Programmation avancée

Programmation orientée objet

Armaël Guéneau

(transparents en partie empruntés à Jean-Christophe Filliâtre)

dans le cours d'aujourd'hui, on se focalise sur les **langages objets**

bref historique :

- Simula I et Simula 67 (années 60)
- Smalltalk (1980)
- C++ (1983)
- Objective-C (1984), Object Pascal (1986), Python (1991), Ruby (1995)...
- Java (1995)
- OCaml (1996)
- Scala (2004), F# (2005), Kotlin (2011)...

on utilise ici le langage Scala à des fins d'illustration

et on introduira les objets de OCaml dans un second temps

motivations : **programmation modulaire**, **réutilisation de code**

questions investiguées assez tôt avec les langages objet

au débuts de Java: on essaye de tout faire avec des objets
influence importante sur le design de langages de programmation

de nos jours : influences venant à la fois des langages objets et de la programmation fonctionnelle (fonctions de première classe, polymorphisme, types algébriques)

concepts objets

le premier concept objet est celui de **classe** ;
la déclaration d'une classe introduit un nouveau type

en toute première approximation, une classe peut être vue comme un enregistrement

```
class Polar(
  var rho: Double,
  var theta: Double
)
```

ici `rho` et `theta` sont les deux **champs** de la classe `Polar` (on parle aussi parfois d'**attributs**)

ils sont ici **mutables**

on crée une **instance** particulière d'une classe, appelée un **objet**, avec la construction `new`

```
val p: Polar = new Polar(2, 0.5)
```

déclare une nouvelle variable locale `p`, de type `Polar`, dont la valeur est une nouvelle instance de la classe `Polar`

on peut accéder aux champs de `p`, et modifier les champs **mutables**, avec la notation usuelle

```
println(p.rho, p.theta) // (2, 0.5)
p.rho = 1
p.theta = 3.14159265
val x = p.rho * Math.cos(p.theta)
p.theta = p.theta / 2
```

on peut ajouter du code exécuté à la **construction** de l'objet

```
class Polar(var rho: Double, var theta: Double) {
  if (theta < 0) throw new Error("Polar: negative length")
}
```

ce qui permet de faire des vérifications à l'initialisation:

```
  val p = new Polar(2, 3.14159265) // OK
  val q = new Polar(-1, 1) // Erreur
```

la classe peut aussi définir des champs supplémentaires, sans qu'ils soient arguments du constructeur

```
class Polar(var rho: Double, var theta: Double) {
  if (theta < 0) throw new Error("Polar: negative length")
  var color : Int = 0
}


val p = new Polar(1, 3)
println(p.color) // 0
p.color = 2
```

Motivation n°1: **programmation modulaire**

Comment maintenir l'**invariant** suivant pour tous les objets de la classe `Polar` ?

$$0 \leq \texttt{rho} \quad \wedge \quad 0 \leq \texttt{theta} < 2\pi$$

Motivation n°1: **programmation modulaire**

Comment maintenir l'**invariant** suivant pour tous les objets de la classe `Polar` ?

$$0 \leq \texttt{rho} \quad \wedge \quad 0 \leq \texttt{theta} < 2\pi$$

Solution 1 : on peut rendre ces champs **immutables** en les déclarant avec `val`:

```
class Polar(val rho : Double, val theta : Double) {
  // vérifie l'invariant
}

val p = new Polar(1, 3.14)
println(p.rho, p.theta) // (1, 3.14)
p.rho = -1 // Erreur
```

(Demi) Solution 2 :

(Demi) Solution 2 : faire de `rho` et `theta` seulement des **paramètres** de la classe, pas des champs

```
class Polar(rho : Double, theta : Double) {
  // vérifie l'invariant
  // ... utilise rho et theta ? ...
}
```

→ alors, un objet de type `Polar` **ne stocke pas** de valeur `rho` et `theta`

```
val p = new Polar(1, 2)
println(p.rho, p.theta) // Erreur
```

