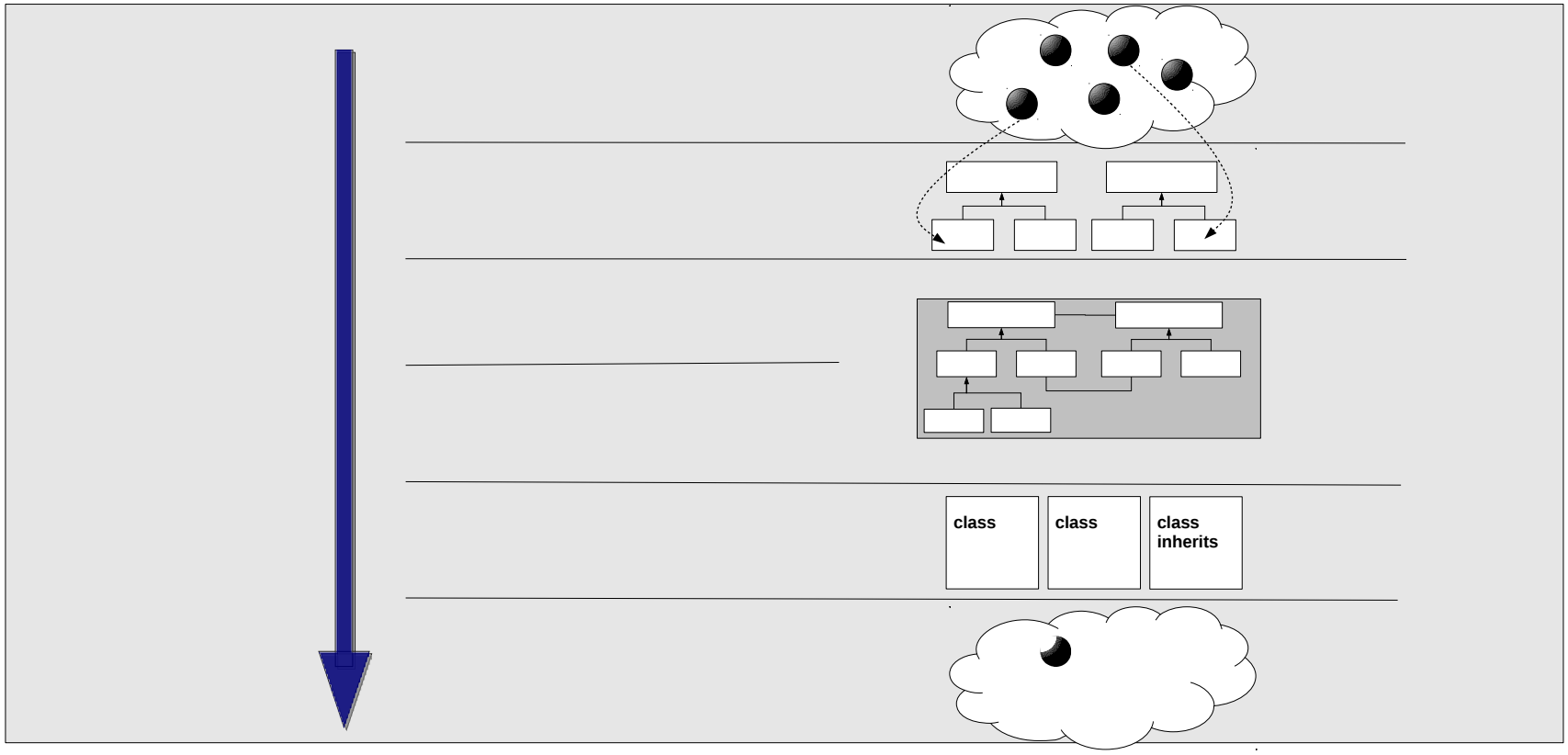
Construire des objets

Yann **Régis-Gianas**
yrg@pps.jussieu.fr

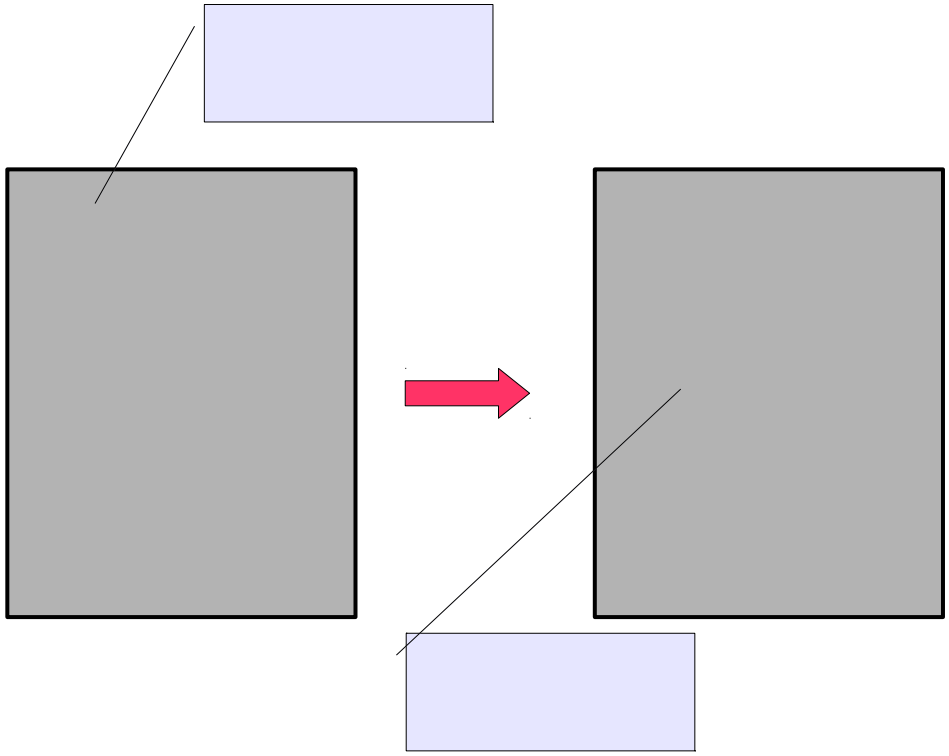
PPS - Université Denis Diderot – Paris 7

6 octobre 2011

Souvenez-vous du dernier cours . . .

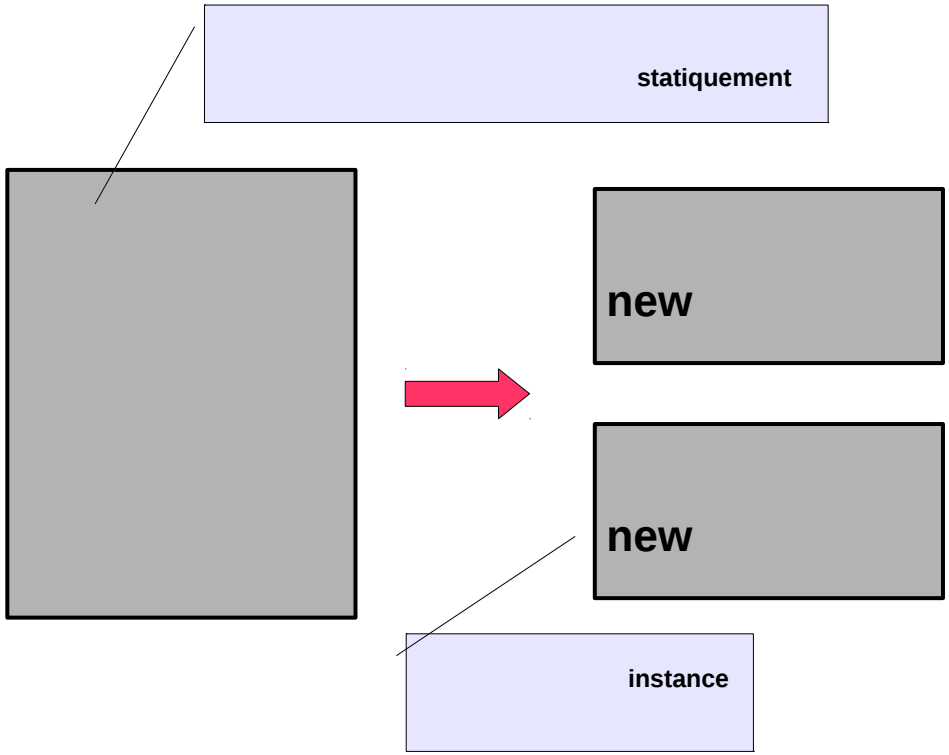


Construire par clonage



- “Object1” et “Object2” se comportent de la même façon.
- Ils ont cependant deux états distincts.

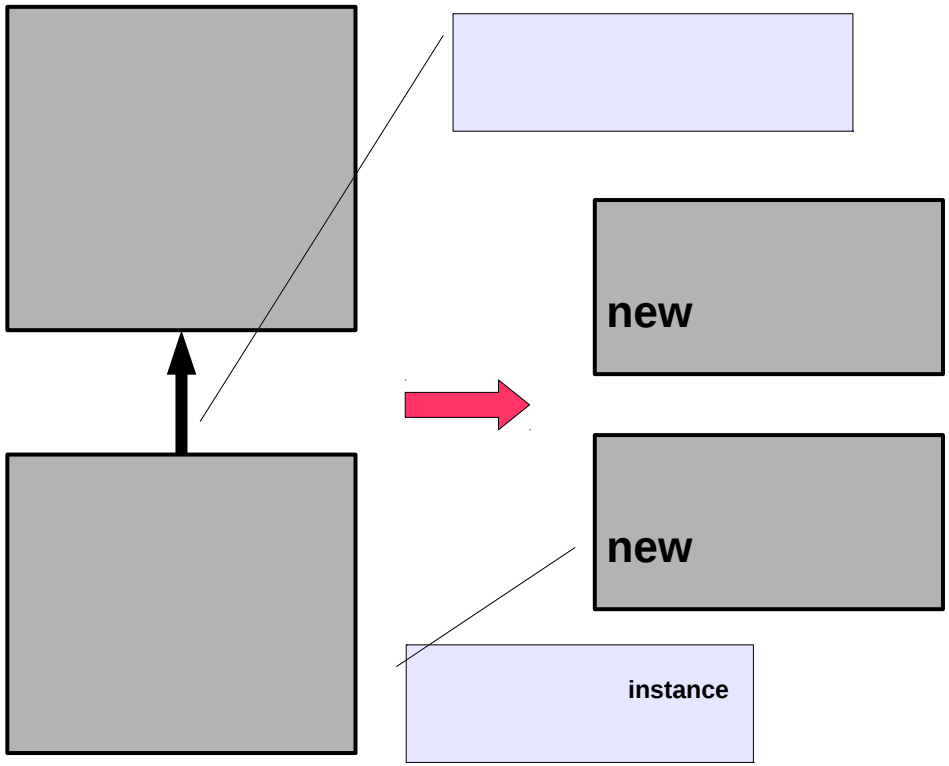
Construire par instanciation de classe



- “Object1” et “Object2” se comportent de la même façon.
- Ils ont cependant deux états distincts.
- Par construction, on a la garantie statique qu’ils ont été construits par la même classe.

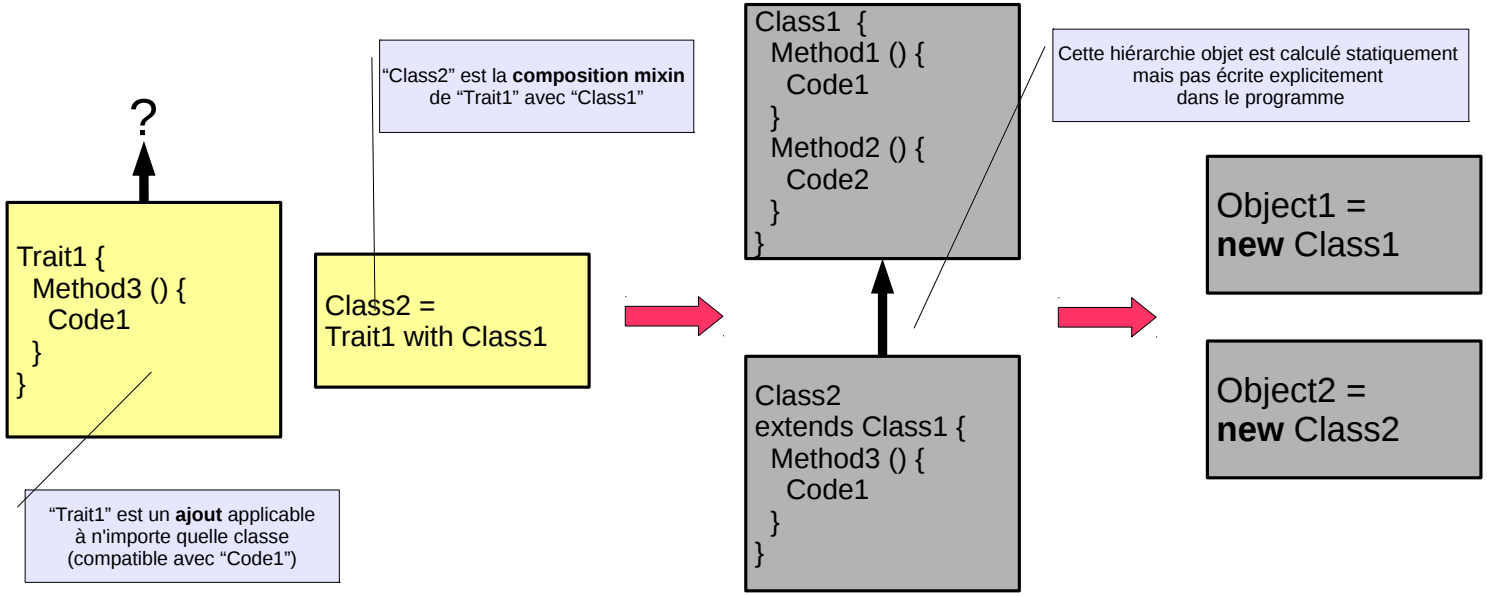
⇒ Ils ont le même type.

Construire des classes par héritage



- On peut construire statiquement de nouvelles classes par héritage.
- Nouvelle garantie statique :
  « Partout où une instance de “Class1” est attendue,
  une instance de “Class2” fait l’affaire ! »

Construire des hiérarchives par composition mixin



- On peut construire statiquement de nouvelles hiérarchies de classe.
- Un trait est un composant logiciel plus modulaire qu'une classe car :
  - Interdépendance faible :
    La définition d'une classe dépendant de la classe dont elle hérite.
    La définition d'un trait est indépendante des autres composants.
  - Forte cohérence :
    La définition d'un trait se concentre sur l'ajout d'une fonctionnalité.

Plan de la partie *“Construire des objets”*

Plan de la partie *“Construire des objets”*

Une bibliothèque de composants d'interface graphique

- Qu'est-ce qu'une interface graphique ?
- De quoi est-elle composée ? Comment l'utilise-t-on ? Comment ses constituants collaborent pour réaliser ce comportement ?

  (Dans un langage fonctionnel, on se poserait plutôt les questions : Quelles sont les entrées et les sorties d'une interface graphique ? Quelles sont les données intermédiaires entre ces entrées et ces sorties ? . . .)

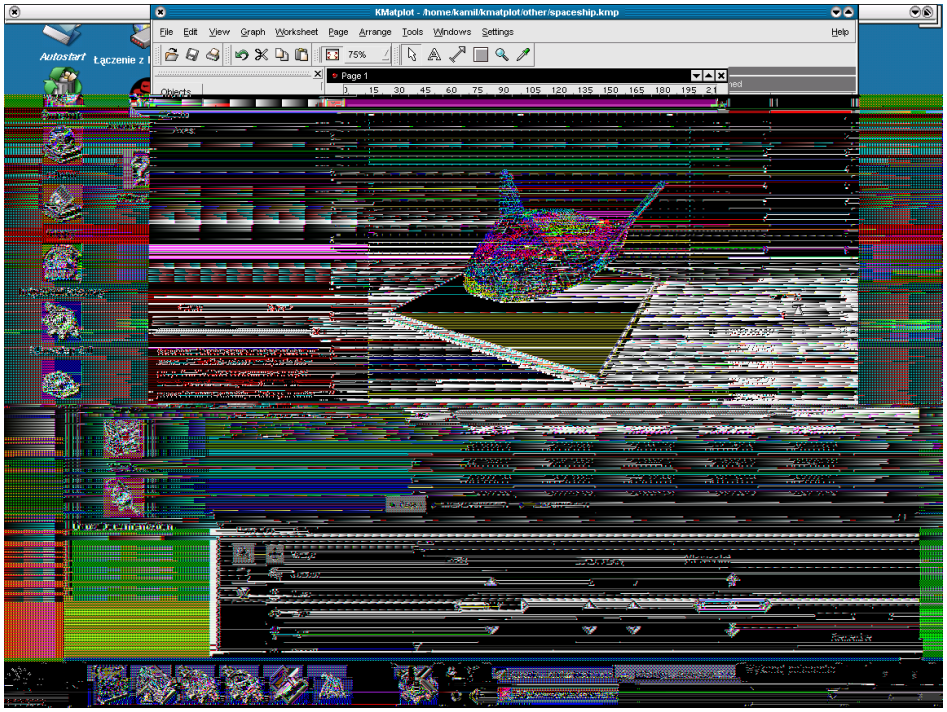
Quel objectif ?

Incrémentalité

Un développement logiciel est incrémental si il est constitué d'itérations de cycles de développement courts.

Incrémentalité

Quel objectif ?



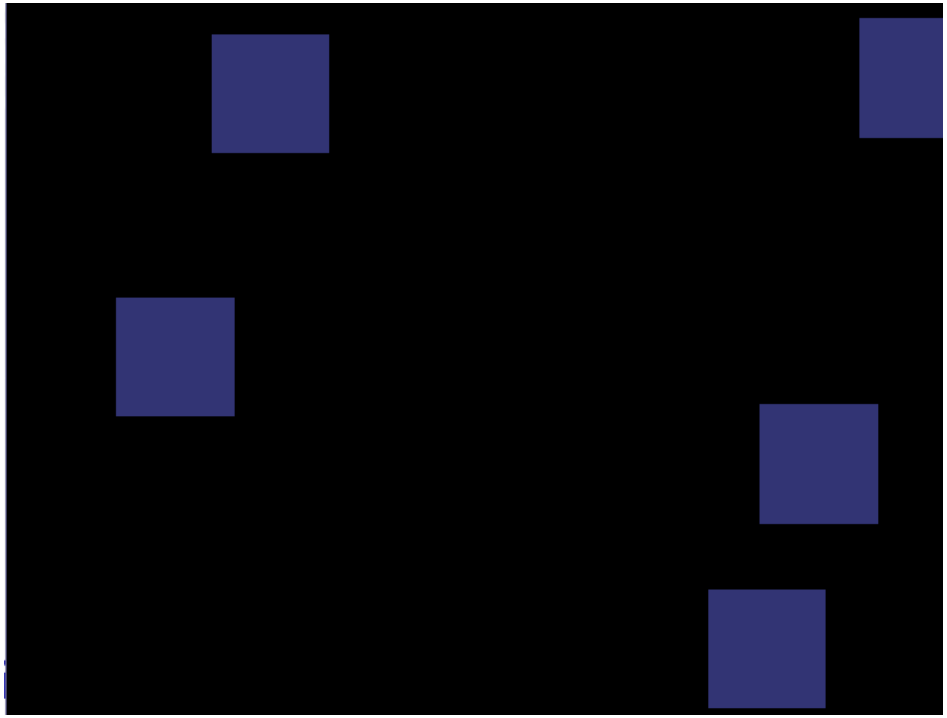
Ceci ?

Incrémentalité

Un développement logiciel est incrémental si il est constitué d'itérations de cycles de développement courts.

Incrémentalité

Quel objectif ?



ou plutôt cela ?

Incrémentalité

Un développement logiciel est incrémental si il est constitué d'itérations de cycles de développement courts.

Incrémentalité

Prérequis

1. Hiérarchisation des fonctionnalités du logiciel
2. Extensibilité du système

Avantages

- Aspects les plus importants et les plus risqués traités en premier
- Rapidement testable

Dangers

- Tension entre les prérequis :
  - Prérequis 1 : une focalisation sur les fonctionnalités « noyau »
  - Prérequis 2 : une vision à long terme

$\Rightarrow$ Nécessite de l'expérience et une compréhension profonde du problème.

Construction incrémentale d'une méthode incrémentale

Méthode de développement « objet » (v0.1, code "Extreme Test Driven")

1. Brainstorming :
   (i) Énumérer les objets du problème.
   (ii) Expliciter des situations de collaboration entre ces objets pour résoudre un problème.
   (iii) Hiérarchiser ces situations et ces objets pour faire apparaître un ensemble « noyau » minimal de fonctionnalités importantes.
2. Écrire un programme qui spécifie suffisamment ces objets pour décrire les situations de collaboration visant à réaliser les fonctionnalités importantes sous la forme de tests exécutables.
   Ici, on ne définit pas le corps des méthodes des objets du système.
3. Implémenter le corps des méthodes en validant les tests.

Apprentissage de Scala

- Découverte progressive du langage en commentant sa spécification :

  http://www.scala-lang.org/sites/default/files/linuxsoft_archives/docu/files/ScalaReference.pdf

  (Un objectif : apprendre à lire une spécification de langage de programmation.)

... et surtout en écrivant des exemples ...

- Pour compléter ce cours, vous devez dès maintenant lire :
  - A Scala Tutorial for Java Programmers
  - An Overview of Scala
  - Scala by Example

  (Vous pouvez cliquer sur les titres, ce sont des hyperliens vers ces documents.)

Structure d'un programme Scala

- Dans un fichier `hworld.scala`, écrivez :

```
01 object HelloWorld {
02   def main(args: Array[String]) {
03     println("Hello, world !")
04   }
05 }
```

- Contrairement à Java, le nom du fichier est dissocié de son contenu.
- Ce programme définit un objet, et non une classe, nommé `HelloWorld`.

Compilation d'un programme Scala

- On compile une unité de compilation ainsi :

```
% scalac hworld.scala
```

- ou encore comme ceci :

```
% fsc hworld.scala
```

- Dans le second cas, un serveur de compilation est créé.
- Ce dernier sert à diminuer le temps de compilation.

Ce que produit le compilateur Scala

- Le compilateur Scala produit des fichiers `.class` pour la JVM.
- Dans notre exemple :

```
% ls
HelloWorld.class HelloWorld$.class hworld.scala
```

- Le fichier `HelloWorld.class` implémente l'objet Scala `HelloWorld`.
- Comme cet objet a une méthode `main (args : Array[String])`, on peut l'exécuter :

```
% scala HelloWorld
Hello, world !
```

- Cette commande lance la JVM dans l'environnement de Scala.

Conventions lexicales

- Prenons connaissance des conventions lexicales de `Scala`.

⇒ Chapitre 1 de la spécification du langage.

Unicode

- Un code source `Scala` est interprété à l'aide d'Unicode.

Identifiants

op ::= opchar {opchar}

Tirée de la spécification de Scala

```
01 scala> def foobar_is_42 = 42
02 scala> def FooBarIs42 = 42
03 scala> def ++ = ((y : Int) => y + 2)
04 scala> def ‘a surprising identifier‘ = 42
05 scala> def Σ (from : Int, stop : Int, f : Int => Int) : Int = {
06       | var accu : Int = 0
07       | for (x ← from to stop) accu += f (x)
08       | return accu
09       | }
10 Σ: (from: Int,start: Int,f: (Int) => Int)Int
11 scala> Σ (1, 100, x => x * x)
12 res2: Int = 338350
13 scala> def Σ′ (from : Int, stop : Int, f : Int => Int) : Int =
14             (0 /: (Range (from, stop, 1))) ((x, y) => x + f (y))
15 scala> Σ′ (1, 100, x => x * x)
16 res3: Int = 338350
```

Points de détails

- '$' est réservé pour les identificateurs créés par le compilateur.
- Les mots-clés sont :

```
abstract do finally import object return trait var _ : case else for
lazy override sealed try while = => ← catch extends forSome match
package super true with <: class false if new private this type yield
<% >: def final implicit null protected throw val # @
```

- Scala fait une différence entre `yield`, le mot-clé et `‘yield‘` l'identifiant (on rajoute des ‘). Ainsi, quand on veut appeler une fonction Java qui est un mot-clé Scala (comme `Thread.yield`), il suffit de l'entourer du caractère ‘.
- Le retour à la ligne et les points-virgules servent de séparateurs entre commandes. Les 3 programmes suivants sont équivalents :

```
01 x = 41
02 println (x + 1)
```

```
01 x = 41 ; println (x + 1)
```

```
01 x = 41 ;
02 println (x + 1) ;
```

(L'interprétation des retours à la ligne est assez complexe, voir la spécification, section 1.2)

Modes lexicaux

- Le compilateur Scala a deux modes de fonctionnement dans sa partie lexicale :
  - Mode « Scala » qui correspond à ce qui précède.
  - Mode « Xml » qui permet d'inclure du XML "verbatim".
- Ce dernier mode est activé par la rencontre d'un espace suivi d'une balise Xml :

```
01 scala> def b =
02       | <html>
03       | <body><p>This is an HTML snippet!</p></body>
04       | </html>
05 b: scala.xml.Elem
```

Espace de noms

- Scala utilise deux espaces de noms, un pour les types, un pour les termes. Ainsi :

```
01 scala> type t = Int
02 defined type alias t
03 scala> val t = "Do not panic!"
04 t: java.lang.String = Do not panic!
```

  définit deux noms différents.
- Chaque nom a une portée lexicale.
- Comme en Java, il est possible d'importer un nom dans l'espace lexical courant défini par un bloc, un paquetage ou un objet.

⇒ Il y a des règles de priorité pour résoudre les ambiguïtés lorsque deux noms égaux définissant des entités différentes sont importés dans la même portée lexicale.

(Voir le chapitre 2 de la spécification du langage.)

Déclarations et définitions

- Déclaration : association entre un nom et une spécification.
- Définition : association un nom, une spécification et une description.

```
Dcl        ::= 'val' ValDcl
```

```
Dcl        ::= 'val' ValDcl
```

```
Dcl        ::= 'var' VarDcl
```

- `val` donne un nom au résultat de l'évaluation d'une expression.
- `def` donne un nom à une expression.
- `var` introduit une cellule mémoire modifiable.

Exemples de définition

```
01 scala> var y = 42
02 y: Int = 42
03
04 scala> val z = y + 1
05 z: Int = 43
06
07 scala> def t = y + 1
08 t: Int
09
10 scala> t
11 res0: Int = 43
12
13 scala> y = 41
14 y: Int = 41
15
16 scala> t
17 res1: Int = 42
```

Valeurs et accesseurs par défaut

- Quand on introduit une variable à l'aide de "`var x : T = e`" :
  1. L'expression "e" de type "T" est évaluée en une valeur "v".
  2. Une cellule mémoire est créé et vaut initialement "v".
  3. Si on est dans une classe, une méthode "def x_= (v : T)" est créée.
  4. Si on est dans une classe, une méthode "def x : T" est créée.
  5. Si on est dans une classe et "e" est "_" alors "v" est une valeur par défaut valant :

0	si T est Int
0L	si T est Long
0.0f	si T est Float
0.0d	si T est Double
false	si T est Boolean
{}	si T est Unit
null	pour tous les autres types T

- Une affectation de la forme "x = e" est remplacée par un appel à la méthode "x_=". Un accès à la variable est remplacée par un appel à méthode "x".

