

# Programmation par Objets 1

## *Module GLIN505*

*Animé par :*  
Marianne Huchard  
Abdelhak-Jamel Serai  
Chouki Tibermacine

<http://www.lirmm.fr/~huchard/>  
onglet “Teaching”

# Programmation par Objets 1

## Contenu

*Rappels et approfondissement*

Classes/instances, attributs, opérations/méthodes  
Héritage, polymorphisme

*Nouveautés*

Assertions, Exceptions

Interfaces-langage

Etude de certaines parties de l'API standard

Généricité (polymorphisme paramétrique)

Introspection

Événements

Interfaces graphiques (premiers éléments)

Sérialisation

# Cours 1

Principes de la POO

Les classes en Java

La conception par contrat : les assertions

# Les principes de la programmation par objets

***Une vision centrée sur une représentation des concepts***

- du domaine métier (**entreprise, client**)
- du problème (**livraison, facture**)
- de la solution informatique (**liste, pile, menu, bouton**)

***Le concept encapsule données et fonctions***

# Bref historique

## 1965-1980

Programme = animation d'un modèle réduit, reproduction des entités du monde réel

Simula, Smalltalk, Flavors

## 1980-1990

Montée de l'engouement dans les services R&D et chez les gros industriels, développement des méthodes

C++, CLOS, Eiffel, Ada9X, Object-Cobol, Smalltalk, OMT

## 1990-2010

Diffusion généralisée

Java, Python, C#, UML

# Les domaines

***Domaines pénétrés***

**Systèmes d'information**

**Bases de données (objet-relationnel)**

**Intelligence Artificielle**

**Programmation**

- standard, distribuée, concurrente, interface graphiques, scientifique, système, réseau, etc.**

# Les principes

***L'équation de la programmation par objets***

Objets/classes

Messages

Héritage

***Programme =   objets + envoi de messages***

# Les bénéfices

- Réutilisabilité
  - briques bien délimitées, généralisation encouragée
- Extensibilité
  - par spécialisation
- Stabilité
  - les concepts d'un domaine sont plus stables que les fonctions
- Passage à l'échelle
  - structuration modulaire

# A propos de Java

**Sun, fin 1995**

**Une allure C++ ...**

- **syntaxe, constructeurs, classes paramétrées (après Java 1.5)**

**Mais un langage simplifié**

- **pas d'héritage multiple entre classes**

- **pas de surcharge d'opérateurs**

**Et s'inspirant des stratégies Smalltalk**

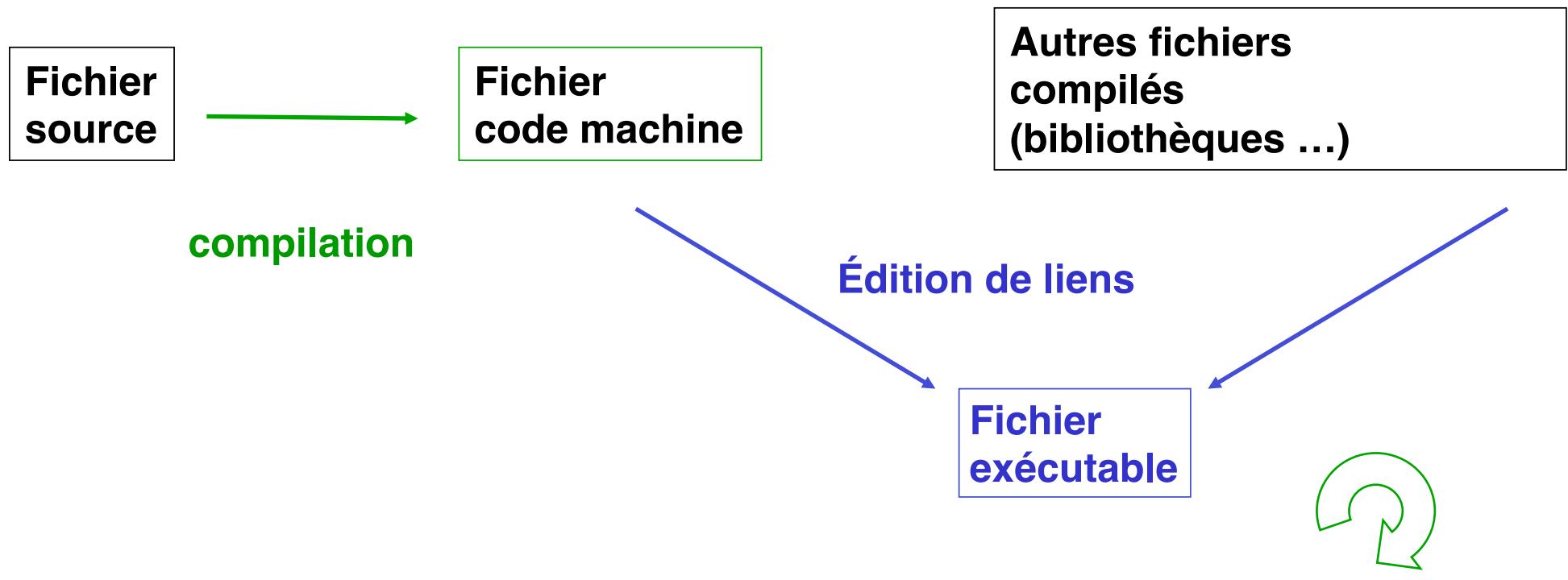
- **semi-compilé/semi-interprété, machine virtuelle**

- **API riche**

- **Allocation/récupération dynamique, ramasse-miettes**

# Langages compilés

*Pascal, ADA, C, C++, ...*



code machine lié à l'architecture  
de la machine

*exécution*

# Langages interprétés

*Lisp, Prolog, shell Unix,  
Javascript, php, ...*

Fichier  
source

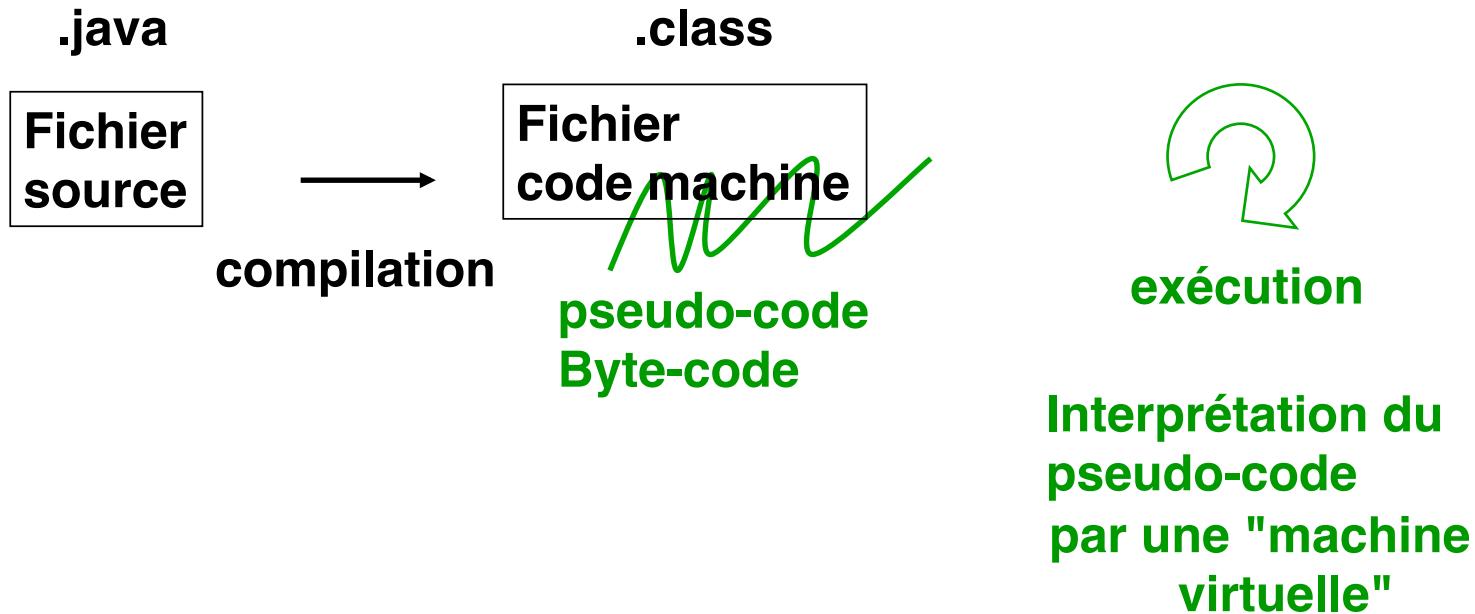


Autres fichiers sources  
chargés à la demande

*Exécution  
par un interpréteur*

Interprétation : traduction du code source en code machine « à la volée »

## Et Java?



**Pseudo-code** : code pour une **machine abstraite** possède des fonctionnalités communes à toutes les architectures

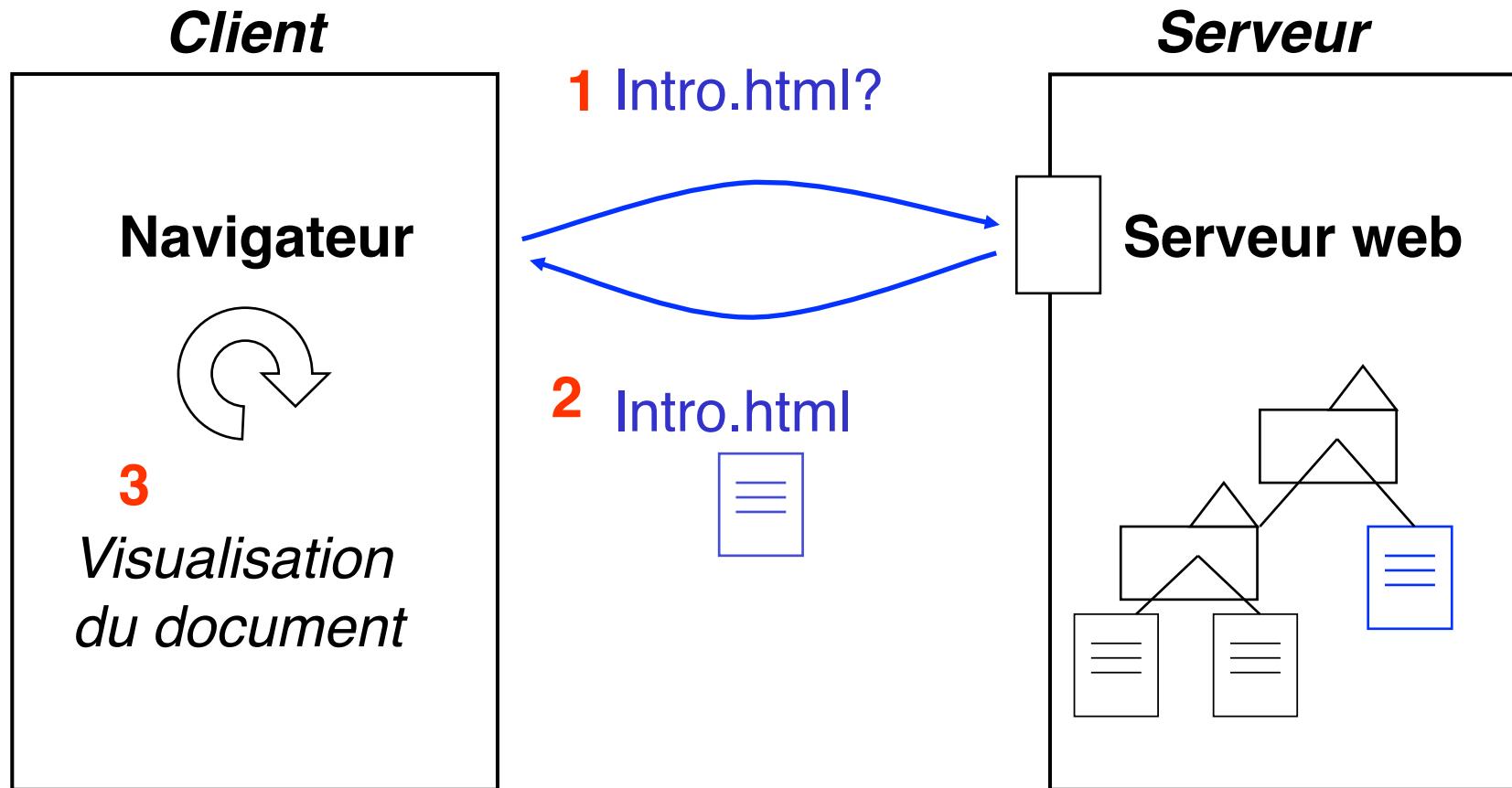
- Pré-compilation en un code universel (**pseudo-code ou byte-code**) donc **indépendant de toute architecture**
- Puis interprétation par une **machine virtuelle** (propre à chaque architecture de machine)

- + **portabilité du code compilé**
- **exécution moins rapide**

Toutefois il existe des compilateurs en code natif

**Les compilateurs JIT – Just In Time**

## Le succès de Java a d'abord été lié au web



# Applet

programme invoqué dans un document **HTML** et exécuté par un **navigateur**

```
<OBJECT  
classid="http://www.ici.fr/monApp.class">  
</OBJECT>
```

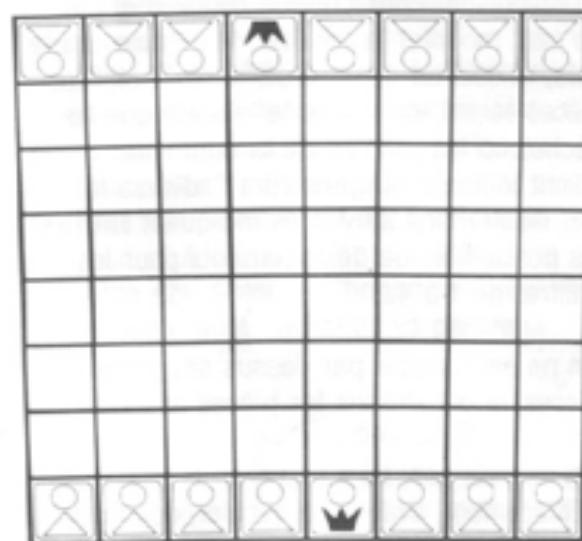
*(depuis HTML 4, <OBJECT> remplace <APPLET>)*

# Trois sortes de programmes Java

- **Applet**  
**programme invoqué dans un document HTML et exécuté par un navigateur**
- **Application** : programme "classique"
- **Servlet**  
**programme invoqué par un serveur web et exécuté sur la machine du serveur**

# Java en tant que langage de programmation par objets

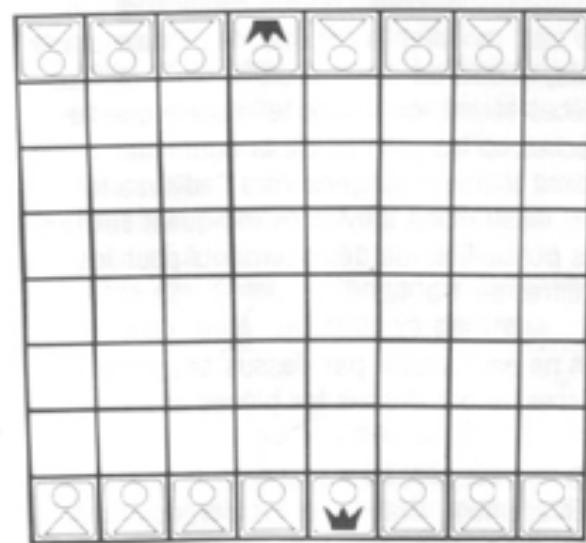
**Exemple d'application : Shogun**



# Shogun

Jeu de pions (pièces)

- deux familles de (7 pions + 1 shogun)
- basé sur la capture de pion
- déplacement rectiligne et à angle droit
- valeur du déplacement suivant déterminée au hasard à chaque arrivée d' un pion sur une case



Une **application = un ensemble d'objets** qui ont chacun leur rôle à jouer et qui interagissent

*Objets informatique, ici des composants graphiques*

**fenêtre**

**barre de menu**

**bouton**

**zones de texte**

**panneau (sur lequel est affiché le jeu)**

*Objets « métier » ...*

## *Objets « métier » :*

**jeu**

*gère une partie*

**échiquier**

*gère la "grille" elle-même*

**ensemble de pièces 1**

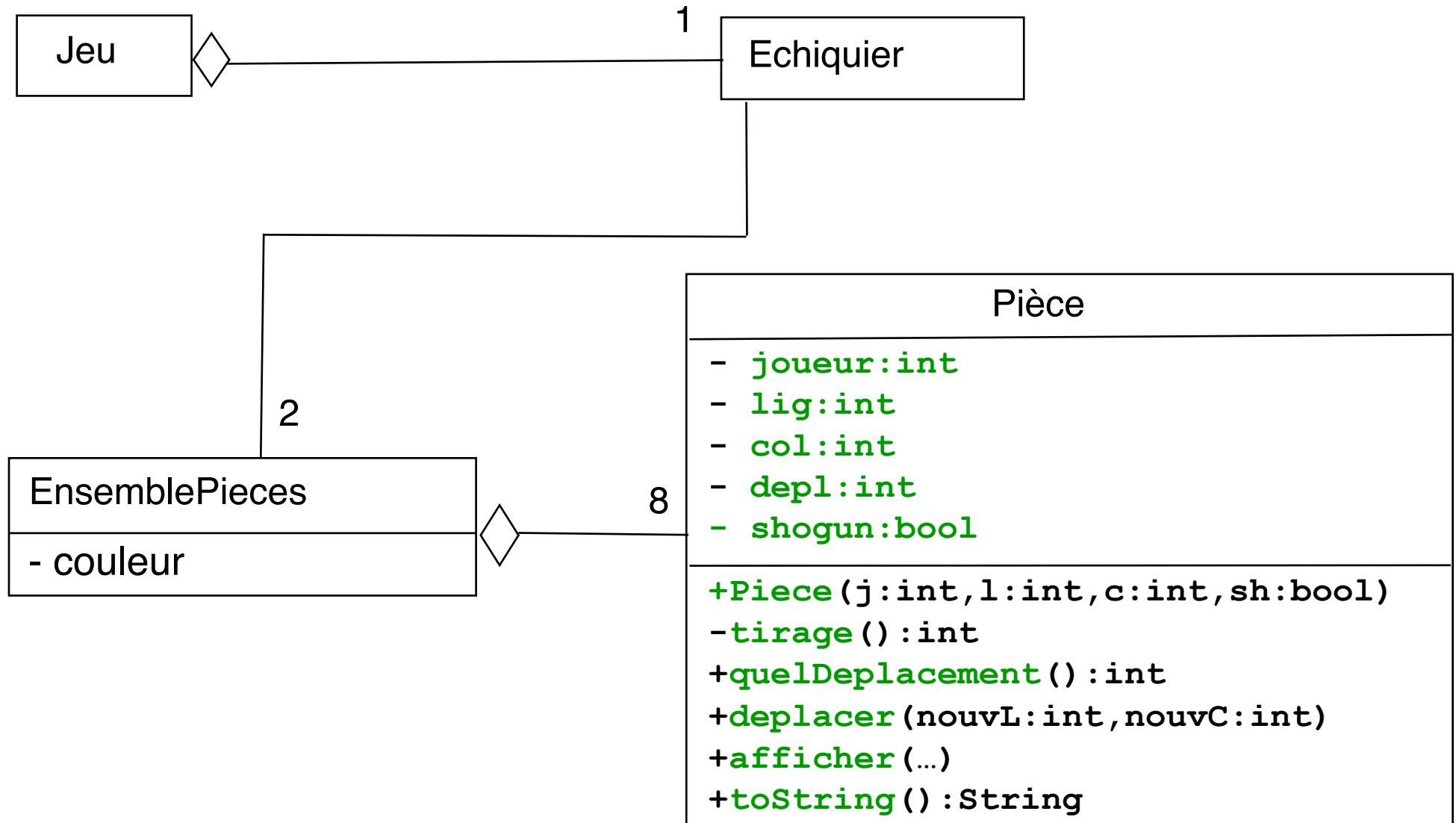
**pièces**

**ensemble de pièces 2**

**pièces**

*une pièce gère sa position sur l'échiquier  
elle sait se déplacer et s'afficher*

