Traits Scala et décorateurs

par Sylvain Leroux (Accueil)

Date de publication :

Dernière mise à jour :

En première approche, les **traits** de Scala s'apparentent à des interfaces (au sens Java). À ceci près qu'une interface ne fait que *déclarer* des méthodes, alors qu'un trait a la possibilité de les *mettre en oeuvre* ainsi que de déclarer des *données membres*.

Mais leur usage dépasse largement celui d'une interface enrichie. Ainsi, dans cet article, nous allons voir comment il est possible de combiner des traits à une classe pour la doter de comportements spécifiques.

Cet article suppose une connaissance élémentaire de la syntaxe de Scala.

I - Intro.....3
II - Au forfait.....4
III - Offres modulaires.....5
III-A - Héritage simple.....5
III-B - Décorateur.....6
III-C - Traits.....8
III-C-1 - super et linéarisation.....11
III-C-2 - Mixer plusieurs traits.....12
III-D - Déclarer les mixins.....13
IV - Que des traits.....14
V - Conclusion.....16
VI - Ressources.....17
VII - Remerciements.....17

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

I - Intro

Pour cet article, nous allons partir d'un cas très simple, que nous allons progressivement complexifier pour voir quelles options s'offrent à nous pour mettre en oeuvre une solution en Scala.

L'exemple qui va nous servir d'illustration ici sera celui d'un logiciel de facturation d'abonnement téléphonique. De façon rudimentaire, chaque compte est un objet doté d'une méthode pour lui imputer une communication :

Compte.scala

```
/**
 * Un compte de facturation téléphonique.
 */
class Compte(client: String) {
  /**
   * Le nombre de secondes consommées (à facturer) ce mois-ci.
   */
  var tempsConsomme = 0
  /**
   * Relevé des communications facturées.
   */
  var releve = new ArrayBuffer[Tuple3[String,Int,Int]]()

  /**
   * Impute une communication sur ce compte.
   */
  def impute(secondes: Int, destinataire: String) : Unit = {
    tempsConsomme += secondes
    releve.append((destinataire,secondes,tempsConsomme))
  }

  override def toString() : String = {
    val result = new StringBuilder()

    result.append("%-10s:".format(client))
    releve.foreach { t => result.append("%n\t%-14s%5d %5d".format(t._1, t._2, t._3)) }
    result.append("%n\t%s".format("-"*25))
    result.append("%n\t%19s %5d".format("Total:", tempsConsomme))

    result.toString()
  }
}
```

On pourrait s'assurer du fonctionnement d'un compte avec le programme de test suivant :

Chapitre_I.scala

```
object Chapitre_I {
  def main(args : Array[String]) : Unit = {

    val clients = Set(
      new Compte("george")
    )

    clients.foreach { compte =>
       /* Crédite 5 communications de 10, 120 et 900 secondes à chaque compte */
       (0 until 5).foreach { n =>
        compte.impute(10, "888")             // 10 secondes au répondeur
        compte.impute(2*60, "888")           // 2 minutes au répondeur
        compte.impute(15*60, "06xxxxxxxx")   // 15 minutes à un portable
        }

        println(compte)
    }
  }
}
```


Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Si vous lancez le programme Demo vous obtiendrez le résultat suivant :

```
george    :
 888             10    10
 888            120   130
 06xxxxxxxx     900  1030
 888             10  1040
 888            120  1160
 06xxxxxxxx     900  2060
 888             10  2070
 888            120  2190
 06xxxxxxxx     900  3090
 888             10  3100
 888            120  3220
 06xxxxxxxx     900  4120
 888             10  4130
 888            120  4250
 06xxxxxxxx     900  5150
 -------------------------
             Total:  5150
```

OK : rien de très excitant pour l'instant. Tout au plus peut-on s'assurer que les appels sont bien décomptés à la seconde. Mais vous le savez, ça n'est pas comme cela que fonctionnent tous les abonnements téléphoniques...

II - Au forfait

Jusqu'à présent, nous avons mis en place une facturation à la seconde. Mais très souvent l'utilisateur opte pour un forfait. Comment coder dans notre application un fait comme *John a un forfait "2 heures"*? Le moyen le plus simple serait de créer un nouveau type de compte. Le réflexe ici serait d'utiliser l'**héritage** :

Forfait.scala

```
/**
 * Un forfait.
 *
 * Compte sur lequel un crédit de communication est pré-facturé.
 */
class Forfait(client: String, credit: Int) extends Compte(client) {
  /**
   * Crédit restant sur le forfait
   */
  var restant = credit


  // facture d'emblée la consommation prévue dans le forfait
  super.impute(credit, "** Forfait **")

  override def impute(secondes: Int, destinataire: String) : Unit = {
    restant -= secondes
    if (restant < 0) {
      // hors forfait!
      super.impute(-restant, destinataire)
      restant = 0
    }
  }

  override def toString() : String = {
    "%s (restant sur forfait: %d)".format(super.toString(), restant)
  }
}
```

Il n'est pas compliqué ensuite de modifier le programme de test pour prendre en compte un nouveau client avec un forfait :

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Chapitre_II.scala

```
// ...
 val clients = Set(
  new Compte("george"),
  new Forfait("john",2*60*60) // forfait 2 heures
 )
// ...
```

Enfin, l'exécution nous donne le résultat attendu :

```
george    :
 888              10    10
 888             120   130
 06xxxxxxxx      900  1030
 888              10  1040
 888             120  1160
 06xxxxxxxx      900  2060
 888              10  2070
 888             120  2190
 06xxxxxxxx      900  3090
 888              10  3100
 888             120  3220
 06xxxxxxxx      900  4120
 888              10  4130
 888             120  4250
 06xxxxxxxx      900  5150
 -------------------------
             Total:  5150
john      :
 ** Forfait **  7200  7200
 -------------------------
             Total:  7200 (restant sur forfait: 2050)
```

À nouveau, rien d'extraordinaire. Et rien de spécifique à Scala : c'est de la programmation orientée objets tout ce qu'il y a de plus classique. Il est temps maintenant de complexifier un peu notre exemple...