Déclaration de fonctions


```
01 scala> def add (x : Int, y : Int) = x + y
02 add: (x: Int,y: Int)Int
03 scala> def add (x : Int, y : Int = 0) = x + y
04 add: (x: Int,y: Int)Int
05 scala> add (1)
06 res3: Int = 1
07 scala> add (2)
08 res4: Int = 2
09 scala> def fact (n : Int) = if (n == 0) 1 else n * fact (n
- 1)
10 <console> :5 : error : recursive method fact needs result type
11 scala> def fact (n : Int) : Int = if (n == 0) 1 else n *
fact (n - 1)
12 fact: (n: Int)Int
```

Déclaration de fonctions

```
Dcl          ::= 'def' FunDcl
```

```
01 scala> var x = 1; def y = x - 1
02 x: Int = 1, y: Int
03 scala> def double (v : => Int) : Int = { x = 0; return v }
04 double: (v: => Int)Int
05 scala> double (y)
06 res5: Int = -1
07 scala> x = 1
08 x: Int = 1
09 scala> val y2 = x - 1
10 y2: Int = 0
11 scala> double (y2)
12 res6: Int = 0
```

Déclaration de fonctions

```
Dcl          ::= 'def' FunDcl
```

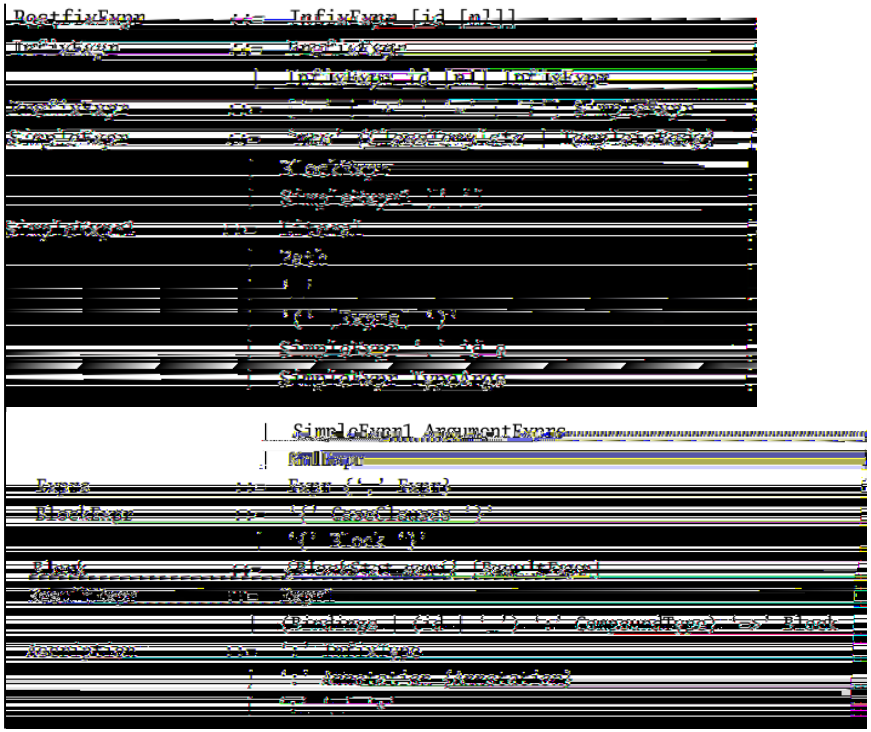
```
// "x" has type "Seq[Int]".
scala> def never_full (x : Int*) = x.length
never_full: (x: Int*)Int
scala> never_full (1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1)
res7: Int = 11
scala> never_full (1, 2, 3, 42)
res8: Int = 4
```

Définition de procédure

```
FunDcl     ::=  FunSig
FunDef     ::=  FunSig [nl] '{' Block '}'

BlockExpr  ::=  '{' Block '}'
Block      ::=  {BlockStat semi} [ResultExpr]

BlockStat  ::=  Import
            |   {Annotation} ['implicit'] Def
            |   {Annotation} {LocalMod} {Def
            |   Expr1

TemplateStat ::= Import
            |   {Annotation} {Modifier} Def
            |   {Annotation} {Modifier} Dcl
            |   Expr
```

- Les blocs sont formés de commandes et d'une expression finale.
- Si l'expression finale est omise alors on suppose qu'elle vaut `()`.
- Un bloc définit un espace lexical sans définitions multiples.

Syntaxe des expressions



- Un ensemble de constructions syntaxiques que nous allons découvrir au fil des exemples. . .
- Voyons tout de même quelques grands principes du langage.

Tout est objet donc tout est appel de méthode

- En Scala, il n'existe pas de type de base comme en Java.
- Quand on écrit "`1 + 2`", c'est la même chose que "`(1).+(2)`"

Fonction de première classe

- Une application de la forme "`f (x)`" est en fait interprétée comme si on avait écrit "`f.apply (x)`".
- Ainsi, un accès à un tableau "`t (42)`" ou encore l'application d'une fonction de première classe "`(x => x + 1) (42)`" sont des appels à la méthode `apply`.
- En particulier, écrire "`x => x + 1`", c'est la même chose qu'écrire :

```
01 new Function[Int, Int] {
02   def apply (x : Int) = x + 1
03 }
```

(Ici, on instancie la classe "`Function[Int, Int]`" en redéfinissant la méthode "`apply`" *à la volée*.)

Plan de la partie *“Construire des objets”*

Plan de la partie *“Construire des objets”*

Quelles opérations sur les objets ?

(On suppose un modèle de calcul séquentiel, *i.e.* un unique pointeur de code pendant l'exécution.)

- **"Envoyer un message M à un objet O"** :

  Donne le contrôle de l'exécution à l'objet O en lui passant le message M. Le contrôle peut ne jamais revenir et si jamais il revient, l'état de l'objet O ainsi que les états de tous les autres objets peuvent avoir été modifiés.

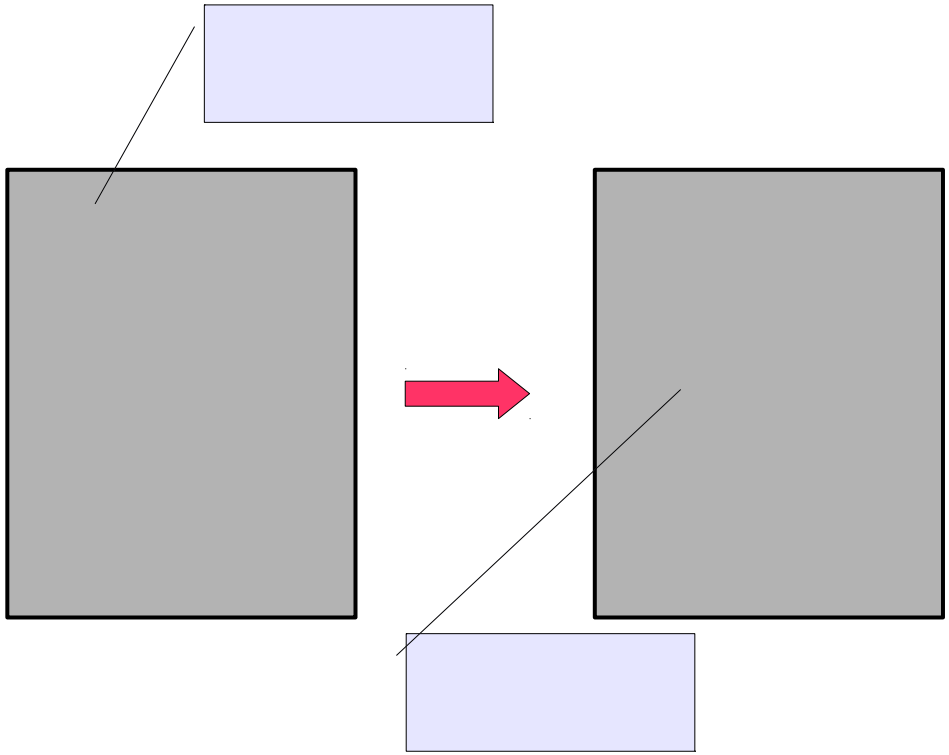
- **"Construire un objet O' par clonage d'un objet O"** :

  Un nouvel objet O est créé avec son état et son comportement propre dont les définitions initiales sont copiées de l'objet O.

- **"Modifier un objet O"** :

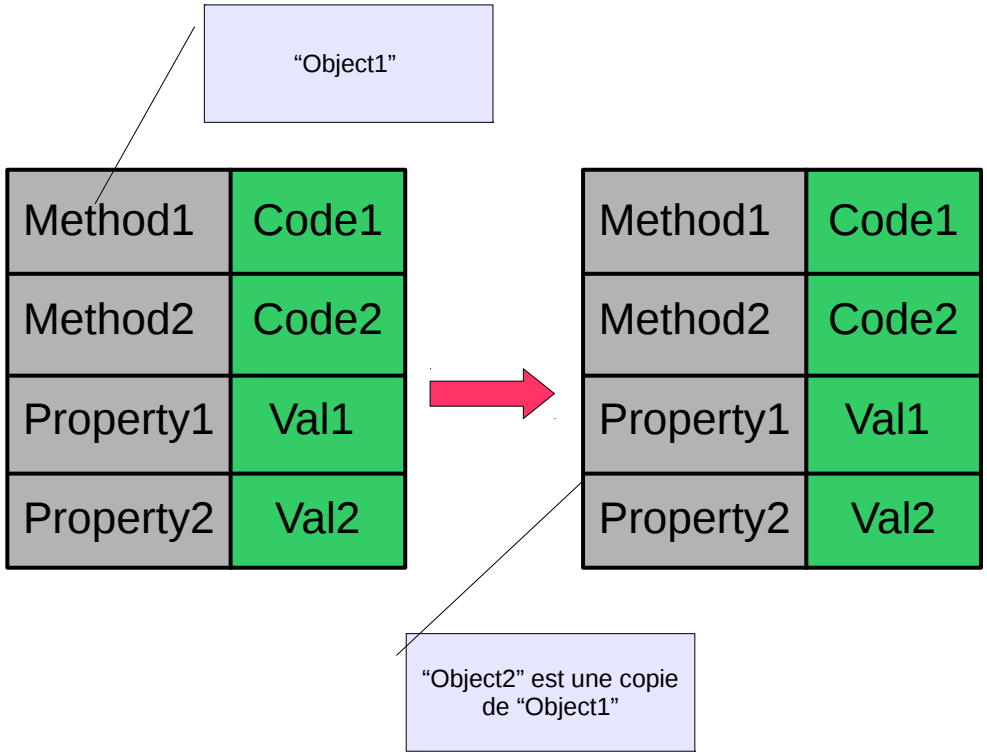
  Le domaine des propriétés et des comportements d'un objet O peut être augmenté ou diminué. La valeur de ses propriétés et la nature de ses comportements peuvent modifiées.

Construction d'objets par clonage



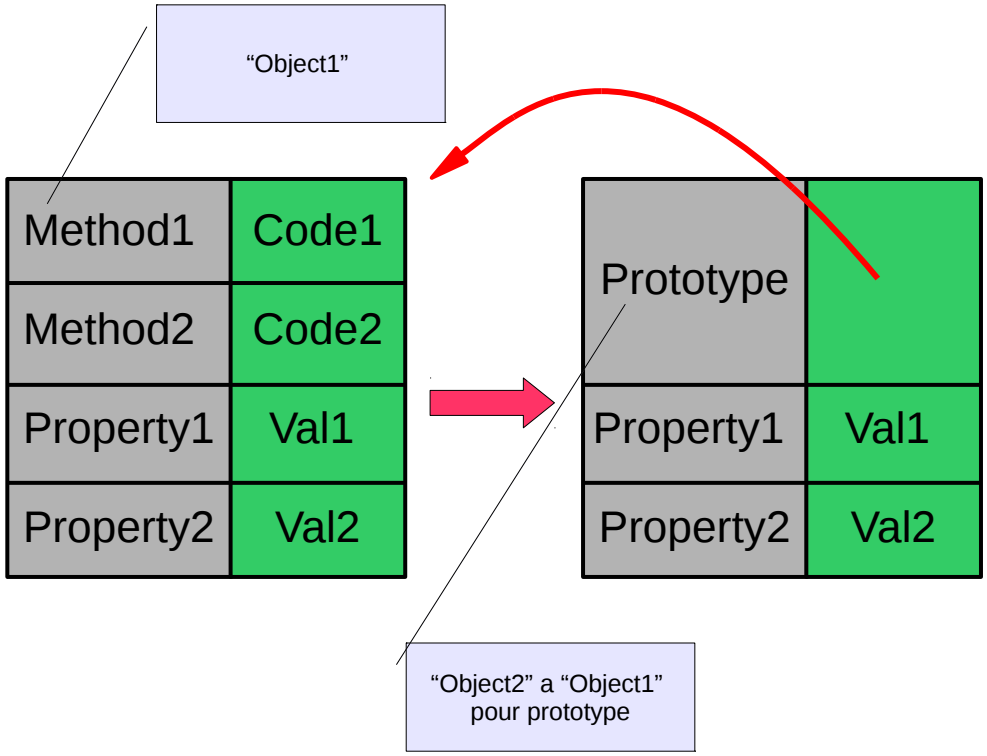
- Le code écrit pour définir `Object1` est réutilisée par `Object2`.
- `Object1` a joué le rôle de **parent** ou de **prototype** pour `Object2`.

Quelle structure de données pour les objets ?



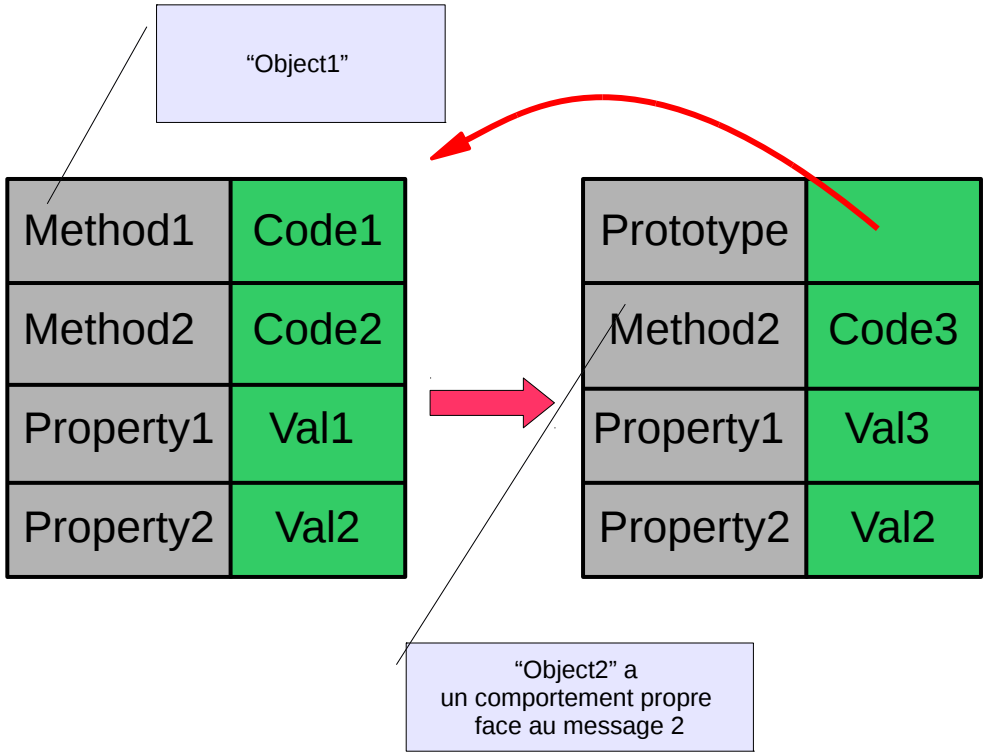
- Une implémentation possible suit un modèle à enregistrement : Ensemble de champs/slots nommés dont la valeur est **modifiable**.

Quelle structure de données pour les objets ?



- La plupart du temps, le comportement d'un objet ne diffère pas de celui du prototype qui l'a créé.

Quelle structure de données pour les objets ?



- Les comportements et l'état du second objet peuvent évoluer indépendemment du premier.

Mécanisme de résolution des appels de méthode

- En programmation orientée prototype,
  lorsqu'un message M est envoyé un objet O :
  - Si l'objet O a un comportement propre pour M alors il est effectué.
  - Sinon, l'objet O délègue le message M à son prototype.
    (et la même procédure de résolution est réitérée sur le prototype)

$\Rightarrow$ On remonte la chaîne des parents.

Langage de programmation basée sur les prototypes

- Self est un langage de programmation développé successivement par Xerox puis Sun et centré sur la notion de prototype.
- Ce langage a inspiré JavaScript.
- D'autres langages dynamiques permettent la programmation orientée prototype. (SmallTalk, Python, ...)

En Python...

```
class A:
    pass

a = A()

def B (self):
    print "bar"

a.__class__.foobar = B

a.foobar()
```

Un composant graphique en JavaScript (et

Instrumentation de code existant

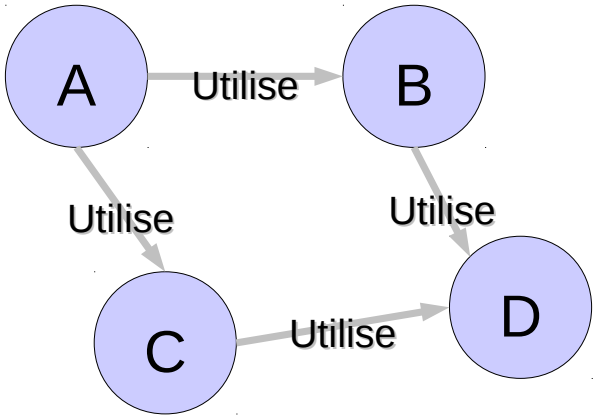
```
var oldShow = win.__proto__.onShow;
// Add a new field 'count' inside the objet 'win'.
if (win.count == undefined) {
    win.count = 0;
}
// Redefine this window's behavior wrt the event 'Show'.
win.onShow = function () {
    win.count++;
    oldShow;
}
// Also redefine the method 'Hide'.
win.onHide = function () {
  Ext.Msg.alert ('foo', 'count = ' + win.count);
}
```

Utilisation courante des aspects dynamiques

- Adaptation de code “à la volée”.
- Correction à chaud d’une erreur.
- *Mocking*

Digression : Qu'est-ce que le *Mocking* ?

- Un objet moqueur (*Mocking Object*) est un objet qui simule le comportement d'un objet réel d'une façon controlée.
- Dans le cadre d'une collaboration entre les 4 objets A, B, C et D qui suivent une relation d'utilisation mutuelle représentée par les flèches dans le diagramme suivant :



On créer un test unitaire de l'objet A en remplaçant les objets B et C par des objets moqueurs. L'objet D disparaît du test.

⇒ On réduit ainsi la portion de code validé par chaque test.

Exemple d'objet moqueur en JavaScript

Tiré de http://iainjmitchell.com/blog/?p=296

- On souhaite tester le programme suivant :

```
function MyController(display){
  return createApi();

  function createApi(){
    var privateFunctions = createPrivateApi();
    return {
      helloWorld : helloWorld
    }
  }

  function helloWorld(name){
    display.show(name);
  }
}
```

Exemple d'objet moqueur en JavaScript

Tiré de http://iainjmitchell.com/blog/?p=296

- Plutôt que de le tester avec une instance réelle de `display`, on utilise un objet moqueur qui simule `display` :

```
MyControllerTest.prototype.test_helloWorld_display_show_called
= function() {
    jack(function(){
        //SET-UP
        var mockDisplay = jack.create("display", ["show"]);
        var target = new MyController(mockDisplay);

        // Check that "display.show" is called only once.
        jack.expect("display.show").once();

        //EXECUTE
        target.helloWorld("Hello");
    });
}
```

Pointeur en avant...

- Nous reviendrons sur les environnements de définition de tests unitaires par objets moqueurs dans le cours sur la vérification de programmes objets.

Construction incrémentale d'une méthode incrémentale

Méthode de développement « objet » (v0.2, code "Extreme Test Driven")

1. Brainstorming :
   (i) Énumérer les objets du problème.
   (ii) Expliciter des situations de collaboration entre ces objets pour résoudre un problème.
   (iii) Hiérarchiser ces situations et ces objets pour faire apparaître un ensemble « noyau » minimal de fonctionnalités importantes.
2. Écrire un programme qui spécifie suffisamment ces objets pour décrire les situations de collaboration visant à réaliser les fonctionnalités importantes sous la forme de tests exécutables. Ces tests doivent être **unitaires**.
3. Implémenter le corps des méthodes, **objet par objet**, en validant les tests.

Simulation de la programmation orientée prototype en Scala

```
01 type MsgLabel = Object
02 type Data = Any
03 type DynResponse = PartialFunction[MsgLabel, Data => Unit]
```

Simulation de la programmation orientée prototype en Scala

```
01  abstract class AbsDynObject {
02    var dispatcher : DynResponse
03
04    def mkMethod (msg : MsgLabel, body : Data => Unit) : DynResponse =
05      { case rmsg if (msg == rmsg) => body }
06
07    def addMethod (msg : MsgLabel, body : Data => Unit) = {
08      dispatcher = (mkMethod (msg, body) orElse dispatcher)
09    }
10
11    def removeMethod (d : MsgLabel) : Unit = {
12      object newdispatcher extends DynResponse {
13        def isDefinedAt (msg : MsgLabel) =
14          !(d == msg)    dispatcher.isDefinedAt (msg)
15        def apply (msg : MsgLabel) =
16          if (d == msg) error ("Undefined") else dispatcher (msg)
17      } ; newdispatcher
18    }
19
20    def receives (d : MsgLabel) : Data => Unit = dispatcher (d)
21
22    override def clone = {
23      val that = this ; new AbsDynObject { var dispatcher = that.dispatcher }
24    }
25 }
```

Simulation de la programmation orientée prototype en Scala

```
01 object empty extends AbsDynObject {
02   override var dispatcher : DynResponse =
03     new DynResponse {
04       def isDefinedAt (dmsg : MsgLabel) =
05         false
06       def apply (dmsg : MsgLabel) =
07         error ("No method in empty object.")
08   }
09 }
```

Plan de la partie *“Construire des objets”*

Simulation de la programmation orientée prototype en Scala

```
01 val o = empty.clone
02 object Show extends MsgLabel
03 object Hide extends MsgLabel
04 o.addMethod (Show, { case x : Int => println (x) })
05 (o receive Show) (42) // Success!
06 (o receive Show) (true) // Runtime error!
07 (o receive Hide) () // Runtime error!
```

Ajout du champ prototype

```
01 def delegates (msg : MsgLabel) : Data => Unit =
02   { d: Data =>
03     dispatcher (Prototype) match {
04       case (prototype : (Data => Unit)) => prototype (msg, d)
05     }
06   }
07
08 def receives (d : MsgLabel) : Data => Unit =
09   if (! dispatcher.isDefinedAt (d))
10     this delegates d
11   else
12     dispatcher (d)
13
14 override def clone = {
15   val that = this
16   new AbsDynObject {
17     override var dispatcher : DynResponse = no_response
18     addMethod (Prototype,
19                { case (msg : MsgLabel, data : Data) =>
20                  (that receives (msg)) (data)
21                })
22 }
```

Utilisation du champ prototype

```
01 val o = empty.clone
02 object Show extends MsgLabel
03 object Hide extends MsgLabel
04 o.addMethod (Show, { case x : Int => println (x) })
05 (o receives Show) (42)
06
07 val s = o.clone
08 s.addMethod (Hide, { case x : Int => println ("-" + x) })
09 (s receives Show) (42)
10 (s receives Hide) (42)
```

Quelques remarques

- Une implémentation efficace de la résolution des méthodes utiliserait la classe `HashMap` de Scala par exemple.

⇒ Exercice !

- *Comment associer un type aux données attachées à un message ?*

  On pourrait garantir qu'une fois l'existence d'une méthode dans un objet vérifiée, le passage d'arguments à cette méthode ne peut que réussir.

⇒ Exercice !

(Beaucoup plus di□cile, et nous verrons sa solution dans le cours sur le typage.)

La solution en O'Caml

tirée de http://eigenclass.org/R2/writings/typesafe-prototype-based-ocaml

```
module type PROPERTYLIST =
sig
  type t
  val create : unit → t
  type α property = (t → α → unit) × (t → α option)
  val new_property : unit → α property
end
```

La solution en O'Caml

tirée de http://eigenclass.org/R2/writings/typesafe-prototype-based-ocaml

```
module MappedList : PROPERTYLIST =
struct
  type t = (int, unit → unit) Hashtbl.t
  type α property = (t → α → unit) × (t → α option)

  let create () = Hashtbl.create 13

  let new_id : unit → int =
    let id = ref 0 in
      fun () → incr id ; !id

  let new_property () =
    let id = new_id () in
    let v = ref None in
    let set t x = Hashtbl.replace t id (fun () → v := Some x) in
    let get t =
      try
        (Hashtbl.find t id) () ;
        match !v with
          | Some _ as s → s
          | None → None
      with Not_found → None
    in
      (set, get)
end
```

La solution en O'Caml

tirée de http://eigenclass.org/R2/writings/typesafe-prototype-based-ocaml

```
module P = struct
  include MappedList
  let get t (set, get) = get t
  let set t (set, get) x = set t x
end

type obj = { dict : P.t ; mutable parent : obj option }

let clone x = { dict = P.create () ; parent = Some x }
```

La solution en O'Caml

tirée de http://eigenclass.org/R2/writings/typesafe-prototype-based-ocaml

```
(* A base object *)
let obj = { dict = P.create () ; parent = None }

(* A shortcut to define a slot in an objet. *)
let set obj selector v = P.set obj.dict selector v

exception Method_not_found
```

La solution en O'Caml

tirée de http://eigenclass.org/R2/writings/typesafe-prototype-based-ocaml

```
let (%) t meth =
  let rec dispatch t meth self =
  match P.get t.dict meth with
    | Some f → f self
    | None → match t.parent with
          | Some p → dispatch p meth self
          | None → raise Method_notfound
  in dispatch t meth t

let (!!) meth obj = obj%meth
```

La solution en O'Caml

tirée de http://eigenclass.org/R2/writings/typesafe-prototype-based-ocaml

```
type α selector = α P.property

let new_selector = P.new_property

let define t f =
  let selector = new_selector () in
    set t selector f;
    selector
```

La solution en O'Caml

tirée de http://eigenclass.org/R2/writings/typesafe-prototype-based-ocaml

```
let duck = clone obj
let quack = define duck (fun self → print_endline "QUACK !")
let _ = duck % quack
```

Plan de la partie *“Construire des objets”*

Langages **sans** vérification statique

- Dans un langage orienté prototype (LOP) :

« Il n'y a **aucune garantie**
sur la réussite d'un appel de méthode. »

Comment écrire un programme robuste en LOP ?

- Pour s'assurer que son programme écrit en LOP est robuste, c'est-à-dire qu'il n'échoue pas de façon non prévue par la spécification, il existe différentes méthodes :
  - La discipline de programmation :
    - **Vérifier manuellement** qu'il n'a pas fait de faute de frappe (!) dans les identificateurs des méthodes au niveau de leurs définitions et de leurs appels.
    - **Se convaincre** qu'en tout point du programme, que la forme des objets apparaissant e□ectivement à l'exécution est celle qu'il attend.
  - La programmation défensive :
    - **Supposer systématiquement** qu'un appel de méthode peut échouer.
    - **Instrumenter** les objets inconnus pour les adapter au contexte.

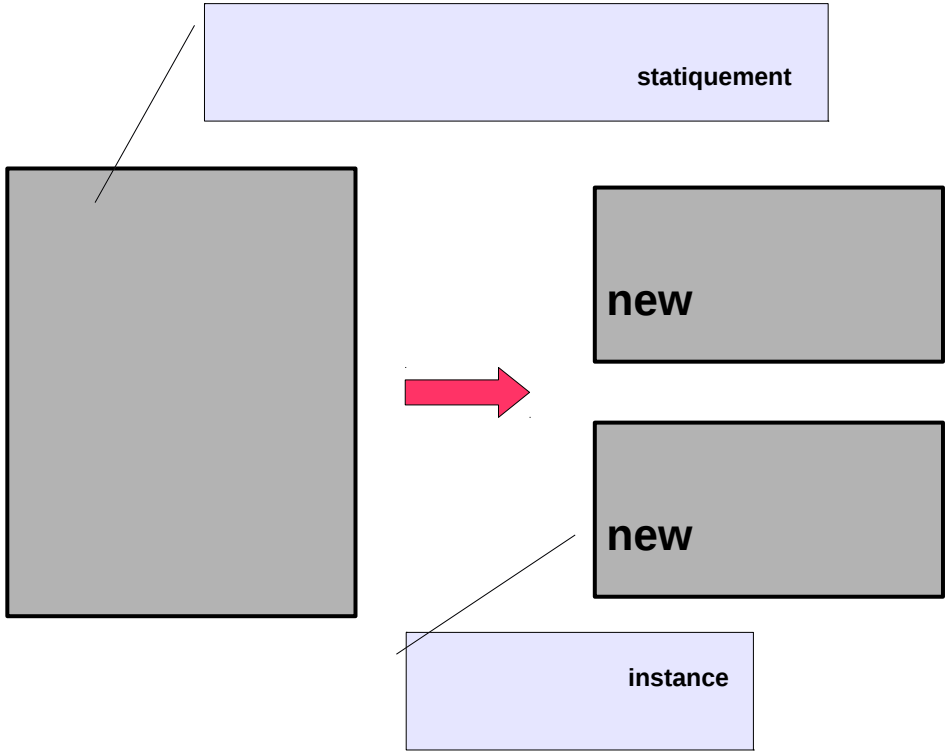
Soyons sérieux. . .

