

Capitolo 6

Iterazioni

Cay S. Horstmann

**Concetti di informatica e fondamenti di Java
quarta edizione**

Obiettivi del capitolo

- Saper programmare cicli con gli enunciati `while`, `for` e `do`
- Evitare cicli infiniti ed errori per scarto di uno
- Comprendere il funzionamento dei cicli annidati
- Capire come ricevere dati in ingresso
- Realizzare simulazioni
- Conoscere il debugger

Cicli `while`

- L'enunciato `while` esegue ripetutamente un blocco di codice.
- Una condizione di terminazione controlla quante volte viene eseguito il ciclo.

```
while (condizione)  
    enunciato;
```

- Il più delle volte l'enunciato è un blocco di enunciati, vale a dire una serie di enunciati racchiusa fra parentesi graffe {...}.

Calcolare la crescita di un investimento

- Saldo iniziale \$10,000, tasso di interesse annuo 5%

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

Calcolare la crescita di un investimento

- Quando il conto raggiunge un particolare saldo?

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

File Investment.java

```
01: /**
02:     Una classe per controllare la crescita di un investimento
03:     che accumula interessi a un tasso annuoale fisso.
04: */
05: public class Investment
06: {
07:     /**
08:         Costruisce un oggetto Investment con un saldo iniziale
09:         e un tasso di interesse.
10:         @param aBalance il saldo iniziale
11:         @param aRate il tasso di interesse percentuale
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
19:
```

Segue

File Investment.java

```
20:    /**
21:        Continua ad accumulare interessi finché il saldo
22:        non raggiunge un valore desiderato.
23:        @param targetBalance the desired balance
24:    */
25:    public void waitForBalance(double targetBalance)
26:    {
27:        while (balance < targetBalance)
28:        {
29:            years++;
30:            double interest = balance * rate / 100;
31:            balance = balance + interest;
32:        }
33:    }
34:
35:    /**
36:        Restituisce il saldo attuale dell'investimento.
37:        @return il saldo attuale
38:    */
```

Segue

File Investment.java

```
39:     public double getBalance()
40:     {
41:         return balance;
42:     }
43:
44:     /**
45:      Restituisce il numero di anni per i quali
46:      l'investimento ha accumulato interessi.
47:      @return il numero di anni trascorsi dall'inizio
48:             dell'investimento
49:     */
50:     public int getYears()
51:     {
52:         return years;
53:     }
54:     private double balance;
55:     private double rate;
56:     private int years;
57: }
```

File InvestmentRunner.java

```
01: /**
02:     Questo programma calcola quanto tempo occorre per il
03:     raddoppio di un investimento.
04: */
05: public class InvestmentRunner
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         Investment invest
12:             = new Investment(INITIAL_BALANCE, RATE);
13:         invest.waitForBalance(2 * INITIAL_BALANCE);
14:         int years = invest.getYears();
15:         System.out.println("The investment doubled after "
16:             + years + " years");
17:     }
18: }
```

Segue

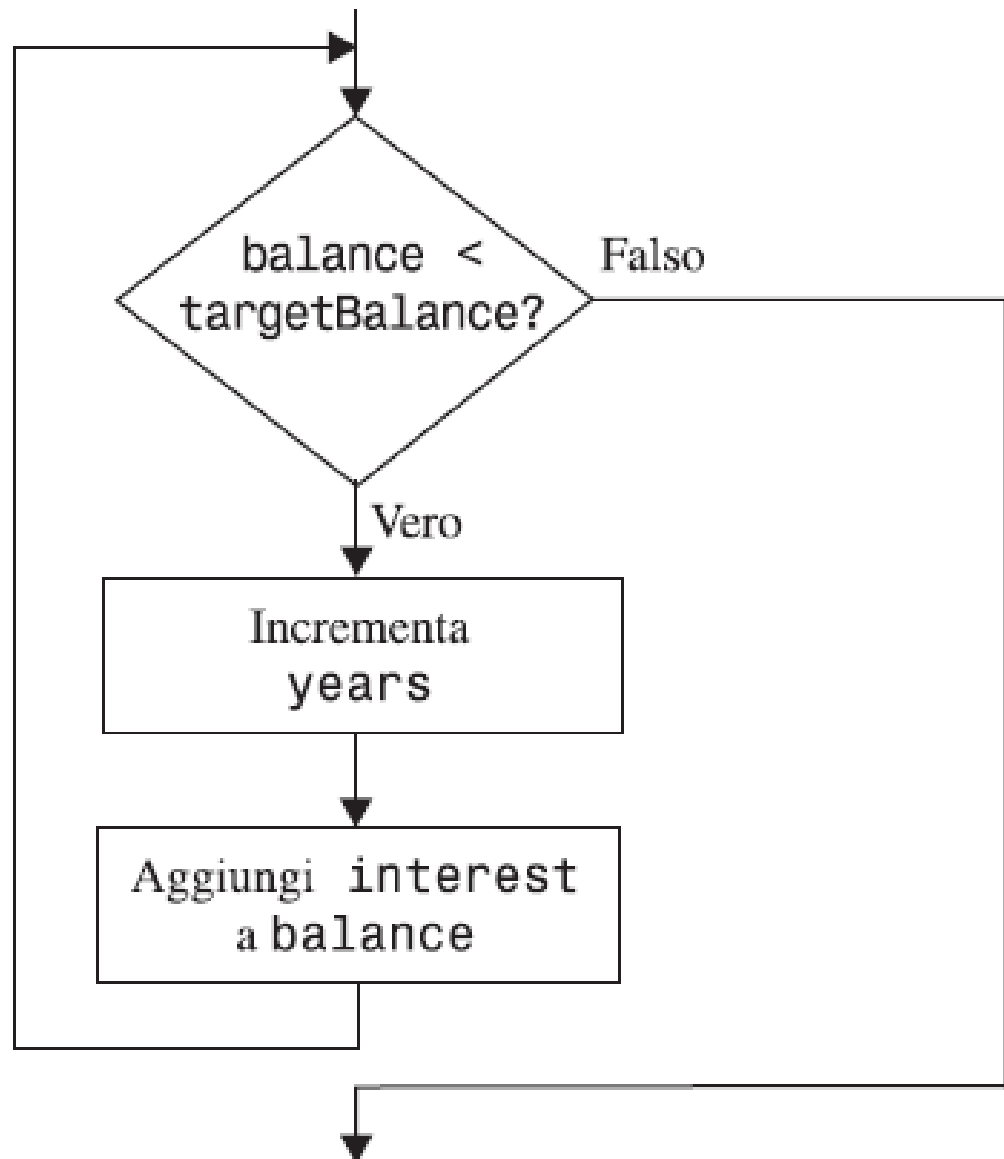
File InvestmentTester.java

Visualizza

```
The investment doubled after 15 years
```

Cicli `while`

Figura 1:
Diagramma di flusso
di un ciclo `while`



Sintassi 6.1: L'enunciato `while`

```
while (condizione)  
    enunciato
```

Esempio:

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

Obiettivo:

Eseguire un enunciato finché una condizione è vera.

Errori comuni: cicli infiniti



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

- Un ciclo che viene eseguito ininterrottamente e che si può fermare solo annullando l'esecuzione del programma o riavviando il computer.

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

Errori comuni: errori per scarto di uno

```
int years = 0;
while (balance < 2 * initialBalance)
{
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
System.out.println("The investment reached the target after "
    + years + " years.");
```

- Il conteggio della variabile `years` inizia da 0 o da 1?
- Nella condizione bisogna inserire l'operatore di confronto `<` oppure `<=`?

Errori comuni: errori per scarto di uno

- Ipotizzate uno scenario con valori semplici:
un saldo iniziale di \$ 100
tasso di interesse: 50%
Dopo un anno, il saldo sarà di \$ 150
dopo due anni, di \$ 225, e comunque oltre \$ 200

L'investimento raddoppia dopo due anni.
se il ciclo si esegue due volte, e se ciascuna volta si incrementa
`years`, significa che `years` deve partire da zero, non da uno.