III - Offres modulaires

Les opérateurs de téléphonie ne sont pas des gens qui sauraient se contenter de deux offres. Ainsi, pour le bonheur de leurs clients, ils aiment à leur proposer une grande variété d'options. Comme par exemple l'accès gratuit à leurs services (répondeur, suivi conso, ...). Et bien sûr, cette option doit pouvoir s'appliquer aux différents types d'abonnements. Ce qui nous donne maintenant 4 offres possibles :

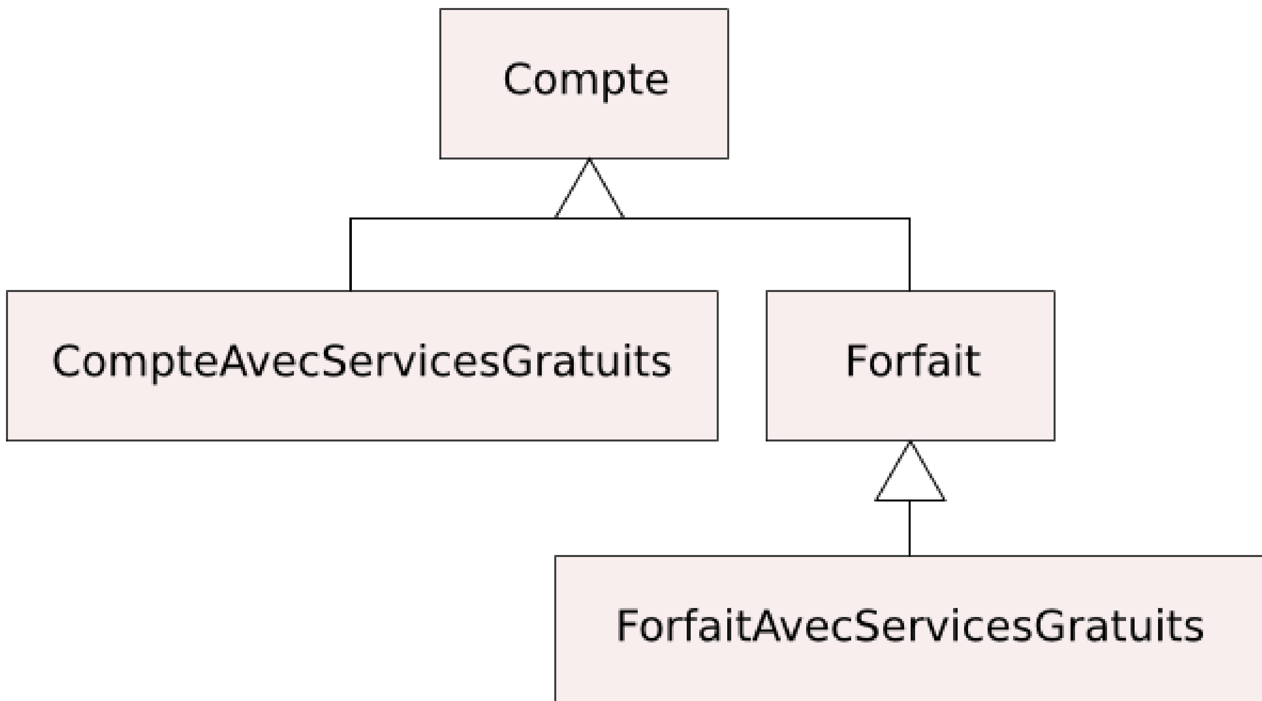
1. Facturation à la seconde ;
2. facturation au forfait ;
3. facturation à la seconde avec accès gratuit aux services de l'opérateur ;
4. facturation au forfait avec accès gratuit aux services de l'opérateur.

Alors, comment coder cela en Scala ? C'est ce que nous allons examiner dans la suite de cet article, en envisageant successivement plusieurs des options disponibles : tout d'abord, nous allons poursuivre sur notre lancée en envisageant une solution basée sur l'**héritage simple**. Puis nous aborderons le **modèle de conception décorateur**. Et enfin, nous terminerons en voyant comment Scala propose au niveau du langage une alternative avec les **traits**.

III-A - Héritage simple

Pour commencer, examinons la possibilité d'utiliser l'**héritage simple** pour modéliser toutes ces différentes offres. Après tout, cette solution avait semblé satisfaisante lors de l'introduction du *forfait*. Alors, pourquoi changer une solution qui marche ?