The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.
– Edsger Dijkstra

Plan de la partie *“Construire des objets”*

Plan de la partie *“Construire des objets”*

Principe



Définitions

Classe

Une classe est une construction logicielle de seconde classe[1] servant de moule (de patron) pour produire des objets.

(Nous allons préciser cette définition dans la suite de ce cours.)

Instance

Une instance est un objet qui a été engendré par une classe. Une instance a un état propre mais partage le même comportement que toutes les instances engendrées par cette classe.

1. On ne peut, en général, pas calculer de classe pendant l'exécution du programme.

Exemple

```
class positive = {
  var x = 0
  def incr { x = x + 1 }
  def decr { if (x > 0) x = x - 1 }
}
val a = new positive
a.incr
val b = new positive
```

Comment définir le comportement d'une classe d'objets ?

- L'ensemble des membres d'une classe définit le comportement de ses instances. Ce comportement est capturé par leurs spécifications.
- La 2nde partie de POCA abordera la spécification des objets.
- Pour le moment, contentons-nous de comprendre comment définir ces membres en `Scala`.

Les membres de classe en Scala

- En Scala, nous avons déjà vu qu'il existait deux grandes familles de membres de classes : les déclarations et les définitions.
- Plus précisément, les sections 5.1, 6.25 de la spécification du langage, nous fournissent la grammaire suivante :

```
        TmplDef   ::=   ['case'] 'class' ClassDef
                   |    ...
       ClassDef   ::=   id [TypeParamClause] {Annotation}
                        [AccessModifier] ClassParamClause ClassTemplateOpt
                   |    ...
  ClassTemplate   ::=   [EarlyDefs] ClassParents [TemplateBody]
   TemplateBody   ::=   [nl] '{' [SelfType] TemplateStat {semi TemplateStat}'}'
   TemplateStat   ::=   Import
                   |    [Annotation][Modifier]Def
                   |    [Annotation][Modifier]Decl
                   |    Expr
                   |
```

(N'oubliez pas que vous devez savoir lire ces spécifications de grammaire.)

Les membres de classe en Scala : Expression

```
scala> class a
defined class a
scala> import scala.util.Random
import scala.util.Random
scala> class b { var x = Random.nextInt (); var y = Random.nextInt ();
     | if (x > y) { println ("Swapping !"); val z = x; x = y; y = z } }
defined class b

scala> val bi = new b
bi: b = b@2a717ef5

scala> val ci = new b
Swapping !
ci: b = b@31602bbc
```

- Au moment de l'instanciation, l'expression est évaluée.
- Le résultat de cette évaluation est jeté.

Les membres de classe en Scala : val/def

```
scala> class c { def x = Random.nextInt () }
defined class c

scala> val ci = new c
ci: c = c@2259e205

scala> ci.x
res5: Int = 759312157

scala> ci.x
res6: Int = 2102202992

scala> class f { val x = Random.nextInt () }
defined class f

scala> val fi = new f
fi: f = f@1896d2c2

scala> fi.x
res7: Int = -2036054592

scala> fi.x
res8: Int = -2036054592
```

Les membres de classe en Scala : val/def

```
scala> class h { def new_name = {
    var count = 0; (x : String) => { count += 1; x + count }
} }
scala> class i { val new_name = {
    var count = 0; (x : String) => { count += 1; x + count }
} }
scala> val hi = new h
hi: h = h@4183aedf

scala> val ii = new i
ii: i = i@77be91c8

scala> hi.new_name ("foo")
res15: java.lang.String = foo1

scala> hi.new_name ("foo")
res16: java.lang.String = foo1

scala> ii.new_name ("foo")
res17: java.lang.String = foo1

scala> ii.new_name ("foo")
res18: java.lang.String = foo2
```

Définition sans argument formel

```
scala> class a { def f () = 1 }
defined class a

scala> val x = new a
x: a = a@36f72f09

scala> x.f
res2: Int = 1

scala> x.f ()
res3: Int = 1

scala> class a { def f = 1 }
defined class a

scala> val x = new a
x: a = a@369133f6

scala> x.f
res4: Int = 1

scala> x.f ()
<console>:8: error: x.f of type Int does not take parameters
        x.f ()
```

Définition sans argument formel

```
scala> class a { val f = () => { println ("foo"); 1 } }
scala> class b { def f () = { println ("foo"); 1 } }
scala> class c { def f = { println ("foo"); 1 } }

scala> def g (x : => Int) = x + x
g: (x: => Int)Int

scala> val x = new a; val y = new b; val z = new c
x: a = a@1922e15b
y: b = b@40f92a41
z: c = c@264532ba

scala> g (y.f)
foo
foo
res22: Int = 2

scala> g (z.f)
foo
foo
res23: Int = 2

scala> g (x.f)
<console>:12: error: type mismatch;
 found : () => Int
 required: Int
        g (x.f)
```

Les membres de classe en Scala : le modificateur lazy

```
scala> def verbose_random = {
| println ("Random is called now.");
| Random.nextInt
| }
verbose_random: Int

scala> class g { lazy val x = verbose_random }
defined class g

scala> val gi = new g
gi: g = g@5115a298

scala> gi.x
Random is called now.
res12: Int = 1605365286

scala> gi.x
res13: Int = 1605365286
```

- L'évaluation paresseuse permet de ne calculer une valeur qu'au besoin et de ne pas la recalculer si on en a besoin de nouveau.

⇒ Utile aussi pour initialiser des composants mutuellement récursifs !

Les membres de classe en Scala : les classes imbriquées

```
scala> class j {
  var x = 0;
  class inner_j { def y = x * 2 + 1 }
}
defined class j

scala> val ji = new j
ji: j = j@746d1683

scala> val jji = new ji.inner_j
jji: ji.inner_j = j$inner_j@165bece2

scala> jji.y
res19: Int = 1

scala> ji.x = 2

scala> jji.y
res20: Int = 5
```

Classes imbriquées

Classe imbriquée

Une classe est imbriquée si le comportement commun de ses instances **dépend d'un objet**.

- Exemples :
  - Les nœuds et les arêtes d'un graphe dépendent de ce graphe.
  - Les clés d'un dictionnaire dépendent de ce dictionnaire.
  - Les items d'une table de Base de Données dépendent de cette BD.

Exemple de classe imbriquée inspiré de http://scala-lang.org/node/115

```
01 class Graph {
02   var nodes: List[Node] = Nil
03   class Node {
04     var connectedNodes: List[Node] = Nil
05     def +-> (node: Node) {
06       if (connectedNodes.find(node.equals).isEmpty) {
07         connectedNodes = node :: connectedNodes
08       } else { println ("Already connected.") }
09     }
10     nodes = this :: nodes
11   }
12 }
```

Exemple de classe imbriquée inspiré de http://scala-lang.org/node/115

```
01 scala> val g = new Graph
02 g: Graph = Graph@20442c19
03
04 scala> val n1 = new g.Node
05 n1: g.Node = Graph$Node@36edc33d
06
07 scala> val n2 = new g.Node
08 n2: g.Node = Graph$Node@1a6313e1
09
10 scala> val n3 = new g.Node
11 n3: g.Node = Graph$Node@f5e12
12
13 scala> n1 +→ n2
14
15 scala> n3 +→ n1
16
17 scala> n1 +→ n2
18 Already connected.
```

Constructeur par défaut

- Les arguments d'une classe définissent un constructeur par défaut.
  (Voir la section 5.3 de la spécification.)
- Exemple :

```
01 scala> class k (x : Int, s : String) {
02       | val y = x
03       | val t = s
04       | }
05 defined class k
06
07 scala> val ki = new k (42, "bar")
08 ki: k = k@6a22a6bb
09
10 scala> ki.t
11 res27: String = bar
```

Constructeur par défaut

- On peut factoriser la définition des membres et des arguments du constructeur par défaut lorsqu'ils sont confondus.
- Exemple :

```
01 scala> class l (val x : Int, val s : String)
02 defined class l
03
04 scala> val li = new l (42, "bar")
05 li: l = l@4fae9ef9
06
07 scala> li.s
08 res29: String = bar
```

Constructeurs auxiliaires

- Il parfois utile de fournir plusieurs façons d'instancier une classe.
- En Scala, il suffit de définir des constructeurs auxiliaires.
- Ce sont des méthodes dont le corps doit être soit :
  - un appel à un autre constructeur avec la syntaxe "`this (e1, ..., en)`" ;
  - ou bien un bloc qui débute par un appel à un autre constructeur.
- Pour éviter d'éventuelle non-terminaison de l'instanciation, le compilateur nous force, dans le corps d'un constructeur auxiliaire, à ne faire appel qu'à des constructeurs qui le précèdent dans la définition de la classe.

Constructeurs auxiliaires

```
01 scala> class LinkedList[A] () {
02     var head : A = _
03     var tail : LinkedList[A] = null
04     def isEmpty = tail != null
05     def this (xs: List[A]) = {
06       this ();
07       if (xs.length != 0) {
08           this.head = xs.head
09           this.tail = new LinkedList[A] (xs.tail)
10         }
11     }
12     def this (xs: A*) = this (xs.toList)
13 }
14 defined class LinkedList
15
16 scala> val x = new LinkedList (1, 2, 3, 4)
17 x: LinkedList[Int] = LinkedList@33ef5e7d
```

Ordre d'initialisation d'une classe

```
01 scala> class m {
02     var x = 0;
03     def this (y : Int) = { this (); x = y };
04     println (x)
05 }
06 defined class m
07
08 scala> val mi = new m
09 0
10 mi: m = m@5081e9d3
11
12 scala> val mi = new m (42)
13 0
14 mi: m = m@72a32604
```

Classe de cas

- Scala offre une facilité pour définir des classes aidant au **raisonnement par cas** : les classes de cas.
- En préfixant une définition de classe par la mot-clé `case`, la définition est étendue automatiquement de la façon suivante :
  - Tous les arguments formels du constructeur par défaut sont préfixés par `val`. (Sauf si un modificateur `var` est déjà présent.)
  - Des méthodes nommées `apply`, `unapply` et `copy` sont rajoutées si elles n'existent pas déjà :
    - `apply` appelle le constructeur par défaut pour construire une nouvelle instance de la classe.
    - `unapply`, si son argument n'est pas `null`, elle produit un n-uplet contenant les arguments passés aux constructeurs par défaut au moment de son instanciation.
    - `copy` qui recopie tous (ou une partie) des constituants de `this`.
  - Les méthodes nommées `toString`, `hashCode`, et `Equals` (contenues dans tout objet en Scala) sont redéfinies si elles n'existent pas déjà.
    - `toString` produit une chaîne de caractère de la forme "X (s1, ..., sN)" où "X" est le nom de la classe et les "sI" sont produits par la méthode `toString` des constituants de `this`.
    - `hashCode` calcule une clé de hachage qui commute avec l'égalité.
    - `equals` implémente une égalité structurelle.

Classe de cas : exemple

Écrire ceci :

```
01 case class Pair (x : Int, y : Int)
```

est équivalent à :

```
01 class Pair (val x : Int, val y : Int) {
02   override def hashCode = x.hashCode * 31 + y.hashCode
03   // Scala compiler probably uses another hash function,
of course.
04   override def equals (that : Any) = {
05     that match {
06       case cthat : Pair => x == cthat.x    y == cthat.y
07       case _ => false
08     }
09   }
10   override def toString = "Pair(" + x.toString + ", " + y.toString + ")"
11   def copy (x : Int = this.x, y : Int = this.y) = new Pair (x, y)
12 }
13
14 object Pair {
15   def apply (x : Int, y : Int) = new Pair (x, y)
16   def unapply (that : Pair) = if (that eq null) None else Some (that.x, that.y)
17 }
```

Analyse de motifs

- Une expression qui analyse une valeur par motifs est de la forme :

```
e match { case p1 [garde1] => e1 ; ... ; case pN [gardeN] => eN}
```

- Les `pI` sont des motifs (dont nous allons voir la syntaxe plus loin).
- Les gardes optionnelles sont de la forme « `if e` » où `e` est une expression booléenne.
- L'évaluation d'une analyse par motifs consiste à :
  1. Évaluer « `e` » en une valeur v.
  2. Analyser v successivement de gauche à droite par un motif `pI` :
     - Si v coïncide avec le motif `pI` et que la garde vaut `true`, on nomme les constituants de v à l'aide des noms de `pI` et on évalue `eI` dans ce contexte.
     - Sinon, on passe au motif suivant.

- L'expressivité du langage de motif de Scala est très grande.

⇒ La définition de « v coïncide avec le motif `p` » est complexe.

Motifs nommant

```
01 def identity (x : Int) =
02   x match { case y => y }
```

Motifs litéraux et motifs par défaut

```
01 def translate (word : String) =
02   word match {
03     case "hello" => "bonjour"
04     case "world" => "monde"
05     case _ => "\"" + word + "\" (in english)"
06   }
07
08 def translate_sentence (sentence : String) =
09   sentence.split (" ") map translate reduceLeft (_ + " " + _)
```

Motifs d'identifiant

- Le programme suivant est bien sûr incorrect :

```
01 def are_equals (x : Int, y : Int) =
02   x match { case y => true ; case _ => false }
```

('y' est juste le nom donné à x)

- Par contre, celui-ci est correct !

```
01 def are_equals (x : Int, y : Int) =
02   x match { case `y` => true ; case _ => false }
```

('y' est entouré du symbole 'apostrophe arrière' (*backquote* en anglais).)

- Ici, le motif fait référence à l'identifiant "y". Il coïncide avec "x" lorsque « `x == y` » vaut `true`.

Exemples de motifs construits

```
01 scala> class o (var x : Int)
02 defined class o
03 scala> object o {
04       | def unapply (that : o) : Option[Int] = if (that == null) None else Some (that.x)
05       | }
06 defined module o
07 scala> def g (x : o) = x match { case o (y) => y }
08 g: (x: o)Int
09 scala> g (new o (42))
10 res0: Int = 42
11 scala> def h (x : Any) = x match { case o (y) => y; case _ => 0}
12 h: (x: Any)Int
13 scala> h ("Foo")
14 res4: Int = 0
15 scala> h (new o (42))
16 res5: Int = 42
17 scala> case class IndexedString (s : String, i : Int)
18 defined class IndexedString
19 scala> def getIndexes (s : List[IndexedString]) =
20          s map { x => x match { case IndexedString (_, idx) => idx } }
21 getIndexes: (s: List[IndexedString])List[Int]
```

Fonction anonyme définie par cas

- On peut remplacer :

```
x => x match { case p1 => e1 ; ... ; case pN => eN }
```

par

```
{ case p1 => e1 ; ... ; case pN => eN }
```

Autres types de motifs

- Il existe d'autres types de motifs, que nous verrons un peu plus tard.

Encapsulation

Encapsulation

On appelle mécanismes d'encapsulation l'ensemble des procédés visant à **cacher** la logique interne d'un objet, c'est-à-dire la représentation de son état interne ainsi que le code qui ne fait pas partie de son interface.

- L'encapsulation minimise le **couplage** entre les implémentation des objets : c'est une propriété améliorant la modularité des composants.

Une classe mal encapsulée

```
01 scala> class positiveInteger (var x : Int) {
02       | if (x < 0) error ("Expecting a positive integer!")
03       | }
04 defined class positiveInteger
05 scala> val p = new positiveInteger (42)
06 p: positiveInteger = positiveInteger@5a6b258d
07 scala> val p = new positiveInteger (-42)
08 java.lang.RuntimeException: Expecting a positive integer!
09 scala> p.x
10 res11: Int = 42
11 scala> p.x = -42
12 scala> p.x
13 res12: Int = -42
```

Une classe bien encapsulée

```
01 class positiveInteger (x0 : Int) {
02   def checked (nx : Int) : Int = {
03     if (nx < 0) error ("Expecting a positive integer!");
04     return nx
05   }
06   private var internal_x : Int = checked (x0)
07   def x : Int = internal_x
08   def x_= (nx : Int) = { this.internal_x = checked (nx) }
09 }
10
11 scala> val p = new positiveInteger (42)
12 p: positiveInteger = positiveInteger@1fe26d9b
13
14 scala> p.x = -42
15 java.lang.RuntimeException: Expecting a positive integer!
```

Un mécanisme d'encapsulation : le mot-clé `private`

- En `Scala`, le mot-clé `private` restreint l'accès à un membre d'une classe C aux définitions de cette même classe et à un éventuel objet du même nom partageant la même unité de compilation que C.

$\Rightarrow$ Dans le vocabulaire de `Scala`, on l'appelle l'objet compagnon de C.

Un objet compagnon

```
01 object positiveIntegerPackage {
02   class positiveInteger (x0 : Int) {
03     private def checked (nx : Int) : Int = {
04       if (nx < 0) error ("Expecting a positive integer!");
05       return nx
06     }
07     private var internal_x : Int = checked (x0)
08     def x : Int = internal_x
09     def x_= (nx : Int) = { this.internal_x = checked (nx) }
10   }
11
12   object positiveInteger {
13     def incr (p : positiveInteger) = p.x += 1
14     def decr (p : positiveInteger) = p.x = p.checked (p.x - 1)
15   }
16 }
```

Utilisation étendue du mot-clé `private`

- `Scala` étend la notion de membre privé en notant :
  - `private[this]` pour restreindre l'accès d'un membre à l'instance courante uniquement. On interdit l'accès à toutes les autres instances même si elle partage la même classe.
    (Par contre, l'objet compagnon a toujours les droits d'accès.)
  - `private[C]` où C est un identifiant, qui peut-être une classe ou un objet englobant.

Droit d'accès privé étendu

```
01 class Graph {
02   private var nodes: List[Node] = Nil
03   class Node {
04     // The graph is the only object
05     // allowed to modify the following member.
06     private[Graph] var connectedNodes: List[Node] = Nil
07     def +-> (node: Node) {
08       if (connectedNodes.find(node.equals).isEmpty) {
09         connectedNodes = node :: connectedNodes
10       } else { println ("Already connected.") }
11     }
12     nodes = this :: nodes
13   }
14 }
```

Mécanisme d'encapsulation : Constructeur privé

- Il est parfois utile de **forcer** l'utilisation d'une fonction pour construire une instance de classe.
- Exemple :
  - Pour ne pas divulguer le constructeur principal d'une classe.
  - Pour ne pas divulguer le lien entre l'interprétation des arguments à fournir pour construire une instance et le processus concret de construction de l'instance.
- On utilise la syntaxe :

```
class a (x1: T1, ..., xN: TN) private { ... }
```

⇒ Mais alors comment écrire la fonction d'instanciation personnalisée ?

Exemple de constructeur privé

```
01 import scala.collection.mutable.Map
02 import scala.collection.mutable.ListMap
03 import scala.collection.mutable.HashMap
04
05 class Dictionary[Key, Value] private (val data : Map[Key, Value]) {
06    def add (k : Key, v : Value) : Unit = data (k) = v
07    def get (k : Key) : Value = data (k)
08 }
09
10 object Dictionary {
11    val max_for_list = 10
12    def create [Key, Value](maxEntries : Int) : Dictionary[Key, Value] = {
13       val data : Map[Key, Value] =
14          if (maxEntries < max_for_list)
15             new ListMap ()
16          else
17             new HashMap () ;
18       return (new Dictionary[Key, Value] (data))
19    }
20
```

Plan de la partie *“Construire des objets”*

Typage

- Quand une classe A est définie, un type A est engendré.

Type

Un type est une information **statique** qui représente la **forme du résultat de l'évaluation** d'une expression.

Typeur

Il existe un algorithme qui calcule le type d'une expression si il existe.

Langage typé

Un langage est typé si il existe un typeur pour ses expressions.

Garanties statiques

- Quand une expression est accepté par le typeur de `Scala` on a la garantie que son évaluation se déroulera sans erreur.

Garanties statiques

- Quand une expression est accepté par le typeur de `Scala` on a la garantie que son évaluation se déroulera sans erreur.

⇒ De quel type d'erreur parle-t-on ?

- Dans un langage objet, les erreurs d'exécution sont essentiellement des accès à des membres d'objet non existant.

Garanties statiques

- Quand une expression est accepté par le typeur de `Scala` on a la garantie que son évaluation se déroulera sans erreur.

⇒ De quel type d'erreur parle-t-on ?

- Dans un langage objet, les erreurs d'exécution sont essentiellement des accès à des membres d'objet non existant.

⇒ Quelles erreurs ne sont pas capturées par le typeur ?

- Les erreurs de conception du programme (un invariant non respecté par exemple) ne sont pas capturés par le typeur. En général, ce type d'erreurs provoque en général des exceptions.

Un langage vérifié dynamiquement : Python

```
  > class A:
... def m (self):  self.h
...
  > a = A ()
  > a.m
<bound method A.m of <__main__.A instance at 0x9609fcc
  > a.m ()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in m
AttributeError:  A instance has no attribute 'h'
```

- La définition de la classe A est acceptée même si il n'existe pas de méthode *h* dans la classe A !

La même session en Scala

```
scala> class A {
     | def m = this.h
     | }
<console>:6: error: value h is not a member of A
          def m = this.h
```

Dynamicité dans les langages statiquement typés

- Parfois, on ne peut pas connaître le type d'une valeur.
- Par exemple, lorsque l'on récupère une valeur via le réseau.
- Cependant, il existe des méthodes pour effectuer des tests sur ses valeurs et reconstruire leur type statique, à condition qu'il fasse partie des types déjà connus au moment de la compilation.

⇒ Nous reviendrons sur ces méthodes.

- Les langages statiquement typés ont plus de difficulté à égaler la souplesse des langages vérifiés dynamiquement quand on doit générer du code et des types totalement nouveaux durant l'exécution.

⇒ Il existe cependant des langages où le typage se fait en plusieurs étapes, en alternant évaluation et typage à l'exécution.

Limite du typage statique

- Il existe des limites au typage statique inhérente à l'**indécidabilité** des propriétés (intéressantes) des programmes.
- Par exemple, on ne peut pas écrire un programme qui décide si un autre programme termine ou non.
- Un autre exemple :

```
scala> (if (1 == 1) 1 else List(1, 2)) - 2
<console>:6: error: value - is not a member of Any
        (if (1 == 1) 1 else List(1, 2)) - 2
```

⇒ Ce programme est pourtant correct !

Les types de Scala

- Plus le langage des types est expressif,
  et plus le typage accepte de programmes corrects.
- Les types de Scala sont très généraux et expressifs.

⇒ Nous allons maintenant voir quelques-unes de leurs caractéristiques
et nous en verrons d'autres petit à petit dans les prochains cours.

Typage nominal (à la Java)

```
01 scala> class Human {
02       | def run : Unit = println ("I am human and I am running.")
03       | }
04 scala> class Software {
05       | def run : Unit = println ("A software is running.")
06       | }
07 scala> def athletic_competition (competitors : List[Human]) {
08       | competitors foreach (_.run)
09       | }
10 athletic_competition: (competitors: List[Human])Unit
11 scala> val human_list = List (new Human, new Human)
12 human_list: List[Human] = List(Human@14c92a7, Human@3040c5)
13 scala> val software_list = List (new Software, new Software)
14 software_list: List[Software] = List(Software@11□1b8, Software@367b19)
15 scala> athletic_competition (human_list)
16 I am human and I am running.
17 I am human and I am running.
18 scala> athletic_competition (software_list)
19 <console>:10: error: type mismatch;
20  found : List[Software]
21  required: scala.List[Human]
22              athletic_competition (software_list)
```

Typage structurel (à la O'Caml)

```
01 scala> def run_many_times (n : Int, x : { def run : Unit }) {
02       | for (i ← 1 to n) x.run
03       | }
04 run_many_times: (n: Int,x: AnyRef{def run: Unit})Unit
05
06 scala> run_many_times (3, new Human)
07 I am human and I am running.
08 I am human and I am running.
09 I am human and I am running.
10
11 scala> run_many_times (3, new Software)
12 A software is running.
13 A software is running.
14 A software is running.
```

Typage structurel

Typage structurel

Un système de type est structurel si la relation de compatibilité entre les types est uniquement basé sur leur structure.

- Le système de type de la partie objet d'O'Caml est structurel.

Typage structurel en O'Caml

```
class human = object
  method run = Printf.printf "I am a human and I am running\n"
end

class software = object
  method run = Printf.printf "A software is running\n"
end

let athletic_competition (competitors : human list) =
  List.iter (fun x → x#run) competitors

let human_list = [new human; new human]
let software_list = [new software; new software];;

athletic_competition human_list ;;
(** L'expression suivante est rejetée par Scala, pas par OCaml. *)
athletic_competition software_list ;;

let rec run_many_times n x =
  if n = 0 then () else (x#run; run_many_times (n - 1) x)

run_many_times 3 (new human)
run_many_times 3 (new software)
```

Typage nominal ou structurel ?

- Le typage structurel apporte plus de flexibilité et permet d'écrire des programmes plus généraux.
- Le typage nominal permet de renforcer la sémantique associée aux types et donc de capturer une spécification plus précise des programmes.

⇒ Comme nous l'avons vu, `Scala` autorise ces deux styles

⇒ Dans le cas où l'on peut intégrer des spécifications aux types d'objet sous la forme d'assertions logiques, le typage structurel subsume le typage nominal.

Nous reviendrons sur ce point dans un prochain cours..

Typage dépendant en Scala

```
01 class Graph {
02   private var nodes: List[Node] = Nil
03   class Node {
04       private[Graph] var connectedNodes: List[Node] = Nil
05     def +→ (node: Node) {
06       if (connectedNodes.find(node.equals).isEmpty) {
07         connectedNodes = node :: connectedNodes
08       } else { println ("Already connected.") }
09     }
10     nodes = this :: nodes
11   }
12 }
13 scala> val g1 = new Graph
14 scala> val n1 = new g1.Node
15 n1: g1.Node = Graph$Node@25997c
16 scala> val n2 = new g1.Node
17 n2: g1.Node = Graph$Node@84c1f9
18 scala> n1 +→ n2
19 scala> val g2 = new Graph
20 scala> val n3 = new g2.Node
21 n3: g2.Node = Graph$Node@11e8d5c
22 scala> n3 +→ n1
23 <console>:11: error: type mismatch;
24  found : g1.Node
25  required: g2.Node
26          n3 +→ n1
```

Typage dépendant

Typage dépendant

Un système de type est dépendant (de valeurs) si la syntaxe des types peut faire référence à des expressions (calculatoires).

- `Scala` offre une forme restreinte de type dépendant pouvant faire mention de **résultats nommés** de calcul arbitraire, appelés identificateurs stables.

Limite d'utilisation des chemins dépendants

```
01 scala> def id (g : Graph) : Graph = g
02 id: (g: Graph)Graph
03
04 scala> val g3 = id (g1)
05 g3: Graph = Graph@1226fe1
06
07 scala> val n4 = new g3.Node
08 n4: g3.Node = Graph$Node@93e86f
09
10 scala> n1 +→ n4
11 <console>:12: error: type mismatch;
12  found : g3.Node
13  required: g1.Node
14          n1 +→ n4
```

Limite d'utilisation des chemins dépendants

```
01 scala> val n5 = n4.asInstanceOf[g1.Node]
02 n5: g1.Node = Graph$Node@93e86f
03 scala> n1 +→ n5
04 scala> n3
05 res20: g2.Node = Graph$Node@11e8d5c
06 scala> val n6 = n3.asInstanceOf[g1.Node]
07 n6: g1.Node = Graph$Node@11e8d5c
08 scala> n6 eq null
09 res21: Boolean = false
10 scala> n1 +→ n6
11
```

⇒ On retrouve alors le typage des classes imbriquées de Java.

