Azzolini Riccardo 2021-04-26

Trait e gerarchia

1 Trait

Come Java, Scala è un linguaggio a **ereditarietà singola** (*single inheritance language*): una classe può avere solo una superclasse. Tramite le interfacce (`interface`), Java permette di simulare l'ereditarietà multipla *sui tipi*, cioè definire un tipo come sottotipo diretto di più supertipi, ma non permette di definire una classe che eredita codice da più superclassi dirette.[1] Scala fornisce invece il meccanismo dei **trait**, che consentono di simulare l'ereditarietà multipla sia sui tipi che sulle classi:

- si può definire un tipo come sottotipo diretto di più supertipi;
- si può ereditare codice da più supertipi diretti.

Un trait è dichiarato in modo simile a una classe astratta, ma usando la parola riservata `trait` invece di `abstract class`. La differenza sostanziale tra trait e classi astratte è che i trait *non hanno costruttori, non ammettono parametri* (questa restrizione è proprio ciò che "tiene in piedi" il meccanismo di ricerca dei metodi a runtime, evitando i problemi di ambiguità che tipicamente si hanno con l'ereditarietà multipla sulle classi).

Il seguente esempio mostra la dichiarazione di un trait `Planar`, che modella una superficie piana:[2]

```
trait Planar {
  def height: Int
  def width: Int
  def surface = height * width
}
```

Esso dichiara due metodi (senza parametri, ovvero campi) astratti, `height` e `width`, e un metodo concreto, `surface`.

[1] A partire da Java 8, le interfacce possono contenere metodi concreti (indicati con la parola riservata `default`), che vengono ereditati dalle classi che implementano l'interfaccia, permettendo così una forma limitata di ereditarietà multipla.

[2] Questa definizione non modella correttamente le superfici piane di tutte le possibili forme, ma serve solo come semplice esempio di trait.

Una classe, un object o un trait può ereditare direttamente da *al più una classe*[3] e da *un numero arbitrario di trait*. La classe o il primo trait da cui si eredita viene indicato con la parola riservata `extends`, mentre ciascuno degli eventuali altri trait va indicato con la parola riservata `with`. Ad esempio:

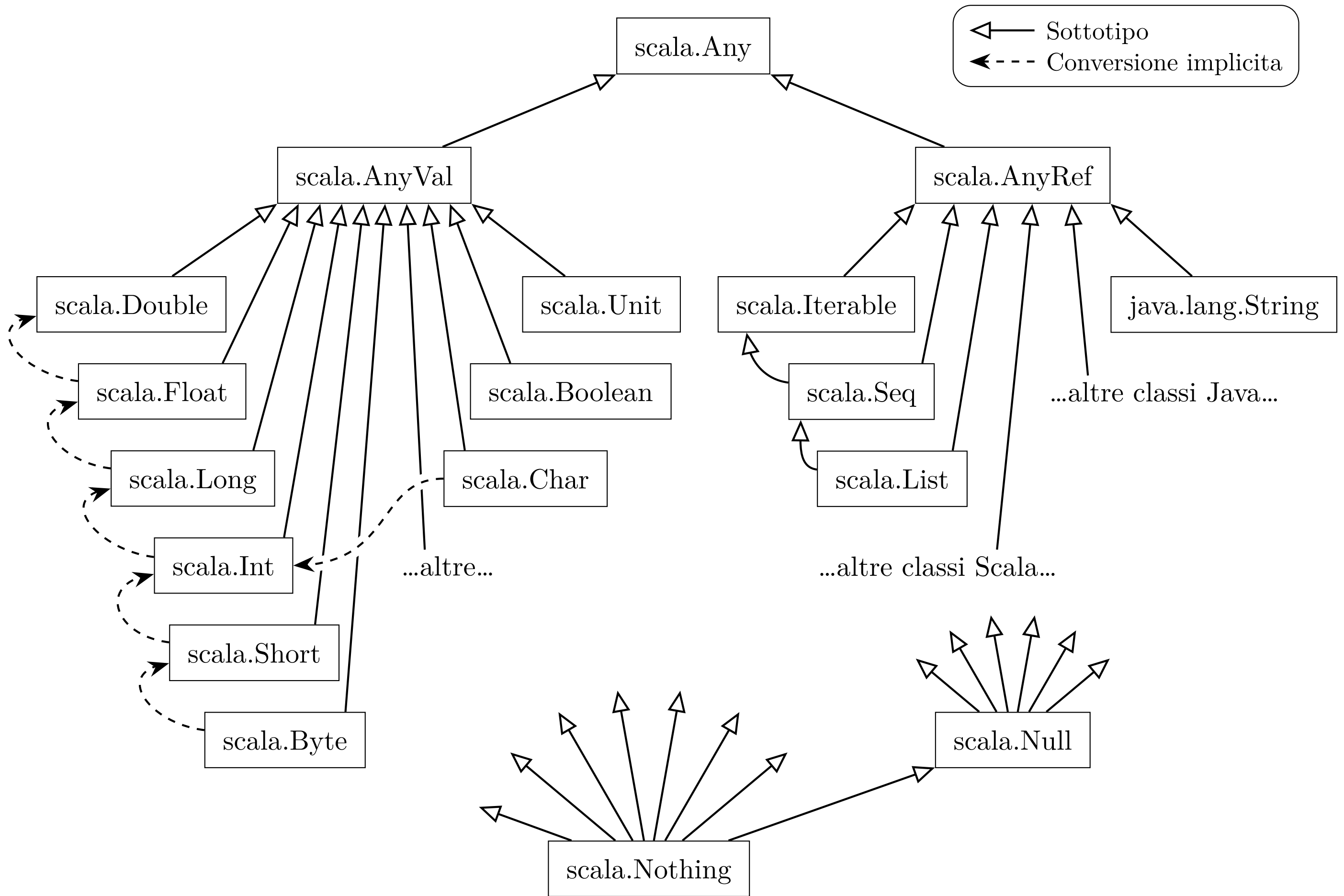
```
class Square extends Shape with Planar with Movable
```