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

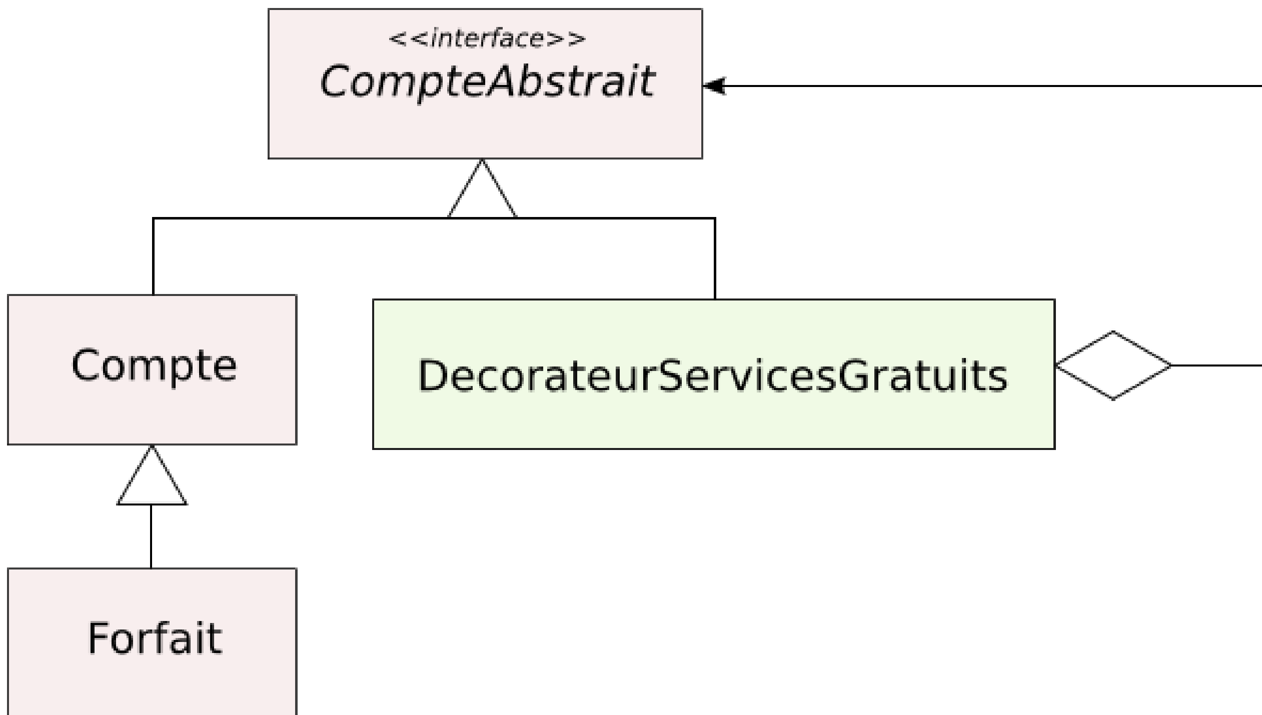


Comme on le constate sur le diagramme UML ci-dessus, ajouter *une* seule option implique de créer *deux* nouvelles classes : une pour les forfaits, une pour les comptes. Inutile de dire que le code risque d'être largement redondant. Par ailleurs, au-delà de cet exemple, il n'est pas difficile de se rendre compte qu'avec cette solution la multiplication des options va entrainer une explosion combinatoire du nombre de classes à coder dans le système. Bref, cette solution mène rapidement à une maintenabilité cauchemardesque. C'est sans doute le moment de faire appel au *Gang des 4* pour voir ce qu'il nous propose...

III-B - Décorateur

La solution classique à ce problème est l'utilisation du **modèle de conception décorateur**. Avec ce modèle, la logique spécifique de chaque option n'a plus à être codée qu'une seule fois. Et peut être utilisée pour décorer n'importe quelle *instance* de CompteAbstrait. Éventuellement un compte déjà décoré, ce qui permet de faire des combinaisons.

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



Une mise en oeuvre serait la suivante :

DecorateurServicesGratuits.scala

```
/**
 * Une classe qui met en oeuvre la gratuité de l'accès aux
 * services de l'opérateur.
 *
 * Ce décorateur peut s'appliquer à n'importe quel CompteAbstrait.
 */
class DecorateurServicesGratuits(base: CompteAbstrait) extends CompteAbstrait {
  override def impute(secondes: Int, destinataire: String) : Unit = {
    val servicesCompris = Set("888", "750", "751")

    // si le destinaire ne fait pas parti des services gratuits, l'imputer
    if (!servicesCompris.contains(destinataire))
      base.impute(secondes, destinataire)
  }

  override def toString() = {
    base.toString()
  }
}
```

Chapitre_III_B.scala

```
// ...
    val clients = Set(
      new Compte("george"),
      new DecorateurServicesGratuits(new Forfait("john",2*60*60)), // forfait 2 heures
      new DecorateurServicesGratuits(new Compte("paul"))
    )
// ...
```

Le décorateur permet de modifier *dynamiquement* le comportement d'un objet *pendant l'exécution*. Par ailleurs, la mise en oeuvre d'une interface commune permet aux décorateurs de se substituer à l'objet qu'ils décorent. C'est d'ailleurs cette caractéristique qui permet de les combiner. Bien sûr, cela implique que le décorateur implémente bien toutes les méthodes utilisables de l'objet qu'il cache - au moins comme des stubs qui passent simplement le contrôle à l'objet de base. C'est le cas dans l'exemple ci-dessus de la méthode toString().

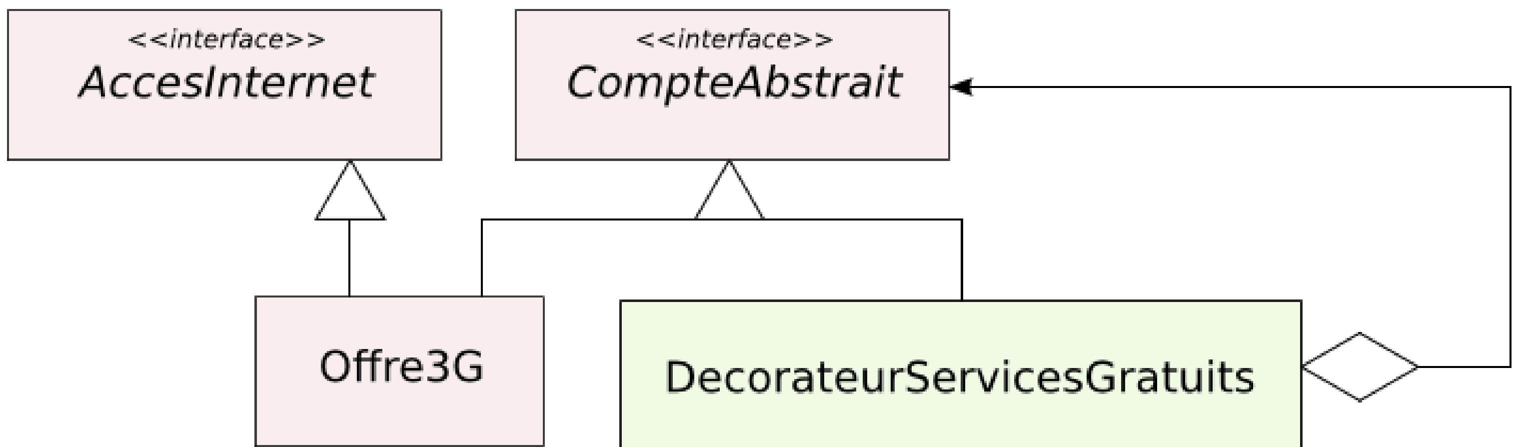
Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Bien qu'éventuellement fastidieux, ça n'est guère un problème dans le cas ou la classe abstraite racine contient bien toutes les méthodes nécessaires. Mais malgré tout, cela peut soulever un problème. Reprenons le cas du forfait avec services gratuits :

```
new DecorateurServicesGratuits(new Forfait("john",2*60*60))
```

