest = balance * rate / 100;
  balance += interest;
  i++;
}
}
```

Ci sono anche altre varianti. Ad esempio si può anche contare all'indietro:

```
for (int i = n; i >= 1; i--) {
  double interest = balance * rate / 100;
  balance += interest;
}
```

Oppure si può incrementare anche di frazioni di unità:

```
for (double x = 0.0; x <= n; x = x + 0.5) {
  ...
}
```

In realtà si potrebbe scrivere del codice estraneo nelle diverse parti del ciclo:

```
for (rate = 5; years-- > 0; System.out.println(balance) {
  ...
}
```

ma ovviamente un uso di questo genere è vivamente **sconsigliato**.

2.3.1 Ciclo `for` e array

Uno dei vantaggi dell'uso degli array è la possibilità di scrivere concisamente del codice che deve eseguire alcune operazioni su tutto un insieme di variabili dello stesso tipo. Un uso tipico del ciclo `for` riguarda le operazioni su array. Un esempio tipico è il seguente:

```
int[] a = new int[10];
for (int i = 0; i < a.length; i++)
   a[i] = i;
```

La prima riga del frammento di codice dichiara una variabile riferimento ad un array di interi e crea l'oggetto array. Lo scopo del ciclo `for` in questo caso è quello di inizializzare tutti gli elementi dell'array con un valore pari alla propria posizione nell'array. Il comando di inizializzazione è la dichiarazione della variabile intera `i` che viene inizializzata a 0 e farà da *indice* di scorrimento per eseguire l'operazione di inizializzazione di tutti gli elementi dell'array. La condizione del ciclo infatti diventerà falsa non appena `i` assumerà il valore `a.length`. Il comando `C2` è l'incremento dell'indice `i`. Questo viene eseguito ad ogni iterazione del ciclo *dopo* che è stato eseguito il comando `C`, che in questo caso particolare è l'assegnamento all'`i`-esimo elemento dell'array del valore `i`.

Seguiamo passo passo cosa avviene. Immediatamente prima del `for` si ha che $\forall i \in [0, \texttt{a.length} - 1].\texttt{a}[\texttt{i}] = \overline{\omega}$. Dopo l'inizializzazione di `i` il ciclo parte poiché ovviamente $\texttt{i} = 0 < \texttt{a.length} = 10$. Viene quindi eseguito il comando `a[i]=i;` in uno stato in cui `i`=0. L'effetto sarà quello di assegnare il valore 0 all'elemento 0 dell'array. A questo punto va eseguito il comando `C2`, cioè `i++`. L'effetto sarà quello di arrivare in uno stato in cui `i`=1. A questo punto bisogna rivalutare l'espressione `E` in questo stato. Essa sarà ancora vera e quindi si farà un'altra iterazione. La seconda iterazione quindi consiste nell'eseguire `a[i]=i;` in uno stato in cui `i`=1. L'effetto sarà quello di assegnare il valore 1 all'elemento 1 dell'array. Si prosegue eseguendo di nuovo `i++` giungendo in uno stato in cui `i`=2. Si ricomincia rivalutando l'espressione in questo nuovo stato. A questo punto è facile convincersi che il tutto andrà avanti fino all'iterazione in cui viene eseguito `a[i] = i;` in uno stato in cui `i`=9. Dopo questo assegnamento avremo completato l'inizializzazione di tutti gli elementi dell'array. A questo punto il comando `i++` porterà in uno stato in cui `i`=10 e la guardia risulterà falsa. All'uscita del `for`, quindi, la variabile `i` scompare e si avrà che $\forall \texttt{i} \in [0, \texttt{a.length} - 1].\texttt{a}[\texttt{i}] = \texttt{i}$, come volevamo.

Vediamo un altro esempio. Supponiamo di avere un array di interi già inizializzato e riferito dalla variabile `a`. Vogliamo scrivere un comando che

ponga a zero l'ultimo elemento dell'array e sposti tutti gli elementi di una posizione verso sinistra. Il primo valore dell'array viene perduto:

```
int i;
for (i = 0; i < a.length - 1; i++)
   a[i] = a[i+1];
a[i] = 0;
```

In questo caso la variabile `i` non è dichiarata nel blocco del `for` quindi essa rimarrà anche dopo l'esecuzione del ciclo e manterrà il suo valore. Il ciclo viene eseguito per tutti gli elementi dell'array tranne l'ultimo (l'elemento in posizione `a.length - 1`). Ad ogni iterazione l'elemento in posizione `i + 1` viene copiato nella posizione `i`. All'uscita del ciclo la variabile `i` varrà esattamente `a.length - 1` e quindi viene riusata per assegnare zero all'ultimo elemento dell'array, come volevamo.

2.3.2 Cicli `for` annidati

Spesso si usano cicli `for` annidati. Facciamo un semplice esempio: supponiamo di voler stampare la forma triangolare:

```
[]
[][]
[][][]
[][][][]
[][][][][]
[][][][][][]
[][][][][][][]
```

L'idea di base è quella di stampare un certo numero, diciamo `height`, di righe del triangolo:

```
for(int i = 1; i <= height; i++) {
  // Costruisce una riga del triangolo
  ...
}
```

Come fare per costruire la riga? Osserviamo che la riga numero 1 contiente 1 coppia di parentesi, che la riga numero 2 contiene 2 coppie di parentesi e questo è vero in generale: la riga numero `n` contiene `n` coppie di parentesi. Possiamo inserire un ciclo `for` interno che costruisce la riga concatenando parentesi quadre da 1 a `i`:

```
String r = ’’’’;
for(int i = 1; i <= width; i++) {
  // Costruisce una riga del triangolo
  for(int j = 1; j <= i; j++)
    r = r + ’’[]’’;
  r = r + ’’\n’’; // aggiunge il newline
}
System.out.println(r); // Stampa tutto il triangolo
```

Si noti che in questo caso il `for` interno ha un solo comando come corpo (`r = r + ’’[]’’;`) e quindi non è stato necessario introdurre un nuovo blocco.

Invece di creare sempre nuove stringhe (ricordiamoci che la concatenazione fra stringhe non modifica mai le stringhe operando, ma crea sempre stringhe nuove) possiamo usare oggetti della classe `java.lang.StringBuffer` la quale fornisce un buffer che può contenere una stringa a cui, fra le altre cose, si possono aggiungere caratteri in coda con il metodo `append`. Quando abbiamo finito di attaccare i pezzi in coda possiamo chiamare il metodo `toString` e ottenere la stringa finale:

```
StringBuffer r = new StringBuffer(); // Buffer inizialmente vuoto
for(int i = 1; i <= width; i++) {
  // Costruisce una riga del triangolo
  for(int j = 1; j <= i; j++)
    r.append(’’\n’’);
  r.append(’’[]’’); // aggiunge il newline
}
System.out.println(r.toString); // Stampa tutto il triangolo
```

Un esempio classico, ma un po’ più complicato, di uso di cicli `for` annidati è quello dell’algoritmo di ordinamento denominato *bubble-sort*. Supponiamo di avere un array di interi già inizializzato e riferito dalla variabile `a`. L’algoritmo di ordinamento ordina i valori effettuando degli scambi: se viene trovato in una posizione `i` un elemento che è maggiore di un altro che si trova in una posizione `j`, maggiore di `i`, allora i due vengono scambiati di posto. Se lo si fa per tutti i valori possibili di `i` e `j` alla fine si otterrà un array di elementi ordinato, vale a dire un array in cui $\forall \texttt{i}, \texttt{j} \in [0, \texttt{a.length} - 1].\texttt{i} \leq \texttt{j} \Rightarrow \texttt{a}[\texttt{i}] \leq \texttt{a}[\texttt{j}]$.

```
int i; // indice esterno
int j; // indice interno
int buf; // variabile di appoggio
for (i = 0; i < a.length -1; i++)
```

```
for (j = i + 1; j < a.length; j++)
   if (a[i] > a[j])
    { // scambio
      buf = a[i];
      a[i] = a[j];
      a[j] = buf;
    }
```

Il ciclo più esterno gestisce l'incremento dell'indice `i` che arriverà alla penultima posizione dell'array. Per ogni posizione `i` si esegue un ciclo a partire dalla posizione successiva (`i + 1`) fino alla fine dell'array. Questo ciclo interno usa l'indice `j`. Per ogni posizione `j` (sempre maggiore di `i`) si controlla se bisogna scambiare gli elementi in posizione `i` e `j` (cioè se quello in posizione, minore, `i` è un elemento maggiore di quello in posizione, maggiore, `j`). Per effettuare lo scambio bisogna usare una variabile di appoggio (`buf`) per mantenere il valore di `a[i]`. Una volta salvato quest'ultimo in `buf` la posizione `i` dell'array viene assegnata con il valore nella posizione `j` ed infine il valore salvato in `buf` viene posto in posizione `j`.

3 Schemi di programmi

Vediamo ora due schemi di programmi molto semplici che possono essere applicati ogni volta che si deve cercare, in una sequenza, la posizione del primo elemento che soddisfa una certa proprietà. Tale proprietà è definita sui valori della sequenza e di solito viene scritta come espressione booleana. Supponiamo di rappresentare la sequenza con un array riferito da una variabile `a`. Sia $P(v)$ una certa proprietà definita sui valori contenuti negli elementi di `a`. Il primo schema che consideriamo si chiama *ricerca lineare certa*. Esso si può applicare sotto l'ipotesi che esista almeno un elemento dell'array che soddisfa la proprietà P. Per trovare il primo elemento che soddisfa P si può allora scrivere il seguente codice:

