Programmazione
Condizioni, Istruzioni iterative

E. Omodeo

Università degli Studi di Trieste, a.a. 2015/16

Trieste, 24/28.09.2015

Per condizioni intendiamo le espressioni a valore booleano

Le condizioni piú semplici designano direttamente i due valori di verità:

false, **true** .

Per condizioni intendiamo le espressioni a valore booleano

Le condizioni piú semplici designano direttamente i due valori di verità:

false, **true** .

Per chi preferisce l'italiano:

```
final boolean falso = false ;

final boolean vero = true ;
```

Per condizioni intendiamo le espressioni a valore booleano

Le condizioni piú semplici designano direttamente i due valori di verità:

false, **true** .

Per chi preferisce l'italiano:

final boolean `falso` = **false** ;

final boolean `vero` = **true** ;

Dichiarando una variabile **final** *si impedisce che il suo valore possa piú cambiare dopo la prima assegnazione: cosí la si rende, in pratica, una costante.*

Anche i confronti hanno esito booleano

Che significato dare a:

```
int      n      = ...
int      d      = ...
boolean  divide = n == (n / d) * d ;
```

Anche i confronti hanno esito booleano

Che significato dare a:

```
int      n       =  ...

int      d       =  ...

boolean  divide  =  n == (n / d) * d ;
```

La variabile `divide` *ci rivelerà se il denominatore* `d` *è, o no, un divisore esatto del numeratore* `n` *.*

N.B.: *Ci stiamo muovendo nell'aritmetica intera!*

Anche i confronti hanno esito booleano

Che significato dare a:

```
int      n       =  ...

int      d       =  ...

boolean  divide  =  n == (n / d) * d ;
```

La variabile `divide` *ci rivelerà se il denominatore* `d` *è, o no, un divisore esatto del numeratore* `n` *.*

N.B.: *Ci stiamo muovendo nell'aritmetica intera!*

(E se `d` valesse 0 ?)

Che ha d'importante un dominio formato da due valori ?

- Le componenti a **2 stati** bastano per la rappresentazione digitalizzata dell'informazione:

Che ha d'importante un dominio formato da due valori ?

- Le componenti a **2 stati** bastano per la rappresentazione digitalizzata dell'informazione:
- a cominciare dai numeri, in un sistema digitale di elaborazione tutto è rappresentato tramite **bit** (binary digit: **0**, **1**).

Che ha d'importante un dominio formato da due valori ?

- Le componenti a **2 stati** bastano per la rappresentazione digitalizzata dell'informazione:
- a cominciare dai numeri, in un sistema digitale di elaborazione tutto è rappresentato tramite **bit** (binary digit: **0**, **1**).
- Gli esiti di confronti a **due vie** (successo / fallimento) possono guidare l'ordine di esecuzione delle istruzioni di un programma.

Rappresentazione dei numeri naturali

Numeri decimali	Numeri binari
0	0
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

Numeri decimali e binari

Diagramma di flusso di una **while**

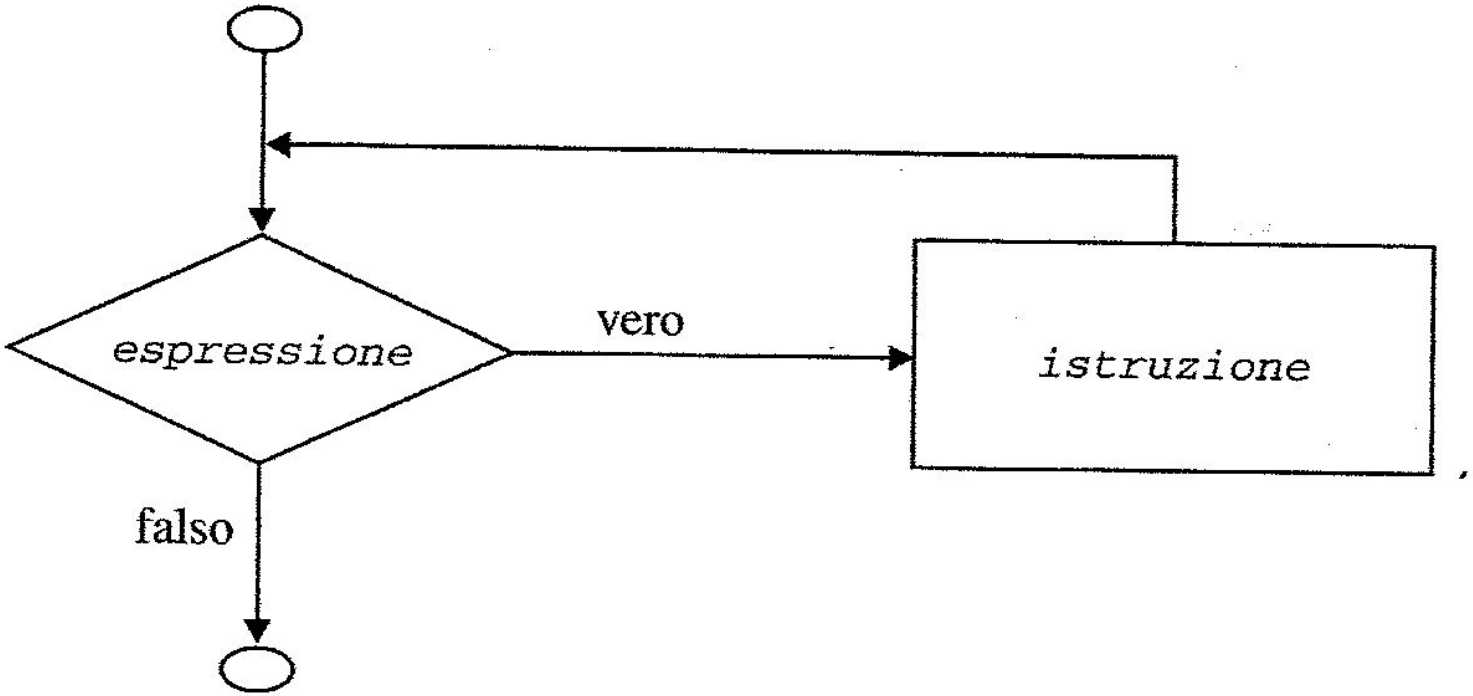


Figura 2.8 Diagramma di flusso con una struttura di controllo del flusso iterativa