Solution 3 : on peut en fait **cacher** un champ en le rendant **privé**
le champ existe mais est inaccessible en dehors de la classe

on utilise pour ça le mot-clef `private`

```
class Polar(private var rho : Double, private var theta : Double) {
  /* vérifie l'invariant */
  // marche aussi pour les champs supplémentaires :
  private var color : Int = 0
}
```

les champs privés peuvent être utilisés par les **méthodes** de la classe
il s'agit de fonctions fournies par la classe `Polar` et applicables à tout objet de cette classe

```
class Polar(private var rho : Double, private var theta : Double) {
  def norm(): Double = { rho }
  def scale(n: Double) = { rho = n * rho }
  def x(): Double = { rho * Math.cos(theta) }
}
```

pour un objet `p` de type `Polar`, on appelle la méthode `norm` ainsi :

```
p.norm()
```

que l'on peut voir naïvement comme l'appel `norm(p)` d'une fonction

```
def norm(Polar x): Double = { x.rho }
```

les objets remplissent donc un premier rôle d'**encapsulation**

l'équivalent en OCaml serait obtenu grâce aux types abstraits
via les signatures de modules

l'équivalent en Rust serait également les types abstraits
via les mécanismes de visibilité publique/privée

Motivation n°2: **réutilisation de code**

le second concept objet est celui d'**héritage** : une classe `B` peut être définie comme héritant d'une classe `A`

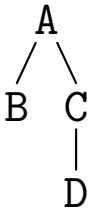
```
class B extends A { ... }
```

les objets de la classe `B` héritent alors de tous les champs et méthodes de la classe `A`, auxquels ils peuvent ajouter de nouveaux champs et de nouvelles méthodes

la notion d'héritage s'accompagne d'une notion de **sous-typage** : toute valeur de type `B` peut être vue comme une valeur de type `A`

en Scala et Java, chaque classe hérite d'au plus une classe ; on appelle cela l'**héritage simple**, par opposition à l'héritage multiple

```
class A { ... }
class B extends A { ... }
class C extends A { ... }
class D extends C { ... }
```



(les "traits" en Scala / "interfaces" en Java permettent une forme d'héritage multiple...)

considérons une classe `Graphical` pour des objets graphiques (cercles, rectangles, etc.)

```
class Graphical {
  var x, y: Int = 0          /* centre */
  var width, height: Int = 0

  def move(int dx, int dy) = { x += dx; y += dy }
  def draw() = { /* ne fait rien */ }
}
```

pour représenter un rectangle, on hérite de la classe `Graphical` avec `extends`

on paramètre la classe par les deux coins du rectangle

```
class Rectangle(x1: Int, y1: Int, x2: Int, y2: Int)
  extends Graphical
{
```

on hérite donc des champs `x`, `y`, `width` et `height` et des méthodes `move` et `draw`

```
  x = (x1 + x2) / 2
  y = (y1 + y2) / 2
  width = Math.abs(x1 - x2)
  height = Math.abs(y1 - y2)
}
```

on peut utiliser directement toute méthode héritée de `Graphical`

```
val p: Rectangle = new Rectangle(0, 0, 100, 50)
p.move(10, 5)
```

pour le dessin, on va **redéfinir** la méthode `draw` dans la classe Rectangle (*overriding*)

```
class Rectangle(...) extends Graphical {
  ...
  override def draw() = { /* dessine le rectangle */ }
}
```

et le rectangle sera alors effectivement dessiné quand on appelle

```
p.draw()
```

on procède de même pour les cercles ; ici on ajoute un champ `radius` pour le rayon, afin de le conserver

```
class Circle(cx: Int, cy: Int, r: Int) extends Graphical {
  var radius : Int = r
  x = cx
  y = cy
  width = 2 * radius; height = 2 * radius
  override def draw() = { /* dessine le cercle */ }
}
```

le code exécuté par un appel de méthode est résolu **dynamiquement** ; il dépend de la classe utilisée pour **créer** l'objet

exemple :

```
val g1 : Graphical = new Rectangle(0, 0, 100, 50)
val g2 : Graphical = new Circle(0, 0, 20)
g1.draw() // dessine le rectangle
g2.draw() // dessine le cercle
```

g1 et g2 ont le type `Graphical`, mais c'est respectivement la méthode `draw` des classes `Rectangle` et `Circle` qui sont exécutées

on peut donc écrire du code **générique** vis-à-vis du type d'objets graphiques

par exemple, définissions un troisième type d'objet graphique :
la réunion de plusieurs objets graphiques

un groupe contient une liste d'objets `Graphical` :

```
class Group extends Graphical {
  var group : List[Graphical] = List() // la liste vide

  def add(g: Graphical) = {
    group = g +: group // ajout en tête de liste
    // + mise à jour de x, y, width, height
  }
```

il reste à redéfinir les méthodes `draw` et `move`

```
  override def draw() = {
    for (g <- group) g.draw()
  }

  override def move(dx: Int, dy: Int) = {
    super.move(dx, dy) // appel au move() de Graphical
    for (g <- group) g.move(dx, dy)
  }
}
```