L'objet créé ainsi peut être utilisé comme un CompteAbstrait - puisqu'en vertu de l'héritage DecorateurServicesGratuits *est* un CompteAbstrait. Par contre, les méthodes et propriétés spécifiques au Forfait ne sont plus accessibles de l'extérieur.

Ce problème est plus susceptible d'apparaître quand on décore un objet mettant en oeuvre *plusieurs* interfaces. Seules les interfaces effectivement mises en oeuvre par le décorateur permettent de manipuler l'objet. Dans l'illustration ci-dessous, l'utilisation du décorateur *ServicesGratuits* restreint les méthodes accessibles sur l'*Offre3G* à celles de l'interface *CompteAbstrait*. Celles de l'interface *AccesInternet* ne sont plus accessibles, bien que mises en oeuvre par l'objet concret.



Autrement dit, un décorateur ne respecte pas le principe de substitution de Liskov : un décorateur *ne peut pas* être utilisé partout où l'objet décoré peut l'être.

D'une certaine manière, le décorateur a les inconvénients de ses avantages : il est conçu pour modifier dynamiquement à l'exécution le comportement d'un *objet*. Pas pour créer statiquement à la compilation de nouveaux *types*. Or c'est justement ce que permettent les traits de Scala...

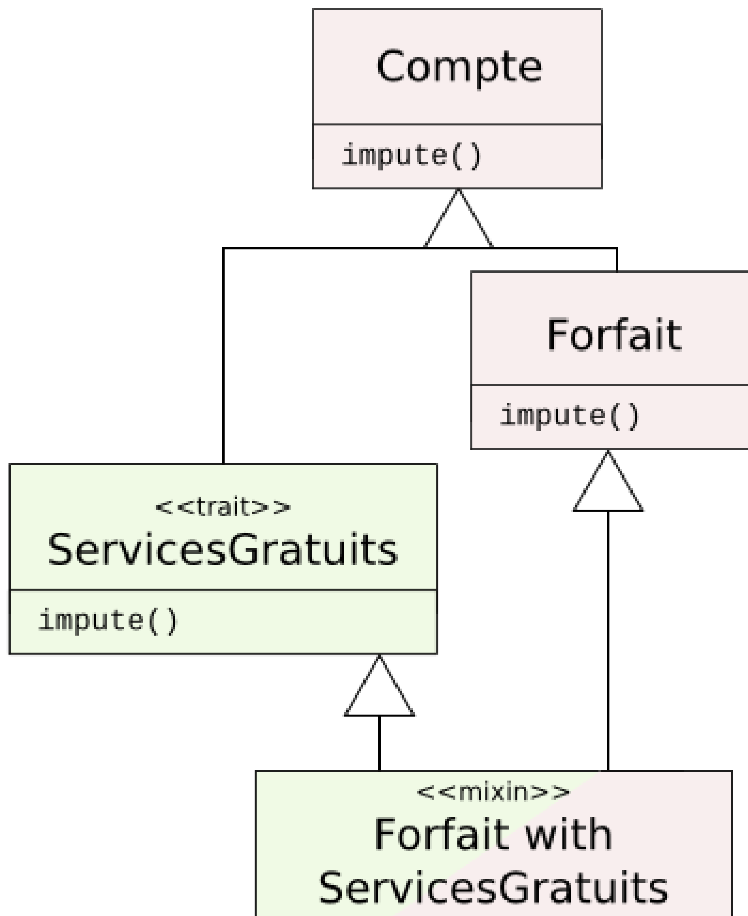
III-C - Traits

Scala nous offre la possibilité de **construire de nouveaux types à la compilation** en combinant les données et méthodes membres d'une classe avec celles d'un ou plusieurs **traits**. Dans la terminologie de Scala, des classes qui sont construites de cette manière sont appelées des **mixins**.

La notation qui permet de créer des mixins utilise le mot-clé *with* comme dans l'exemple ci-dessous. J'attire votre attention sur le fait que le trait ServicesGratuits peut être indifféremment *mixé* à un Compte ou un Forfait :

Chapitre_III_C.scala
```
// ...
    val clients = Set(
      new Compte("george"),
      new Forfait("john",2*60*60) with ServicesGratuits, // forfait 2 heures avec services gratuits
      new Compte("paul") with ServicesGratuits           // compte avec services gratuits
    )
// ...
```


Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



La différence fondamentale avec le modèle de conception décorateur est illustrée dans le diagramme UML ci-dessus : vous y constaterez que **seules des relations d'héritage sont présentes**. Par conséquent, un forfait avec service gratuit *est* un Forfait. Autrement dit encore **un *mixin* peut être utilisé partout où une instance d'*une de ses classes de base* ou d'*un des traits mixés* peut l'être**.