- Tasso di interesse: 100%
dopo un anno: investimento raddoppiato
il ciclo viene eseguito una volta sola
Occorre utilizzare l'operatore `<`

Cicli do

- Eseguire un ciclo almeno una volta:

```
do
    enunciato
while (condizione);
```

- Esempio: verificare l'input

```
double value;
do
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
}
while (value <= 0);
```

Cicli do

- Alternativa, utilizzare una variabile di controllo booleana:

```
boolean done = false;
while (!done)
{
    System.out.print("Please enter a positive number: ");
    value = in.nextDouble();
    if (value > 0) done = true;
}
```

Cicli do

Figura 2:
Diagramma di flusso
di un ciclo `do`

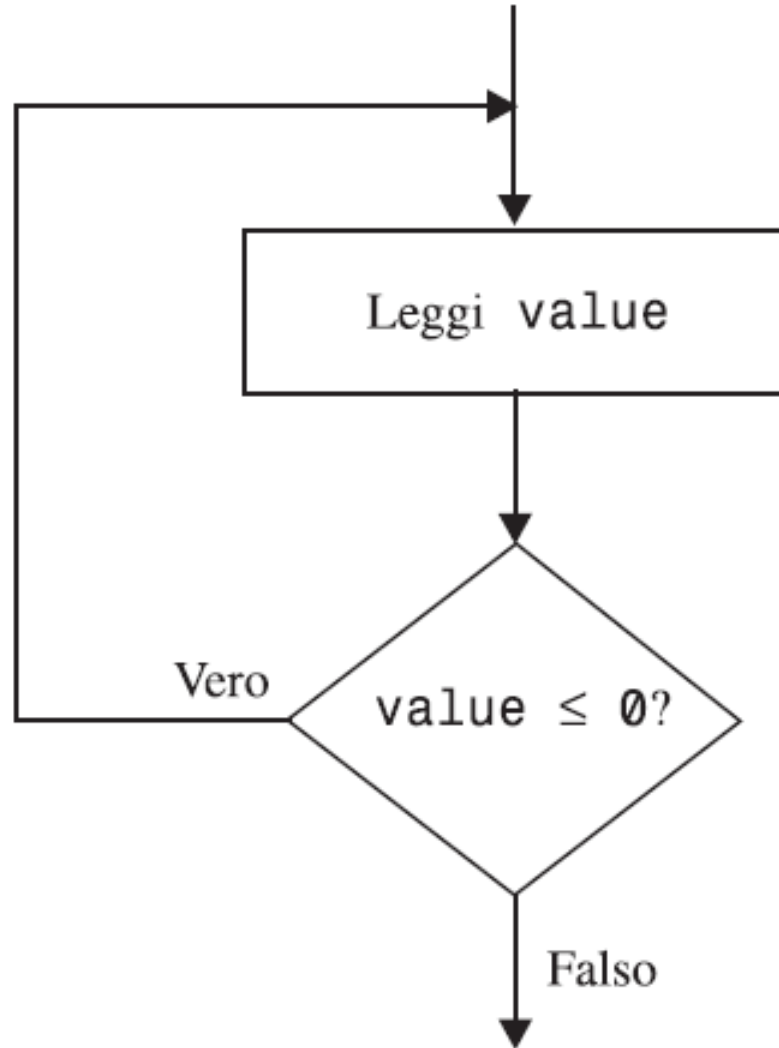
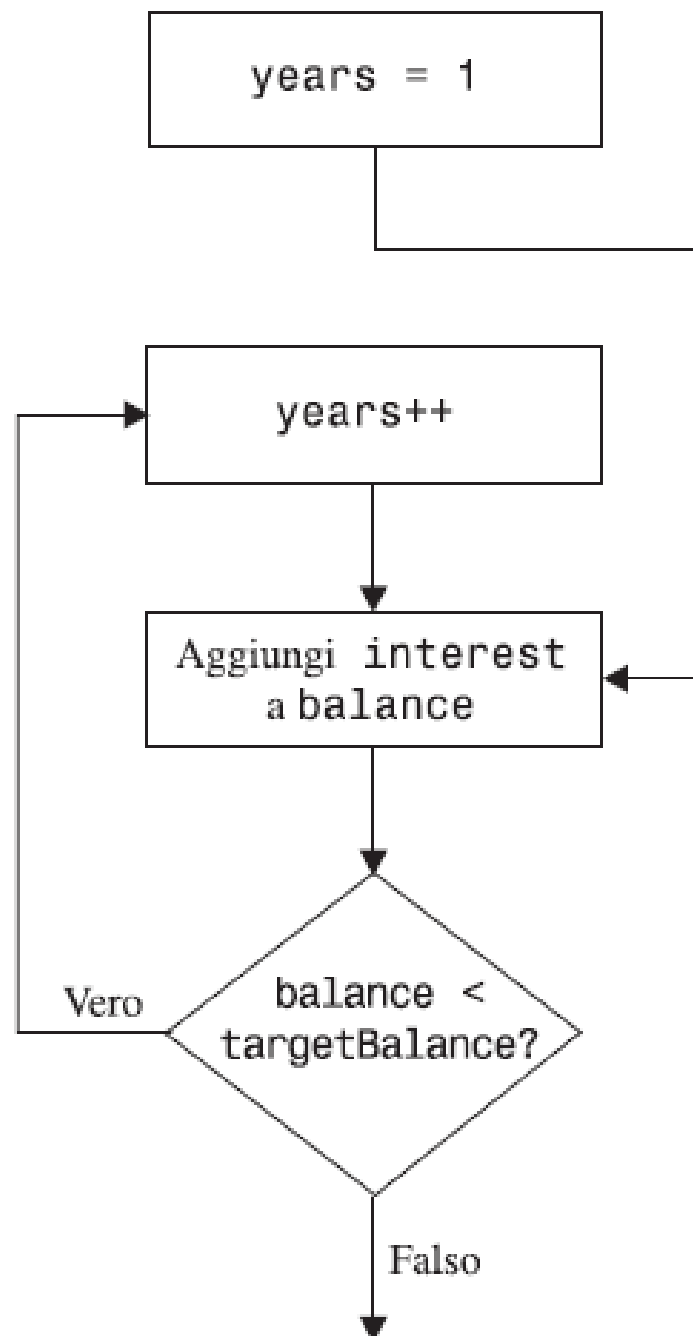


Figura 3:
Codice "spaghetti"



Ciclo for

- `for (inizializzazione; condizione; aggiornamento)
 enunciato`

Esempio:

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

Segue

Ciclo `for`

- Si usa un ciclo `for` quando una variabile viene modificata da un valore iniziale a un valore finale con un incremento o decremento costante.
- Equivale a

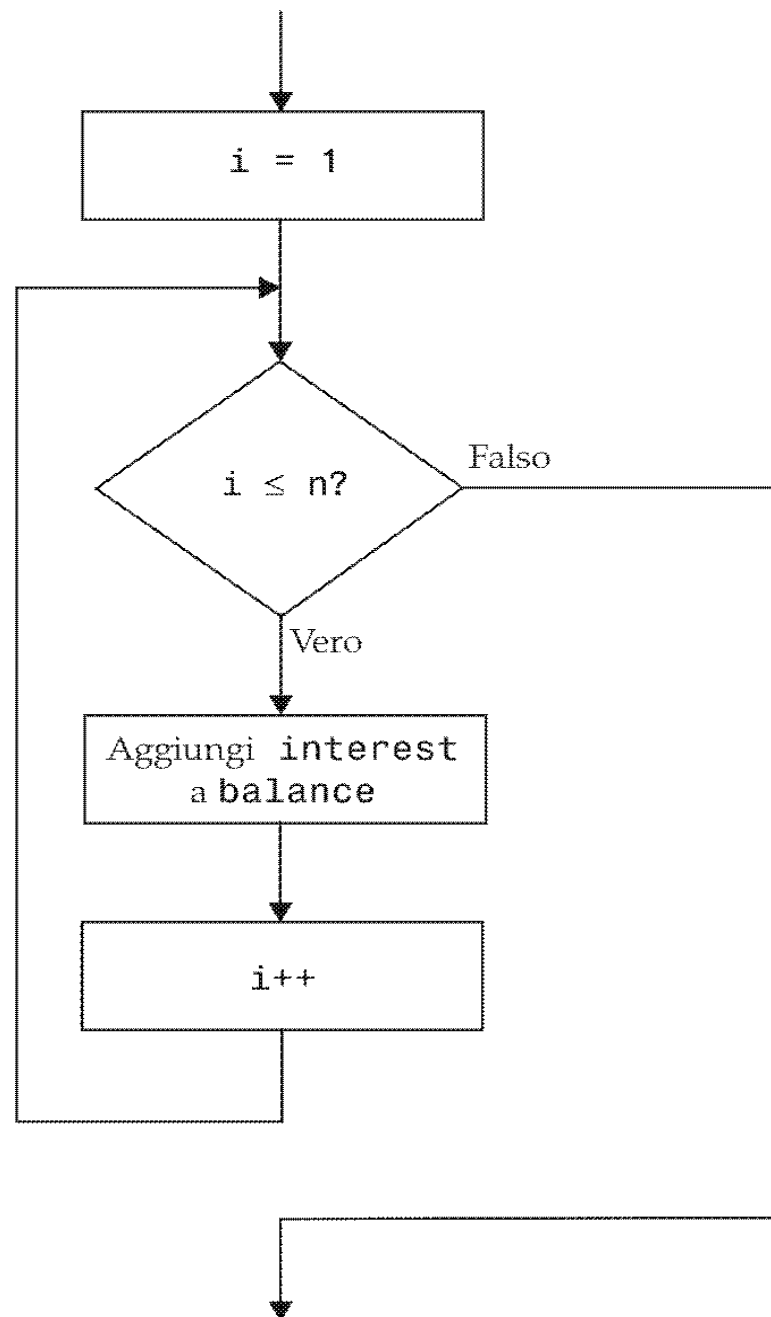
```
inizializzazione;  
while (condizione)  
{ enunciato; aggiornamento; }
```