Diagramma di flusso di una **while**

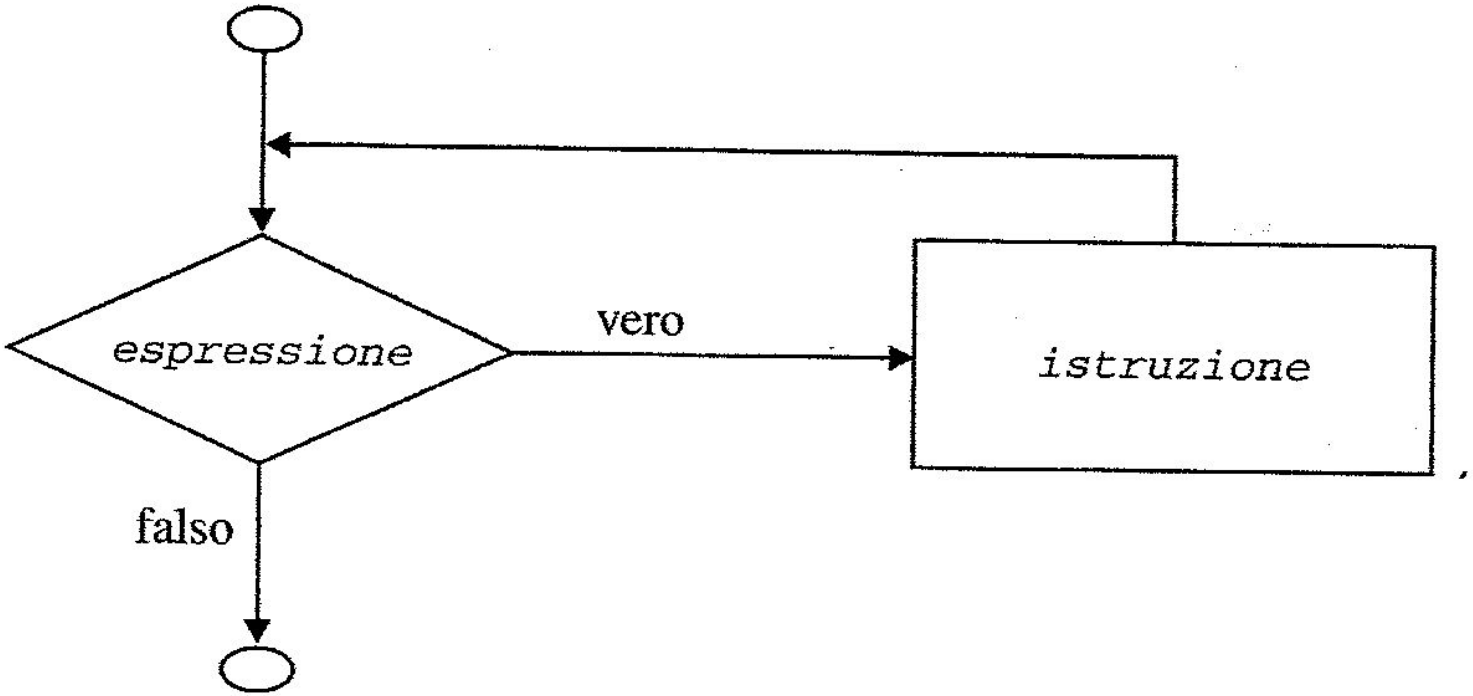


Figura 2.8 Diagramma di flusso con una struttura di controllo del flusso iterativa

(N.B: Qui l'espressione è booleana !)

La negazione

Per comodità—solo per la durata di cinque lucidi—rappresentiamo

- il falso come 0 ,
- il vero come 1 .

La negazione

Per comodità—solo per la durata di cinque lucidi—rappresentiamo

- il falso come 0 ,
- il vero come 1 .

L'operazione logica dalla tabellina piú semplice è la *negazione*:

!	0	1
	1	0

La negazione

Per comodità—solo per la durata di cinque lucidi—rappresentiamo

- il falso come 0 ,
- il vero come 1 .

L'operazione logica dalla tabellina piú semplice è la *negazione*:

!	0	1
	1	0

Cosí, ad es., subito dopo

```
int n = 0;
```

risulta che

```
! ( n == 2 )
```

```
n != 1 - 1
```

La negazione

Per comodità—solo per la durata di cinque lucidi—rappresentiamo

- il falso come 0 ,
- il vero come 1 .

L'operazione logica dalla tabellina piú semplice è la *negazione*:

!	0	1
	1	0

Cosí, ad es., subito dopo

```
int n = 0;
```

risulta che

```
! ( n == 2 )
```

true

```
n != 1 - 1
```

false

Congiunzione, o prodotto logico

La tabellina della *congiunzione*—stretta parente della "ed" italiana—è :

&&	0	1
0	0	0
1	0	1

Congiunzione, o prodotto logico

La tabellina della *congiunzione*—stretta parente della "ed" italiana—è :

&&	0	1
0	0	0
1	0	1

Cosí, ad es., le istruzioni Java

```
System.out.println( 3 != 0 && 21 == (21 / 3) * 3 );
System.out.println( 2 != 0 && 21 == (21 / 2) * 2 );
System.out.println( 0 != 0 && 21 == (21 / 0) * 0 );
```

produrranno la stampa incolonnata di , e (?!)

