# *INF 321*
# *Classes, Objets, Exceptions...*

Eric Goubault

Cours 4

21 mai 2011

# Dans l'épisode précédent...

## Cours précédent

- Les types produits - retour sur les tableaux
- Les types somme
- Les types récursifs

## Dans ce cours

- Classes et méthodes dynamiques
- Le traitement des erreurs et les exceptions
- Classes abstraites, interfaces, packages

# Programmation *orientée objet*?

**class? : jusqu'à présent:**

- On n'avait défini que des `class` contenant des données (avec constructeurs néanmoins) - cf. types produits, enregistrements
- Ou des `class` ne contenant que du code: les "programmes"

**Mêlons les deux!**

# Programmation *orientée objet*?

### class? : jusqu'à présent:

- On n'avait défini que des `class` contenant des données (avec constructeurs néanmoins) - cf. types produits, enregistrements
- Ou des `class` ne contenant que du code: les "programmes"

### Mêlons les deux!

# Exemple: static→dynamique

## Philosophie orientée objet (1ère approximation)

Au lieu de:

```
class Pile {
  List c;
  Pile(List x) { this.c = x;} }

class Prog {
  static void push(int a, Pile l) {
    l.c = new List(a,l.c); }
  static void pop(Pile l) {
    l.c = l.c.tl; }
  static int top(Pile l) {
    return l.c.hd; } }
```

- Mettre toutes les méthodes concernant les piles dans `class Pile`!

## On obtiendrait ainsi:

```
class Pile {
  List c;
  Pile(List x) { this.c = x; }
  static void push(int a, Pile l) {
    l.c = new List(a,l.c); }
  static void pop(Pile l) {
    l.c = l.c.tl; }
  static int top(Pile l) {
    return l.c.hd; } }
class Prog {
  ... Pile.push(1,p); ... }
```

- Les fonctions s'appliquant à la *collection* ou *classe* des piles sont comme des *champs* (ou enregistrements - cf. types produits, cours 2 - cf. von Neuman!): *méthodes*
- On les appelle en faisant `Pile.`*méthode*...

# Critique

## Enregistrement?

- Quand on a une `Pile p`, on fait `p.c` pour obtenir son champ de type `List`
- Pourquoi `Pile.push(1,p)` pour les méthodes?

## Champs/méthodes statiques...ou pas

- `c` est un champ non statique (pas de qualificatif `static`)
- `push` est une méthode statique (qualificatif `static`)

Commençons par expliquer les champs statiques/non-statiques avant les méthodes...

## Champs statiques

```
class Stat {
  static int x = 2; }

class Prog {
 public static void main(String[] args) {
  Stat s, t; s = new Stat(); t = new Stat();
  System.out.println(s.x);
  t.x = 3;
  System.out.println(s.x);
```

Donne:

```
> java Prog
2
3
```

Blup?

## Champs non statiques

On change juste:

```
class Stat2 {
  int x = 2; }
```

Donne:

```
> java Prog
2
2
```

Ouf.... ;-)

# Explication

## Champs statiques/non statiques

- `class` (classe) définit un ensemble d'*objets*
- `Stat` (version 1, statique) ne contient qu'un singleton
- `Stat2` (version 2, dynamique) peut contenir n'importe quel nombre d'objet, dont le trait commun est qu'ils contiennent un champ entier `x` initialisé à 2
- Plus généralement, les champs `static` sont communs à tous les *objets* de cette classe

## Et les méthodes dans tout cela?

- Moche ce que l'on a fait: `Pile.push(1,p)`
- On voudrait écrire comme pour les champs quelque chose comme `p.push(1)`: possible avec une méthode `push` non statique!