(dove `Shape` potrebbe essere una classe o un trait, mentre `Planar` e `Movable` sono dei trait).

2 Gerarchia delle classi

La gerarchia delle classi di Scala, che comprende classi, object e trait, ha la seguente struttura:

[3] Un trait può estendere una classe, ma in tal caso si applicano alcune regole particolari sulle classi che ereditano a loro volta da tale trait, altrimenti sarebbe possibile usare questo meccanismo per ereditare da classi multiple (tramite trait che le estendono).



Come si può osservare, tale gerarchia è più complessa rispetto a quella di Java. Le parti interessanti sono, in particolare, i *top types* e i *bottom types*, cioè rispettivamente i tipi (le classi) in cima alla gerarchia e quelli in fondo (non presenti in Java).

Nel grafo della gerarchia appena mostrato sono indicate anche le *conversioni implicite* sui tipi base, che rispecchiano quelle sui tipi primitivi di Java. Esse *non sono relazioni supertipo/sottotipo*: quando un'istanza di un sottotipo viene interpretata come un'istanza di un supertipo l'oggetto rimane invariato, mentre le conversioni implicite tra i tipi base modificano gli oggetti a cui si applicano. Ad esempio, ogni valore del tipo `Byte` è concettualmente un valore ammesso anche per `Short`, dunque si potrebbe in un certo senso dire che `Byte` è un sottotipo di `Short`, ma per interpretare un valore di `Byte` come valore di `Short` è necessario trasformare la sua rappresentazione su un byte in una rappresentazione su due byte.

2.1 Top types

La radice della gerarchia è la classe `scala.Any`, che è il tipo base, il supertipo di tutti i tipi. Essa definisce dei metodi *universali*, che vengono ereditati da tutti gli oggetti, tra cui:

```
def equals(that: Any): Boolean
final def ==(that: Any): Boolean
final def !=(that: Any): Boolean

def hashCode: Int
final def ##: Int

def toString: String
```

- I metodi `equals`, `hashCode` e `toString` sono analoghi a quelli definiti in Java da `java.lang.Object` (ma questi ultimi sono disponibili solo per le istanze dei tipi riferimento, e non per i tipi primitivi, in quanto essi non sono oggetti, dunque non hanno metodi). Per questi metodi, `Any` fornisce delle implementazioni di default, che però è spesso opportuno sovrascrivere perché nella maggior parte dei casi non realizzano i comportamenti desiderati.
- Gli operatori `==` e `!=` invocano il metodo equals: l'espressione `x == y` corrisponde a `x.equals(y)`, e `x != y` corrisponde a `!(x.equals(y))`. Essi sono dichiarati con il modificatore `final`, che come in Java impedisce di sovrascriverli: per modificarne il comportamento, bisogna invece sovrascrivere il metodo `equals`.
- Il metodo `##` corrisponde in genere ad `hashCode`, ma nel caso dei tipi base ha un comportamento leggermente diverso, necessario per fare sì che i valori restituiti siano coerenti con il metodo `equals`.

`Any` ha due sottoclassi immediate:

- `scala.AnyVal`, il supertipo di tutti i tipi base (tranne `String`);
- `scala.AnyRef`, alias di `java.lang.Object`, la classe base di tutti i tipi riferimento, ovvero:
  - le classi Java, compresa `String`;
  - tutte le classi di Scala che non sono tipi base, comprese le classi definite dal programmatore.

2.2 Bottom types

A differenza di quella di Java, la gerarchia di Scala comprende delle classi *bottom*, `scala.Nothing` e `scala.Null`, che sono sottoclassi di tutte le classi presenti nella gerarchia o in una parte della gerarchia. Esse sono importanti in ambito funzionale poiché servono per poter dare un tipo a tutte le espressioni.