## *Objets « métier » --> Représentés par des classes*



Les objets collaborent entre eux, chacun étant **responsable** des actions qu'il effectue

*Ex : afficher le jeu*

**panneau jeu** : « jeu, affiche-toi »

**jeu** : « échiquier, affiche-toi »

**échiquier** : « ens. pièces 1, affiche-toi »

**ens. pièces 1** : à chaque pièce : « pièce, affiche-toi »

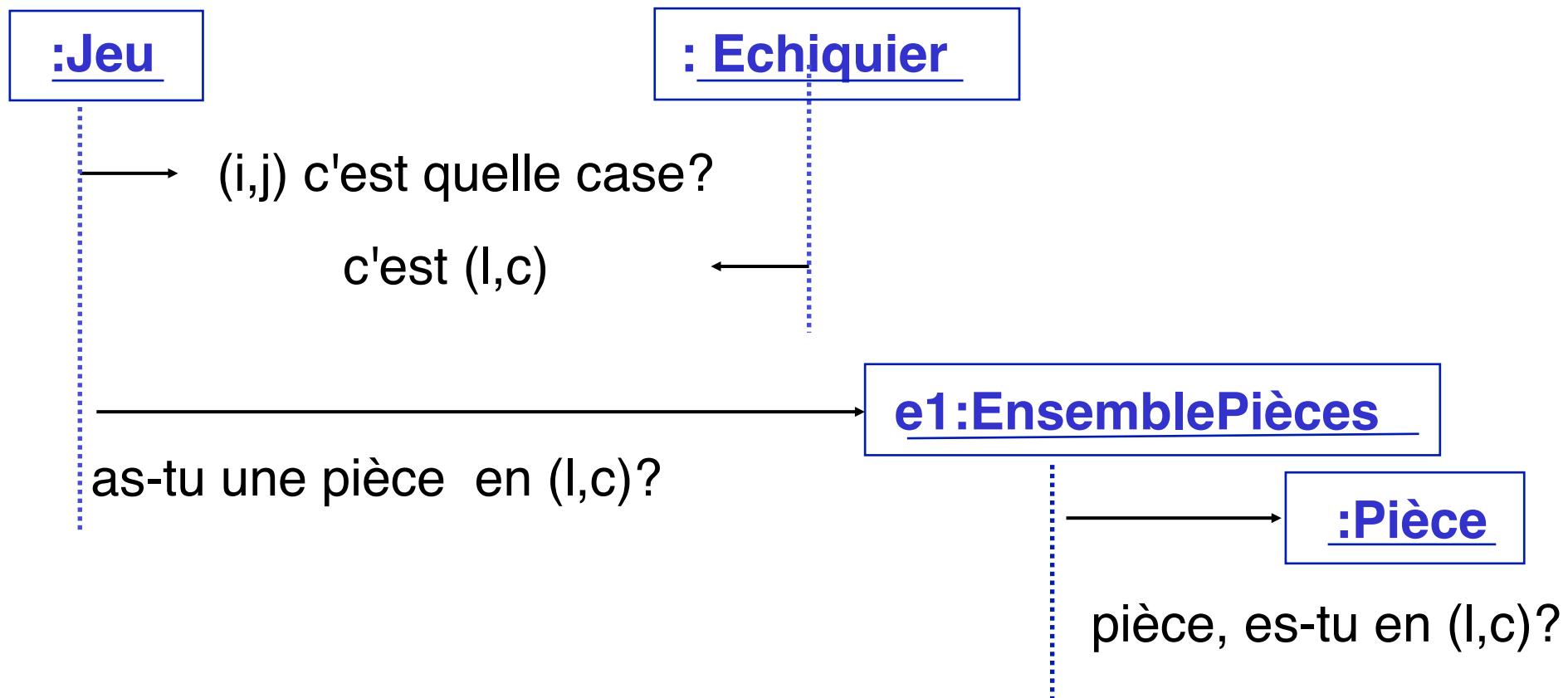
« ens. pièces 2, affiche-toi »

**ens. pièces 2** : à chaque pièce : « pièce, affiche-toi »

## Ex: gérer le début de mouvement du joueur 1

Clic! sur panneau de jeu en (i,j)

jeu est averti



Les objets sont des **instances** de **classes**

Une classe est un « modèle » qui définit :

- la structure d'un objet (**ses attributs**)
- et son comportement (**ses méthodes**)

La structure des objets est (+ ou -) **encapsulée**,  
*ainsi que certaines de leurs méthodes*

## ***Une classe ordinaire***

**Partie privée**

**"encapsulée"**

**attributs**

**+ des méthodes de « cuisine interne »**

**Partie publique**

**méthodes de « services offerts »**

**private** : accessible **seulement** dans le corps des méthodes de la classe

**public** : accessible par **toute** méthode de toute classe

```
// fichier Piece.java

public class Piece
{
    // attributs

    private int joueur;
    private int lig;
    private int col;
    private int depl;
    private boolean shogun = false;

    // méthodes

    ...
}
```

0	1
2	1
3	1
4	3
false	false

Deux instances  
de Piece

*La classe Piece (simplifiée)*

```
// méthodes  
private int tirage()  
{ ... }  
  
public Piece(int j, int l, int c, boolean sh)  
{ ... }  
public int quelDeplacement()  
{ ... }  
public void deplacer(int nouvL,int nouvC)  
{ ... }  
public void afficher(...)  
{ ... }  
  
public String toString()  
{ ... }  
}// fin classe
```

*Inutile de connaître le corps  
des méthodes pour communiquer  
avec une instance de Piece*

Ex: l'objet *jeu* demande à la pièce *p* de se déplacer en case (4,3)

`p.deplacer(4, 3)`

**jeu n'a pas besoin de savoir comment p procède**

D'ailleurs, *jeu* n'a pas le droit de modifier directement *p* :

~~`p.lig = 4;`~~  
~~`p.col = 3;`~~

car col et lig sont encapsulés

**C'est p qui modifie son propre état**

les objets communiquent par **envoi de message** :

`p.deplacer(4,3)`

jeu *envoie à p le message "deplacer(4,3)"*

**jeu est l'expéditeur du message**

**(on est dans une méthode de jeu)**

**p est le receveur du message**

```
public Piece(int j, int l, int c,boolean sh)  
{ joueur = j;  
    lig = l;  
    col = c;  
    depl = tirage();  
    shogun = sh; }
```

## Constructeur

- méthode qui sert à contrôler l'initialisation *automatique* d'un objet lors de sa création
- porte le nom de la classe, pas de type de retour

Et si la classe ne définissait aucun constructeur?

→Constructeur par défaut (sans paramètre, corps vide) qui n'existe plus dès qu'un autre est défini

## Commençons par savoir nommer les objets

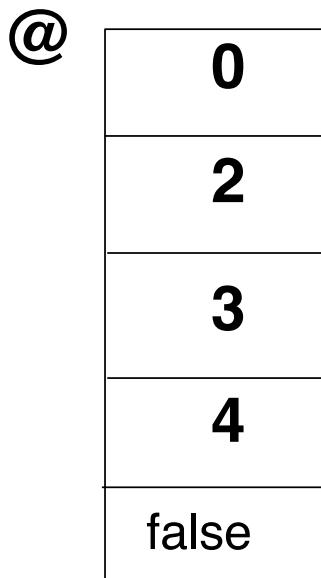
Il existe des variables appelées **références**, qui servent à désigner des objets

```
Piece p1;  
// déclaration d'une référence
```

```
Piece p1 = null;  
// déclaration + initialisation
```

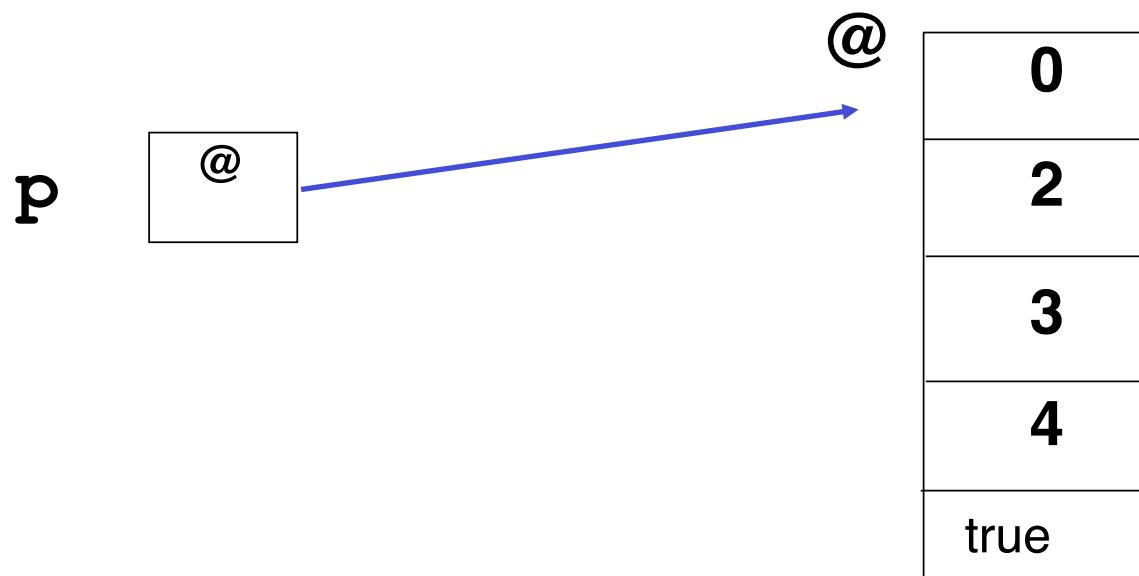
```
new Piece(0,2,3,false)
```

Création d'un objet



1. Crée une instance de la classe Piece  
*réservation d'une place en mémoire*
2. Appelle un constructeur sur cet objet  
*Initialisation de l'objet*
3. Retourne l'adresse @ de l'objet ...  
*... reste à récupérer cette adresse*

```
Piece p;  
p = new Piece (0,2,3, true);
```



## Objets et références ne sont pas liés "à vie"

Un objet peut avoir plusieurs noms, aucun nom, ou changer de nom

```
Piece p1 = new Piece (0,2,3,true);  
Piece p2 = p1;  
  
p1 = new Piece(0,1,1,false);  
  
p2 = null;
```

# Structure d'une application Java

un ensemble de classes,  
dont l'une comporte une méthode main

```
public class Hello
{
    public static void main (String args[])
    {
        System.out.println("Bonjour, monde!");
    }
}
```

```
public class MonAppliTest
{
    public static void main (String args[])
    {
        Piece p = new Piece(0,1,1,false);
        String etat = p.toString();
        System.out.println("état de p :" + etat);
        int d = p.quelDeplacement();
        p.deplace(1+d,1);
        etat = p.toString();
        System.out.println("état de p :" + etat);
    }
}
```

# this

Dans une méthode, **this** référence l'objet receveur

```
public Piece(int joueur, int lig, int col,  
            boolean shogun)  
{ this.joueur = joueur;  
  this.lig = lig;  
  this.col = col;  
  this.depl = tirage();  
  this.shogun = shogun; }
```

Convention : dans les constructeurs, on nomme chaque paramètre comme l'attribut qu'il initialise

# Accesseurs

Méthodes de manipulation et de contrôle des attributs

```
public int getJoueur() {return joueur;}
```

```
public boolean joueurCorrect(int j)  
{return (j==0 || j==1);}
```

```
public void setJoueur(int joueur) {  
if (joueurCorrect(joueur)) this.joueur = joueur;}
```

```
public boolean isShogun() {return shogun;}
```

# Conception par contrat

Java possède différentes techniques de mise en place de **contrats** pour les classes

- Les exceptions (un prochain cours)
- Les assertions

Les assertions permettent de vérifier des propriétés

Le programme est interrompu en cas de non respect d'une assertion  
Elles peuvent être inhibées et aident le débogage

# Assertions

## Syntaxe

```
assert conditionQuiDoitEtreVraie ;  
assert conditionQuiDoitEtreVraie : objet ;
```

## Exemple

```
assert (this.lig >= 0 && this.lig < 8) ;  
assert (this.lig >= 0 && this.lig < 8) : " ligne inexiste";
```

# Assertions

## Bonnes pratiques

- invariant d'algorithme
  - *à la fin de l'itération  $i$  dans la recherche du minimum la variable  $\min$  contient la plus petite valeur rencontrée entre 0 et  $i$*
- Invariant de flux
  - *Ce point de programme ne peut être atteint*
- Pré/posconditions
  - *À la fin d'une fonction de tri le tableau est trié*
- Invariants de classe
  - *Une voiture a 4 roues*

# Assertions

Mauvaises pratiques

Il ne faut pas ...

- vérifier les **paramètres** d'une méthode à l'aide d'assertions car ces paramètres viennent d'ailleurs et peuvent vraisemblablement être faux (donc ne pas sortir en échec, traiter plutôt le problème)
- vérifier les résultats **d'interactions** avec un utilisateur (vont souvent comporter des erreurs, à traiter)
- créer **des effets de bord** = de modification de l'état du programme avec les assertions (dans aucune des deux expressions)

# Assertions

Quelques fonctions utiles

```
public boolean joueurCorrect(int j)
{return (j==0 || j==1) ;}
```

```
public boolean ligneCorrecte(int l)
{return (l >= 0 && l < 8) ;}
```

```
public boolean colonneCorrecte(int c)
{return (c >= 0 && c < 8) ;}
```

# Assertions

```
public Piece(int j, int l, int c, boolean sh)
{
    if (joueurCorrect(j)) this.joueur = j;
    if (ligneCorrecte(l)) this.lig = l;
    if (colonneCorrecte(c)) this.col = c;
    this.shogun = sh;
    assert ((joueurCorrect(joueur)) &&
            ligneCorrecte(lig) && colonneCorrecte(col))
        : "mauvaise initialisation de la pièce j= »
          +joueur+" l="+lig+" c="+col;
    depl = tirage();
}
```

# Assertions

```
private int tirage() {  
    int t = ((int)(Math.random()*100%6));  
    assert (t >= 0 && t < 6):"problème de tirage t="+t;  
    return t;  
}
```

# Assertions

```
public void deplacer(int nouvL, int nouvC)
{
    // mettre à la position si elle est autorisée
    // on peut aller à angle droit de depl cases // puis faire nouveau tirage

    if (((nouvL==lig+depl && nouvC==col) ||
        (nouvL==lig && nouvC==col+depl) ||
        (nouvL==lig-depl && nouvC==col) ||
        (nouvL==lig && nouvC==col-depl)) &&
        (ligneCorrecte(nouvL)) &&
        (colonneCorrecte(nouvC)))
    {
        lig = nouvL; col = nouvC;
    }
    else
        System.out.println("Déplacement impossible");
    assert (ligneCorrecte(lig) && colonneCorrecte(col))
        : "nouvelle position incorrecte nouvL="+lig+" nouvC="+col;
    depl = tirage();
}
```

## Expression dont la valeur est un objet

```
class MessageAssertion
{ String m;
  public MessageAssertion(String m){this.m=m;}
  public String toString(){return m;}
}

public void setJoueur(int joueur) {
  if (joueurCorrect(joueur))
    this.joueur = joueur;
  assert (joueurCorrect(this.joueur))
    : new MessageAssertion("pb joueur j="+this.joueur);
}
```

# Assertions

```
public void setJoueur(int joueur) {  
    if (joueurCorrect(joueur)) this.joueur = joueur;  
    this.joueur = 9;  
    assert (joueurCorrect(this.joueur)) :  
        "Problème joueur j=" + joueur;  
}
```

Si un programmeur étourdi a ajouté **this.joueur = 9;**  
Cela provoquera une erreur du programme

Une **AssertionError**

# Assertions

Elles sont désactivées par défaut car elles ont un certain coût

Activation **-ea**

sur certaines classes ou packages

```
java -ea:shogun AppliTest
```

Désactivation

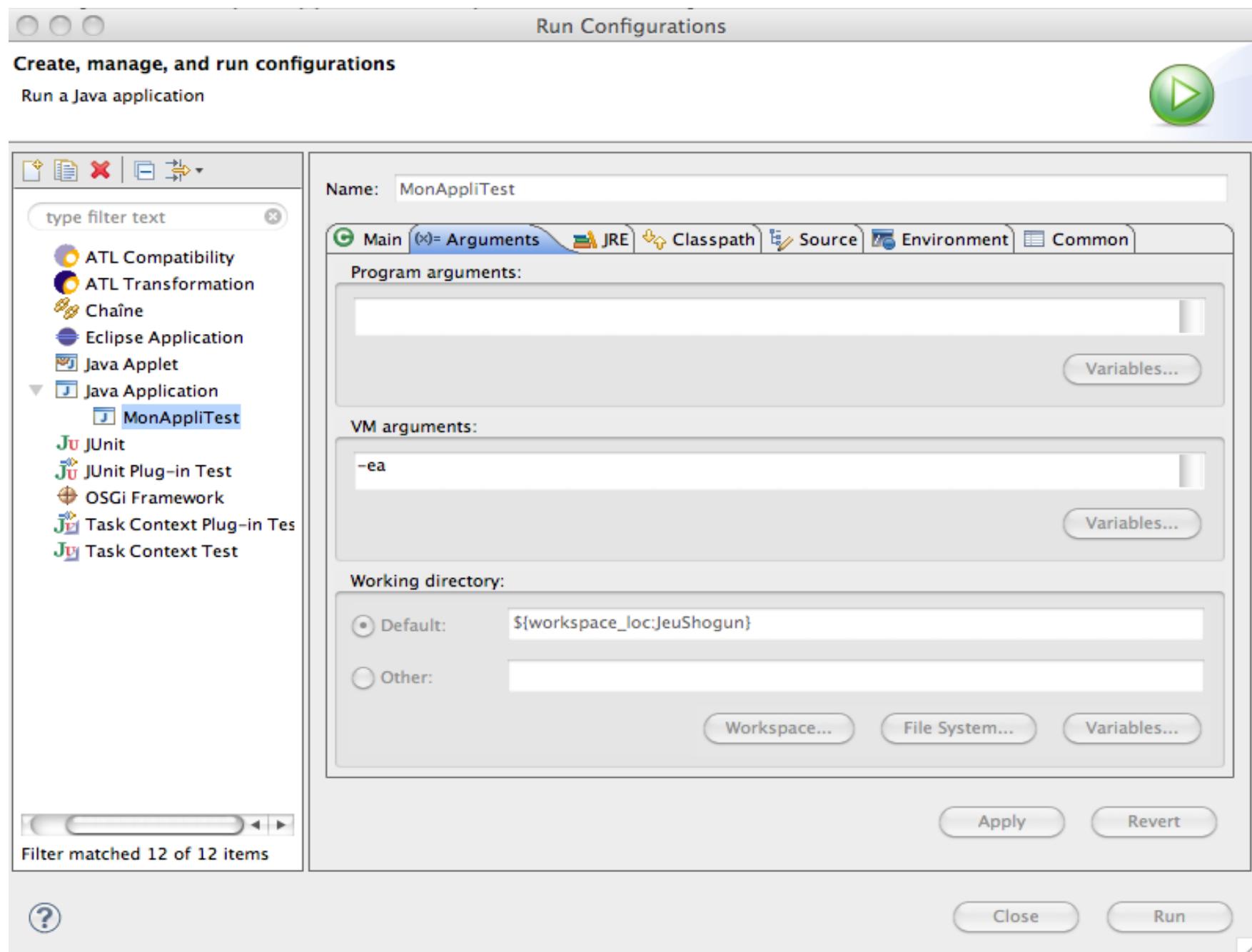
**-da**

Désactivation sur certaines classes ou packages

```
java -ea:shogun  
-da:shogun.Piece AppliTest
```

# Assertions

Pour les activer sous eclipse ..