Congiunzione, o prodotto logico

La tabellina della *congiunzione*—stretta parente della "ed" italiana—è :

&&	0	1
0	0	0
1	0	1

Cosí, ad es., le istruzioni Java

```
System.out.println( 3 != 0 && 21 == (21 / 3) * 3 );
System.out.println( 2 != 0 && 21 == (21 / 2) * 2 );
System.out.println( 0 != 0 && 21 == (21 / 0) * 0 );
```

produrranno la stampa incolonnata di **true**, **false** e **false** (?!)

Disgiunzione, o 'somma' logica

La tabellina della *disgiunzione*—stretta parente della "vel" latina—è :

$\|\|$	0	1
0	0	1
1	1	1

Disgiunzione, o 'somma' logica

La tabellina della *disgiunzione*—stretta parente della "vel" latina—è :

$\|\|$	0	1
0	0	1
1	1	1

Cosí, ad es., le istruzioni Java

```
System.out.println( !( 3 == 0 || 21 != (21 / 3) * 3 ) );
System.out.println( !( 2 == 0 || 21 != (21 / 2) * 2 ) );
System.out.println( !( 0 == 0 || 21 != (21 / 0) * 0 ) );
```

produrranno la stampa incolonnata di , e (?!)

Disgiunzione, o 'somma' logica

La tabellina della *disgiunzione*—stretta parente della "vel" latina—è :

$\|\|$	0	1
0	0	1
1	1	1

Cosí, ad es., le istruzioni Java

```
System.out.println( !( 3 == 0 || 21 != (21 / 3) * 3 ) );
System.out.println( !( 2 == 0 || 21 != (21 / 2) * 2 ) );
System.out.println( !( 0 == 0 || 21 != (21 / 0) * 0 ) );
```

produrranno la stampa incolonnata di **true**, **false** e **false** (?!)

Morale (importante!): Occhio a cosa mettere prima!

Per quanto la **&&** e la || siano commutative, dall'ordine dei loro operandi può dipendere se un programma

- s'interrompe con un *crash* o meno ;

Morale (importante!): Occhio a cosa mettere prima!

Per quanto la **&&** e la || siano commutative, dall'ordine dei loro operandi può dipendere se un programma

- s'interrompe con un *crash* o meno ;
- si addentra in un ciclo perpetuo o lo aggira .

Leggi di De Morgan

Quando E ed F sono espressioni booleane,

$!!E$	equivale a	E
$!(\ E\ \&\&\ F\ )$	equivale a	$(\ !E\ )\ \|\|\ (\ !F\ )$
$!(\ E\ \|\|\ F\ )$	equivale a	$(\ !E\ )\ \&\&\ (\ !F\ )$

Prima sgrossata di un test di quadrato perfetto

Come escludere, controllando solo l'ultima cifra di un dato naturale `n` che possa essere un quadrato ?

Prima sgrossata di un test di quadrato perfetto

Come escludere, controllando solo l'ultima cifra di un dato naturale `n` che possa essere un quadrato ?

0 1 2 3 4 5 6 7 8 9

Prima sgrossata di un test di quadrato perfetto

Come escludere, controllando solo l'ultima cifra di un dato naturale `n` che possa essere un quadrato ?

0 1 2 3 4 5 6 7 8 9

Prima sgrossata di un test di quadrato perfetto

Come escludere, controllando solo l'ultima cifra di un dato naturale `n` che possa essere un quadrato ?

0 1 2 3 4 5 6 7 8 9

. . . .

```
int ult = n%10;
boolean xclQuad = ult==2 || ult==3 || ult==7 || ult==8;
```

Prima sgrossata di un test di quadrato perfetto

Come escludere, controllando solo l'ultima cifra di un dato naturale `n` che possa essere un quadrato ?

0 1 2 3 4 5 6 7 8 9

o anche (dualmente) :

. . . .

```
boolean forseQuad = (n%10)%5 != 2 && (n%10)%5 != 3;
```

Test pitagorico di quadrato perfetto 'ottimizzato'

```
class QQuadrato{ // metodo del VI secolo a.C.

    public static void main( String[ ] aaa ){

        int n = Leggi.leggiInt(  ), // numero da testare

            q = 0; // numeri quadrati in successione

        boolean forseQuadr = n >= 0 && (n % 10 % 5 != 2) && (n % 10 % 5 != 3);

        if ( forseQuadr ) // risparmiamo calcoli escludendo casi banali

            for( int d = 1; q < n; d += 2 ) q += d;

        Leggi.emettiMessaggio( n + (( forseQuadr && q == n ) ? "" : " non")
                                 + " e` quadrato" );
    }
}
```

La disgiunzione esclusiva

Al di là dell'estetica delle leggi ora viste, c'è da chiedersi se non sia da considerarsi *somma logica*, invece di ||, la *disgiunzione esclusiva*—stretta parente della "aut" latina—, che certo inglese tecnico chiama "xor":

xor	0	1
0	0	1
1	1	0

La disgiunzione esclusiva

Al di là dell'estetica delle leggi ora viste, c'è da chiedersi se non sia da considerarsi *somma logica*, invece di ||, la *disgiunzione esclusiva*—stretta parente della "aut" latina—, che certo inglese tecnico chiama "xor":

