Class Extensions

via

Implicit Conversion

Class Extension

- È possibile aggiungere metodi "al volo" ad una classe o a qualche istanza, senza usare subclassing?

```
class String
  def print_self
    puts self
  end
end

"Mario Rossi".print_self
```

Ruby

- **aggiunge un nuovo metodo alla classe String** (quella predefinita)
- il nuovo metodo è **disponibile a tutte le istanze di String** in questo scope

Conversione implicita di tipo

converte un valore di un tipo **T1** in un valore di tipo **T2**

```
implicit  def m(x:T1):T2 =  new T2(…x…)
```

abilita il compilatore ad usare m automaticamente:

se un contesto ***si aspetta un valore di tipo T2, posso passare un valore di tipo T1***
e il compilatore farà la conversione

```
implicit def str2int(str:String):Int = Integer.parseInt(str)
def addTwo(a:Int, b:Int) = a+b
addTwo("123", 450)
```

*…ispirato dai **costruttori come convertitori di tipo in C++***

Class Extension

```
class MyString(str:String){
  def printSelf()={
    println(str)
  }
}
```

una classe che fa da wrapper per String

```
implicit def str2mystring(str:String) = new MyString(str)

"Mario Rossi".printSelf()
```

- il compilatore vede che String non ha il metodo richiesto
- si accorge che ***siamo nello scope di un convertitore*** di tipo per String

Class Extension

```
class String
  def print_self
    puts self
  end
end

"Mario Rossi".print_self
```

Ruby

Static Type Checks

Scala

```
class MyString(str:String){
  def printSelf()={
    println(str)
  }
}
implicit def str2mystring(str:String) = new MyString(str)
"Mario Rossi".printSelf()
```


Class Extension e DSL

```
import java.util._

class DateHelper(num:Int) {
  def days (when:String) :Date = {
    var date=Calendar.getInstance()
    when match {
      case "ago" => date.add(Calendar.DAY_OF_MONTH, -num)
      case "form_now" => date.add(Calendar.DAY_OF_MONTH, num)
    } }
}

object DateHelper {
  val ago = "ago"
  val from_now = "from_now"
  implicit def int2datehelp(num:Int)= new DateHelper(num)
}
```

mette converter e "keywords" nello scope

```
import DateHelper._

val past = 2 days ago

val meeting = 5 days from_now

println(past)              // Mon Jan 5 2015
println(meeting)           // Mon Jan 12 2015
```

il feeling è un vero e proprio DSL

Ereditarietà

Composizione

Combinatori

Ereditarietà

```
class Cerchio (val raggio:Int) extends Shape {
  override def toString = "cerchio di raggio "+raggio
}
```

- ***obbligatorio*** quando si ridefinisce un membro concreto di superclasse
- ***opzionale*** se si implementa un membro astratto
- ***vietato*** se non si fa nessun overriding/implementazione

Overriding

- ✓ **C++**
  - si indicava con `virtual` un metodo ridefinibile in una sottoclasse
  - per ***efficienza***: per i metodi non virtuali c' e` static dispatch
- ✓ **Java**
  - ogni metodo e` implicitamente virtual
  - c'è ***l'annotazione*** `@Override`, ma è opzionale
- ✓ **Scala**
  - si indica con `override` una ridefinizione in una sottoclasse

- aiuta il **"fragile base class problem"**:

  se si aggiungono nuovi membri ad una superclasse, si rischia di corrompere il codice client provocando degli overriding accidentali (magari ***difficili da individuare***). Ora, se il codice cliente conteneva già quei membri, non compila piu', o meglio, il compilatore segnala gli overriding "accidentali"

- aiuta nella mixins-composition

Ereditarietà

```
class Cat {
 val dangerous = false
}

class Tiger ( override val dangerous:Boolean,
              private var age: Int
            ) extends Cat

val t=new Tiger(true,3)
```

```
class Felino(val dangerous:Boolean) { }

class Pantera (danger:Boolean) extends Felino(danger) { }
```

invoca il costruttore di superclasse

Esempio: Factory Object

Un **factory object** contiene metodi per costruire altri oggetti.

- il codice cliente usa i factory methods e non costruisce direttamente oggetti con **`new`**
- cosi` la costruzione di oggetti è centralizzata e i dettagli sulla loro rappresentazione sono completamente nascosti.
- il codice cliente deve capire meno dettagli ed e' robusto a cambiamenti di implementazione degli oggetti che usa

- *dove definire i factory methods?*
  - In una classe o in un singleton object?
  - …nel ***companion object*** della classe!