# Assertions

```
Piece p = new Piece(0, 1, 1, false);
String etat = p.toString();
System.out.println("état de p :" + etat);
p.setJoueur(0);
System.out.println(p.getJoueur());
```

état de p :lig 1 col 1

Exception in thread "main"  
java.lang.AssertionError: pb joueur j=9  
at Shogun.Piece.setJoueur([Piece.java:92](#))  
at Shogun.MonAppliTest.main([MonAppliTest.java:32](#))

## Résumé

- principes généraux
- classes
  - *attributs, this*
  - *méthodes, constructeurs, accesseurs*
- assertions
  - *vérification de propriétés (contrats) lors de la mise au point du programme*

# **Spécialisation/généralisation**

## **Héritage**

## **Polymorphisme**

### **(partie 1)**

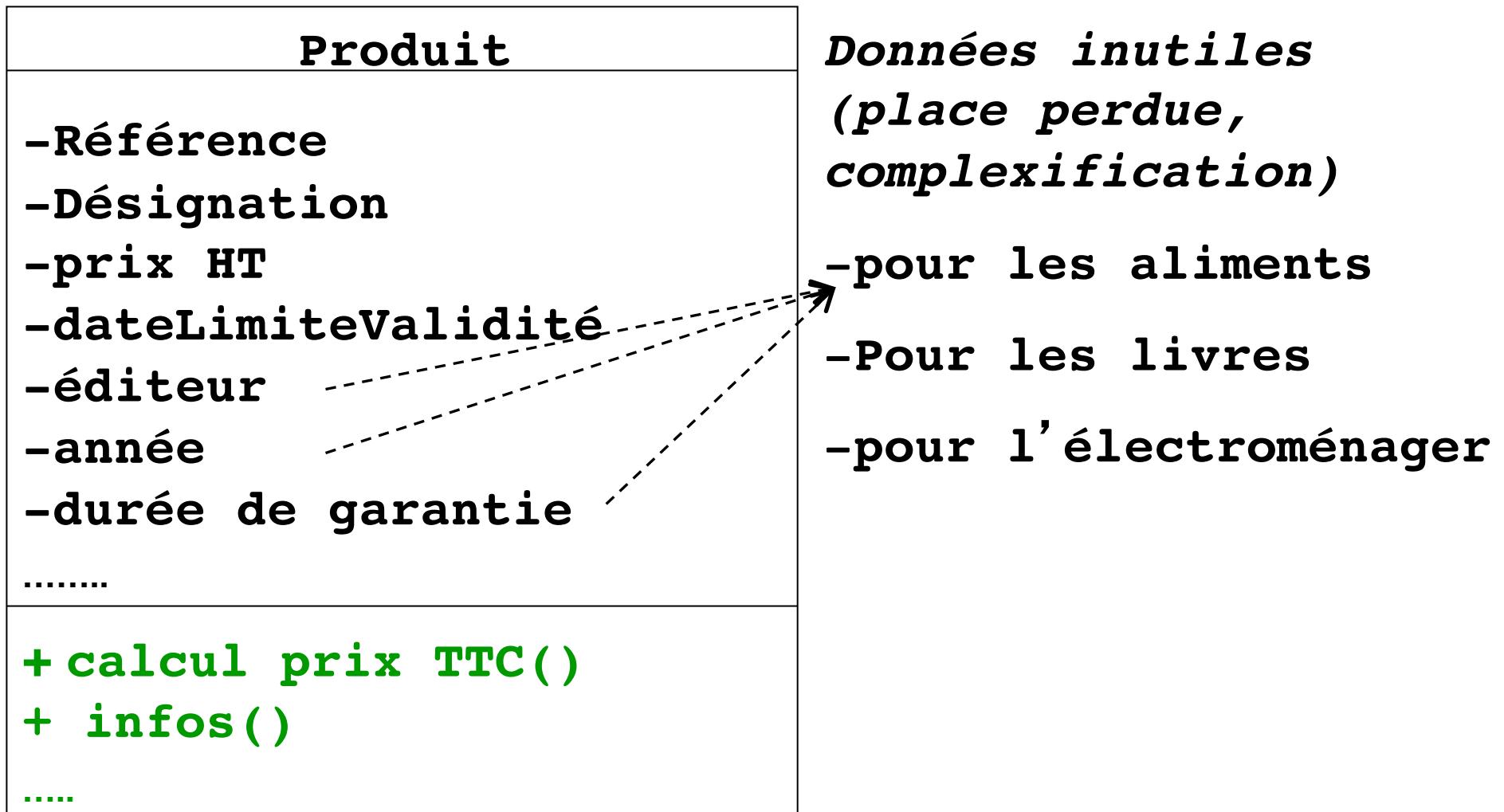
# Un exemple

- Représentation de produits
  - Livres, aliments, articles d' électroménager
- Données spécifiques
  - Livre : éditeur, année
  - Aliments : date limite de validité
  - Electroménager : garantie
- Calculs de prix spécifiques
  - Livre et Aliment : TVA 5,5%
  - Aliment : réduction lorsqu' on approche de la date de péremption
  - Electroménager : TVA 19,6%

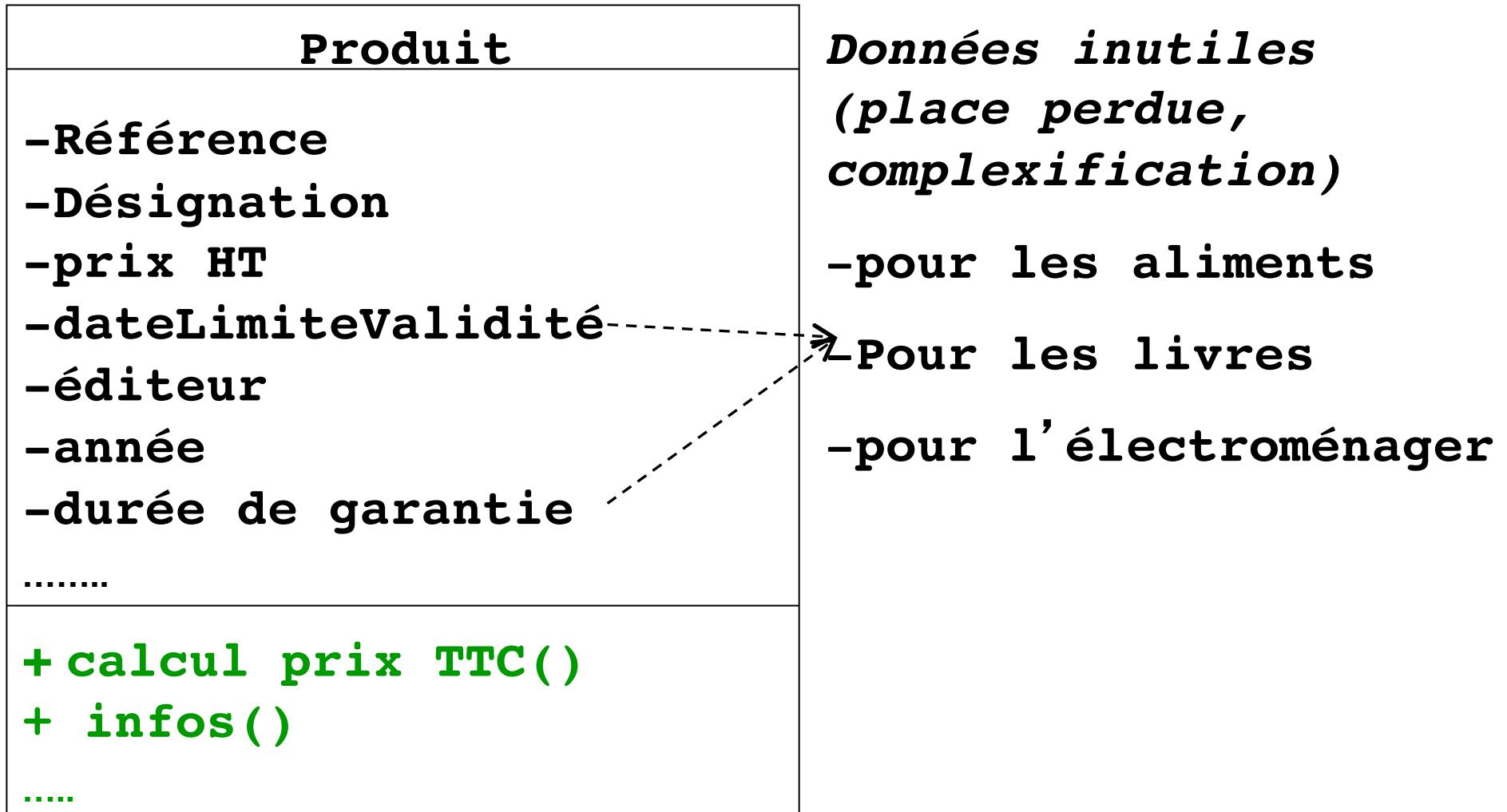
# Solution 1 : une seule classe pour représenter tous les produits

<b>Produit</b>
<b>-Référence</b>
<b>-Désignation</b>
<b>-prix HT</b>
<b>-date limite de validité</b>
<b>-éditeur</b>
<b>-année</b>
<b>-durée de garantie</b>
.....
<b>+ calcul prix TTC()</b>
<b>+ infos()</b>
.....

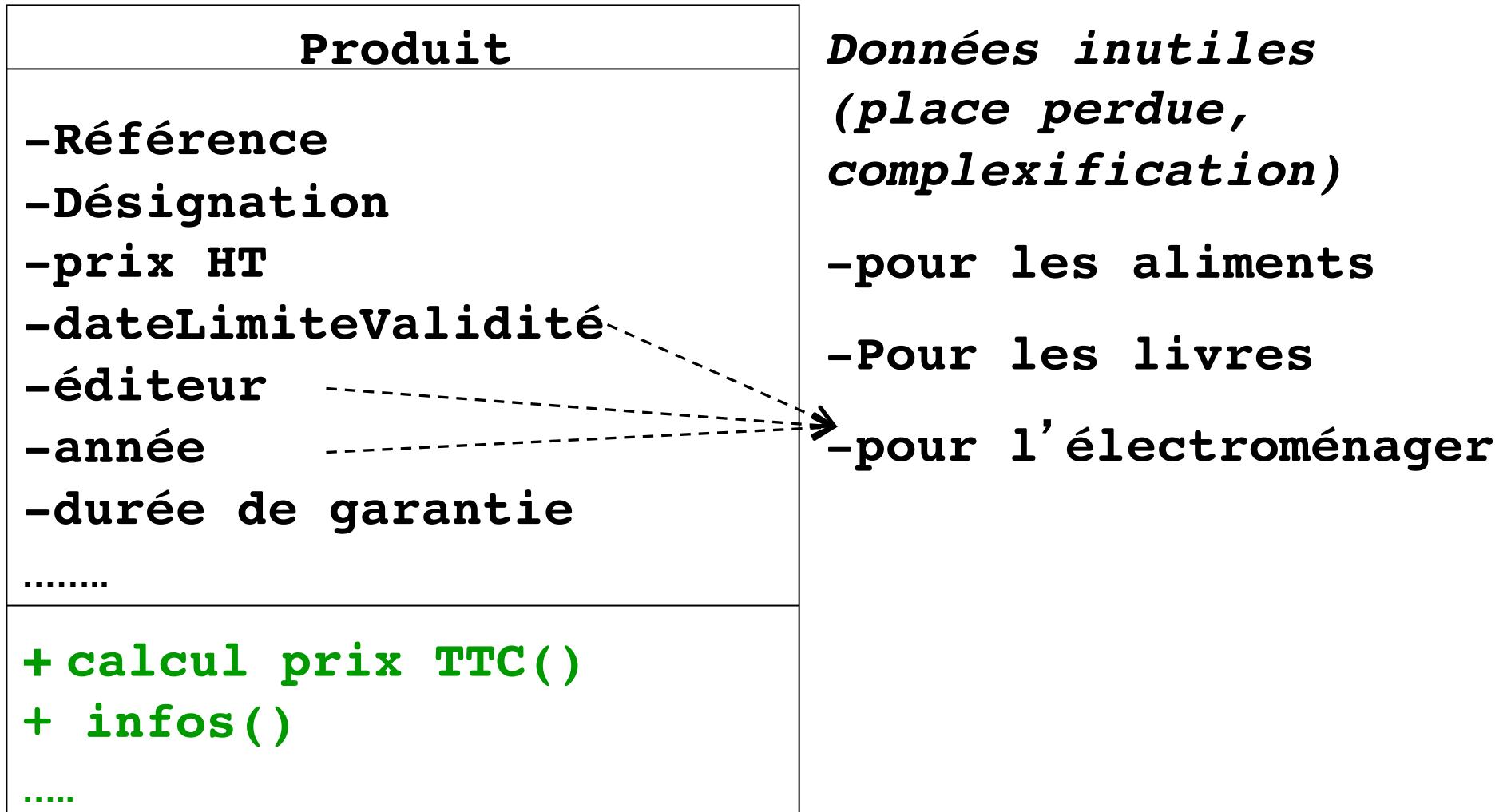
# Solution 1 : une seule classe pour représenter tous les produits



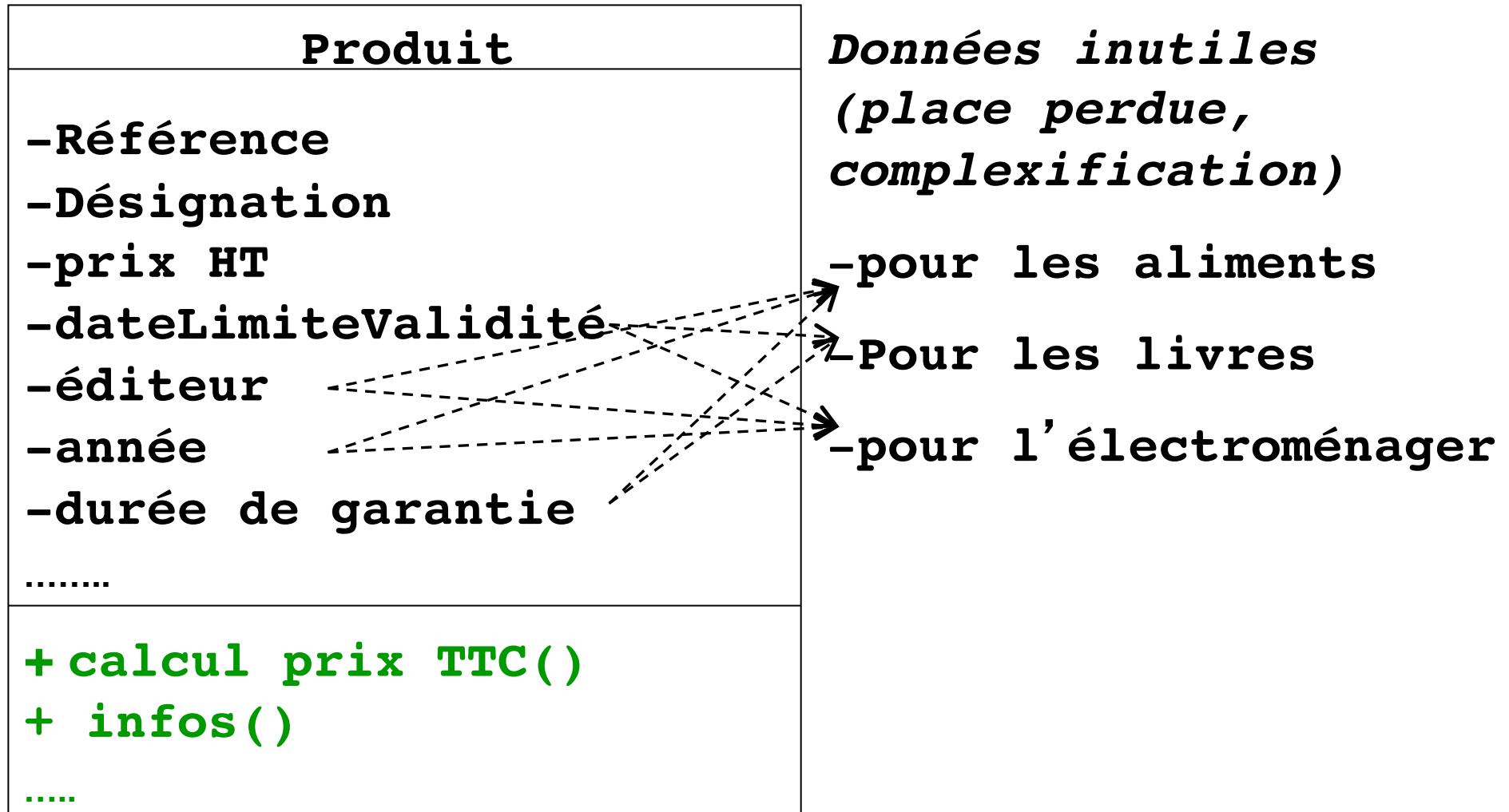
# Solution 1 : une seule classe pour représenter tous les produits



# Solution 1 : une seule classe pour représenter tous les produits



# Solution 1 : une seule classe pour représenter tous les produits



# Solution 1 : une seule classe pour représenter tous les produits

Produit
-Référence
-Désignation
-prix HT
-date limite de validité
-éditeur
-année
-durée de garantie
.....
+ calcul prix TTC()
+ infos()
.....

*Code complexe et non extensible dynamiquement à un nouveau type de produit*

**double prixTTC()**

**si(livre ou aliment) { ... }**  
**si(aliment){ ... }**  
**si (électro.){ ... }**

**String infos()**

**Réf + désignation**  
**si(livre) { ... éditeur/an }**  
**si(aliment){ ... date limite}**  
**si (électro.){ ... garantie}**

## Solution 2 : une classe par produit

Référence

Désignation

prix HT

date limite

**prix TTC()** 5,5%

et moins cher près

date limite

**infos()**

Réf + désignation

date limite .....

Référence

Désignation

prix HT

éditeur

année

**prix TTC()** 5,5%

**infos()**

Réf + désignation

éditeur/an .....

Référence

Désignation

prix HT

durée de garantie

**prix TTC()** 19,6%

**infos()**

Réf + désignation

garantie .....

**Aliment**

**Livre**

**Electro**

Référence

Désignation

prix HT

date limite

**prix TTC()**

**infos()**

.....

Référence

Désignation

prix HT

éditeur

année

**prix TTC()**

**infos()**

.....