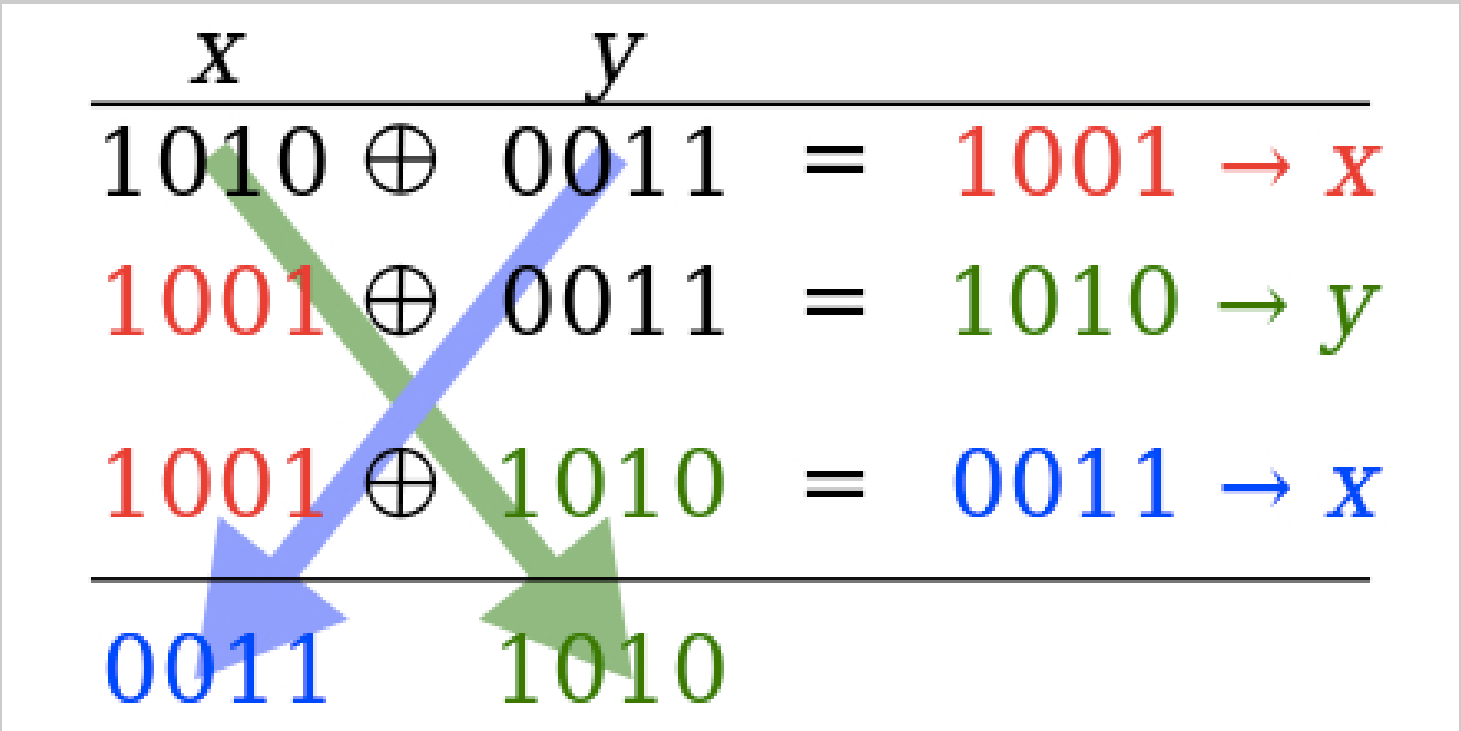
xor	0	1
0	0	1
1	1	0

Si può esprimere quest'operazione logica in Java ?

Una tecnica per scambiare di posto due valori (Illustraz.)

```
class Scambio{

  public static void main( String[] aa){
// Swap xor algorithm, vedi
//     http://en.wikipedia.org/wiki/XOR_swap_algorithm

    int x = Leggi.leggiInt( ), y = Leggi.leggiInt( );

    System.out.println("x="+x+"; y="+y);
    x = x ^ y;     y = y ^ x;      x = x ^ y;
    System.out.println("x="+x+"; y="+y);

    long a = Leggi.leggiLong( ), b = Leggi.leggiLong( );

    System.out.println("a="+a+"; b="+b);
    a ^= b;     b ^= a;      a ^= b;
    System.out.println("a="+a+"; b="+b);
   }
}
```

Una tecnica per scambiare di posto due valori (Specifica)



Using the XOR swap algorithm to exchange nibbles between variables without the use of temporary storage

E se al posto di ^ mettessi & oppure | ?

Che succede, ad es., se faccio:

```
a = 300;   b = 400;
a &= b;    b &= a;    a &= b
```

?

E se al posto di ^ mettessi & oppure | ?

Che succede, ad es., se faccio:

```
a = 300; b = 400;
a &= b; b &= a; a &= b ?
```

(Sperimentalm. ho osservato che risulta $\boxed{a = b = 256}$)

Un test di divisibilità

Un test piú soddisfacente di quello visto all'inizio :

```
int      n      = ...
int      d      = ...
boolean  divide = d != 0 && n == (n / d) * d ;
```

Un test di divisibilità

Un test piú soddisfacente di quello visto all'inizio :

```
int     n      = ...

int     d      = ...

boolean divide = d != 0 && n == (n / d) * d ;
```

D.: Potremmo qui scambiare i fattori della && ?

Un test di divisibilità

Un test piú soddisfacente di quello visto all'inizio :

```
int     n      =  ...

int     d      =  ...

boolean divide =  d != 0 && n == (n / d) * d ;
```

D.: Potremmo qui scambiare i fattori della && ?

E: Vedete qui un ritrovato per determinare il resto , `n % d` , della divisione tra due interi ?