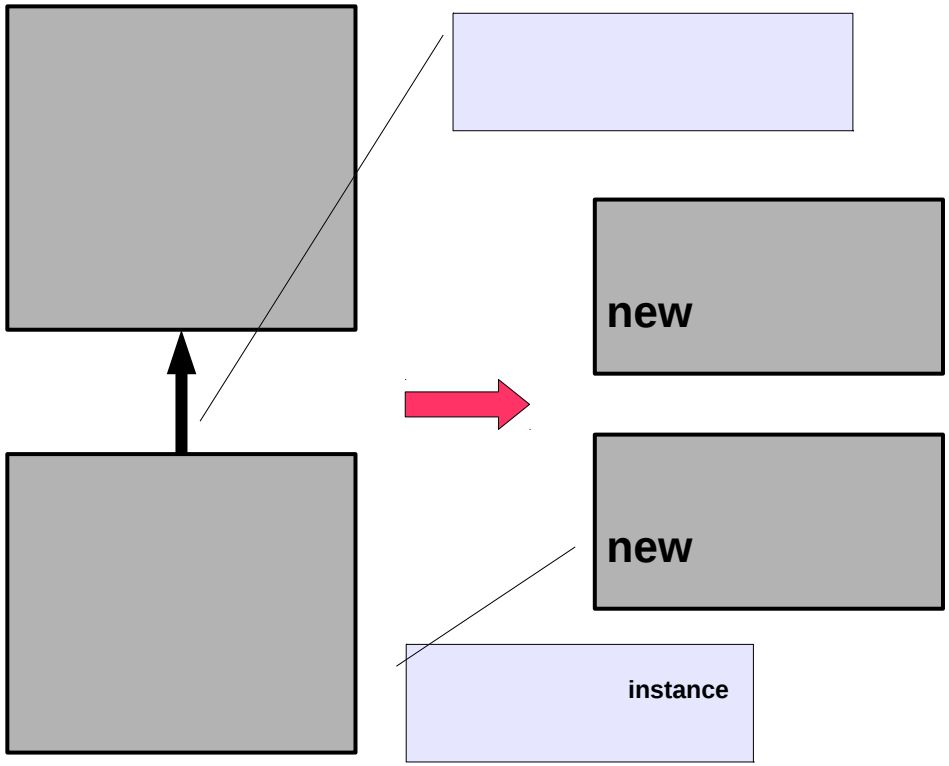
Solution : utiliser le type le plus précis d'un objet

```
01 scala> val also_g1 = (id (g1)).asInstanceOf[g1.type]
02 also_g1: g1.type = Graph@1226fe1
03
04 scala> val n7 = new also_g1.Node
05 n7: also_g1.Node = Graph$Node@1123968
06
07 scala> n1 +→ n7
08
09 scala> n3 +→ n7
10 <console>:13: error: type mismatch;
11  found : also_g1.Node
12  required: g2.Node
13          n3 +→ n7
```

- Le type le plus précis que l'on peut donner à l'identificateur « `also_g1` » est «`g1.type` », c'est-à-dire « le type de g1 ».
- Ce type caractérise exactement la valeur $g1$, c'est un type singleton.

Plan de la partie *“Construire des objets”*

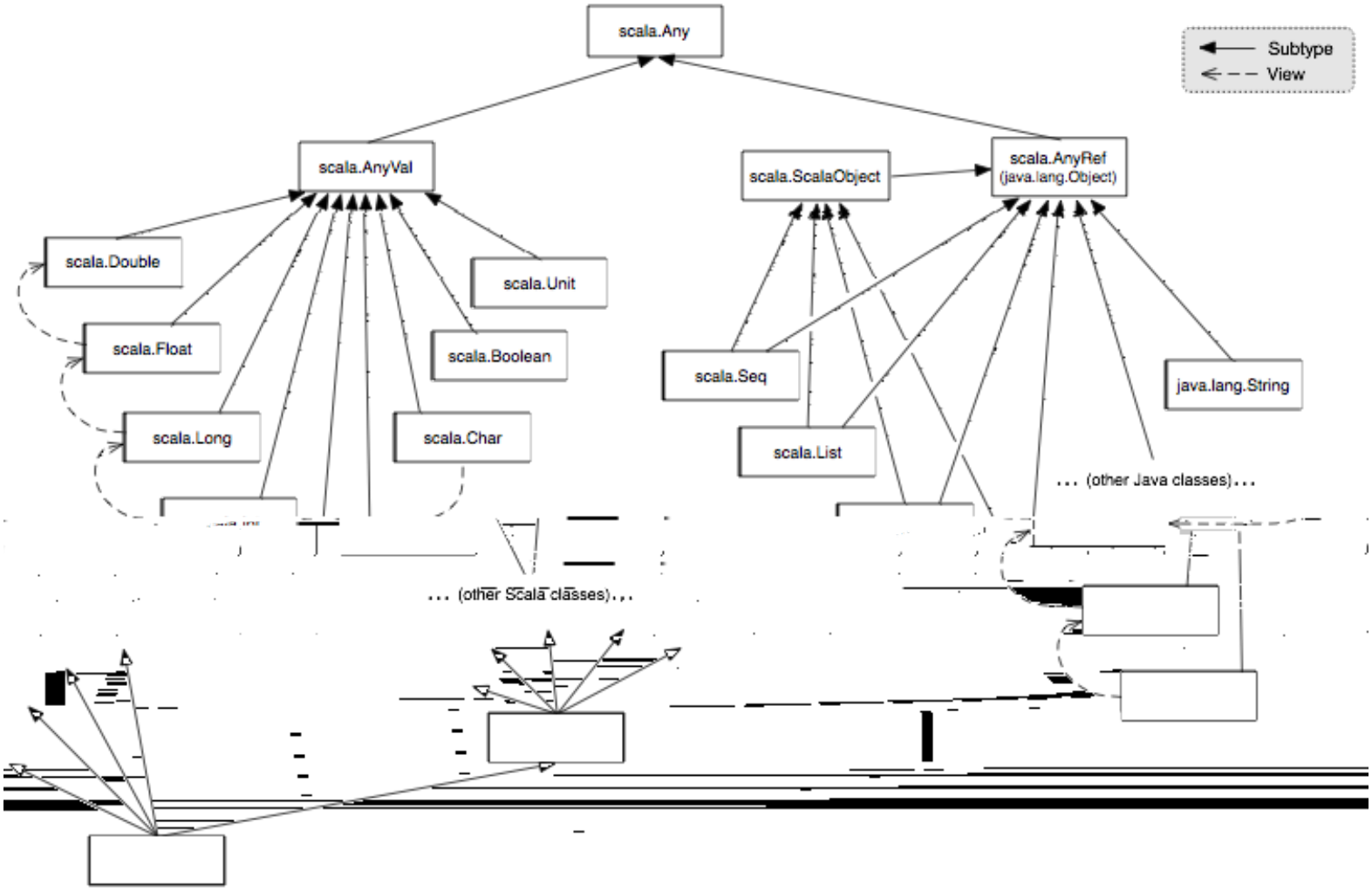
Construction de classe par héritage



L'héritage en Scala

```
01 class A {
02    def m1 = 0
03 }
04
05 class B extends A {
06    def m2 = 1
07 }
08
09 val ib = new B
10 val x = ib.m2
11 val y = ib.m1
```

La hiérarchie des types prédéfinis de Scala



Héritage et constructeur par défaut

```
01 class A (x : Int) {
02    def m1 = x
03 }
04
05 class B (x : Int, y : Int) extends A (x + y) {
06    def m2 = x
07 }
08
09 val ib = new B (1, 2)
10 val x = ib.m2
11 val y = ib.m1
```

Quelles sont les valeurs des variables `x` et `y` ?

Héritage et constructeurs auxiliaires

```
01 class A (x : Int) {
02    def m1 = x
03    def this (x : Int, y : Int) = this (x + y)
04 }
05
06 class B (x : Int, y : Int) extends A (x, y) {
07    def m2 = x
08    def this () = this (0, 0)
09 }
10
11 val ib = new B ()
12 val x = ib.m2
13 val y = ib.m1
```

Quelles sont les valeurs des variables `x` et `y` ?

Interdire l'héritage

Pour des raisons de sécurité ou pour améliorer l'efficacité de la compilation, on peut interdire l'introduction de nouvelles sous-classes à une hiérarchie. Pour cela, il suffit de préfixer la définition d'une classe `A` par le mot-clé `sealed`. Dès lors, il ne sera plus possible d'hériter de `A` sauf à l'intérieur du fichier source dans lequel cette classe `A` est définie.

Quand une classe est marquée `sealed`, le compilateur peut détecter les analyses par motifs sur des valeurs de cette classe qui ne sont pas **exhaustives**. Par exemple :

```
sealed class Color
case class Red () extends Color
case class Blue () extends Color
case class Green () extends Color

scala> def f (x : Color) =
  x match {
    case Red () => "red"
    case Blue () => "blue"
  }
<console>:15: warning: match is not exhaustive!
missing combination Color
missing combination Green
```

Le raisonnement par cas

La décomposition des différents cas d'une situation est essentielle dans la conception d'un système ou d'un algorithme. Grâce aux classes de cas scellées et aux analyses par cas, le compilateur de `Scala` vérifie pour vous que vous avez bien traités tous les cas.

Redéfinition de définitions en Scala

```
01 class A {
02    def m : Int = 0
03 }
04
05 class B extends A {
06    override def m : Int = 1
07 }
08
09 val bi = new B
10 val x = bi.m
```

Redéfinition de définitions en Scala

```
01 class A {
02   def m : Int = 2 + 2
03 }
04
05 class B extends A {
06   override val m : Int = 2 + 2
07 }
08
09 val bi = new B
10 val x = bi.m
```

À quoi sert cette forme de redéfinition ?

Redéfinition de valeurs en Scala

```
01 class A {
02    def m : Int = 0
03 }
04
05 class B extends A {
06    override def m : Int = 1
07 }
08
09 val bi = new B
10 val x = bi.m
```

Redéfinition de valeurs en Scala

```
01 class A {
02    val m : Int = 0
03 }
04
05 class B extends A {
06    override def m : Int = 1
07 }
08
09 // error: overriding value m in class A of type Int;
10 // method m needs to be a stable, immutable value
```

D'après vous, pourquoi une telle limitation ?

Redéfinition des variables

```
01 class A {
02    var x = 0
03 }
04 class B extends A {
05    override var x = 1
06 }
07 class C extends A {
08    override val x = 1
09 }
10 class D extends A {
11    override def x = 1
12 }
13
14 // error: overriding variable x in class A of type Int;
15 // variable x cannot override a mutable variable
```

Récursion ouverte

Lorsque l'on écrit une classe objet, il faut s'imaginer que tous les appels de méthodes peuvent faire appel à des redéfinitions de ces méthodes dans les sous-classes de la classe en cours de définition. On dit que l'on travaille sous l'hypothèse d'un monde ouvert.

En d'autres termes, on peut voir l'ensemble des définitions des membres d'une classe comme un ensemble de **définitions mutuellement récursives** sur lesquelles on se donne la liberté de revenir dans le futur en les redéfinissant dans une classe fille. Cette propriété (méta-linguistisque) des objets s'appelle la **récursion ouverte**.

Par quel mécanisme la récursion ouverte est-elle implémentée ?

Récursion ouverte

Lorsque l'on écrit une classe objet, il faut s'imaginer que tous les appels de méthodes peuvent faire appel à des redéfinitions de ces méthodes dans les sous-classes de la classe en cours de définition. On dit que l'on travaille sous l'hypothèse d'un monde ouvert.

En d'autres termes, on peut voir l'ensemble des définitions des membres d'une classe comme un ensemble de **définitions mutuellement récursives** sur lesquelles on se donne la liberté de revenir dans le futur en les redéfinissant dans une classe fille. Cette propriété (méta-linguistisque) des objets s'appelle la **récursion ouverte**.

Par quel mécanisme la récursion ouverte est-elle implémentée ? Grâce au fait que tout appel de méthode s'effectue à travers `this`. Comme `this` ne correspond pas à une instance de la classe en cours de définition mais à une **instance d'une de ses sous-classes**, une version redéfinie de la méthode peut être exécutée.

Fermer la récursion

Il est possible d'interdire la redéfinition d'un membre à l'aide du mot-clé `final`. Ce mécanisme a deux intérêts :

- Il simplifie le raisonnement sur la classe : on connaît exactement le code executé par un appel à une méthode marquée `final` puisque c'est nécessairement celui de la classe en cours de définition.
- Le compilateur peut optimiser un appel à une méthode `final`.

Plan de la partie *“Construire des objets”*

Sous-typage induit

Quand on définit une relation d'héritage entre deux classes `A` et `B`, une relation de sous-typage entre les types (nommés) `A` et `B` est déclarée dans le système.

Sous-typage

Un type B est un sous-type d'un type A si dans tout contexte où une valeur de type A est attendue, une valeur de type B peut être utilisée.

Nous avons vu dans précédemment que `Scala` propose du typage nominal et du typage structurel. La relation d'héritage déclarée entre deux classes augmente seulement la relation de sous-typage **nominale**.

Nous noterons

$$T \prec: U$$

pour la propriété « le type T est un sous-type de U ».

Covariance

Propriété

Si $A \prec: A$ et $B \prec: B$ alors $A \times B \prec: A \times B$.

Le type des paires formées de deux types A et B qui sont des sous-types de A et B est un sous-type des paires formées des deux types A et B. On dit que l'opérateur de type de formation des paires est covariant.

On peut clairement généraliser cette propriété aux n-uplets. De même, les enregistrements[2], qui sont des n-uplets nommés, vérifient aussi cette propriété.

2. Les enregistrements sont aussi appelés "structures" en C.

Le type des fonctions

Soient $A, B, A\ , B\ .$ À quelles conditions a-t-on $A \rightarrow B \prec: A \rightarrow B$?

```
01 class A { var x = 0 }
02 class B extends A { var y = 1 }
03 class C extends B { var z = 2 }
04
05 def g (f : B => B) = { val b = f (new B) ; b.y = 1 ; b }
06
07 def idA : A => A = (x : A) => x
08 def idB : B => B = (x : B) => x
09 def idC : C => C = (x : C) => x
```

Le type des fonctions

Soient $A, B, A\ , B\ .$ À quelles conditions a-t-on $A \rightarrow B \prec: A \rightarrow B$?

```
01 class A { var x = 0 }
02 class B extends A { var y = 1 }
03 class C extends B { var z = 2 }
04
05 def g (f : B => B) = { val b = f (new B) ; b.y = 1 ; b }
06
07 def idA : A => A = (x : A) => x
08 def idB : B => B = (x : B) => x
09 def idC : C => C = (x : C) => x
```

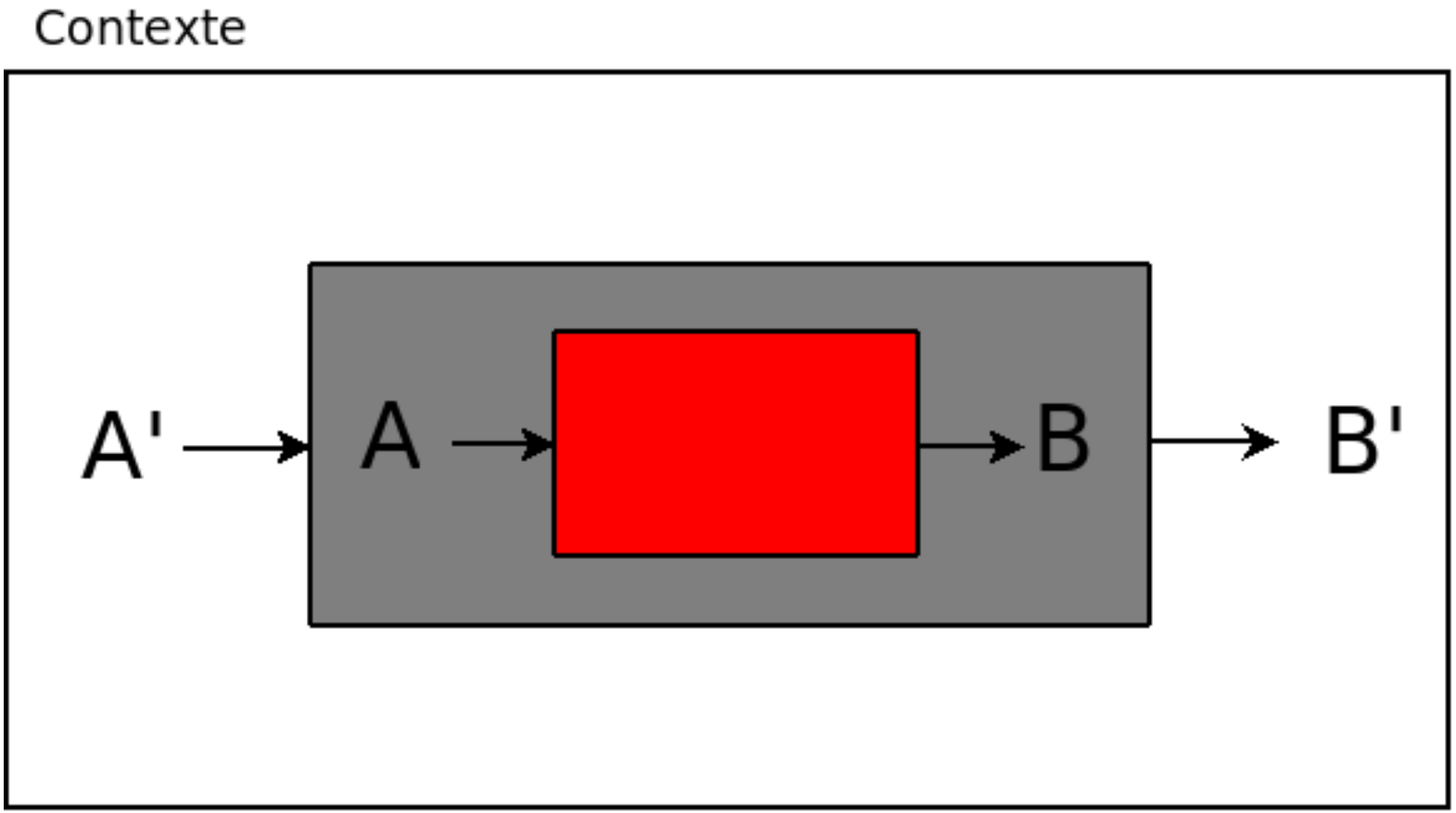
```
01 val b = g (idB) // OK
02 val a = g (idA) // Error
03 val c = g (idC) // Error
04
05 def fAC (a : A) : C = { val c = new C ; c.x = a.x ; c }
06 val c = g (fAC) // OK
```

Quel est le type de c ?

Le type des fonctions

Contexte

Le type des fonctions



Contravariance

Propriété

Si $A \prec: A$ et $B \prec: B$ alors $A \to B \prec: A \to B$.

On dit que le constructeur de type des fonctions est contravariant sur son domaine. Il est covariant sur son codomaine.

Le type des références

- Soient A et B tels que $A \prec: B$.
- Le type des références sur A est muni de deux opérations :
  - $\mathsf{write} : \mathsf{ref}\, A \to A \to \mathsf{unit}$
  - $\mathsf{read} : \mathsf{ref}\, A \to A$

L'opérateur de type des références est-il covariant ou contravariant ?

Invariance

- Le type A devant être utilisé en position covariante et en position contravariante, l'opérateur de type ref est donc invariant.

Qu'en déduire sur les objets ?

Structurellement, un objet obtenu par instanciation d'une classe peut être vu comme un enregistrement (une structure en C) dont les champs correspondent aux membres de la classe.

Ainsi, d'un point de vue structurel, le type structurel de la classe `A` suivante est un sous-type du type de la classe `B` tandis que le type de la classe `C` n'est pas en relation de sous-typage avec celui de `B`. Pourquoi ?

```
class A { def m : Any => Int = (m : Any) => ... }
class B { def m : Int => Int = (m : Int) => ... }
class C { def m : Int => Any = (m : Int) => ... }
```

Qu'en déduire sur les objets ?

Règle de bon typage des redéfinitions

Quand on redéfinit un membre de type T d'une classe A dans une sous-classe B, cette nouvelle définition doit avoir un type U qui est un sous-type de T.

Corollaire

Cas particulier important : Quand le membre redéfini est une méthode, cela signifie que les types des arguments de la méthode dans B doivent être des **sur-types** des types des arguments de la méthode dans A.

Héritages valides

Pour être valide, la définition d'une sous-classe doit contenir des redéfinitions vérifiant la règle de bonne formation énoncée dans le transparent précédent ?

Qu'en est-il en Java, en C++, etc ?

Qu'en est-il en Java ?

En Java, la règle est plus stricte : une redéfinition ne doit pas changer le type des arguments des méthodes ! Ces types sont invariants dans toutes les redéfinitions tout au long de hiérarchie d'objets.

```
public class A {
 void m (Integer x) {}
}

public class B extends A {
   void m (Integer x) { /* Ici la redéfinition de m. */ }
}
```

Mais pourtant. . .

Vous avez sûrement déjà écrit un programme Java de la forme suivante :

```
public class A {
 int equal (A x) { return 0 ; }
}

public class B extends A {
   int equal (B x) { return 0 ; }
}
```

Plus précisément. . .

```
public class OneInteger {
    public int x;
    public boolean equal (OneInteger that) { return (that.x == this.x); }
}

public class TwoIntegers extends OneInteger {
    public int y;
    public boolean equal (TwoIntegers that) {
      return (that.y == this.y && super.equal (that));
    }
}

static void test_them (OneInteger o1, OneInteger o2) {
    System.out.println (o1.equal (o2));
}

public void test () {
    TwoIntegers t1 = new TwoIntegers ();
    TwoIntegers t2 = new TwoIntegers ();
    t1.x = 0; t1.y = 1; t2.x = 0; t2.y = 2;
    test_them (t1, t2);
}
```

Qu'affiche ce programme ?

Surcharge et redéfinition

Le programme précédent affiche « true ». En effet, la méthode « equal » de la classe B **n'est pas une redéfinition** de la méthode « equal » de la classe A !

Il s'agit d'une méthode totalement différente – parce qu'elle attend des arguments de type différents – qui « par hasard » [3] a le même nom que la méthode de B.

Ainsi, même si les définitions de classe précédentes semblent parfaitement naturelles, l'exécution de ce programme ne correspondent pas **du tout** à ce que l'on attend !

3. En fait, parce que c'est plus « pratique » comme ça.

Surcharge et redéfinition

Le programme précédent affiche « true ». En effet, la méthode « equal » de la classe B **n'est pas une redéfinition** de la méthode « equal » de la classe A !

Il s'agit d'une méthode totalement différente – parce qu'elle attend des arguments de type différents – qui « par hasard » [3] a le même nom que la méthode de B.

Ainsi, même si les définitions de classe précédentes semblent parfaitement naturelles, l'exécution de ce programme ne correspondent pas **du tout** à ce que l'on attend !

Il ne faut donc **jamais** écrire ce type de programme !

3. En fait, parce que c'est plus « pratique » comme ça.

Mais, peut-être que . . .

Peut-être que la définition du sous-typage est incorrecte. . . Essayons de voir ce qui se passerait dans un hypothétique langage qui autorisait la méthode « equal » de la sous-classe B à être comprise comme une redéfinition de celle de la classe A.

```
class A =
  attribute x : int
  message equal (that : A) =
    (x = that.x)
end

class B =
  inherits A
  attribute y : int
  message equal (that : B) =
     (x = that.x    y = that.y)
end
```

Mais, peut-être que . . .

Peut-être que la définition du sous-typage est incorrecte. . . Essayons de voir ce qui se passerait dans un hypothétique langage qui autorisait la méthode « equal » de la sous-classe B à être comprise comme une redéfinition de celle de la classe A.

```
class A =
  attribute x : int
  message equal (that : A) =
    (x = that.x)
end

class B =
  inherits A
  attribute y : int
  message equal (that : B) =
    (x = that.x   y = that.y)
end
```

```
function will_explode (i : A) =
  let i' = new A { x = 1 } in
    i.equal (i')
```

Mais, peut-être que . . .

Peut-être que la définition du sous-typage est incorrecte. . . Essayons de voir ce qui se passerait dans un hypothétique langage qui autorisait la méthode « equal » de la sous-classe B à être comprise comme une redéfinition de celle de la classe A.

```
class A =
  attribute x : int
  message equal (that : A) =
    (x = that.x)
end

class B =
  inherits A
  attribute y : int
  message equal (that : B) =
     (x = that.x    y = that.y)
end
```

```
function will_explode (i : A) =
  let i' = new integer { x = 1 } in
     i.equal (i')

function explosion () =
  let i =
     new B { x = 42, bounded = 51 }
  in
    will_explode i
```

Et en Scala ?

Ce problème nommé « Le problème des méthodes binaires » est **intrinsèque à la programmation objet**. Il existe donc aussi en Scala. Cependant, nous allons voir qu'il existe des solutions élégantes de ce problème en Scala, rendues possible grâce à son riche système de type.

```
01 class A (val x : Int) {
02   def equal (that : A) = (that.x == x)
03 }
04
05 class B (x : Int, val y : Int) extends A (x) {
06   def equal (that : B) = super.equal (that)    that.y == y
07 }
08
09 def check (x : A, y : A) =
10   println (x equal y)
11
12 scala> check (new B (1, 0), new B (1, 1))
13 true
```

Première (quasi) solution

```
01 class A (val x : Int) {
02   def equal (that : A) = (that.x == x)
03 }
04
05 class B (x : Int, val y : Int) extends A (x) {
06   override def equal (that : A) = that match {
07     case thatB : B => this.equal (thatB)
08     case thatA : A => super.equal (thatA)
09   }
10   def equal (that : B) = super.equal (that)    that.y == y
11 }
12
13 def check (x : A, y : A) =
14   println (x equal y)
15
16 scala> check (new B (1, 0), new B (1, 1))
17 false
```

L'idée consiste à rajouter une redéfinition effective de la méthode `equal` de la classe `A` dans `B` en lui donnant la même signature. Depuis cette méthode, il est alors possible de déterminer si l'objet auquel on se compare est un `B` et alors appeler la méthode spécialisée pour `B`.

Inconvénients de cette solution

- Elle nécessite l'écriture manuelle de la version redéfinie de la méthode. Ce travail est répétitif, systématique et donc sujet à erreur. Il existe des compilateurs qui font se rajout mais pas en `Scala`, ni dans aucun langage non académique.
- Elle rajoute des tests dynamiques supplémentaires, ce qui peut être une source d'inefficacité.
- Plus grave : elle casse la **transitivité** de l'égalité !

Démonstration de la perte de la transitivité

```
01 val x = new B (1, 0)
02 val y = new B (1, 1)
03 val z = new A (1)
04
05 scala> x.equal (z)
06 res18: Boolean = true
07
08 scala> x.equal (y)
09 res19: Boolean = false
10
11 scala> y.equal (z)
12 res20: Boolean = true
```

Une idée ?

Seconde (quasi) solution

Une façon de résoudre le problème consiste à décider que la comparaison entre deux objets qui n'ont pas le même type échoue toujours.

```
class A (val x : Int) {
  def equal (that : A) = that match {
    case thatA : A => (that.x == x)    getClass == that.getClass
    case _ => false
  }
}

class B (x : Int, val y : Int) extends A (x) {
  override def equal (that : A) = that match {
    case thatB : B => this.equal (thatB)    getClass == that.getClass
    case _ => false
  }
  def equal (that : B) = super.equal (that)    that.y == y
}
```

Il y a toujours un problème avec cette approche, une idée ?

Le problème des classes anonymes

```
val t = new B (1, 0) { override val y = 1 }

scala> t.equals (y)
res21: Boolean = false
```

La classe anonyme créée par Scala ne redéfinit pas une nouvelle notion d'égalité : on aimerait qu'elle s'intègre à la relation d'égalité existante sur B.

Dernière version de cette solution au problème des méthodes binaires

```
class A (val x : Int) {
  def equal (that : A) = that match {
    case thatA : A => (that.x == x)    (thatA canEqual this)
    case _ => false
  }
  def canEqual (that : A) = that.isInstanceOf[A]
}

class B (x : Int, val y : Int) extends A (x) {
  override def equal (that : A) = that match {
    case thatB : B =>
      this.x == that.x    this.y == thatB.y    (thatB canEqual this)
    case _ => false
  }
  override def canEqual (that : A) = that.isInstanceOf[B]
}
```

Plan de la partie *“Construire des objets”*

Classe abstraite

Classe abstraite

Une classe abstraite est une classe dont certains membres sont **indéfinis**.

Membre abstrait

Un membre est dit abstrait ou virtuel si il est indéfini dans une classe.

En Scala

En Scala, il suffit de déclarer un membre sans en donner de définition pour le rendre abstrait. Une classe abstraite doit être marquée du modificateur `abstract`.

```
abstract class Shape {
  def description : String
  def show : Unit = println (description)
}

class Square extends Shape {
  override def description = "Hello I am a square."
}

class Circle extends Shape {
  override def draw = "Hello I am a circle."
}
```

Le mot-clé `override` est optionnel ici puisqu'il n'y a que des définitions.

