Azzolini Riccardo 2021-04-08

Classi

1 Introduzione

Scala è un linguaggio a oggetti, in cui la principale astrazione sui dati è la nozione di classe. In seguito, il meccanismo classi in Scala verrà illustrato facendo riferimento a un classico esempio: i numeri razionali.

Si vogliono modellare i numeri razionali, definendo un "pacchetto" che fornisca l'aritmetica su tali numeri. In generale, un numero razionale $\frac{x}{y}$ è una coppia di numeri interi: il numeratore x e il denominatore y. Per modellare $\frac{x}{y}$ in modo elegante, è necessario definire una struttura dati che combini numeratore e denominatore e permetta di trattarli come un'unica entità, definendo funzioni sull'intera struttura e non sui due numeri interi che essa contiene.

2 Classe `Rational`

Per iniziare, si definisce la classe `Rational`, specificando una lista di parametri formali chiamati **parametri di classe** (ciascuno dei quali è caratterizzato, come al solito, da un nome e un tipo); in questo caso, i parametri di classe sono il numeratore `x` e il denominatore `y`, entrambi di tipo `int`:

```
class Rational(x: Int, y: Int)
```

Questo è già un frammento di codice completo: siccome (per ora) la classe non contiene codice (ha solo i comportamenti ereditati dalla superclasse di default), si può evitare di indicare le parentesi graffe che racchiuderebbero il corpo (vuoto).

La definizione appena mostrata introduce due elementi:

- un nuovo **tipo** di nome `Rational`;
- un **costruttore primario**, chiamato anch'esso `Rational`,[1] che permette di creare elementi del tipo `Rational`, e prende come argomenti i parametri di classe.

[1] Siccome in Scala i nomi dei tipi e dei valori appartengono a spazi dei nomi distinti, non c'è conflitto tra le definizioni di `Rational` come nome del tipo e come costruttore: a partire dal contesto, è sempre possibile determinare a quale delle definizioni faccia riferimento un uso del nome `Rational`.

Il valore di un oggetto è determinato dai valori dei parametri di classe passati come argomenti al costruttore primario. Si vedrà più avanti che ciò consente l'uso delle classi nel modello di sostituzione.

3 Metodo `toString`

Un primo metodo che è opportuno definire, o meglio ridefinire, è il metodo `toString` che tutte le classi ereditano da `java.lang.Object` (si può immaginare che `Object` sia la superclasse di tutto, anche se la gerarchia delle classi Scala è più complessa). Esso viene utilizzato ogni volta che è necessario produrre una stringa per descrivere un oggetto, in particolare, è il metodo usato per descrivere il risultato della valutazione di un'espressione nell'interprete.

Non avendo esplicitamente definito un metodo `toString`, la classe `Rational` eredita appunto quello di `Object`, che restituisce stringhe del tipo `Rational@653a0c41`. Se si vuole ottenere una rappresentazione più utile, bisogna sovrascrivere il metodo `toString`:

```
class Rational(x: Int, y: Int) {
  override def toString(): String =
    if (y == 1) x.toString() else x + "/" + y
}
```

Su questo codice possono essere fatte varie osservazioni:

- I metodi vengono definiti esattamente come le funzioni, con la differenza che sono scritti nel corpo di una classe e possono avere dei modificatori prima della parola riservata `def`.
- In Scala, quando si sovrascrive un metodo è obbligatorio dichiarare esplicitamente la ridefinizione mediante il modificatore `override`, altrimenti si ha un errore in compilazione. In questo modo, il compilatore aiuta il programmatore a non sovrascrivere accidentalmente un metodo esistente quando ne vuole invece definire uno nuovo, e viceversa.
- È possibile usare `x.toString()`, dove `x` è di tipo `Int`, perché in Scala i valori di tutti i tipi, compresi i tipi base, sono oggetti sui quali è definito il metodo `toString` — a differenza di Java, non c'è la distinzione tra tipi primitivi e tipi riferimento (almeno al livello della macchina astratta, che è quello che interessa al programmatore; a un livello più basso, il compilatore Scala può usare i tipi primitivi per migliorare l'efficienza del codice).
- Nel ramo `else` non è invece necessario effettuare esplicitamente le conversioni `x.toString()` e `y.toString()`, perché queste vengono fatte implicitamente dall'operatore di concatenazione di stringhe (`+`), secondo le stesse regole che valgono in Java (se almeno un operando di `+` è una stringa, allora anche l'altro operando viene convertito in una stringa).

