Unità didattica 16

Conversioni

CHE COSA IMPARERAI A FARE

- Convertire il tipo di un oggetto
- Convertire i vettori
- Conoscere il tipo di un oggetto

CHE COSA DOVRAI STUDIARE

- La conversione dei tipi
- Upcasting
- Downcasting
- La conversione di un vettore
- Le conversioni di null

Unità didattica 16 Conversioni

Ogni espressione scritta nel linguaggio di programmazione Java ha un *tipo* che può essere determinato analizzando i tipi dei metodi, delle variabili e delle espressioni letterali contenuti nell'espressione stessa, oltre che ovviamente la sua struttura. È possibile, a ogni modo, scrivere un'espressione in un contesto dove il tipo non è appropriato, per esempio, utilizzando valori `float` come parametri di una istruzione `switch`; in questo caso si ottiene un errore in fase di compilazione. In altri casi, il contesto potrebbe prendere in input il tipo relativo all'espressione: Java esegue allora una conversione implicita dal tipo dell'espressione a un tipo accettabile dal contesto in oggetto. Le conversioni sono state accennate nell'Unità didattica 15 relativamente alle primitive; nella presente Unità didattica verranno discussi questi meccanismi in relazioni a classi e sottoclassi.

16.1 Conversioni

Nel linguaggio Java esiste una serie di conversioni possibili, le cui principali sono:

- **allargamento delle primitive** (**primitive widening**): è il processo di conversione di una variabile più piccola in una più capiente; per esempio, un `int` può essere convertito in un `long` senza perdita di dati, perché quest'ultimo è più capiente (vedi Unità didattica 15);
- **restringimento delle primitive** (**primitive narrowing**): è il processo opposto, quando una variabile viene convertita in una meno capiente, come nella trasformazione di un `long` in un `int`; in questo caso è necessario esplicitare la conversione, perché può verificarsi una perdita di informazione (vedi Unità didattica 15);
- **allargamento dei riferimenti** (**reference widening**); vedi il paragrafo 16.2: *Upcasting*;
- **restringimento dei riferimenti** (**reference narrowing**); vedi il paragrafo 16.3: *Downcasting*;
- **conversioni di stringhe**: qualsiasi tipo di dato primitivo, incluso `null`, può essere convertito in una stringa, semplicemente utilizzando l'operatore di concatenamento (+); per esempio: `int a = 5; String s = "" + a;`

Le conversioni si applicano principalmente in due contesti: quando avviene un assegnamento e anche quando le espressioni vengono passate come parametri di metodo.

16.2 Upcasting

Quando si crea una sottoclasse di una classe base, si sta creando un nuovo tipo di dato. Se, per esempio, `A` deriva da `B`, si dice che `A` è "di tipo `B`"; allo stesso modo, se `Cane` deriva da `Animale`, si dice che `Cane` è "di tipo `Animale`".

Se una variabile è di tipo `Animale`, può contenere un riferimento anche a oggetti di tipo `Cane`:

```
public class Animale {
   //. . .
}
public class Cane extends Animale {
   //. . .
}
public static void main( String[] args ) {
   Animale a = new Cane();
}
```

Il fatto di poter ospitare un riferimento a un oggetto il cui tipo è derivato dal tipo della variabile è una caratteristica costruita nel linguaggio Java, ed è chiamata *upcasting*.

> L'**upcasting** è l'esecuzione di una conversione da una classe derivata a una delle possibili superclassi.

L'operazione di upcasting (chiamata, in generale e per brevità, *cast*) è sempre sicura, perché si passa sempre da un dato più specifico a uno più generale. Una sottoclasse è infatti l'insieme degli elementi della classe base più eventuali aggiunte; la sottoclasse può contenere dunque anche più metodi della classe base, ma contiene *come minimo* tutti i metodi della superclasse. L'unica cosa che può succedere all'interfaccia di un oggetto sottoposto a upcasting è di perdere dei metodi, mai di guadagnarne. Per questo motivo, l'operazione è supportata direttamente dal compilatore senza richiedere una sintassi specifica né generare errori. L'upcasting può essere esplicito, ma è assolutamente opzionale, non obbligatorio. Per esempio, è possibile esplicitare la conversione dell'oggetto `Cane` in un riferimento di tipo `Animale`:

```
public static void main( String[] args ) {
   Animale a = (Animale)new Cane();
}
```

Questa operazione, però, è assolutamente opzionale.

16.3 Downcasting

L'operazione inversa dell'upcasting è il *downcasting* che, invece, deve essere esplicitato dal programmatore e che consente di passare da un tipo `A` a una sua sottoclasse `C`.

> Il **downcasting** è l'operazione che permette di convertire un riferimento da una classe base a una classe più specifica.