```
int i = 0;
while (!P(a[i]))
  i = i + 1;
(1)
```

Nel punto `(1)` del programma si ha allora che la posizione `i` è quella in cui si trova il primo elemento dell'array che soddisfa P. Vediamo un esempio. Supponiamo di avere un array a valori interi e supponiamo che esista almeno un elemento pari. La nostra proprietà P sarà allora $P(n) : n \ \% \ 2 == 0$. Istanziando lo schema avremo il seguente codice:

```
int i = 0;
while (a[i] % 2 != 0)
  i = i + 1;
```

All'uscita del `while` avremo che la variabile `i` contiene la posizione del *primo* elemento pari dell'array.

Se non si ha la certezza che l'array contenga almeno un elemento che soddisfa la proprietà P allora bisogna applicare un altro schema denominato ricerca lineare incerta. Esso troverà, se esiste, il primo elemento dell'array che soddisfa P oppure, se non esiste nessun elemento dell'array che soddisfa P, esso conclude arrivando in uno stato in cui una variabile booleana ha valore falso:

```
int i = 0;
boolean trovato = false;
while (i < a.length && !trovato)
   if (P(a[i]))
      trovato = true;
   else
      i = i + 1;
(1)
```

Nel punto `(1)` del programma, se la variabile booleana `trovato` vale `tt`[4] allora il primo elemento dell'array che soddisfa P si trova nella posizione contenuta nella variabile `i`. Se la variabile `trovato` vale `ff`[5] allora si ha che nessun elemento dell'array soddisfa P.

Vediamo un esempio. Supponiamo di avere un array di interi e cerchiamo il primo elemento che soddisfa $P(n) : n > 100$.

```
int i = 0;
boolean trovato = false;
while (i < a.length && !trovato)
   if (a[i] > 100)
     trovato = true;
   else
     i = i + 1;
```

Questi schemi possono essere applicati anche se non si deve operare su un array, oppure alcune semplici varianti di essi possono essere applicate

[4]Indichiamo così il valore semantico di `true`.

[5]Valore semantico di `false`.

per verificare una proprietà più complessa. Un esempio di applicazione della ricerca lineare certa è quello in cui si cerca, in un intervallo $[0, n]$ di numeri naturali, la parte intera della radice quadrata di n. È chiaro che questo numero si trova necessariamente nell'intervallo $[0, n]$:

```
int i = 0;
while ((i+1)*(i+1) <= n)
  i = i + 1;
```

All'uscita del ciclo la variabile `i` contiene la parte intera della radice quadrata di n.

Una variante della ricerca lineare incerta può essere quella in cui si verifica se un certo array di interi è ordinato. In questo caso è sufficiente verificare che ogni elemento dell'array, tranne l'ultimo, sia minore o uguale del successivo. Quindi possiamo applicare la ricerca lineare incerta sull'array (trascurando l'ultimo elemento) e cercare se c'è un elemento che è maggiore del successivo. Se un tale elemento non è presente nell'intervallo $[0, \texttt{a.length} - 2]$, allora abbiamo verificato che l'array è ordinato:

```
int i = 0;
trovato = false;
while ( (i < a.length - 1) && !trovato)
  if (a[i] > a[i+1])
     trovato = true;
  else
     i = i +1;
```

All'uscita del ciclo l'array è ordinato se e solo se la variabile booleana `trovato` vale **ff**.

Lo schema della ricerca lineare incerta è un esempio del cosiddetto "ciclo e mezzo". Un ciclo e mezzo si ha quando la condizione che decide se si deve uscire o no dal ciclo assume il valore significativo durante l'esecuzione del corpo del ciclo stesso (né all'inizio né alla fine). In questi casi è opportuno, come abbiamo fatto, usare una variabile `boolean` di supporto che serva per "trasportare" la vera condizione di uscita nella guardia del ciclo.