- Solitamente, per motivi di documentazione, si indicano esplicitamente i tipi restituiti dai metodi anche quando il compilatore è in grado di dedurli (la deduzione del tipo restituito da un metodo è possibile in base alle stesse regole che valgono per le funzioni).
- Siccome il metodo `toString` non ha argomenti, si possono omettere le parentesi vuote sia nella definizione che nell'invocazione:

```
class Rational(x: Int, y: Int) {
  override def toString: String =
    if (y == 1) x.toString else x + "/" + y
}
```

4 Metodo `add`

Adesso si vuole implementare il metodo `add`, che calcola la somma tra due numeri razionali — quello su cui il metodo viene invocato e un altro passato come argomento. Un primo tentativo di implementazione potrebbe essere il seguente:

```
class Rational(x: Int, y: Int) {
  override def toString: String =
    if (y == 1) x.toString else x + "/" + y

  def add(that: Rational): Rational =
    new Rational(x * that.y + that.x * y, y * that.y)
}
```

Tuttavia, questo codice non è corretto, dà degli errori in compilazione, perché *i parametri di classe non sono campi*: la loro visibilità è limitata all'*oggetto* (non alla classe) in cui sono definiti, dunque non è possibile far riferimento ai parametri di classe di un altro oggetto tramite un riferimento a tale oggetto. In questo caso, ad esempio, i riferimenti `x` e `y` ai parametri di classe locali all'oggetto che esegue il metodo sono corretti, mentre non lo sono i riferimenti `that.x` e `that.y` relativi all'oggetto `that`.

Per poter implementare correttamente `add`, è necessario associare esplicitamente i valori dei parametri di classe a dei **campi**, cioè dei nomi (`val` o `def`) che siano accessibili dall'esterno dell'oggetto:

```
class Rational(x: Int, y: Int) {
  val num = x
  val den = y

  // ...
}
```

Così, gli oggetti della classe `Rational` hanno due campi: `num` e `den`. È però importante sottolineare che i campi in Scala sono appunto nomi, e non variabili come in Java. In generale, a un campo di una classe può essere associato un valore qualunque, ma è molto comune associarvi il valore di un parametro di classe; perciò, come si vedrà più avanti, Scala mette a disposizione i *campi parametrici*, una sintassi compatta per definire insieme un parametro di classe e un corrispondente campo.

Se i campi sono definiti con `val`, le espressioni che ne determinano i valori vengono valutate al momento della creazione dell'oggetto. Infatti, le espressioni e le definizioni inserite nel corpo della classe costituiscono di fatto il codice del costruttore primario.

Un oggetto può essere creato invocandone un costruttore (quello primario o un altro, come si vedrà in seguito), tramite un'*espressione di creazione* che, come in Java, è formata da:

1. l'operatore `new`;
2. il nome del costruttore (che è uguale al nome della classe);
3. i parametri del costruttore, che nel caso del costruttore primario sono i parametri di classe.

Un'espressione di creazione può essere letta come un'invocazione di una funzione che restituisce una nuova istanza di una classe. Anche selezione dei **membri** (campi e metodi) di un oggetto avviene come in Java, utilizzando la **dot-notation**. Ad esempio:

```
val r = new Rational(1, 2)
r.num       // 1
r.den       // 2
r.toString  // 1/2
```

Per accedere ai membri dell'oggetto corrente è sufficiente scriverne il nome, ma per rendere più esplicito l'accesso si può opzionalmente usare la **self-reference** `this`, una pseudo-variabile che fa riferimento all'oggetto attualmente in esecuzione.

Usando i meccanismi appena presentati, si può definire correttamente il metodo `add` (e, volendo, si potrebbero definire in modo simile anche le altre operazioni sui numeri razionali: `sub`, `mul`, `div`, ecc.):

```
class Rational(x: Int, y: Int) {
  val num = x
  val den = y

  override def toString: String =
    if (den == 1) num.toString else num + "/" + den

  def add(that: Rational): Rational = new Rational(
    this.num * that.den + that.num * this.den,
    this.den * that.den
```

```
  )
}
```

Alcune osservazioni su questo codice sono le seguenti:

- Nel metodo `add`, l'unico modo per ottenere al numeratore e al denominatore dell'oggetto `that` è utilizzare i campi di tale oggetto. Invece, per il numeratore e il denominatore dell'oggetto corrente si potrebbero utilizzare direttamente i parametri di classe `x` e `y`, ma utilizzare anche in questo caso i campi (`this.num` e `this.den`) è più elegante, uniforme. Per lo stesso motivo, i campi vengono adesso usati anche nel metodo `toString`, che in precedenza era definito usando i parametri di classe.
- La visibilità di default dei membri, in assenza di modificatori, è *pubblica* (mentre in Java l'assenza di modificatori indica la visibilità limitata al package). Si vedrà più avanti che Scala prevede una granularità di accesso più raffinata di quella di Java.
- Lo stato degli oggetti è **immutable**: il valore di un oggetto è determinato al momento della creazione, in base ai parametri attuali di classe passati al costruttore primario, dopodiché non può cambiare, rimane invariato fino alla distruzione dell'oggetto.