Si à ce point nous avons détaillé l'*utilisation* d'un trait, nous n'avons pas encore vu comment le *définir*. Avec le code ci-dessous ce sera fait. Comme vous le verrez, la définition d'un trait ressemble beaucoup à celle d'une classe. La seule différence évidente étant l'utilisation du mot-clé *trait* à la place de *class* :

ServicesGratuits.scala

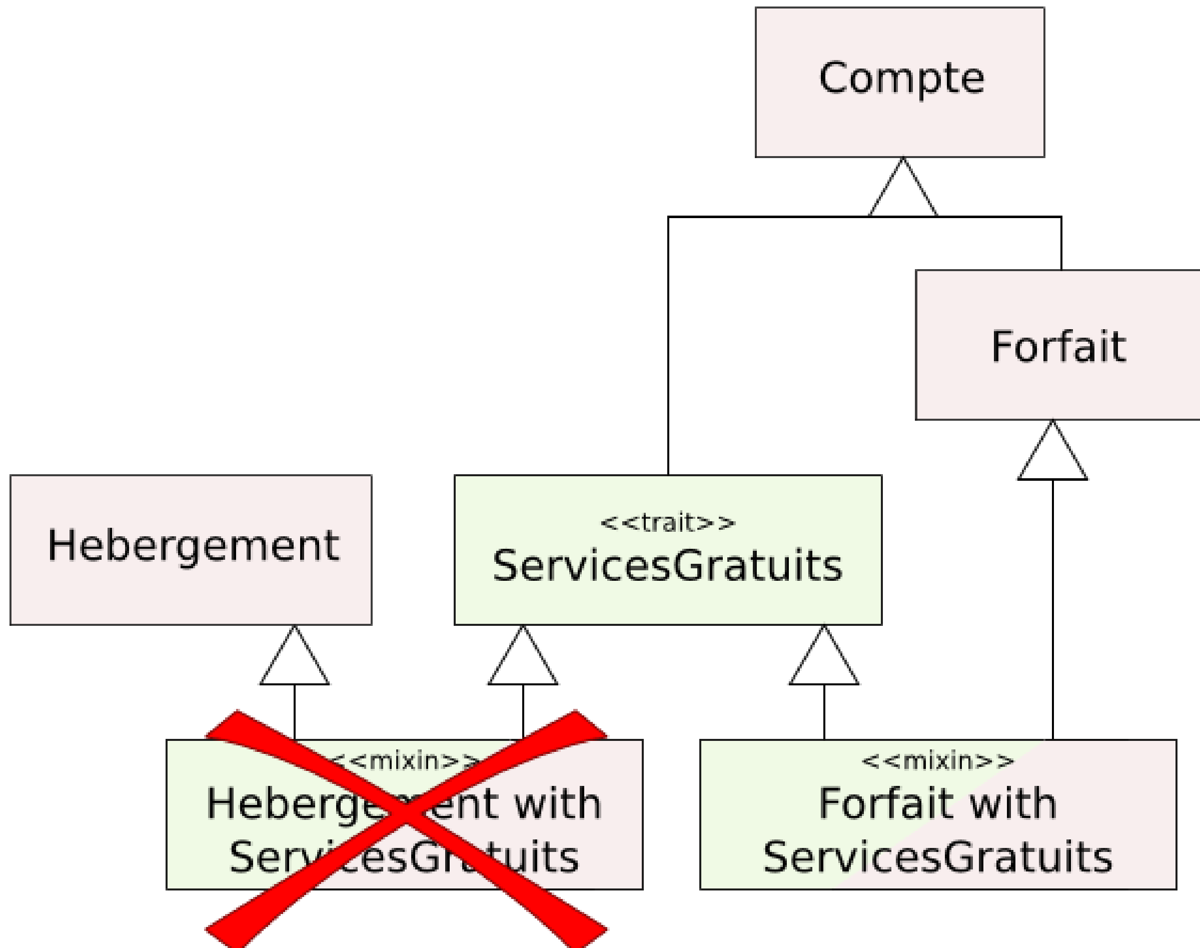
```
/**
 * Un trait qui met en oeuvre la gratuité de l'accès aux
 * services de l'opérateur.
 *
 * Le trait ServicesGratuits ne peut s'appliquer qu'à un Compte.
 */
trait ServicesGratuits extends Compte {
  override def impute(secondes: Int, destinataire: String) : Unit = {
    val servicesCompris = Set("888", "750", "751")

    // si le destinaire ne fait pas parti des services gratuits, l'imputer
    if (!servicesCompris.contains(destinataire))
      super.impute(secondes, destinataire)
  }
}
```

Remarquez que le trait ServicesGratuits *étend* la classe Compte. Mais pour un trait, cette déclaration est plutôt une déclaration de *contrainte* : **un trait ne peut être mixé qu'avec sa classe de base ou une classe dérivée de celle-ci**. Le corrolaire est qu'**une classe ne peut être combinée qu'avec un trait dont le parent direct est dans sa hiérarchie parente**.

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Cette contrainte à pour conséquence que la sémantique de la relation d'héritage est conservée au niveau des objets. En effet, puisque ServicesGratuits ne peut être mixé qu'avec Compte ou une de ses classes dérivées, on peut donc dire avec certitude que n'importe quelle instance mixant ServicesGratuits *est aussi* une instance de Compte.



Il est impossible de mixer le trait ServicesGratuits à la classe Hebergement si celle-ci n'est pas une classe dérivée de Compte.

Plus subtil encore, remarquez dans le code du trait l'appel *super.impute* :

ServicesGratuits.scala
```
trait ServicesGratuits extends Compte {
  override def impute(secondes: Int, destinataire: String) : Unit = {
    // ...
      super.impute(secondes, destinataire)
  }
...
```

Dans le contexte d'une classe, cette notation signifie "appeler la méthode définie dans la *classe de base*". Mais *pas nécessairement* dans le cas d'un trait ! En effet, au moment de la définition du trait, Scala ne sait pas encore avec quelle classe il va être mixé. Par conséquent, la classe référencée par *super* n'est résolue qu'au moment de la compilation du *mixin*. Par ailleurs, d'un *mixin* à l'autre une référence à super dans le même trait ne référencera pas nécessairement le même ancêtre.

