Capitolo 5

Decisioni

Cay S. Horstmann
Concetti di informatica e fondamenti di Java
quarta edizione

Obiettivi del capitolo

- Saper realizzare decisioni usando enunciati `if`
- Capire come raggruppare enunciati in blocchi
- Imparare a confrontare numeri interi, numeri in virgola mobile, stringhe e oggetti
- Identificare il corretto ordine delle decisioni nelle ramificazioni multiple
- Programmare condizioni usando operatori e variabili booleani
- Comprendere l'importanza della copertura del collaudo

L'enunciato `if`

- L'enunciato `if` consente a un programma di compiere azioni diverse in dipendenza dal verificarsi di una condizione.

```
if (amount <= balance)
   balance = balance - amount;
```

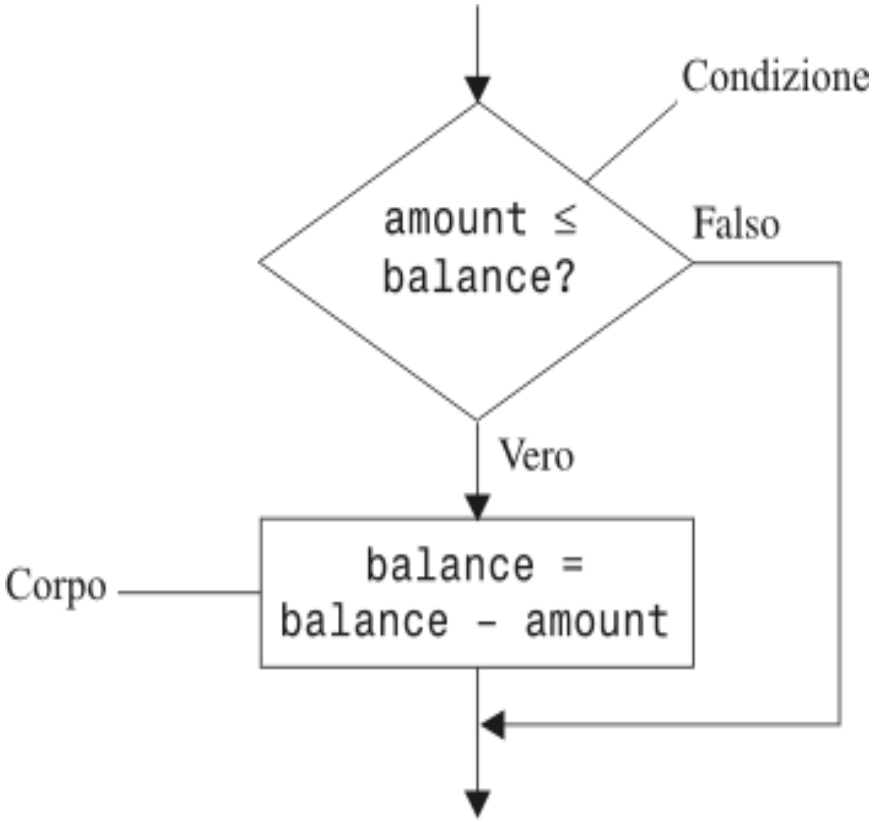

Figura 1:
Diagramma di flusso di un enunciato `if`

L'enunciato `if/else`

```
if (amount <= balance)
   balance = balance - amount;
else
   balance = balance - OVERDRAFT_PENALTY;
```

