Azzolini Riccardo 2018-11-16

`instanceof`, cast, classi astratte e interfacce

1 Operatore `instanceof`

```
espressione_riferimento instanceof tipo_riferimento
```

Questo operatore forma un'espressione di tipo `boolean`, il cui valore è:

- `true` se `espressione_riferimento` si riferisce a un'istanza di `tipo_riferimento` (o di un suo sottotipo, dato che le istanze di una classe sono anche istanze delle sue superclassi), altrimenti `false`
- sempre `false` se il valore di `espressione_riferimento` è `null`

1.1 Esempio

```
Rettangolo r;
// ...
if (r instanceof Quadrato)
    out.print("Quadrato: ");
else
    out.print("Rettangolo: ");
out.println(r.toString());
```

2 Cast di tipi riferimento

Non è possibile assegnare direttamente a una variabile di tipo della sottoclasse un riferimento di tipo della superclasse.

Ciò è invece possibile mediante un cast. In fase di esecuzione, però, si può verificare un errore se l'oggetto contenuto nel riferimento non è effettivamente un'istanza della sottoclasse. È quindi consigliabile eseguire prima un controllo con l'operatore `instanceof`.

2.1 Esempio

```
Quadrato q;
Rettangolo r;

// ...

q = r; // Non accettato dal compilatore

q = (Quadrato) r; // Accettato, ma possibile errore in esecuzione

if (r instanceof Quadrato)
    q = (Quadrato r); // Accettato, senza rischio di errori
```

3 La classe `Cerchio`

Le sue istanze rappresentano cerchi.

3.1 Costruttori

```
public Cerchio(double r);
```

3.2 Metodi

```
public double getRaggio();
public double getCirconferenza();
public double getArea();
public double getPerimetro();
public boolean equals(Cerchio c);
public boolean haAreaMaggiore(Cerchio c);
public boolean haPerimetroMaggiore(Cerchio c);
public string toString(); // es. "raggio = 3.1"
```

Molti di questi metodi sono presenti anche nella classe `Rettangolo`. Sarebbe quindi opportuno definire sia `Rettangolo` che `Cerchio` come sottoclassi di un'unica superclasse che contenga i metodo in comune. Si ha però un problema: non è possibile fornire un'implementazione per alcuni di questi metodi nella superclasse, dato che i calcoli da effettuare dipendono dal tipo di figura.

4 Classe astratta

Una **classe astratta** è una classe che può contenere **metodi astratti** (`abstract`), cioè metodi per i quali specifica il prototipo ma non l'implementazione.

Una classe non astratta si dice **concreta**.

Una classe astratta *non può essere istanziata*, ma può essere estesa: ciascuna sottoclasse deve implementare tutti i metodi astratti, oppure deve essere a sua volta astratta. Come quelle concrete, le classi astratte forniscono un supertipo comune i cui valori sono tutte le possibili istanze di tutte le sottoclassi.

Una classe astratta può contenere anche metodi implementati, ma al contrario una classe concreta *non* può contenere metodi astratti. In altre parole, una classe che contiene uno o più metodi astratti *deve* essere dichiarata astratta.

5 La classe astratta `Figura`

È superclasse di `Rettangolo` e `Cerchio`.

5.1 Metodi

```
public abstract double getArea();
public abstract double getPerimetro();
public boolean haAreaMaggiore(Figura f);
public boolean haPerimetroMaggiore(Figura f);
```

Poiché tutte le figure hanno un'area (`getArea`) e un perimetro (`getPerimetro`), che però si calcolano in modi completamente diversi, `Figura` definisce tali metodi in modo astratto, delegandone l'implementazione alle sottoclassi.

`haAreaMaggiore` e `haPerimetroMaggiore` possono invece essere definiti concretamente, sfruttando i due metodi astratti per ricavare le informazioni necessarie al confronto. In questo modo, nelle sottoclassi è sufficiente implementare `getArea` e `getPerimetro` per avere a disposizione anche `haAreaMaggiore` e `haPerimetroMaggiore`.

6 Interfaccia

Un'**interfaccia** (intesa come costrutto del linguaggio Java) è una parte di codice che specifica dei comportamenti senza fornirne le implementazioni. In particolare, specifica solo prototipi e contratti di metodi (che sono quindi astratti) o costanti.

Una classe può **implementare** *un numero qualsiasi* di interfacce. Per fare ciò, deve:

- dichiarare quali interfacce implementa nell'intestazione
- fornire le implementazioni di tutti i metodi specificati da tali interfacce, a meno che la classe non sia astratta

A ogni interfaccia corrisponde un tipo riferimento, il quale è supertipo di tutte le classi che la implementano. Si applicano le stesse regole di promozione e cast che valgono per le classi. Infine, non è possibile costruire istanze di un'interfaccia.

7 L'interfaccia `Comparable<T>`

Definisce un ordine totale sugli oggetti che la implementano. A tale scopo, specifica un unico metodo che consente di confrontare l'oggetto che esegue il metodo con altri oggetti di tipo `T`:

```
public int compareTo(T o);
```

Tale metodo restituisce:

- un intero negativo se l'oggetto che esegue il metodo è minore di quello specificato come argomento
- zero se i due oggetti sono uguali
- un intero positivo se l'oggetto che esegue il metodo è maggiore dell'argomento

7.1 Esempio di applicazione: `SequenzaOrdinata<E>`

Un esempio di applicazione dell'interfaccia `Comparable` è la classe generica `SequenzaOrdinata<E>`: essa richiede infatti che il *tipo argomento* implementi tale interfaccia.

Ad esempio, `SequenzaOrdinata<Frazione>` viene accettato dal compilatore perché `Frazione` implementa `Comparable<Frazione>`, mentre `SequenzaOrdinata<Rettangolo>` provoca un errore in fase di compilazione dato che `Rettangolo` non implementa `Comparable<Rettangolo>`.

8 L'interfaccia `Iterable<E>`

Ogni classe che la implementa rappresenta una collezione di dati che può essere **iterata** (scandita) un elemento alla volta, mediante un oggetto detto **iteratore**. Quest'ultimo è in sostanza un elenco degli elementi presenti nella collezione.

`Iterable<E>` prevede un solo metodo:

```
public Iterator<E> iterator();
```

Esso restituisce un riferimento a un iteratore degli oggetti presenti nella collezione.

`Iterator<E>` è a sua volta un'interfaccia che specifica due metodi:

```
public boolean hasNext();
public E next();
```

- `hasNext` restituisce `true` se l'iteratore contiene elementi, altrimenti `false`.
- `next` restituisce il prossimo elemento dell'iteratore e lo elimina da quest'ultimo (ma non dalla collezione originale). Se l'iteratore è vuoto, si verifica un errore in fase di esecuzione.

8.1 Scansione di `Iterable`

Per scandire gli elementi di una classe che implementa `Iterable` si possono usare vari tipi di ciclo, tra cui in particolare il ciclo *for-each*.

```
Iterator<E> iter = elenco.iterator();
while (iter.hasNext()) {
    E elemento = iter.next();
    // ... uso di elemento ...
}

for (Iterator<E> iter = elenco.iterator; iter.hasNext(); ) {
    E elemento = iter.next();
    // ... uso di elemento ...
}

for (E elemento : elenco) {
    // ... uso di elemento ...
}
```