# On obtient:

```
class Pile {
  List c;
  Pile(List x) { this.c = x; }
...
  void push(int a) {
    this.c = new List(a,this.c); }

  void pop() {
    this.c = this.c.tl; }

  int top() {
    return this.c.hd; }
}
```

"Module" cohérent parlant de piles, avec toutes les opérations légales associées

# Explication?

## Méthodes dynamiques

- De même: méthode `static`, une seule par classe, exemple `main`
- `p.push` est une fonction différente de `q.push` (pour deux piles `p` et `q` distinctes)
- `push` (dynamique) définit une fonction partielle, instanciée par le passage de `p` (appelé `this` - objet courant sur lequel s'applique la méthode) lors de son appel par `p.push(...)` (règles de passage d'argument inchangées)
- Quand on fait `p.push(...)`, la machine regarde le type de `p`, voit `Pile`, trouve l'"enregistrement" (fonctionnel) `push`, passe la référence sur le contenu de `p` à `push` puis exécute son code...

# Remarques importantes!

## THIS...

est le plus souvent implicite: on aurait pu écrire:

```
void push(int a) {
  c = new List(a,c); }

void pop() {
  c = c.tl; }
```

## Le cas de null...

Attention ne jamais faire `p.push(...)` quand `p` vaut `null`!

# Autres remarques

## Types somme

Rend possible une autre implémentation des types somme

## Classes entièrement static

Il en existe... exemple, fonctions de librairie `Math.sin`, `Math.cos` etc.

### Quelques éléments de vocabulaire

communs aux langages *orientés objet*:

- *objet*: structure de donnée composé de:
  - *attributs*: ou champs (cf. types produit, ou enregistrement!), ce sont les données élémentaires composant l'objet
  - *méthodes*: fonctions pouvant s'appliquer à ce type d'objet
- *classe*: ensemble d'objets du même type
- *instance*: on dit qu'un objet est une instance de sa classe

## Méthodes: fonctions associées à une classe d'objets

- *Objet* instance d'une *classe*

```
class Toto {
  int champ1; int champ2;
  int methode1(int parametre) { ... } }
Toto x = new Toto(); ...
```

- contient des *champs* de données et des *méthodes* qui sont des fonctions *s'appliquant* sur tout objet de la classe: `x.methode1(parametre)` (en C, non orienté objet, on écrirait `methode1(x,parametre)`)

## Qualificatifs

- méthodes `static` (pas *dynamiques*), ou méthodes de classe, on ne peut faire que `methode1(parametre)` ou `Toto.methode1(parametre)`, pas de connaissance d'un objet `x` particulier, mais juste de la classe `Toto`
- méthodes `public` sont connues de tout le monde
- méthodes particulières dites *constructeurs* (déjà vues au cours 3 etc.)

# Petit historique de l'"orienté objet"

- Simula 67 (basé sur Algol 60)
- Smalltalk 71/80
- Objective C, C++ 83...
- Beaucoup depuis...(Java, OCaml en particulier)

Exemple typique de programmation orienté objet: interfaces graphiques (ex. QT4), ateliers graphiques de programmation (Google App Inventor, cf. `http://appinventor.googlelabs.com/about/`) etc.

## Utilité principale

C'est essentiellement un *style* de programmation:

- Permet de bien "cloisonner" le code, en unités cohérentes, cf. diagrammes de classes et UML (*Unified Modelling Language*)
- Permet de réutiliser du code, plus aisément ("composants"), par héritage en particulier (après...)

## Coupure pub?

- Dans ce qui suit, démo d'un environnement graphique de programmation d'applications Android (smartphones)
- Dans la grande tradition des langages graphiques, pédagogiques (Logo, Alice, Kodu etc.), ou non (Hypercard, Scade/Lustre, QT etc.)
- Cet environnement permet de manipuler des *classes* d'objets:
  - graphiques (pour l'interface) comme des boutons, des zones de texte, des images etc.
  - non-apparents: son, données GPS, connexions wifi/3G etc.
- La création d'objets se fait graphiquement, leurs enregistrements peuvent être modifiés à l'interface
- Pour chaque objet, on a une représentation graphique des méthodes associées que l'on combine pour programmer...