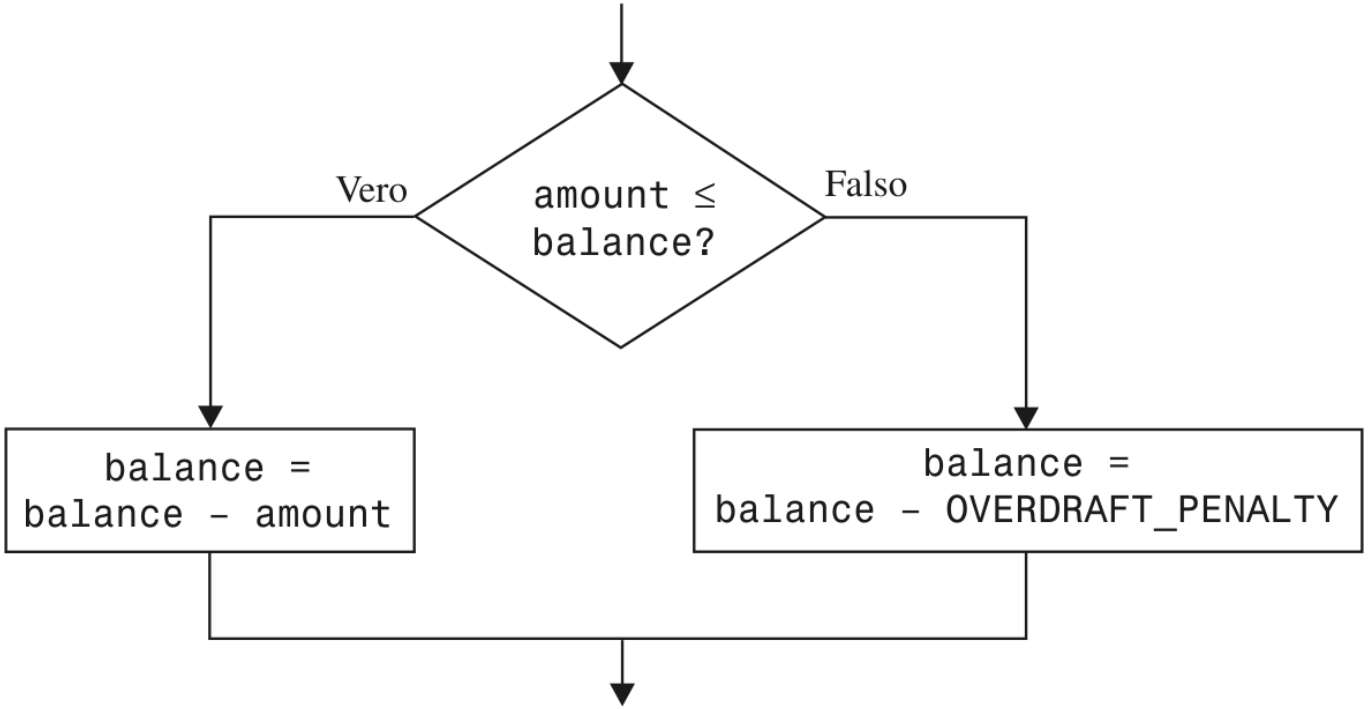

Figura 2: Diagramma di flusso di un enunciato if/else

Tipi di enunciati

- Enunciato semplice

```
balance = balance - amount;
```

- Enunciato composto

  Anche `while`, `for`, ecc. (cicli – Capitolo 7)

```
if (x >= 0) y = x;
```

- Blocco di enunciati

```
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Sintassi 5.1: L'enunciato `if`

```
if(condizione)
   enunciato

if (condizione)
   enunciato1
else
   enunciato2
```

Esempio:

```
if (amount <= balance)
   balance = balance - amount;

if (amount <= balance)
   balance = balance - amount;
else
   balance = balance - OVERDRAFT_PENALTY;
```

Obiettivo:

Eseguire un enunciato quando una condizione è vera o falsa

Sintassi 5.2: Blocco di enunciati

```
{
   enunciato1
   enunciato2
   . . .
}
```

Esempio:

```
{
   double newBalance = balance - amount;
   balance = newBalance;
}
```

Obiettivo:

Raggruppare diversi enunciati per formare un enunciato unico

Confrontare valori: Operatori relazionali

- Gli operatori relazionali confrontano valori.

Operatore Java	Notazione matematica	Descrizione
>	>	Maggiore
>=	≥	Maggiore di o uguale a
<	<	Minore
<=	≤	Minore di o uguale a
==	=	Uguale
!=	≠	Diverso

- L'operatore == verifica l'uguaglianza.

```
a = 5; // assegna 5 ad a
if (a == 5) . . . // verifica se a è uguale a 5
```

Confrontare numeri in virgola mobile

- Il codice:

```
double r = Math.sqrt(2);
double d = r * r -2;
if (d == 0)
   System.out.println("sqrt(2)squared minus 2 is 0");
else
   System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
```

- stampa:

```
sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16
```

Confrontare numeri in virgola mobile

- Confrontando numeri in virgola mobile, non fate verifiche di uguaglianza (= =), ma controllate se i valori sono *sufficientemente prossimi*.
- Per confrontare numeri double, di solito si usa un valore di ε uguale a 10^{-14} .
- Analogamente, potete verificare se due numeri sono prossimi l'uno all'altro controllando se la loro differenza è prossima a 0.
  - $|x - y| \leq \varepsilon$

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
   // x è quasi uguale a y
```

Confrontare stringhe

- Non utilizzare == per le stringhe

```
if (string1 == string2) // inutile
```

- Utilizzare il metodo `equals`:

```
if (input.equals("Y"))
```

- == verifica se le due variabili stringa si riferiscono al medesimo oggetto stringa.
  `equals` verifica che il contenuto sia uguale.

