Typage et Analyse Statique

Cours 4

Emmanuel Chailloux

Spécialité Science et Technologie du Logiciel
Master mention Informatique
Université Pierre et Marie Curie

année 2016-2017

Plan du cours 4

Programmation par objets:

- Types en Java, sous-typage
- Surcharge en Java
  - par l'exemple
  - algorithmes de résolution
- Classes de type en Haskell

Langages et Typage

- caractéristiques
  - dynamique : la vérification des types s'effectue à l'exécution (Scheme, Smalltalk, Python, ...)
  - statique : la vérification des types est effectuée à la compilation
  - typage fort : aucun test de type à l'exécution (Ada, OCaml, ...)
  - typage faible (non fort) : en Java ou C#, le transtypage (cast, upcast, downcast) introduit des tests de types à l'exécution
- caractéristiques du typage statique
  - meilleure sûreté d'exécution
  - tout en restant général (généricité)
  - permet de garantir l'usage de bibliothèques
  - meilleur code produit
  - peut refuser des programmes corrects

Types en Java

définition inductive de l'ensemble des types Java :

- Si τ est un type primitif alors τ est un type
- Si I est une interface, alors I est un type
- Si C est une classe, alors C est un type
- Si τ est un type, alors τ [ ] est un type

---

auxquels on ajoute les types **enum** les classes et interfaces paramétrées :

- si τ est un type énuméré alors τ est un type
- Si $C < T_1, ..., T_n >$ est une classe, avec $T_1, ..., T_n$ des paramètres de types, alors si $\tau_1, ..., \tau_n$ sont des types, $C < \tau_1, ..., \tau_n >$ est un type
- Si $I < T_1, ..., T_n >$ est une interface, avec $T_1, ..., T_n$ des paramètres de types, alors si $\tau_1, ..., \tau_n$ sont des types, $I < \tau_1, ..., \tau_n >$ est un type

Vérifications statiques

- vérification de types
  - des déclarations de variable
  - des affectations
  - des passages de paramètres
  - de la valeur de retour d'une méthode
- permet de vérifier des portées de variables
  - variables locales (même type si même nom)
  - portée liée aux modifieurs de visibilité
- des implantations d'interfaces
- des concrétisatiosn de classes abstraites

- de résoudre la surcharge

Relation de subsomption, sous-typage

- principe de subsomption
  - utilisation d'un objet d'une certaine classe/spécification à la place et lieu d'un objet d'une autre classe/spécification
- vérification
  - par la relation de sous-typage (notée $\leq$)
    on peut utiliser une valeur d'un sous-type lorsqu'une valeur d'un super-type est attendue. Dans ce cas la valeur n'est pas changée, elle est juste vue sous un type différent, c'est toujours la même référence.

Sous-types en Java (1)

La relation de sous-typage est réflexive et transitive.
Elle est définie inductivement de la manière suivante :

- si τ est un type alors $\tau \leq \tau$;
- si τ est un sous-type de τ' et τ' est un sous-type de τ'', alors $\tau \leq \tau''$;
- si SC est sous-classe de C, alors $SC \leq C$;
- si SI est sous-interface de I, alors $SI \leq I$;
- si C implante I alors $C \leq I$;
- si $\tau_2 \leq \tau_1$, alors $\tau_2[] \leq \tau_1[]$
- si τ est un type alors $\tau[] \leq \mathit{Object}$

Sous-types en Java (2)

Il faut de plus ajouter pour pouvoir typer la valeur **null**, valeur indéfinie pour tous les types de valeurs allouées (tableaux, objets). On ajoute un type, appelé `nil` qui n'aura que cette valeur.