Un esempio di uso del nuovo metodo `add` è

```
val r = new Rational(1, 2)
val s = new Rational(2, 3)
r.add(s)
```

dove la valutazione dell'espressione `r.add(s)` dà il risultato `7/6` (qui mostrato nel formato prodotto da `toString`, che è quello utilizzato per la visualizzazione nell'interprete).

5 Semplificazioni

Il modo in cui i numeri razionali sono stati rappresentati finora non è particolarmente "furba", perché i valori non sono semplificati, il che introduce principalmente due problemi:

- Nell'interprete, i risultati delle operazioni non sono mostrati nella forma semplificata che ci si aspetterebbe. Ad esempio, il risultato dell'espressione

  ```
  new Rational(1, 2).add(new Rational(1, 2))
  ```

  viene visualizzato come `4/4` anziché `1`.
- Lo stesso numero razionale è rappresentato da oggetti `Rational` con valori diversi. Ad esempio, `new Rational(1, 3)` e `new Rational(2, 6)` rappresentano entrambi il numero razionale $\frac{1}{3}$.

L'unica soluzione che permetta di risolvere entrambi questi problemi, soprattutto considerando che lo stato degli oggetti è immutable, è eseguire la semplificazione al momento della costruzione:

1. si definiscono come **membri privati** della classe `Rational` (cioè accessibili solo dal codice definito all'interno di questa classe, analogamente a Java) un metodo `gcd` per il calcolo del massimo comune divisore e un metodo `abs` per il calcolo del valore assoluto[2] (questi sono uguali alle funzioni `gcd` e `abs` viste in precedenza — se le si avesse già a disposizione all'esterno della classe,[3] la si potrebbe tranquillamente usare, senza bisogno di definirle nuovamente come metodi);
2. si calcolano i valori semplificati dei campi `num` e `den` dividendo i parametri `x` e `y` per il loro MCD.

```
import scala.annotation.tailrec

class Rational(x: Int, y: Int) {
  val num = x / abs(gcd(x, y))
  val den = y / abs(gcd(x, y))

  private def abs(n: Int): Int = if (n < 0) -n else n

  @tailrec
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  override def toString: String =
    if (den == 1) num.toString else num + "/" + den

  def add(that: Rational): Rational = new Rational(
    this.num * that.den + that.num * this.den,
    this.den * that.den
  )
}
```

Si noti che, siccome i campi sono definiti tramite `val`, le loro definizioni vengono valutate una volta sola al momento della costruzione dell'oggetto. Se invece li si definisse con `def`,

```
def num = x / abs(gcd(x, y))
def den = y / abs(gcd(x, y))
```

[2] Senza l'uso del valore assoluto, in alcuni casi il processo di semplificazione scambierebbe i segni di numeratore e denominatore.

[3] Come detto in precedenza, la funzione `abs` esiste nella libreria standard di Scala, ma qui per semplicità si preferisce evitare di importare funzioni di libreria.

ma si ripeterebbe la valutazione di `gcd(x, y)` a ogni valutazione dei campi, il che è inutile (e inefficiente) perché i valori di `x` e `y` non possono cambiare, quindi il risultato della valutazione è sempre lo stesso.

Le implementazioni dei metodi `toString` e `add` non cambiano, dato che la semplificazione è gestita interamente nel costruttore primario, ma adesso è importante, soprattutto in `toString`, usare i campi `num` e `den` invece dei parametri di classe `x` e `y` anche per l'oggetto corrente, poiché i valori dei parametri di classe rimangono non semplificati (essendo immutable): solo i campi contengono i valori semplificati.

Con questa nuova versione della classe `Rational`, l'espressione

```
new Rational(1, 2).add(new Rational(1, 2))
```

dà il risultato desiderato, `1`, e non più `4/4`.

6 Valori non validi

Al momento, il costruttore `Rational` accetta qualunque valore come denominatore, compreso zero, dunque è possibile costruire oggetti che non sono ammessi nello spazio dei razionali, cioè non rappresentano numeri razionali validi:

```
val notARational = new Rational(2, 0)
```

Questo è particolarmente problematico perché ci sono operazioni che, applicate a un valore non valido, producono comunque un valore valido, quindi può essere difficile rendersi conto dell'errore. Ad esempio, se si implementasse l'operazione di divisione,

```
class Rational(x: Int, y: Int) {
  // ...
  def div(that: Rational): Rational =
    new Rational(this.num * that.den, this.den * that.num)
  // ...
}
```

la valutazione dell'espressione

```
new Rational(3, 5).div(notARational)
```

produrrebbe il risultato `0` (`0/1`), che è un numero razionale valido.