- Per ignorare le differenze tra maiuscolo e minuscolo usate il metodo `equalsIgnoreCase`

```
if (input.equalsIgnoreCase("Y"))
```

Confrontare stringhe

- `s.compareTo(t) < 0` significa che `s` precede la `t` in ordine alfabetico.
- "car" precede "cargo"
- Tutte le lettere maiuscole precedono quelle minuscole

  “B” precede “a”

Confronto lessicografico

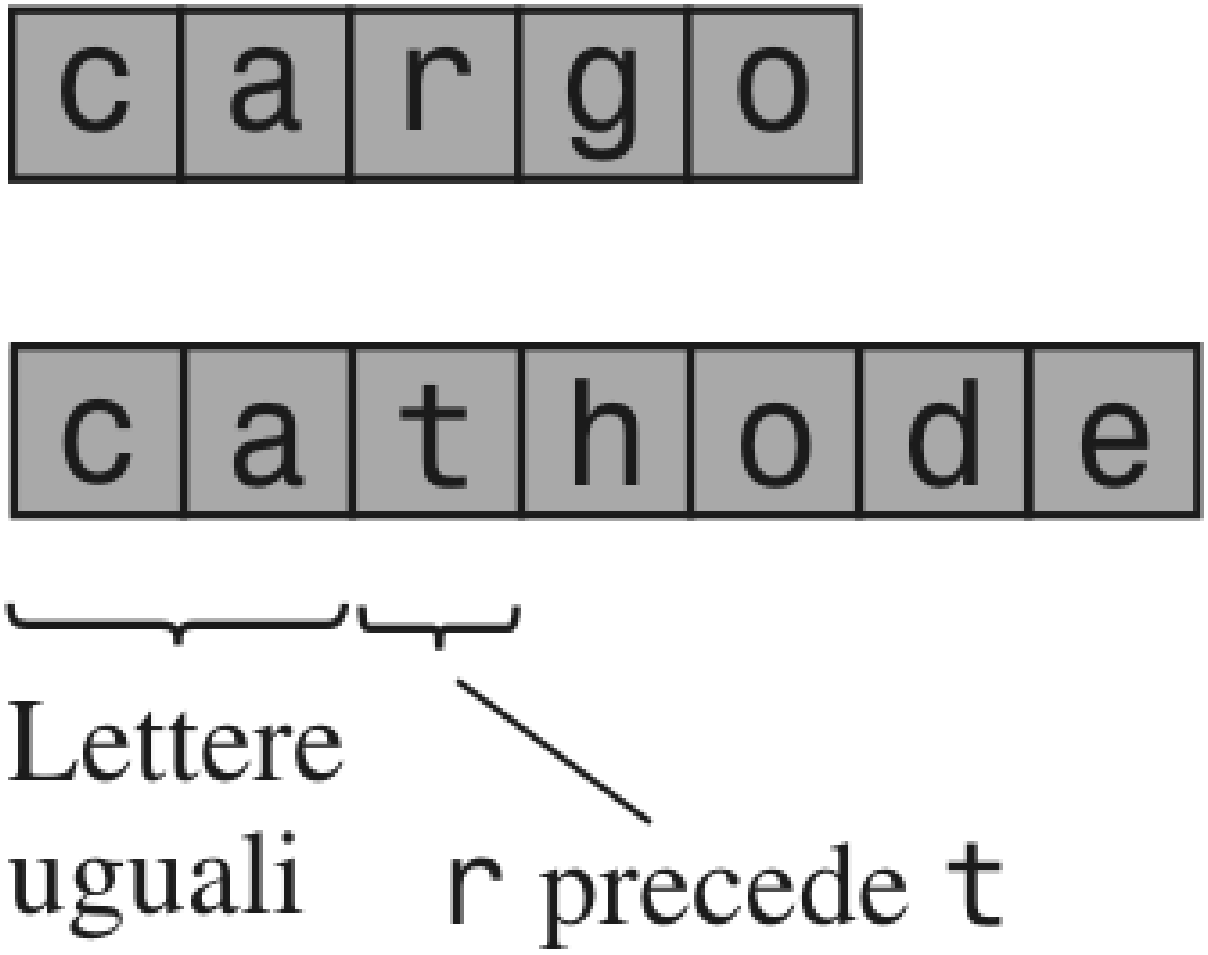

Figura 3: Confronto lessicografico

Confrontare oggetti

- L'operatore `==` verifica se due riferimenti a oggetto sono identici. Per confrontare, invece, i contenuti di oggetti, si deve usare il metodo `equals`.

```
Rectangle box1 = new Rectangle(5, 10, 20, 30);
Rectangle box2 = box1;
Rectangle box3 = new Rectangle(5, 10, 20, 30);
```

- non `box1 == box3`,
  ma `box1.equals(box3)`
- `equals` deve essere definito nella classe