- si C est une classe, alors *nil* est un sous-type de C
- si I est une interface, alors *nil* est un sous-type de I
- si τ est un type, alors *nil* est un sous-type de $\tau[]$

Conversion de types explicite

opération de transtypage : $(\tau)expr$

indique que l'expression *expr* doit être considérée de type τ

- types primitifs
  - sans perte d'information : octet vers entier
  - avec perte d'information : flottant vers entier
- types de valeurs allouées (objets ou tableaux)
  - même référence vue d'un autre type
  - selon la relation de sous-typage, ajoute ou non un test dynamique pour vérifier la correction du transtypage
    - si $\tau_{expr} \leq \tau$: code correct du point de vue des types

```
1 PointColore pc0 =
2   new PointColore(2,3,"Bleu");
3 String s0 = pc0.getCouleur();
4 Point p2 = (Point)pc0;
```

    - si $\tau_{expr} \not\leq \tau$: nécessite l'ajout d'un test dynamique

```
1 String s1 = ((PointColore) p2).getCouleur();
```

Conversion de types implicite

- en cas d'affectation ou de passage de paramètre, si la valeur passée est d'un sous-type de la valeur attendue :

```
1  Point p0 = new PointColore();
```

- pour certains opérateurs : arithmétiques ou de concaténation des chaînes de caractères

```
1  int x = 3.14 + 4;
2  String s0 = pc0 + " " + p0;
```

- dans le cas où il faut transformer la valeur d'un type primitif vers sa valeur «équivalente» de sa classe associée (classe encapsulante ou *wrapper*) : autoboxing

Boxing/unboxing et autoboxing

- boxing (encapsulation) : la conversion d'une valeur d'un type primitif vers sa classe associée :

```
1 int x = 421;
2 Integer ix = new Integer(x);
```

- unboxing (désencapsulation) : récupérer la valeur d'un objet d'une classe associée à un type primitif

```
1 Double iy = new Double(32.45);
2 double y = iy.doubleValue();
```

- autoboxing : opération automatique d'encapsulation (boxing) ou de désencapsulation (unboxing)

```
1  IListe nil = new ListeVide();
2  IListe l = new ListePleine(33,nil);
```

Autoboxing et égalité

Le fait de convertir un entier (int) en objet de la classe `Integer` fait changer la représentation et donc l'égalité physique.

```
1  package pobj.cours4;
2
3  class Classic {
4      public static void main(String[] args) {
5        int i = 0;
6        int j = 10;
7        i = j << 32;
8        System.out.println (i); // i = 10
9        if (i == j) System.out.println ("0"); // i = j
10       Integer a = new Integer(i);
11       Integer b = new Integer(j);
12       if (a == b) System.out.println ("1"); // a != b
13   }
14 }
```

Autoboxing et égalité

Avec quelquesfois des surprises (exemple tiré du cours de Yann Régis-Gianas) :

```
1  package pobj.cours4;
2
3  class Strange {
4      public static void main(String[] args) {
5        int i = 0;
6        int j = 10;
7        i = j << 32;
8        System.out.println (i); // i = 10
9        Integer a = new Integer(10);
10       Integer b = new Integer(10);
11       if (a == b) System.out.println ("1"); // a != b
12       a++;  // conside'e' comme un int
13       b++;
14       if (a == b) System.out.println("2"); // a == b
15       a = 317; // toujours conside're' comme un Integer
16       b = 317;
17       if (a == b) System.out.println("3"); // a != b
18   }
19 }
```

Types énumérés (1)

Déclaration pour les types énumérés

- une classe avec un nombre fini d'instances correspondant à l'énumération

```
1  package pobj.cours4;
2
3  public enum Couleur {
4    Pique, Coeur, Carreau, Trefle;
5
6    public boolean est_atout(Couleur c) {
7      return c == this;
8    }
9  }
```

- `Couleur.Pique` est une des quatre instances de `Couleur`
- deux méthodes statiques prédéfinies
  - **values()** : retourne un tableau avec toutes les valeurs possibles
  - **valueOf(String)** : retourne l'instance correspondante au nom