- Altri esempi:

```
for (years = n; years > 0; years--) . . .
```

```
for (x = -10; x <= 10; x = x + 0.5) . . .
```

Figura 4:
Diagramma di flusso
per un ciclo `for`



Sintassi 6.2: L'enunciato for

```
for (inizializzazione; condizione; aggiornamento)  
    enunciato
```

Esempio:

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

Obiettivo:

Eseguire una inizializzazione, quindi ripetere l'esecuzione di un enunciato e aggiornare un'espressione, finché una condizione è vera

File Investment.java

```
01: /**
02:     Una classe per controllare la crescita di un investimento
03:     che accumula interessi a un tasso annuo fisso.
04: */
05: public class Investment
06: {
07:     /**
08:         Costruisce un oggetto Investment con un saldo iniziale
09:         e un tasso di interesse.
10:         @param aBalance il saldo iniziale
11:         @param aRate il tasso d'interesse percentuale
12:     */
13:     public Investment(double aBalance, double aRate)
14:     {
15:         balance = aBalance;
16:         rate = aRate;
17:         years = 0;
18:     }
```

Segue

File Investment.java

```
19:
20:     /**
21:         Continua ad accumulare interessi finché il saldo
22:         non raggiunge un valore desiderato.
23:         @param targetBalance il saldo desiderato
24:     */
25:     public void waitForBalance(double targetBalance)
26:     {
27:         while (balance < targetBalance)
28:         {
29:             years++;
30:             double interest = balance * rate / 100;
31:             balance = balance + interest;
32:         }
33:     }
34:
```

Segue

File Investment.java

```
35:    /**
36:        Continua ad accumulare interessi per un dato numero di anni.
37:        @param n il numero di anni
38:    */
39:    public void waitYears(int n)
40:    {
41:        for (int i = 1; i <= n; i++)
42:        {
43:            double interest = balance * rate / 100;
44:            balance = balance + interest;
45:        }
46:        years = years + n;
47:    }
48:
49:    /**
50:        Restituisce il saldo attuale dell'investimento.
51:        @return il saldo attuale
52:    */
```

Segue

File Investment.java

```
53: public double getBalance()
54: {
55:     return balance;
56: }
57:
58: /**
59:     Restituisce il numero di anni per i quali l'investimento
60:     ha accumulato interessi.
61:     @return il numero di anni dall'inizio dell'investimento
62: */
63: public int getYears()
64: {
65:     return years;
66: }
67:
```

Segue

File Investment.java

```
68:     private double balance;  
69:     private double rate;  
70:     private int years;  
71: }
```

File InvestmentRunner.java

```
01: /**
02:     Questo programma calcola di quanto aumenta un investimento
03:     in un dato numero di anni.
04: */
05: public class InvestmentTester
06: {
07:     public static void main(String[] args)
08:     {
09:         final double INITIAL_BALANCE = 10000;
10:         final double RATE = 5;
11:         final int YEARS = 20;
12:         Investment invest = new Investment(INITIAL_BALANCE, RATE);
13:         invest.waitYears(YEARS);
14:         double balance = invest.getBalance();
15:         System.out.printf("The balance after %d years is %.2f\n",
16:             YEARS, balance);
17:     }
18: }
```

Segue

File Investment.java

Visualizza

```
The balance after 20 years is 26532.98
```

Errori comuni: punto e virgola

- Punto e virgola nella posizione sbagliata

```
sum = 0;  
for (i = 1; i <= 10; i++);  
    sum = sum + i;  
System.out.println(sum);
```

- Punto e virgola mancante

```
for (years = 1;  
    (balance = balance + balance * rate / 100) <  
        targetBalance;  
    years++)  
System.out.println(years);
```

Cicli annidati

- I cicli possono essere annidati. Un esempio tipico di ciclo annidato è quello che visualizza tabelle con righe e colonne.
- Creare una forma triangolare

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

- Stampare un certo numero di righe

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

Cicli annidati

- Usare un altro ciclo per mettere insieme le parentesi quadre [] per la riga

```
for (int j = 1; j <= i; j++)  
    r = r + "[]";  
r = r + "\n";
```

- Se si mettono assieme entrambi i cicli si ottengono due cicli annidati

```
String r = " ";  
for (int i = 1; i <= width; i++)  
{  
    // costruisci una riga del triangolo  
    for (int j = 1; j <= i; j++)  
        r = r + "[]";  
    r = r + "\n";  
}  
return r;
```

File Triangle.java

```
01: /**
02:     Questa classe descrive oggetti triangolari che possono
03:     essere visualizzati in questo modo:
04:     []
05:     [][]
06:     [][][]
07: */
08: public class Triangle
09: {
10:     /**
11:         Costruisce un triangolo.
12:         @param aWidth il numero di [] nell'ultima riga
13:         del triangolo.
14:     */
15:     public Triangle(int aWidth)
16:     {
17:         width = aWidth;
18:     }
19: }
```

Segue

File Triangle.java

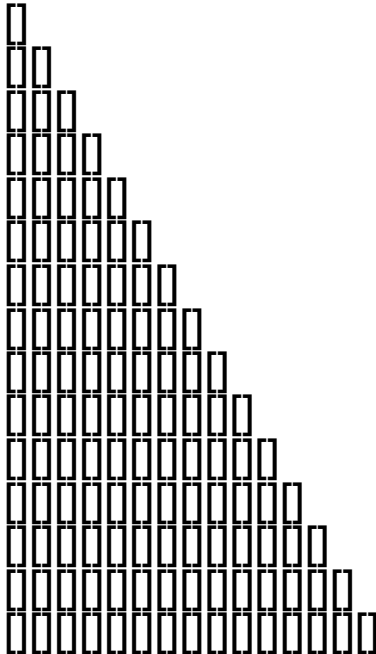
```
19:    /**
20:        Calcola una stringa che rappresenta il triangolo.
21:        @return una stringa composta di caratteri []
22:        e ritorni a capo
23:    */
24:    public String toString()
25:    {
26:        String r = "";
27:        for (int i = 1; i <= width; i++)
28:        {
29:            // costruisci una riga del triangolo
30:            for (int j = 1; j <= i; j++)
31:                r = r + "[";
32:            r = r + "\n";
33:        }
34:        return r;
35:    }
36:    private int width;
37: }
```

File TriangleRunner.java

```
01: /**
02:     Questo programma visualizza due triangoli.
03: */
04: public class TriangleRunner
05: {
06:     public static void main(String[] args)
07:     {
08:         Triangle small = new Triangle(3);
09:         System.out.println(small.toString());
10:
11:         Triangle large = new Triangle(15);
12:         System.out.println(large.toString());
13:     }
14: }
```

File TriangleRunner.java

Visualizza



Utilizzo di valori sentinella

- Metodo molto utilizzato per segnalare la fine di un insieme di dati
- Non è una buona idea scegliere 0 oppure -1 come sentinelle: questi valori possono essere validi come dati del problema. E' meglio utilizzare la lettera Q.

```
System.out.print("Enter value, Q to quit: ");
String input = in.next();
if (input.equalsIgnoreCase("Q"))
    Abbiamo finito
else
{
    double x = Double.parseDouble(input);
    . . .
}
```

Ciclo e mezzo

- A volte la condizione di terminazione di un ciclo può essere valutata soltanto nel mezzo di un ciclo, che può essere controllato mediante una variabile booleana.

```
boolean done = false;
while (!done)
{
    Visualizza una richiesta di dati
    String input = leggi un dato
    if (i dati sono terminati)
        done = true;
    else
    {
        // elabora il dato letto
    }
}
```

File DataAnalyzer.java

```
01: import java.util.Scanner;
02:
03: /**
04:  Questo programma calcola il valore medio e il valore massimo
05:  di un insieme di dati forniti in ingresso.
06: */
07: public class DataAnalyzer
08: {
09:     public static void main(String[] args)
10:     {
11:         Scanner in = new Scanner(System.in);
12:         DataSet data = new DataSet();
13:
14:         boolean done = false;
15:         while (!done)
16:         {
```