Confrontare oggetti

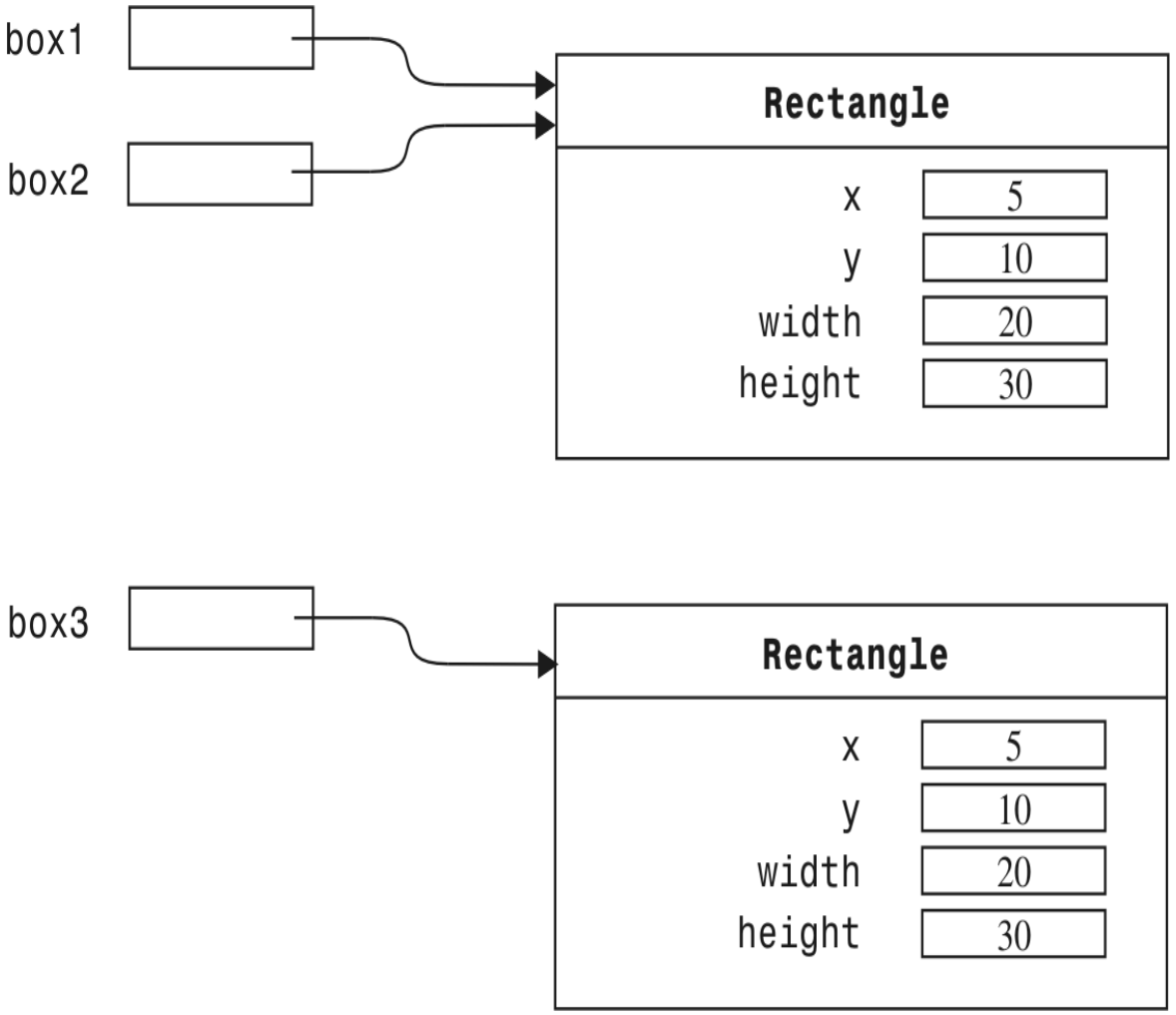

Figura 4:
Confronto tra riferimenti a oggetti

Confrontare con `null`

- Il riferimento `null` non fa riferimento ad alcun oggetto.

```
String middleInitial = null; // Valore iniziale non valido
if ( . . . )
   middleInitial = middleName.substring(0, 1);
```

- Per verificare se un riferimento è null si usa l'operatore == (e non `equals`):

```
if (middleInitial == null)
   System.out.println(firstName + " " + lastName);
else
   System.out.println(firstName + " " + middleInitial + ". "
         + lastName);
```

Confrontare con `null`

- Un riferimento `null` è diverso da una stringa vuota `""`.
- La stringa vuota è una stringa valida di lunghezza 0, mentre il valore `null` indica che una variabile di tipo stringa non si riferisce ad alcuna stringa.

