Azzolini Riccardo 2021-04-19

Overriding e singleton object

1 Overriding

Si consideri l'esempio della classe astratta `IntSet`, con le due sottoclassi concrete che ne realizzano un'implementazione (basata su un BST):

```
abstract class IntSet {
  def add(x: Int): IntSet

  def contains(x: Int): Boolean
}

class Empty extends IntSet {
  def contains(x: Int): Boolean = false

  def add(x: Int): IntSet = new NonEmpty(x, new Empty(), new Empty())
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true

  def add(x: Int): IntSet =
    if (x < elem) new NonEmpty(elem, left add x, right)
    else if (x > elem) new NonEmpty(elem, left, right add x)
    else this
}
```

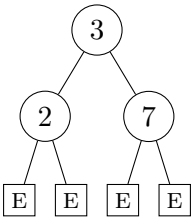
Le definizioni di `contains` e `add` delle sottoclassi `Empty` e `NonEmpty` *implementano* i metodi astratti dichiarati nella superclasse `IntSet`. Se invece si implementa in metodo che è *già definito* in una classe base, allora si parla di **overriding**, **sovrascrittura**. Come già detto, Scala richiede di dichiarare esplicitamente la sovrascrittura dei metodi usando il modificatore `override`. Si può mettere `override` anche quando si implementa un metodo astratto della superclasse, ma non è obbligatorio.

1.1 Esempio: `toString`

Per poter visualizzare nell'interprete i risultati delle operazioni sui BST, è `Empty` e `NonEmpty` devono sovrascrivere l'implementazione di default del metodo `toString`. Un modo comune di rappresentare come stringhe gli alberi binari etichettati è il seguente:

- l'albero vuoto viene rappresentato con un punto, `"."`;
- un albero non vuoto viene rappresentato da una stringa della forma `"(LxR)"`, dove x è l'etichetta del nodo radice, mentre L e R sono le stringhe che rappresentano, rispettivamente, i sottoalberi sinistro e destro.

Ad esempio, l'albero



è rappresentato dalla stringa `"((.2.)3(.7.))"`.

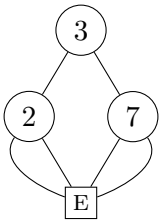
Per ottenere tale rappresentazione, le classi `Empty` e `NonEmpty` sovrascrivono il metodo `toString` con le seguenti implementazioni:

```
class Empty extends IntSet {
  // ...
  override def toString: String = "."
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  // ...
  override def toString: String = "(" + left + elem + right + ")"
}
```

2 Definizione di singleton object

Siccome la classe `Empty` non ha parametri e non fornisce operazioni di modifica, tutte le sue istanze sono identiche, dunque non vi è la necessità di avere istanze diverse. Allora, si potrebbero risparmiare il tempo e la memoria necessari a costruire le istanze di `Empty` riutilizzando sempre un'unica istanza, ottenendo così alberi con una struttura del genere:



In Java, si può "forzare" la creazione di un'unica istanza di una classe usando un design pattern: il singleton. Scala internalizza tale pattern nel linguaggio tramite le **object definition**: usando la parola riservata `object` invece di `class` si definisce **singleton object**, cioè un oggetto che è l'unica istanza della sua classe ed è associato a un nome uguale a quello della classe.

Ad esempio, la definizione

```
object Empty extends IntSet {
  def contains(x: Int): Boolean = false

  def add(x: Int): IntSet = new NonEmpty(x, Empty, Empty)

  override def toString: String = "."
}
```

crea un singleton object di nome `Empty`. Più nel dettaglio, il nome `Empty` fa riferimento all'unico oggetto di tipo `Empty`, che viene automaticamente costruito dal compilatore, mentre non è possibile costruire esplicitamente istanze di `Empty` (usando l'operatore `new`).

Un singleton object è un valore, che nel modello di sostituzione viene rappresentato dal suo nome. Di conseguenza, tale nome (ad esempio `Empty`) viene valutato in se stesso, è "già valutato".

