2023-12-04 - Abstract ed Override, Linearization Algorithm, Companion Objects, Apply, Unapply

Abstract ed Override

Sia le classi che i traits possono definire dei **membri astratti**: campi (**fields**), metodi (**methods**) e tipi (**types**). Prima che un'**istanza** possa essere creata, questi membri devono venir **definiti** *(sovrascritti, keyword opzionale `override`)*.

```
trait Animal {
    // tutti i membri di questo trait sono astratti
    val name: String
    type Food
    def eat(food: Food): Unit
}

abstract class AbstractDog extends Animal {
    override val name = "cane" // keyword override opzionale
    type Food
    def eat(food: Food): Unit
}

class Dog extends AbstractDog {
    // keyword override opzionale
    override type Food = String
    override def eat(food: Food) = println(s"$name is eating $food")
}
```

Linearization Algorithm

L'algoritmo di **linearization** viene utilizzato per stabilire il **tipo** di un oggetto, scegliendo l'ordine in cui **eredita traits e classi**. Vengono effettuati i seguenti **passaggi** su una lista:

- viene impostato come **primo elemento** della lista il **tipo concreto** dell'oggetto
- calcolare la linearization di **ogni tipo** esteso dall'oggetto, **da destra a sinistra**, aggiungendo i risultati alla lista *(append)*
- rimuovere i **duplicati**, **da sinistra a destra** *(lasciando quello più a destra)*
- **aggiungere** alla fine della lista `ScalaObject`, `AnyRef`, `Any`

Esempio:

```
class C1 { ... }
trait T1 extends C1 { ... }
trait T2 extends C1 { ... }
trait T3 extends C1 { ... }
class C2A extends T2 { ... }
class C2 extends C2A with T1 with T2 with T3 { ... }
```

Linearization di C2:

- tipo concreto: `C2`
- linearization T3: `C2, T3, C1`
- linearization T2: `C2, T3, C1, T2, C1`
- linearization T1: `C2, T3, C1, T2, C1, T1, C1`
- linearization C2A: `C2, T3, C1, T2, C1, T1, C1, C2A, T2, C1`
- duplicati: `C2, T3, T1, C2A, T2, C1`
- aggiungere: `C2, T3, T1, C2A, T2, C1, ScalaObject, AnyRef, Any`

Companion Objects: `apply` e `unapply`

Una classe (`class`, `type` o `trait`) ed un oggetto (`object`) nello stesso package e con lo stesso nome vengono chiamati **companion class** e **companion object**. L'oggetto ha accesso ai **membri privati**

della classe.

*Non avvengono collisioni sul nome dato che classe viene salvata nel namespace dei tipi (**type namespace**) mentre l'oggetto nel namespace dei termini (**term namespace**)*

Vengono utilizzati per definire **metodi statici** associati alla classe *(dato che non esistono metodi statici nelle classi)*.

```
class MyClass(val value: Int)

// companion object
object MyClass {
  def createInstance(value: Int): MyClass = new MyClass(value)
}
```

Comunemente contengono **factory methods** per il cast e le **conversioni** *(in entrambi i versi, sia verso la classe che dalla classe ad altri tipi)*. Due metodi speciali contenuti nel companion object sono:

- `apply`: **costruisce** un oggetto della classe partendo dai **parametri** passati al metodo *(quindi una conversione alla classe)*
- `unapply`: **decostruisce** un oggetto della classe, spesso utilizzato per effettuare **pattern matching** *(quindi una conversione dalla classe)*

Esempio: Pair

`+T`: **covarianza**, tutti i supertipi di `T` (`? super T` in Java)
`-T`: **controvarianza**, tutti i sottotipi di `T` (`? extends T` in Java)

```
type Pair[+A, +B] = Tuple2[A, B]

object Pair {
    def apply[A, B](x: A, y: B) = Tuple2(x, y)
    def unapply[A, B](x: Pair[A, B]): Tuple2[A, B] = x
}
```

Il tipo `Pair` *(classe)*, che è generico sui tipi `A` e `B` (e tutti i loro supertipi), **non ha costruttore**, ma la creazione è affidata all'`apply` del suo companion object.

```
val pair = Pair(1, 2) // viene chiamato apply

pair match {
    case Tuple2(a, b) => println(a, b) // viene chiamato unapply
    case _ => println("error")
}
```

Esempio: collezioni

È possibile utilizzare un companion object anche quando il **numero di parametri** alla costruzione (e decostruzione) possono **essere variabili**, attraverso l'utilizzo del variatico in costruzione (`String*`) e del metodo `unapplySeq` in decostruzione.

```
class L

object L {
    def apply(stuff: String*) = stuff.mkString(" ")
    def unapplySeq(s: String): List[String] = s.split(",").toList
}

// apply
val l = L("ciao", "mondo") // "ciao mondo"
// unapplySeq, effettuato destructuring
val L(a, b) = l // a = "ciao", b = "mondo"
```