(Rappel: Android basé sur Linux, applications écrites majoritairement en Java - plus la dessus en INF 422)

# Demo Google App Inventor (merci à youtube)

Ma video

# Héritage (simple): intérêt important de l'orienté objet!

## Réutilisation de code

On a défini une classe `Point` et ses méthodes:

```
class Point {
  int x, y; // coordonnees
  translation(int u, int v) {
    x = x+u; y = y+v; } ... }
```

On veut maintenant des points colorés, au lieu de redéfinir (à gauche), on peut écrire (à droite):

```
class ColorPoint {
 int x, y; // coordonnees
 int col; // couleur
 translation(int u, int v)
  x = x+u; y = y+v; } ...
```

```
class ColorPoint
        extends Point {
  int col; // couleur
  ... }
```

### Déclaration d'héritage

Par mot clé `extends`: `class A extends B ...` , `A` hérite de `B`

### Champs

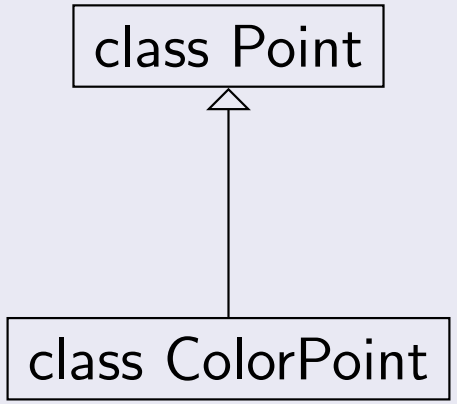
- Un `ColorPoint cp` aura des champs `col` (accessible par `cp.col`), mais aussi `x` et `y` (accessibles par `cp.x` et `cp.y`)

### Méthodes

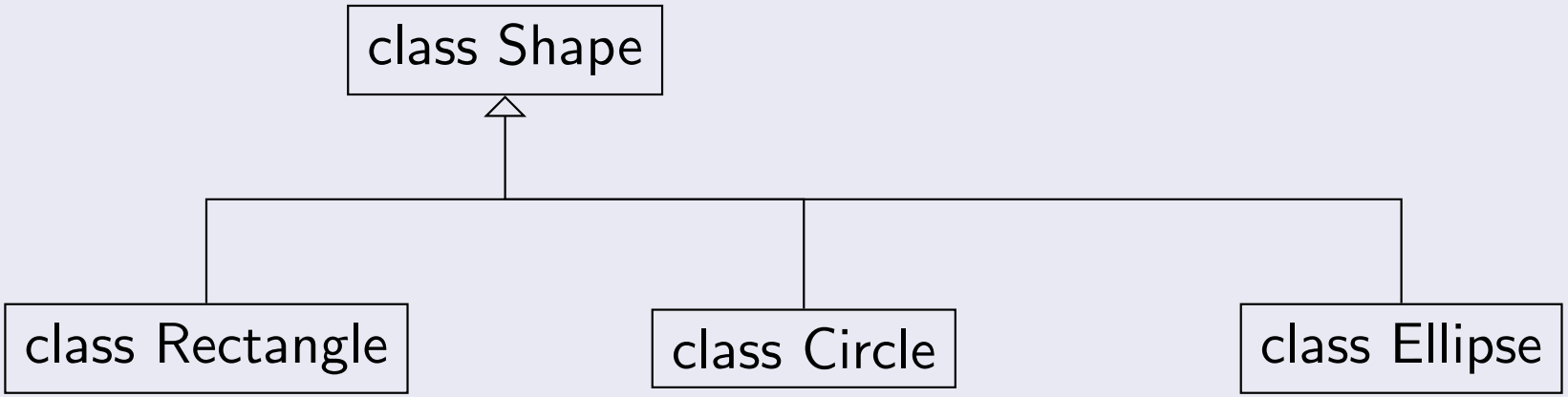
- `translation` s'applique implicitement sur un objet de la classe `ColorPoint` en s'appliquant à sa "sous-partie" `Point`

Cela s'appelle l'*héritage*, et permet une économie et une structuration du code meilleure (en Java et Caml, pas en C)
Remarque: tous les objets Java héritent d'une classe unique: `Object`

## Diagramme de classe pour Point, ColorPoint




## Autre exemple



# Au delà de l'héritage simple?

## Héritage multiple

- pouvoir hériter du code de plusieurs classes
- compliqué sémantiquement!
- existe en C++ pas en Java
- forme dégradée en Java, sémantiquement raisonnable: héritage simple de code, multiple de *signatures* (`interface`) - fin du cours

## Android (cf. http://developer.android.com/reference/packages.html)

java.lang.Object
↪ android.graphics.Paint

The Paint class holds the style and color information about how to draw geometries, text and bitmaps.