Référence

Désignation

prix HT

durée de  
garantie

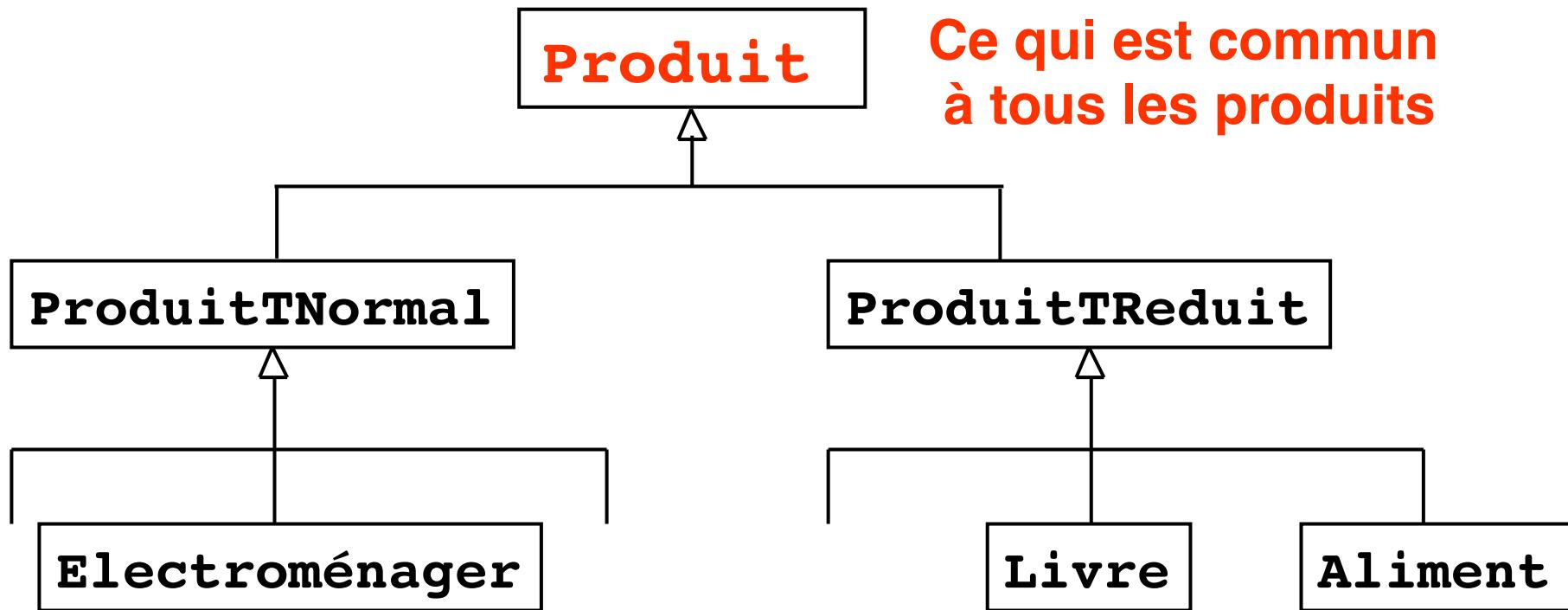
**prix TTC()**

**infos()**

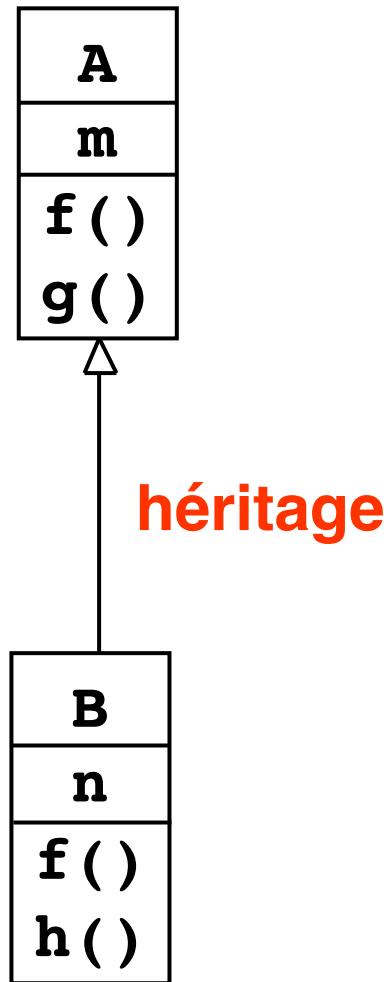
.....

Répétitions -> Factoriser attributs et méthodes !

## Solution 3 : organiser les classes en une hiérarchie d'héritage



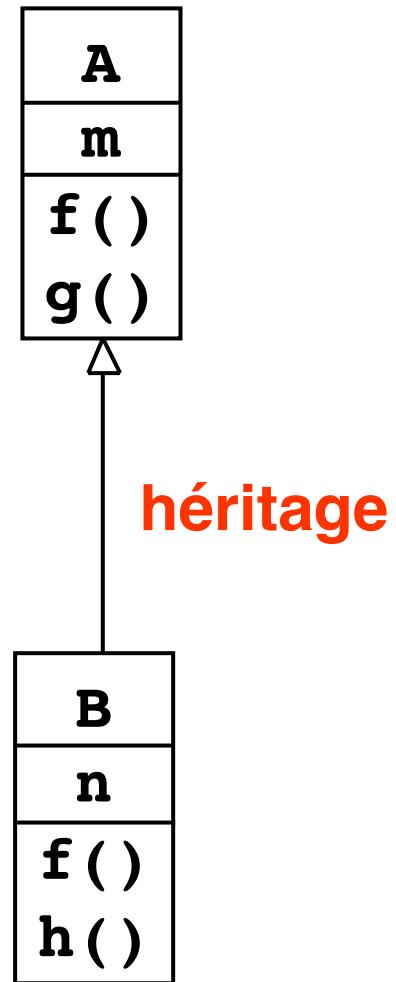
# Héritage



B construite à partir de A en

- ajoutant des attributs
- ajoutant des méthodes
- rédéfinissant des méthodes

- Une instance de B est composée des attributs m et n
- La redéfinition de f() dans B masque f() de A



```
B b = new B();
b.f();           // f de B
b.g();           // g de A
b.h();           // h de B
```

# La classe Produit

première approche

factorise ce qui est commun

```
public class Produit  
{ // attributs  
  
private String reference;  
private String designation;  
private double prixHT;  
  
// quelques méthodes  
  
public double leprixTTC(){ } //vide ..  
public String infos(){ ..}  
}
```

## Produit

-Référence  
-Désignation  
-prixHT

+ calcul prix TTC()  
+ infos()  
....

# La classe Produit

## première approche

Constructeur,  
Accesseurs  
aux attributs communs

```
public class Produit
```

```
{ // attributs
```

```
private String reference;
```

```
private String designation;
```

```
private double prixHT;
```

```
//constructeur
```

```
public Produit(String r, String d, double p)  
{reference=r; designation=d; prixHT=p;}
```

```
// quelques méthodes pour accéder aux attributs
```

```
public String getReference(){return reference;}
```

```
public void setReference(String r){reference=r;}
```

```
}
```

## Produit

-référence

-désignation

-prixHT

+ calcul prix TTC()

+ infos()

....

La classe Produit  
n'est pas instanciable !

La méthode leprixTTC()  
a un comportement inconnu

```
abstract public class Produit

{ // attributs

private String reference;

...
// quelques méthodes
.....
abstract public double leprixTTC();
public String infos(){...//ref+des+prix}

}
```

## Produit

-référence  
-résignation  
-prixHT

+calcul prix TTC()  
.....

- Calcul du prix TTC : méthode abstraite

```
public abstract double leprixTTC();
```

## Règles

- Toute classe possédant une méthode abstraite est abstraite
- Une classe peut être abstraite même si elle n'a pas de méthode abstraite

"Toute classe qui **possède** une méthode abstraite  
est abstraite"

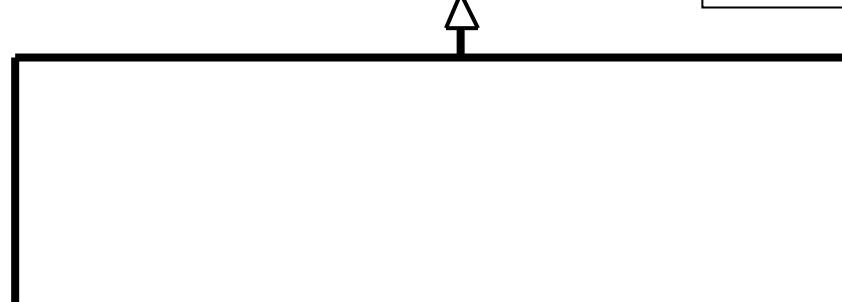


définit  
hérite de

A abstraite

Classe A

f() abstraite



Classe B

B abstraite

Classe C

f() concrète

Toujours dans la classe Produit ...

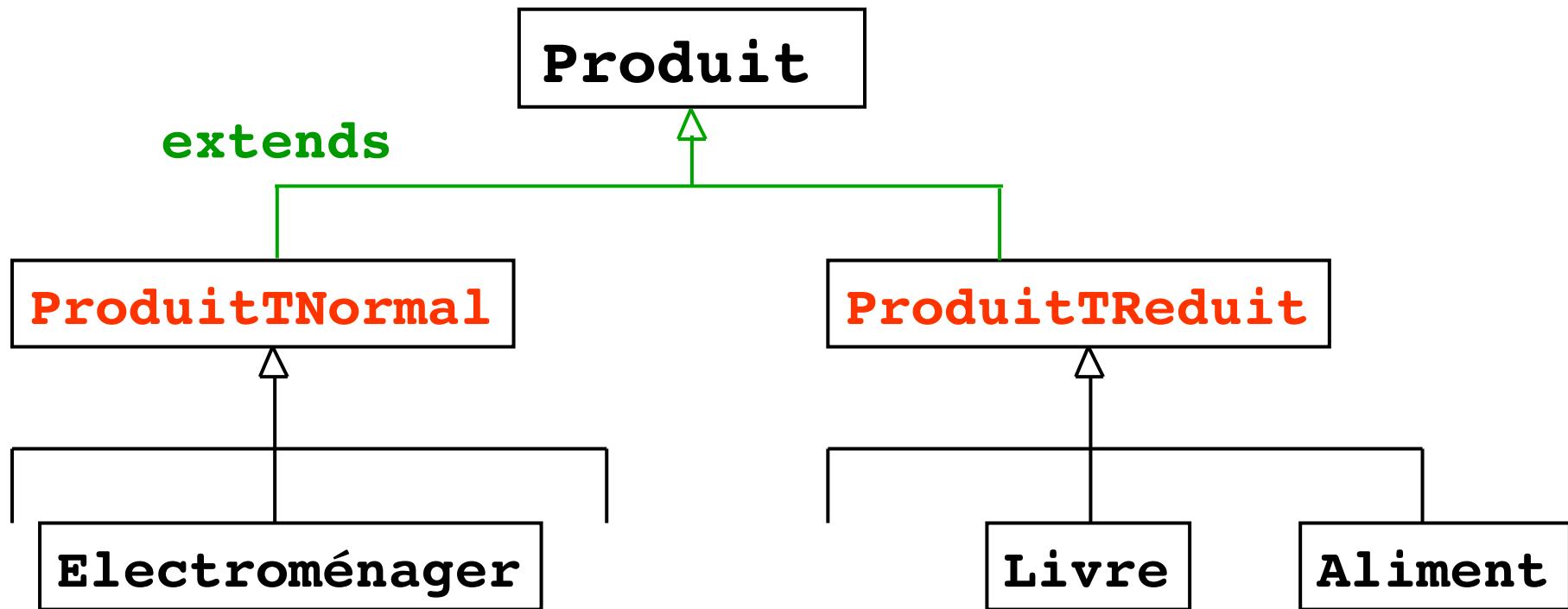
```
public String infos()
{
    String s = reference + ' ' + designation;
    s += '\n' + "prix HT: " + prixHT
    s += '\n' + "prix TTC: " + lePrixTTC();
    return s;
}
```

**lePrixTTC() est abstraite**

N'est-ce pas gênant ?

Non : on regarde plus loin ...

## Déclarer la relation d'héritage



# La classe ProduitTNormal

- Rôle : concrétiser la méthode **lePrixTTC()**,  
**version simple\***

$$\text{prix TTC} = \text{prix HT} + (\text{prixHT} * 19,6\%)$$

```
public double lePrixTTC()
{ return getPrixHT() * 1.196; }
```

- Faut-il des constructeurs ?  
on y répondra en regardant la branche d'héritage parallèle

\*Une version plus générale utiliserait une constante `TauxNormalTVA` (exercice)

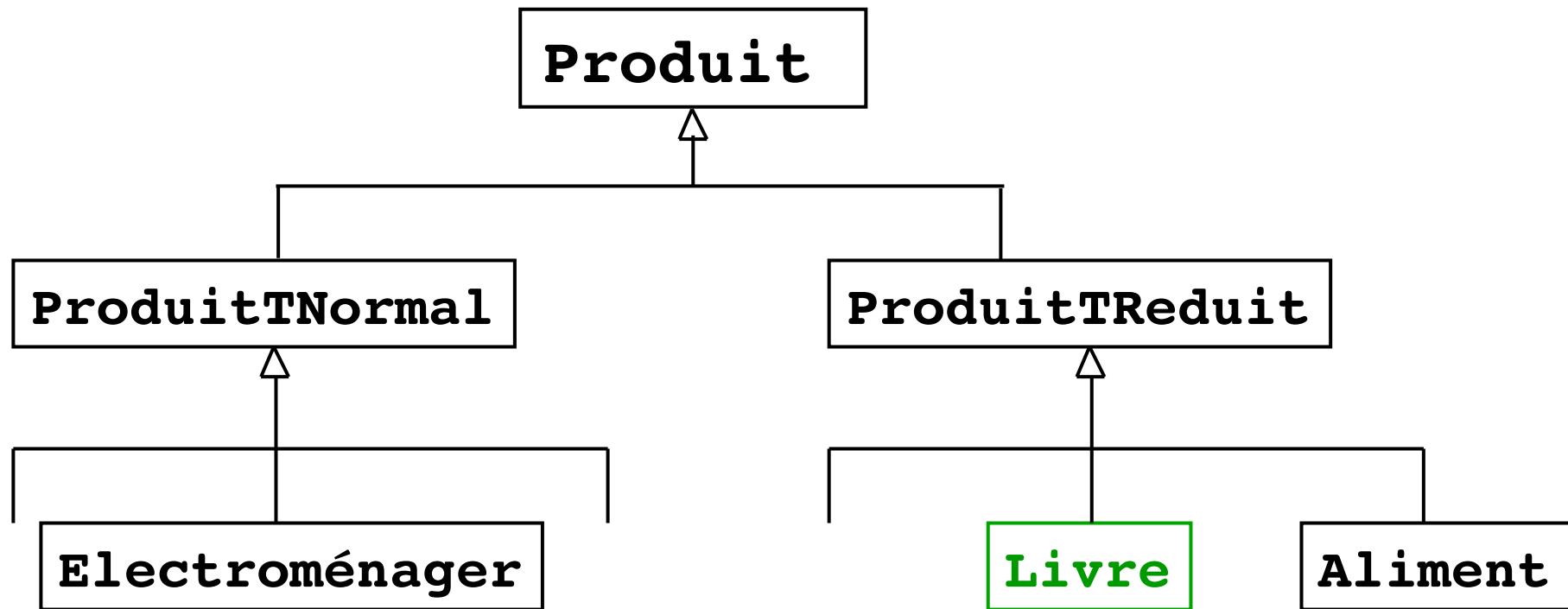
On récapitule :

- La déclaration d' héritage : **extends**
- La définition concrète de la méthode
- L' accès aux attributs grâce aux accesseurs

```
public class ProduitTNormal extends Produit
{
    ...
    public double lePrixTTC()
    {return getPrixHT() * 1.196; }

}
```

Descendons dans l'autre branche ...



# La classe Livre

- Une instance de Livre a **5 attributs**  
3 hérités de Produit (on ne les répète pas !)  
éditeur, année

```
public class Livre extends ProduitTReducit  
  
{//attributs  
private String editeur;  
private int annnee;  
  
//accesseurs  
  
public String getEditeur(){return editeur;}  
public void setEditeur(String e){editeur=e;}  
... idem pour annnee  
}
```

# La classe Livre

- Une instance de Livre a **5 attributs**  
3 hérités de Produit  
éditeur, année

Règle générale :

- chaque classe s'occupe des attributs qu'elle définit.
- pour les attributs hérités, elle délègue à ses super-classes

Appliquer à :  
constructeur  
méthode infos()

**La méthode infos() retourne :**

**le résultat de la méthode infos() héritée**

**+ présentation de éditeur et année**

```
public String infos()  
  
{ return super.infos()  
    + '\n' + éditeur + ' ' + année;  
}
```

super  
= accès à la méthode *masquée*

Classe A

```
String f() {return "fA";}  
String g() {return "gA";}
```

Classe B

```
String f() {return "fB " + super.f();}  
String h() {return "hB " + super.f();}  
String k() {return "kB " + super.g();}
```

Dans h() : **super.f()** bof - conception ?

Dans k() : **super.g()** NON - écrire **g()**

Règle : on utilise **super.f()** dans une nouvelle définition de **f()**

Classe A

```
String f() {return "fA";}  
String g() {return "gA";}  
String i() {return "i "+f();}
```

Classe B

```
String f() {return "fB " + super.f();}  
String h() {return "hB "+ super.f();}  
String k() {return "kB " + g();}
```

```
B b = new B();  
System.out.println(b.f()); fB fA  
System.out.println(b.g()); gA  
System.out.println(b.h()); hB fA  
System.out.println(b.k()); kB gA  
System.out.println(b.i()); i fB fA
```

On ne peut faire appel qu'à la méthode *masquée*

Classe A

```
void f()
```



Classe B

```
void f()
```



Classe C

```
void f() { super.f() ; }
```



f() de B

Aucun moyen d'appeler directement f() de A

# Constructeur de Livre

## Règle générale :

- chaque classe s'occupe des attributs qu'elle définit
- pour les attributs hérités, elle délègue à ses super-classes

## Constructeur de Livre :

- délègue à ses super-classes l'initialisation des attributs hérités
- initialise **éditeur** et **an**

**Mais ...**

**dans un constructeur, on ne peut appeler  
qu'un constructeur de la super-classe **directe****

**donc **ProduitTReduit** pour **Livre****

**Il faut donc un constructeur dans **ProduitTReduit****

**Qui ne sert que de "passeur"...**

## Produit

```
Produit(reference, designation, prixHT)
{initialise les attributs de même nom}
```

## ProduitTR

```
ProduitTR (reference, designation, prixHT)
{ passe les paramètres à Produit }
```

## Livre

```
Livre (reference, designation, prixHT,
       editeur, an)
{ - passe les 3 premiers paramètres
  à ProduitTR
  - initialise les attributs editeur et annee }
```

```
public Livre(String reference,  
            String designation, float prixHT,  
            String editeur, int an)  
{  
    super(reference, designation, prixHT);  
    this.editeur = editeur;  
    this.annee = an;  
}
```

```
public ProduitTReducit(String reference,  
                        String designation,  
                        float prixHT)  
{  
    super(reference, designation, prixHT);  
}
```

# Initialisation d'un Livre

```
Livre L = new Livre(r, d, p, e, a);
```

Appel de **Livre(r, d, p, e, a)**

Appel de **ProduitTReduit(r, d, p)**

Appel de **Produit(r,d,p)**

Init de **reference,  
designation, prixHT**

Init de **edition, annee**

L'appel **super(...)** est la première instruction du constructeur.