Alternative multiple: Sequenze di confronti

```
if (condizione1)
   enunciato1;
else if (condizione2)
   enunciato2;
. . .
else
   enunciato4;
```

- La prima sequenza di confronti è eseguita
- Ordinare le condizioni

```
if (richter >= 0) // sempre vero
   r = "Generally not felt by people";
else if (richter >= 3.5) // mai verificato
   r = "Felt by many people, no destruction
. . .
```

Alternative multiple: Sequenze di confronti

- Non dimenticare

```
if (richter >= 8.0)
   r = "Most structures fall";
if (richter >= 7.0) // else mancante -- ERRORE
   r = "Many buildings destroyed
```

File Earthquake.java

```
01: /**
02:    Una classe che descrive gli effetti di un terremoto.
03: */
04: public class Earthquake
05: {
06:    /**
07:       Costruisce un oggetto che rappresenta un terremoto.
08:       @param magnitude the magnitude on the Richter scale
09:    */
10:    public Earthquake(double magnitude)
11:    {
12:       richter = magnitude;
13:    }
14:
15:    /**
16:       Restituisce la descrizione dell'effetto del terremoto.
17:       @return the la descrizione dell'effetto
18:    */
```

Segue

File Earthquake.java

```
19:     public String getDescription()
20:     {
21:        String r;
22:        if (richter >= 8.0)
23:           r = "Most structures fall";
24:        else if (richter >= 7.0)
25:           r = "Many buildings destroyed";
26:        else if (richter >= 6.0)
27:           r = "Many buildings considerably damaged, some
                    collapse";
28:        else if (richter >= 4.5)
29:           r = "Damage to poorly constructed buildings";
30:        else if (richter >= 3.5)
31:           r = "Felt by many people, no destruction";
32:        else if (richter >= 0)
33:           r = "Generally not felt by people";
34:        else
35:           r = "Negative numbers are not valid";
36:        return r;
37:     }
```

Segue

File Earthquake.java

```
38:
39: private double richter;
40: }
```

File EarthquakeRunner.java

```
01: import java.util.Scanner;
02:
03: /**
04:    Programma che visualizza la descrizione di un terremoto.
05: */
06: public class EarthquakeRunner
07: {
08:    public static void main(String[] args)
09:    {
10:       Scanner in = new Scanner(System.in);
11:
12:       System.out.print("Enter a magnitude on the Richter scale: ");
13:       double magnitude = in.nextDouble();
14:       Earthquake quake = new Earthquake(magnitude);
15:       System.out.println(quake.getDescription());
16:    }
17: }
```

Visualizza: Enter a magnitude on the Richter scale: 7.1 Many buildings destroyed

Alternative multiple: Diramazioni annidate

- Una diramazione all'interno di un'altra

```
if (condizione1)
{
   if (condizione1a)
      enunciato1a;
   else
      enunciato1b;
}
else
   enunciato2;
```

Tabella 1

Se il vostro stato civile è "non coniugato"		Se il vostro stato civile è "coniugato"	
Scaglione fiscale	**Aliquota**	**Scaglione fiscale**	**Aliquota**
$ 0 ... $ 21 450	15%	$ 0 ... $ 35 800	15%
Reddito superiore a $ 21 450 fino a $ 51 900	28%	Reddito superiore a $ 35 800 fino a $ 86 500	28%
Reddito superiore a $ 51 900	31%	Reddito superiore a $ 86 500	31%

Tabella 1 Aliquote per categorie d'imposta federali (1992)

Diramazioni annidate

- Calcolare il carico fiscale, fornendo uno stato civile e l'importo di un reddito.
  1. scegliere la diramazione relativa allo stato civile
  2. per ciascuno stato civile, imboccare un'altra diramazione in base allo scaglione di reddito
- Il processo decisionale a due livelli si traduce mediante due livelli di enunciati `if`
- Si dice che la verifica del reddito è *annidata* all'interno di quella per la classificazione dello stato civile