Rôle des classes abstraites

Nous verrons que les classes abstraites ont un rôle double dans la programmation orientée objet : elles servent à **factoriser** des composants en s'abstrayant de leur différence et elles servent à dénoter des **concepts**.

Dans l'exemple précédent, effectivement, la classe forme permet d'une part de factoriser le code de la méthode `show` et d'autre part, elle représente le concept de forme, qui n'a pas d'existence physique réelle. En particulier, il n'existe pas de description précise d'une forme mais il en existe pour toutes les instances de ce concept.

Exemple

Les classes abstraites permettent de décrire des relations abstraites entre des concepts. Pour comprendre cette idée, prenons l'exemple de situation suivante :

> *Imaginons l'application d'une séquence de filtres dans un logiciel de retouche d'images. Pour tenir l'utilisateur au courant de la progression du calcul, plusieurs objets s'affichent dans son interface : une barre de progression, un pourcentage dans la barre de titre, une icône sur son bureau ... Le nombre de ces indicateurs n'est a priori pas limité. Comment s'assurer qu'ils seront tous mis au courant de l'avancement du processus ?*

Écrivez un diagramme de classes UML pour décrire une architecture qui serait une solution à ce problème.

Principe de généralisation

Généralisation

Le principe d'inférence ou de généralisation consiste à établir des règles générales s'appliquant à des instances particulières.

C'est un des principes fondamentaux pour concevoir une architecture générale.

Une mauvaise réponse

Une mauvaise façon de traiter le problème précédent consisterait à intégrer le mécanisme de notification des objets au cas par cas, par exemple en appelant explicitant une méthode de chaque objet à notifier.

```
object filter {
  def apply = {
    /* Do something that apply a filter to an image. */
    icon.notify
    titlebar.notify
    progressionbar.notify
  }
}
```

Une bonne réponse

Une meilleure façon de procéder pour résoudre ce problème consiste à **expliciter les concepts** mis en jeu.

Dans ce problème, il y a un **sujet d'observation** et des **observateurs**. Au moment de leur création, les observateurs doivent s'inscrire auprès du sujet pour que ce dernier décide de les notifier lorsque son état est modifié.

Cela signifie qu'il y a deux classes abstraites et qu'il faut définir leur interaction en des termes abstraits. Il suffira ensuite, par héritage, de **spécialiser** ce comportement au cas qui nous intéresse.

Premier essai

```
abstract class Subject {
  private var observers : List[Observer] = Nil
  def register (new_observers : Observer*) =
    observers ++= new_observers
  def notifyObservers =
    observers.foreach (_.onNotify (this))
}

abstract class Observer {
  def onNotify (s : Subject)
}
```

Nous avons capturé la logique de l'interaction entre les observateurs et le sujet. On pourrait cependant être plus fin en autorisant une notification *asynchrone* des observateurs. (Exercice !)

Spécialisation

```
class Filter extends Subject {
  var currentFilterName : String = _
  def apply (filterName: String) = {
    currentFilterName = filterName;
    notifyObservers
  }
}
class Icon extends Observer {
  def onNotify (s : Subject) = s match {
    case filter: Filter =>
      println ("Icon sees " + filter.currentFilterName)
  }
}
class ProgressBar extends Observer {
  def onNotify (s : Subject) = s match {
    case filter: Filter =>
      println ("Progress bar sees " + filter.currentFilterName)
  }
}
```

Critiquez cette implémentation !

Critique de ce premier essai

Cette implémentation a deux défauts apparents principaux.

Tout d'abord, il n'y a pas aucune garantie qu'un observateur soit nécessairement lié à un sujet car au moment de sa création ce n'est pas le cas. Or, notre définition suggère que ce lien doit toujours exister.

Ensuite, bien qu'un observateur sache le type exact de son sujet, il y a une vérification du type de ce dernier dans la méthode qui implémente la réaction à une notification.

Premier problème

Il y a une solution immédiate au premier défaut qui consiste à rajouter un argument au constructeur de tout observateur correspondant au sujet observé. On supprime le second défaut du même coup !

```
abstract class Subject {
  private var observers : List[Observer] = Nil
  def register (new_observers : Observer*) = observers ++= new_observers
  def notifyObservers = observers.foreach (_.onNotify)
}
abstract class Observer (s: Subject) {
  s.register (this)
  def onNotify
}
class Filter extends Subject {
  var currentFilterName : String = _
  def apply (filterName: String) = {
    currentFilterName = filterName;
    notifyObservers
  }
}
class Icon (s: Filter) extends Observer (s) {
  def onNotify = println ("Icon sees " + s.currentFilterName)
}
class ProgressBar (s: Filter) extends Observer (s) {
  def onNotify = println ("Progress bar sees " + s.currentFilterName)
}
```

Une solution peu flexible

La solution précédente peut sembler séduisante et dans certaines situations, elle est effectivement pertinente. Cependant, elle interdit à un observateur d'avoir plusieurs sujets ou même tout simplement de changer de sujet.

Enfin, plus grave encore : la durée de vie d'un sujet est maintenant liée à la vie de ces observateurs puisque ces derniers gardent un pointeur sur leur sujet. Ainsi, même si le sujet est inactif et inutile, il n'est pas libéré de la mémoire.

La première solution n'avait pas se problème : un observateur pouvait se faire connaître auprès de plusieurs sujets et à la destruction d'un sujet, la relation avec son observateur disparaissait avec lui.

Nous revenons donc sur le premier défaut : cela n'en était pas un.

C'était notre définition initiale qui était fausse !

Nouvelle définition

On passe de :

> *Il y a un **sujet d'observation** et des **observateurs**. Au moment de leur création, les observateurs doivent s'inscrire auprès du sujet pour que ce dernier décide de les notifier lorsque son état est modifié.*

à

> *Il y a des **sujets d'observation** et des **observateurs potientiels** de ces sujets. Les observateurs doivent s'inscrire auprès d'un ou plusieurs sujets pour que ces derniers décident de les notifier lorsque leur état est modifié.*

Second problème

Le second problème est plus complexe : il faut synchroniser le type des observateurs avec le type du sujet qu'ils observent. Une première tentative naïve consisterait à écrire :

```
class Icon extends Observer {
  def onNotify (filter : Filter) =
      println ("Icon sees " + filter.currentFilterName)
  }
}
```

Mais cela ne fonctionne évidemment pas. Pourquoi ?

Second problème

Le second problème est plus complexe : il faut synchroniser le type des observateurs avec le type du sujet qu'ils observent. Une première tentative naïve consisterait à écrire :

```
class Icon extends Observer {
  def onNotify (filter : Filter) =
        println ("Icon sees " + filter.currentFilterName)
  }
}
```

Mais cela ne fonctionne évidemment pas. Pourquoi ? Parce que cette méthode `onNotify` n'est pas une redéfinition de la méthode `onNotify` de la classe `Observer` car `Filter` n'est pas un surtype de `Subject`, ce qui est imposé par la contravariance des arguments de méthode.

Types virtuels à la rescousse !

Une solution élégante à ce problème consiste à définir les deux classes `Object` et `Observer` à partir des types (encore inconnus) de leurs sous-classes.

```
abstract class SubjectObserver {
  /* O is the type of a concrete Observer. */
  type O <: Observer
  /* S is the type of a concrete Subject. */
  type S <: Subject
  /* The following classes are defined using these types. */
  abstract class Subject {
    /* The type of this must be S */
    self:S =>
    var observers : List[O] = Nil
    def register (new_observers : O*) =
      observers ++= new_observers
    def notifyObservers =
      observers.foreach (_.onNotify (this))
  }
  abstract class Observer { def onNotify (s : S) }
}
```

Instanciation

```
object FilterLib extends SubjectObserver {
  type S = Filter
  type O = FilterObserver

  class Filter extends Subject {
    var currentFilterName : String = _
    def apply (filterName: String) = {
      currentFilterName = filterName;
      notifyObservers
    }
  }

  abstract class FilterObserver extends Observer {
    def onNotify (s : S)
  }
}
class Icon extends FilterLib.FilterObserver {
  def onNotify (filter : FilterLib.Filter) =
      println ("Icon sees " + filter.currentFilterName)
}
class ProgressBar extends FilterLib.FilterObserver {
  def onNotify (filter : FilterLib.Filter) =
      println ("Progress bar sees " + filter.currentFilterName)
}
```

Ingrédients : Type virtuel et Type de `this`

Type virtuel

Un type virtuel est un type abstrait qui sera concrétisé uniquement dans une sous-classe concrète.

Type de `this`

Dans une classe, le type de `this` est le type qu'aura l'objet final instancié par la classe courante ou par une de ses sous-classes.

Nous reviendrons sur ces notions dans la suite du cours.

Comment organiser les classes ?

Comme vous venez de le voir, organiser des classes demandent de la réflexion et de la créativité. Pour vous aider, des schémas récurrents de conception ont été exhibés. On les appelle des patrons de conception (*Design Pattern*). La solution précédente est le patron de conception « Sujet-observateur ».

Nous allons en découvrir d'autres tout au long du cours.

Plan de la partie *“Construire des objets”*

Le sous-typage, un mécanisme de modularité

Outre son utilité pour la réutilisation du code, le sous-typage favorise aussi la modularité car c'est aussi un mécanisme d'**encapsulation**.

Pourquoi ? Le sous-typage permet de cacher le **type exact** des objets en ne conservant que leur **type utile**. On dissocie un ensemble de (sur-)types servant d'**interfaces** de communication entre les objets de l'ensemble des (sous-)types servant à décrire leur représentation interne, leur **implémentation**.

Exemple

Imaginons une bibliothèque fournissant des structures de données de dictionnaires :

```
01 class ListDictionary[Key, Data] {
02   private var entries : List[(Key, Data)] = Nil
03   def assoc (key : Key) : Option[Data] =
04     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
05   def inject (key : Key, data : Data) =
06     entries = (key, data) :: entries
07 }
08
09 class ArrayDictionary[Key, Data] {
10   private var entries : ResizableArray[(Key, Data)] = new ArrayBu□er[(Key, Data)]
11   def get (key : Key) : Option[Data] =
12     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
13   def push (key : Key, data : Data) =
14     entries :+ (key, data)
15 }
```

Critiquez la conception de cette bibliothèque !

Première critique : Non uniformité du nommage

La première critique évidente vis-à-vis de la modularité s'appuie sur la non uniformité du nommage. En effet, imaginons un client qui utiliserait de façon intensive la classe `ListDictionary` de cette bibilothèque. Son code source (faisant plusieurs millions de ligne de code) contiendrait des milliers d'instanciation et d'appels de méthode de la forme :

- `new ListDictionary[K, D]`
- e.assoc (key)
- e.inject (key, data)

Si ce client décide de remplacer ses utilisations de cette classe par la classe `ArrayDictionary` alors il devra modifier toutes ces expressions ! Ouch !

Correction du problème de nommage

```
01 class ListDictionary[Key, Data] {
02   private var entries : List[(Key, Data)] = Nil
03   def assoc (key : Key) : Option[Data] =
04     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
05   def inject (key : Key, data : Data) =
06     entries = (key, data) :: entries
07 }
08
09 class ArrayDictionary[Key, Data] {
10   private var entries : ResizableArray[(Key, Data)] = new ArrayBu□er[(Key, Data)]
11   def assoc (key : Key) : Option[Data] =
12     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
13   def inject (key : Key, data : Data) =
14     entries :+ (key, data)
15 }
```

Il reste encore une certaine duplication de code. . .

Classe abstraite pour factoriser du code

```
01 abstract class SearchableEntries[Key, Data] {
02   type Entries <: {
03     def find (p : ((Key, Data)) => Boolean) : Option[(Key, Data)]
04   }
05   protected var entries : Entries
06   def assoc (key : Key) : Option[Data] =
07     entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
08 }
09
10 class ListDictionary[Key, Data] extends SearchableEntries[Key, Data] {
11   type Entries = List[(Key, Data)]
12   override protected var entries : Entries = Nil
13   def inject (key : Key, data : Data) =
14     entries = (key, data) :: entries
15 }
16
17 class ArrayDictionary[Key, Data] extends SearchableEntries[Key, Data] {
18   type Entries = ResizableArray[(Key, Data)]
19   override protected var entries : Entries = new ArrayBu□er[(Key, Data)]
20   def inject (key : Key, data : Data) =
21     entries :+ (key, data)
22 }
```

Le type virtuel `Entries` dénote la représentation final de l'ensemble des entrées du dictionnaries. Ce n'est toujours pas totalement satisfaisant...

Nécessité d'encapsulation

Un client de cette bibliothèque respecte maintenant un protocole uniforme d'interaction avec un dictionnaire. Cependant, il reste deux problèmes.

D'abord, il doit encore choisir explicitement une classe concrète à instancier et si il veut passer d'une classe à une autre, il faut qu'il change le nom de cette classe en tout point de son code.

Ensuite, comme la bibliothèque divulgue l'implémentation interne des deux structures de données, il se peut qu'il s'appuie de façon abusive sur la logique interne de ces types d'objets. Dans notre cas, un client utilisant la classe `ListDictionary` pourrait appuyer ses algorithmes le fait que si deux entrées sont ajoutées pour la même clé, la dernière prévaut.

Qu'arrive-t-il si il décide d'utiliser la classe `ArrayDictionary` ou si dans une nouvelle version de la bibliothèque, cette propriété n'est plus respectée ?

Une classe abstraite pour interface d'utilisation

```
abstract class Dictionary[Key, Data] {
  /* Add a new entry that associates [key] to [data]. If there */
  /* already is an entry with that key, the behavior is unspecified. */
  def inject (key : Key, data : Data) : Unit
  def assoc (key : Key) : Option[Data]
}

abstract class SearchableDictionary[Key, Data] extends Dictionary[Key, Data] {
  type Entries <: {
    def find (p : ((Key, Data)) => Boolean) : Option[(Key, Data)]
  }
  protected var entries : Entries
  def assoc (key : Key) : Option[Data] =
    entries.find ({ case (dkey, _) => key == dkey }) map ({ case x => x._2 })
}

class ListDictionary[Key, Data] extends SearchableEntries[Key, Data] {
...
}

class ArrayDictionary[Key, Data] extends SearchableEntries[Key, Data] {
...
}
```

L'interface est un contrat

Même si les classes concrètes sont publiées par la bibliothèque, il est souvent plus modulaire et plus robuste pour le client de spécifier ces propres interfaces et d'écrire ses propres fonctions en s'appuyant seulement sur les interfaces. En effet, une interface décrit ce que **promet** le concepteur de la bibliothèque à ces clients. D'une version à l'autre d'une bibliothèque, les modifications des APIs sont en général minimes.

(Nous reviendrons sur cette propriété un peu plus loin.)

Existe-t-il un moyen de renforcer cette discipline ?

Le patron de conception Usine

```
abstract class Dictionary[Key, Data] {
  /* Add a new entry that associates [key] to [data]. If there */
  /* already is an entry with that key, the behavior is unspecified. */
  def inject (key : Key, data : Data) : Unit
  def assoc (key : Key) : Option[Data]
}

object Dictionary {
  sealed abstract class Criteria
  case object FastestInject extends Criteria
  case object FastestAssoc extends Criteria
  // The dictionary must not weight too much in memory
  case object SmallestFootPrint extends Criteria

  /* Instantiate a dictionary w.t.r. a list of ordered criteria. */
  def apply [Key, Data](criteria : List[Criteria]) : Dictionary[Key, Data] = {
    type Default = ListDictionary[Key, Data]
    criteria match {
      case Nil => new Default
      case FastestAssoc :: _ => new ArrayDictionary[Key, Data]
      case FastestInject :: _ => new ListDictionary[Key, Data]
      case SmallestFootPrint :: xs => apply (xs)
  }
}
  /* Here the implementation of the subclasses, which are */
  /* private to the package. */
```

Extensibilité et modularité via l'utilisation des usines

Pour construire un dictionnaire, un client doit désormais nécessairement utiliser l'objet compagnon `Dictionary` en fournissant les critères de performances attendues du dictionnaire à construire.

Par conséquent, le client ne fait plus référence directement aux noms des sous-classes. La documentation de bibliothèque ne mentionne alors plus du tout ces sous-classes. Il n'y a donc aucune bonne raison pour que le client s'appuie sur leur logique interne.

De plus, la bibliothèque peut proposer de nombreux protocoles distincts de construction de dictionnaires (en fonction du nombre maximal d'entrées par exemple).

Enfin, si la bibliothèque rajoute une nouvelle implémentation de dictionnaire, par exemple un dictionnaire avec une recherche plus rapide (mais une empreinte mémoire plus importante) s'appuyant sur un arbre de recherche, la migration ne nécessite aucune adaptation pour le client. Le concepteur de la bibliothèque ne devra modifier que l'usine pour prendre en compte cette nouvelle sous-classe.

Plan de la partie *“Construire des objets”*

Présentation du problème

Utiliser l'héritage comme un procédé de réutilisation de code peut s'avérer assez subtil. Nous avons déjà vu avec le problème des méthodes binaires que certaines redéfinitions qui pourraient sembler licites ne le sont en fait pas pour des raisons de sûreté de l'exécution. Heureusement, dans ce cas, le système de type du langage de programmation nous empêche de se méprendre sur la forme des redéfinitions autorisées dans les sous-classes.

Malheureusement, il existe des problèmes de **conception** des classes de base pour lesquels un système de types standard n'est d'aucune aide. C'est le cas du problème dit de la classe de base fragile.

Exemple

Tiré de *A study of the fragile base class problem.* Mikhajlov, Sekerinski

Voici une classe qui implémente une structure de données contenant des entiers :

```
01 class Bag {
02    private var elements : List[Int] = _
03    def add (x: Int) = elements = x :: elements
04    def addAll (xs : List[Int]) = xs foreach add
05    def cardinal = elements.size
06 }
```

Exemple

Tiré de *A study of the fragile base class problem.* Mikhajlov, Sekerinski

On veut proposer une version optimisée de la structure précédente calculant le cardinal incrémentalement :

```
01 class CountingBag extends Bag {
02    var counter = 0
03    override def add (x: Int) = { counter += 1 ; super.add (x) }
04    override def cardinal = counter
05 }
```

Quelles sont les hypothèses faites ici sur la classe mère ?

Le problème de la classe de base fragile

On optimise la classe mère en modifiant sa méthode `addAll` :

```
01 class Bag {
02   private var elements : List[Int] = _
03   def add (x: Int) = elements = x :: elements
04   def addAll (xs : List[Int]) = elements ++= xs
05   def cardinal = elements.size
06 }
```

Est-ce que la classe fille est toujours correctement implémentée ?

Principe de substitutivité étendu aux raffinements

Une nouvelle contrainte sur les héritages interdits ou autorisés apparaît ici : une sous-classe B peut hériter sans danger d'une classe A, si son type induit est un sous-type de cette classe A et **des ra□nements futurs de cette classe** A.

Cette idée est assez difficile à formaliser et à mettre en pratique. En observant des exemples pathologiques qui ne respectent pas ce principe, nous allons établir une discipline de conception permettant de résoudre ce problème.

Exemple 1 : Récursion mutuelle non anticipée

```
01 class I {
02    var i : Int = 0
03    def m = i += 1
04    def n = i += 1
05 }
06
07 class SI extends I {
08    override def n = m
09 }
```

Imaginez une nouvelle version de `I` qui mettrait en cause l'implémentation de `SI`.

Exemple 1 : Récursion mutuelle non anticipée

```
01 class I {
02    var i : Int = 0
03    def m = n
04    def n = i += 1
05 }
06
07 class SI extends I {
08    override def n = m
09 }
```

Ouch ! L'héritage peut créer des cycles inattendus !

Fausses hypothèses dans la classe mère

```
01 class SquareRoots {
02    def root2 (x : Float) = sqrt (x)
03    def root4 (x : Float) = pow (x, 0.25)
04 }
05
06 class OtherSquareRoots extends {
07    def root2 (x : Float) = - sqrt (x)
08 }
```

Imaginez une nouvelle version de `SquareRoots` qui mettrait en cause l'implémentation de `OtherSquareRoots`.

Fausses hypothèses dans la classe mère modifiée

```
01 class SquareRoots {
02    def root2 (x : Float) = sqrt (x)
03    def root4 (x : Float) = sqrt (sqrt (x))
04 }
05
06 class OtherSquareRoots extends {
07    def root2 (x : Float) = - sqrt (x)
08 }
```

Ouch ! Même si on a le code de `root2` devant ses yeux lorsque l'on programme `root4`, il ne faut pas faire d'hypothèses trop fortes sur la façon dont la méthode `root2` est implémentée.

Fausses hypothèses dans la classe fille

```
01 class A {
02    def l (k : Int) = assert (k ≥ 5)
03    def m (k : Int) = l (k)
04    def n (k : Int) = {}
05 }
06
07 class B extends A {
08    override def l (k : Int) = {}
09    override def n (k : Int) = m (k)
10 }
```

Imaginez une nouvelle version de `A` qui mettrait en cause l'implémentation de `B`.

Fausses hypothèses dans la classe fille

```
01 class A {
02    def l (k : Int) = assert (k ≥ 5)
03    def m (k : Int) = assert (k ≥ 5)
04    def n (k : Int) = {}
05 k}
06
07 class B extends A {
08    override def l (k : Int) = {}
09    override def n (k : Int) = m (k)
10 }
```

Ouch ! L'implémentation d'une classe fille ne doit pas non plus faire d'hypothèses trop fortes sur l'implémentation de la classe mère.

Accès direct à l'état de la classe mère

```
01 class C {
02    var x : Int = 0
03    def m = x += 1
04    def n = x += 2
05 }
06
07 class D extends C {
08    override def n = x += 2
09 }
```

Imaginez une nouvelle version de `C` qui mettrait en cause l'implémentation de `D`.

Accès direct à l'état de la classe mère

```
01 class C {
02    var x : Int = 0
03    var y : Int = 0
04    def m = { y += 1 ; x = y }
05    def n = { y += 2 ; x = y }
06 }
07
08 class D extends C {
09    override def n = x += 2
10 }
```

Ouch ! La classe fille casse l'invariant de la classe mère maintenant le fait que `x = y`.

Fausse hypothèse sur les invariants d'une classe de base

```
01 abstract class E {
02    private var x : Int = 0
03    def getX = x
04    def incrX = { x += 1 ; onIncrX }
05    def onIncrX : Unit
06 }
07
08 class F extends E {
09    var y : Int = 0
10    override def incrX = { y += 1 ; super.incrX }
11    def onIncrX = assert (getX == y)
12 }
```

Imaginez une nouvelle version de `E` qui mettrait en cause l'implémentation de `F`.

Fausse hypothèse sur les invariants d'une classe de base

```
01 abstract class E {
02    private var x : Int = 0
03    def getX = x
04    def incrX = { onIncrX ; x += 1 }
05    def onIncrX : Unit
06 }
07
08 class F extends E {
09    var y : Int = 0
10    override def incrX = { y += 1 ; super.incrX }
11    def onIncrX = assert (getX == y)
12 }
```

La classe fille a supposé à tort que les appels de la méthode `onIncrX` s'effectuait après que la méthode `incrX` ait été évaluée.

Qu'en retenir ?

Nous venons de voir beaucoup de cas distincts de hiérarchies de classe (trop) sensibles aux évolutions futures. Pour prévenir ce genre de problème, il faut **découpler** au maximum l'implémentation des sous-classes de celle des classes mères, donc utiliser avec beaucoup de précaution l'héritage d'implémentation.

- Il ne faut pas qu'une sous-classe et qu'une révision puisse introduire des cycles de récursion mutuelle.
- Quand on développe une méthode d'une classe mère, il ne faut pas faire d'hypothèses trop fortes sur l'implémentation des autres méthodes de cette classe.
- Quand on développe une méthode sous-classe, on doit faire comme si les méthodes de classe de base ne pouvaient pas être redéfinies.
- Une sous-classe ne doit jamais accéder directement à l'état interne d'une classe de base (*i.e.* il faut bannir le mot-clé `protected`).

Comment écrire des classes de base stables ?

Il y a une autre façon de résoudre le problème de la classe fragile, beaucoup plus extrême : ne jamais modifier la logique interne d'une classe dont dépendent d'autres classes. On peut se permettre de corriger des erreurs sans changer fondamentalement la logique interne ou bien encore rajouter des fonctionnalités, mais c'est tout !

Comment maximiser les chances d'être dans cette situation ?
C'est l'art de la conception... Le mot d'ordre est de voir une classe de base comme une interface aussi bien pour les classes des autres hiérarchies que pour ses futures sous-classes. Il faut alors suivre le mot d'ordre suivant valable pour toute conception d'interface :

Publier une interface simple et réduite.

Comment m'assurer que l'interface est simple et réduite ?

Un moyen de vérifier la simplicité de votre interface est de **documenter** votre classe mère en direction de deux types de client :

- Celui qui l'**utilise** directement en lui expliquant le lien entre les entrées et les sorties des méthodes ainsi que leurs effets sur l'état de l'objet, décrit de façon abstraite.
- Celui qui **hérite** en décrivant la façon dont les méthodes peuvent être redéfinies et en décrivant les états de l'objet qui doivent être supposés avant l'évaluation de chacunes d'elles.

Cette documentation devra servir de référence à respecter pour les futures évolutions de votre classe.

Le patron de conception Facade

Un cas assez courant de simplification vertueuse d'interface survient lorsque l'on utilise une bibliothèque externe dans un développement.

Prenons l'exemple d'une bibliothèque externe servant à interagir avec un système de gestion de base de données. Ce système peut être très évolué et proposer un très grand nombre de fonctionnalités réalisées par l'interaction d'un grand nombre d'objets.

Une classe facade est une **couche d'abstraction** de cet ensemble de fonctionnalités en un sous-ensemble restreint correspondant à ce dont à besoin le reste du développement. Elle correspond à un **point d'entrée** dans ce système externe.

Exemple de façade

Voici une façade très simple pour interagir avec une base de donnée :

```
01 abstract class DB {
02    type Table = List[String]
03    type Row = List[String]
04    type TableName = String
05    def connect : Unit
06    def create (table : Table) : String
07    def insert (table : TableName, row : Row) : Unit
08    def select (table : TableName, field : String, value : String) : List[Row]
09 }
```

La migration d'un système de base de données à un autre est facilité par ce genre d'abstraction. (Attention cependant à ne pas confondre "abstraction simple" et "abstraction simpliste"...)

Adaptateur

Plutôt que de revenir sur une interface déjà fixée pour la modifier de façon importante, il est parfois possible et plus pertinent de fournir des **adaptateurs** permettant de passer d'une interface à une autre.

Exemple

```
01 abstract class OptimisticDictionary[Key, Data] {
02   def inject (key : Key, data : Data) : Unit
03   def assoc (key : Key) : Data
04 }
05
06 class ToOptimisticAdapter[Key, Data, D <: Dictionary[Key, Data]]
07 (dict : D)
08 extends OptimisticDictionary[Key, Data] {
09   def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
10   def assoc (key : Key) : Data =
11     dict.assoc (key) match {
12       case None => error ("Unknown key.")
13       case Some (v) => v
14     }
15 }
16
17 class FromOptimisticAdapter[Key, Data, D <: OptimisticDictionary[Key, Data]]
18 (dict : D)
19 extends Dictionary[Key, Data] {
20   def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
21   def assoc (key : Key) : Option[Data] =
22     try Some (dict.assoc (key))
23     catch {
24       case exn => None
25     }
26 }
```

Coercions implicites en Scala

```
01 object adapters {
02
03   implicit def
04   toOptimisticAdapter [Key, Data, D <: Dictionary[Key, Data]]
05   (dict : D)
06   : OptimisticDictionary[Key, Data] = new OptimisticDictionary[Key, Data] {
07     def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
08     def assoc (key : Key) : Data =
09       dict.assoc (key) match {
10         case None => error ("Unknown key.")
11         case Some (v) => v
12       }
13   }
14
15   implicit def
16   fromOptimisticAdapter[Key, Data, D <: OptimisticDictionary[Key, Data]]
17   (dict : D)
18   : Dictionary[Key, Data] = new Dictionary[Key, Data] {
19     def inject (key : Key, data : Data) : Unit = dict.inject (key, data)
20     def assoc (key : Key) : Option[Data] =
21       try Some (dict.assoc (key))
22     catch {
23       case exn => None
24     }
25   }
26 }
```

Coercions implicites en Scala

```
01 object test {
02   import adapters._
03   def main (args: Array[String]) : Unit = {
04     val d = Dictionary[Int, Int] (List (Dictionary.FastestInject))
05     d.inject (1, 2)
06     d.inject (2, 1)
07     val od : OptimisticDictionary[Int, Int] = d
08     od.inject (3, 0)
09   }
10 }
```

L'appel à la fonction **implicite** de conversion du type `Dictionary` vers le type `OptimisticDictionary` est automatiquement inséré par le compilateur. Nous reviendrons sur ce mécanisme.