L'exécution d'un constructeur **commence toujours par un appel à un constructeur de la super-classe directe**

*(sauf pour Object qui n'a pas de super-classe)*

Ce peut être implicite : au besoin le compilateur insère l'appel

**super();**

// appel au constructeur sans paramètre  
// de la super-classe directe

## Exemple : classe Produit

```
public Produit(String reference,  
               String designation, float prixHT)  
{  
    super();  
  
    this.reference = reference;  
    this.designation = designation;  
    this.prixHT = prixHT;  
}
```

**super()** fait appel à quel constructeur? Celui de **Object**

Qui n'en définit pas...      Donc **constructeur par défaut**

# A retenir

- Déclarer une sous-classe : **extends**
  - Spécialiser une méthode : **super**
  - Spécialiser un constructeur : **super**
- 
- ✓ Héritage des attributs et des opérations
  - ✓ Liaison dynamique des méthodes
  - ✓ Appel des constructeurs

# **Spécialisation/généralisation**

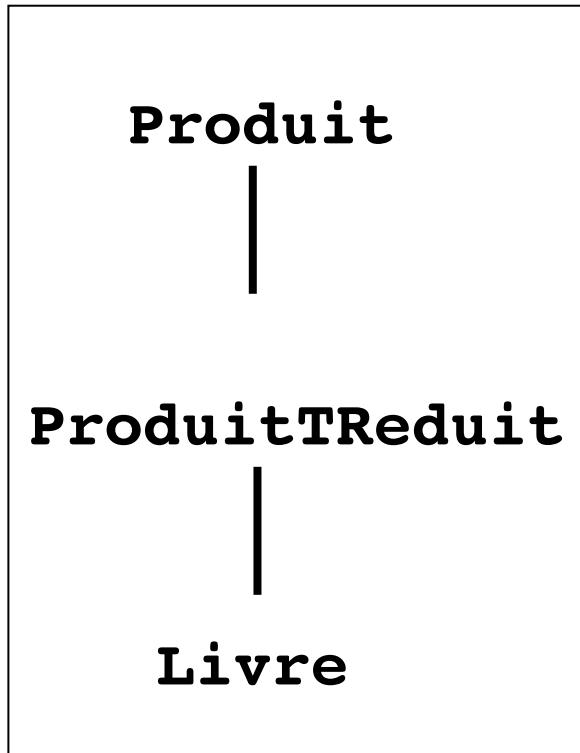
## **Héritage**

## **Polymorphisme**

### **(partie 2)**

# Héritage et Polymorphisme

# Idée 1 : dériver c'est spécialiser



"Un **produit à TVA réduite**, c'est un **produit (particulier)**,  
Un **livre**, c'est un **produit à TVA réduite**,  
c'est aussi un **produit**"

Partout où une instance de **Produit** convient,  
une instance de **ProduitTReduit** ou de **Livre**  
convient **aussi** : **substituabilité**

# Principe de substitution de Liskov

- Lorsqu'un type **S** est sous-type d'un type **T**

Tout objet de type **S** peut remplacer un objet de type **T** sans que le programme n'ait de comportement imprévu ou ne perde de propriétés souhaitées

- Application
  - S            Square
  - T            Rectangle

```
public class Rectangle {  
    private double width, height;  
  
    public Rectangle() {}  
    public Rectangle(double width, double height) {  
        this.width = width; this.height = height;  
        assert(this.getWidth() == width && this.getHeight() == height);}  
  
    public double getWidth() {return width;}  
    public void setWidth(double width) {  
        this.width = width; assert(this.getWidth() == width);}  
  
    public double getHeight() {return height;}  
    public void setHeight(double height) {  
        this.height = height; assert(this.getHeight() == height);}  
  
    public String toString()  
    {return "Rectangle : "+this.getWidth()+" "+this.getHeight();}  
}
```

```
public static void mystery(Rectangle r)
{
    r.setWidth(5.0);
    System.out.println(r);
    r.setHeight(4.0);
    System.out.println(r);
    assert(r.getWidth() * r.getHeight() == 20.0);
}
```

Attention :

- utiliser `r.` car nom du paramètre
- Méthode statique : `this` n'a pas de sens
- les attributs sont privés :  
on utilise les accesseurs

```
public class Square extends Rectangle {  
    public Square() {}  
    public Square(double width) {  
        super(width, width); assert(getWidth()==getHeight());}  
  
    public void setWidth(double width) {  
        super.setWidth(width); super.setHeight(width);  
        assert(getWidth()==getHeight()); assert(this.getWidth() == width); }  
  
    public void setHeight(double height) {  
        this.setWidth(height); }  
  
    public String toString()  
{return super.toString()+" Square : "+this.getWidth();}  
}
```

### Attention :

- les attributs sont privés :  
on utilise les accesseurs

```
public static void main(String[] args) {  
    Rectangle r = new Rectangle(8.0, 7.0);  
    System.out.println(r);  
    mystery(r); // se passe bien ...  
  
    r = new Square(3.0);  
    System.out.println(r);  
    mystery(r);  
        // la dernière instruction de mystery,  
        // r.setHeight()  
        // attribue 4 à la hauteur ET la largeur  
        // donc r.getWidth() * r.getHeight() == 16.0  
        // Déclenchement de l'AssertionError  
}
```

## Conclusion :

- Même si un square peut remplacer un rectangle dans le programme Java sans erreur de compilation
- il ne se comporte pas de la même manière
- Le principe de substitution de Liskov n'est pas vérifié en ce qui concerne la partie comportementale
- Justification par le programme main précédent où le comportement du programme est sain pour un rectangle et est altéré pour le carré.

## Règle n°1

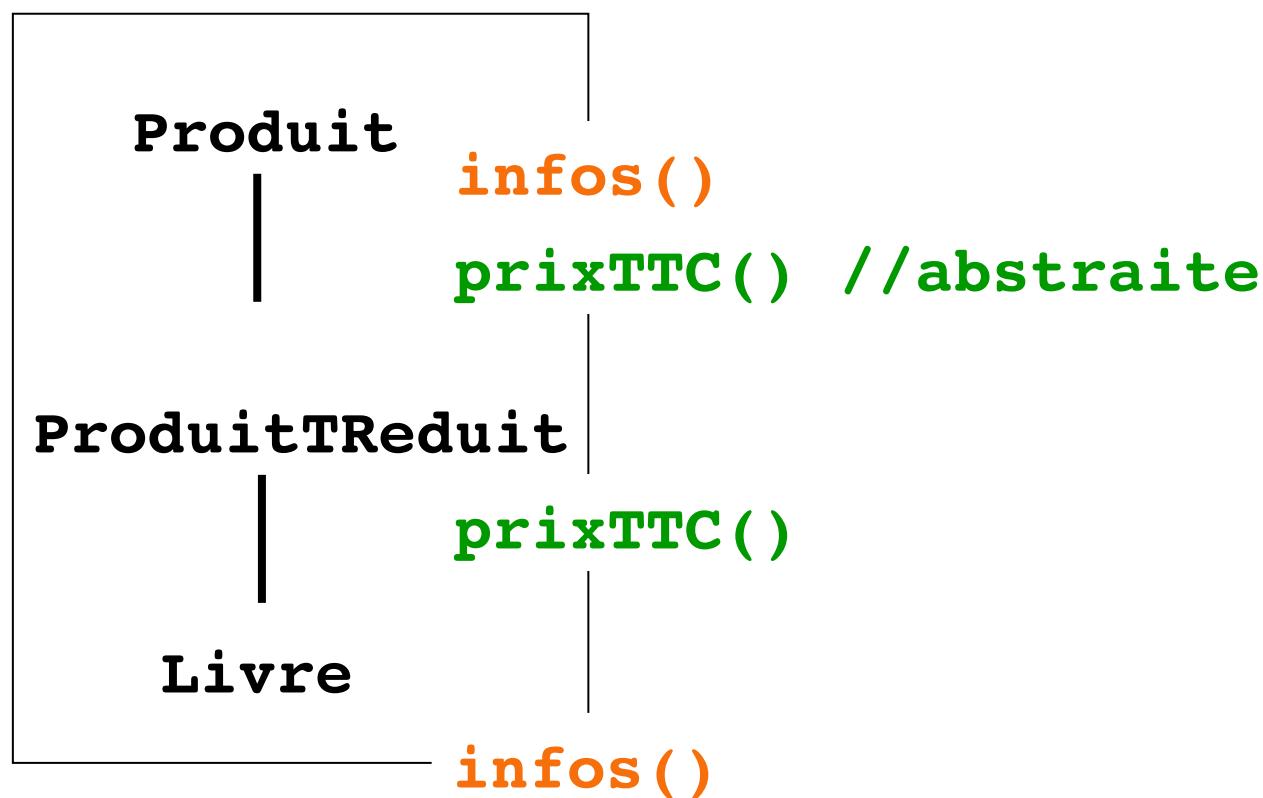
**Une référence de type A peut repérer une instance de A ou de toute classe dérivée de A**

```
Produit P = new Livre ( . . . );
```

```
Livre L = new Livre();
```

```
Produit P = L;
```

## Idée 2 : une redéfinition de méthode masque la méthode héritée



## Règle n°2

Lors d'un appel de méthode, c'est la **classe de l'objet** qui détermine quelle méthode exécuter et non pas le **type de la référence**

```
Produit P = new Livre (...);  
System.out.println(P.infos());  
  
// c'est infos() de Livre  
// qui doit être appelée
```

# Polymorphisme

Un **même** appel de méthode  
peut exécuter des corps de méthode **differents**  
selon la **classe** de l'objet appelant

```
String s;  
  
Produit P = new Livre (. . .);  
s = P.infos();           // infos() de Livre  
  
P = new Aliment(. . .);  
s = P.infos();           // infos() de Aliment
```

```
Produit P = new Livre(...);  
s = P.infos();
```

Quelles sont toutes les méthodes exécutées  
par cet appel ?

**Ce qui importe, ce n'est pas le type de la référence,  
mais la classe de l'objet référencé**

**Quand connaît-on l'objet référencé ?**

**Compilation ?**

**Exécution ?**

**À la compilation : ce serait plus efficace**

**Mais est-ce possible?**

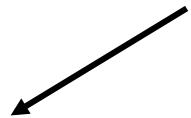
## Le choix ne peut pas se faire *toujours* à la compilation

```
Produit P;  
char choix;  
// demande à l'utilisateur : 'L' ou 'A' ?  
// affectation de choix  
  
if (choix == 'L')  
    {... P = new Livre (...); }  
  
else P = new Aliment(...);  
  
String s = P.infos();
```

Classe de l'objet référencé par P?

**Le choix de la méthode à exécuter se fait donc  
à l'exécution**

**Liaison dynamique**



**Liaison entre l'appel et le corps à exécuter**

Cf. C++ ou Pascal Objet : méthode **virtuelle**

[ Si c'était à la compilation :  
on parlerait de liaison **statique** ]

**En C++ ou Pascal Objet : méthode **non virtuelle** ]**

"statique", on en a déjà parlé ?

## Rapport avec méthodes statiques?

- Méthode de classe (**statique**) : pas d'objet receveur  
donc la liaison peut se faire à la compilation  
**liaison statique**
- Méthode d'instance : **liaison dynamique**

Liaison dynamique donc  
mais un **contrôle** est fait à la **compilation**

```
Object P = new Livre();  
  
float f = P.lePrixTTC();
```

Erreur de compilation :

La classe Object ne possède pas de méthode  
lePrixTTC()

```
Object P;  
char choix;  
// demande à l'utilisateur : 'L' ou 'A' ?  
// affectation de choix  
  
if (choix == 'L')  
    {... P = new Livre (...); }  
  
else P = new Aliment(...);  
  
String s = P.infos(); // ERREUR
```

Le compilateur ne sait pas quelle méthode infos() sera exécutée

mais il s'assure qu'il y en aura une

# Combien coûte mon caddie?

```
Produit [] monCaddie = new Produit [50];  
// remplissage du caddie  
  
// passage à la caisse  
  
float prixTotal = 0;  
  
for(int i = 0; i < nb ; i ++)  
  
{ prixTotal += monCaddie[i].lePrixTCC();  
}
```

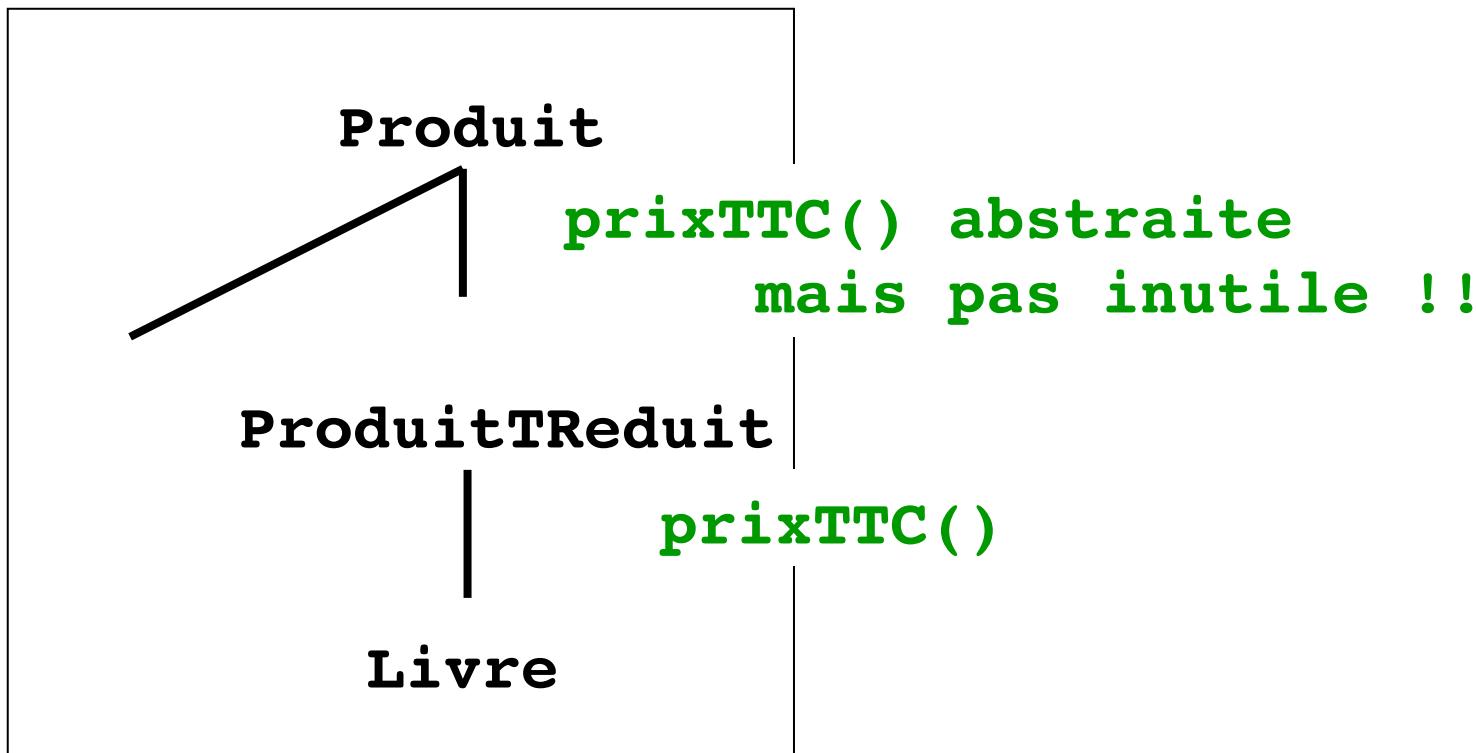


Référence de type Produit

Le compilateur vérifie que Produit possède une méthode lePrixTTC()

L'interpréteur choisit la bonne version de la méthode lePrixTTC()

Il est donc important de **prévoir toutes**  
les méthodes applicables à tous les produits  
au niveau de la racine (classe Produit)



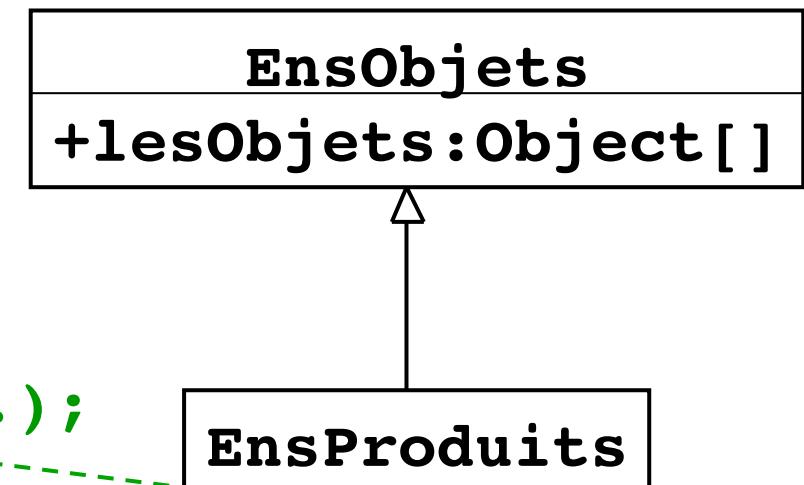
# Gestion d'un ensemble d'objets

```
Object [] lesObjets = new Object [50];
```

Gère un tableau de Object



Donc, n'importe quel objet



```
lesObjets[i]=new Produit(...);
```



## La classe EnsProduits

```
public String infos ()  
{  
    String S = "";  
    for (int i=0; i<lesObjets.length ; i++)  
    { S +=  
        lesObjets[i].infos()  
        + "\n\n";  
    }  
}
```

**Erreur de compilation  
(on suppose le tableau plein de produits):**

**Method `infos()` not found in `java.lang.Object`**

```
public Produit element (int i)
{
    return lesObjets[i-1];
}
```

**Erreur de compilation :**

**Incompatible type for return  
Explicit cast needed  
to convert java.lang.Object to Produit**

```
public Produit element (int i)  
{  
    return (Produit)lesObjets[i-1];  
}
```

La **coercition** de type est sans risque si **lesObjets** ne contient que des objets de la classe **Produit** ou de ses sous-classes

Pour le tester

```
if (lesObjets[i-1] instanceof Produit)  
    return (Produit)lesObjets[i-1];  
else return null;
```

Règle : le moins possible de coercition et de tests de type  
Ils révèlent souvent une mauvaise conception

# La coercition peut-elle permettre d'appeler la méthode *masquée* ?

Classe A

```
void f()
```

|

Classe B

```
void f()
```

|

Classe C

```
void f() { super.f(); }
```

Aucun moyen d'appeler directement f() de A

~~((A)super).f();~~

Que fait ((A)this).f(); ??