Types énumérés (2)

```
1  package pobj.cours4;
2
3  class TestEnum {
4    public static void main(String[] args) {
5      Couleur c1 = Couleur.Pique;
6      Couleur c2 = Couleur.Coeur;
7      Couleur atout = Couleur.Coeur;
8
9      if (c1.est_atout(atout)) System.out.println("1");
10     if (c2.est_atout(atout)) System.out.println("2"); //
11     if (c1.est_atout(Couleur.valueOf("Pique")))
12        System.out.println("3"); //
13     if (c1.est_atout(Couleur.valueOf("Coeur")))
14        System.out.println("4");
15   }
16 }
```

```
1  > java pobj/cours4/TestEnum
2  2
3  3
```

Polymorphisme d'inclusion

- polymorphisme = plusieurs formes, se rapporte au type des paramètres
- autre nom pour la généricité
- différentes classes de polymorphisme
  - paramétrique : un seul code pour des données de types différents (fonctions OCaml, génériques en Java, ...)
  - ad hoc ou de surcharge : un code différent selon les types et le nombre des arguments (surcharge de méthodes en Java)
  - objet ou d'inclusion : un code différent selon les sous-types (sous-classes) ;

Exemple 1 : Point et PointColore

```
1  package pobj.cours4;
2  import pobj.cours3.*;
3  class ExPointsBis {
4    public static void main(String[] args) {
5      Point p0 = new Point();
6      Point p1 = new Point(2,3);
7      PointColore pc0 = new PointColore();
8      PointColore pc1 = new PointColore(2,3,"Bleu");
9      System.out.println(p0 + " " + p1);
10     System.out.println(pc0 + " " + pc1);
11     System.out.println("--------------");
12     p0 = pc0;
13     System.out.println(p0 + " " + p1);
14     System.out.println(pc0 + " " + pc1);
15   }
16 }
```

```
1  > java pobj/cours4/ExPointsBis
2  (0.0,0.0) (2.0,3.0)
3  (0.0,0.0)-INDEFINIE (2.0,3.0)-Bleu
4  --------------
5  (0.0,0.0)-INDEFINIE (2.0,3.0)
6  (0.0,0.0)-INDEFINIE (2.0,3.0)-Bleu
```

Exemple 2 : StackAL

```
1  class Exemple2 {
2    public static void main(String[] args) {
3      IStack s = new StackAL();
4      Point p0 = new Point(1,2);
5      PointColore pc0 = new PointColore(10,12,"Bleu");
6      s.push(p0);
7      s.push(pc0);
8      System.out.println(s.pop());
9      System.out.println(s.pop());
10   }
11 }
```

```
1  > java pobj/cours4/Exemple2
2  (10.0,12.0)-Bleu
3  (1.0,2.0)
```

Résolution de la surcharge

- choix du type de la méthode à employer lors d'un appel de méthode
- résolue STATIQUEMENT selon une relation d'ordre sur la classe de définition et le type des arguments
- le type du résultat n'est pas pris en compte
- il y a des cas où la résolution échoue

```
1  class A :  m2(A) m2(A,A)
2  class B :  m2(B) m2(A,B) m2(B,A)
3  class C :  m2(A) m2(B,C) m2(C,A)
```

avec C qui hérite de B qui hérite de A.

```
1  A a1 = new A();
2  B b1 = new B();
3  C c1 = new C();
4  c1.m2(x,y)
```

Quel est le type de la méthode à utiliser en fonction des types de x et y ?

Sélection du type de la méthode (1)

Au niveau de la classe C on a 8 méthodes dont 5 à 2 arguments

	classe de définition	type de la méthode
m2	A	(A)
	A	(A,A)
	B	(B)
	B	(A,B)
	B	(B,A)
	C	(C)
	C	(B,C)
	C	(C,A)

Sélection du type de la méthode (2)

```
1  c1.m(x,y);
```

Sur l'ensemble des méthodes `m2` du receveur (ici `c1`), on ne conserve que celles dont le type (t1,t2) vérifie :

- type de x $\leq t_1$
- type de y $\leq t_2$

c1.m2(a1,b1) ;

	classe de définition	type de la méthode
m2	A	(A,A)
	B	(A,B)

Le choix de la signature de la méthode se portera sur la méthode la plus spécifique au sens du sous-typage (ici celle de signature `(A,B)`.