Diramazioni annidate

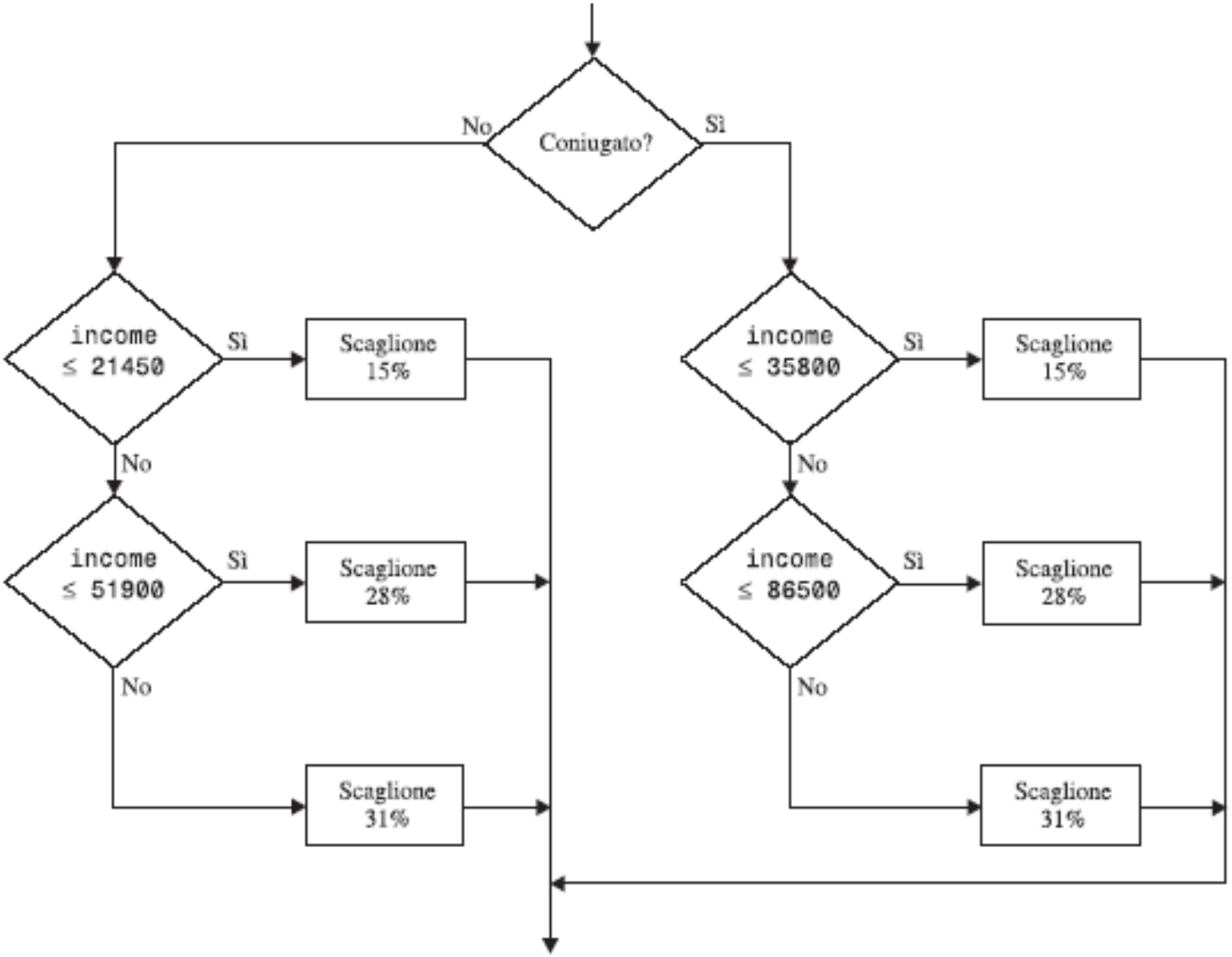

Figura 5 Schema per il calcolo dell'imposta sul reddito 1992

File TaxReturn.java

```
01: /**
02:    Dichiarazione dei redditi per un contribuente nel 1992.
03: */
04: public class TaxReturn
05: {
06:    /**
07:       Costruisce una dichiarazione per un dato importo
08:       del reddito e per un dato stato civile
09:       @param anIncome il reddito del contribuente
10:       @param aStatus la costante SINGLE o MARRIED
11:    */
12:    public TaxReturn(double anIncome, int aStatus)
13:    {
14:       income = anIncome;
15:       status = aStatus;
16:    }
17:
```

Segue

File TaxReturn.java

```
18:    public double getTax()
19:    {
20:       double tax = 0;
21:
22:       if (status == SINGLE)
23:       {
24:          if (income <= SINGLE_BRACKET1)
25:             tax = RATE1 * income;
26:          else if (income <= SINGLE_BRACKET2)
27:             tax = RATE1 * SINGLE_BRACKET1
28:                   + RATE2 * (income - SINGLE_BRACKET1);
29:          else
30:             tax = RATE1 * SINGLE_BRACKET1
31:                   + RATE2 * (SINGLE_BRACKET2 - SINGLE_BRACKET1)
32:                   + RATE3 * (income - SINGLE_BRACKET2);
33:       }
```

Segue

File TaxReturn.java

```
34:         else
35:         {
36:            if (income <= MARRIED_BRACKET1)
37:               tax = RATE1 * income;
38:            else if (income <= MARRIED_BRACKET2)
39:               tax = RATE1 * MARRIED_BRACKET1
40:                     + RATE2 * (income - MARRIED_BRACKET1);
41:            else
42:               tax = RATE1 * MARRIED_BRACKET1
43:                     + RATE2 * (MARRIED_BRACKET2 - MARRIED_BRACKET1)
44:                     + RATE3 * (income - MARRIED_BRACKET2);
45:         }
46:
47:         return tax;
48:      }
49:
50:      public static final int SINGLE = 1;
51:      public static final int MARRIED = 2;
52:
```