# Quelques précisions sur l'héritage

## Niveaux de protection

**private (privé)**: accessible seulement par les méthodes de **cette classe**

**rien (package)**: accessible seulement par les méthodes des classes du **même package**

**protected (protégé)**: accessible seulement par les méthodes des classes du **même package** et des classes dérivées

**public (public)**: accessible sans restriction

**Convention : pour l'instant les attributs sont privés (sauf éventuellement des constantes)**

## Peut-on "redéfinir" une méthode privée?

A    **private** f()



B    **public** f()  
    { ~~super.f()~~ . . . }

*super.f() impossible car f()  
privée en A*

**En fait c'est comme si B définissait une nouvelle  
méthode**

Ne pas confondre  
surcharge (overloading)  
et redéfinition (overriding)

A    public void f(int a)



surcharge

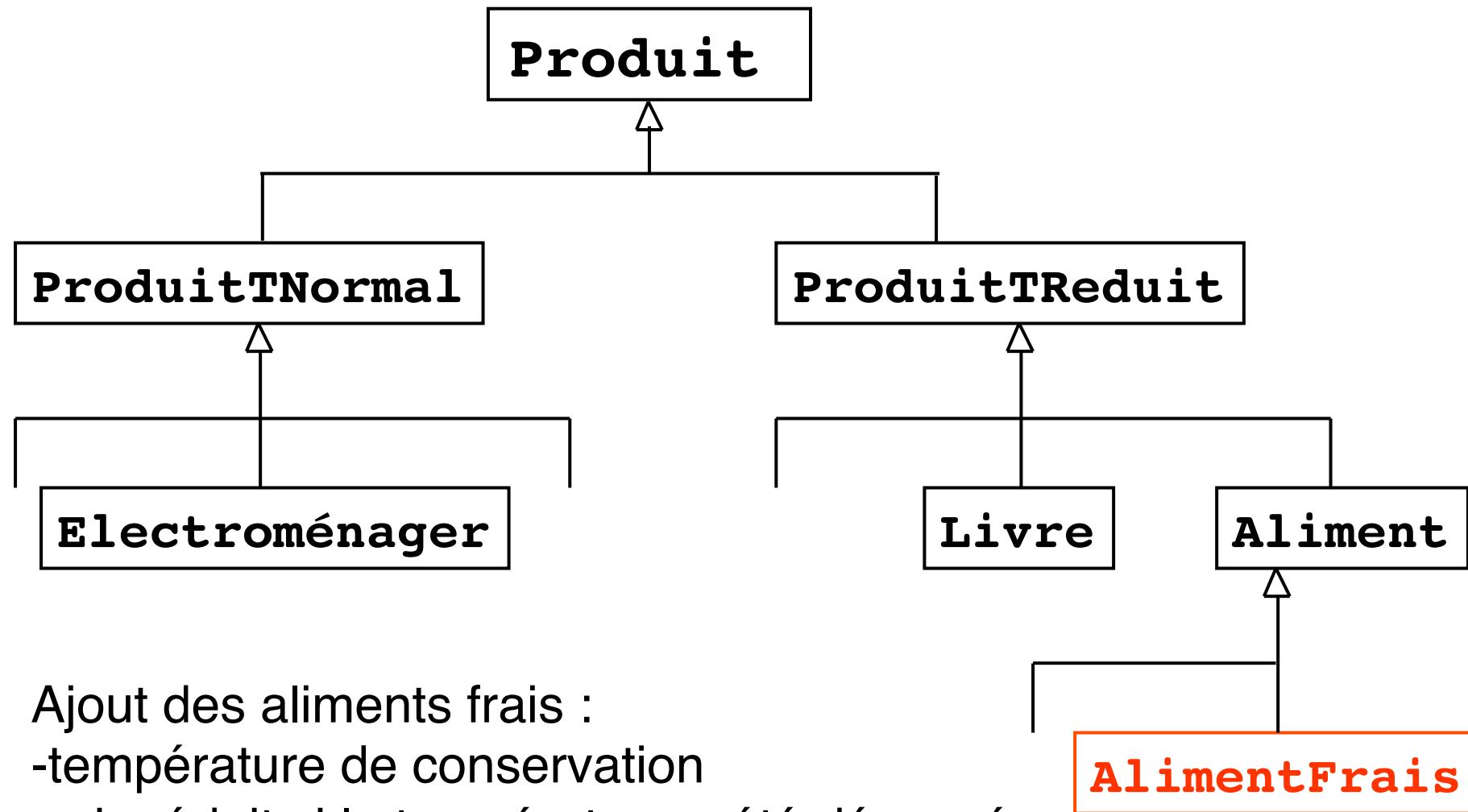
B    public void f(String b)

Une instance de B possède 2 méthodes f

```
B b = new B();  
b.f(5);      //f de A  
b.f("5");   // f de B
```

# Extensibilité

on complète en s' appuyant sur l' existant

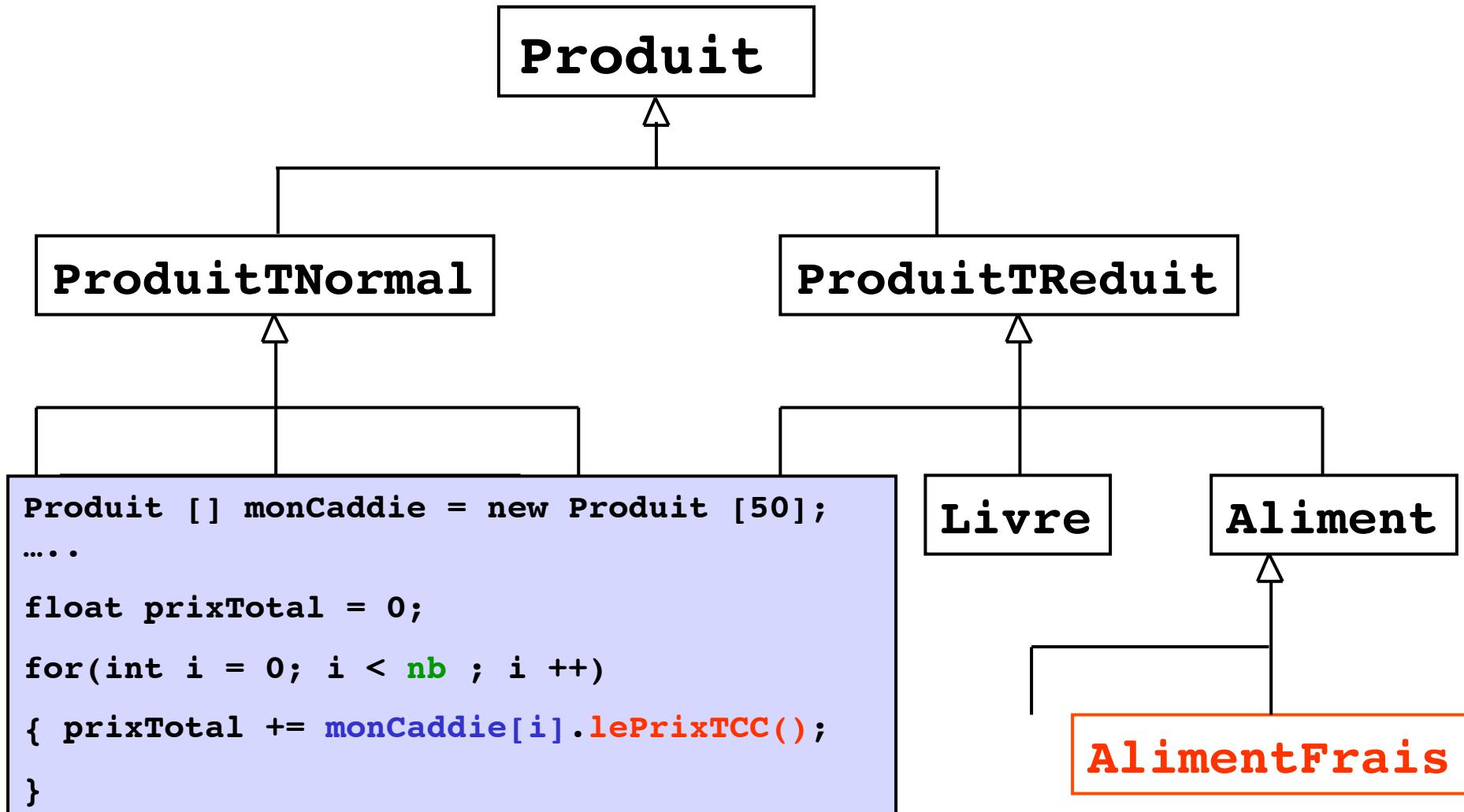


Ajout des aliments frais :

- température de conservation
- prix réduit si la température a été dépassée
- réduction de la date limite de validité

# Extensibilité

Les expressions précédemment écrites fonctionnent toujours !



# Rédéfinition de méthode

**Une redéfinition se fait**

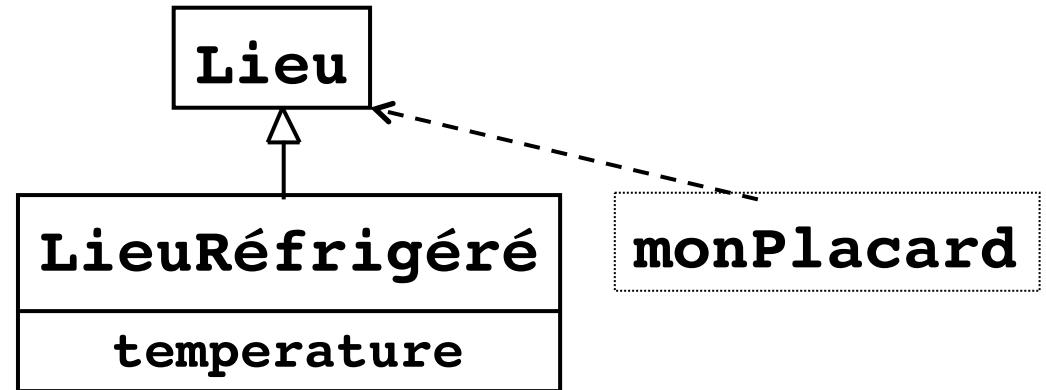
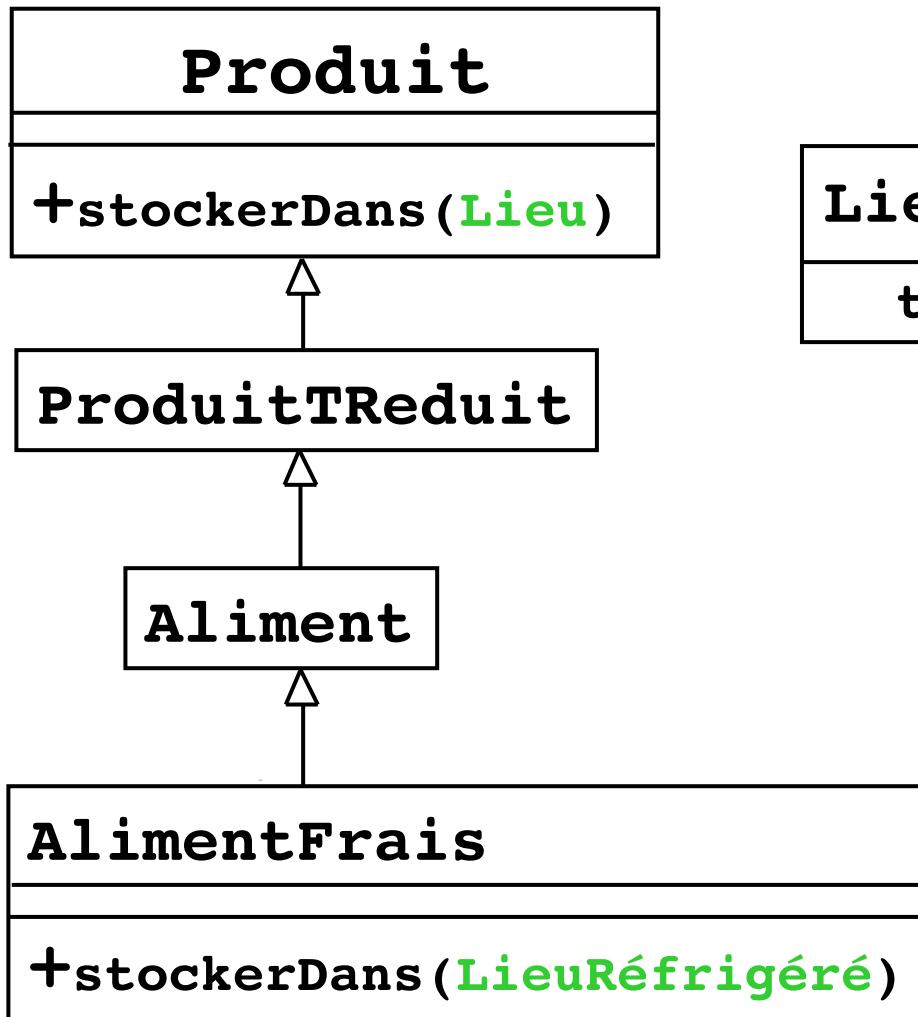
- sans changer la liste de paramètres
- le type de retour peut être spécialisé (Java 1.5)

La protection peut être affaiblie  
(package -> protégée -> publique)

**Ce sont des mesures pour garantir la substituabilité**

# Rédéfinition de méthode

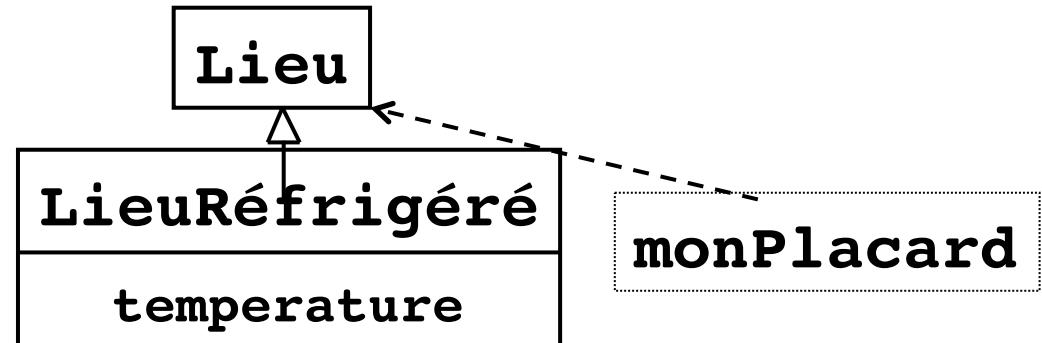
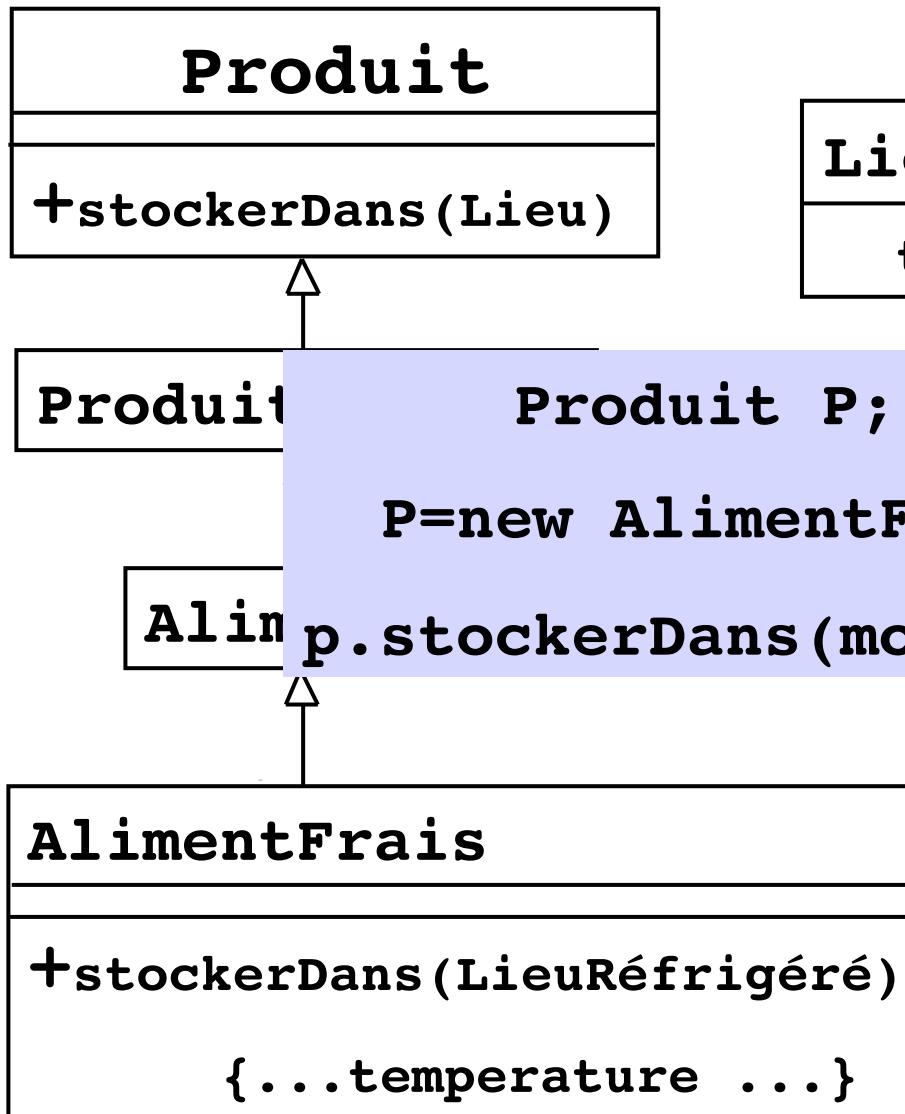
Un exemple de problème de substituabilité



Imaginons une  
redéfinition  
avec covariance  
des paramètres

# Rédéfinition de méthode

Un exemple de problème de substituabilité



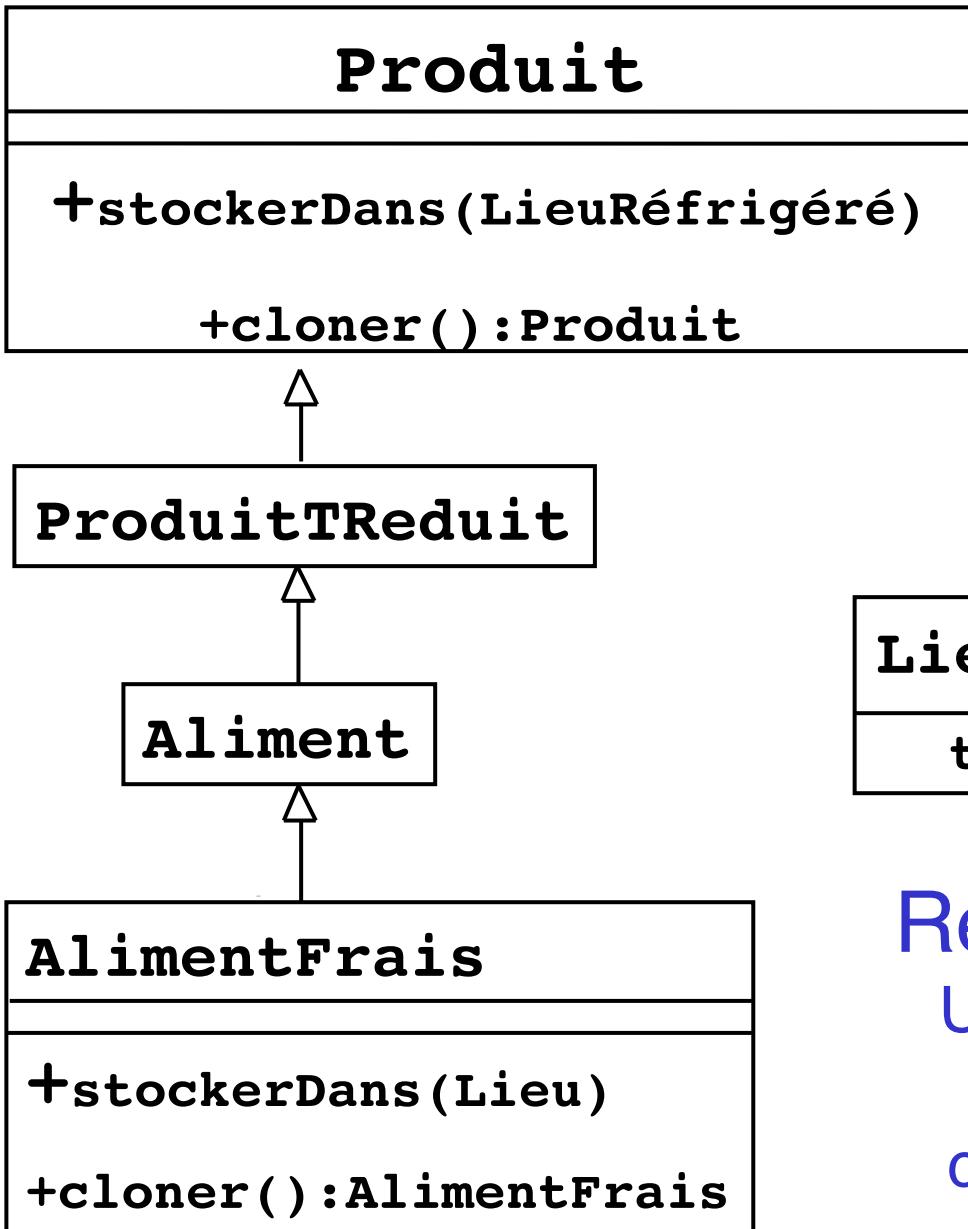
```
Produit P; .....
P=new AlimentFrais(...);
Aline p.stockerDans(monPlacard);
```

Compile mais à l'exécution ... poserait problème par exemple si stocker dans un lieu réfrigéré fait accès à la température, inconnue en général pour les lieux