Public Constructors

```
Paint()
```

Create a new paint with default settings.

```
Paint(int flags)
```

Create a new paint with the specified flags.

```
Paint(Paint paint)
```

Create a new paint, initialized with the attributes in the specified paint parameter.

## Paint

Public Methods ...

```
int getColor()
```

Return the paint's color.

```
float getFontMetrics(Paint.FontMetrics metrics)
```

Return the font's recommended interline spacing, given the Paint's settings for typeface, textSize, etc.

```
float getTextSize()
```

Return the paint's text size.

```
void setColor(int color)
```

Set the paint's color.

...

## Android...

java.lang.Object
↪ android.graphics.Paint
↪ android.text.TextPaint

Class Overview

TextPaint is an extension of Paint that leaves room for some extra data used during text measuring and drawing.

Summary

Inherited Constants

```
From class android.graphics.Paint
...
```

Fields

```
public int baselineShift
public int bgColor
public float density
public int[] drawableState
public int linkColor
```

# TextPaint

## Public Constructors

```
TextPaint()
TextPaint(int flags)
TextPaint(Paint p)
```

## Public Methods

```
void set(TextPaint tp)
```

Copy the fields from tp into this TextPaint, including the fields inherited from Paint.

## Inherited Methods

```
From class android.graphics.Paint
From class java.lang.Object
```

# Programmation orientée objet - deuxième résumé

## Vocabulaire

- *héritage*: une classe peut être une *sous-classe* d'une autre: la sous-classe l'*étend* en rajoutant des méthodes et des attributs, elle accède aux attributs et aux méthodes de sa sur-classe
- *polymorphisme (par sous-typage)*: un objet d'une sous-classe `A` de `B` en accédant aux méthodes de `B` est considéré du type `B` (et `A`)

# Mais aussi polymorphisme pour les méthodes!

## Polymorphisme des opérateurs

Exemple: calcul garanti, par arithmétique rationnelle:

```
class Rat {
 int p, q;
 Rat(int x, int y) { p=x; q=y; }
 Rat plus(Rat y) {
   return new Rat(this.p*y.q+this.q*y.p,
                                  this.q*y.q); }
 void show() {
   System.out.println(p+"/"+q); }
}
```

# Exemple de redéfinition d'arithmétique

## Polymorphisme des opérateurs

Exemple: calcul garanti, par arithmétique d'intervalles:

```
class Doubleint {
  double inf, sup;
  Doubleint(double x, double y) { inf=x; sup=y; };
  Doubleint plus(Doubleint y) {
    return new Doubleint(this.inf+y.inf,
                         this.sup+y.sup); }
  void show() {
    System.out.println("["+inf+","+sup+"]"); }
}
```

`plus` et `show` sont *polymorphes*: peuvent prendre des `Rat` ou des `Doubleint`! (pratique, on peut ne changer que la donnée et pas le programme...)

# Exemple d'exécution

## Polymorphisme des opérateurs

```
class Prog {
 public static void main(String[] args) {
   Rat r = new Rat(1,2);
   Rat s = new Rat(1,3);
   Rat t = r.plus(s); // = s.plus(r);
   Doubleint ri = new Doubleint(0.50,0.50);
   Doubleint si = new Doubleint(0.33,0.34);
   Doubleint ti = ri.plus(si); // = si.plus(ri);
   t.show();
   ti.show();
 }
}
```

Donne:

```
5/6
[0.8300000000000001,0.8400000000000001]
```

### Reprenons le code de pop():

```
void pop() {
  this.c = this.c.tl; }
```

### Comment traiter le cas this.c == null?

```
void pop() {
  if (this.c != null)
   this.c = this.c.tl; }
```

Problème: laisser `this.c` à `null` est trompeur... mais que faire d'autre? renvoyer un code d'erreur? (`int pop()`): moche...