Segue

File DataAnalyzer.java

```
17:         System.out.print("Enter value, Q to quit: ");
18:         String input = in.next();
19:         if (input.equalsIgnoreCase("Q"))
20:             done = true;
21:         else
22:         {
23:             double x = Double.parseDouble(input);
24:             data.add(x);
25:         }
26:     }
27:
28:     System.out.println("Average = " + data.getAverage());
29:     System.out.println("Maximum = " + data.getMaximum());
30: }
31: }
```

File DataSet.java

```
01: /**
02:     Calcola informazioni relative a un insieme di dati.
03: */
04: public class DataSet
05: {
06:     /**
07:         Costruisce un insieme di dati vuoto.
08:     */
09:     public DataSet()
10:     {
11:         sum = 0;
12:         count = 0;
13:         maximum = 0;
14:     }
15:
16:     /**
17:         Aggiunge un valore all'insieme dei dati
18:         @param x un valore
19:     */
```

Segue

File DataSet.java

```
20: public void add(double x)
21: {
22:     sum = sum + x;
23:     if (count == 0 || maximum < x) maximum = x;
24:     count++;
25: }
26:
27: /**
28:     Restituisce la media dei valori inseriti.
29:     @return la media, o 0 se non ci sono dati
30: */
31: public double getAverage()
32: {
33:     if (count == 0) return 0;
34:     else return sum / count;
35: }
36:
```

Segue

File DataSet.java

```
37:    /**
38:        Restituisce il valore massimo tra i valori inseriti.
39:        @return il massimo, o 0 se non ci sono dati
40:    */
41:    public double getMaximum()
42:    {
43:        return maximum;
44:    }
45:
46:    private double sum;
47:    private double maximum;
48:    private int count;
49: }
```

File DataSet.java

Visualizza

```
Enter value, Q to quit: 10
Enter value, Q to quit: 0
Enter value, Q to quit: -1
Enter value, Q to quit: Q
Average = 3.0
Maximum = 10.0
```

Numeri casuali e simulazioni

- In una simulazione si generano ripetutamente numeri casuali, usandoli per simulare una determinata attività.
- Generatore di numeri casuali

```
Random generator = new Random();  
int n = generator.nextInt(a); // 0 <= n < a  
double x = generator.nextDouble(); // 0 <= x < 1
```

- Lancio del dado (numero casuale tra 1 e 6)

```
int d = 1 + generator.nextInt(6);
```

File Die.java

```
01: import java.util.Random;
02:
03: /**
04:     Questa classe costituisce un modello di un dado, che,
05:     quando viene lanciato, atterra su una faccia scelta a caso.
06: */
07: public class Die
08: {
09:     /**
10:         Costruisce un dado con un dato numero di facce.
11:         @param s il numero di facce, 6 in un dado normale
12:     */
13:     public Die(int s)
14:     {
15:         sides = s;
16:         generator = new Random();
17:     }
18:
```

Segue

File Die.java

```
19:    /**
20:        Simula un lancio del dado
21:        @return la faccia del dado
22:    */
23:    public int cast()
24:    {
25:        return 1 + generator.nextInt(sides);
26:    }
27:
28:    private Random generator;
29:    private int sides;
30: }
```

File DieSimulator.java

```
01: /**
02:     Questo programma simula il lancio di un dado per 10 volte
03: */
04: public class DieSimulator
05: {
06:     public static void main(String[] args)
07:     {
08:         Die d = new Die(6);
09:         final int TRIES = 10;
10:         for (int i = 1; i <= TRIES; i++)
11:         {
12:             int n = d.cast();
13:             System.out.print(n + " ");
14:         }
15:         System.out.println();
16:     }
17: }
```

File DieSimulator.java

Visualizza

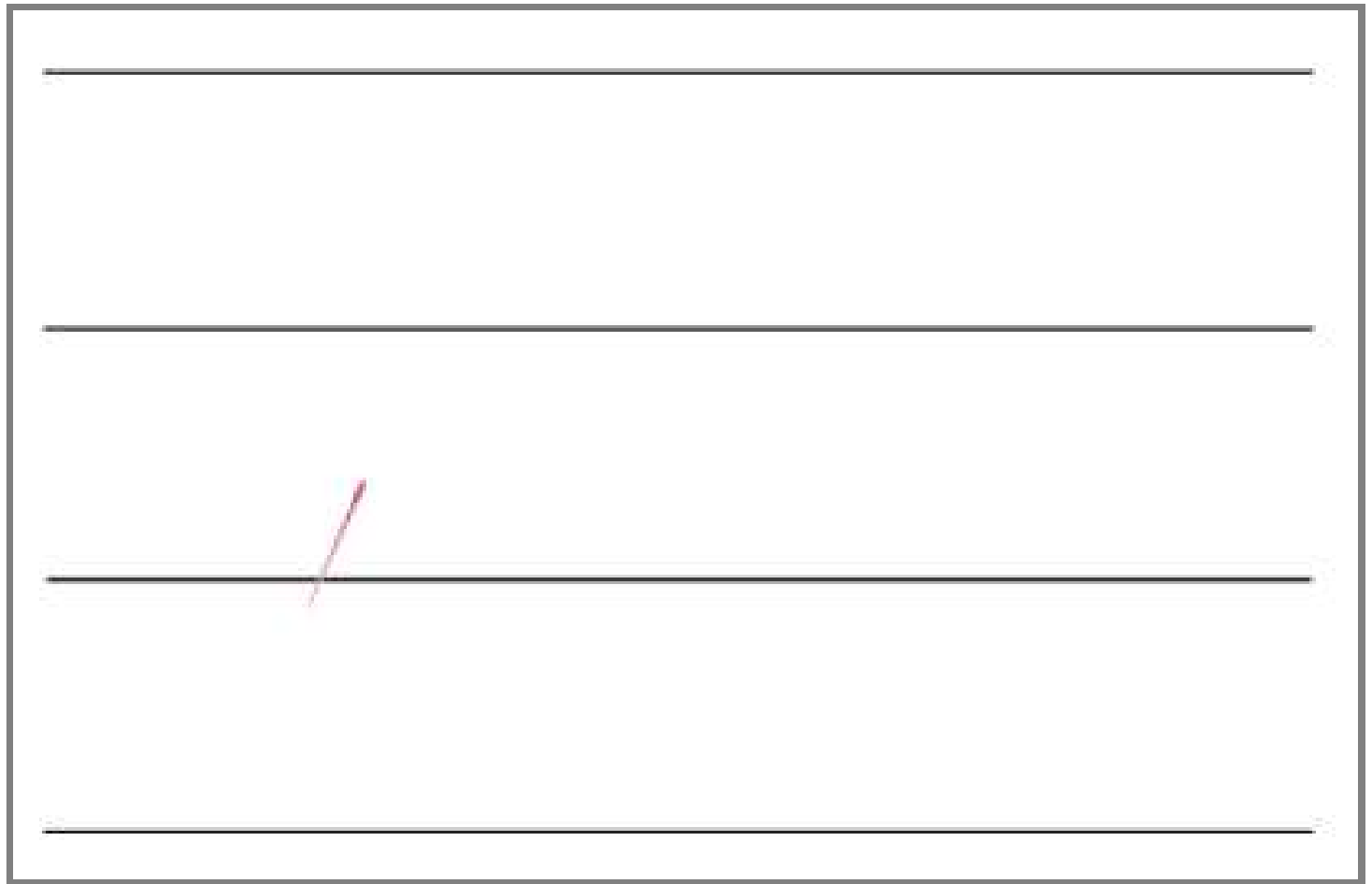
6 5 6 3 2 6 3 4 4 1

Visualizza (alla seconda esecuzione)