2.1 Applicazioni stand-alone

Quando si realizza un'applicazione in Java, il punto di ingresso (da cui inizia l'esecuzione del codice) è il *metodo statico* `main`. Anche se i metodi statici sono scritti all'interno di una classe, essi non sono associati ad alcuna istanza, dunque la creazione del primo "vero" oggetto di un programma orientato agli oggetti spetta al programmatore.

Invece, Scala adotta un approccio orientato agli oggetti più puro: il `main` è un metodo "tradizionale", non statico ma appartenente a un singleton object che viene istanziato automaticamente dal compilatore (come tutti i singleton object). Quello che invece rimane uguale a Java è il tipo del metodo `main`: esso deve avere un singolo parametro di tipo array di stringhe (gli argomenti passati al programma dalla riga di comando):

```
def main(args: Array[String]) = /* ... */
```

Ad esempio, il punto d'ingresso di un'applicazione Scala potrebbe essere definito da un file con i seguenti contenuti:

```
package lib.intset

object Main {
  def main(args: Array[String]) = {
    println("Some IntSets")
    val set1 = Empty.add(1)
    println(set1)
    // ...
  }
}
```

Una volta compilata l'applicazione, per eseguirla da riga di comando è necessario usare il comando `scala` (analogo a `java`), passando come argomento il nome completo dell'oggetto che contiene il metodo `main`:

```
scala lib.intset.Main
```

3 Operazione di unione

Un'altra operazione sugli insiemi di interi che è utile implementare per fare alcune importanti considerazioni è l'unione. A tale scopo, si dichiara nella classe astratta `IntSet` un nuovo metodo `union`,

```
abstract class IntSet {
  def add(x: Int): IntSet
  def contains(x: Int): Boolean
  def union(other: IntSet): IntSet
}
```

che viene implementato nelle sottoclassi in questo modo:

```
object Empty extends IntSet {
  override def union(other: IntSet): IntSet = other

  // ...
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
  override def union(other: IntSet): IntSet =
    ((left union right) union other).add(elem)
```

```
  // ...
}
```

- Il risultato dell'unione dell'insieme vuoto `Empty` con un qualunque altro insieme è semplicemente l'altro insieme.
- Per fare l'unione di un insieme non vuoto (istanza di `NonEmpty`) con un altro insieme bisogna in qualche modo combinare gli elementi dei due alberi. La soluzione più semplice è costruire un nuovo albero, partendo dall'albero vuoto e aggiungendo uno a uno gli elementi dei due alberi da unire. Allora, l'implementazione di `union` in `NonEmpty` funziona in questo modo:

  1. invoca ricorsivamente `union` per ottenere un albero contenente tutti gli elementi dei due sottoalberi del nodo corrente (`left` e `right`) e dell'altro albero con cui si sta facendo l'unione (`other`);
  2. utilizza il metodo `add` (implementato in precedenza) per aggiungere a questo albero l'etichetta del nodo corrente, che è l'ultimo elemento mancante per completare l'unione.

  Così, grazie alle chiamate ricorsive effettuate al passo 1, l'invocazione di `add` al passo 2 avviene per ogni elemento dei due alberi di cui si sta facendo l'unione, dunque l'insieme risultante contiene tutti gli elementi di entrambi gli alberi.

Le considerazioni da fare sul metodo `union` riguardano la **correttezza** e la **terminazione** dell'implementazione.

- *Correttezza*: Secondo la definizione dell'unione tra insiemi, l'insieme costruito da `union` deve contenere *tutti e soli* gli elementi di `this` e `other`. Nel caso `Empty`, ciò è immediatamente verificato perché `this` non contiene elementi e viene restituito `other`. Invece, nel caso `NonEmpty` la correttezza può essere verificata informalmente osservando che:
  - come già detto, il risultato è dato dall'unione dei sottoalberi sinistro e destro di `this` con l'albero `other`, alla quale viene aggiunta l'etichetta della radice di `this`, ovvero complessivamente vengono inclusi tutti gli elementi di `this` e di `other`;
  - nessun altro elemento viene aggiunto.
- *Terminazione*: La terminazione del metodo `union` è garantita perché il primo argomento delle invocazioni ricorsive è ogni volta un insieme strettamente più piccolo di `this`:
  - in `left union right`, `left` è più piccolo di `this` in quanto suo sottoalbero sinistro;
  - in `(left union right) union other`, l'insieme `left union right` è più piccolo di `this` perché contiene un elemento in meno, l'etichetta della radice.

