Types algébriques de donnés: tuples et `case class`

Jean-Luc Falcone

10 mars 2022

Tuples

Tuples: Définition

Type composé immutable un nombre définis de valeurs aux types éventuellements différents.

Structure de donnée anonyme

Déclaration

On utilise des valeurs entre parenthèses séparées par des virgules:

```
val amount = (91, "USD")
val color = (0x12, 0xF3, 0x5E)
val customer = ("Jane", "Doe", 2022, true)
```

Tuples: Types

Les types de tuples sont indiqués de la même manière

```
def setColor( fg: (Byte,Byte,Byte) ): Unit

def convertToCHF( amount: (Int, String) ):
                                    (Int,String)

def createCustomer(
    data: (String, String, Int, Boolean)
): Customer
```

Tuples2: Syntaxe alternative

Pour les tuples de 2 éléments on peut utiliser une autre notation avec une flêche:

```
val t1 = "a" -> 10

val t2 = ("a", 10)

//t1 et t2 sont identiques
```

Déconstruction

Ont peut extraire les paramètres d'un tuples en le déconstruisant:

```
val x = (10, false)

val (n,b) = x  //Assignation n=10 et b=false

if( b ) n else -n //Eval: -10
```

Déconstruction: Wildcard

Le caractère *underscore* permet d'ignorer des éléments (*wildcard*):

```
val color = (0x12, 0xF3, 0x5E)
val (r, _ , b) = color

//Assigne r=0x12 et b=0x5E,
//  mais ignore le second élément
```

Pattern matching

Instruction switch/case (C/Java/...)

```
int day = getDay();
String dayString;
switch (day) {
    case 1:  dayString = "Lundi"; break;
    case 2:  dayString = "Mardi"; break;
    case 3:  dayString = "Mercredi"; break;
    case 4:  dayString = "Jeudi"; break;
    case 5:  dayString = "Vendredi"; break;
    case 6:  dayString = "Samedi"; break;
    case 7:  dayString = "Dimanche"; break;
}
System.out.println(dayString);
```

Expression Match/Case (scala)

```scala
val day = getDay()
val dayString = day match {
    case 1 => "Lundi"
    case 2 => "Mardi"
    case 3 => "Mercredi"
    case 4 => "Jeudi"
    case 5 => "Vendredi"
    case 6 => "Samedi"
    case 7 => "Dimanche"
}
println( dayString )
```

Plus de flexibilité

```
val number: Int = f()
val kind = number match {
  case 0              => "Zero"
  //Plusieurs valeurs: 1, 2, ou 3
  case 1|2|3          => "Small"
  //Par défaut
  case _              => "Large"
}
```

Pattern matching sur des tuples

```scala
def getAmount(id: Long): (Int,String)

getAmount(xxx) match {
  //Capture 'cur0 si le 1er param est 0
  case (0,cur) =>
    println(s"Empty transaction for $cur")
  //Capture 'n' si le snd param est "CHF"
  case (n,"CHF") =>
    println(s"Domestic transaction: $n")
  //capture le snd param en ignorant le premier
  case (_, cur) =>
    println(s"Foreign transaction for: $cur")
}
```

Gardes

Les gardes permettent de tester des conditions en plus:

```
def getAmount(id: Long): (Int,String)

getAmount(xxx) match {
  //Capture 'cur0 si le 1er param est < 0
  case (x,cur) if x < 0 => //garde
    println(s"Negative transaction for $cur")
  //Ne capture rien
  case _ =>
    printn(s"Positive or empty transaction")
}
```

Case class

`case class`: définition

- tuples nommés
- Similaire aux nouveaux `records` en java, ou aux `dataclass` en python nommés
- permet l'ajout de méthodes

Déclaration: exemples

```
case class Point( x: Double, y: Double )

case class Color( r: Byte, g: Byte, b: Byte )

case class Transaction( amount: Long, currency: Str
                        date: Date )
```

Construction: exemples

```
val p = Point(2,-3)

val purple = Color( 0xFF, 0, 0xFF )

val trans = Transation( 12, "CHF",
                        LocalDateTime.now )
```

Déconstruction

```
//val p = Point(2,-3)
val (x,y) = p

//val purple = Color( 0xFF, 0, 0xFF )
val (r,g,b) = purple

//val trans = Transation( 12, "CHF",
//                        LocalDateTime.now )
val (_, currency, _) = trans
```

Propriétés

Les champs d'une `case class` sont publics et immutables.

```
//val p = Point(2,-3)
val dist = sqrt( p.x*p.x + p.y*p.y )

//val purp = Color( 0xFF, 0, 0xFF )
val negative = Color( 0xFF-purp.r,
                      0xFF-purp.g,
                      0xFF-purp.b)

//val trans = Transation( 12, "CHF", ...)
if( trans.amount < 0 )
  println("Warning: negative transaction")
```

Méthodes

On peut ajouter des méthodes aux cases class:

```
case class Point( x: Double, y: Double ) {
  private def sq( n: Double ) = x*x
  def dist( that: Point ) =
    math.sqrt( sq( x-that.x) + sq( y-that.y) )
}

val p = Point( 1.5, -2 )
val q = Point( 2.1, 0 )
val d = p.dist(q)
```

Méthodes synthétisées

Les méthodes suivantes sont créées par le compilateur:

- `toString`
- `equals`
- `hashCode`
- `copy`

On peut les redéfinir en utilisant le mot clé `override`

Attention ! Egalité et identité

En java

```
// Egalité par reference ...ou valeur
p == q
// Egalité par valeur
p.equals(q)
```

Attention ! Egalité et identité

En java

```
// Egalité par reference ...ou valeur
p == q
// Egalité par valeur
p.equals(q)
```

En scala

```
// Egalité par reference
p.eq(q)
p eq q
// Egalité par valeur
p.equals(q)
p equals q
p == q
```

Méthode copy

La méthode `copy` retourne une copie en modifant éventuellement un ou plusieurs paramètres:

```
val c1 = Color( 128, 24, 19 )

val c2 = c1.copy( r = 0 )
//c2 == Color( 0, 24, 19 )

val c3 = c1.copy( r=c1.r/2, g=20 )
//c3 == Color( 64, 20, 19 )
```

Pattern matching

```scala
val c = Color( a, b, c ) //quelconque

c match {
  case Color(0,0,0) => "Black"

  case Color(r, 0, 0 ) if r > 100 => "Red"

  case Color(r,g,b) if r==g && g==b => "Grey"

  case Color(_,b,_) => s"With blue: $b"

  case _ => "..."
}
```

Case class et pattern matching imbriqué

```
case class Style( fg: Color, bg: Color )

myStyle match {
  case Style( Color(0,0,0),
              Color(255,255,255) =>
    "Noir sur blanc"

  case Style(Color(r,_,_), _ ) =>
    s"Rouge avant plan: $r"
}
```