3 2 2 1 6 5 3 4 1 2

Esperimento dell'ago di Buffon

Figura 5



Posizione dell'ago

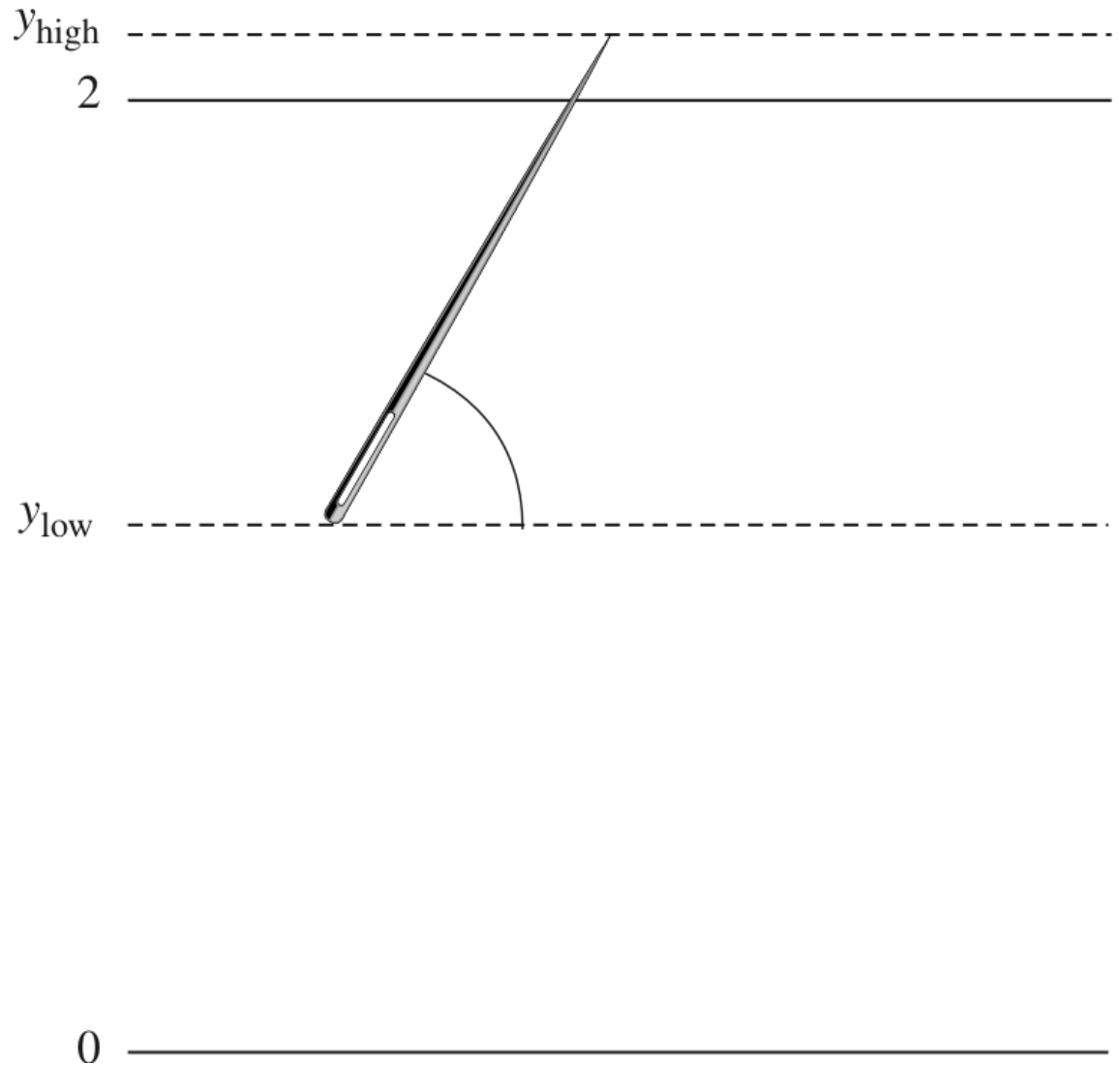


Figura 6:
Quando l'ago cade su
una riga?

Posizione dell'ago

- Lunghezza dell'ago = 1; distanza tra le linee = 2
- y_{low} corrisponde a un numero casuale tra 0 e 2
- L'angolo α può avere qualsiasi valore compreso tra 0 e 180 gradi
- $y_{high} = y_{low} + \text{sen}(\alpha)$
- Si colpisce il bersaglio quando il valore di y_{high} è almeno uguale a 2

File Needle.java

```
01: import java.util.Random;
02:
03: /**
04:     Questa classe simula un ago nell'esperimento di Buffon.
05: */
06: public class Needle
07: {
08:     /**
09:         Costruisce un ago.
10:     */
11:     public Needle()
12:     {
13:         hits = 0;
14:         tries = 0;
15:         generator = new Random();
16:     }
17:
```

Segue

File Needle.java

```
18:  /**
19:      Lascia cadere l'ago sulla griglia di linee e ricorda
20:      se l'ago colpisce una riga.
21:  */
22:  public void drop()
23:  {
24:      double ylow = 2 * generator.nextDouble();
25:      double angle = 180 * generator.nextDouble();
26:
27:      // Calcola il punto più alto dell'ago
28:
29:      double yhigh = ylow + Math.sin(Math.toRadians(angle));
30:      if (yhigh >= 2) hits++;
31:      tries++;
32:  }
33:
34:  /**
35:      Restituisce il numero di bersagli colpiti.
36:      @return il numero di bersagli colpiti
37:  */
```

Segue

File Needle.java

```
38:     public int getHits()
39:     {
40:         return hits;
41:     }
42:
43:     /**
44:         Restituisce il numero totale di lanci dell'ago.
45:         @return il numero di lanci
46:     */
47:     public int getTries()
48:     {
49:         return tries;
50:     }
51:
52:     private Random generator;
53:     private int hits;
54:     private int tries;
55: }
```

File NeedleSimulator.java

```
01: /**
02:     Questo programma simula l'esperimento del lancio di aghi di Buffon
03:     e visualizza la risultante approssimazione del valore di pi greco.
04: */
05: public class NeedleSimulator
06: {
07:     public static void main(String[] args)
08:     {
09:         Needle n = new Needle();
10:         final int TRIES1 = 10000;
11:         final int TRIES2 = 1000000;
12:
```

Segue

File NeedleSimulator.java

```
13:     for (int i = 1; i <= TRIES1; i++)
14:         n.drop();
15:     System.out.printf("Tries = %d, Tries / Hits = %8.5f\n",
16:         TRIES1, (double) n.getTries() / n.getHits());
17:
18:     for (int i = TRIES1 + 1; i <= TRIES2; i++)
19:         n.drop();
20:     System.out.printf("Tries = %d, Tries / Hits = %8.5f\n",
21:         TRIES2, (double) n.getTries() / n.getHits());
22: }
23: }
```

Visualizza

```
Tries = 10000, Tries / Hits = 3.08928
Tries = 1000000, Tries / Hits = 3.14204
```

Usare un *debugger*

- Un *debugger* è un programma che potete usare per eseguire un altro programma e analizzare il suo comportamento durante l'esecuzione.
- Potete interrompere e far proseguire il vostro programma, vedere il contenuto delle variabili quando arrestate temporaneamente l'esecuzione, scegliere quanti passi di programma eseguire prima dell'interruzione successiva.
- Più grandi sono i vostri programmi, più difficile sarà correggerli inserendo semplicemente enunciati di stampa.
- I *debugger* possono far parte di ambienti di sviluppo integrato (es. BlueJ, Eclipse) o essere programmi a se stanti (es. Jswat).
- Tre concetti chiave:
 - *punti di arresto (breakpoints)*
 - *esecuzione del programma una riga alla volta (single-stepping)*
 - *ispezione del valore di singole variabili (inspecting variables)*