Plan de la partie *“Construire des objets”*

Problème de l'extensibilité fonctionnelle

Étant donnée une hiérarchie de classes comme celle-ci :

```
01 abstract class Shape { def description : String}
02 class Square extends Shape { def description = "A square" }
03 class Circle extends Shape { def description = "A circle" }
```

Il est très simple de l'étendre en rajoutant un **nouveau type de donnée**.
Il suffit d'écrire une nouvelle sous-classe :

```
01 class Triangle extends Shape { def description = "A triangle" }
```

Mais comment rajouter une nouvelle méthode **sans modifier** le code existant ?

Première tentative

La façon naïve d'opérer consiste à rajouter un nouvel étage à la hiérarchie :

```
01 class XCircle extends Circle { def area : Int = ... }
02 class XSquare extends Square { def area : Int = ... }
```

Très bien, mais qu'en est-il de la classe abstraite `Shape` ? Comment lui rajouter la méthode `area` ?

Deuxième tentative

Si on essaie d'étendre la hiérarchie en partant de la classe `Shape`...

```
01 class XShape extends Shape { def area : Int }
02 class XSquare extends XShape { ? }
03 class XCircle extends XShape { ? }
```

...on ne sait pas trop de quelle classe hériter dans les classes filles.

Héritage multiple ?

Une (fausse) solution consisterait à utiliser l'héritage multiple (qui n'existe pas en Scala d'ailleurs) :

```
01 class XShape extends Shape { def area : Int }
02 class XSquare extends XShape, Square { def area = ... }
03 class XCircle extends XShape, Circle { def area = ... }
```

En C++, qui propose l'héritage multiple, cette solution fonctionnerait ... temporairement puisqu'à la prochaine extension du même genre un héritage en diamant surviendrait (et tous les problèmes que vous connaissez sur ce sujet).

Le patron de conception Visitor

Si le nombre d'extensions de fonctionnalités d'une hiérarchie de classes n'est pas prévisible à l'avance, on peut définir une classe de **visiteur** capable de discriminer entre les différentes sous-classes :

```
01 abstract class ShapeVisitor {
02   def visit (s : Square)
03   def visit (c : Circle)
04   def visit (s : Shape) = s.accept (this)
05 }
06
07 abstract class Shape {
08   def description : String
09   def accept (v: ShapeVisitor)
10
11 }
12
13 class Square extends Shape {
14   def description = "A square"
15   def accept (v: ShapeVisitor) = v visit this
16 }
17
18 class Circle extends Shape {
19   def description = "A circle"
20   def accept (v: ShapeVisitor) = v visit this
21 }
```

Extension des données

Lorsque l'on rajoute une nouvelle sous-classe, il faut étendre le visiteur pour qu'il la prenne en compte :

```
01 abstract class NewShapeVisitor extends ShapeVisitor {
02    def visit (t : Triangle)
03 }
04
05 class Triangle extends Shape {
06    def description = "A triangle"
07    def accept (v: ShapeVisitor) =
08       v match {
09          case v: NewShapeVisitor => v visit this
10       }
11 }
```

Qu'arrive-t-il si on utilise un visiteur de type `ShapeVisitor` et non pas `NewShapeVisitor` pour visiter une instance de la classe `Triangle` ?

Extension des fonctionnalités

```
01 class AreaVisitor extends ShapeVisitor {
02    def visit (s: Square) = ...
03    def visit (s: Circle) = ...
04 }
```

Pour utiliser cette nouvelle fonctionnalité, on peut écrire une fonction de la forme :

```
01 def area (s: Shape) = {
02    (new AreaVisitor) visit s
03 }
```

Remarquez que cette fonction travaille en monde clos : si une nouvelle sous-classe est rajoutée, elle ne sera pas prise en compte.

Problème ouvert : Comment travailler en monde ouvert et toujours permettre de nouvelles extensions de données et de fonctionnalités ?

Remplacement du visiteur par une analyse par motifs

La dernière remarque nous incite à jeter à poubelle le visiteur. Autant, le remplacer par une analyse de motif de la forme :

```
01 def area (s: Shape) =
02   s match {
03     case square: Square => ...
04     case circle : Circle => ...
05   }
06 }
```

Cette fonction est équivalente et ne nécessite pas l'écriture de la classe `ShapeVisitor` et des méthodes associées.

Nous verrons cependant que les `traits` de Scala redonneront de la pertinence au patron de visiteur...

Un problème en suspens. . .

Problème ouvert :
Comment travailler en monde ouvert et toujours permettre de nouvelles extensions de données et de fonctionnalités ?

En d'autres termes, comment **composer les extensions** ?

Plan de la partie *“Construire des objets”*

Situation-problème

Réfléchissons sur la conception d'un logiciel de dessin :

> *Dans le module noyau responsable de la gestion des formes que l'on peut utiliser pour produire une image, on souhaite proposer plusieurs types de formes (un carré, un triangle, un cercle. . .) et aussi plusieurs façon de représenter cette forme (une équation, une énumération de points, une image binaire. . .). Une forme doit proposer une méthode pour s'afficher à une position donnée dans une image binaire. Les cas particuliers de formes doivent apparaître dans l'interface de ce module.*

Comment représenter un cercle représenter par une équation, un carré représenté par une équation, un cercle représenté par une énumération de points, . . . bref, **toute combinaison** d'un cas particulier de forme et de représentation de l'ensemble de ses points ?

Une solution (très mauvaise)

L'idée sans doute la plus maladroite serait de créer une classe par combinaison !

```
abstract class Shape { def draw (p : Position, i : Bitmap) : Unit }
class EquationCircle (val radius: Int) extends Shape {
  def draw (p : Position, i : Bitmap) =
    // Set all point p' in the bitmap
    // such that | p' - p | = radius.
}
class EquationSquare (val dimension : Int) extends Shape {
  def draw (p : Position, i : Bitmap) =
    // Connect the four corner points p'
    // such that | p' - p | = sqrt(2)/2 * dimension.
}
class EnumerationCircle (val radius: Int) extends Shape {
  val points : List[Position] =
    // Compute an enumeration of points that represents
    // a circle.
  def draw (p : Position, i : Bitmap) =
    // For each point p' in points, set p + p'.
}
```

Deux solutions standards

Une première solution consiste à utiliser l'**héritage multiple** si le langage de programmation le propose. Nous allons voir que ce mécanisme pose de nombreux problèmes.

Une autre solution consiste à une utilise une relation de **composition** entre deux classes qui sont en relation parce que la première **délègue** un certain nombre d'opérations à la seconde qui les implémentent concrètement. Ce patron de conception s'appelle le **pont**.

Le pont

```
abstract class Points { def draw (p : Position, i : Bitmap) : Unit }

abstract class Shape {
  val points : Points
  def draw (p : Position, i : Bitmap) : Unit = points.draw (p, i)
}

class Enumeration (l : List[Position]) extends Points {
  def draw (p : Position, i : Bitmap) : Unit = ...
}

class Equation (e : Bitmap => List[Position]) extends Points {
  def draw (p : Position, i : Bitmap) : Unit = ...
}

abstract class PointSetKind
case object EnumerationKind extends PointSetKind
case object EquationKind extends PointSetKind

class Circle (val radius : Int, val kind: PointSetKind) extends Shape {
  val points : Points = kind match {
    case EnumerationKind => new Enumeration (...)
    case EquationKind => new Equation ((b : Bitmap) => ...)
  }
}
```

Différence entre composition et héritage

Rôle et généralisation

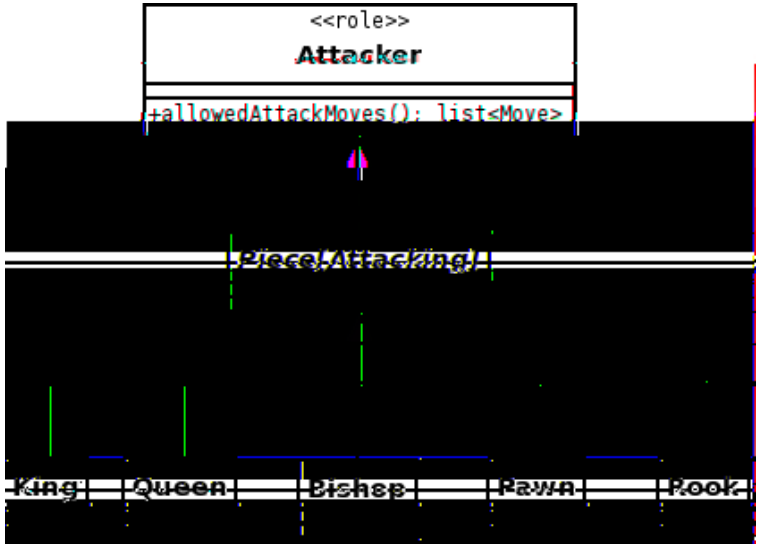
Un objet peut vérifier un prédicat de la forme “Est(X)” pendant un temps **limité**. On dit qu'il a alors un **rôle**. Un rôle sert de classification **dynamique** tandis que les relations de généralisations entre classe fille et classe mère servent de classification statique.

Exemple

Dans un jeu d'échec, lors qu'une pièce tente la prise d'une autre pièce, elle a le rôle d'un attaquant. Le gestionnaire de règle doit vérifier que le coup est valide, en fonction du type de la pièce. Il attend donc d'un attaquant qu'il décrive les règles qui lui sont spécifiques.

Vision logique d'un rôle

En UML, on note les rôles de la façon suivante :

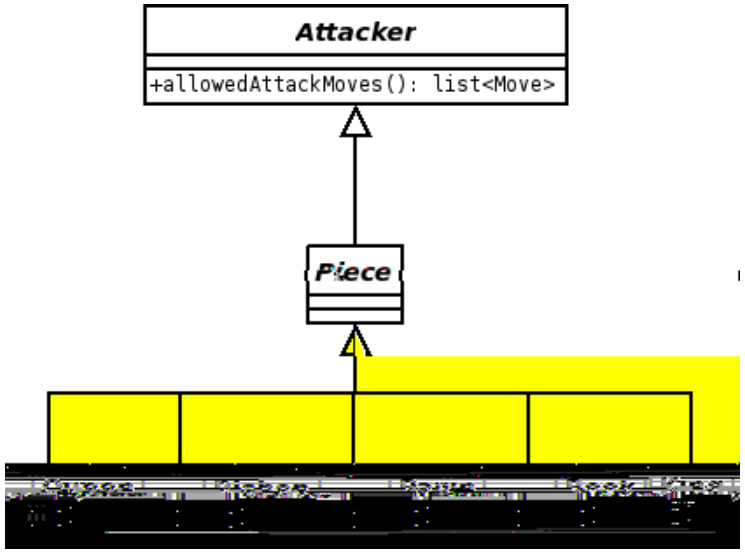


La notation "Piece[Attacking]" signifie que ce diagramme décrire cette classe lorsqu'une de ses instances jouent le rôle d'attaquant.

La flêche pointillée signifie une relation de **dépendance** entre une instance de la classe "Piece" quand elle joue le rôle d'attaquant et la classe qui décrit ce qu'est un attaquant.

Comment **implémenter** cette vue logique de l'architecture ?

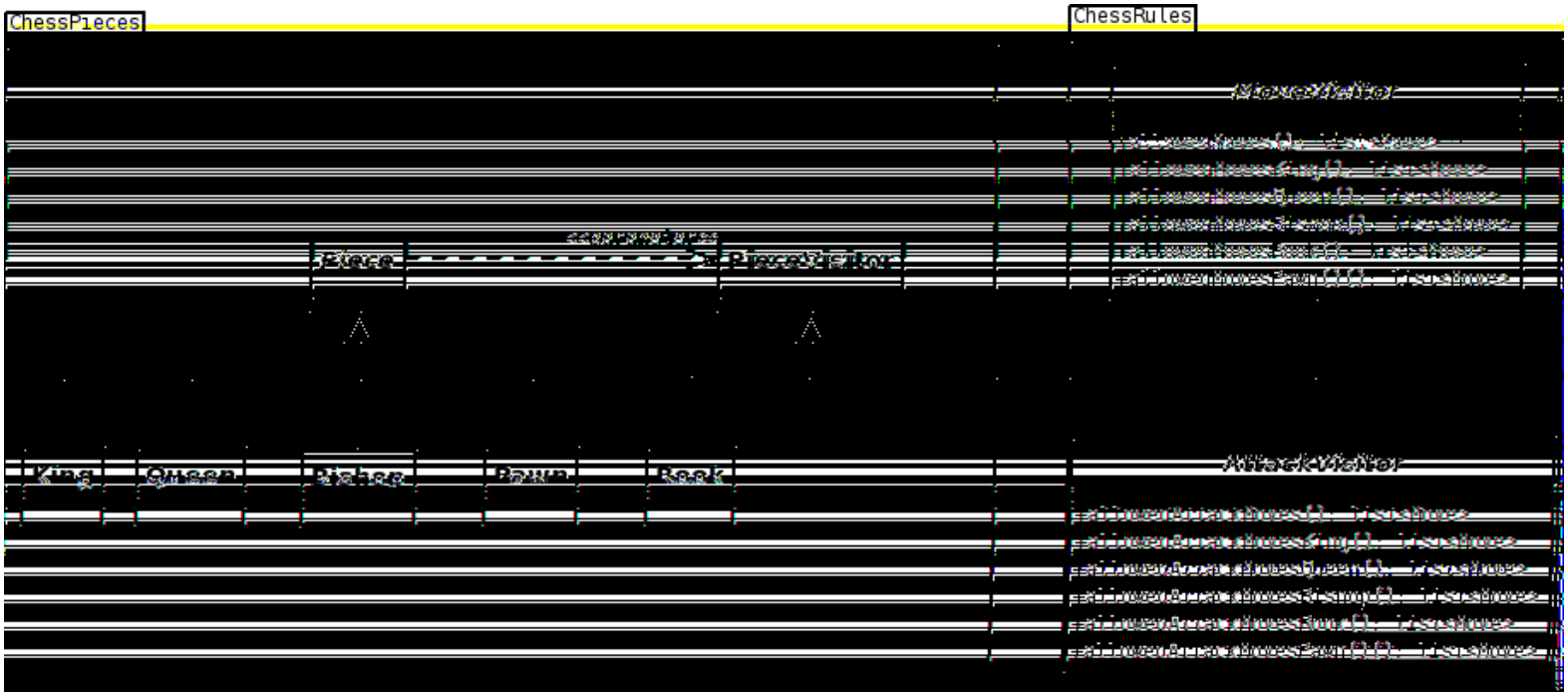
Première (mauvaise) proposition



On a remplacé la relation de dépendance par une relation de généralisation et on traduit cette dernière par un héritage. Dès lors, toutes les sous-classes doivent implémenter la méthode `allowedAttackMoves` qui se confondera certainement avec la méthode `allowedMoves` implémentée par ailleurs.

À l'aide d'une autre architecture, peut-on obtenir une garantie, grâce au typage, que l'on ne peut appeler cette méthode `allowedAttackMoves` qu'en présence d'une pièce ayant le rôle d'un attaquant ?

Seconde proposition



Une première idée consiste à fragmenter les méthodes d'une pièce en extrayant un sous-ensemble de méthode qui peut varier dynamiquement. On peut utiliser deux visiteurs différents pour implémenter les deux types de coups autorisés en fonction du rôle de la pièce.

Critiquez cette architecture !

Troisième proposition : le patron “décorateur”

Une autre façon de voir ce problème est de considérer une pièce attaquante comme une **pièce temporairement décorée** par une fonctionnalité (le calcul des coups d’attaque autorisés).

```
abstract class Shape {
  def allowed_moves : List[Move]
}

class Queen extends Shape { def allowed_moves = Nil }
class Kind extends Shape { def allowed_moves = Nil }

class Attacking[A <: Shape] (shape: A) extends Shape {
  def allowed_attack_moves : List[Move] =
    shape match {
      case as_queen: Queen => Nil
      case as_kind: Kind => Nil
    }
  // Delegate the other methods to the shape
  def allowed_moves = shape.allowed_moves
}
class ChessBoard {
  def display_attack_moves (s : Attacking[Shape]) { ... }
}
```

Garantie statique

L'intérêt de la classe `Attacking` est qu'elle décrit statiquement le rôle de l'objet. Si une fonction attend une pièce attaquante, elle peut maintenant le préciser dans son prototype. En d'autres termes, le programmeur est obligé d'enpaqueter explicitement une pièce dans ce décorateur pour pouvoir l'utiliser comme une pièce attaquante.

Cette solution est assez lourde à utiliser. Pourquoi ?

Garantie statique

L'intérêt de la classe `Attacking` est qu'elle décrit statiquement le rôle de l'objet. Si une fonction attend une pièce attaquante, elle peut maintenant le préciser dans son prototype. En d'autres termes, le programmeur est obligé d'enpaqueter explicitement une pièce dans ce décorateur pour pouvoir l'utiliser comme une pièce attaquante.

Cette solution est assez lourde à utiliser. Pourquoi ? Parce que l'on doit écrire explicitement les délégations des méthodes de la classe `Shape` à l'instance contenue dans l'attribut `shape`. De plus, les méthodes spécifiques aux sous-classes ne sont plus accessible. . .

Solution qui ne fonctionne pas

Malheureusement, en Scala, on ne peut pas écrire :

```
class Attacking[A <: Shape] (shape: A) extends A {
  ...
}
```

qui permettrait d'hériter automatiquement des méthodes de classe A :

```
scala> class Attacking[A <: Shape] (shape: A) extends A {}
<console>:6: error: class type required but A found
        class Attacking[A <: Shape] (shape: A) extends A {}
```

D'ailleurs, cela ne règlerait pas tout à fait le problème puisqu'on aimerait aussi dire que `this` est en fait `shape`...

Heureusement, les traits vont offrir une solution élégante à ce problème.

L'héritage multiple

Revenons maintenant sur l'héritage multiple. C'est un mécanisme qui permet d'hériter de plusieurs classes en même temps. En théorie, ce serait une solution idéale à notre problème. . . mais en pratique, c'est un mécanisme qui souffre de nombreux défauts.

Le problème de l'ambiguité

Voici un exemple tiré de *Programming in Scala* de Martin Odersky :

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBu□er[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
class Doubling extends IntQueue {
  override def put(x: Int) { super.put(2 * x) }
}
class Incrementing extends IntQueue {
  override def put(x: Int) { super.put(x + 1) }
}

// The following code is not accepted in Scala
class MyQueue extends Doubling and IntQueue {}
```

Quelle serait la méthode `put` de la classe `MyQueue` ?

Le problème du diamant

La tentative de suppression de l'ambiguïté suivante :

```
class MyQueue extends Doubling and IntQueue {
  def put(x: Int) {
    Incrementing.super.put(x) // (Not real Scala)
    Doubling.super.put(x)
  }
}
```

fait apparaître un autre problème : la méthode `put`

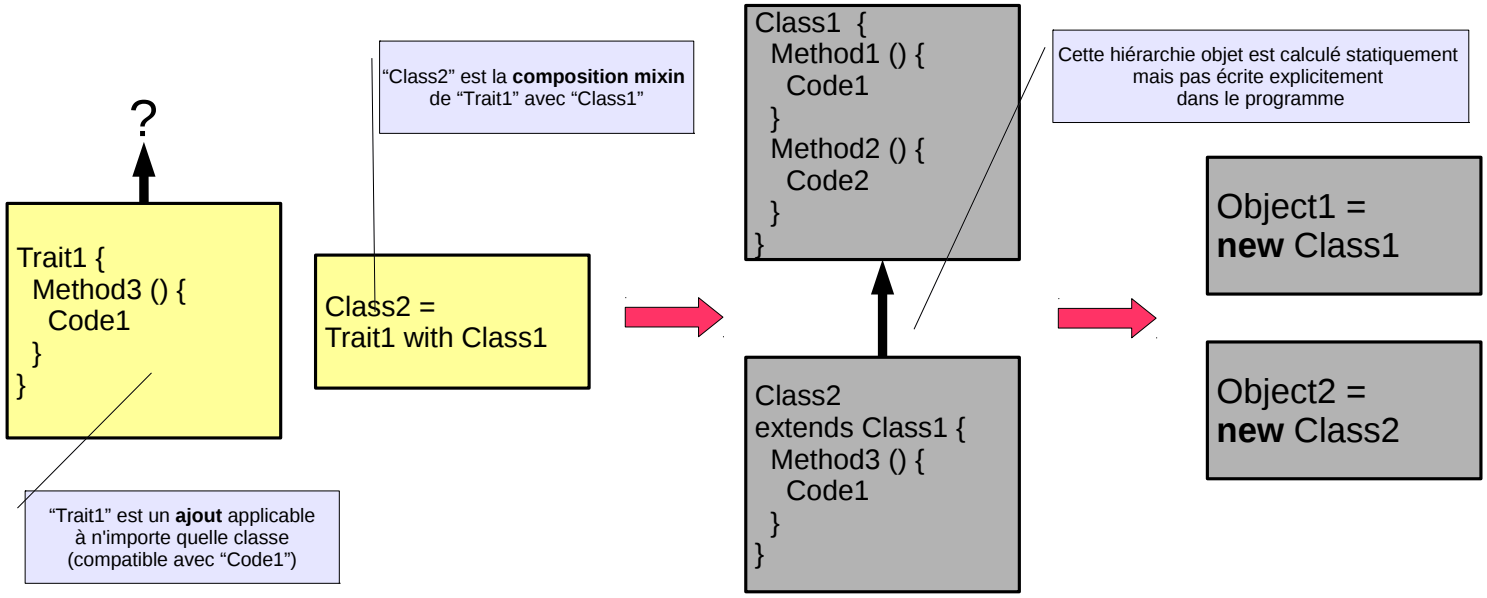
Les problèmes d'implémentation

L'héritage multiple pose enfin des problèmes d'implémentation dans les compilateurs. (Souvenez de l'héritage `public virtual` de C++.) En effet, déterminer statiquement la disposition des champs des différents attributs de l'objet est difficile car la relation d'héritage ne définit plus un ordre total. (La solution par C++ nécessite une modification silencieuse du pointeur qui tient l'objet lorsque l'on veut voir une sous-classe du point de vue d'une de ses classes mères.)

Plan de la partie *“Construire des objets”*

Plan de la partie *“Construire des objets”*

Construire des hiérarchies par composition mixin



Qu'est-ce qu'un trait ?

Un **trait** représente une **modification** (un ou plusieurs ajouts ou redéfinitions de méthodes) à appliquer à une classe.

Lorsque l'on écrit un trait, il faut considérer que l'on est paramétré par l'instance de sa super-classe : `super` est inconnu. Habituellement, la définition de l'instance que l'on spécifie quand on écrit une classe, est seulement paramétrée par l'instance d'une éventuelle sous-classe.

Les traits sont donc doublement récursifs.

Composition *mixin* avec un trait

Pour instancier un trait, il faut utiliser le mécanisme de **composition mixin**. Il s'agit de donner la classe mère sur laquelle le trait, la modification, doit être appliqué. Ainsi :

```
class A { def n = ... }

trait C { def m = ... }

class B extends A with C {}
```

est (à peu près) équivalent à :

```
class A { def n = ... }
class B extends A {
  def m = ...
}
```

Un trait est donc une **changement incrémental réutilisable**.

Création de classe à la volée

On peut appliquer la composition mixin au moment de l'instanciation ; ce qui évite de créer explicitement des classes (et de leur donner un nom) :

```
class A { def n = ... }

trait C { def m = ... }

val x = new A with C
```

Redéfinition et composition *mixin*

Que se passe-t-il dans le programme suivant ?

```
class A { def n = println ("A") }
trait C { override def n = println ("C") }
trait D { override def n = println ("D") }
val x = (new A with C with D).n
```

Redéfinition et composition *mixin*

Que se passe-t-il dans le programme suivant ?

```
class A { def n = println ("A") }
trait C { override def n = println ("C") }
trait D { override def n = println ("D") }
val x = (new A with C with D).n
```

Scala rejette le programme, on ne sait pas quelle méthode `n` de quelle classe, on est en train de redéfinir.

Redéfinition et composition *mixin*

`trait C extends A` signifie que la classe mère de `C` est au une sous-classe de `A` :

```
class A { def n = println ("A") }
trait C extends A { override def n = println ("C") }
trait D extends A { override def n = println ("D") }
val x = (new A with C with D).n
```

Qu'affiche ce programme ?

Redéfinition et composition *mixin*

`trait C extends A` signifie que la classe mère de `C` est au une sous-classe de `A` :

```
class A { def n = println ("A") }
trait C extends A { override def n = println ("C") }
trait D extends A { override def n = println ("D") }
val x = (new A with C with D).n
```

Qu'affiche ce programme ? `"D"` car l'ordre suivant lequel la composition est effectuée compte !

Redéfinition et composition *mixin*

Que se passe-t-il si on fait un appel à une méthode abstraite dans un trait ?

```
class A { abstract def n : Unit }
trait C extends A { override def n = { super.n ; println ("C") } }
trait D extends A { override def n = { super.n ; println ("D") } }
val x = (new A with C with D).n
```

Redéfinition et composition *mixin*

Que se passe-t-il si on fait un appel à une méthode abstraite dans un trait ?

```
class A { abstract def n : Unit }
trait C extends A { override def n = { super.n ; println ("C") } }
trait D extends A { override def n = { super.n ; println ("D") } }
val x = (new A with C with D).n
```

`Scala` rejette ce programme : comment s'assurer que la méthode `super` est bien définie ?

Redéfinition et composition *mixin*

On peut indiquer que l'on promet que le trait sera composé avec une sous-classe de `A` qui implémente `n`.

```
class A { abstract def n : Unit }
trait C extends A { abstract override def n = { super.n ; println
("C") } }
trait D extends A { abstract override def n = { super.n ; println
("D") } }
class E extends A { override def n = println ("E") }
val x = (new E with C with D).n
```

Redéfinition et composition *mixin*

On peut indiquer que l'on promet que le trait sera composé avec une sous-classe de `A` qui implémente `n`.

```
class A { abstract def n : Unit }
trait C extends A { abstract override def n = { super.n ; println
("C") } }
trait D extends A { abstract override def n = { super.n ; println
("D") } }
class E extends A { override def n = println ("E") }
val x = (new E with C with D).n
```

Scala rejette ce programme : comment s'assurer que la méthode `super` est bien définie ?

Le mécanisme d'héritage entre traits

Un trait peut aussi hériter d'un autre trait :

```
trait A { def n = ... }
trait B extends A { def m = ... }
```

Il s'agit juste ici d'étendre des modifications par d'autres modifications.

Le Graal de la modularité

Nous avons voir dans les prochains cours que les traits favorisent la programmation modulaire et extensible.

Exercice pour la prochaine fois : en quoi les traits apportent une solution au problème évoqué plus tôt sur les visiteurs ?

Plan de la partie *“Construire des objets”*

Retour sur l'exemple

Comment utiliser les traits de Scala pour simuler l'héritage multiple ?