File TaxReturn.java

```
53:    private static final double RATE1 = 0.15;
54:    private static final double RATE2 = 0.28;
55:    private static final double RATE3 = 0.31;
56:
57:    private static final double SINGLE_BRACKET1 = 21450;
58:    private static final double SINGLE_BRACKET2 = 51900;
59:
60:    private static final double MARRIED_BRACKET1 = 35800;
61:    private static final double MARRIED_BRACKET2 = 86500;
62:
63:    private double income;
64:    private int status;
65: }
```

File TaxCalculator.java

```
01: import java.util.Scanner;
02:
03: /**
04:    Programma che calcola una semplice dichiarazione dei redditi.
05: */
06: public class TaxCalculator
07: {
08:    public static void main(String[] args)
09:    {
10:       Scanner in = new Scanner(System.in);
11:
12:       System.out.print("Please enter your income: ");
13:       double income = in.nextDouble();
14:
15:       System.out.print("Are you married? (Y/N) ");
16:       String input = in.next();
17:       int status;
18:       if (input.equalsIgnoreCase("Y"))
19:          status = TaxReturn.MARRIED;
20:       else
21:          status = TaxReturn.SINGLE;
22:
```

File TaxCalculator.java

```
23:         TaxReturn aTaxReturn = new TaxReturn(income, status);
24:
25:         System.out.println("Tax: "
26:               + aTaxReturn.getTax());
27:     }
28: }
```

Visualizza:

```
Please enter your income: 50000
Are you married? (Y/N) N
Tax: 11211.5
```

Utilizzare le espressioni booleane
Il tipo `boolean`

- George Boole (1815-1864): pioniere nello studio della logica
- Il valore dell'espressione `amount < 1000` è `true` o `false`.
- Il tipo boolean ha due valori: `true` e `false`

Utilizzare le espressioni booleane
Metodi predicativi

- Un metodo predicativo restituisce un valore booleano

```
public boolean isOverdrawn()
{
   return balance < 0;
}
```

- Il valore restituito dal metodo può essere utilizzato come condizione di un enunciato `if`:

```
if (harrysChecking.isOverdrawn()) . . .
```

Segue

Utilizzare le espressioni booleane
Metodi predicativi

- Nella classe `Character` sono presenti parecchi utili metodi predicativi statici:

```
isDigit
isLetter
isUpperCase
isLowerCase
```

- che consentono di verificare i caratteri

```
if (Character.isUpperCase(ch)) . . .
```

- La classe `Scanner` ha metodi predicativi utili per verificare se una successiva richiesta di dati in ingresso andrà a buon fine: `hasNextInt` e `hasNextDouble`

```
if (in.hasNextInt()) n = in.nextInt();
```

Utilizzare le espressioni booleane
Gli operatori booleani

- `&&` and
- `||` or
- `!` not
-
```
if (0 < amount && amount < 1000) . . .
```
-
```
if (input.equals("S") || input.equals("M")) . . .
```