## Le problème des erreurs en informatique...

- D'ailleurs que se passe t-il si l'on fait?:

```
Pile p = empty();
p.pop();
```

```
Exception in thread "main" java.lang.NullPointerException
at Prog.pop(Pile.java:16)
at Prog.main(Pile.java:23)
```

- Trait moderne des langages de programmation: les exceptions

# Qu'est-ce qu'une exception?

## Un essai de définition

- C'est le résultat d'un comportement erroné
- Pas vraiment comme une erreur au sens classique du terme: c'est un *objet* traitable par le programme
- Tout calcul peut *lancer* ("throw") une exception, c.-à-d. retourner en plus d'une valeur, un type d'erreur, à son appelant
- Qui pourra la *récupérer* ("catch") pour la traiter, ou la relancer à son appelant etc.

## A distinguer d'une *erreur*

- Qui ne peut être traitée
- Qui arrête le programme dans un état potentiellement incohérent

# Une remarque en passant...

### Concept absent du C

Il faut traiter les cas d'erreur de façon ad hoc...

### Concept présent dans les langages modernes

Caml, Java, Ada... Sauf qu'il faut traiter les exceptions, sinon...

# Une remarque en passant...

## Concept absent du C

Il faut traiter les cas d'erreur de façon ad hoc...

## Concept présent dans les langages modernes

Caml, Java, Ada... Sauf qu'il faut traiter les exceptions, sinon...

## Classes

- `Exception`
- avec pour sous-classes, `IOException`, `RuntimeException` etc.

## D'où création d'une nouvelle exception...

...par héritage:

```
class Monexception extends Exception {
  ...
}
```

Et instanciation par:

```
new Monexception();
```

## Déclarer qu'une méthode peut lancer une exception

```
void pop() throws Monexception {
  ... }
```

dit que `pop()` ne renvoie rien comme donnée, mais peut lancer une exception, rattrapable par l'environnement...

## *Lancer* une exception

On ne doit pas faire `return new Monexception();`, mais plutôt `throw new Monexception();` par exemple:

```
void pop() throws Monexception {
  if (this.c == null) throw new Monexception();
  this.c = this.c.tl; }
```

`throw` a une sémantique un peu comme `return`, interrompt le code...

# *Rattraper* une exception

## Code

```
try { p.pop(); }
catch (Monexception e) {
 System.out.println("Pile␣vide!");
 ... } ...
```

## Exécutions possibles:

- Si `p.c` n'est pas `null`, alors `p.pop()` termine normalement, sans lever d'exception; le code se poursuit normalement au "..."
- Si `p.c` est `null`, alors `p.pop()` lance une exception, qui est rattrapée par `catch (Monexception e)`, `e` vaut alors l'objet de type `Monexception` crée par `p.pop()`
- On peut lancer de nouveau cette exception à l'appelant de l'appelant et ainsi de suite...

Pour les TDs: utilisez `Ppl.failWith(...);`

# *Rattraper* une exception

## Code

```
try { p.pop(); }
catch (Monexception e) {
 System.out.println("Pile␣vide!");
 ... } ...
```

## Exécutions possibles:

- Si `p.c` n'est pas `null`, alors `p.pop()` termine normalement, sans lever d'exception; le code se poursuit normalement au "..."
- Si `p.c` est `null`, alors `p.pop()` lance une exception, qui est rattrapée par `catch (Monexception e)`, `e` vaut alors l'objet de type `Monexception` crée par `p.pop()`
- On peut lancer de nouveau cette exception à l'appelant de l'appelant et ainsi de suite...

Pour les TDs: utilisez `Ppl.failWith(...);`

## Principales différences avec Java

- Enregistrements et objets sont les mêmes choses en Java
- Enregistrements et objets sont différents en Caml (la partie orienté objet est une sur-couche, venue longtemps après...)
- Seules les méthodes peuvent accéder aux champs en Caml (pas visible autrement) - correspond à *private* de Java (hors programme...)