Il debugger fermo a un punto di arresto

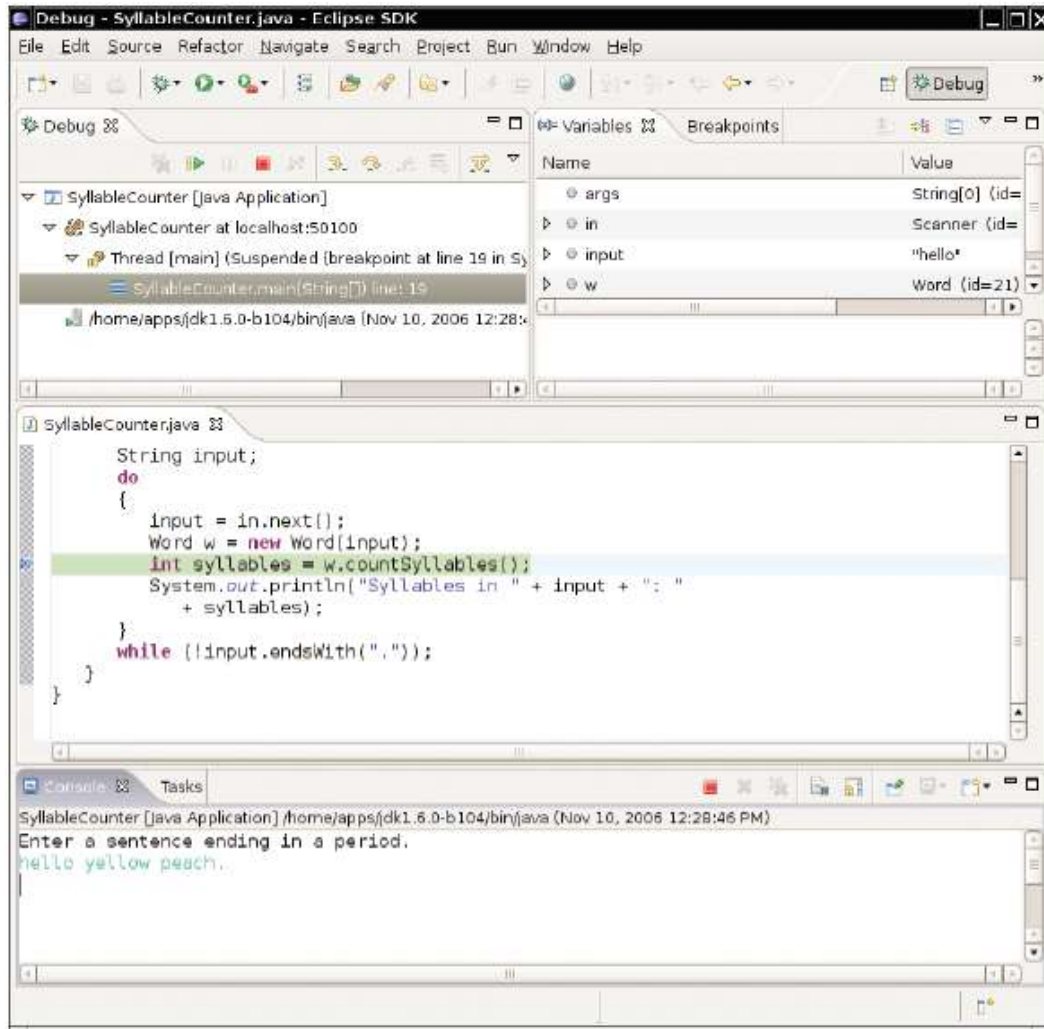


Figura 7: Il debugger fermo a un punto di arresto

Ispezionare variabili

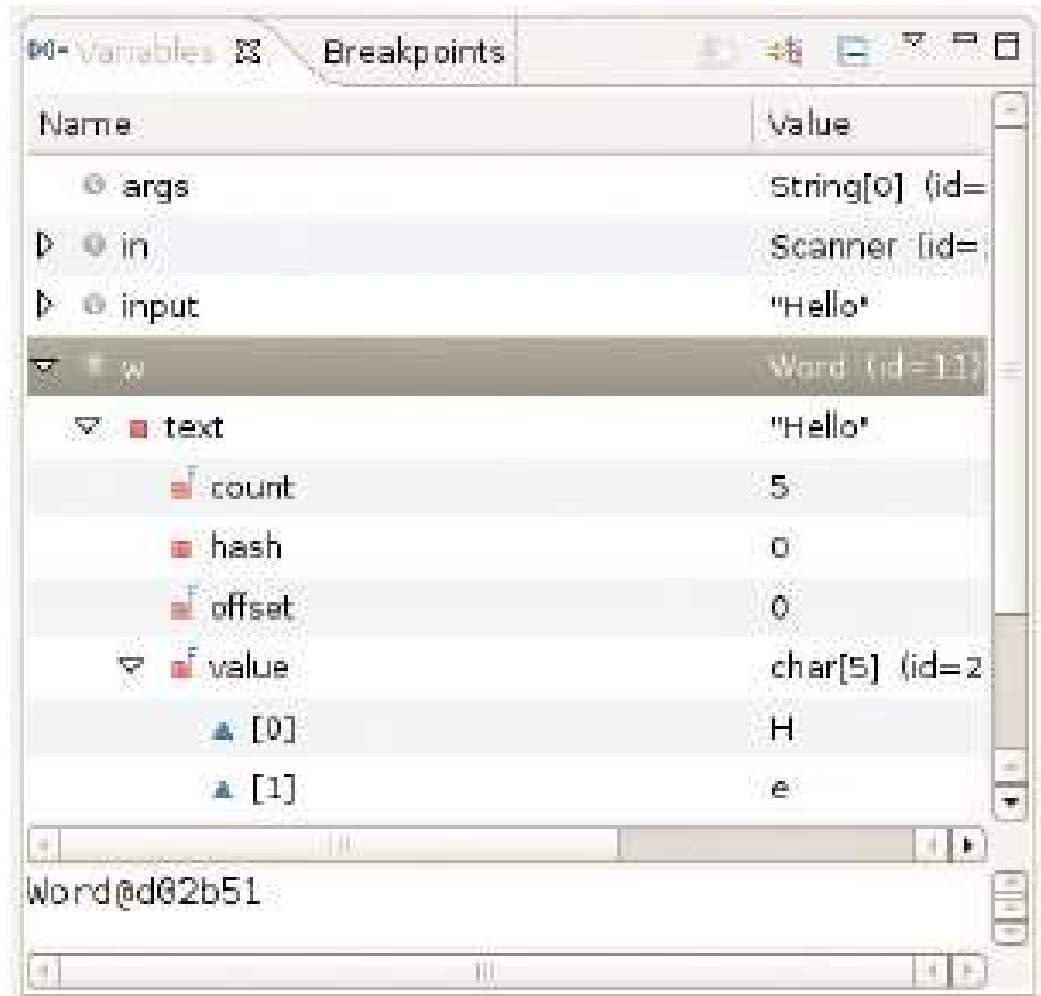


Figura 8: Ispezionare variabili

Debugging

- L'esecuzione viene sospesa al raggiungimento del breakpoint
- Quando fate partire il debugger, il programma viene eseguito a piena velocità fino al raggiungimento di un punto di arresto
- Quando il programma si è fermato è possibile:
 - Ispezionare le variabili
 - Eseguire il programma una riga alla volta
 - Oppure continuare l'esecuzione del programma a velocità piena fino al raggiungimento del breakpoint successivo
- Quando il programma termina, termina anche l'esecuzione del debugger
- I punti di arresto rimangono attivi finché non vengono esplicitamente rimossi
- Due tipi di comando single-step:
 - step into (fai un passo all'interno)
 - step over (fai un passo scavalcando)

Attività di debugging

- Riprodurre l'errore
- Semplificare l'errore
- Divide and conquer / Top-down
- Sapere cosa dovrebbe fare il programma / metodo
- Controllare tutti i dettagli
- Capire bene l'errore prima di correggerlo

Esempio di *single step*

Linea attuale:

```
String input = in.next();  
Word w = new Word(input);  
int syllables = w.countSyllables();  
System.out.println("Syllables in " + input + ": " +  
    syllables);
```

Quando eseguite un comando di tipo *step over*, procedete fino alla riga successiva:

```
String input = in.next();  
Word w = new Word(input);  
int syllables = w.countSyllables();  
System.out.println("Syllables in " + input + ": " +  
    syllables);
```

Esempio di *single step*

Tuttavia se eseguite un comando di tipo *step into* vi trovate nella prima linea del metodo `countSyllables`:

```
public int countSyllables()  
{  
    int count = 0;  
    int end = text.length() - 1;  
    . . .  
}
```

Un esempio di sessione di debugging

- La classe word ha il compito di contare le sillabe in una parola
- Ogni gruppo di vocali adiacenti (a, e, i, o, u, y) fa parte di una sillaba
- Tuttavia, una “e” alla fine di una parola non forma una sillaba
- Ogni parola ha almeno una sillaba anche se le regole precedenti forniscono un conteggio nullo
- Eventuali caratteri all’inizio o alla fine della stringa che non siano lettere vengono eliminati

File Word.java

```
01: /**
02:     Classe che descrive una parola in un documento.
03: */
04: public class Word
05: {
06:     /**
07:         Costruisce una parola eliminando caratteri iniziali e
08:         finali che non siano lettere, come i segni di punteggiatura.
09:         @param s la stringa da analizzare
10:     */
11:     public Word(String s)
12:     {
13:         int i = 0;
14:         while (i < s.length() && !Character.isLetter(s.charAt(i)))
15:             i++;
16:         int j = s.length() - 1;
17:         while (j > i && !Character.isLetter(s.charAt(j)))
18:             j--;
19:         text = s.substring(i, j);
20:     }
21:
```

Segue

File Word.java

```
22:    /**
23:       Restituisce il testo della parola dopo aver rimosso
24:       caratteri iniziali e finali che non siano lettere.
25:       @return il testo della parola
26:    */
27:    public String getText()
28:    {
29:        return text;
30:    }
31:
32:    /**
33:       Conta le sillabe presenti nella parola.
34:       @return il numero di sillabe
35:    */
36:    public int countSyllables()
37:    {
38:        int count = 0;
39:        int end = text.length() - 1;
40:        if (end < 0) return 0; // La stringa vuota non ha sillabe
41:
```