il est clair sur cet exemple que le compilateur ne peut pas connaître statiquement l'implémentation de `g.draw()`

remarque : il n'y a pas lieu de créer d'instance de la classe `Graphical` ; c'est ce que l'on appelle une **classe abstraite**

on peut alors demander que des méthodes et attributs soient définis dans les sous-classes sans les définir nous même : ici `draw, x, y, width, height`

```
abstract class Graphical {
  var x, y, width, height: Int
  def move(int dx, int dy) = { x += dx; y += dy }
  def draw(): Unit
}
```

il est alors **obligatoire** de les définir dans toute sous-classe (non abstraite) de `Graphical`

```
val g1 : Rectangle = new Rectangle(0, 0, 100, 50)
g1.move(10, 5)
g1.draw()
val g2 : Graphical = new Circle(10,10,2)
val g3 = new Group(); g3.add(g1); g3.add(g2)
g3.draw()
g3.move(-5,-7)
g3.draw()
```

```
$ scala graphical.scala
```

la notion de classe remplit plusieurs fonctions

- **encapsulation**, à travers des règles de visibilité
- organisation de l'**espace de noms**, à travers la possibilité d'utiliser le même nom dans des classes différentes ou pour des profils différents
- **factorisation de code**, à travers l'héritage et la redéfinition

ce sont des objectifs essentiels du **génie logiciel**, atteints par des moyens différents dans d'autres langages (exemple : polymorphisme, ordre supérieur, ou encore système de modules, dans le cas d'OCaml)

type statique, type dynamique

en Scala (et Java), la notion de type est déterminée par la hiérarchie de classes
le type d'un objet à un point du programme correspond à la classe à laquelle il est vu

```
val p : Rectangle = new Rectangle(0, 0, 100, 50)
```

on parle du **type statique** d'un objet, calculé statiquement par le compilateur

la notion de sous-classe donne lieu à une notion de **sous-typage**
on peut toujours typer un objet au type d'une classe parent

```
val p : Graphical = new Rectangle(0, 0, 100, 50)
```

le type statique de p est ici Graphical.

autre exemple:

```
def movedraw(g: Graphical) = { ... }

val r : Rectangle = new Rectangle(0, 0, 100, 50)
val c : Circle = new Circle(0, 0, 20)
// possible via sous-typage vers Graphical :
movedraw(r)
movedraw(c)
```

la construction `new C(...)` construit un objet de la classe `C` : à l'exécution, son comportement est déterminé par `C` : on parle du **type dynamique** de l'objet

exemple

```
val g : Graphical = new Rectangle(0, 0, 100, 50)
g.draw() // dessine le rectangle
```

pour le compilateur, g a le type statique `Graphical`, mais c'est bien la méthode `draw` de la classe `Rectangle` qui est exécutée

un **appel de méthode** est résolu d'après le **type dynamique**

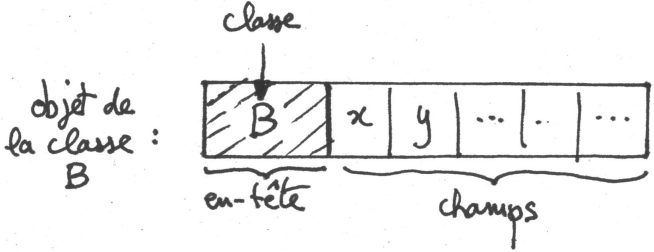
pour résumer :

le **type statique** est vérifié à la compilation puis **effacé**

il permet de vérifier la **sûreté** du programme : “l’objet a-t-il bien une méthode `m()` ?”

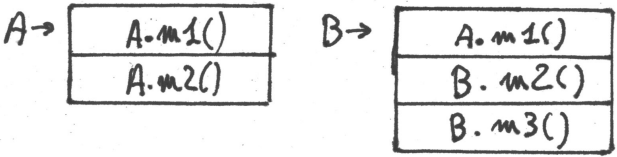
le **type dynamique** est **stocké** à l’exécution dans chaque objet

il permet de déterminer **quelle méthode appeler** à l’exécution :
“quelle classe implémente `m()` ?”




```
B extends A {
  override def m2() {...}
  def m3() {...}
}
```

tables de méthodes des classes :



type dynamique et redéfinition de méthode (1)

```
class A {
  def foo() = { println("A") }
}

class B extends A {
  override def foo() = { println("B") }
}

val a: A = new B
a.foo()
```