Cas d'ambiguïté

Soit une classe B avec 2 méthodes `m2(A,B)` et `m2(B,A)` :

```
1  A a1 = new A();
2  B b1 = new B();
3  b1.m2(a1,a1);
```

Sur les types on obtient B x (A, B) et B x (B, A)
Aucune de ces deux méthodes ne possède un type tel que (A,A) soit plus petit. il y a un clash à la compilation ! ! !

```
1  b1.m2(b1,b1)
```

Ici les deux méthodes ont un type tel que (B,B) soit plus petit, mais aucune des deux méthodes est plus petite que l'autre il y a un clash à la compilation ! ! !

Surcharge et liaison tardive

```
1  package pobj.cours4;
2  class A {
3      void m2(A a) {System.out.println("A1");}
4      void m2(A a1, A a2) {System.out.println("A2");}    }
5  class B extends A {
6      void m2(B  b) {System.out.println("B1");}
7      void m2(B b1, A a2) {System.out.println("B2");}    }
8  class C extends B {
9      void m2(C  c) {System.out.println("C1");}
10     void m2(A a1, A a2) {System.out.println("C2");}
11     void m2(B b1, B b2) {System.out.println("C3");}    }
12 class TestSurcharge {
13   public static void main(String [] args) {
14      A a1 = new A();
15      B b1 = new B();
16      C c1 = new C();
17      B b2 = c1;
18      A a2 = c1;
19      a1.m2(c1,c1);  // A2
20      b1.m2(c1,c1);  // B2
21      c1.m2(c1,c1);  // C3   b2, c1 et a2 partagent un objet
22      b2.m2(c1,c1);  // B2   mais ne sont pas de me^me type
23      a2.m2(a2,a2);  // C2
24 }}
```

Visiteur : extension des traitements

On cherche à séparer les données des traitements dans le but d'étendre les traitements. Pour cela les classes pour les expressions arithmétiques accepteront de recevoir un visiteur qui effectuera un traitement particulier (calcul de la valeur de l'expression, transformation en chaînes de caractères, ...).
On défini alors une classe abstraite visiteur qui pourra visiter chaque élément des expressions arithmétiques :

```
1 abstract class Visiteur {
2   public abstract void visite(CteV c);
3   public abstract void visite(AddV a);
4   public abstract void visite(MultV m);
5 }
```

la classe abstaite pour les expressions devient alors :

```
1 abstract class ExprArV {
2   public abstract void accepte(Visiteur v);
3 }
```

séparation effective des données et des traitements.

Expressions arithmétiques acceptant un visiteur

```
1  class CteV extends ExprArV {
2    private int val;
3    public CteV(int v) {val=v;}
4    public int getVal(){return val;}
5    public void accepte(Visiteur v){v.visite(this);}
6  }
7  abstract class OpBinV extends ExprArV {
8    protected ExprArV fg, fd;
9    ExprArV sous_expr_g(){return fg;}
10   ExprArV sous_expr_d(){return fd;}
11 }
12 class AddV extends OpBinV {
13   public AddV(ExprArV fg, ExprArV fd){
14     this.fg = fg; this.fd = fd;
15   }
16   public void accepte(Visiteur v){v.visite(this);}
17 }
18 class MultV extends OpBinV {
19   public MultV(ExprArV fg, ExprArV fd){
20     this.fg = fg; this.fd = fd;
21   }
22   public void accepte(Visiteur v){v.visite(this);}
23 }
```

Un visiteur de traitement de calcul

```
1  class VisiteurEval extends Visiteur  {
2    private int   res;
3    VisiteurEval(){res=0;}
4    public int  getRes(){return res;}
5
6    public void visite(CteV c){res=c.getVal();}
7
8    public void visite(AddV a){
9      int i;
10     a.sous_expr_g().accepte(this);
11     i=res;
12     a.sous_expr_d().accepte(this);
13     res=res + i;
14   }
15
16   public void visite(MultV m){
17     int i;
18     m.sous_expr_g().accepte(this);
19     i=res;
20     m.sous_expr_d().accepte(this);
21     res=res * i;
22   }
23 }
```