- `Nothing` è un tipo che *non ha valori* ed è *sottotipo di ogni tipo*.[4] Esso ha alcune applicazioni importanti, che verranno riprese in seguito, ma in sintesi esso viene principalmente usato:
  - per segnalare una *terminazione anomala* (cioè il sollevamento di un'eccezione);
  - come *element type* delle collezioni vuote.
- `Null` è il tipo del valore `null`, ed è *sottotipo di ogni tipo riferimento*. Infatti, ogni tipo riferimento (sottotipo di `AnyRef`) ammette `null` come valore; ad esempio:

  ```
  val x: String = null
  ```

  Come in Java, l'invocazione di un metodo su `null` dà luogo a un errore in fase di esecuzione.

  Invece, `null` non è un valore ammissibile per i sottotipi di `AnyVal`,

  ```
  val y: Int = null // Errore in compilazione
  ```

  poiché nel modello di esecuzione essi vengono implementati tramite i tipi primitivi di Java, e una variabile di un tipo primitivo corrisponde a una locazione di memoria che contiene sempre un valore, anche se magari non significativo (ad esempio se la variabile non è ancora stata inizializzata), mentre una variabile di un tipo riferimento può non fare riferimento ad alcun oggetto, il che corrisponde appunto al valore `null`.

3 Eccezioni

La gestione delle eccezioni in Scala è simile a quella in Java. Per sollevare un'eccezione si usa la clausola `throw`, ad esempio:

```
throw new RuntimeException()
```

[4] Il fatto che `Nothing` sia un sottotipo di ogni tipo è possibile proprio perché non esistono valori di tipo `Nothing`: per definizione, un tipo *T* è sottotipo di *U* se ogni valore di *T* è anche un valore di *U*, ma se *T* è `Nothing` non ci sono valori da considerare, dunque la definizione è vuotamente verificata a prescindere da quale sia il tipo *U*.

Tale clausola è un'*espressione* (non un'istruzione) di tipo `Nothing`, che forza la terminazione della valutazione del metodo in cui compare, e del suo chiamante, e così via, finché non viene eventualmente intercettata.

La clausola `throw` ha il tipo `Nothing` perché la valutazione di tale espressione termina sempre (a causa dell'eccezione sollevata) prima che venga prodotto un valore, cioè in sostanza l'espressione non produce alcun valore, e infatti `Nothing` è il tipo che non ha valori. Dato che `Nothing` è sottotipo di tutti i tipi, una clausola `throw` è ammessa ovunque sia prevista un'espressione, a prescindere dal tipo che l'espressione prevista dovrebbe avere.

4 Tipi delle espressioni in presenza della gerarchia

Per determinare il tipo di un'espressione, il compilatore Scala individua il più piccolo tipo al quale appartengono tutti i valori che l'espressione può assumere. In particolare, se l'espressione può assumere valori di tipi diversi il compilatore seleziona il supertipo comune minimo, la cui esistenza è garantita in ogni caso dal fatto che la gerarchia ha un'unica radice: se non ci sono supertipi più specifici per i valori di un'espressione il compilatore può semplicemente selezionare `Any`, il supertipo comune a tutti i tipi.

Ad esempio, se si definisce il nome

```
val x = null
```

senza indicarne esplicitamente il tipo, il compilatore assegna al nome il tipo dell'espressione `null`, che è `Null`. In questo caso, determinare il tipo dell'espressione è immediato perché essa può assumere un solo valore, `null`, ovvero può assumere solo valori di tipo `Null`.

Un caso più interessante è invece la seguente espressione:

```
if (true) 1 else false
```

Essa può assumere due valori: `1`, di tipo `Int`, e `false`, di tipo `Boolean`. Il tipo dell'espressione è dunque il più piccolo supertipo comune a `Int` e `Boolean`, cioè il loro antenato comune più vicino nella gerarchia, che è `AnyVal`. Si noti che il valore della condizione dell'`if-else` è irrilevante per la determinazione del tipo in fase di compilazione: siccome in generale il compilatore non può sapere a priori se una condizione sarà vera o falsa, il tipo di un'espressione condizionale è sempre il supertipo comune minimo delle espressioni nei rami `if` e `else`.

Un altro esempio simile è

```
if (true) 1 else "pippo"
```

che ha tipo `Any` perché i valori presenti nei due rami hanno rispettivamente i tipi `Int` e `String`. Infine, l'espressione

```
if (true) "pippo" else throw new RuntimeException()
```

ha tipo `String`: il tipo del ramo `if` è `String`, mentre il tipo del ramo `else` è `Nothing`, che è sottotipo di `String`, dunque il supertipo comune minimo è ancora `String`.