Et comment Scala détermine la classe référencée par super ? Grâce à une technique appelée **linéarisation** que nous allons maintenant aborder.

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

III-C-1 - super et linéarisation

Le point à comprendre, c'est qu'avec l'introduction des traits, le graphe d'héritage (d'implémentation) n'est plus forcément un arbre comme ça l'est en Java. Afin de déterminer à quelle classe ou trait *super* va faire référence, Scala établit à la compilation *pour chaque mixin* un ordre strict pour les différentes classes et traits qui le composent. C'est la **linéarisation**.

Les spécifications de Scala précisent l'algorithme précis employé. Mais en faisant simple, dans l'ordre déterminé par la linéarisation, un mixin apparaît toujours avant ses traits mixés et avant sa super-classe. Et si une même classe ou un même trait apparaît plusieurs fois, seule la *dernière* occurrence est conservée. Ce dernier point a en particulier pour conséquence que selon la classe avec laquelle un trait est mixé, super ne fait pas nécessairement référence au même type.

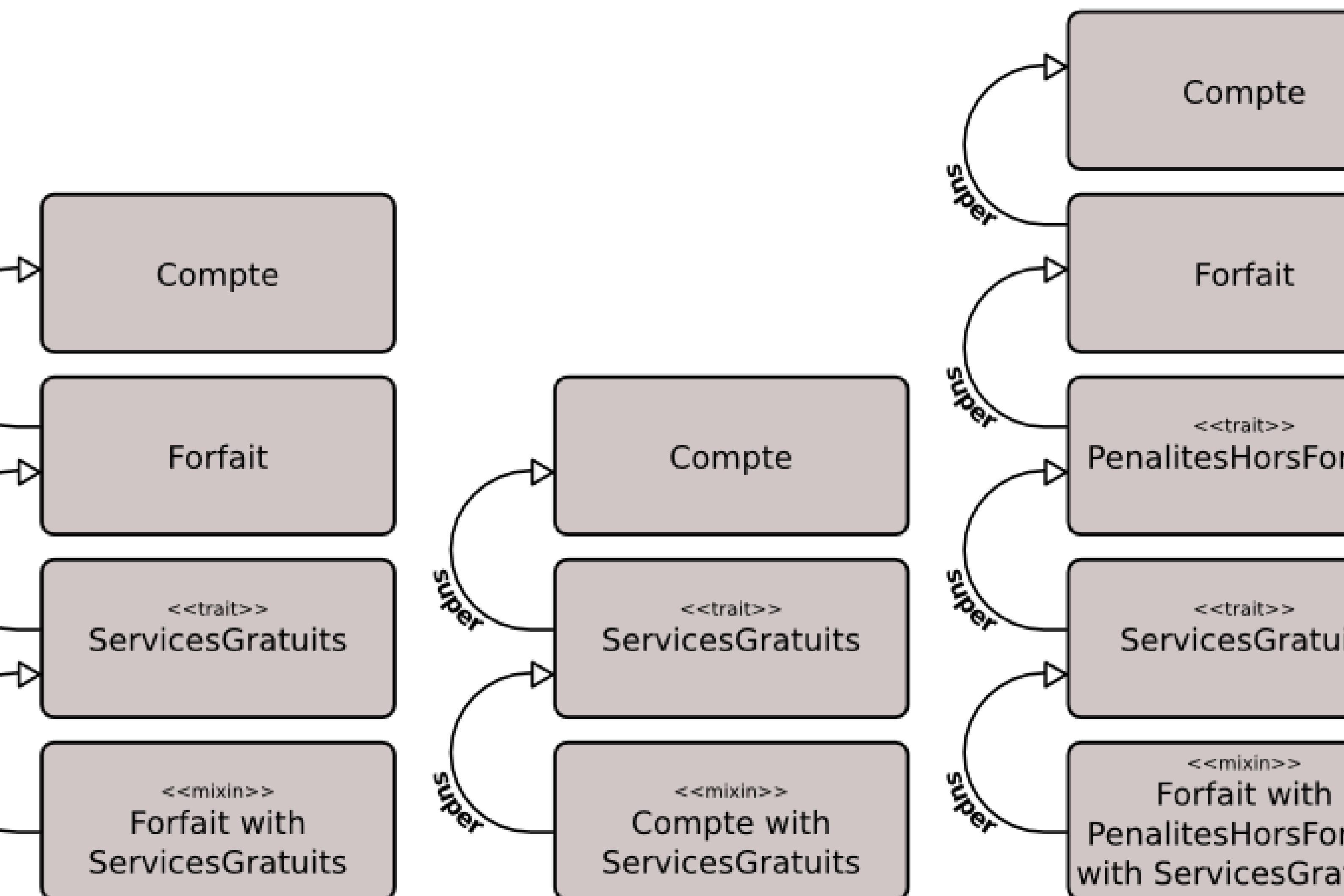
Pour être plus concret, l'ordre retenu pour le mixin *Forfait with ServicesGratuits* est le suivant :

1. Forfait with ServicesGratuits (le mixin lui-même) ;
2. ServicesGratuits (les traits sont linéarisés avant la classe de base) ;
3. Forfait (une classe est linéarisée avant sa super-classe) ;
4. Compte

A titre de comparaison, l'ordre pour le mixin Compte with ServicesGratuits est:

1. Compte with ServicesGratuits (le mixin lui-même)
2. ServicesGratuits (les traits sont linéarisés avant la classe de base)
3. Compte

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.



Scala linéarise chaque mixin indépendamment. Ainsi, l'appel super.impute dans la méthode du trait ServicesGratuits invoque dans un cas Forfait.impute, dans l'autre Compte.impute. Nous verrons aussi dans la prochaine section que l'on peut mixer plusieurs traits en même temps et que super peut aussi désigner un autre trait.