Segue

File Word.java

```
42:      // Una e alla fine della parola non conta come vocale
43:      char ch = Character.toLowerCase(text.charAt(end));
44:      if (ch == 'e') end--;
46:      boolean insideVowelGroup = false;
47:      for (int i = 0; i <= end; i++)
48:      {
49:          ch = Character.toLowerCase(text.charAt(i));
50:          String vowels = "aeiouy";
51:          if (vowels.indexOf(ch) >= 0)
52:          {
53:              // ch è una vocale
54:              if (!insideVowelGroup)
55:              {
56:                  // inizia un nuovo gruppo di vocali
57:                  count++;
58:                  insideVowelGroup = true;
59:              }
60:          }
61:      }
62:
```

Segue

File Word.java

```
63:         // Ogni parola ha almeno una sillaba
64:         if (count == 0)
65:             count = 1;
66:
67:         return count;
68:     }
69:
70:     private String text;
71: }
```

File SyllableCounter.java

```
01: import java.util.Scanner;
02:
03: /**
04:     Questo programma conta le sillabe di tutte le parole di una frase.
05: */
06: public class SyllableCounter
07: {
08:     public static void main(String[] args)
09:     {
10:         Scanner in = new Scanner(System.in);
11:
12:         System.out.println("Enter a sentence ending in a period.");
13:
14:         String input;
15:         do
16:         {
17:             input = in.next();
18:             Word w = new Word(input);
19:             int syllables = w.countSyllables();
20:             System.out.println("Syllables in " + input + ": "
21:                 + syllables);
22:         }
```

Segue

File SyllableCounter.java

```
23:         while (!input.endsWith("."));  
24:     }  
25: }
```

Il programma di debug

- Fornendo questi dati in ingresso: "hello yellow peach. " viene visualizzato quanto segue (che non è molto promettente):
Syllables in hello: 1
Syllables in yellow: 1
Syllables in peach.: 1
- Impostare un punto di arresto nella prima linea del metodo del `countSyllables` della classe `word`
- Fate partire il programma che chiederà i dati in ingresso, ricevuti i quali si arresterà al breakpoint impostato
- Il metodo controlla l'ultimo carattere della parola per vedere se è una "e"

Il programma di debug

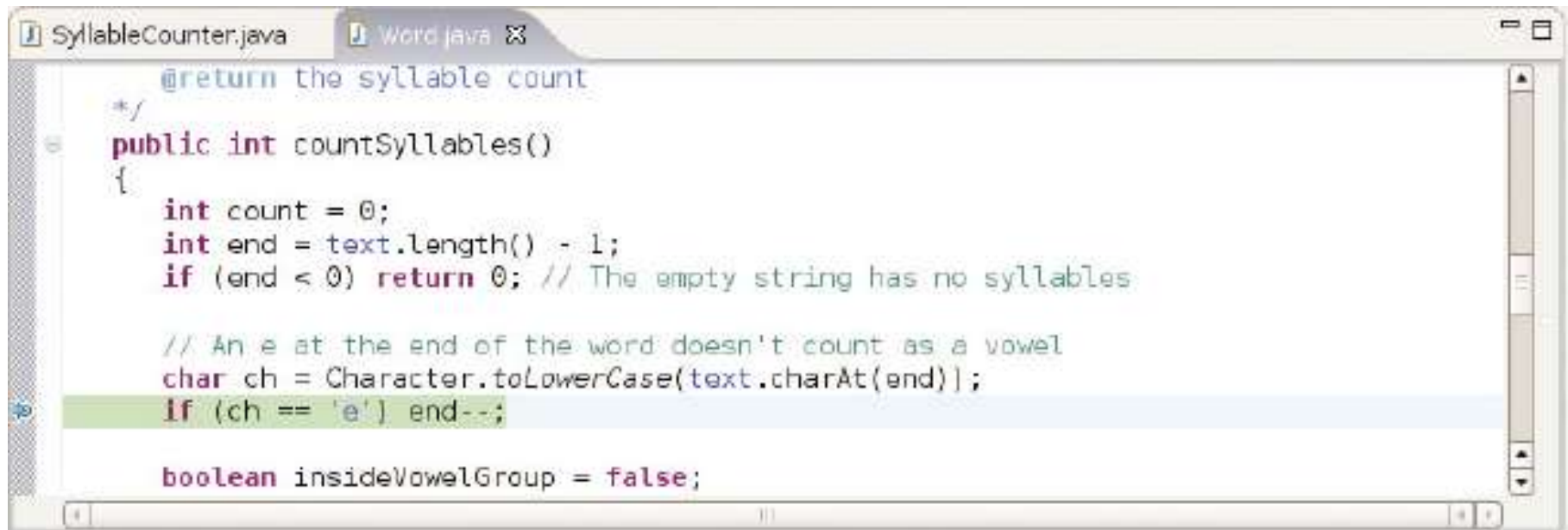


Figura 9: Collaudare il metodo `countSyllables` con il debugger

- Vediamo se questo funziona correttamente: eseguite il programma fino alla linea in cui viene fatto il controllo. Ora ispezionate la variabile `ch`.
- Potete verificare che `ch` contiene il valore `"l"`

Ulteriori problemi

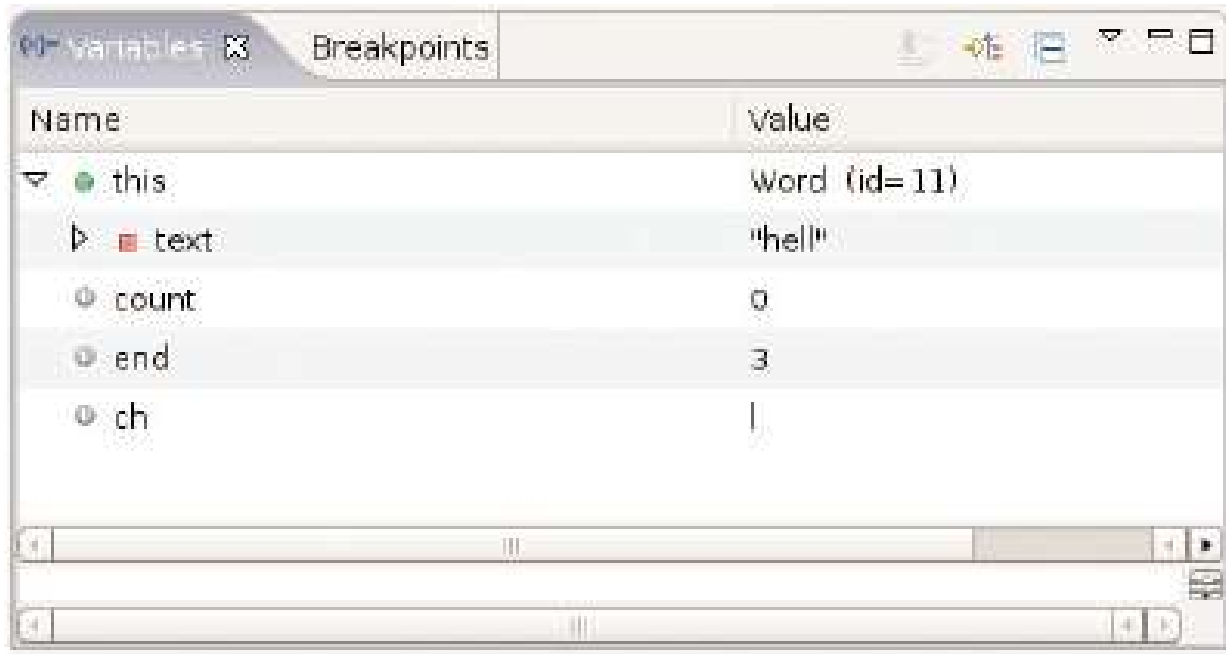


Figura 10: I valori attuali delle variabili locali e di esemplare

- `end` vale 3 e non 4
- `text` contiene la stringa `"hell"` e non `"hello"`
- Non c'è da meravigliarsi che `countSyllable` fornisca la risposta 1

Ulteriori problemi

- La soluzione è da ricercare altrove
- Un debugger non può tornare indietro nel tempo
- Interrompere il programma, impostare un breakpoint nel costruttore di word e far ripartire il debugger