Factory Object

Elem=0

elemento del layout 2D:
rettangolo con righe di testo

ArrayElem

LineElem

```
abstract class Elem {
  def contenuto: Array[String]
  def altezza :Int = contenuto.lenght
  def larghezza:Int = if (altezza==0) 0 else contenuto(0).lenght
  override def toString= contenuto mkString "\n"
}

object Elem {

  class ArrayElem( val contenuto:Array[String] ) extends Elem

  class LineElem(s:String) extends Elem {
    val contenuto = Array(s)
    override def altezza = 1
    override def larghezza = s.lenght
  }

  def elem(cont:Array[String]):Elem = new ArrayElem(cont)
  def elem(line:String):Elem = new LineElem(line)
}
```

classi private

Traits

combinazione di

Java's interfaces e Ruby's mixins

Trait

In prima approssimazione:

- come una ***classe astratta***...

  ...contiene membri astratti e membri concreti

- come un'***interfaccia***...

  ...non si istanzia e una classe può *estendere* più trait

  implementare

  mix-in

Traits

...come classe astratta...

mix di membri astratti e concreti,

```
trait Libro {
  def titolo:String
  def titolo_=(n:String):Unit
  def calcolaPrezzo = titolo.length * 10
}
```

```
class MyLibro (var titolo:String) extends AnyRef with Libro
```

costruisco una classe con un trait

```
class MyLibro (var titolo:String) extends Libro

val l=new MyLibro("Guerra e Pace")
```

implementa i metodi astratti con un campo dati!

Traits

```
trait Filosofo {
  def filofeggia() = {
    println("Consumo memoria,dunque esisto")
  }
}
```

tutti membri concreti

un trait non si istanzia

```
val f = new Filosofo
error: trait Filosofo is abstract;
       cannot be instantiated

class Rana extends Filosofo

val rana=new Rana
rana.filosofeggia()
```

un trait **definisce un tipo**:

```
val fil:Filosofo = rana
fil.filosofeggia()
```

Traits

```
trait Filosofo {
  def filofeggia() = {
    println("Consumo memoria,dunque esisto")
  }
}
```

```
class Animale
```

estende **una classe**
e *mix-in* un trait

```
class Rana extends Animale with Filosofo {
  override def toString = "verde"
}
```

```
trait Canterino
```

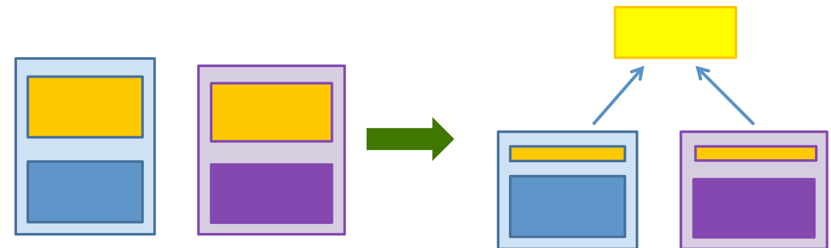
estende **1 classe**
e *mix-in* 2 traits

```
class Grillo extends Animale with Filosofo with Canterino {
  override def toString = "grillo parlante"
  override def filosofeggia()={ println("pensa prima di fare")  }
}
```

ridefinisce il metodo ereditato dal trait

Uso dei traits

1. **Per fattorizzare codice comune**
   - con *interfacce* non è possibile: non contengono codice
   - con *classi astratte* si può ma c' è ereditarietà singola



Uso dei traits

2. **Design di Interfacce: piccole o ricche??**

se l'interfaccia ha **molti metodi**,

- è più facile che il client trovi il metodo con la funzionalità che si adatta precisamente al suo caso
- ma più lunga da implementare

se l'interfaccia ha **pochi metodi**

- è più facile da implementare,
- ma se non c'è un metodo che si adatta perfettamente al caso del client, il client dovrà aggiungere codice per usare quel metodo

Uso dei trait come rich interfaces

- Per definire un concetto di "<" su un tipo.
  - va definito il senso di "<" ma poi è comodo avere anche <=, >,…

…todi di confronto stanno in interfaccia, …ementati tutti quanti ogni volta!
…cia ha solo il metodo "<", basta …re quello, ma ogni altro confronto …riscritto a mano

ridefinisco operatori!!
in Scala non ci sono vincoli sulla sintassi dei nomi di metodi

0 se sono uguali, negativo se `this` è minore di `that`

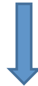
```
trait …ered[A] {
 def …mpare(that:A):Int
 def < (that:A):Boolean = if (this.compare(that)<0)true else false
 def <= (that:A):Boolean = if ...
 def > (that:A):Boolean = if ...
 def >= (that:A):Boolean = if ...
 def compareTo(that:A):Int = this.compare(that)
}
```

Uso dei traits

- **Design di Interfacce piccole o ricche??**

Con i trait:

- si definisce un trait con
  - un piccolo numero di metodi astratti
  - e un ricco insieme di metodi concreti ***definiti in termini dei metodi astratti***



la classe che mix-in il trait
- dovrà implementare solo pochi metodi
- e ne avrà a disposizione molti!