III-C-2 - Mixer plusieurs traits

Nous allons maintenant aborder le coeur de cette technique: à savoir la possibilité de **combiner plusieurs traits dans un mixin**. Pour illustrer ce point, nous allons nous appuyer sur le cas de Ringo, qui a pris un forfait "1 heure", avec accès gratuit aux services de l'opérateur, mais dans lequel *les communications en dehors du forfait comptent double*. Pour mettre en oeuvre cette dernière spécification, à nouveau un trait peut être utilisé :

PenalitesHorsForfait.scala

```
trait PenalitesHorsForfait extends Forfait {
  override def impute(secondes: Int, destinataire: String) : Unit = {
    if (secondes > restant)
      super.impute(secondes-restant, "** Pénalité **")

    super.impute(secondes, destinataire)
  }
}
```

La PenalitesHorsForfait ne peut s'appliquer qu'à un Forfait. Ce qui est explicitement indiqué dans la déclaration du trait :

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
trait PenalitesHorsForfait extends Forfait { ...
```

Nous voici donc avec un utilisateur dont le forfait doit mixer deux traits. Ce qui s'écrit ainsi en Scala :

Chapitre_III_C_2.scala

```
// ...
    val clients = Set(
      new Compte("george"),
      new Forfait("john",2*60*60) with ServicesGratuits, // forfait 2 heures
      new Compte("paul") with ServicesGratuits,
      new Forfait("Ringo", 1*60*60) with PenalitesHorsForfait with ServicesGratuits
    )
// ...
```

Avec deux traits mixés avec la classe Forfait se pose à nouveau la question de l'ordre dans lequel invoquer les méthodes. La règle est simple, les derniers traits (dans la déclaration) sont prioritaires. Ce qui implique que **l'ordre des traits est important** !

Sur mon exemple, ServicesGratuits.impute arrivera avant PenalitesHorsForfait.impute. Ce qui est le comportement désiré : dans notre application il est souhaitable que ServicesGratuits ait l'opportunité d'écarter certaines communications du système de facturation avant que le calcul des pénalités éventuelles ne soit fait.

```
george    :
 888              10    10
 888             120   130
 06xxxxxxx        900  1030
 888              10  1040
 888             120  1160
 06xxxxxxx        900  2060
 888              10  2070
 888             120  2190
 06xxxxxxx        900  3090
 888              10  3100
 888             120  3220
 06xxxxxxx        900  4120
 888              10  4130
 888             120  4250
 06xxxxxxx        900  5150
 -------------------------
             Total:  5150
john      :
 ** Forfait **   7200  7200
 -------------------------
             Total:  7200 (restant sur forfait: 2700)
paul      :
 06xxxxxxx        900   900
 06xxxxxxx        900  1800
 06xxxxxxx        900  2700
 06xxxxxxx        900  3600
 06xxxxxxx        900  4500
 -------------------------
             Total:  4500
Ringo     :
 ** Forfait **   3600  3600
 ** Pénalité **   900  4500
 06xxxxxxx        900  5400
 -------------------------
             Total:  5400 (restant sur forfait: 0)
```

III-D - Déclarer les mixins

Jusqu'à présent, nous avons mixé les traits (*with ...*) directement au moment d'instancier les objets (*new ...*). Ce n'est guère élégant. Une meilleure option aurait été de déclarer les *mixin* une fois pour toute. Puis de les instancier au besoin:

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Chapitre_III_D.scala

```
 /* Déclaration de trois nouveaux types */
 class OffreALaCarte(client: String) extends Compte(client) with ServicesGratuits
 class OffreLiberte(client: String, plafond: Int) extends Forfait(client, plafond) with
ServicesGratuits
 class OffrePrimo(client: String, plafond: Int) extends Forfait(client, plafond) with
PenalitesHorsForfait with ServicesGratuits
```

Chapitre_III_D.scala

```
// ...
    val clients = Set(
      new Compte("george"),
      new OffreLiberte("john",2*60*60)
      new OffreALaCarte("paul"),
      new OffrePrimo("Ringo", 1*60*60)
    )
// ...
```

IV - Que des traits...

Au cours de cet article, nous sommes parti d'une solution *classique* à base d'héritage simple pour évoluer vers la mise en oeuvre de traits pour représenter les différentes options gérées par notre application. Pour poursuivre dans cet esprit, on pourrait être tenté de transformer la *classe* Forfait en *trait*.

Rien ne nous y oblige. À part peut-être le désir d'uniformiser notre solution. Pour être honnête, cela va surtout être l'occasion pour moi d'introduire quelques notions supplémentaires. Mais n'anticipons pas trop, et commençons avec ce que nous savons déjà. *A priori* pour transformer la *classe* Forfait en *trait*, il suffit de remplacer le mot-clé *class* par *trait* :

```
trait Forfait(client: String, credit: Int) extends Compte(client) {
  // contenu inchangé par rapport à la classe
  // ...
}
```

Le problème est que le code ci-dessus n'est pas du code Scala valide ! En effet, si vous tentez de le compiler, vous obtiendrez un message vous précisant :

```
traits or objects may not have parameters
```

Contrairement à une classe, **un trait ne peut pas avoir de paramètre**. En fait, un trait ne peut pas avoir de constructeur. Et encore moins invoquer le constructeur de sa classe de base. Bref, la définition de notre trait ne peut ressembler qu'à ceci :

```
// Le constructeur implicite du trait ne peut pas recevoir d'argument
// Et il ne peut pas non plus passer des paramètres au constructeur de la classe de base
trait Forfait extends Compte {
  // ...
}
```

Ce qui nous pose problème : en effet, il est nécessaire de pouvoir fixer le crédit disponible sur le forfait. Heureusement, Scala nous offre une possibilité de contourner ce problème en utilisant des **valeurs abstraites**. Le principe est le même que celui des *méthodes* abstraites que vous connaissez dans d'autres langages de programmation : elles sont *déclarées* et peuvent être *utilisées* dans une classe de base. Mais elles ne seront *définies* que dans une classe dérivée.

Dans notre cas, la *valeur* dont nous avons besoin est le crédit disponible sur le forfait. Nous allons donc faire de cette valeur une valeur abstraite. Et celle-ci sera définie dans le *mixin* :