Conversione di booleani in interi

```
// per cambiare la rappresentazione dei due valori di verita',
// tramutandoli in interi corti, useremo qui
// il ( raro ) operatore condizionale (vedi [Savitch] pag. 108)

    final short ffalso = boolean2short( falso ) ;
    final short vvero  = boolean2short( vero  ) ;

public static short boolean2short( boolean valBoo ) {

                        return ???;
}
```

Conversione di booleani in interi

```
// per cambiare la rappresentazione dei due valori di verita',
// tramutandoli in interi corti, useremo qui
// il ( raro ) operatore condizionale (vedi [Savitch] pag. 108)

    final short ffalso = boolean2short( falso ) ;
    final short vvero  = boolean2short( vero  ) ;

public static short boolean2short( boolean valBoo ) {

            return ( valBoo ) ? 1 : 0 ;
}
```

Altre conversioni di valore fra tipi primitivi (Digressione)

Osservazione: Generalmente le conversioni di tipo si effettuano in modo implicito, come in

```
double sette = 7 // assegna 7.0
```

Altre conversioni di valore fra tipi primitivi (Digressione)

Osservazione: Generalmente le conversioni di tipo si effettuano in modo implicito, come in

```
double sette = 7 // assegna 7.0
```

o tramite '*casting*', ad es.:

```
double contoCena = 26.99
int quantiEuro = (int)contoCena
                 // assegna 26
```

Istruzioni che incorporano condizioni

if: **if** (condizione) istruzione

if-else: **if** (condizione) istruzione_1 **else** istruzione_2

while: **while** (condizione) istruzione

do-while: **do** istruzione **while** (condizione)

for: **for** (inizializzaz.; condizione; aggiornam.) istruzione

Istruzioni che incorporano condizioni

if: **if** (condizione) istruzione

if-else: **if** (condizione) istruzione_1 **else** istruzione_2

while: **while** (condizione) istruzione

do-while: **do** istruzione **while** (condizione)

for: **for** (inizializzaz.; condizione; aggiornam.) istruzione

Spesso le istruzioni subordinate alle condizioni in questi quattro tipi di istruzione sono 'multiple', ossia composte a formare un blocco.

Diagramma di flusso di una **if**...

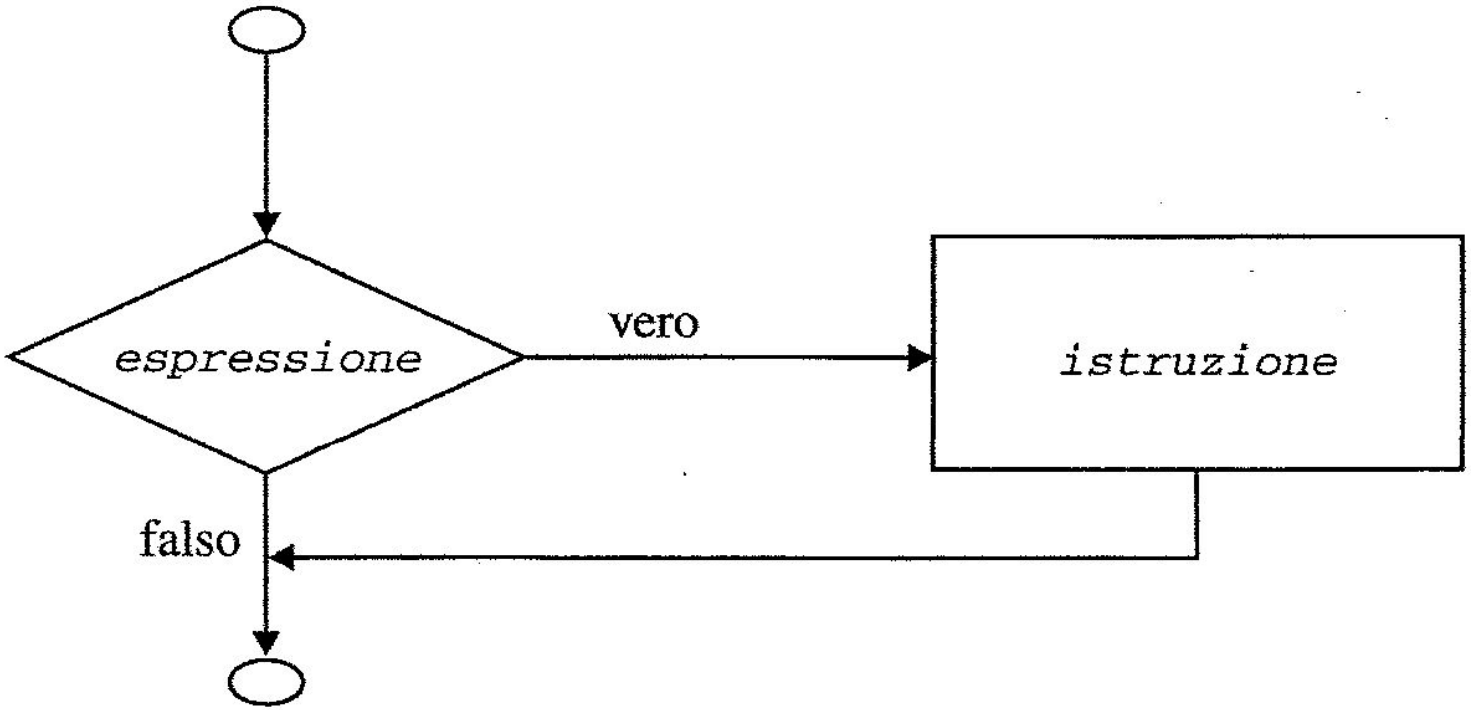


Figura 2.6 Diagramma di flusso con una struttura di controllo del flusso decisionale

Diagramma di flusso di una **if**...