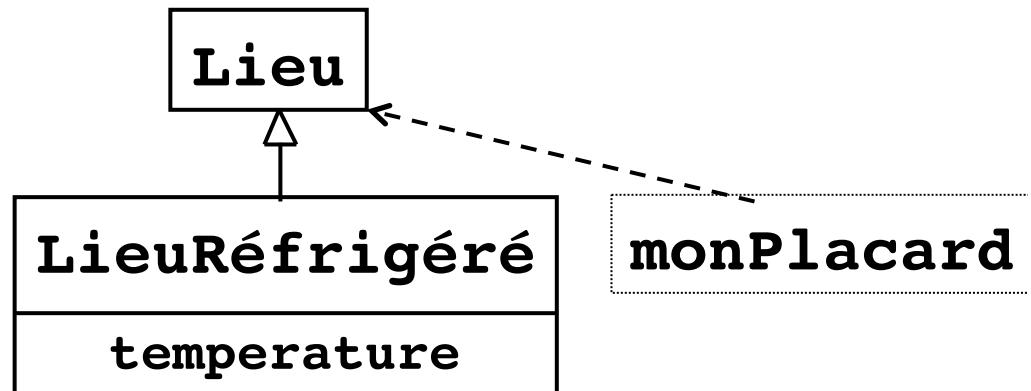
# Substituabilité

**La substituabilité demande**

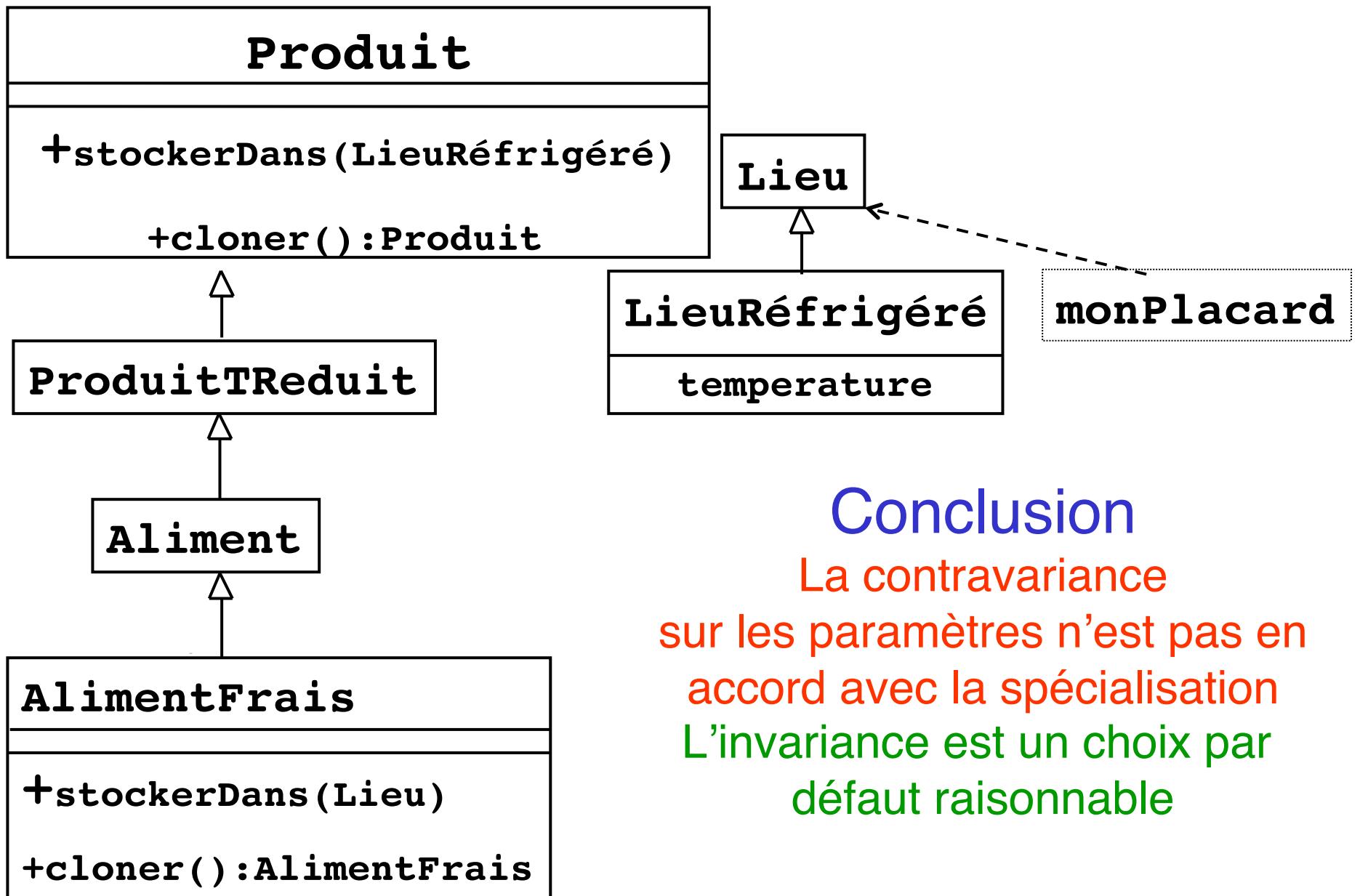
- **contravariance des paramètres**
  - les paramètres varient en sens inverse de la redéfinition
- **covariance du type de retour**
  - le type de retour varie dans le même sens que la redéfinition



Peut-on l'écrire  
En Java ?



Rédéfinition de méthode  
Un exemple de substituabilité  
avec  
covariance et contravariance



## Conclusion

La contravariance sur les paramètres n'est pas en accord avec la spécialisation  
L'invariance est un choix par défaut raisonnable

# Généricité Polymorphisme paramétrique

GLIN505  
Programmation par Objets 1  
Java

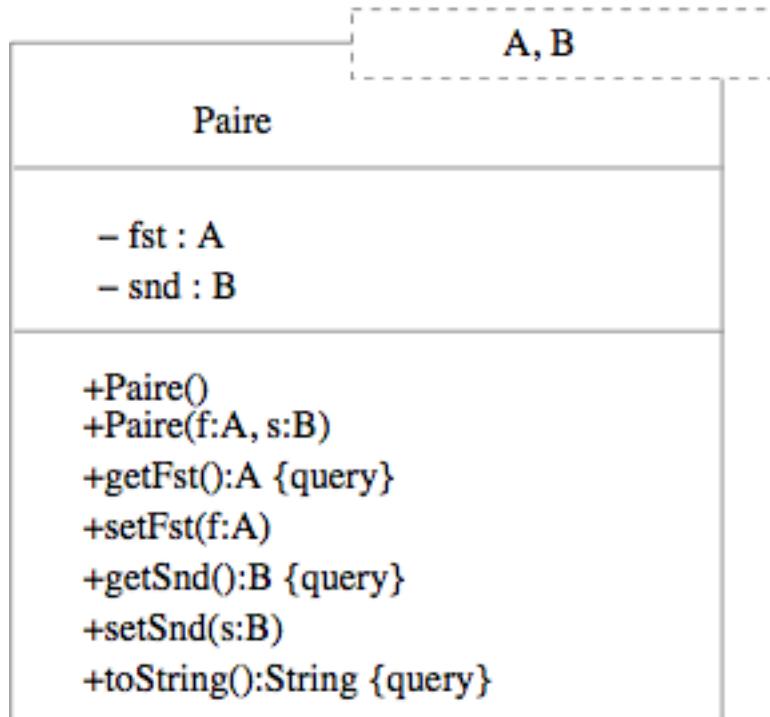
# Motivations

- Imaginons que l'on développe
  - Une Pile d'entiers,
  - Une Pile de String,
  - Une Pile d'Assiettes, etc.
- Comment ne pas écrire plusieurs fois des codes approchants, différent seulement par le traitement des types des valeurs empilées

# Une définition de la générnicité paramétrique

- Le polymorphisme paramétrique (ou générnicité) autorise la définition d'algorithmes et de types complexes (classes, interfaces) paramétrés par des types : `int` serait un paramètre de `Pile`
- Présence dans les langages :
  - de programmation : Java (>1.5), Eiffel, Ada, C+, Haskell, etc.
  - de modélisation : UML

Un type **Paire** paramétré par  
le type du premier élément (fst)  
et le type du second élément (snd)



*Modèles de classes*

Paire< A-> Integer, B -> String >

*Classes*

Représentation en UML

## Pour se convaincre : Que ferait-on sans ?

- soit une unique copie du code utilisant un type universel, **Object** en Java
  - traduction homogène
- soit une copie spécialisée du code pour chaque situation  
(int,int), (int, String), (int, Piece), etc.
  - traduction hétérogène

# Représentation homogène

```
public class Paire{  
    private Object fst, snd;  
    public Paire(Object f, Object s){fst=f; snd=s;}  
    public Object getFst(){return fst;}  
    ....}
```

L'utilisation demande de la **coercition (typecast)**

```
Paire p1 = new Paire("Paques",27);  
String p1fst = (String)p1.getFst();
```

# Représentation homogène

## Inconvénients

- la coercition n'est vérifiée qu'à l'exécution  
on peut seulement vérifier par un `instanceof`  
ou récupérer l'exception `ClassCastException`

```
Paire p1 = new Paire("jour",27);
if ((p1.getFst() instanceof String)
    String p1fst = (String)p1.getFst();
```

- le code est alourdi, plus difficile à comprendre et à mettre à jour
- la vérification est coûteuse à l'exécution

# Représentation hétérogène

```
public class PaireStringString{  
    private String fst, snd;  
    public Paire(String f, String s){fst=f; snd=s;}  
    public String getFst(){return fst;}....  
}
```

```
public class PaireintString{  
    private int fst, private String snd;  
    public Paire(int f, String s){fst=f; snd=s;}  
    public int getFst(){return fst;}....  
}
```

# Représentation hétérogène

## Inconvénients

- **duplication** excessive de code qui est source potentielle d'erreur lors de l'écriture ou de la modification du programme
- nécessité de **prévoir** toutes les combinaisons possibles de paramètres pour un programme donné

# Pour résumer

## Objectifs du polymorphisme paramétrique

- éviter des **duplications** de code ;
- éviter des ***typecast*** et des contrôles **dynamiques**
- effectuer des contrôles à la compilation (**statiques**)
- faciliter l'écriture d'un code **générique** et **réutilisable**

# Historique de l' introduction en Java

- **1995** - Naissance de Java
- **1999** - Dépôt d'une JSR (Java Specification Request) par G. Bracha pour l'introduction des génériques en Java
- **Propositions** : Pizza, GJ, NextGen, MixGen, Virtual Types, Parameterized Types, PolyJ
- **2004** : Java 1.5 (Tiger) - JDK 5.0
  - Paramétrage des classes et des interfaces
  - L' API des collections devient générique

# Les préconisations de la jsr 014 (1)

- Style Java
  - paramétrage des classes et des interfaces
  - syntaxe proche de C++
  - simplicité
- Environnement logiciel
  - compatibilité ascendante
  - compatibilité avec l'API, y compris les exceptions
  - les génériques sont des types comme les autres  
*(first class types)*
  - l'introspection doit fonctionner

# Les préconisations de la jsr 014 (2)

- Tirer les leçons des études précédentes
  - Ada, Eiffel et Haskell proposent des contraintes sur les types passés en paramètres
- Environnement de développement
  - changer le compilateur est obligatoire,
  - la compilation séparée est préconisée,
  - les modifications doivent garantir l'efficacité de l'exécution,
  - la taille des fichiers de bytecodes doit rester raisonnable,
  - la sécurité doit être respectée.
- Elles sont à peu près respectées sauf
  - L'introspection, avec effet sur les coercitions
  - Les exceptions

# Le paramétrage des classes (et des interfaces)

- Une classe générique admet des paramètres formels qui sont des types
- Ces paramètres portent sur les attributs et méthodes d'instance
- Ils ne portent pas sur les attributs et méthodes de classe (static)

# La classe paramétrée Paire

```
public class Paire<A,B>
{
    private A fst;
    private B snd;
    public Paire(){}
    public Paire(A f, B s){fst=f; snd=s;}
    public A getFst(){return fst;}
    public B getSnd(){return snd;}
    public void setFst(A a){fst=a;}
    public void setSnd(B b){snd=b;}
    public String toString(){return getFst()+"-"+getSnd();}
}
```

# Instanciation/invocation

```
Paire<Integer,String> p =  
    new Paire<Integer,String>(9,"plus grand chiffre");  
Integer i=p.getFst(); // pas de typecast !  
String s=p.get_snd(); // pas de typecast !  
System.out.println(p);
```

```
Paire<String,Piece> pp =  
    new Paire<String,Piece>("une pièce", new Piece(...));
```

Mais **pas de paramétrage par un type primitif**, on ne peut écrire :

```
Paire<int,Piece> = new Paire<int,Piece>(9,new Piece(...));
```

# Paramétrage des méthodes d'instance

## Cas standard

paramétrage par les paramètres de la classe

```
public class Paire<A,B>
{
    private A fst;
    ...
    public A getFst(){return fst;}
    public void setFst(A a){fst=a;}
    ...
}
```

# Paramétrage des méthodes d'instance

## En cas de besoin

### paramétrage par des paramètres supplémentaires

- Comparaison des deux premières composantes de deux paires
  - la deuxième composante n'est pas forcément de même type.

```
public class Paire<A,B>
{
    ...
    public <C> boolean memeFst(Paire<A,C> p)
    {return p.getFst()==this.getFst();}
    ....
}
```

# Paramétrage des méthodes de classe paramétrage obligatoire

```
public class Paire<A,B>
{
    ...
    public static<X,Y> void copieFstTab
        (Paire<X,Y> p,
         X[] tableau, int i)
    {tableau[i]=p.getFst();}...
}
```

# Paramétrage des méthodes (une utilisation)

```
Paire<Integer,String> p5 = new  
    Paire<Integer,String>(9,"plus grand chiffre");  
Integer[] tab=new Integer[2];  
Paire.copieFstTab(p5,tab,0);
```

```
Paire<Integer,Integer> p2 = new  
    Paire<Integer,Integer>(9,10);  
System.out.println(p5.memeFst(p2));
```

# L'effacement de type

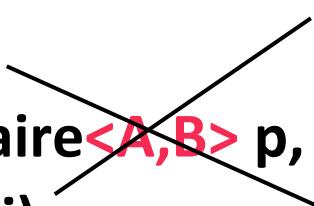
- Lors de la compilation, toutes les informations de type placées entre chevrons sont effacées
  - class Paire { ... }
- Les variables de types restantes sont remplacées par la borne supérieure (Object en l' absence de contraintes)
  - class Paire{private Object fst; private Object snd;..}
- Insertion de *typecast* si nécessaire (quand le code résultant n'est pas correctement typé)
  - Paire p = new Paire(9,"plus grand chiffre");  
Integer i=(Integer)p.getFst();

# L'effacement de type

Conséquences :

- A l'exécution, il n'existe en fait qu'une classe qui est partagée par toutes les instances
  - Testez : `p2.getClass() == p5.getClass()`
- Les variables de type paramétrant une classe ne portent pas sur les méthodes et variables statiques

```
public class Paire<A,B>
{
    public static void copieFstTab(Paire<A,B> p, A[] tableau,
                                    int i)
    {
        tableau[i] = p.getFst();
    }
}
```



# L'effacement de type

Conséquences :

- Une variable statique n'existe qu'en un exemplaire (et pas en autant d'exemplaires que d'instanciations)
  - `class Paire<A,B>{  
 static Integer nbInstances=0;  
 public Paire(..){... nbInstances++;} ...}`
  - `Paire<Integer, String> p = new Paire ...`  
`Paire<String, String> p2 = new Paire ...`
  - `Paire.nbInstances` vaut 2 !
- Pas d'utilisation dans le contexte de vérification de type `instanceOf` ou de coercition (*typecast*)
  - ~~(Paire<Integer, Integer>)p~~

# L'effacement de type

- Type brut (*raw type*) = le type paramétré sans ses paramètres

Paire p7=new Paire() fonctionne !

- Assure l'interopérabilité avec le code ancien (Java 1.4 et versions antérieures)
- **Attention** le compilateur ne fait pratiquement pas de vérification en cas de type brut

# Combinaisons de dérivations et d'instanciations

- Classe générique dérivée d'une classe non générique

```
class Graphe{}
```

```
class GrapheEtiquete<TypeEtiq> extends Graphe{}
```

- Classe générique dérivée d'une classe générique

```
class TableHash<TK,TV> extends Dictionnaire<TK,TV>{}
```

- Classe dérivée d'une instantiation d'une classe générique

```
class Agenda extends Dictionnaire<Date,String>{}
```

# Quelques exemples dans l'API des collections

- public interface Collection<E> extends Iterable<E>
- public class Vector<E> extends AbstractList<E>
- public class HashMap<K,V> extends AbstractMap<K,V>  
*K - type des clefs (Keys)*  
*V - type des valeurs (Values)*

# Quelques exemples dans l'API des collections

```
public class Stack<E> extends Vector<E>
{
    public Stack();
    public E push(E item);
    public E pop();
    public E peek();
    public boolean empty();

    ...
}
```

# Mariage Polymorphisme paramétrique / héritage

- Sous-typage des classes pour un paramètre fixé

Stack<String> est bien sous type de Vector<String>

Vector<String> pi=new Stack<String>();

# Mariage Polymorphisme paramétrique / héritage

- Pas de sous-typage basé sur celui des paramètres  
= pas de substitution possible

**String** sous-type d' **Object**

**Stack<String>** n'est pas un sous-type de **Stack<Object>**

- certaines opérations admises sur une **Pile<Object>**, telles que **empile(Object o)**, ne sont pas correctes pour une **Pile<String>**  
sauf si les types sont immuables

~~Stack<Object> pi=new Stack<String>();~~

# Le paramétrage constraint (ou borné)

- Pourquoi des contraintes sur les types passés en paramètres :
  - lorsque ceux-ci doivent fournir certains services (méthodes, attributs) ;
  - plus généralement, pour exprimer qu'ils correspondent à une certaine abstraction.