Gli operatori booleani

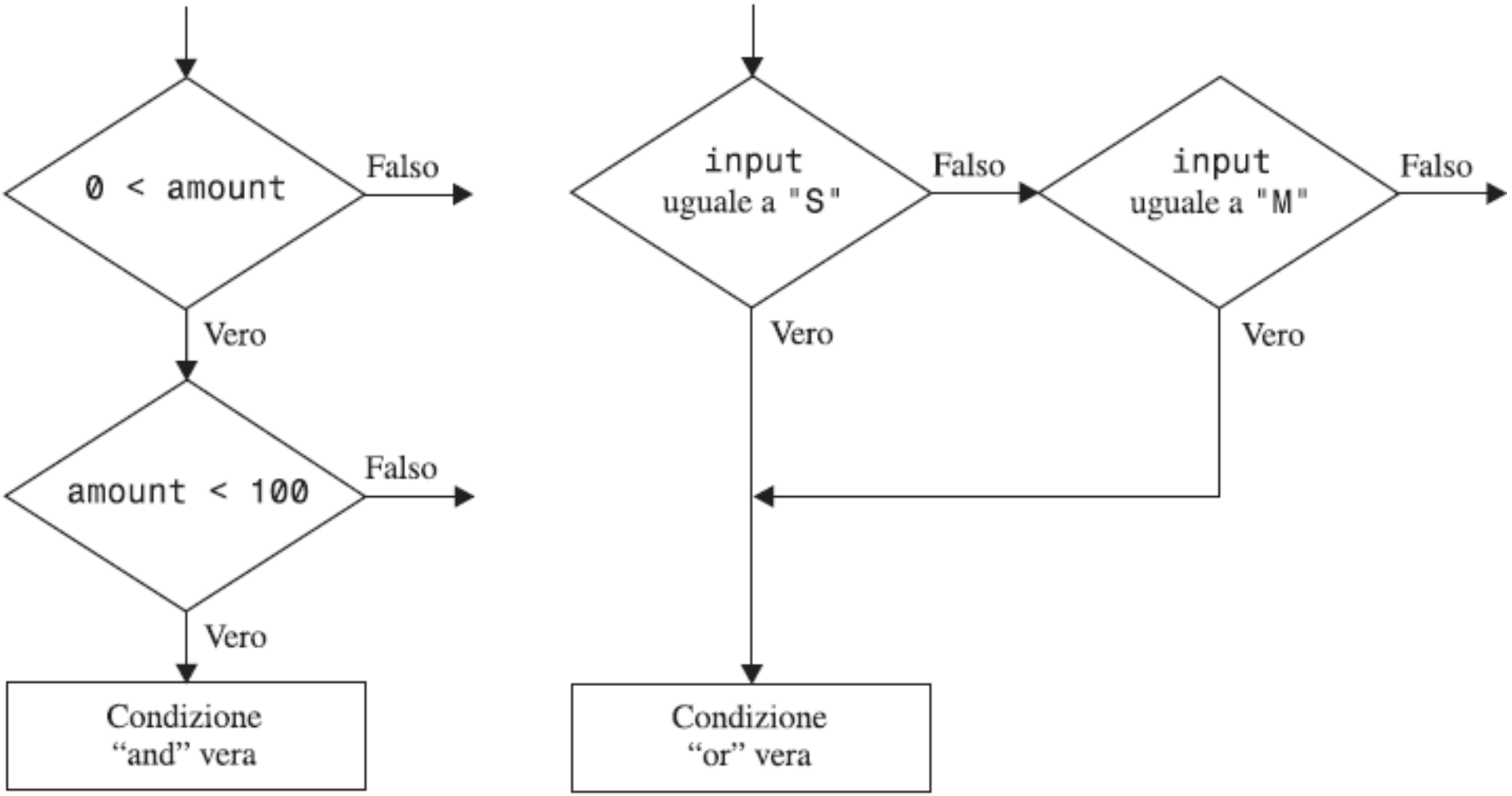

Figura 6 Diagrammi di flusso con operatori `&&` e `||`

Riepilogo delle tre operazioni logiche

A	B	A && B
vero	vero	vero
vero	falso	falso
falso	*qualsiasi valore*	falso

A	B	A || B
vero	*qualsiasi valore*	vero
falso	vero	vero
falso	falso	falso

A	!A
vero	falso
falso	vero

Utilizzare variabili booleane

- ```
  private boolean married;
  ```

- Il risultato della verifica di una condizione può essere memorizzato in una variabile booleana.

```
if (married) . . . else . . .
if (!married) . . .
```

- Le variabili booleane sono dette *flag*, perché hanno solo due stati, "su" o "giù".
```

Utilizzare variabili booleane

- Valutare attentamente i nomi da assegnare alle variabili booleane dà i risultati migliori.
  - Da non fare

```
if (married == true) . . . // Da non fare
```

  - Usare una condizione semplice

```
if (married) . . .
```

Tipi enumerativi

- Una variabile di un tipo enumerativo può assumere soltanto un valore che appartenga a un insieme appositamente definito
- Al posto delle costanti

```
public static final int SINGLE = 1;
public static final int MARRIED = 2;
```

- Nella classe, al livello degli attributi si dichiara il tipo

```
public enum FilingStatus { SINGLE, MARRIED }
private FilingStatus status;
```

```
FilingStatus status = FilingStatus.SINGLE;
```

Copertura del collaudo

- Il collaudo a scatola chiusa (black-box testing) descrive una metodologia di collaudo che non prende in considerazione la struttura dell'implementazione.

- Il collaudo trasparente (white-box testing) usa informazioni sulla struttura del programma.

- La copertura di un collaudo è una misura di quante parti di un programma siano state collaudate.

Copertura del collaudo

- Assicurarsi che **ogni** porzione del programma venga collaudata da almeno uno dei casi di prova.

- Considerate anche i **casi limite**: valori validi dei dati di ingresso che si trovano al limite di tali valori accettabili.

- Suggerimento: **calcolare alcuni casi di prova prima di scrivere il programma**. Questa pratica aiuta a comprendere la natura del problema e agevola la realizzazione del programma stesso.

Copertura del collaudo

Caso di prova	Sposato	Previsto	Commento
30 000	No	3000	Scaglione 10%
72 000	No	13 200	3200 + 25% di 40 000
50 000	Sì	5000	Scaglione 10%
104 000	Sì	16 400	6400 + 25% di 40 000
32 000	No	3200	Caso limite
0		0	Caso limite

Logging

- Invece di usare
  - System.out.println(“status is SINGLE”);
- Usare
  - Logger.getGlobal().info(“status is SINGLE”);

- I messaggi di tracciamento possono essere disattivati dopo aver completato il collaudo
  - Logger.getGlobal().setLevel(Level.OFF);