Trait

In prima approssimazione:

- come una
- come un'

un trait è <u>una modifica</u>

(aggiunte e/o ridefinizioni di metodi)

applicabile ad una classe

ti

it

implementare

mix-in

Trait

```
class A { def n = ... }
  trait C { def m1 = ... }
  trait D { def m2 = ... }
class B extends A with C
class B2 extends A with D
class B3 extends A with C with D
```

```
val x=new B
x.m1 ; x.n
val y = new B2
val z = new B3
z.m2; z.m1; z.n
```

un trait è

una **modifica** ***incrementale***

riutilizzabile

deve esserci una classe
a cui fare la modifica!

- un trait non si può istanziare
- si usa la ***mixin-composition***: si **fornisce la classe base a cui applicare le modifiche** rappresentate dai trait

Trait: esempio

- modelliamo il ***comportamento*** di un amico

```
class Persona (val nome:String) {
  def ascolta() = print("il tuo amico "+nome+" ti ascolta")
}
class Uomo(nome:String) extends Persona(nome)
class Donna(nome:String) extends Persona(nome)
```

- però il comportamento amicizia è *immerso nella gerarchia degli umani*

  …ma **un Cane è un ottimo amico**! come **<u>riuso</u>** il comportamento?

```
class Cane(nome:String) extends Persona(nome)
```

NO!!

Trait: esempio

Java

```
interface Amico { public void ascolta(); }
class Persona implements Amico { public void ascolta(){…} }
class Cane implements Amico { public void ascolta(){ ... }}
```

duplico il codice…

```
class Animale { … }  // non tutti gli animali sono amici
class Cane extends Animale -- Amico { }
```

NO: ereditarietà singola

Scala

```
trait Amico {
  val nome:String
  def ascolta() = print("il tuo amico "+nome+" ti ascolta")
}

class Persona(val nome:String) extends AnyRef with Amico
class Uomo(nome:String) extends Persona(nome)
class Donna(nome:String) extends Persona(nome)

class Animale

class Cane(val nome:String) extends Animale with Amico {
  override def ascolta() = print(nome+" ti ascolta in silenzio")
}
```

codifica un concetto specializzato in sottoclasse

codifica un comportamento (riusablie da altre gerarchie)

```
val giova = new Uomo("Giovanni")
val pluto = new Cane("Pluto")

def aiutoDaUnAmico(a:Amico) = a.ascolta

aiutoDaUnAmico(giova) // il tuo amico Giovanni ti ascolta
aiutoDaUnAmico(pluto) // Pluto ti ascolta in silenzio
```



```
trait Amico {
  val nome:String
  def ascolta() = print("il tuo amico "+nome+" ti ascolta")
}

class Umano(val nome:String) extends AnyRef with Amico

class Animale

class Cane(val nome:String) extends Animale with Amico {
  override def ascolta() = print(nome+" ti ascolta in silenzio")
}

class Gatto(val nome:String) extends Animale
```

E i gatti? *non tutti i gatti ti ascoltano in silenzio…*

…magari qualche gatto si'!

```
val felix = new Gatto("Felix") with Amico
felix : Gatto with Amico = $anon$1@338bd37a
felix.ascolta  // il tuo amico Felix ti ascolta
```



```
trait Amico {
  val nome:String
  def ascolta() = pri
}

class Umano(val nome:

class Animale

class Cane(val
  override def
}

class Gatto(val nome:String) extends Animale
```


i trait forniscono un modo molto ***snello e veloce*** di creare **codice estremamente estensibile**

invece di creare gerarchie di classi e interfacce, si fa un **mixin "al volo"**, decorando oggetti con i comportamenti richiesti.

E i gatti? *non tutti i gatti ti ascoltano in silenzio…*

…magari qualche gatto si'!

```
val felix = new Gatto("Felix") with Amico
felix : Gatto with Amico = $anon$1@338bd37a
felix.ascolta  // il tuo amico Felix ti ascolta
```

S.Crafa UniPd AA17/18