Test sur les expressions avec visiteur

```
1  package pobj.cours4;
2
3  class TestExprArV {
4    public static void main(String[] args) {
5      VisiteurEval v = new VisiteurEval();
6
7      ExprArV e1 = new CteV(10);
8      ExprArV e2 = new CteV(20);
9      ExprArV e3 = new MultV(e1,e2);
10     ExprArV e4 = new AddV(e1,e3);
11
12     e4.accepte(v);
13     System.out.println(e4 + " = " + v.getRes());
14   }
15 }
```

```
1  > java pobj/cours4/TestExprArV
2  pobj.cours4.AddV@1a0f73c1 = 210
```

Il manque un visiteur de conversion en chaîne de caractères pour afficher la formmule, mais le résultat est correct.

Redéfinition et co-variance du résultat

Depuis la version 1.5, il est possible dans le cadre d'une redéfinition de préciser (au sens du sous-typage) le type de retour. Le type du résultat de la méthode redéfinie est en relation de sous-typage avec celui défini dans la sur-classe. On parle de co-variance du type résultat.

- toujours pour éviter les *cast* inutiles

Exemple :

- dans `Point` :

```
1  protected Point clone () {
2    return new Point(getX(), getY());  }
```

- dans `PointColore` :

```
1  protected PointColore  clone () {
2    return new PointColore(getX(), getY());  }
```

- pas besoin de retourner un `Object` pour le transtyper ensuite en `Point` ou `PointColore`.

Surcharge et autoboxing

```
1  package pobj.cours4 ;
2
3  class TS2 {
4   public static int m1(int y){return 2 * y;}
5   public static Integer m1(Integer z){return 5 * z;}
6   public static int m2(int z){return 10 * z;}
7   public static void main(String[] a) {
8   int i = 1;
9   Integer k = new Integer(10);
10  System.out.println(m1(i) + "," + m1(k)); // 2,50
11  System.out.println(m1(m1(i)) +","+ m1(m1(k))); // 4,250
12  System.out.println(m2(i) +","+ m2(k)); // 10,100
13  System.out.println(m2(m2(i)) +","+ m2(m2(k))); // 100,1000
14   }
15 }
```

Algorithmes de résolution de la surcharge (1)

- First Match : on garde la première signature de méthode trouvée.
  simple mais peu efficace

- Perfect Match :
  ne garder que la signature de méthode pour laquelle tous les types sont identiques aux expressions passées en paramètre.
  Pas d'ambiguité mais très restrictive.

Algorithmes de résolution de la surcharge (2)

- Mult : Multiplication des distances entre les arguments et les paramètres de la méthode.

  calcul d'une distance de parentée entre deux types correspondant au nombre de liens de parenté nécessaires pour aller d'un type à l'autre : si B hérite de A et C hérite de B , alors la distance entre A et C est de 2. On calcule donc l'ensemble des distances, auxquelles on ajoute 1, et on les multiplie. On gardera bien entendue la méthode ayant la valeur minimale (si tous les arguments ont le même type que les paramètres de la méthode, on obtient la plus petite valeur possible : 1).

Algorithme de résolution de la surcharge (3)

- Java 1.2 : produit cartésien de sa classe de définition et des paramètres de la méthode.

  Les méthodes plus précises possèdent alors des valeurs plus grandes que les autres, ce qui permet de les sélectionner. A noter que plusieurs méthodes peuvent avoir la même valeur, ce qui peut mener soit à une impossibilité à typer, soit à une sélection purement arbitraire (au choix, par exemple la première méthode trouvée).

- Java 1.5 : produit cartésien des paramètres de la méthode. La classe fournissant la méthode n'est donc plus prise en compte. Cela évite qu'une méthode moins précise mais issue d'une sous-classe soit sélectionnée.