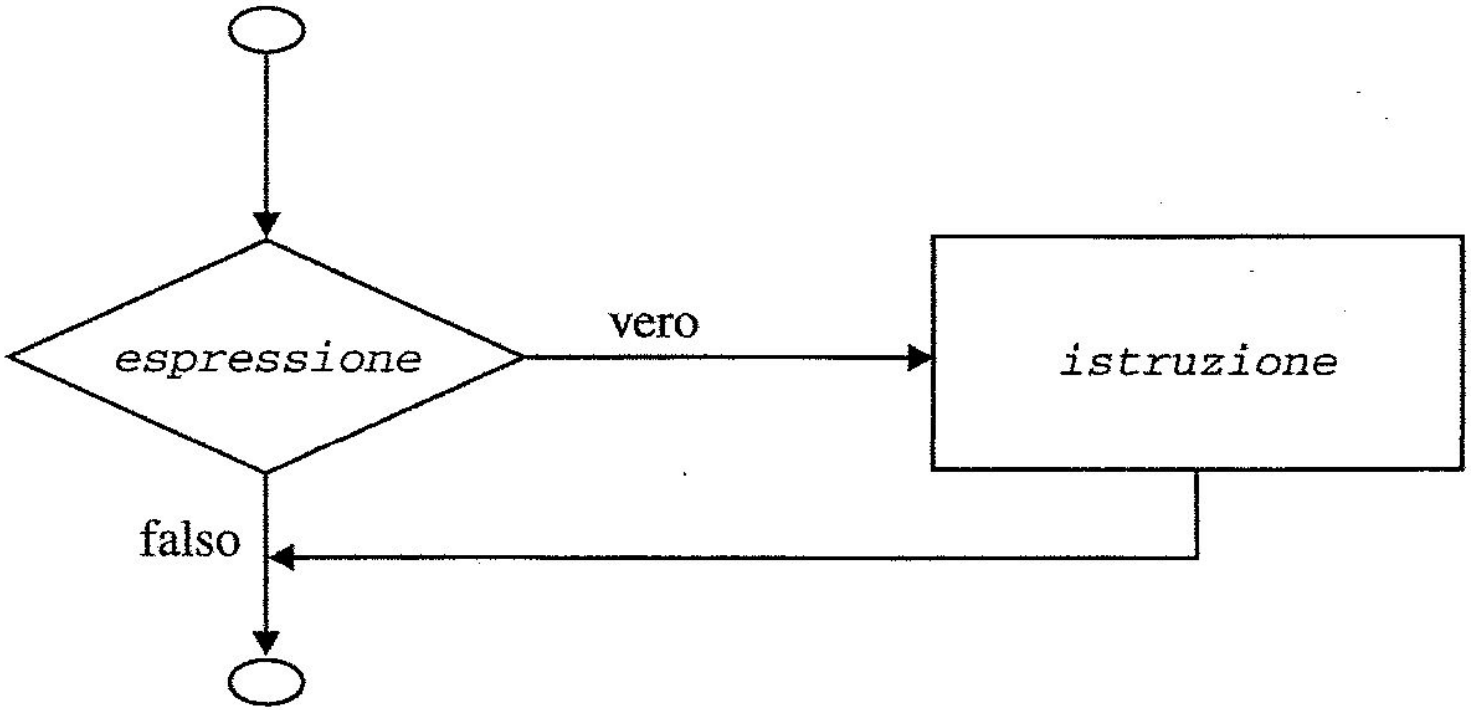


Figura 2.6 Diagramma di flusso con una struttura di controllo del flusso decisionale

(N.B: Anche qui l'espressione è booleana !)

. . . di una **if-else**. . .

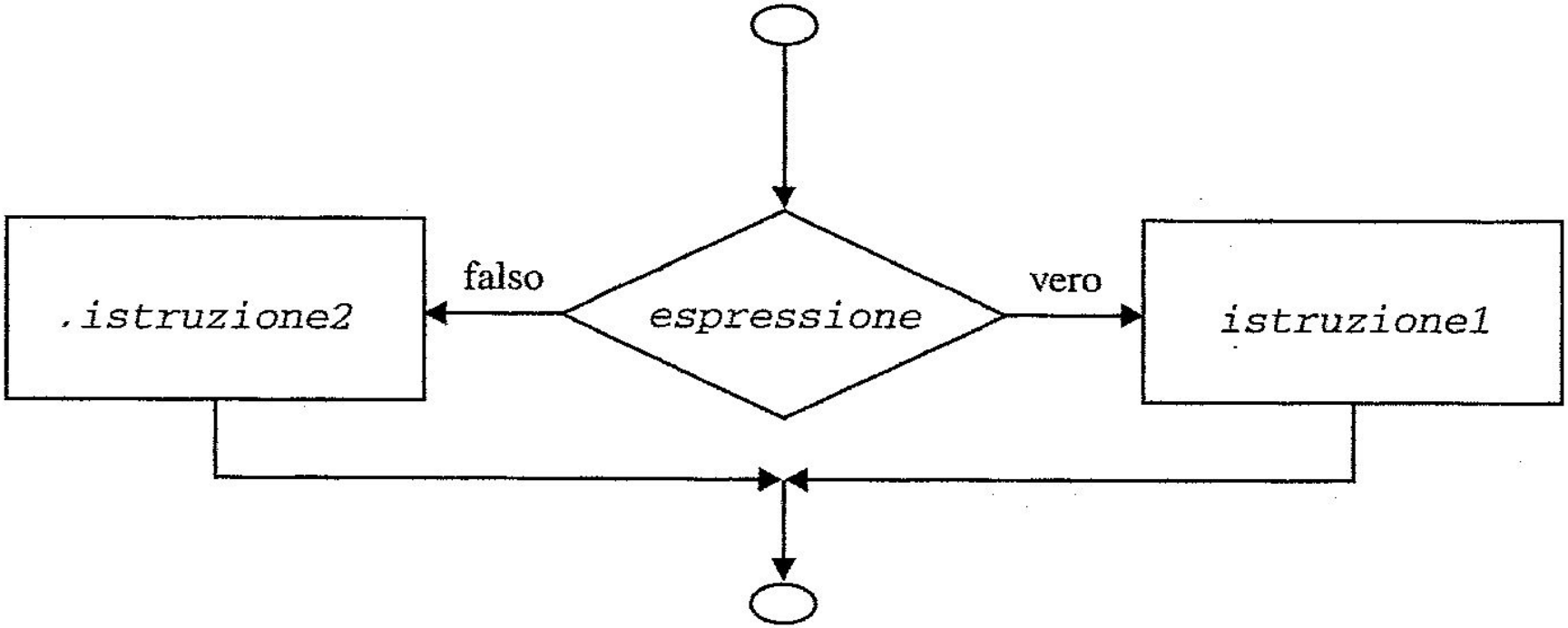


Figura 2.7 Diagramma di flusso con una struttura di controllo del flusso alternativa doppia

. . . di una **if-else**. . .

