Scala un tutorial pratico

Introduzione

A grande richiesta (una :D) parliamo di campi.

Campi e metodi di accesso.

L'accesso, in lettura o scrittura, ad un campo, pubblico privato o protetto che sia, è sempre mediato da un metodo. E' un punto importante per l'interoperabilità Scala → Java, cioè per l'uso di codice Scala da parte di programmi Java.

```
classe Classe {

	var x = 10
}
```

La compilazione di questo codice produce una classe con due metodi pubblici, un getter e un setter:

```
//javap Classe
public class Classe extends java.lang.Object implements scala.ScalaObject{
    public int x();
    public void x_$eq(int);
    public Classe();
}
```

I metodi di accesso generati hanno una forma predefinita il cui equivalente, sempre in Scala, è:

```
class Classe {

	private var variabile = 10

	def x: Int = { variabile }

	def x_=(v: Int): Unit = { variabile = v }
}
```

Qui equivalenza significa che dire:

```
class Classe {

	var x = 10
}
```

o dire:

```
classe Classe {

	def x: Int = qualcosa //getter

	def x_=(v: Int) = qualcosa //setter
}
```

E' indifferente per il codice che “usa” il campo x. In entrambi i casi posso scrivere, da qualche altra parte:

```
val istanza = new Classe
istanza.x = 33 //setter
val valore = istanza.x //getter
```

Il fatto che esista questa equivalenza comporta che il codice che dichiara un campo è sempre **trasformabile** nella corrispondente coppia di metodi di lettura e scrittura. Trasformabile nel senso

che si può cambiare il codice sorgente della classe che dichiara il campo pubblico, rinominarlo (e probabilmente renderlo privato) e inserire un getter e un setter nella forma vista, senza che questo comporti alcuna variazione nel codice utente, cioè nel codice che accede in lettura o scrittura al campo. Questa è la situazione:

```
//libreria
class Vector3i {
  var x = 0
  var y = 0
  var z = 0
}

//codice utente
object Main {

	def main(args: Array[String]) = {
		val v = new Vector3i
		v.x = 1
		v.y = 2
		v.z = 3
		v.x = 33
		v.y = v.x + v.z
		System.out.println(v.x + "," + v.y + "," + v.z)
	}
}
```

A un certo punto scopro che, per una qualche ragione, il valore del campo x del mio Vector3i non è più un semplice intero ma il prodotto di un qualche calcolo. Io posso rimpiazzare il campo con i suoi metodi di accesso, all'interno dei quali fare i miei conti in tre passaggi:

1. rinomino il campo

```
//libreria
class Vector3i {

  private var xField = 0

  var y = 0
  var z = 0
}
```

2. introduco il metodo di lettura

```
//libreria
class Vector3i {

  private var xField = 0

  def x = math.sqrt(xField).intValue //un conto a caso

  var y = 0
  var z = 0
}
```

3. introduco il metodo di scrittura

```
//libreria
class Vector3i {

  private var xField = 0

  def x = math.sqrt(xField).intValue //lettura def x: Int = { expr }

  def x_=(valore: Int) = xField = valore //scrittura, def x_=(v: Int): Unit = { expr }

  var y = 0
  var z = 0
}
```

Dal punto di vista di Main, cioè del codice utente, non è cambiato nulla. Non cambia nulla perché

quando prima Main accedeva ad x già lo faceva attraverso i metodi x e x_=, scritti automaticamente dal compilatore.
Ovviamente c'è la complicazione, non poteva essere tutto rose e fiori. In scala non è possibile, per una ragione che mi sfugge, ridefinire un campo in una sottoclasse.

```
class Classe {

	var x = 10
}

class SottoClasse extends Classe {

	override var x = 20 //errore
}
```

E tiene conto anche dii metodi di accesso, impliciti o espliciti.

```
class Classe {

	var x = 10
}

class SottoClasse extends Classe {

	override def x: Int = { 20 } //errore, ridefinizione di campo variabile

	override def x_=(v: int): Unit = { qualcosa } //errore, ridefinizione di campo variabile
}
```

Significa che l'unica specializzazione ammessa per un campo è la ridefinizione del suo valore, cioè posso assegnargli un valore diverso.
Se voglio ridefinire un campo (ridefinibile, quindi pubblico o protetto) in una sottoclasse, DEVO cambiare il codice sorgente della superclasse, cioè devo passare da:

```
class Classe {
	var x = 10
}

class SottoClasse extends Classe {

	private var vx = 10

	override def x = vx //errore

	override def x_=(v: Int) { vx = v * 2 } //errore
}
```

a:

```
class Classe {
	private var campox = 10

	def x = campox

	def x_=(v: Int) = campox = v
}

class SottoClasse extends Classe {

	private var vx = 10

	override def x = vx //adesso è ok

	override def x_=(v: Int) { vx = v * 2 } //adesso è ok
}
```

In pratica non posso sovrascrivere i metodi di accesso dichiarati implicitamente dal compilatore ma posso sovrascrivere gli stessi metodi di accesso se sono dichiarati esplicitamente. Notate che il codice utente accede sempre alla proprietà con la notazione campo:

```
//codice utente
object Main {

        def main(args: Array[String]) = {
          var c = new Classe
          System.out.println(c.x)
          c.x = 33
          System.out.println(c.x)
          c = new SottoClasse
          System.out.println(c.x)
          c.x = 44
          System.out.println(c.x)
        }
}
```

Perché questa cosa strana? Be', secondo me è un bug ma non scarterei l'ipotesi che sia una caratteristica introdotta per via di una super intelligente proprietà della programmazione funzionale. Più probabilmente una ragione c'è e io non la conosco. Resta la questione: senza avere accesso al codice sorgente, non è possibile ridefinire il comportamento di una variabile membro ereditata. Quindi se dichiaro un campo, come “scrittore” del codice so che avrò sempre la possibilità di ridefinire o specializzare l'accesso a quel campo, senza dover intervenire sulle altre parti del programma che usano quel campo. Ma se voglio che l'utente del mio codice possa ridefinire il mio campo, senza costringerlo a modificare il mio codice sorgente, allora devo dichiarare esplicitamente i metodi di accesso. In entrambi i casi, non è necessaria alcuna modifica al codice sorgente che accede, in lettura o scrittura, al campo.