Algorithme de résolution de la surcharge (4)

- Mixte : On prend la signature la plus petite dans la classe la plus petite. A la différence de java 1.2 on ne recherche que dans les méthodes définies les dernières (recherche dans la classe la plus petite).

Exemple en Java 1.2 et 1.5 (1)

B hérite de A :

```
1  class A {
2    int m (A x) { System.out.println(1+" "); return (1); }
3    boolean n (B x) {System.out.println(2+" "); return (true); }
4  }
5
6  class B extends A {
7    int m (B x) { System.out.println(5+" "); return (5); }
8    boolean n (A x) {System.out.println(6+" "); return(true); }
9  }
```

```
1  A a1 = new A ();
2  B b1 = new B();
3  A a2 = b1;
```

Exemple en Java 1.2 et 1.5 (2)

1er appel :

```
1  b1.m(b1);
```

on a 2 méthodes candidates : A.m(A x) et B.m(B x) La résolution ne posera ici aucun souci : la méthode B.m(B x) est dans la plus petite classe, et possède une signature qui colle parfaitement avec le type de l'argument. C'est donc celle qui sera sélectionnée.

Exemple en Java 1.2 et 1.5 (3)

2ème appel :

```
1  b1.n(b1);
```

Les 2 candidats sont ici A.n(B x) et B.n(A x).

- Pour Java 1.2, ces méthodes sont représentées par les couples (A , B) et (B, A), en on n'en trouve pas un plus petit que l'autre : il y a ambiguïté.
- Pour Java 1.5, les méthodes sont représentées par (B) et (A), et la méthode A.n(B x) est donc plus précise. C'est cette signature qui sera sélectionnée.
- Pour Perfect Match et Multiply, la même méthode sera sélectionnée.

Cette différence de traitement entre Java 1.2 et les autres algorithmes montre sur un exemple simple l'importance de l'algorithme de résolution de la surcharge, en particulier dans les cas où il n'y a pas d'erreurs de compilation ; c'est-à-dire quand l'algorithme sélectionne une signature.

Haskell

- fonctionnel
- pur
  sans effets de bord
- à évaluation paresseuse
  stratégie standard du λ-calcul
- typé statiquement (fortement)
  polymorphe paramétrique et
  ad hoc : types de classes pour la surcharge
- avec inférence de types

plusieurs compilateurs (ghc (Glasgox Haskell Compiler)), livres, sites dont le très riche `http://www.haskell.org`.

de OCaml à Haskell (1)

déclaration de type:

`type` pour les synonymes

`data` pour les types avec constructeurs

```
1 data Couleur = Pique | Coeur | Carreau | Trefle
2 data Maybe a = Just a | Nothing
3 data Arbre a b = Feuille b | Noeud a (Arbre a b) (Arbre a b)
```

contrainte de type pour une déclaration :

```
1 succ :: Integer -> Integer
2 succ n = n + 1
```

de OCaml à Haskell (2)

fonctions et applications:

fonction anonyme ($\lambda xy.x + y$)

```
1 \x y ->  x + y
```

composition ($f\ o\ g\ o\ h$)

```
1 f . g . h
```

opérateur d'application $

```
1 a b c x = ((a b) c) x
2 a $ b $ c $ x = a ( b ( c x))
```

de OCaml à Haskell (3)

une écriture équationnelle :

```
1  sign x |  x >  0       =   1
2         |  x == 0       =   0
3         |  x <  0       =  -1
```

la fonction map :

```
1      map :: (a -> b) -> [a] -> [b]
2      map _ [] = []
3      map f (x:xs) = f x : map f xs
4
5  > map (+3) [1,5,3,1,6]
6  [4,8,6,4,9]
7  > map reverse ["abc","cda","1234"]
8  ["cba","adc","4321"]
```

Surcharge

- en OCaml
  - un opérateur différent pour chaque type argument
    - opérateurs de calcul : `+` pour les *int*, `+.` pour les *float*
    - ou des fonctions de conversion explicite : `string_f_int`, `string_of_float`
  - écrire des foctions paramétriques qui explorent la strucutre des valeurs (builtin)
    - égalité structurelle : (=) : `'a -> 'a -> bool` qui explore la structure des arguments en exécutant un code différent
      limitation : impossible à écrire dans le langage
- en Haskell
  - classes de types (ou type class)