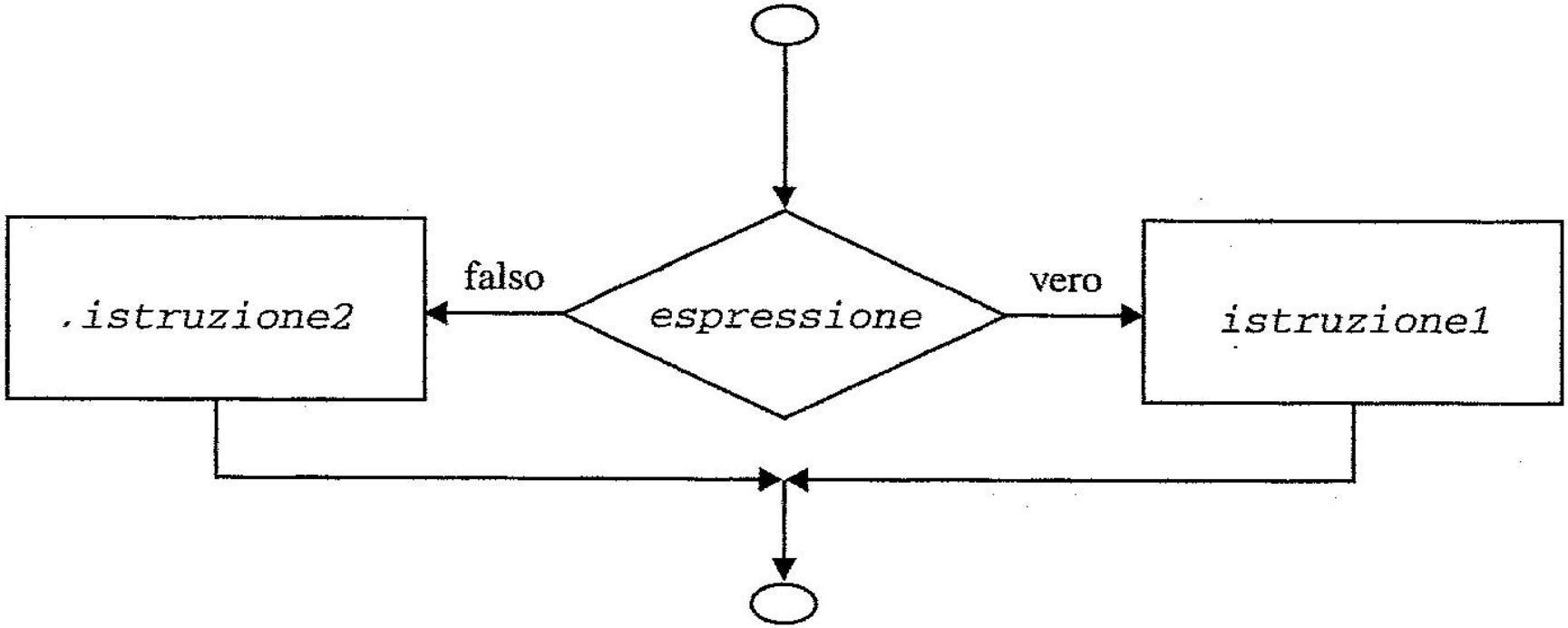


Figura 2.7 Diagramma di flusso con una struttura di controllo del flusso alternativa doppia

(N.B: Anche qui l'espressione è booleana !)

. . . di una **do-while**. . .

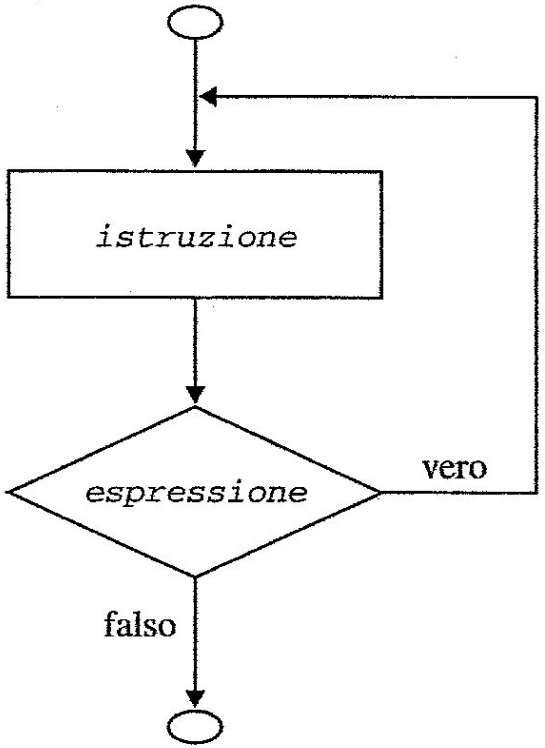


Figura 2.9 Diagramma di flusso con una seconda struttura di controllo del flusso iterativa

. . . di una **do-while**. . .