```
abstract class IntQueue {
  def get(): Int
  def put(x: Int)
}
class BasicIntQueue extends IntQueue {
  private val buf = new ArrayBu□er[Int]
  def get() = buf.remove(0)
  def put(x: Int) { buf += x }
}
class Doubling extends IntQueue {
  override def put(x: Int) { super.put(2 * x) }
}
class Incrementing extends IntQueue {
  override def put(x: Int) { super.put(x + 1) }
}

// The following code is not accepted in Scala
class MyQueue extends Doubling and IntQueue {}
```

Retour sur l'exemple

```
trait Doubling extends IntQueue {
  abstract override def put (x: Int) { super.put (2 * x) }
}

trait Incrementing extends IntQueue {
  abstract override def put (x: Int) { super.put (x + 1) }
}

class MyQueue extends BasicIntQueue with Doubling with Incrementing

object test {
  def main (args: Array[String]) {
    val q = new MyQueue
    q.put (1)
    println (q.get)
  }
}
```

Qu'affiche ce programme ?

Retour sur l'exemple

```
trait Doubling extends IntQueue {
  abstract override def put (x: Int) { super.put (2 * x) }
}

trait Incrementing extends IntQueue {
  abstract override def put (x: Int) { super.put (x + 1) }
}

class MyQueue extends BasicIntQueue with Doubling with Incrementing

object test {
  def main (args: Array[String]) {
    val q = new MyQueue
    q.put (1)
    println (q.get)
  }
}
```

Qu'affiche ce programme ? 4 !

Une pile de modifications

La composition mixin permet d'appliquer une pile de modification à une classe. Par construction, il n'y a qu'un unique chemin dans la hiérarchie de classe entre la classe obtenue et la classe initiale, ce qui élimine le problème du diamant par construction. Ce mécanisme d'héritage s'appelle linéarisation.

Linéarisation

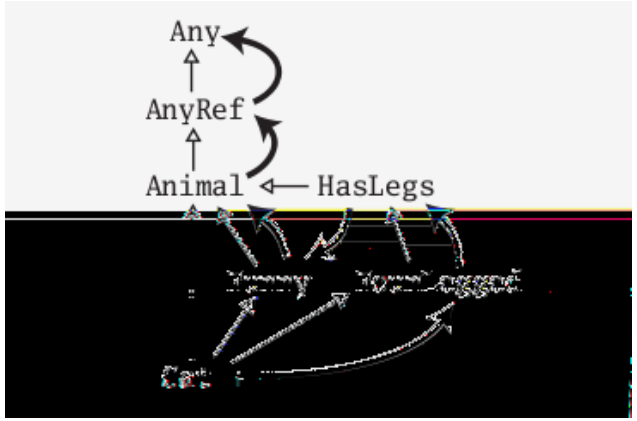
Exemple tiré de *Programming in Scala* de Martin Odersky

```
class Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends HasLegs
class Cat extends Animal with Furry with FourLegged
```

Quelles hiérarchies de classe sont construites par ces compositions mixins ?

Linéarisation

Exemple tiré de *Programming in Scala* de Martin Odersky



- Animal → AnyRef → Any
- Furry → Animal → AnyRef → Any
- Cat → FourLegged → HasLegs → Furry → Animal → AnyRef → Any

Sous-typage nominal pour les traits ?

Par défaut, Scala offre un sous-typage **nominal** qui s'appuie sur les relations de sous-typage entre instances de classe définies par les hiérarchies. Pour étendre le sous-typage aux classes construites avec des traits, il faut pouvoir écrire des types pour les représenter.

Par exemple, si on a un trait `A` :

```
trait A { def x : Int }
```

On peut parler des types des instances de classe construite à l'aide de `A` ainsi :

```
def f (a : A) { println (a.x) }
```

Et si on se donne un autre trait de la forme `trait B { def x : Int }`, est-ce que `f` est compatible avec les classes composées avec `B` ?

Des domaines de fonctions très précis

Les types de la forme `A with B` permettent de donner des spécifications aux domaines des fonctions qui mélangent les notions de sous-typage nominal et structurel. Par exemple, on peut parler de toutes les sous-classes de `A` qui possèdent une méthode `x` de la façon suivante :

```
class A {
  def f (x : Int) = x + 1
}
class B extends A { def x : Int = 42 }
class C extends A { def y : Int = 42 }
object test {
  def g (a : A { def x : Int }) = a.f (a.x)

  g (new B)
  // g (new C) // Rejected
  g (new A { def x = 42 })
}
```

Traits paramétrés par des types

Tout comme les classes, les traits peuvent être paramétrés par des types.

La syntaxe Scala est de la forme :

```
trait A[T] {
  // ...
}
```

On peut imposer des **contraintes de sous-typage** sur ces paramètres :

```
trait B
trait A[T <: B] {}
trait C[T <: B { def x : Int }] {}
```

Exemple : le trait `Ordered[T]`

```
trait Ordered[A] {
  def compare(that: A): Int

  def < (that: A): Boolean = (this compare that) < 0
  def > (that: A): Boolean = (this compare that) > 0
  def ≤ (that: A): Boolean = (this compare that) ≤ 0
  def ≥ (that: A): Boolean = (this compare that) ≥ 0
  def compareTo (that: A): Int = compare(that)
}
```

En quoi ce trait est-il utile ?

Exemple de trait paramétré avec borne récursive

Voici un drôle de trait :

```
trait A[S <: A[S]] {
  // ...
}
```

À quoi peut-il servir ?

Traits paramétrés par un paramètre modélisant le type de *Self*

```
trait A[S <: A[S]] {
  def self : S
  def x : Int = 42
}

trait B[S <: B[S]] extends A[S] {
  def self : S
  def y : Int = 21
}

class C extends B[C] {
  def self = this
}
```

Dans quelle situation la connaissance du type le plus précis de l'objet peut elle être intéressante ?

Chaînage de méthode

Voici deux classes écrites en objet « classique » :

```
class A {
  var x = 0
  def incr = { x += 1 ; this }
}
class B extends A {
  def decr = { x -= 1 ; this }
}
object test {
  val x = new B
  x.incr.decr
  // Error: error: value decr is not a member of A
}
```

Le type le plus précis de l'instance `x` a été oublié lors de l'appel à la méthode `incr`. Le compilateur rejette ce programme pourtant licite !

Solution au problème de chaînage de méthodes

```
trait A[S <: A[S]] {
  var x : Int = 0
  def self : S
  def incr : S = { x += 1 ; self }
}
trait B[S <: B[S]] extends A[S] {
  def decr : S = { x -= 1 ; self }
}
class C extends B[C] {
  def self = this
}
object test {
  val x = new C
  x.incr.decr
}
```

Extension de l'idée

On peut faire des hypothèses supplémentaires sur `S`, c'est-à-dire sur la façon dont la classe `A` sera étendue :

```
// All subclasses of A must eventually define a method y.
trait A[S <: A[S] { def y : Int } ] {
  def self : S
  def x : Int = self.y
}

trait B[S <: B[S]] extends A[S] {
  def self : S
  def y : Int = 21
}

class C extends B[C] {
  def self = this
}
```

Traits paramétrés et type *Self*

L'utilisation d'un paramètre de type représentant de la type le plus précis de `this` est très pratique pour exprimer des contraintes sur les futurs héritages et en tirer parti dans la classe mère.

Pour en faciliter l'usage, une syntaxe spéciale a été introduite en Scala.

L'exemple précédent s'écrit de la façon suivante :

```
trait A { self : A { def y : Int } =>
  def x : Int = y
}

class B extends A { def y : Int = 42 }
```

Définition modulaire d'une structure d'arbre

Un arbre est composé d'une étiquette et d'un ensemble de sous-arbre.

On suppose de plus que chaque étiquette a une arité, c'est-à-dire un entier naturel. Pour être bien formé, un arbre doit respecter l'arité de ses étiquettes : l'ensemble de sous-arbre associé à une étiquette doit avoir pour cardinal l'arité de l'étiquette.

À l'aide de traits, comment obtenir une définition modulaire d'arbre qui permet de faire varier indépendamment la représentation de l'étiquette et la représentation de l'ensemble des sous-arbres ?

Une définition modulaire des arbres

```
trait Tree {
  type Label
  def arity : Int
  def label : Label

  type Children
  def children : Children
  def childrenCount : Int
  def childrenGet (x: Int) : Tree

  def labelsOfPath (path : List[Int]) : List[Label] =
    path match {
      case Nil =>
        List (label)
      case x :: xs => {
        label :: childrenGet (x).labelsOfPath (xs).asInstanceOf[List[Label]]
      }
    }
  def toString : String
}
```

Notez que les types `Label` et `Children` sont abstraits.

Une implémentation pour les étiquettes

Voici une implémentation particulière des étiquettes par des entiers. On choisit un nouveau numéro pour chaque nouvelle instance. L'arité de tous les nœuds est fixée à 2.

```
var count = 0
trait AutomaticIntBinaryLabel { self: Tree =>
  type Label = Int
  val label = { count += 1 ; count }
  def arity = 2
  override def toString =
    label.toString +
    (if (childrenCount == 0) ""
     else "(" + children.toString + ")")
}
```

Pour pouvoir afficher l'arbre, il suffit de savoir afficher une étiquette. En fixant le type de *self* à `Tree`, on peut utiliser les méthodes travaillant sur les sous-arbres comme `childrenCount` par exemple.

Une implémentation d'une forêt d'arbres

Voici une implémentation de forêt à l'aide de liste.

```
trait ListChildren { self: Tree =>
  assert (arity == childrenCount)
  type Children = List[Tree]
  def childrenCount = children.length
  def childrenGet (x: Int) = children (x)
  def children : Children
}
```

Notez comme cette définition fait référence à la méthode `arity` définie dans la classe précédente. Encore une fois, cette référence est permise puisque l'on promet que la classe finale obtenue après composition *mixin* sera de type `Tree`, comme l'indique l'annotation de type de *self*.

Classe finale par composition mixin

Les deux définitions précédentes sont **mutuellement récursives**. On peut néanmoins les composer de façon à obtenir le point fixe de leurs définitions :

```
class TreeImpl (val children : List[TreeImpl]) extends Tree
with AutomaticIntBinaryLabel
with ListChildren
```

Grâce à la composition mixin, on peut définir des fragments de classes avec une granularité très fine car ils peuvent faire références les uns aux autres d'une façon très libre. Ce mécanisme s'appuie sur les annotations de type *self* qui déclarent *a priori* la forme de la classe finale.

Retour sur le visiteur modulaire

Problème ouvert :
Comment travailler en monde ouvert et toujours permettre de nouvelles extensions de données et de fonctionnalités ?

En d'autres termes, comment **composer les extensions** ?

Existence d'un visiteur sur une hiérarchie de classe

```
trait ShapesBase {

  abstract class Shape {
    def accept (v: ShapeVisitor)
  }

  type ShapeVisitor <: AbstractShapeVisitor
  trait AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s : Shape) = s.accept (this)
  }
}
```

On introduit **un type virtuel** qui va être celui d'un visiteur travaillant sur une hiérarchie dont la base est `Shape`. Le trait `AbstractShapeVisitor` représente l'interface que doit implémenter un visiteur concret travaillant sur cette hiérarchie (qui n'a pas de sous-classe pour l'instant).

Première extension de données

```
trait ShapesLibExtension1 extends ShapesBase {
  class Circle extends Shape {
    def accept (v: ShapeVisitor) = v.visit (this)
  }

  type ShapeVisitor <: AbstractShapeVisitor
  trait AbstractShapeVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s : Circle) : Unit
  }
}
```

On raffine ici le type des visiteurs sur cette nouvelle hiérarchie : un visiteur doit maintenant traiter un cas supplémentaire, celui des cercles.

Première extension de fonctionnalité

```
trait ShapesLibExtension1_2 extends ShapesLibExtension1 {
  trait DescriptionVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s: Circle) = println ("Here is a circle!")
  }
}
```

Pour implémenter un visiteur qui travaille sur cette hiérarchie, il suffit de créer une sous-classe concrète de la classe `AbstractShapeVisitor`. Pour pouvoir l'instancier, il faut fixer le type virtuel :

```
object ShapesLibVersion1_2 extends ShapesLibExtension1_2 {
  type ShapeVisitor = AbstractShapeVisitor
  class DescriptionVisitor extends super.DescriptionVisitor
}
```

Seconde extension des données

```
trait ShapesLibExtension2 extends ShapesBase {
  class Square extends Shape {
    def accept (v: ShapeVisitor) = v.visit (this)
  }

  type ShapeVisitor <: AbstractShapeVisitor
  trait AbstractShapeVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s : Square) : Unit
  }
}
```

Remarquez que cette extension-ci est **indépendante** de la première.

Seconde extension de fonctionnalités

```
trait ShapesLibExtension2_2 extends ShapesLibExtension2 {
  trait DescriptionVisitor extends super.AbstractShapeVisitor {
    self: ShapeVisitor =>
    def visit (s: Square) = println ("Here is a square!")
  }
}

object ShapesLibVersion2_2 extends ShapesLibExtension2_2 {
  type ShapeVisitor = AbstractShapeVisitor
  class DescriptionVisitor extends super.DescriptionVisitor
}
```

Par le même schéma que précédemment, on instancie un visiteur qui travaille sur une hiérarchie qui contient une classe `Square`

Question centrale

Comment produire une hiérarchie possédant ces deux types de données et un visiteur correspondant à la composition des deux visiteurs spécifiques à ces deux types ?

Une première solution

En utilisant la composition *mixin*, on fait l'**union** les deux hiérarchies.

```
trait ShapesLibExtension12
  extends ShapesLibExtension1
  with ShapesLibExtension2 {

  type ShapeVisitor <: AbstractShapeVisitor

  trait AbstractShapeVisitor
  extends super[ShapesLibExtension1].AbstractShapeVisitor
  with super[ShapesLibExtension2].AbstractShapeVisitor {
    self: ShapeVisitor =>
  }

  trait DescriptionVisitor
  extends super[ShapesLibExtension2].AbstractShapeVisitor
  with super[ShapesLibExtension1].AbstractShapeVisitor {
      self: ShapeVisitor =>
      def visit (s: Square) = println ("Here is a square!")
      def visit (s: Circle) = println ("Here is a circle!")
  }
}
```

Première solution : concrétisation

```
object ShapesLibVersion12 extends ShapesLibExtension12 {
  type ShapeVisitor = AbstractShapeVisitor
  class DescriptionVisitor extends ShapeVisitor
  with super.DescriptionVisitor {
    self: ShapeVisitor =>
  }
}

object testShapesLibVersion12 {
  def main (args: Array[String]) {
    import ShapesLibVersion12._
    (new DescriptionVisitor).visit (new Square)
    (new DescriptionVisitor).visit (new Circle)
  }
}
```

Seconde solution

On peut faire mieux en composant aussi les visiteurs :

```
trait ShapesLibExtension12_bis
  extends ShapesLibExtension1_2
  with ShapesLibExtension2_2 {

  type ShapeVisitor <: AbstractShapeVisitor

  trait AbstractShapeVisitor
  extends super[ShapesLibExtension1_2].AbstractShapeVisitor
  with super[ShapesLibExtension2_2].AbstractShapeVisitor {
    self: ShapeVisitor =>
  }

  trait DescriptionVisitor
  extends super[ShapesLibExtension1_2].DescriptionVisitor
  with super[ShapesLibExtension2_2].DescriptionVisitor {
      self: ShapeVisitor =>
  }
}
```

Pourquoi est-ce que cela compose bien le comportement des visiteurs ?

Seconde solution : concrétisation

```
object ShapesLibVersion12_bis
extends ShapesLibExtension12_bis {
  type ShapeVisitor = AbstractShapeVisitor
  class DescriptionVisitor extends ShapeVisitor
  with super.DescriptionVisitor {
    self: ShapeVisitor =>
  }
}


object testShapesLibVersion12_bis {
  def main (args: Array[String]) {
    import ShapesLibVersion12_bis._
    (new DescriptionVisitor).visit (new Square)
    (new DescriptionVisitor).visit (new Circle)
  }
}
```

Plan de la partie *“Construire des objets”*

Les patrons de conception : des « recettes »

- Un programmeur ou un concepteur expérimenté réutilise des procédés d'organisation qui ont fonctionné dans le passé.
- Ces solutions sont indépendantes (en général) du langage objet utilisé.
- Votre expérience :
  - XHTML + CSS : séparer un contenu de la description de sa mise en forme.

    Idée à retenir :
    - un composant traitant les données (modèle),
    - un composant de visualisation (vue).
  - Compilateur : composition de traductions bien spécifiées.

    Idée à retenir :
    - la fameuse *separation of concerns* de Dijkstra.
    - "Un composant bien conçu se focalise sur un unique problème."

~Menu~

Création

Fabrique abstraite (Abstract Factory)
Monteur (Builder)
Fabrique (Factory Method)
Prototype (Prototype)
Singleton (Singleton)

Structure

Adaptateur (Adapter)
Pont (Bridge)
Objet composite (Composite)
Décorateur (Decorator)
Façade (Facade)
Poids-mouche ou poids-plume (Flyweight)
Proxy (Proxy)

Comportement

Chaîne de responsabilité (Chain of responsibility)
Commande (Command)
Interpréteur (Interpreter)
Itérateur (Iterator)
Médiateur (Mediator)
Memento (Memento)
Observateur (Observer)
État (State)
Stratégie (Strategy)
Patron de méthode (Template Method)
Visiteur (Visitor)

- Chefs : Gamma, Helm, Johnson, Vlissides (GoF).
- Livre de recettes (référence) : *Design Patterns : Elements of Reusable Object-Oriented Software.* Addison-Wesley

  (source : Wikipedia)

Classification des patrons de conception

- GoF ont explicité trois grandes classes de patrons dans leur livre, chacune spécialisée dans :
  - la création d'objets ;
  - la structure des relations entre objets ;
  - le comportement des objets.
- D'autres catégories ont été repertoriées pour la destruction des objets, la concurrence, la persistance ...

Patrons de conception de « Création et Destruction »

Fabrique (Factory Method) – Description

- Exemple de situation :

  Je veux fournir un composant réutilisable implémentant un dictionnaire. Ce dictionnaire doit être implémenté à l'aide de la structure de données la plus adaptée possible compte tenu du nombre d'entrées qu'il va contenir (par exemple, une table de hachage tant que sa taille n'excède pas quelques mégas, une base de données sinon). Les prochaines versions de mon composant intègreront de nouvelles structures de données. J'utilise donc une classe abstraite Dictionnary et une sous-classe par structure de données.

- Problème : Comment bien choisir la structure de données sans rendre publique l'existence des sous-classes ?
- Solution : On fournit une classe *DictionaryFactory* qui répond au message *CreateDictionary (n)* où *n* est le nombre potentiel d'entrées dans le dictionnaire. L'implémentation de cette classe choisit la bonne sous-classe et en retourne une instance vue comme un *Dictionary*. On ne divulgue donc que *DictionaryFactory* et *Dictionary*.

Fabrique (Factory Method) – en UML

Dessiner le diagramme de classes correspondant !

Fabrique (Factory Method) – Interface en C++

```
// Dictionary from string to string.
class Dictionary {
public:
   struct NotFound {};

   virtual Dictionary*
   addEntry (const std::string& key, const std::string& value) = 0;

   virtual const std::string&
   operator[](const std::string& key) const throw (NotFound) = 0;
};
// Dictionary factory is choosing the best implementation
// given the potential number of entries.
class DictionaryFactory {
public:
   // You are responsible for the life of the resulting dictionary.
   Dictionary* create (long int potential_entry_number);
};
```

Fabrique (Factory Method) – Implémentation en C++

```
class RedBlackTreeDictionary : public Dictionary {
  virtual Dictionary*
  addEntry (const std::string& key, const std::string& value) { ... }

  virtual const std::string&
  operator[](const std::string& key) const throw (NotFound) { ... }
};

class DataBaseDictionary : public Dictionary {
  virtual Dictionary*
  addEntry (const std::string& key, const std::string& value) { ... }

  virtual const std::string&
  operator[](const std::string& key) const throw (NotFound) { ... }
};

Dictionary* DictionaryFactory::create (long int potential_entry_number) {
  if (potential_entry_number ≤ 100000000L)
    return (new RedBlackTreeDictionary ());
  else
    return (new DataBaseDictionary ());
}
```

Fabrique (Factory Method)

- Une fabrique peut aussi servir à différencier les constructeurs de classe à l'aide d'un nom.
- En effet, il n'est pas toujours évident de dissocier les constructeurs seulement à l'aide de leur signature (*ie* du types des arguments).
- Dans notre exemple, on peut vouloir construire un dictionnaire à l'aide d'un entier correspondant :
  - au nombre moyen d'entrées ;
  - ou au nombre minimal d'entrées ;
  - ou au nombre maximal d'entrées.
- Les constructeurs auraient tous la même signature !
- Dans ce cas, on peut plutôt donner un nom explicite aux messages de la fabrique pour dénoter le cas de construction (`createFromMean`, `createFromMin`, `createFromMax`).

Fabrique (Factory Method) – Vision fonctionnelle

- Introduire une indirection pour la construction des données est un procédé d'encapsulation très courant dans tous les paradigmes de programmation.
- Lorsque le langage de programmation fournit des types abstraits, on implémente des « constructeurs intelligents » (*smart constructor*).
- En O'Caml :

```
module Dictionary : sig
  type t (* Abstract type of dictionaries. *)
  val mk_dictionary : int → t (* Smart constructor. *)
  (* Functionalities *)
  val add_entry : string → string → t → t
  val get_entry : string → t → t
end =
struct
  type t =
   | RedBlackTreeDictionary of RedBlackTreeDictionary.t
   | DataBaseDictionary of DataBaseDictionary.t

  let mk_dictionary nb_potential_entries =
    if nb_potential_entries ≤ 10000000L then
      RedBlackTreeDictionary RedBlackTreeDictionary.empty
    else
      DataBaseDictionary DataBaseDictionary.empty
end
```

Fabrique abstraite (Abstract Factory) – Description

- Exemple de situation :

  Je veux distribuer une nouvelle version de mon composant : les dictionnaires peuvent maintenant être persistants (la méthode `addEntry` renvoie une nouvelle instance du dictionnaire sur le tas et ne détruit pas la précédente).

- Problème : Comment minimiser la quantité de code à modifier pour assurer la migration vers la nouvelle version (sans pour autant divulguer les sous-classes mise en jeu) ?
- Solution : Appliquer le motif de Fabrique à la Fabrique elle-même ! La classe *DictionaryFactory* devient abstraite. On crée une classe *PersistantDictionaryFactory* et une classe *NonPersistantDictionaryFactory*. On peut ensuite définir une classe *DictionaryFactoryCreator* qui choisit d'instancier la bonne fabrique en fonction d'un booléen.

Abstraction de l'allocation et gestion explicite de la mémoire

```
typedef std::list< std::pair<std::string, std::string> > entry_list;

Dictionnary*
add_list (Dictionnary* dict, const entry_list& entries) {
   Dictionnary* accu = dict;
   for (entry_list::iterator i = entries.begin (); i != entries.end (); ++i)
      accu = accu→add_entries (i→first, i→second);
   return (accu);
}
```

Quelle est l'empreinte mémoire de l'exécution de ce programme ?

Pointeur intelligent (Smart Pointer) – Description

(ce patron s'applique seulement si le langage d'implémentation ne propose pas de ramasse-miettes)

- Exemple de situation :

  Après leur migration à la version persistante de mon composant, les utilisateurs se plaignent de la présence de fuites de mémoire.

- Problème : Lors le mécanisme d'allocation est abstrait, on ne peut pas demander au client de désallouer l'instance. En d'autres termes, il faut aussi implémenter un mécanisme abstrait de désallocation.
- Solution : Les objets implémentent le comportement de pointeurs munis d'un mécanisme de destruction automatique lorsque l'on ne les utilise plus. Il existe plusieurs implémentations de ces pointeurs « intelligents » :
  - les auto-pointeurs (présents dans la bibliothèque standard de C++) ;
  - les compteurs de références (présents dans la bibliothèque Boost).

Pointeur intelligent (Smart Pointer)

- Dans notre exemple, la méthode *addEntry* renvoie désormais un pointeur intelligent vers un dictionnaire.

  boost::shared_ptr<Dictionary> addEntry (...) ;

- Il est malheureusement nécessaire de modifier le code client car la signature de la méthode *addEntry* a changé.

Monteur (Builder) – Description

- Exemple de situation :

  Pour obtenir une partie d'échec pleinement initialisée, je dois avoir un plateau, attendre que deux joueurs soient prêts à jouer, s'accorder sur le type de partie voulue, et enfin déterminer la couleur de chacun des joueurs.
  Mon programme permet de jouer sur différentes formes de plateau, les joueurs peuvent se connecter localement ou via un serveur, il y a 30 types de partie proposés . . .

- Problème : Comment décorréler le protocole de création d'une partie de la nature de ces parties ?
- Solution : Un objet `GameDirector` fait office de fabrique implémentant le protocole en appelant les fabriques des différents composants de la partie dans le bon ordre.

Prototype (Prototype) – Description

- Exemple de situation : On veut créer des modèles de partie d'échec (par exemple, une partie d'échec en Blitz sur un plateau en 3 dimensions avec deux joueurs sur le réseau ou encore une partie d'échec sans limitation de temps sur un plateau en ASCII avec un joueur local jouant contre un programme, ...).
- Problème : Doit-on faire une classe de fabrique par combinaison ?
- Solution : Pour éviter la multiplication des sous-classes, on représente une combinaison particulière des paramètres à l'aide d'une instance d'une classe *GamePrototype* qu'on est capable de cloner.

Prototype (Prototype) – En Scala (1/2)

```
abstract class Board { def cloneMe () : Board }
class Board2d extends Board { override def cloneMe () = this }
class Board3d extends Board { override def cloneMe () = this }

abstract class Kind { def cloneMe () : Kind }
class Blitz extends Kind { override def cloneMe () = this }
class Standard extends Kind { override def cloneMe () = this }

abstract class Player { def cloneMe () : Player }
class LocalPlayer extends Player { override def cloneMe () = this }
class AIPlayer extends Player { override def cloneMe () = this }

class Game (board : Board, kind : Kind, player_1 : Player, player_2 : Player) {
  var _board : Board = board
  var _kind : Kind = kind
  var _player_1 : Player = player_1
  var _player_2 : Player = player_2

  def play () = ...
}
```

Prototype (Prototype) – En Scala (2/2)

```
class GamePrototypeFactory (board : Board, kind : Kind, player_1 : Player, player_2 : Player) {
  var _board : Board = board
  var _kind : Kind = kind
  var _player_1 : Player = player_1
  var _player_2 : Player = player_2

  def makeGame () = new Game (_board.cloneMe (), _kind.cloneMe (),
                                _player_1.cloneMe (), _player_2.cloneMe ())
}

val standard_game_factory =
  new GamePrototypeFactory (new Board2d, new Standard, new LocalPlayer, new LocalPlayer)

val alone_game_factory =
 new GamePrototypeFactory (new Board2d, new Standard, new LocalPlayer, new AIPlayer)

val blitz_game_factory =
 new GamePrototypeFactory (new Board2d, new Blitz, new LocalPlayer, new AIPlayer)
```

Prototype (Prototype) – Dans un langage fonctionnel

- Dans un cadre fonctionnel, on peut composer les constructeurs pour obtenir de nouveaux constructeurs (ce sont des objets de première classe).
- En O'Caml, si on a :

Singleton (Singleton) – Description

- Exemple de situation :

  Un programme s'exécute dans un environnement qui a une existence physique particulière (un système d'exploitation, un numéro de processus, . . .). On veut le représenter par un objet qui est une instance d'une classe `Environment`.

- Problème : Pour une exécutation donnée, il ne doit exister qu'une unique instance de la classe `Environment`.
- Solution : On empêche l'instanciation explicite (par un constructeur) de la classe `Environment`. Une classe `EnvironmentSingleton` est fournie et offre le service `getInstance` qui construit une instance de la classe `Environment` la première fois qu'il est appelé puis renvoie cette même instance lors des invocations suivantes.

Singleton (Singleton) – En C++

```
class EnvironmentSingleton;

class Environment {
private:
   Environment () {}
   friend class EnvironmentSingleton;
};

class EnvironmentSingleton {
private:
   static Environment* _instance;

public:
   Environment& getInstance() {
      if (EnvironmentSingleton::_instance == NULL) {
         EnvironmentSingleton::_instance = new Environment ();
      }
      return (*EnvironmentSingleton::_instance);
   }
};

Environment* EnvironmentSingleton::_instance = NULL;
```

Singleton (Singleton) – Critique

- Le patron Singleton permet de définir des variables globales dont l'initialisation est contrôlée.
- On peut raffiner l'implémentation précédente pour s'assurer que la classe se comporte bien dans un environnement concurrent à l'aide de verrous (*mutex*).
- L'utilisation de variables globales peut mettre en danger l'encapsulation et l'extensibilité du système. Il faut donc utiliser ce patron avec parcimonie.
- Pour une discussion intéressante sur ce sujet :

  http://www.codingwithoutcomments.com/2008/10/08/singleton-i-love-you-but-youre-bringing-me-down/

Réserve d'objets (Object pool) – Description

- Exemple de situation :

  Un serveur peut traiter un nombre N de clients simultanés. Chaque client se voit allouer des ressources dont l'allocation est longue (une connection sur une base de données externes, des périphériques systèmes, . . .).

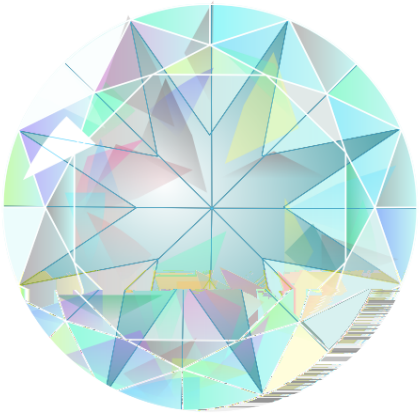
- Problème : Lorsque les ressources utilisées par chaque instance sont coûteuses et bornées, on veut contrôler le nombre d'instances d'une classe donnée.
- Solution : On peut utiliser un gestionnaire de ressources (une fabrique) fournissant un service `acquire`, donnant accès à une instance si c'est possible, et un service `release` permettant de réinjecter l'instance dans l'ensemble des instances accessibles.
- Consultez :

  http://www.kircher-schwanninger.de/michael/publications/Pooling.pdf

Réserve d'objets (Object pool) – Réflexion

Que faire si il ne reste plus d'objets en réserve ?

Patrons de conception de “Structure”



Adaptateur (Adapter) – Description

- Exemple de situation :

  J'utilise une bibliothèque de traitement d'images (dont je ne peux pas modifier le code source). Pour fonctionner, elle attend un objet fournissant une interface d'accès en lecture et en écriture à un tableau en deux dimensions contenant des triplets d'octets. J'aimerais l'interfacer avec une bibliothèque d'entrées/sorties fournissant une abstraction sur des tableaux unidimensionnels stockés de manière persistante dans une base de données ou dans un système de fichiers.

- Problème : Comment concilier les services proposés par la bibliothèque d'entrées/sorties et l'interface attendue par la bibliothèque de traitement d'images.
- Solution : Utiliser un objet qui implémente l'interface attendue en faisant appel aux services proposés par une instance de la bibliothèque d'entrées/sorties.

Adaptateur (Adapter) – en Java

```
public interface ColorImage2D {
    public int get_width (int x) ;
    public int get_height (int x) ;
    public short get_red (int x, int y) ;
    public short get_green (int x, int y) ;
    public short get_blue (int x, int y) ;
}

public static class ImageProcessing {
    public static void blur (ColorImage2D image) {}
}
```

Bibliothèque de traitement d'images.