## Exemple Caml

```
class pile = object
 val mutable c = ([]: int list)
 method get_c () = c
 method testempty () = c=[]
 method push x = c<-x::c
 method pop () = c<-List.tl c
 method top () = List.hd c
end
```

## Qu'est-ce que c'est?

- Ensemble de déclaration de méthodes sans implémentation
- Définies par le mot clé `interface`
- Permettent de déclarer des variables avec le type de l'interface, mais pas instanciable! (pas de constructeur en particulier)
- Une classe peut implémenter une ou plusieurs interfaces, par le mot clé `implements`
- Permet de faire de l'héritage multiple en quelque sorte, mais seulement simple, de code

# Exemple - quasi fonctionnel ;-)

## Interface pour les fonctions de $\mathbb{N}$ dans $\mathbb{N}$

```
interface Function {
  public int apply(int n);
}
```

Une seule méthode à implémenter pour être une fonction de $\mathbb{N}$ dans $\mathbb{N}$: l'application!

# Exemple - quasi fonctionnel ;-)

## Des fonctions implémentant cette interface

```
public class Carre implements Function {
  public int apply(int n) {
    return n*n; }  }

public class Fact implements Function {
  public int apply(int n) {
    ... return fact; }  }

public class Exemple {
  public static void main(String[] args) {
    Carre x = new Carre(); Fact y = new Fact();
    System.out.println("Carre(3)="+x.apply());
    System.out.println("Fact(4)="+y.apply());  }
```

### Relation de sous-typage

On note $T \leftarrow S$ si $S$ est un *sous-type* de $T$, défini par:

- $T \leftarrow T$
- si la classe $S$ est une sous-classe de $T$, on a $T \leftarrow S$
- si l'interface $S$ est une sous-interface de $I$, on a $I \leftarrow S$ ;
- si la classe $C$ implémente l'interface $I$, on a $I \leftarrow C$
- si $T \leftarrow S$, alors $T[] \leftarrow S[]$
- si $S \leftarrow SS$ et $T \leftarrow S$, alors $T \leftarrow SS$

### Propriété fondamentale

Si $T \leftarrow S$, alors toute valeur du type $S$ peut être utilisée en lieu et place d'une valeur de type $T$ (transtypage implicite)

# Transtypage?

## Implicite

Si $S$ sous-type de $T$, on peut faire le transtypage (cast) implicite:

```
S x = new S(...);
T y = x;
```

## Dans l'autre sens?

Interdit! (même quand finalement assez similaires...)

```
class X { float n; }
class Y extends X {
  public static void main(String[] args) {
    X x = new X();
    Y y = x; }  }
```

A la compilation on obtient une erreur:

```
javac Y.java
Y.java:5: incompatible types
found   : X
required: Y
    Y y = x; }
          ^
1 error
```

# Transtypage explicite

Il faut faire un transtypage (cast) explicite:

```
class X { float n; }
class Y1 extends X {
  public static void main(String[] args) {
    X x = new X();
    Y1 y = (Y1) x; }  }
```

A la compilation tout se passe bien...mais le bon comportement dans ce cadre dépend entièrement de l'utilisateur ($\neq$ typage fort à la CAML, cf. cours 9)

## En C

Le *cast* est d'emploi très courant, par exemple à l'allocation (cf. cours 2, listes...):

```
List res = (List) malloc(sizeof(struct stList));
```

car `malloc` renvoie le type fourre-tout `void *`...

# Classes abstraites

## Qu'est-ce que c'est?

- Sorte d'intermédiaire entre interfaces et classes (dites *concrètes*)
- Permettent de définir des variables avec le type correspondant et permettent l'héritage (comme les classes concrètes)
- Permettent de spécifier des *méthodes abstraites* avec l'attribut *abstract*, qui sont des spécifications de méthodes, mais pas leur implémentation; pas de *new* possible (comme les interfaces);
- On peut mélanger des définitions de *méthodes concrètes* (implémentées) et des *méthodes abstraites* (différent des interfaces)
- Une sous-classe d'une classe abstraite n'implémentant pas toutes les méthodes abstraites d'une classe abstraite doivent être déclarées *abstract*

## Expressions arithmétiques:

$$expr = Var \mid Cste \mid expr + expr \mid expr * expr \mid -expr$$

```
class Expr {
  int select;
  int Cste;
  String Var;
  Typop Op;
  Expr gauche;
  Expr droite; ... }
```

```
enum Typop {
  plus, minus, times }
```

- on inclut tout dans le type produit... ($\mathbb{R} + \mathbb{R} \subseteq \mathbb{R} \times \mathbb{R}$, égalité dans les espaces vectoriels...)