Classes de types

Une classe de types définit une propriété sur les types.
Par exemple les types avec égalité :

```
1  class Eq a where
2    (==) :: a -> a -> Bool
```

indique que les types de la classe `Eq` possèdent l'opérateur d'égalité `==`

une classe de types peut avoir une ou plusieurs implantations pour des types donnés :

```
1  instance Eq Bool where
2    x == y = if x then y else not y
3  instance Eq Couleur where
4    x == y = ...
```

Contraintes de typage

On peut ajouter des contraintes de types sur des paramètres de types de fonctions polymorphes : ici le prédicat `mem` qui retourne True si un élément appartient à une liste et False sinon doit pouvoir tester l'égalité du paramètre x avec les éléments de la liste.

```
1 mem : Eq a => a -> [a] -> Bool
2 mem x [] = False
3 mem x (y :: ys) = x == y || mem x ys
```

```
1 > mem True [False, False, False]
2 False
3 > mem True [False, True, False]
4 True
```

Conjonctions de contraintes

Soit la classe `Show a` définissant `show` :

```
1  class Show a where
2    show :: a -> String
```

on peut utiliser des conjonctions de contraintes comme par exemple suivant :

```
1  filter_show :: (Eq a, Show a) => (a -> Bool) -> [a] -> String
2  filter_show p l = foldl  write ""  (filter p l)
3     where write ""  obj = show obj
4           write str obj = str ++ "," ++ show obj
```

la fonction `filter_show` filtre les éléments d'un liste en fonction d'un prédicat et convertit en une seule chaîne de caractères les éléments sélectionnés. Ces éléments de liste sont des instances de `Eq a` et `Sort a`.

contraintes sur les instances

Une instance peut être être paramétrée, et donc on peut faire porter une contrainte sur les paramètres de types.

```
1 data Ordering = LT | EQ | GT
2 class Ord a where
3 compare :: a -> a -> Ordering
```

voici une instance des éléments ordonnés pour des listes quelconques dont les éléments sont ordonnés.

```
1 instance Ord a => Ord [a] where
2 compare [] [] = EQ
3 compare _  [] = GT
4 compare [] _  = LT
5 compare (x : xs) (y : ys) = case compare x y of
6    XEQ -> compare xs ys
7    other -> other
```

Contraintes sur les classes

On peut contraindre qu'une classe de type `SC` fournisse toutes les fonctions d'uen classe de type `C` à la manière de l'héritage.
Par exemple la classe des éléments bornés (Bounded) intègre les méthodes de la classe des éléments ordonnés (Ord)

```
1  class Ord a => Bounded a where
2    maxBound :: a
3    minBound :: a
```

Ainsi toute instance de `Bounded a` aura d'une part la fonction `compare` mais aussi les bornes `maxBound` et `minBound`.

Classes standards

- types avec égalitét ordre :
  - Eq a – x == y
  - Ord a – compare, (<) , (<=)
  - Bounded – minBound, maxBound
- types numériques
  - Num a – structure d'anneau (+, -, *, ...)
  - Frational a – inverse, quotient (/)
  - Floating - ajoute les fonctions trigonométriques et transcendentes
- utilitaires
  - Show a – show : : a -> String
  - Enum a – types indexables par des entiers (fromEnum, toEnum)

Dérivations

- définition automatique sur les énumérations :

```
1  data Couleur = Pique | Coeur | Carreau | Trefle deriving Enum
```

on obtient un fromEnum : `Couleur -> Int` et un `Int -> Couleur`

```
1  data Arbre a = Feuille a | Noeud a (Arbre a) (Arbre a)
2  deriving (Eq, Show)
```

engendrent les instances suivantes :

```
1  instance Eq a => Eq (Arbre a) where ...
2  instance Show a => Show (Arbre a) where ...
```