Di conseguenza, siccome gli alberi hanno dimensioni arbitrarie ma finite, dopo un numero finito di invocazioni le chiamate ricorsive hanno come primo argomento `Empty`, cioè si raggiunge sicuramente il caso base della ricorsione.

4 Dynamic binding

In Scala, come in Java, la selezione della segnatura dei metodi da invocare (cioè la risoluzione dell'overloading) avviene in *fase di compilazione*, mentre la selezione degli specifici metodi da eseguire avviene in *fase di esecuzione*:

1. Quando il compilatore trova un'invocazione di un metodo $e.m(e_1, \ldots, e_n)$, considera tutte le segnature del metodo m disponibili (definite o ereditate) nella classe corrispondente al *tipo* dell'espressione e, e in base ai *tipi* degli argomenti $e_1, \ldots, e_n$ sceglie la segnatura "più adeguata" (che richiede meno promozioni dei tipi degli argomenti).
2. Quando poi viene eseguita l'invocazione $e.m(e_1, \ldots, e_n)$, si cerca nella gerarchia delle classi un metodo avente la segnatura stabilita in fase di compilazione, ma la ricerca avviene a partire dalla classe di cui è istanza il *valore* dell'espressione e, che potrebbe essere una sottoclasse di quella corrispondente al *tipo* di e noto in fase di compilazione. Si parla allora di **dynamic binding**: il collegamento tra il nome (o meglio la segnatura) di un metodo e il codice da eseguire è *dinamico*.

4.1 Dynamic binding nel modello di sostituzione

Nel modello di sostituzione, il dynamic binding corrisponde alla regola che determina come cercare il codice del corpo da sostituire all'invocazione di un metodo: come nel modello di esecuzione concreto di Java/Scala, la ricerca avviene a partire dalla classe di appartenenza dell'oggetto su cui il metodo è invocato.

Ad esempio, date le classi definizioni

```
abstract class IntSet {
  def contains(x: Int): Boolean
  // ...
}

class Empty extends IntSet {
  def contains(x: Int): Boolean = false
  // ...
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
```

```
  def contains(x: Int): Boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  // ...
}

val set: IntSet = /* ... */
```

la valutazione dell'espressione `set contains 7`, ovvero `set.contains(7)`, dipende dal valore di `set` in fase di esecuzione:

- se `val set = Empty`, allora l'invocazione di `contains` viene sostituita con il corpo definito nella classe (singleton object) `Empty`:

  `set.contains(7)`
  $\rightarrow$ `Empty.contains(7)`
  $\rightarrow$ `[7/x][][Empty/this]false`
  $\rightarrow$ `false`

- se invece `val set = new NonEmpty(7, Empty, Empty)`, allora si sostituisce l'invocazione di `contains` con il corpo definito nella classe `NonEmpty`:

  `set.contains(7)`
  $\rightarrow$ `new NonEmpty(7, Empty, Empty).contains(7)`
  $\rightarrow$ `[7/x][7/elem, Empty/left, Empty/right]`
  `[new NonEmpty(7, Empty, Empty)/this]`
  `if (x < elem) left contains x`
  `else if (x > elem) right contains x`
  `else true`
  $\rightarrow$ `if (7 < 7) Empty contains 7`
  `else if (7 > 7) Empty contains 7`
  `else true`
  $\rightarrow\!\!\!\!\rightarrow$ `true`

Sempre come nel modello di esecuzione, se la classe dell'oggetto su cui un metodo è invocato non definisse tale metodo, allora lo si cercherebbe nella superclasse, e così via. Si noti che, come tutti i meccanismi nel modello di sostituzione, anche questo è di natura sintattica, perché la superclasse di ogni classe è indicata nella sintassi del codice (anche la superclasse implicita `java.lang.Object` è di fatto indicata esplicitamente dall'assenza della parola riservata `extends`).