# Mais "propre" avec les constructeurs

## Du coup

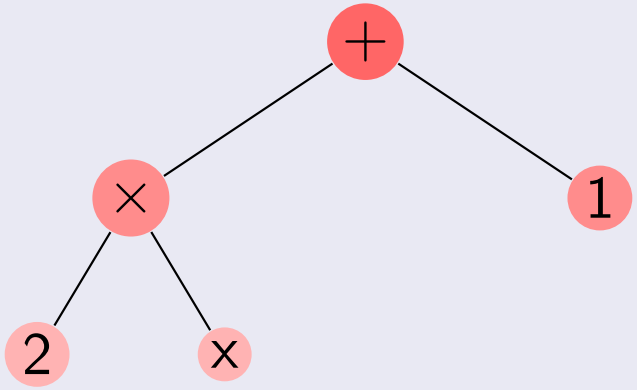
- On utilise `select`: si 0, l'expression est une constante (champ `Cste`), si 1, l'expression est une variable (champ `Var`), si 2, l'expression est un opérateur binaire (`Op` est `plus` ou `times`) et les sous-expressions sont `gauche` et `droite`, ou l'expression est unaire (`Op` est `minus`) et `gauche` est la sous-expression
- il est alors commode de définir plusieurs constructeurs selon les cas (qui "imitent" les injections dans le type somme):

```
Expr(int constante) {
  select = 0; Cste = constante; }
Expr(String variable) {
  select = 1; Var = variable; }
Expr(Typop operateur, Expr argl, Expr argr) {
select=2; Op=operateur;gauche=argl;droite=argr;}
Expr(Expr arg) {
  select = 2; Op = minus; gauche = arg; }
```

# Exemple

**Une expression: $2 * x + 1$ est représentée comme:**

```
Expr e1 = new Expr(1);
Expr e2 = new Expr(2);
Expr e3 = new Expr("x");
Expr e4 = new Expr(times, e2, e3);
Expr e5 = new Expr(plus, e4, e1);
```



# Exemple (Denis Monasse)

## Retour sur les types somme...

```
abstract class Carte {
  public static final int PIQUE = 0,
    COEUR = 1, CARREAU = 3, TREFLE = 4;
  int couleur;

  abstract int valeur(int couleur_atout); }

class As extends Carte {
  int valeur(int couleur_atout) { return 11; }

class Valet extends Carte {
  int valeur(int couleur_atout) {
    if (couleur==couleur_atout) return 20;
    else return 1; }
...
```

De même, vous pourriez reprendre les expressions arithmétiques...

# Paquetages et collections

- Permet d'organiser un ensemble de classes et d'interfaces en un tout cohérent (sorte de *module* à la CAML)
- Limite en particulier la portée des identificateurs
- Un paquetage peut avoir un nom, et des sous-paquetages (une arborescence, comme des répertoires UNIX...)

## Exemple

- L'API standard de JAVA est organisée en paquetage et sous-paquetages!
- Paquetage `java` qui contient comme sous-paquetages `java.lang`, `java.io`, `java.util` etc.

### Définir un paquetage

```
package monPackage;

public class A { public methodeA() { ... } }
public class B { public methodeB() { ... } }
```