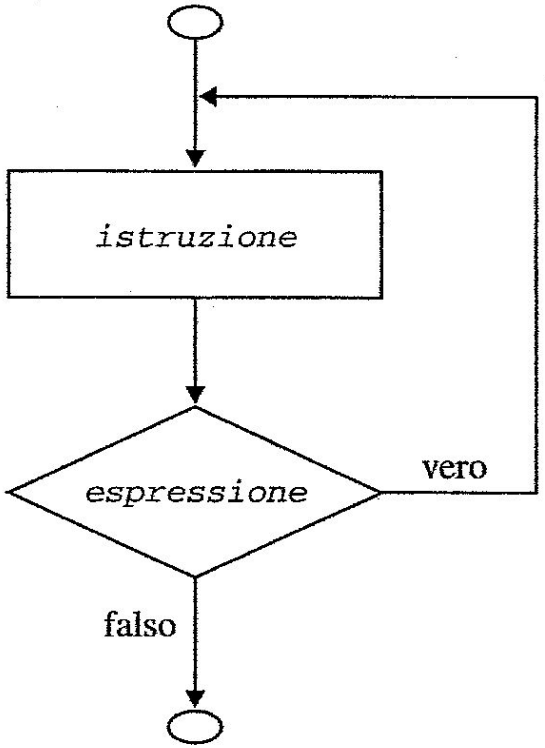


Figura 2.9 Diagramma di flusso con una seconda struttura di controllo del flusso iterativa

(N.B: Anche qui l'espressione è booleana !)

. . . di un'istruzione composta (o '*blocco*')

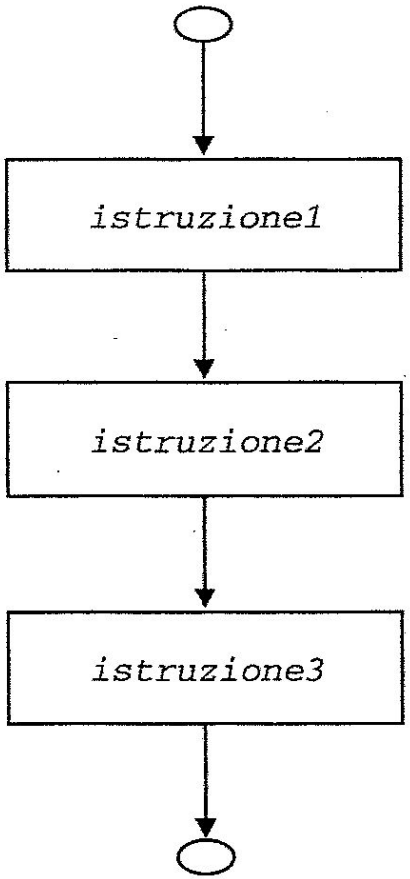


Diagramma di flusso con una struttura di controllo del flusso sequenziale

Test di primalità, per mezzo di una **while**...

```
boolean primo = ( n > 1 ) ; // sommaria approssimaz.
int d = 2; // primo numero da tentare come divisore
while ( primo && d < n ) {
   primo  =  n != ( n / d ) * d ;
   d = d + 1 ;
}
System.out.print(n) ;
if ( primo ) ; else System.out.print( " NON" ) ;
System.out.println( "e' un numero primo" ) ;
```

. . . per mezzo di una **for**. . .

```
boolean primo = ( n > 1 ) ; // sommaria approssimaz.
for ( int d = 2; d < n; d++ )
   primo  =  primo && n != ( n / d ) * d ;

// ecc.
```

Riduciamo il numero dei tentativi

```
boolean primo = ( n > 1 ) && ( n == 2 || n != (n/2) * 2 );
int d = 3; // primo numero da tentare come divisore
while ( primo && d * d <= n ) {
   primo  =  n != ( n / d ) * d ;
   d += 2 ; // abbreviazione di d = d + 2
}
// ecc.
```

Nuova variante che fa uso della **for**

```
boolean primo ;
// qui lo inizializziamo con un'istruzione if-else
if ( n <= 3 ) primo = n > 1 ; else
   primo  =  n != ( n / 2 ) * 2 ;

for ( int d = 3;  primo && d * d <= n;  d = d + 2 )
   primo  =  n != ( n / d ) * d ;
// ecc.
```

Variante della variante

```
// qui inizializzeremo primo tramite operatore condizionale
boolean primo  =
   ( n <= 3 )  ?  n > 1  :  n != ( n / 2 ) * 2 ;

for ( int d = 3;  primo && d * d <= n;  d = d + 2 )
   primo  =  n != ( n / d ) * d ;
// ecc.
```

Esercizi

- Leggere da tastiera una sequenza di numeri e stampare la loro somma ed il loro prodotto
- Leggere da tastiera un numero n, calcolarne il fattoriale $n!$ e stamparlo
- Dopo aver letto un numero n, stampare la sequenza dei primi n numeri di Fibonacci

Esercizio sulla for

Modificate il programma che effettua il test di primalità in modo che generi una tavola di *tutti* i numeri primi compresi fra 2 ed il numero n fornito in ingresso.

Esercizio sui diagrammi di flusso

Descrivete il funzionamento dell'istruzione **for** tramite un diagramma di flusso.

Esercizio sulla rappresentazione dei naturali in diverse basi

Scrivete in Java un convertitore di base che, ricevendo un numero n e l'indicazione di una base b tale che $1 < b \leqslant 16$, stampi il numero n nella sua rappresentazione b-aria.

Uno sguardo in avanti. . .

```
class CheTipoHa{

   public static void main( String[] aa){

     Object x = Integer.MIN_VALUE;

     System.out.println( x + est( x instanceof Integer ) + "e` intero" );

     System.out.println( x + est( x instanceof Short ) + "e` un intero piccolo" );

     System.out.println( x + est( x instanceof Long ) + "e` un intero grande" );

     System.out.println( x + est( x instanceof Float ) + "e` un reale" );

     System.out.println( x + est( x instanceof Object ) + "e` un oggetto" );
    }

   private static String est( boolean valVerita){

     return (valVerita) ? " " : " NON ";
   }
}
```

Uno sguardo in avanti...

```
-2147483648 e` intero
-2147483648 NON e` un intero piccolo
-2147483648 NON e` un intero grande
-2147483648 NON e` un reale
-2147483648 e` un oggetto
```