```
trait Forfait extends Compte {
```


Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

```
  // Valeur "abstraite" - devra être définie par le mixin
  val credit: Int

  // Le reste du code est identique
  /**
   * Crédit restant sur le forfait
   */
  var restant = credit

  // facture d'emblée la consommation prévue dans le forfait
  super.impute(credit, "** Forfait **")

  override def impute(secondes: Int, destinataire: String) : Unit = {
    restant -= secondes
    if (restant < 0) {
      // hors forfait!
      super.impute(-restant, destinataire)
      restant = 0
    }
  }

  override def toString() : String = {
    "%s (restant sur forfait: %d)".format(super.toString(), restant)
  }
}
```

Maintenant que notre trait possède une valeur abstraite, celle-ci doit être définie au moment d'instancier une classe mixant ce trait :

```
new Compte("john") with Forfait with ServicesGratuits {
    // Définit la valeur abstraite pour cette instance
    val credit = 2*60*60
  }
```

Il est aussi possible de définir la valeur abstraite dans le mixin :

Chapitre_IV.scala

```
  class OffreLiberte(client: String, plafond: Int)
    extends Compte(client)
    with Forfait
    with ServicesGratuits {
    // Définit la valeur abstraite à partir du paramètre
    // du constructeur implicite de ce mixin
    val credit = plafond
  }

  // ...

  new OffreLiberte("john", 2*60*60)
```

```
...
john       :
 ** Forfait **   7200   7200
 -------------------------
              Total:   7200  (restant sur forfait: 2700)
...
```

Excellent, n'est-ce pas ? Eh bien, pas tout à fait ! Imaginons que nous modifions un peu notre programme pour que la durée des forfaits soit exprimée en heure au moment de la création de ceux-ci. Le changement est trivial :

Chapitre_IV.scala

```
  // Maintenant, le plafond est exprimé en heures
  class OffreLiberte(client: String, plafondEnHeure: Int)
    extends Compte(client)
```


Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

Chapitre_IV.scala
```
    with Forfait
    with ServicesGratuits {
    val credit = plafondEnHeure*3600 // converti les heures en secondes
  }

  // ...

  new OffreLiberte("john", 2) // forfait 2 heures
```

```
...
john      :
 ** Forfait **     0     0
 06xxxxxxxx      900   900
 06xxxxxxxx      900  1800
 06xxxxxxxx      900  2700
 06xxxxxxxx      900  3600
 06xxxxxxxx      900  4500
 -------------------------
             Total:  4500 (restant sur forfait: 0)
...
```

Hein ? Que s'est-il passé ? Notez en particulier la valeur du forfait imputé : 0 seconde ! L'explication de ce mystère est que, lorsqu'une valeur abstraite est initialisée par une expression, celle-ci est évaluée *après* l'initialisation des classes de bases et des traits mixés. C'est exactement le contraire de ce qui se passe quand on transmet des arguments à un constructeur. Dans ce cas, les valeurs des arguments sont évaluées *avant* l'exécution des différents constructeurs. C'est particulièrement piégeux, car comme nous l'avons vu précédemment, quand on initialise une valeur abstraite avec une constante, le compilateur propage celle-ci, ce qui dissimule ce comportement.

Si ces explications permettent de comprendre ce qui s'est passé, reste le problème de comment obtenir le comportement souhaité ? Une solution est de **pré-initialiser** les valeurs (en anglais, on parle de *early definitions*). Concrètement, il s'agit de faire apparaître l'initialisation des valeurs abstraites dans un bloc placé *avant* la classe de base :

Chapitre_IV.scala
```
  // Maintenant, le plafond est exprimé en heures
  class OffreLiberte(client: String, plafondEnHeure: Int)
    extends { // pré-initialisation de la valeur abstraite
      val credit = plafondEnHeure*3600
    } with Compte(client) // exceptionnellement ici, la classe de base est introduite par 'with'
    with Forfait
    with ServicesGratuits

  // ...

  new OffreLiberte("john", 2) // forfait 2 heures
```

```
...
john      :
 ** Forfait **  7200  7200
 -------------------------
             Total:  7200 (restant sur forfait: 2700)
...
```

V - Conclusion

Voilà, cette présentation des traits sous la forme de comportements empilables est terminée. Nous avons vu ici que les traits permettent de composer de manière modulaire des classes. Bien sûr, cette technique n'est pas une panacée. Elle a ses propres limitations. Justement liées au fait que les mixins sont créés statiquement à la compilation. Ainsi, il est nécessaire de recompiler tous ses mixins si vous ajoutez une méthode à un trait. Et on ne peut pas (facilement) créer à l'exécution de nouveaux mixins. Néanmoins, il s'agit d'une technique intéressante pour rendre le code modulaire. À ajouter à votre trousse à outils si vous programmez en Scala !

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.

VI - Ressources

- **Introductions aux traits - par djo.mos sur Developpez.com**
- **Le modèle de conception Scala**
- **Discution relative à la différence entre**
- *Programming in Scala: A Comprehensive Step-by-step Guide* - par Martin Odersky, Lex Spoon et Bill Venners (978-0981531601)

VII - Remerciements

Un grand merci à Eric Siber pour m'avoir proposé d'héberger des articles sur Developpez.com et pour ses relectures de cet article. À Damien Guichard pour son intérêt pour Scala. Ainsi qu'à Jacques Thery pour son oeil d'aigle qui m'a évité bien des coquilles !

Copyright © 2010 - Sylvain Leroux. Aucune reproduction, même partielle, ne peut être faite de ce site et de l'ensemble de son contenu : textes, documents, images, etc sans l'autorisation expresse de l'auteur. Sinon vous encourez selon la loi jusqu'à 3 ans de prison et jusqu'à 300 000 E de dommages et intérêts.