## Importer un paquetage

```
import monPackage;

...
new A().methodeA();
new B().methodeB();
```

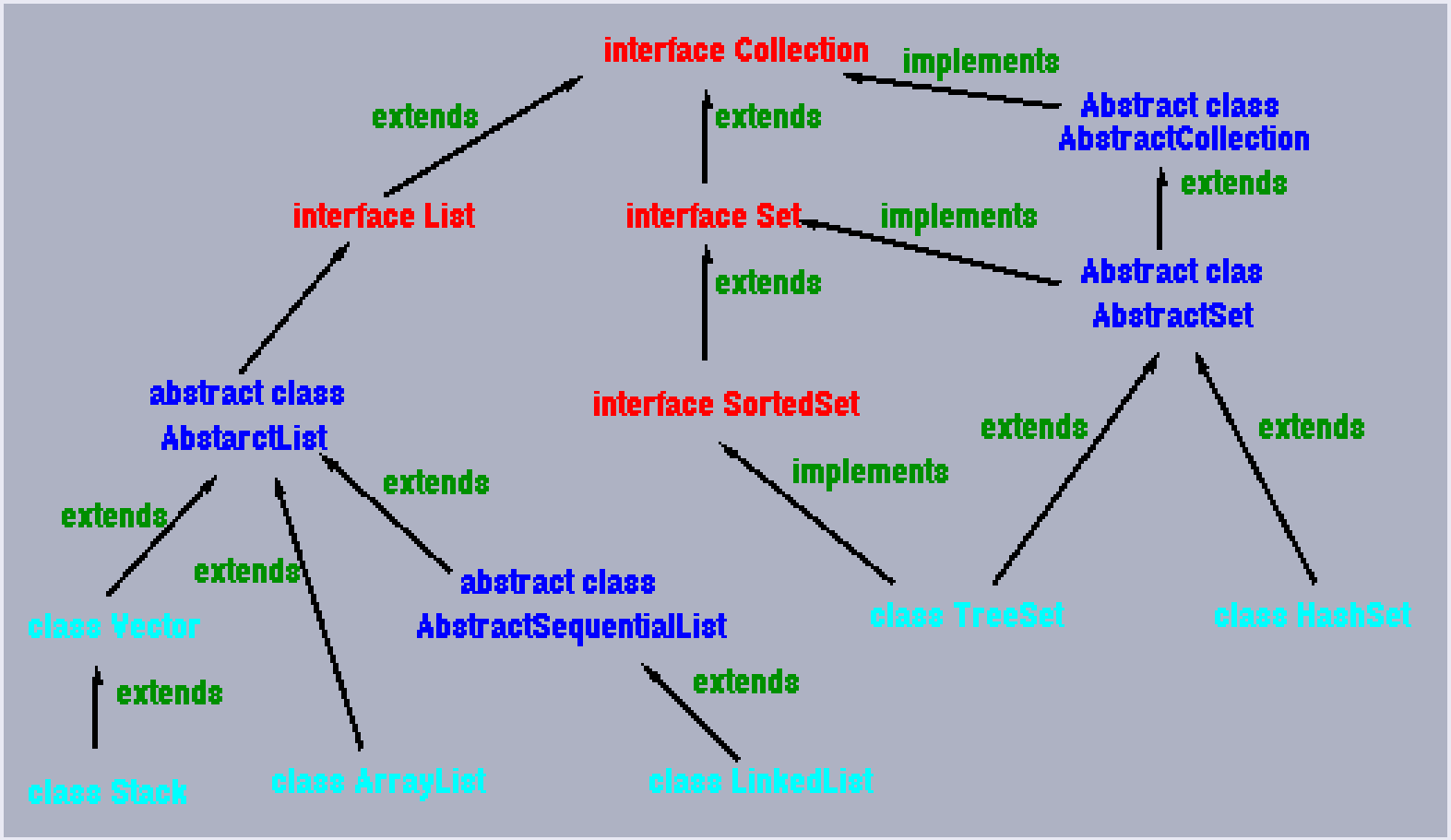
- On peut importer toutes les fonctions d'un paquetage (mais pas récursivement dans les sous-paquetages):

```
import java.io.*;
```

- Remarques: `java.lang` est toujours importé automatiquement - toutes les classes définies jusqu'à présent étaient dans le paquetage *anonyme*!

## Collections

Pour aller plus loin, les collections, qui permettent de représenter divers types d'éléments d'objets JAVA...



# C'est tout pour aujourd'hui..

## La prochaine fois

- Les fonctions récursives (programmation)
- Les fonctions récursives primitives et partielles (théorie de la calculabilité)

## Bon TD!

# C'est tout pour aujourd'hui..

## La prochaine fois

- Les fonctions récursives (programmation)
- Les fonctions récursives primitives et partielles (théorie de la calculabilité)

## Bon TD!