Per definire il tipo `Rational` in modo corrente, la scelta migliore è prevenire completamente la costruzione dei valori non ammissibili per tale tipo, in modo da sfruttare uno dei vantaggi dell'astrazione fornita dagli oggetti: la possibilità di garantire che i dati incapsulati all'interno di un oggetto siano consistenti per tutta la vita dell'oggetto.

In Java, per prevenire la costruzione di valori non validi si usano le eccezioni. Anche Scala sfrutta le eccezioni, ma fornisce un meccanismo un po' più elegante per sfruttarle: un costruttore o un metodo può definire delle **precondizioni** che esprimono vincoli (*constraint*) sui valori passati a esso come argomenti; tali vincoli devono essere soddisfatti dal chiamante, altrimenti viene sollevata un'eccezione. Le precondizioni vengono specificate mediante l'invocazione del metodo predefinito `require`, che prende come argomenti un'espressione booleana e, opzionalmente, un messaggio da fornire all'utente in caso di violazione della precondizione.

Nel caso di `Rational`, si vuole imporre una precondizione sul valore specificato per il parametro di classe `y`, il denominatore, al momento della costruzione dell'oggetto. Di conseguenza, il metodo `require` viene invocato all'interno del costruttore primario, cioè l'espressione di invocazione viene scritto direttamente nel corpo della classe:

```
class Rational(x: Int, y: Int) {
  require(y > 0, "denominator must be positive")

  val num = x / abs(gcd(x, y))
  val den = y / abs(gcd(x, y))

  private def abs(n: Int): Int = if (n < 0) -n else n

  @tailrec
  private def gcd(a: Int, b: Int): Int =
    if (b == 0) a else gcd(b, a % b)

  override def toString: String =
    if (den == 1) num.toString else num + "/" + den

  def add(that: Rational): Rational = new Rational(
    this.num * that.den + that.num * this.den,
    this.den * that.den
  )
}
```

Così, se la precondizione `y > 0` non è soddisfatta, come ad esempio nell'espressione di creazione `new Rational(2, 0)`, il metodo `require` interrompe la creazione dell'oggetto sollevando un'`IllegalArgumentException`:

```
java.lang.IllegalArgumentException: requirement failed: denominator must
    be positive
  at scala.Predef$.require(Predef.scala:338)
  ...
```

7 Costruttori ausiliari

Oltre al costruttore primario, una classe può avere altri costruttori, chiamati **costruttori ausiliari**. Essi sono introdotti tramite la sintassi `def this(...)`, cioè usando la solita sintassi di definizione dei metodi, ma con `this` come identificatore speciale al posto di un nome arbitrario (a differenza di Java, non si usa il nome della classe) e sempre senza l'indicazione di un tipo restituito.

Il corpo di un costruttore ausiliario, come quello di un normale metodo, può essere costituito da un'espressione semplice o da un blocco di espressioni e definizioni, ma in ogni caso la prima (eventualmente unica) espressione *deve essere l'invocazione di un altro costruttore della stessa classe*, indicata mediante la sintassi `this(...)`. Il costruttore invocato può essere direttamente il costruttore primario, oppure può essere un altro costruttore ausiliario *già definito* nel codice della classe (la definizione del costruttore invocato deve essere scritta prima di quella del costruttore chiamante nel codice sorgente della classe): in questo modo, qualunque catena di invocazioni di costruttori termina con il costruttore primario della classe (che poi l'unico a invocare un costruttore della superclasse). Si vedrà prossimamente che questa regola, non presente in Java,[4] è necessaria al fine di definire il valore di un oggetto nella semantica basata sul modello di sostituzione.

Come esempio, nella classe `Rational` è utile definire un costruttore ausiliario che permetta di specificare solo il numeratore quando si vuole costruire un oggetto che rappresenta un numero intero. Tale costruttore non fa altro che richiamare il costruttore primario, fissando il denominatore a `1` e specificando il numeratore passato come argomento:

```
class Rational(x: Int, y: Int) {
  require(y > 0, "denominator must be positive")

  val num = x / abs(gcd(x, y))
  val den = y / abs(gcd(x, y))

  def this(x: Int) = this(x, 1)

  // ...
}
```

Così, l'espressione di creazione `new Rational(2, 1)` può essere scritta equivalentemente come `new Rational(2)`.

[4] Si potrebbe dire che Java ammette "più costruttori primari", perché ciascun costruttore di una classe può invocare direttamente un costruttore della superclasse (in modo implicito o esplicito), senza bisogno di "passare" obbligatoriamente da un altro costruttore della classe corrente.