Azzolini Riccardo 2018-10-23

Conversioni e tipi numerici in virgola mobile

1 Conversioni implicite tra interi

In generale, i valori di un tipo più ristretto possono essere **convertiti implicitamente** (**promossi**) a un tipo più ampio. Non c'è possibilità di una perdita di informazione.

Ad esempio, un valore di tipo `int` viene promosso a `long` se:

- viene assegnato a una variabile di tipo `long`
- si deve effettuare un'operazione tra il valore `int` e un valore `long` (in tal caso, l'espressione risultante è di tipo `long`)

1.1 Esempi

```
int y;
long x, z;
x = y + z;
```

- Il valore di `y` viene convertito al corrispondente valore di tipo `long` prima dell'operazione.
- Il risultato è di tipo `long`.

```
int y, z;
long x;
x = y + z;
```

- Il risultato `int` di `x + y` viene convertito implicitamente a `long` prima di salvarlo in `x`.

```
int x, z;
long y;
x = y + z;
```

- Quest'espressione produce un errore: il risultato di tipo `long` non può essere assegnato a una variabile di tipo `int`.

2 Conversioni esplicite

Si effettuano tramite l'operatore di **cast** (*forzatura*):

```
(nome_di_tipo) espressione
```

La conversione da un tipo più ampio a uno più ristretto *può comportare una perdita di informazione*. Usando l'operatore di cast, il programmatore dichiara di esserne cosciente.

3 Tipi numerici in virgola mobile

- Rappresentano numeri **floating point** (cioè in *notazione scientifica*) basati sullo standard IEEE 754.
- Si distinguono per il range di valori rappresentabili e per la precisione.
- Sono rappresentazioni **approssimate**: non è possibile rappresentare tutti gli infiniti numeri nel range[1].
- Sono disponibili gli stessi operatori aritmetici definiti sui tipi numerici interi, con la stessa sintassi (anche se gli algoritmi corrispondenti sono diversi).

3.1 Formato

Un numero a virgola mobile è composto da *segno* (S), *esponente* (E) e *mantissa* (M). Il valore rappresentato è:

$$(-1)^S \cdot 1.M \cdot 2^E$$

	Segno	Esponente	Mantissa
`float` (32 bit)	1 bit	8 bit	23 bit
`double` (64 bit)	1 bit	11 bit	52 bit

[1] L'uso dei tipi floating point deve quindi essere evitato se sono necessari valori e risultati esatti, come ad esempio in ambito finanziario.

3.2 Letterali

- Possono essere scritti in notazione
  - scientifica (es. `2E10`, `+1.2e-5`, `-444.33E12`)
  - decimale (es. `123.456`, `-77.0`, `144.33`)
- Di default sono di tipo `double`: per i letterali di tipo `float` si aggiunge il suffisso `f` o `F` (esiste anche il suffisso `d` o `D` per i `double`, ma non è necessario).
- Il compilatore controlla che il valore sia nel range ammesso.

3.3 Valori speciali

Lo standard IEEE 754 prevede 5 valori speciali:

- *infinito* positivo e negativo
- *zero* positivo e negativo
- `NaN`, *Not a Number* (che è particolare perché non è uguale a se stesso, `NaN != NaN`, quindi `espr == NaN` è sempre `false`)

```
double inf = 1.0 / 0.0;       // Infinity
double negInf = -1.0 / 0.0;   // -Infinity
double zero = 1.0 / inf;      // 0.0
double negZero = -1.0 / inf; // -0.0
double nan = 0.0 / 0.0;       // NaN
```

4 Conversioni implicite tra interi e floating point

È sempre possibile effettuare conversioni implicite secondo la gerarchia

$$\texttt{int} \rightarrow \texttt{long} \rightarrow \texttt{float} \rightarrow \texttt{double}$$

ma non al contrario.

Tali conversioni implicite vengono effettuate nelle espressioni aritmetiche e negli assegnamenti che coinvolgono questi tipi.

4.1 Perdita di precisione

Non tutti i valori `int` e `long` possono essere rappresentati in modo esatto dai tipi `float` e `double`. Di conseguenza, si può verificare una **perdita di precisione**.

Esempio:

```
int x = 2109876543; // x =  2109876543
float y = x;        // y = 2.10987648e9
int z = (int) y;    // z =  2109876480
```

5 Conversioni esplicite tra floating point e interi

$$\texttt{double} \rightarrow \texttt{float} \rightarrow \texttt{long} \rightarrow \texttt{int}$$

Si può verificare una perdita di informazioni.

Esempio:

```
double x, y;
int i;
byte b;

x = 127.3;
y = 2.7;

i = (int) x;           // i = 127
b = (byte) (x + y);    // b = -126
i = (int) (x + y);     // i = 130
i = (int) x + (int) y; // i = 129
```

6 Conversioni e divisione

Se si vuole eseguire una divisione tra due numeri interi e ottenere un risultato a virgola mobile, è necessario convertire almeno uno degli operandi a `double` o `float`: altrimenti, viene eseguita la divisione intera e solo in seguito il risultato viene convertito.

6.1 Esempio: calcolo della media

```
int x, y, z;
double media;

media = (x + y + z) / 3;
```

non è il valore che ci interessa, dato che viene eseguita la divisione intera (ad esempio, il risultato di `(1 + 1 + 2) / 3` è `1` invece di `1.33333...`).

Perché la divisione produca un risultato `double`, è necessario che almeno uno degli operandi sia di tipo `double` (l'altro, se non lo è, viene convertito implicitamente):

```
media = (x + y + z) / 3.0;
// oppure
media = (double) (x + y + z) / 3;
```

7 Conversioni implicite a `String`

- Se almeno un argomento dell'operatore `+` è un riferimento a `String`, `+` rappresenta la concatenazione di stringhe.
- Se uno degli operandi dell'operatore di concatenazione delle stringhe è di tipo primitivo, viene convertito a una stringa che rappresenta il suo valore prima di concatenarlo.
- Se uno degli operandi della concatenazione è un riferimento, viene ottenuta una stringa richiamando il metodo `toString` (presente in *ogni classe*, anche se non definito esplicitamente dal programmatore) dell'oggetto riferito.

7.1 Esempi

```
int i = 1;
double pi = 3.14;

out.print("La somma vale ");
out.println(i + pi);
// La somma vale 4.14

out.println("La somma vale " + i + pi);
// La somma vale 13.14

out.println("La somma vale " + (i + pi));
// La somma vale 4.14
```

```
String s1, s2;
Frazione f;

s1 = s2 + f; // equivale a: s1 = s2 + f.toString();
```