type dynamique et redéfinition de méthode (1)

```
class A {
  def foo() = { println("A") }
}

class B extends A {
  override def foo() = { println("B") }
}

val a: A = new B
a.foo()
// affiche: B
```

type dynamique et redéfinition de méthode (2)

```
class A {
  def foo() = { println("A") }
  def bar() = { foo() }
}

class B extends A {
  def foo() = { println("B") }
}

val b : B = new B
b.bar()
```

type dynamique et redéfinition de méthode (2)

```
class A {
  def foo() = { println("A") }
  def bar() = { foo() }
}

class B extends A {
  def foo() = { println("B") }
}

val b : B = new B
b.bar()
// affiche: B
```

type dynamique et redéfinition de méthode (2)

```
class A {
  def foo() = { println("A") }
  def bar() = { foo() }
}

class B extends A {
  def foo() = { println("B") }
}

val b : B = new B
b.bar()
// affiche: B
val a: A = new B
a.bar()
```

type dynamique et redéfinition de méthode (2)

```
class A {
  def foo() = { println("A") }
  def bar() = { foo() }
}

class B extends A {
  def foo() = { println("B") }
}

val b : B = new B
b.bar()
// affiche: B
val a: A = new B
a.bar()
// affiche: B
```

type dynamique et redéfinition de méthode (3)

il s'agit du comportement de **liaison tardive** : l'appel de méthode est résolu **dynamiquement** ; et il est possible pour une sous-classe de redéfinir des méthodes

c'est un mécanisme important de la programmation objet, puissant mais subtil (redéfinir une méthode doit fait de façon compatible avec son utilisation dans la classe parent...)

```
class A { def foo() = { println("A") } }

class B extends A {
  override def foo() = { println("B") }
}

def callfoo(x: A) = { println("callfoo: A"); x.foo() }

def callfoo(x: B) = { println("callfoo: B"); x.foo() }

val x : A = new B
callfoo(x) // ???
```

attention : **surcharge statique $\neq$ redéfinition de méthodes**

le mécanisme de **surcharge** d'une fonction (cf cours précédent) s'appuie sur le type statique

il est distict du mécanisme de **redéfinition** (*overriding*) de méthodes

programmation fonctionnelle vs programmation objet

un point de comparaison intéressant : l'**extensibilité**

pour une bibliothèque fournissant certaines fonctionnalités, comment étendre son comportement sans modifier le code ?

programmation avec types algébriques vs programmation avec objets :
permettent l'extensibilité dans des **directions orthogonales**

qu'est-ce qui distingue

```
type graphical = Circle of ... | Rectangle of ...
```

de

```
class Graphical {...} class Circle extends Graphical {...}
```

en OCaml, le code de `move`, `draw` est à un seul endroit et traite tous les cas

```
let move = function Circle _ -> ... | Rectangle _ -> ...
let draw = function Circle _ -> ... | Rectangle _ -> ...
```

en Java, il est éclaté dans l'ensemble des classes

si l'on veut ajouter une nouvelle forme géométrique :

avec un type algébrique, il faut modifier chaque fonction existante

```
type graphical = ... | Triangle of ...

let move = ... | Triangle _ -> ...
let draw = ... | Triangle _ -> ...
```

avec des classes, il suffit d'ajouter une nouvelle sous-classe

```
class Triangle extends Graphical {
   override def move(dx: Int, dy: Int) = { ... }
   override def draw() = { ... }
}
```

ajouter une sous-classe : pas besoin de changer le code existant !

mais si l'on veut ajouter une nouvelle opération :

avec un type algébrique, il suffit de définir une nouvelle fonction

```
let scale = function Circle _ -> ... | Rectangle _ -> ...
```

pas besoin de changer le code existant !

avec des classes, il faut ajouter une méthode à chaque sous-classe

```
class Rectangle extends Graphical {
  ... override def scale() = { ... }
}
class Circle extends Graphical {
  ... override def scale() = { ... }
}
```

	extension horizontale = ajout d'un cas	extension verticale = ajout d'une fonction
Java/ (Scala)	**facile** (un seul fichier)	pénible (plusieurs fichiers)
OCaml	pénible (plusieurs fichiers)	**facile** (un seul fichier)

en style fonctionnel :

```
type expr =
  | Cst of int
  | Add of expr * expr
type target = ...

val compile : expr -> target
```

ajouter une passe d'optimisation revient à ajouter une fonction :

```
val optimize : target -> target
```

mais pour étendre le langage d'entrée, il faut modifier `compile`

avec des objets :

```
class Target { ... }
abstract class Expr { def compile(): Unit }

class Cst(val n: Int) extends Expr {
  override def compile() = { ... }
}
class Add(val e1: Expr, val e2: Expr) extends Expr {
  override def compile() = { ... }
}
```

on peut alors étendre le langage d'entrée en ajoutant une sous-classe de `Expr`

objets en OCaml

Référence : manuel de référence OCaml, Chapitre 3

les objets en OCaml s'ajoutent au fragment fonctionnel du langage

conséquence :

en Scala/Java :