```
public class IOArray {
    public short get (int x) { return 0 ; }
    public int size () { return 0 ; }
}
```

Bibliothèque d'entrées/sorties.

Adaptateur (Adapter) – en Java

```
public class IOArrayToColorImage2DAdapter implements ColorImage2D {
    private int width;
    private int height;
    private IOArray array;

    IOArrayToColorImage2DAdapter (IOArray array, int width, int height) {
      assert (array.size () == width * height);
      this.width = width;
      this.height = height;
      this.array = array;
    }

    public int get_width (int x) { return width; }
    public int get_height (int x) { return height; }
    public int get_point (int x, int y) { return array.get (y * width + x); }
    public short get_red (int x, int y) { return (short) (this.get_point (x, y) & 0x□); }
    public short get_green (int x, int y) { return (short) (this.get_point (x, y)     8 & 0x□); }
    public short get_blue (int x, int y) { return (short) (this.get_point (x, y)    16 & 0x□); }
}
```

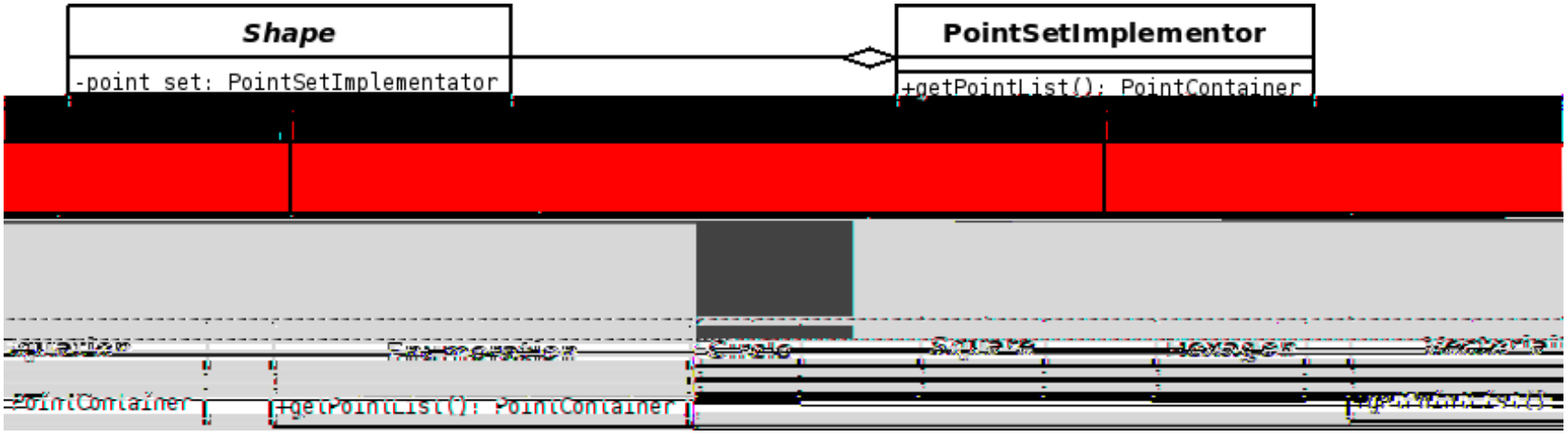
Pont (Bridge) – Description

- Exemple de situation :

  Un logiciel de dessin propose une hiérarchie de formes (carré, cercle, losange, . . .). Ces formes sont implémentées comme des ensembles de points qui peuvent être décrits de multiples façons (équation vectorielle, énumération, image bitmap, . . .).

- Problème : Pour éviter la multiplication des classes et modulariser le système, il est nécessaire de décorréler la hiérarchie des abstractions (les formes) de leurs implémentations (les ensembles de points) car elles peuvent varier de manière indépendante.
- Solution : Les implémentations sont organisées dans une hiérarchie indépendante. Une instance de la classe abstraite du sommet de cette hiérarchie (une classe nommée `Implementor`) est aggrégée à toute forme.

Pont (Bridge) – Diagramme UML



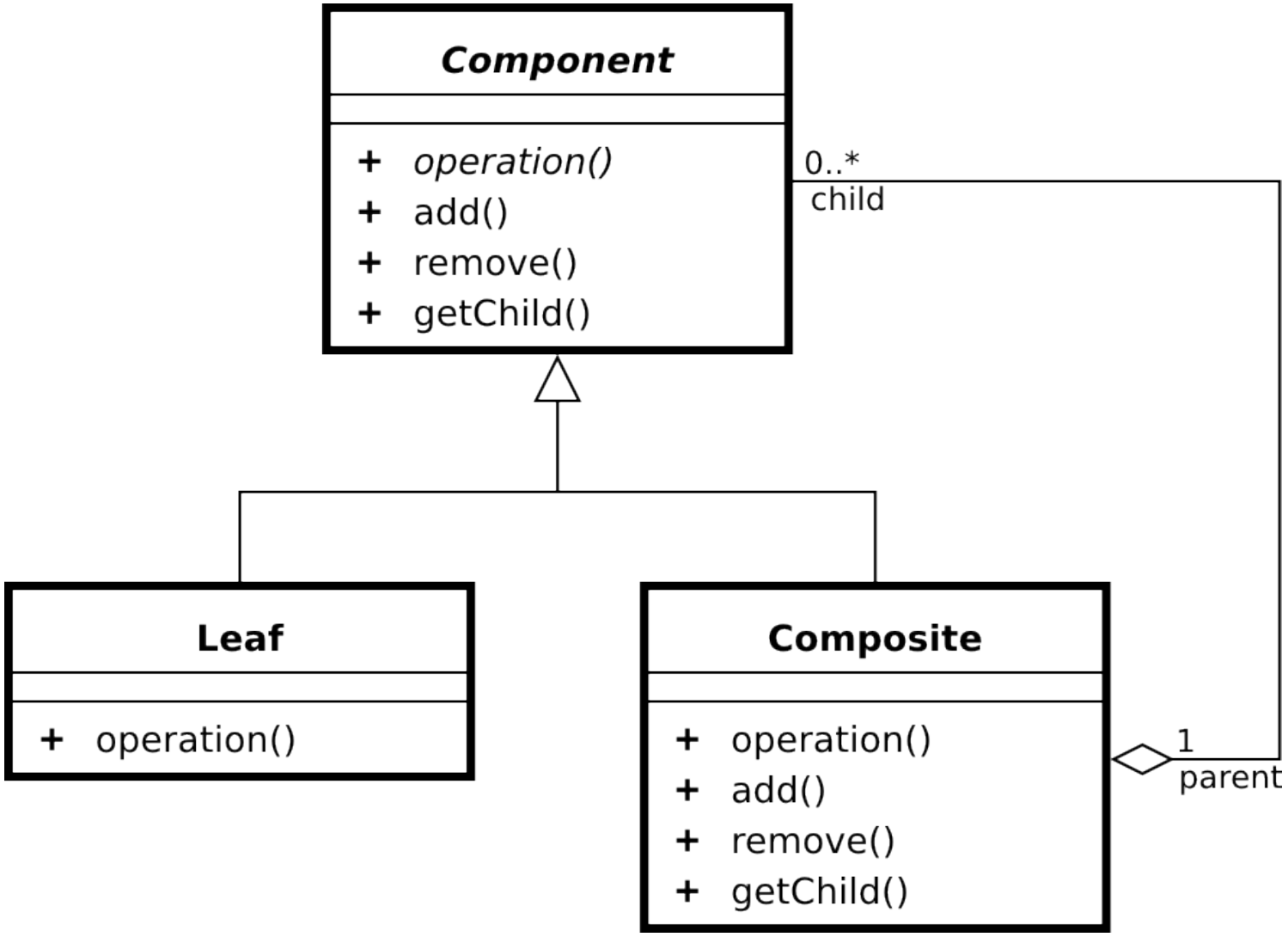
Objet composite (Composite) – Description

- Exemple de situation :

  Dans un logiciel de dessin, il est possible de créer des groupes de formes. Un groupe de formes est alors considéré comme une forme .

- Problème : Comment voir une composition de formes comme une forme ?
- Solution : Fournir un objet *Shape* qui implémente l'interface d'une forme ainsi que les services *add* et *remove*. Un objet *ShapeComposite*, héritant de *Shape*Shape... .

Objet composite (Composite) – Diagramme UML



(source : http://en.wikipedia.org/wiki/Image:Composite_UML_class_diagram.svg)

Objet composite (Composite) – Critique

- L'implémentation des méthodes *add* et *remove* n'a pas vraiment de sens pour les formes atomiques (cercle, carré, ...). On perd ainsi la sûreté (une exception sera lancée si le message *add* est envoyé à un cercle).
- On pourrait n'imposer ces méthodes que dans la classe *ShapeComposite*.
- On perd alors la transparence car on doit alors avoir un traitement différent pour le cas atomique et pour le cas composé.

⇒ Le GoF estime qu'il s'agit d'un compromis à décider au cas par cas ... La sûreté du logiciel doit, à mon avis, prévaloir.

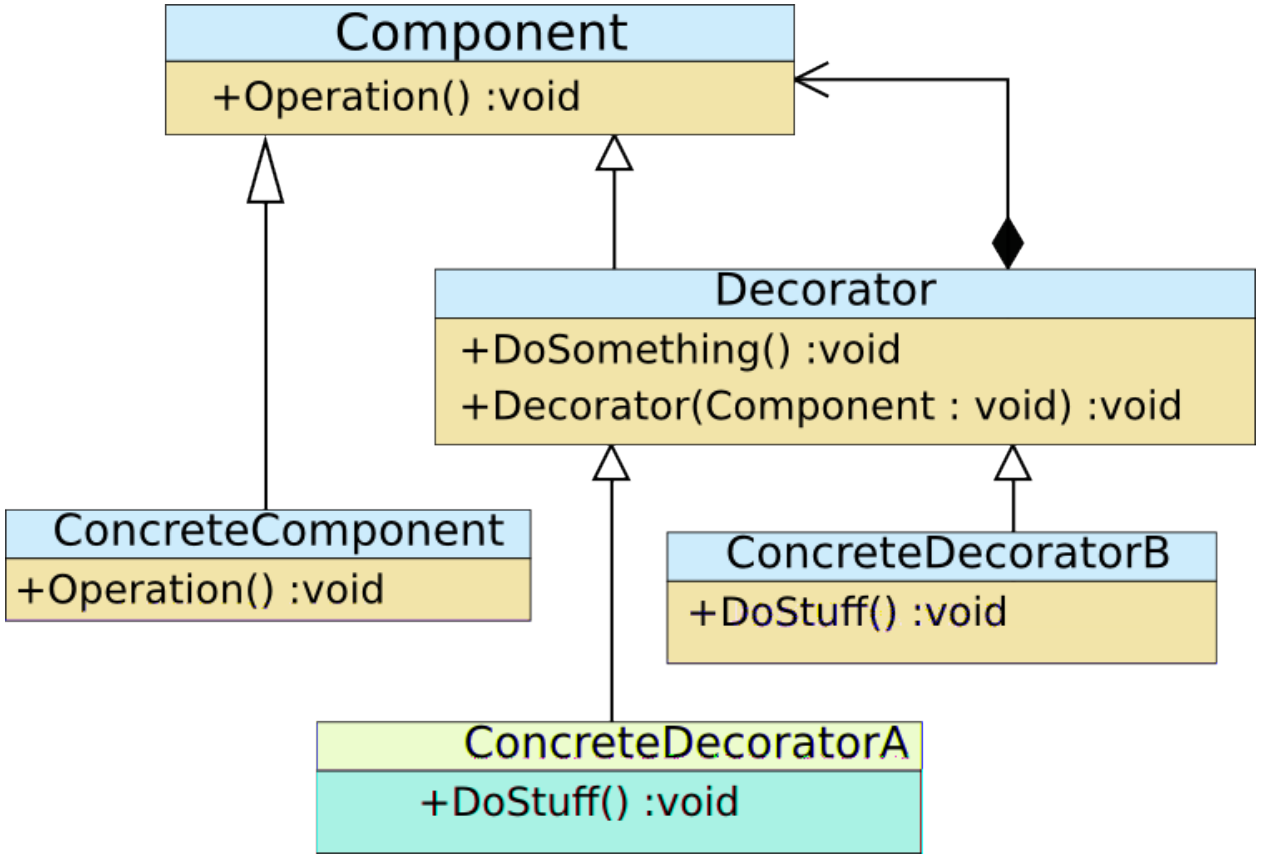
Objet composite *sûr* (Safe composite) – en O'Caml

```
class virtual shape = object
  method virtual draw : unit → unit
  method virtual as_composite : composite option
end
and composite = object (self)
  inherit shape
  val mutable children : shape list = []
  method as_composite = Some (self :> composite)
```

Décorateur (Decorator) – Description

- Exemple de situation :

Décorateur (Decorator) – Diagramme UML



(source : http://en.wikipedia.org/wiki/Image:Decorator_Pattern_ZP.svg)

Façade (Facade)

- Exemple de situation :

  Pour traiter les données d'un programme de gestion de bibliothèque multimédia, je désire utiliser l'API de Postgresql*, un gestionnaire de base de données très élaboré. Mon utilisation de la base de données est essentiellement un dictionnaire associant des noms de fichier et des descriptions à des noms symboliques.*

- Problème : Comment fournir une interface simple en utilisant un système complexe ?
- Solution : Le patron facade implémente seulement les méthodes utiles à mon dictionnaire en utilisant l'API de Postgresql. Il s'agit donc d'implémenter une couche d'abstraction.

Poids-mouche ou poids-plume (Flyweight) – Description

- Exemple de situation :

  Un compilateur traduit un code source en code machine. Une des premières phases consiste à savoir à quoi les identifiants utilisés dans le source font référence. Ce travail de résolution peut-être coûteux en temps et en espace (pour stocker la description de l'objet auquel on fait référence). Dans un même espace de noms, on voudrait ne pas avoir à refaire cette résolution plusieurs fois et partager les descriptions.

- Problème : Comment partager les calculs et le résultat de ces calculs de façon transparente ?
- Solution : Les identificateurs sont créés par une fabrique. Cette dernière stocke une table associant un descripteur à un identifiant. Si l'identifiant n'est pas dans la table, le calcul concret est effectué, sinon on renvoie l'élément déjà calculé.

Poids-mouche ou poids-plume (Flyweight) – En O'Caml

- Dans les langages fonctionnels, on peut implémenter ce procédé de façon générale pour toute fonction.

```
let memoize f =
  let table = Hashtbl.create 21 in
    fun x →
      try Hashtbl.find x
      with Not_found →
        let y = f x in
          Hashtbl.add x y ;
          y
```

- `memoize f` calcule une fonction qui exécute `f` au plus une fois pour une certaine valeur d'entrée.
- Il est possible de raffiner cette implémentation pour pouvoir « oublier » certains calculs.

Proxy (Proxy)

- Exemple de situation :

  Un système de navigation par GPS charge des images satellites de la planète par le réseau, c'est une opération très coûteuse. On veut pourtant voir l'ensemble de toutes ces images comme une unique image en profitant du fait que le téléchargement d'une section n'est nécessaire qu'au moment où cette section est effectivement affichée.

- Problème : Comment implémenter l'interface d'une image sans avoir toutes les données de cette image chargées en mémoire.
- Solution : Une classe `WorldImageProxy` implémente l'interface d'une image en chargeant ses attributs de manière paresseuse lors d'une réception de message nécessitant des données concrètes.

Comportement

Chaîne de responsabilité (Chain of responsibility) – Description

- Exemple de situation :

  On veut intégrer un mécanisme de greffons (plugins) a un traitement de texte pour pouvoir rajouter dynamiquement la gestion de certains types de composants.

- Problème : Dans un langage à typage statique basé sur des classes, on ne peut pas rajouter de sous-classes dynamiquement à une hiérarchie d'objets. Dans ce cas, comment implémenter un système de greffons ?
- Solution : On se donne un identifiant unique pour les types de composants à traiter. Cet identifiant va servir de message de première classe. On définit une interface pour les greffons contenant une méthode d'interprétation de ces messages. La liste des greffons est sotckée par l'application. On itère sur cette liste : lorsqu'un greffon ne sait pas traiter un message, il le transfère au greffon suivant.

Commande (Command) – Description

- Exemple de situation :

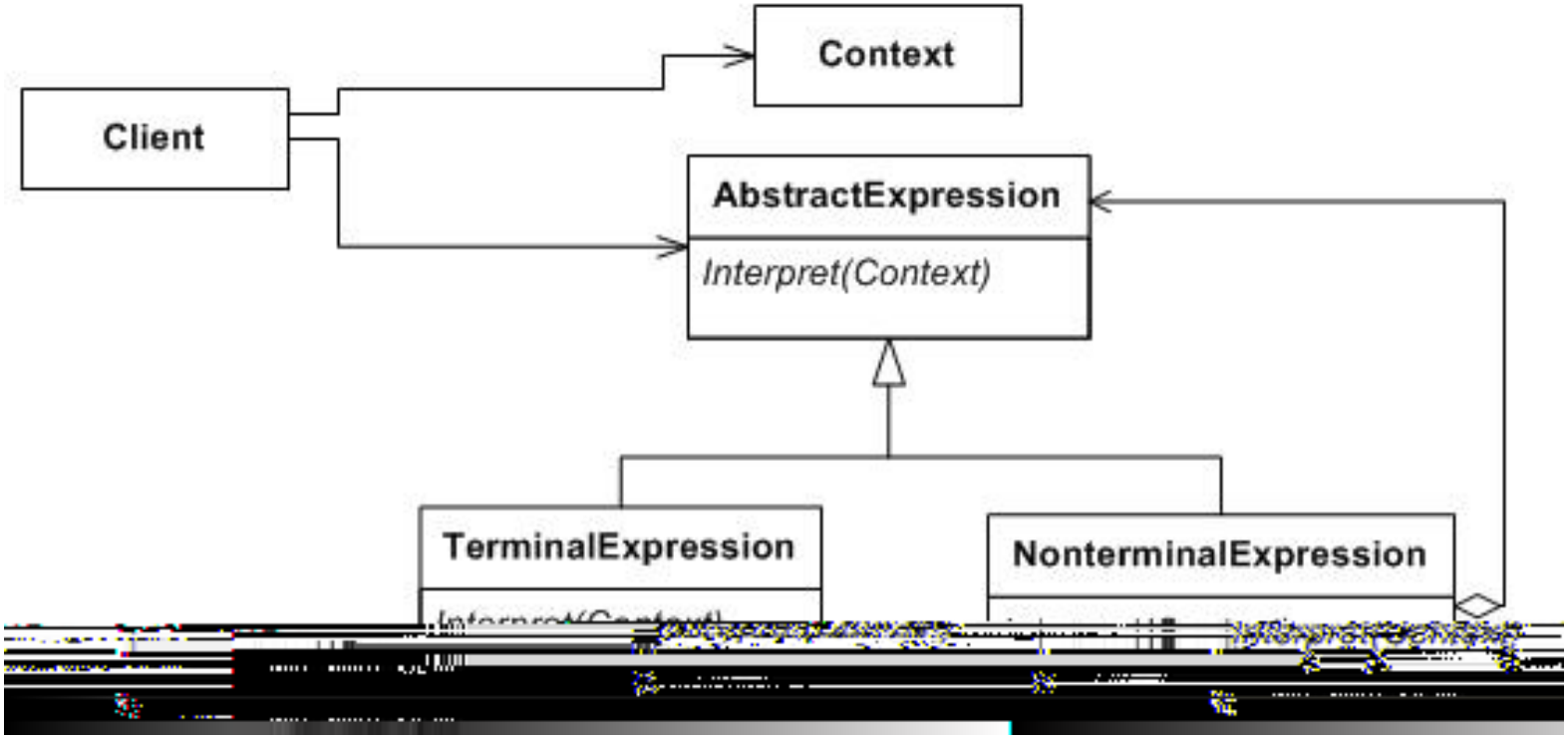
  Un logiciel de traitement d'image propose un ensemble de filtres à l'utilisateur. On veut donner la possibilité à l'utilisateur de construire ses propres filtres comme combinaison de filtres existants. On veut aussi lui donner la possibilité d'associer son filtre à un bouton de son application.

- Problème : Comment utiliser des opérations comme des données ?
- Solution : Il suffit d'implémenter un objet contenant une méthode `execute`... On peut alors associer ces fonctions à d'autres objets.

Interpréteur (Interpreter) – Description

- Exemple de situation : Composer les filtres n'est pas suffisant, un utilisateur avancé doit pouvoir programmer ses propres filtres (visuellement ou textuellement).
- Problème : Comment représenter la syntaxe d'un langage à l'aide d'objet ?
- Solution : Une classe abstraite *Expression* dont dérive les diverses constructions du langage.

Interpréteur (Interpreter) – Description



Itérateur (Iterator)

- Exemple de situation :

  Revenons à notre bibliothèque de dictionnaires. Je veux maintenant proposer un service permettant à un client d'inspecter l'ensemble des couples (clé, valeur) de mon dictionnaire sans divulguer la façon dont celui-ci est implémenté.

- Problème : Comment fournir un objet préservant l'encapsulation d'une structure de donnée tout en permettant de la parcourir ?
- Solution : Un itérateur fournit les méthodes :
  - *hasNext* indiquant si il existe un élément qui le suit dans la structure de donnée ;
  - *next* renvoyant un itérateur vers l'élément suivant ;
  - *getElement* renvoyant l'élément pointé par l'itérateur.

  Une structure de donnée implémente une méthode *iterator* renvoyant une itérateur sur le premier de ses éléments.

Médiateur (Mediator) – Description

- Exemple de situation :

  Une bibliothèque servant à construire des interfaces graphiques proposent en général des centaines de widgets différents. Ces widgets communiquent entre eux de façon complexe : asynchrone/synchrone, par broadcast, par groupes . . .

- Problème : Comment découpler l'implémentation d'un *widget* de la façon dont il communique avec les autres ?
- Solution : Implémenter un mécanisme de communication où les messages sont des objets de première classe et où les *widgets* ne sont pas référencés explicitement mais par des identifiants symboliques.

Médiateur (Mediator) – Critique

- La souplesse des messages implémentés par des données est claire : on peut reconfigurer, tracer, contrôler, les interactions entre les objets.
- Cependant, remplacer le mécanisme des méthodes par un tel système n'est pas forcément une bonne idée car on perd alors les garanties fournies par le typage (statique).

Memento (Memento)

- Exemple de situation :

  Dans un contexte d'accès concurrent à une ressource, il est parfois utile d'être optimiste : on lance la requête d'accès de façon asynchrone, on continue la partie du calcul qui ne dépend pas de l'accès et on termine le calcul une fois que le résultat de la requête est arrivé. Mais que faire lorsque la requête a échoué ? On doit se replacer dans l'état initial et faire comme si on n'avait pas continuer ses calculs...

- Problème : Comment restaurer l'état d'un objet ?
- Solution : Une classe Memento sert à représenter l'état d'un objet (en Java, un objet de type Object fait très bien l'affaire). Une classe CareTaker sert à stocker ses états. Elle est aggrégée par l'objet qu'on veut être capable de restaurer. Avant une section critique, l'objet se stocke sous la forme d'un Memento dans son attribut de type CareTaker et se restore si la requête a échoué.

Observateur (Observer)

- Exemple de situation :

  Imaginons l'application d'une séquence de filtres dans un logiciel de retouche d'images. Pour tenir l'utilisateur au courant de la progression du calcul, plusieurs objets s'affichent dans son interface : une barre de progression, un pourcentage dans la barre de titre, une icône sur son bureau . . . Le nombre de ces indicateurs n'est a priori pas limité. Comment s'assurer qu'ils seront tous mis au courant de l'avancement du processus ?

- Problème : Comment modéliser l'observation de l'état d'un objet par un ensemble dynamiquement défini d'autres objets ?
- Solution : L'objet observé offre une méthode *attach* permettant à un observateur de s'enregistrer pour être informé. Pour cela, celui-ci doit proposer une méthode *notify*. A chaque fois qu'il est mis à jour, l'objet observé itère sur l'ensemble de ses observateurs et les notifie de sa modification.

État (State)

- Exemple de situation :

  Revenons maintenant sur le problème des figures géométriques atomiques et composites. J'ai décidé d'implémenter le comportement suivant : lorsque le message `add` est envoyé à une figure atomique, elle devient tout simplement composite !

- Problème : Comment donner l'illusion que le type d'un objet a changé en fonction de son état ?
- Solution : Une forme stocke un attribut qui peut être une forme atomique ou une forme composite. Les formes atomiques et composites ne dérivent plus de la classe *Shape* mais d'une nouvelle classe abstraite *RawShape*. Lorsque la méthode *add* est appelée et que l'attribut est une forme atomique, on le remplace par une nouvelle forme composite la contenant ainsi que la nouvelle forme à rajouter.

Stratégie (Strategy) – Description

- Exemple de situation :

  Une intelligence artificielle s'appuie sur une heuristique, c'est-à-dire une évaluation approximative de la valeur d'une situation pour le joueur. Il existe de nombreuses heuristiques possibles pour un jeu d'échec. Est-ce à dire qu'il faille créer une sous-classe par heuristique ?

- Problème : Comment paramétrer une classe par un comportement ?
- Solution : Un comportement est implémentée par une classe. Le comportement est donné en argument au constructeur de la classe à paramétrer. Remarquez encore une fois qu'en présence de fonction de première classe, ce patron est trivialement implémentable.

Patron de méthode (Template Method) – Description

- Exemple de situation : Je souhaite implémenter un objet effectuant le tri d'un tableau d'objet paramétré par la relation de comparaison entre les objets stockés.
- Problème : Comment paramétrer une méthode par une fonction ?
- Solution : Encore une fois, il s'agit de programmation d'ordre supérieur, on utilise un objet pour représenter la fonction pris en paramètre.

Visiteur (Visitor) – Description

- Exemple de situation : Je veux traduire en une multitude de format un document composé par des instances de ma hiérarchie de formes. Pour des raisons de modularité, il est hors de question d'écrire ce code dans chacune des classes !
- Problème : Comment étendre fonctionnellement une hiérarchie existante ?
- Solution : Un visiteur implémente une méthode par cas de la hiérarchie. Chaque sous-classe implémente une méthode *visit* qui appelle la méthode du visiteur correspond à son cas.