Collaudare il costruttore di word con il debugger

- Fornire di nuovo i dati in ingresso
- Impostare un breakpoint dopo la fine del secondo ciclo
- Ispezionare i valori `i` e `j`
- `i` vale 0 e `j` vale 4. Ciò ha senso perché non ci sono segni di punteggiatura da ignorare
- Perché `text` viene impostato a `"hell"`?
- Tipico errore “per scarto di uno”: il metodo `substring` considera le posizioni fino al secondo parametro senza includerlo

```
text = substring(i, j);
```

dovrebbe essere:

```
text = s.substring(i, j + 1);
```

Collaudare il costruttore di word con il debugger

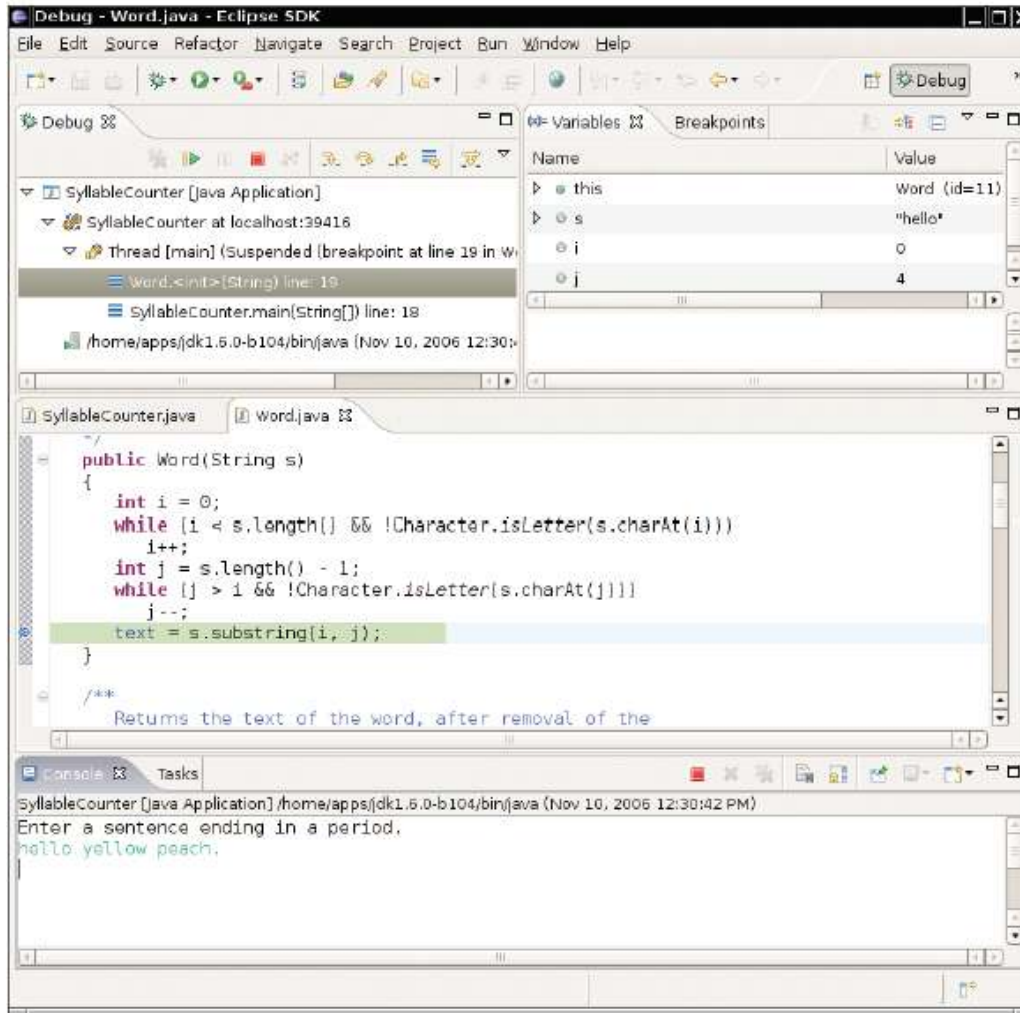


Figura 11: Collaudare il costruttore di Word con il debugger

Un altro errore

- Correggete questo errore
- Compile nuovamente il programma

- Provate di nuovo i tre casi di prova

```
Syllables in hello: 1  
Syllables in yellow: 1  
Syllables in peach.: 1
```

- C'è ancora un problema
- Eliminate tutti i punti di arresto e impostate un nuovo breakpoint nel metodo `countSyllables`
- Fornite in ingresso la stringa `"hello"`

Collaudare di nuovo con il debugger

countSyllables

- Iniziate a eseguire il programma passo dopo passo, in modalità single step, all'interno del metodo

```
boolean insideVowelGroup = false;
for (int i = 0; i <= end; i++)
{
    ch = Character.toLowerCase(text.charAt(i));
    if ("aeiouy".indexOf(ch) >= 0)
    {
        // ch è una vocale
        if (!insideVowelGroup)
        {
            // Inizia un nuovo gruppo di vocali
            count++;
            insideVowelGroup = true;
        }
    }
}
```

Collaudare di nuovo con il debugger

`countSyllables`

- Nella prima iterazione del ciclo, il debugger non esegue l'enunciato `if`. Ciò è corretto perché la prima lettera `"h"` non è una vocale.
- Nella seconda iterazione il debugger entra nell'enunciato `if` come dovrebbe perché la seconda lettera `"e"` è una vocale. La variabile `count` viene incrementata.
- Nella terza iterazione l'enunciato `if` viene ignorato nuovamente perché la lettera `"l"` non è una vocale.
- Nella quinta iterazione succede qualcosa di strano: la lettera `"o"` è una vocale e l'enunciato `if` viene eseguito, ma il secondo enunciato `if` viene ignorato e la variabile `count` non viene nuovamente incrementata.

Correggere l'errore

- La lettura di una consonante dovrebbe riportare `insideVowelGroup` al valore `false`

- Per correggerlo:

```
if ("aeiouy".indexOf(ch) >= 0)
{
    . . .
}
else insideVowelGroup = false;
```

- Ora compilate nuovamente ed eseguite di nuovo il collaudo, ottenendo:

```
Syllables in hello: 2
Syllables in yellow: 2
Syllables in peach.: 1
```

- Il debugger è ora privo di errori? Questa non è una domanda alla quale il debugger possa rispondere.

Il primo bug

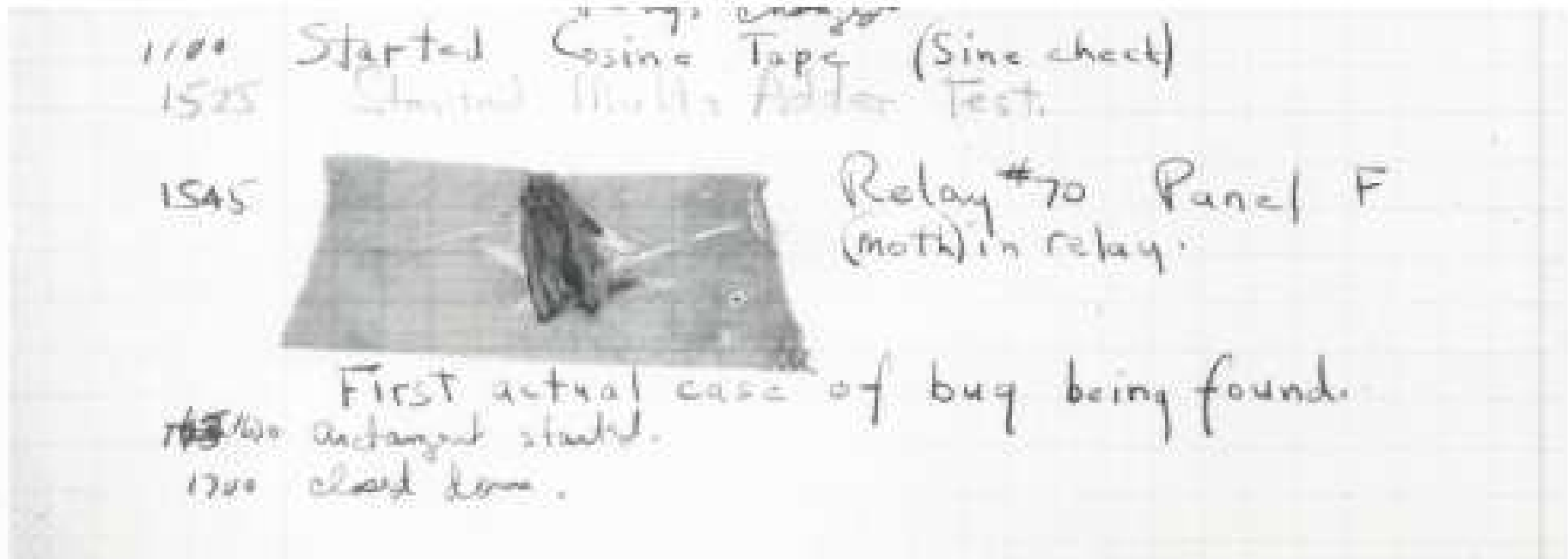


Figura 12: Il primo bug