Per esempio, un oggetto `Cane` contenuto in una variabile di tipo `Animale` (operazione consentita per via dell'upcasting automatico), può essere riconvertito nel tipo `Cane`, ma questa operazione deve essere esplicitata tramite un cast, indicando cioè il tipo a cui convertire, indicato tra parentesi tonde:

```
Animale a1 = new Cane();

Cane a2 = (Cane)a1;
```

Se si omette il downcasting (chiamato, in generale e per brevità, anch'esso, *cast*), il compilatore solleva un errore. Specificare il tipo tra parentesi è il modo che il programmatore ha a disposizione per dichiarare al compilatore: "so che i due tipi di dato non sono perfettamente compatibili, ma so anche che l'oggetto che voglio convertire è effettivamente del tipo di dato a cui voglio convertire".
Quando affronta un'operazione di downcasting, Java verifica che effettivamente l'oggetto da convertire sia del tipo richiesto, e in caso contrario solleva un'eccezione di tipo `ClassCastException`. Per esempio, la porzione di codice seguente crea un oggetto stringa e prova a eseguire un casting, che però fallisce, perché l'oggetto `o` contiene un oggetto che non è di tipo `Cane` (né di classe `Cane` né di una delle sue superclassi):

```
Object o = new String("Prova");

Cane c = (Cane)o;
```

osservazioni

I prefissi *up* e *down* utilizzati nelle parole *upcasting* e *downcasting* derivano storicamente dalla rappresentazione grafica dei diagrammi di classe, che visualizzano gerarchicamente un insieme di oggetti. In questi diagrammi si parte infatti solitamente dalla classe base e si scrivono, in colonna verso il basso, le varie classi derivate, in sequenza. Per questo motivo, *upcasting* si riferisce al fatto di risalire nella gerarchia, mentre *downcasting* allude alla discesa nella stessa.

16.4 Conversioni di vettori e null

Le conversioni disponibili nel linguaggio Java supportano anche due tipi particolari di oggetti: `null` e i vettori. È infatti possibile eseguire una serie di operazioni di conversione tra oggetti e `null`, tra vettore e vettore e, considerando che i vettori sono oggetti essi stessi, tra normali oggetti e vettore e viceversa.
L'upcasting è supportato per il tipo `null` e per qualsiasi vettore della classe `Object`. È dunque possibile convertire `null` in un oggetto di tipo `Object`:

```
Object o = null;
```

Di conseguenza, questo meccanismo consente a Java di assegnare il valore `null` a qualsiasi oggetto, visto che tutti derivano da `Object`; inoltre, è possibile assegnare a una variabile di tipo `Object` un qualsiasi tipo di vettore. In questa porzione di codice, per esempio, viene creato un vettore di interi di 10 elementi, che viene poi assegnato alla variabile `o` di tipo `Object`:

```
int[] c = new int[ 10 ];
Object o = c;
```

Il downcasting dei vettori funziona in modo simile: è infatti possibile convertire qualsiasi riferimento di tipo `Object` in un vettore.
Nell'esempio seguente, viene creato un vettore di interi lungo 10 elementi e subito associato a una variabile di tipo `Object` (eseguendo dunque un upcasting); nella riga di codice successiva, la variabile di tipo `Object` subisce il downcasting al tipo vettore `int[]`.
L'operazione si esegue senza errori perché l'oggetto è in effetti un vettore di interi:

```
Object o = new int[ 10 ];
int[] c = (int [])o;
```

Il casting funziona anche tra vettori di tipo diverso, a patto che l'operazione richiesta sia disponibile per le primitive che formano un vettore.
Per esempio, è possibile convertire un vettore di interi in un vettore di interi lunghi utilizzando l'upcasting in questo modo:

```
long[] al = new int[ 10  ];
```

Il downcasting avviene in modo simile al downcasting delle primitive, ma ovviamente il tipo indicato tra le parentesi tonde riporta le doppie parentesi quadre che identificano i vettori:

```
int[] a1 = (int [])new long[ 10 ];
```

Un meccanismo non diverso da quanto visto per i vettori delle primitive si applica ai vettori di oggetti: l'upcasting e il downcasting di questa tipologia di vettori è possibile se è possibile la stessa operazione di cast a livello dei semplici oggetti.
Si considerino, per esempio, le due classi `Animale` e `Cane`, dove `Cane` è una sottoclasse di `Animale`:

```
public class Animale {
   //. . .
}
public class Cane extends Animale {
   //. . .
}
```

Essendo `Animale` e `Cane` parte della medesima gerarchia, è possibile eseguire delle conversioni.
Per esempio, se si crea un vettore di oggetti `Cane`, è possibile eseguire un upcasting a un vettore di tipo `Animale`:

```
Animale[] a = new Cane[ 10 ];
```