- la notion de type est issue du système de classes
- une valeur du langage est un objet, appartenant à une classe donnée

en OCaml :

- on a une notion préexistante de type
- les objets ne sont qu'une des valeurs possibles du langage

voyons comment les notions d'objet ont été adaptées en OCaml...

- on peut construire des valeurs objets sans utiliser de classe
- leur type est un **type objet** : `< foo : t1; bar : t2; ..>`
- l'appel de méthode se fait avec `#`

```
# let counter =
    object
      val mutable x = 0
      method get = x
      method incr = x <- x + 1
    end;;
val counter : < get : int; incr : unit > = <obj>
# counter#incr;; (* appel de méthode *)
- : unit = ()
# counter#get;;
- : int = 1
```

comme pour le reste du langage, on bénéficie de **l'inférence** de types

quizz: quel type est inféré pour `f` ?

```
# let f o = o#get;;
```

comme pour le reste du langage, on bénéficie de **l'inférence** de types

quizz: quel type est inféré pour `f` ?

```
# let f o = o#get;;
val f : < get : 'a; .. > -> 'a = <fun>
```

`< get : 'a; .. >` est le type des objets contenant **au moins une méthode** `get` renvoyant un `'a`.

comme pour le reste du langage, on bénéficie de **l'inférence** de types

quizz: quel type est inféré pour `f` ?

```
# let f o = o#get;;
val f : < get : 'a; .. > -> 'a = <fun>
```

`< get : 'a; .. >` est le type des objets contenant **au moins une méthode** `get` renvoyant un `'a`.

```
# f counter;; (* counter : < get : int; incr : unit > *)
- : int = 1
```

- pour produire des objets selon un schéma donné, on pourrait simplement utiliser une fonction renvoyant des objets
- l'utilité première des classes est le mécanisme d'héritage associé

équivalent de notre classe Scala `Graphical` :

```
class graphical = object
  val mutable x = 0  (* attribut mutable *)
  val mutable y = 0  (* idem ... *)
  val mutable width = 0
  val mutable height = 0
  method move dx dy = y <- x+dx; y <- y+dy
  method draw = ()
end
```

```
class rectangle x1 y1 x2 y2 = object
  inherit graphical  (* héritage *)
  initializer  (* code d'initialisation de l'objet *)
    x <- (x1+x2)/2; y <- (y1+y2)/2;
    width <- abs (x1 - x2); height <- (y1 - y2)

  (* redéfinition de méthode *)
  method draw = ...
end
```

- `inherit` indique que `rectangle` est une sous-classe de `graphical`
- `initializer` correspond au code d'initialisation de la classe qu'on écrit en Scala

utilisation de la classe `rectangle` :

```
# let r = new rectangle 0 0 50 100;;
val r : rectangle = <obj>
# r#move 10 10;;
- : unit = ()
# r#draw;;
...
- : unit = ()
```

les classes/méthodes abstraites de Scala sont ici appellées **virtuelles**

```
class virtual graphical = object
  val mutable x = 0
  val mutable width = 0
  val mutable height = 0
  method move dx dy = x <- x+dx; y <- y+dy
  method virtual draw : unit  (* on spécifie le type *)
end
```

dans un objet ou une classe, il faut explicitement donner un nom à l'objet courant pour appeler d'autres méthodes du même objet

```
let counter =
  object(self)
    val mutable x = 0
    method get = x
    method incr = x <- x+1
    (* appel des méthodes incr, get de l'objet courant *)
    method getnext = self#incr; self#get
  end
```

syntaxe : `object(self) ... end`; aussi possible en définissant une classe

attention !
en Scala : sous-typage **nominal**, déterminé par la hiérarchie d'héritage
en OCaml : "sous-typage" **structurel**, décorrélé de l'héritage

```
# class a = object method foo = "foo A" end;;
# class b = object
    method foo = "foo B"
    method bar = "bar B"
  end;;
# let x:b = new b;;
# let y:a = (x :> a);; (* coercion explicite *)
```

- sous-typage uniquement déterminé par le type des méthodes
- requiert en général une coercion explicite `x :> ty`

- les types d'objets peuvent mentionner des variables de types...
- et les définitions de classes peuvent également être polymorphes

```
# class ['a] pcounter (zero: 'a) (succ: 'a -> 'a) = object
    val mutable x = zero
    method get = x
    method incr = x <- succ x
  end;;

# let c = new pcounter 0 succ in c#incr; c#get;;
- : int = 1
# let c' = new pcounter 0. (fun x -> x +. 1.) in
  c'#incr; c'#get;;
- : float = 1.
```

exemple où la POO est au service de la programmation fonctionnelle : le pattern visiteur

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

Objectif 1 : compter la somme des constantes d'une expression

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

Objectif 1 : compter la somme des constantes d'une expression

```
let rec sum_constants = function
  | Cst n -> n
  | Add (e1, e2) -> sum_constants e1 + sum_constants e2
  | Sub (e1, e2) -> sum_constants e1 + sum_constants e2
```

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

Objectif 2 : compter le nombre d'opérations d'addition

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

Objectif 2 : compter le nombre d'opérations d'addition

```
let rec count_add = function
  | Cst n -> 0
  | Add (e1, e2) -> count_add e1 + count_add e2 + 1
  | Sub (e1, e2) -> count_add e1 + count_add e2
```

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

Objectif 3 : compter le nombre d'opérations de soustraction

```
type expr =
  | Cst of int
  | Add of expr * expr
  | Sub of expr * expr
```

Objectif 3 : compter le nombre d'opérations de soustraction

```
let rec count_sub = function
  | Cst n -> 0
  | Add (e1, e2) -> count_sub e1 + count_sub e2
  | Sub (e1, e2) -> count_sub e1 + count_sub e2 + 1
```

problème : on réécrit systématiquement le code de traversée récursive du type `expr`

problème : on réécrit systématiquement le code de traversée récursive du type `expr`

Solution : une fonction `iter` d'ordre supérieur ?

```
let rec iter
  (cst : int -> unit)
  (add : expr -> expr -> unit)
  (sub : expr -> expr -> unit)
  (e : expr) = ...
```

Exercice : réimplémenter `sum_constants`, `count_add` et `count_sub`

il faut quand même gérer tous les cas, même ceux où on ne fait rien...

on définit une classe **visiteur** qui parcourt `expr` en ne faisant rien (!)

```
class iter = object (self)
  method visit_Cst n = ()
  method visit_Add e1 e2 =
    self#visit_expr e1; self#visit_expr e2
  method visit_Sub e1 e2 =
    self#visit_expr e1; self#visit_expr e2
  method visit_expr e =
    match e with
    | Cst n -> self#visit_Cst n
    | Add (e1, e2) -> self#visit_Add e1 e2
    | Sub (e1, e2) -> self#visit_Sub e1 e2
end
```

on utilise ensuite le visiteur en héritant et en **redéfinissant** certaines méthodes

Comment implémenter `count_add` ?

on utilise ensuite le visiteur en héritant et en **redéfinissant** certaines méthodes

Comment implémenter `count_add` ?

```
let count (e : expr) : int =
  let v = object
    val mutable count = 0
    method count = count
    inherit iter as super
    method! visit_Add e1 e2 =
      count <- count + 1;
      super#visit_Add e1 e2
  end in
  v#visit_expr e;
  v#count
```

on bénificie de l'**extensibilité** du mécanisme d'héritage sur deux aspects :

- on implémente uniquement les cas des constructeurs pertinents
- le code ne change pas **même quand on étend** `expr` avec d'autres constructeurs

de plus, la classe visiteur peut être **générée automatiquement**
(cf. la bibliothèque `visitors`)

une des forces de la programmation objet : l'extensibilité horizontale

les concepts objets se retrouvent dans de (très) nombreux langages de programmation

cela dit, en OCaml, pour structurer des programmes et écrire du code générique, on est souvent mieux servi par d'autres mécanismes (polymorphisme, ordre supérieur, modules...)

en pratique, quelle utilisation des objets en OCaml ? exemples :

- `lablgtk`, `lambda-term` (interfaces graphiques)
- `visitors` (parcours extensibles de types de données)
- `js_of_ocaml`, `tyxml` (types objets pour types fantômes avancés)