Lo stesso vettore può essere riconvertito in un vettore di oggetti `Cane`, utilizzando il downcasting esplicito:

```
Cane[] b = (Cane [])a;
```

16.5 L'operatore instanceof

Per conoscere il tipo dell'oggetto contenuto in una variabile durante l'esecuzione del programma, è possibile utilizzare l'operatore `instanceof`, che indica se un determinato oggetto è di tipo specificato. Per esempio, per sapere se `a1` è di tipo `Cane`, si può scrivere:

```
Cane a1 = new Cane();

if( a1 instanceof Cane ) {
   //. . . a1 è un cane
}
```

Si noti che `instanceof` non indica solo se un oggetto è di un tipo specifico, ma anche se questo è di quel tipo in senso lato, come indicato nel paragrafo 16.2. Se, per esempio, `Cane` deriva da `Animale` e si esegue questo codice:

```
Cane a1 = new Cane();

if( a1 instanceof Animale ) {
   //. . . a1 è un cane
}
```

la condizione dell'istruzione `if` risulta verificata, perché `Cane` è anche di tipo `Animale`, in quanto `Cane` estende `Animale`.

Esercizi

Unità didattica 16

Esercizi di comprensione

1 Il ..... è ..... delle variabili primitive e permette di convertire una variabile più piccola in una più capiente.

2 Il ..... è il ..... delle variabili primitive e permette di convertire una variabile più grande in una meno capiente.

3 L'upcasting è il ..... dei riferimenti.

4 Il downcasting è il ..... dei riferimenti.

5 Se `B` deriva da `A`, `C` deriva da `B`, e `b = new B()`, quali delle seguenti istruzioni sono sintatticamente corrette?

a) `A a = b`
b) `B b1 = b`
c) `C c = b`
d) `C c = (A)b`
e) `A a = (A)b`

6 Se `X` deriva da `Y` e `x = new X()`, quali delle seguenti istruzioni sono sintatticamente corrette?

a) `X x1 = x`
c) `x = new Y()`
d) `Y y = x`
e) `X x1 = new Y()`

7 Quali delle seguenti istruzioni sono sintatticamente corrette?

a) `Object o = null`
b) `String s = null`
c) `Object o = new int[ 10 ]`
d) `String[] a = (String[])(new int[ 10 ])`
e) `long[] a = new int[ 10 ]`
f) `int[] f = (new int[ 10 ])`
g) `int[] f = (new String[ 10 ])`
h) `int[] a = (int [])new long[ 10 ]`

8 Se `B` deriva da `A`, `C` deriva da `B`, e `b = new B[ 5 ]`, quali delle seguenti istruzioni sono sintatticamente corrette?

a) `C c = (C [])b`
b) `C[] c = b`
c) `A[] a = b`
d) `B b1 = b`
e) `B b1 = b[ 0 ]`
f) `B[] b1 = b`
g) `B[] b1 = (B[])b`
h) `B[] b1 = b; A[] b1 = b`
i) `B[] b1 = b; A a = b1[ 0 ]`

Esercizi

Unità didattica 16

9 Se `B` deriva da `A`, `C` deriva da `B`, e `b = new B()`, quali delle seguenti espressioni restituiscono `true`?
a) `b instanceof B`
b) `b instanceof A`
c) `b instanceof C`
d) `b[] instanceof A`

Esercizi di applicazione

1 Crea un programma composto da:
a) classe `Soggetto` dotata dell'attributo descrizione di tipo `String` e metodo `getDescrizione()` che la restituisce;
b) classe `Cliente` che deriva da `Soggetto`.

2 Scrivi la classe `Main` dotata del metodo `stampa(Soggetto)` che stampa la descrizione del soggetto.

3 Modifica la classe `Main` e aggiungi il metodo `stampa(Cliente)` che stampa la descrizione del cliente.

4 Modifica la classe `Main` e aggiungi il metodo `tipo(Soggetto)` che stampa "S" se il parametro attuale è un soggetto oppure "C" se è un cliente.

5 Crea la classe `Fornitore` che deriva da `Soggetto` e modifica la classe `Main` per includere il metodo `stampa(Fornitore)`.

6 Modifica la classe `Main` e cambia il metodo `tipo(Soggetto)` per stampare anche "F" nel caso il parametro attuale sia `Fornitore`.

7 Modifica `Fornitore` per includere l'attributo `int punteggio`; modifica la classe `Main` in modo che `stampa()` nel caso di `Fornitore` produca in output oltre che la descrizione anche il punteggio.

8 Sposta la logica del metodo `tipo()` da `Main` alla gerarchia delle classi, creando un metodo `tipo()` in `Soggetto`, `Cliente` e `Fornitore`.

9 Sposta la logica del metodo `stampa()` da `Main` alla gerarchia delle classi, creando un metodo `stampa()` in `Soggetto`, `Cliente` e `Fornitore`.