# Le paramétrage constraint (ou borné)

- Objectif : munir la classe Paire<A,B> d'une méthode de saisie
- Contrainte : les types A et B doivent disposer d'une méthode de saisie également
- La contrainte peut être une classe ou mieux une interface

public interface Saisissable

{

    public abstract void saisie(Scanner c);

}

# Le paramétrage contraint (ou borné)

```
class PaireSaisissable<A extends Saisissable, B extends Saisissable>
    implements Saisissable
{
    private A fst; private B snd;
    public PaireSaisissable(A f, B s){fst=f; snd=s;}
    public A getFst(){return fst;}
    public B getSnd(){return snd;}
    public void setFst(A a){fst=a;}
    public void setSnd(B b){snd=b;}
    public String toString(){return getFst()+"-"+getSnd();}
    public void saisie(Scanner c){
        System.out.print("Valeur first:"); fst.saisie(c);
        System.out.print("Valeur second:"); snd.saisie(c);}
}
```

# Le paramétrage contraint (ou borné)

- Un type concret qui répond à la contrainte

```
public class StringSaisissable implements Saisissable
{
```

```
    private String s;
```

```
    public StringSaisissable(String s){this.s=s;}
```

```
    public void saisie(Scanner c)
```

```
    {s=c.next();}
```

```
    public String toString(){return s;}
```

```
}
```

# Le paramétrage contraint (ou borné)

- Un programme

```
Scanner c = new Scanner(System.in);
```

```
StringSaisissable s1 = new StringSaisissable("");
```

```
StringSaisissable s2 = new StringSaisissable("");
```

```
PaireSaisissable<StringSaisissable, StringSaisissable> mp =
```

```
    new PaireSaisissable<StringSaisissable, StringSaisissable>(s1,s2);
```

```
mp.saisie(c);
```

# Le paramétrage constraint (ou borné)

- Contraintes multiples
  - Les paires sont saisissables et sérialisables

```
class Paire<A extends Saisissable & Serializable,  
          B extends Saisissable & Serializable>  
{.....}
```

# Le paramétrage contraint (ou borné)

- Contraintes récursives
  - un ensemble ordonné est paramétré par le type **T**
  - **T** = les éléments qui sont comparables avec des éléments du même type **T**

```
public interface Comparable<A>
{public abstract boolean infStrict(A a);}
```

```
public class orderedSet<A extends Comparable<A>>
{....}
```

# Le paramétrage par des jokers (*wildcards*)

- $\text{Paire}\langle\text{Object},\text{Object}\rangle$  n'est pas super-type de  $\text{Paire}\langle\text{Integer},\text{String}\rangle$ 
  - mais il existe quand même un super-type à toutes les instanciations d'une classe paramétrée
- Le super-type de toutes les instanciations
  - Caractère joker ?
  - $\text{Paire}\langle ?, ? \rangle$  super-type de  $\text{Paire}\langle\text{Integer}, \text{String}\rangle$

# Le paramétrage par des jokers

Utilisation pour le typage d'une variable

Mais ... tout n'est pas possible

```
Paire<?,?> p3 = new Paire<Integer, String>();
```

~~p3.setFst(12);~~ NON : setFst dépend du paramètre de type

```
System.out.println(p3); // oui : ne dépend pas du paramètre de type
```

# Le paramétrage par des jokers

Utilisation pour simplifier l'écriture du code

A et B ne sont pas utilisés dans la vérification de l'écriture suivante :

```
public static<A,B> void affiche(Paire<A,B> p)
{
    System.out.println(p.getFst()+" "+p.getSnd());
}
```

On peut donc les faire disparaître :

```
public static void affiche(Paire<?,?> p)
{
    System.out.println(p.getFst()+" "+p.getSnd());
}
```

# Le paramétrage par des jokers

Utilisation pour élargir le champ d'application des méthodes

Écrivons une méthode qui prend la valeur de la première composante d'une paire dans une liste (à la première position) :

```
public class Paire<A,B>{  
    public void prendListFst(List<A> c)  
    {setFst(c.get(0));} ....}
```

Utilisation :

```
Paire<Object,String> p6 = new Paire<Object,String>();  
List<Object> lo = new LinkedList<Object>();  
List<Integer> li = new LinkedList<Integer>();  
lo.add(new Integer(6)); li.add(new Integer(6));  
p6.prendListFst(lo); p6.prendListFst(li);
```

Pourtant il n'y a pas d'erreur sémantique :  
un Integer est une sorte d' object mais A=Object≠Integer

# Le paramétrage par des jokers

Pourtant il suffirait que le type des objets dans la liste c soit A ou un sous-type de A

```
public class Paire<A,B>{
    public void prendListFst(List<A> c)
    {setFst(c.get(0));} ....}
```

On réécrit (première possibilité)

```
public <X extends A> void prendListFst(List<X> c)
{setFst(c.get(0));}
```

Mais X ne sert à rien pour le compilateur (deuxième possibilité)

```
public void prendListFst(List<? extends A> c)
{setFst(c.get(0));}
```

# Le paramétrage par des jokers contrainte super

- `extends` --> borne supérieure pour le type
- `super` --> borne inférieure
- Utilisation : puits de données
- Exemple `copieFstColl`, qui écrit le premier composant d'une paire dans une collection

version initiale

```
public void copieFstColl(Collection<A> c)
{c.add(getFst());}
```

# Le paramétrage par des jokers contrainte super

- version initiale trop stricte

```
public void copieFstColl(Collection<A> c)
{c.add(getFst());}
```

```
Paire<Integer,Integer> p2 = new
    Paire<Integer,Integer>(9,10);
Collection<Object> co = new LinkedList<Object>();


p2.copieFstColl(co); // pourtant mettre un Integer dans
// une Collection d' objets ne devrait
// pas poser problème


```

# Le paramétrage par des jokers contrainte super

- Nouvelle version : on peut mettre un A dans une collection de A ou d'un type supérieur à A

```
public void copieFstColl(Collection<? super A> c)
{c.add(getFst());}
```

```
Paire<Integer,Integer> p2 = new
    Paire<Integer,Integer>(9,10);
Collection<Object> co = new LinkedList<Object>();
p2.copieFstColl(co);
```

# Résumé

- Paramétrage des classes <A>
- Paramétrage des méthodes
- Principe de l'effacement de type
- Paramétrage constraint <A extends ...>
- Jokers ? Et *super*

Gilad Bracha,

Generics in the Java Programming Language,

<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>,

2004

Notes de cours/TP correspondant aux transparents

<http://www.lirmm.fr/~huchard/Enseignement/UMINM202/genericJava15.pdf>

# Generics in the Java Programming Language

Gilad Bracha

July 5, 2004

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Defining Simple Generics</b>	<b>3</b>
<b>3</b>	<b>Generics and Subtyping</b>	<b>4</b>
<b>4</b>	<b>Wildcards</b>	<b>5</b>
4.1	Bounded Wildcards . . . . .	6
<b>5</b>	<b>Generic Methods</b>	<b>7</b>
<b>6</b>	<b>Interoperating with Legacy Code</b>	<b>10</b>
6.1	Using Legacy Code in Generic Code . . . . .	10
6.2	Erasure and Translation . . . . .	12
6.3	Using Generic Code in Legacy Code . . . . .	13
<b>7</b>	<b>The Fine Print</b>	<b>14</b>
7.1	A Generic Class is Shared by all its Invocations . . . . .	14
7.2	Casts and InstanceOf . . . . .	14
7.3	Arrays . . . . .	15
<b>8</b>	<b>Class Literals as Run-time Type Tokens</b>	<b>16</b>
<b>9</b>	<b>More Fun with Wildcards</b>	<b>18</b>
9.1	Wildcard Capture . . . . .	20
<b>10</b>	<b>Converting Legacy Code to Use Generics</b>	<b>20</b>
<b>11</b>	<b>Acknowledgements</b>	<b>23</b>

# 1 Introduction

JDK 1.5 introduces several extensions to the Java programming language. One of these is the introduction of *generics*.

This tutorial is aimed at introducing you to generics. You may be familiar with similar constructs from other languages, most notably C++ templates. If so, you'll soon see that there are both similarities and important differences. If you are not familiar with look-a-alike constructs from elsewhere, all the better; you can start afresh, without unlearning any misconceptions.

Generics allow you to abstract over types. The most common examples are container types, such as those in the Collection hierarchy.

Here is a typical usage of that sort:

```
List myIntList = new LinkedList(); // 1  
myIntList.add(new Integer(0)); // 2  
Integer x = (Integer) myIntList.iterator().next(); // 3
```

The cast on line 3 is slightly annoying. Typically, the programmer knows what kind of data has been placed into a particular list. However, the cast is essential. The compiler can only guarantee that an `Object` will be returned by the iterator. To ensure the assignment to a variable of type `Integer` is type safe, the cast is required.

Of course, the cast not only introduces clutter. It also introduces the possibility of a run time error, since the programmer might be mistaken.

What if programmers could actually express their intent, and mark a list as being restricted to contain a particular data type? This is the core idea behind generics. Here is a version of the program fragment given above using generics:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'  
myIntList.add(new Integer(0)); //2'  
Integer x = myIntList.iterator().next(); // 3'
```

Notice the type declaration for the variable `myIntList`. It specifies that this is not just an arbitrary `List`, but a `List` of `Integer`, written `List<Integer>`. We say that `List` is a generic interface that takes a *type parameter* - in this case, `Integer`. We also specify a type parameter when creating the list object.

The other thing to pay attention to is that the cast is gone on line 3'.

Now, you might think that all we've accomplished is to move the clutter around. Instead of a cast to `Integer` on line 3, we have `Integer` as a type parameter on line 1'. However, there is a very big difference here. The compiler can now check the type correctness of the program at compile-time. When we say that `myIntList` is declared with type `List<Integer>`, this tells us something about the variable `myIntList`, which holds true wherever and whenever it is used, and the compiler will guarantee it. In contrast, the cast tells us something the programmer thinks is true at a single point in the code.

The net effect, especially in large programs, is improved readability and robustness.

## 2 Defining Simple Generics

Here is a small excerpt from the definitions of the interfaces `List` and `Iterator` in package `java.util`:

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

This should all be familiar, except for the stuff in angle brackets. Those are the declarations of the *formal type parameters* of the interfaces `List` and `Iterator`.

Type parameters can be used throughout the generic declaration, pretty much where you would use ordinary types (though there are some important restrictions; see section 7).

In the introduction, we saw *invocations* of the generic type declaration `List`, such as `List<Integer>`. In the invocation (usually called a *parameterized type*), all occurrences of the formal type parameter (`E` in this case) are replaced by the *actual type argument* (in this case, `Integer`).

You might imagine that `List<Integer>` stands for a version of `List` where `E` has been uniformly replaced by `Integer`:

```
public interface IntegerList {
    void add(Integer x)
    Iterator<Integer> iterator();
}
```

This intuition can be helpful, but it's also misleading.

It is helpful, because the parameterized type `List<Integer>` does indeed have methods that look just like this expansion.

It is misleading, because the declaration of a generic is never actually expanded in this way. There aren't multiple copies of the code: not in source, not in binary, not on disk and not in memory. If you are a C++ programmer, you'll understand that this is very different than a C++ template.

A generic type declaration is compiled once and for all, and turned into a single class file, just like an ordinary class or interface declaration.

Type parameters are analogous to the ordinary parameters used in methods or constructors. Much like a method has *formal value parameters* that describe the kinds of values it operates on, a generic declaration has formal type parameters. When a method is invoked, *actual arguments* are substituted for the formal parameters, and the method body is evaluated. When a generic declaration is invoked, the actual type arguments are substituted for the formal type parameters.

A note on naming conventions. We recommend that you use pithy (single character if possible) yet evocative names for formal type parameters. It's best to avoid lower

case characters in those names, making it easy to distinguish formal type parameters from ordinary classes and interfaces. Many container types use `E`, for element, as in the examples above. We'll see some additional conventions in later examples.

### 3 Generics and Subtyping

Let's test our understanding of generics. Is the following code snippet legal?

```
List<String> ls = new ArrayList<String>(); //1  
List<Object> lo = ls; //2
```

Line 1 is certainly legal. The trickier part of the question is line 2. This boils down to the question: is a `List` of `String` a `List` of `Object`. Most people's instinct is to answer: "sure!".

Well, take a look at the next few lines:

```
lo.add(new Object()); //3  
String s = ls.get(0); // 4: attempts to assign an Object to a String!
```

Here we've aliased `ls` and `lo`. Accessing `ls`, a list of `String`, through the alias `lo`, we can insert arbitrary objects into it. As a result `ls` does not hold just `Strings` anymore, and when we try and get something out of it, we get a rude surprise.

The Java compiler will prevent this from happening of course. Line 2 will cause a compile time error.

In general, if `Foo` is a subtype (subclass or subinterface) of `Bar`, and `G` is some generic type declaration, it is **not** the case that `G<Foo>` is a subtype of `G<Bar>`. This is probably the hardest thing you need to learn about generics, because it goes against our deeply held intuitions.

The problem with that intuition is that it assumes that collections don't change. Our instinct takes these things to be immutable.

For example, if the department of motor vehicles supplies a list of drivers to the census bureau, this seems reasonable. We think that a `List<Driver>` is a `List<Person>`, assuming that `Driver` is a subtype of `Person`. In fact, what is being passed is a `copy` of the registry of drivers. Otherwise, the census bureau could add new people who are not drivers into the list, corrupting the DMV's records.

In order to cope with this sort of situation, it's useful to consider more flexible generic types. The rules we've seen so far are quite restrictive.

## 4 Wildcards

Consider the problem of writing a routine that prints out all the elements in a collection. Here's how you might write it in an older version of the language:

```
void printCollection(Collection c) {
    Iterator i = c.iterator();
    for (k = 0; k < c.size(); k++) {
        System.out.println(i.next());
    }
}
```

And here is a naive attempt at writing it using generics (and the new `for` loop syntax):

```
void printCollection(Collection<Object> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

The problem is that this new version is much less useful than the old one. Whereas the old code could be called with any kind of collection as a parameter, the new code only takes `Collection<Object>`, which, as we've just demonstrated, is *not* a supertype of all kinds of collections!

So what *is* the supertype of all kinds of collections? It's written `Collection<?>` (pronounced "collection of unknown"), that is, a collection whose element type matches anything. It's called a *wildcard type* for obvious reasons. We can write:

```
void printCollection(Collection<?> c) {
    for (Object e : c) {
        System.out.println(e);
    }
}
```

and now, we can call it with any type of collection. Notice that inside `printCollection()`, we can still read elements from `c` and give them type `Object`. This is always safe, since whatever the actual type of the collection, it does contain objects. It isn't safe to add arbitrary objects to it however:

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // compile time error
```

Since we don't know what the element type of `c` stands for, we cannot add objects to it. The `add()` method takes arguments of type `E`, the element type of the collection. When the actual type parameter is `?`, it stands for some unknown type. Any parameter we pass to `add` would have to be a subtype of this unknown type. Since we don't know what type that is, we cannot pass anything in. The sole exception is `null`, which is a member of every type.

On the other hand, given a `List<?>`, we **can** call `get()` and make use of the result. The result type is an unknown type, but we always know that it is an object. It is

therefore safe to assign the result of `get()` to a variable of type `Object` or pass it as a parameter where the type `Object` is expected.

## 4.1 Bounded Wildcards

Consider a simple drawing application that can draw shapes such as rectangles and circles. To represent these shapes within the program, you could define a class hierarchy such as this:

```
public abstract class Shape {  
    public abstract void draw(Canvas c);  
}  
public class Circle extends Shape {  
    private int x, y, radius;  
    public void draw(Canvas c) { ... }  
}  
public class Rectangle extends Shape {  
    private int x, y, width, height;  
    public void draw(Canvas c) { ... }  
}
```

These classes can be drawn on a canvas:

```
public class Canvas {  
    public void draw(Shape s) {  
        s.draw(this);  
    }  
}
```

Any drawing will typically contain a number of shapes. Assuming that they are represented as a list, it would be convenient to have a method in `Canvas` that draws them all:

```
public void drawAll(List<Shape> shapes) {  
    for (Shape s: shapes) {  
        s.draw(this);  
    }  
}
```

Now, the type rules say that `drawAll()` can only be called on lists of exactly `Shape`: it cannot, for instance, be called on a `List<Circle>`. That is unfortunate, since all the method does is read shapes from the list, so it could just as well be called on a `List<Circle>`. What we really want is for the method to accept a list of *any* kind of shape:

```
public void drawAll(List<? extends Shape> shapes) { ... }
```

There is a small but very important difference here: we have replaced the type `List<Shape>` with `List<? extends Shape>`. Now `drawAll()` will accept lists of any subclass of `Shape`, so we can now call it on a `List<Circle>` if we want.

`List<? extends Shape>` is an example of a *bounded wildcard*. The `?` stands for an unknown type, just like the wildcards we saw earlier. However, in this case, we know that this unknown type is in fact a subtype of `Shape`<sup>1</sup>. We say that `Shape` is the *upper bound* of the wildcard.

There is, as usual, a price to be paid for the flexibility of using wildcards. That price is that it is now illegal to write into `shapes` in the body of the method. For instance, this is not allowed:

```
public void addRectangle(List<? extends Shape> shapes) {
    shapes.add(0, new Rectangle()); // compile-time error!
}
```

You should be able to figure out why the code above is disallowed. The type of the second parameter to `shapes.add()` is `? extends Shape` - an unknown subtype of `Shape`. Since we don't know what type it is, we don't know if it is a supertype of `Rectangle`; it might or might not be such a supertype, so it isn't safe to pass a `Rectangle` there.

Bounded wildcards are just what one needs to handle the example of the DMV passing its data to the census bureau. Our example assumes that the data is represented by mapping from names (represented as strings) to people (represented by reference types such as `Person` or its subtypes, such as `Driver`). `Map<K,V>` is an example of a generic type that takes two type arguments, representing the keys and values of the map.

Again, note the naming convention for formal type parameters - `K` for keys and `V` for values.

```
public class Census {
    public static void
        addRegistry(Map<String, ? extends Person> registry) { ... }
    }
    ...
    Map<String, Driver> allDrivers = ...;
    Census.addRegistry(allDrivers);
```

## 5 Generic Methods

Consider writing a method that takes an array of objects and a collection and puts all objects in the array into the collection.

Here is a first attempt:

```
static void fromArrayToCollection(Object[] a, Collection<?> c) {
    for (Object o : a) {
        c.add(o); // compile time error
    }}
```

By now, you will have learned to avoid the beginner's mistake of trying to use `Collection<Object>` as the type of the collection parameter. You may or may not

---

<sup>1</sup>It could be `Shape` itself, or some subclass; it need not literally extend `Shape`.

have recognized that using `Collection<?>` isn't going to work either. Recall that you cannot just shove objects into a collection of unknown type.

The way to do deal with these problems is to use *generic methods*. Just like type declarations, method declarations can be generic - that is, parameterized by one or more type parameters.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c) {  
    for (T o : a) {  
        c.add(o); // correct  
    }  
}
```

We can call this method with any kind of collection whose element type is a supertype of the element type of the array.

```
Object[] oa = new Object[100];  
Collection<Object> co = new ArrayList<Object>();  
fromArrayToCollection(oa, co); // T inferred to be Object  
String[] sa = new String[100];  
Collection<String> cs = new ArrayList<String>();  
fromArrayToCollection(sa, cs); // T inferred to be String  
fromArrayToCollection(sa, co); // T inferred to be Object  
Integer[] ia = new Integer[100];  
Float[] fa = new Float[100];  
Number[] na = new Number[100];  
Collection<Number> cn = new ArrayList<Number>();  
fromArrayToCollection(ia, cn); // T inferred to be Number  
fromArrayToCollection(fa, cn); // T inferred to be Number  
fromArrayToCollection(na, cn); // T inferred to be Number  
fromArrayToCollection(na, co); // T inferred to be Object  
fromArrayToCollection(na, cs); // compile-time error
```

Notice that we don't have to pass an actual type argument to a generic method. The compiler infers the type argument for us, based on the types of the actual arguments. It will generally infer the most specific type argument that will make the call type-correct.

One question that arises is: when should I use generic methods, and when should I use wildcard types? To understand the answer, let's examine a few methods from the `Collection` libraries.

```
interface Collection<E> {  
    public boolean containsAll(Collection<?> c);  
    public boolean addAll(Collection<? extends E> c);  
}
```

We could have used generic methods here instead:

```
interface Collection<E> {  
    public <T> boolean containsAll(Collection<T> c);  
    public <T extends E> boolean addAll(Collection<T> c);  
    // hey, type variables can have bounds too!  
}
```

However, in both `containsAll` and `addAll`, the type parameter `T` is used only once. The return type doesn't depend on the type parameter, nor does any other argument to the method (in this case, there simply is only one argument). This tells us that the type argument is being used for polymorphism; its only effect is to allow a variety of actual argument types to be used at different invocation sites. If that is the case, one should use wildcards. Wildcards are designed to support flexible subtyping, which is what we're trying to express here.

Generic methods allow type parameters to be used to express dependencies among the types of one or more arguments to a method and/or its return type. If there isn't such a dependency, a generic method should not be used.

It is possible to use both generic methods and wildcards in tandem. Here is the method `Collections.copy()`:

```
class Collections {
    public static <T> void copy(List<T> dest, List<? extends T> src){...}
}
```

Note the dependency between the types of the two parameters. Any object copied from the source list, `src`, must be assignable to the element type `T` of the destination list, `dst`. So the element type of `src` can be any subtype of `T` - we don't care which. The signature of `copy` expresses the dependency using a type parameter, but uses a wildcard for the element type of the second parameter.

We could have written the signature for this method another way, without using wildcards at all:

```
class Collections {
    public static <T, S extends T>
        void copy(List<T> dest, List<S> src){...}
}
```

This is fine, but while the first type parameter is used both in the type of `dst` and in the bound of the second type parameter, `S`, `S` itself is only used once, in the type of `src` - nothing else depends on it. This is a sign that we can replace `S` with a wildcard. Using wildcards is clearer and more concise than declaring explicit type parameters, and should therefore be preferred whenever possible.

Wildcards also have the advantage that they can be used outside of method signatures, as the types of fields, local variables and arrays. Here is an example.

Returning to our shape drawing problem, suppose we want to keep a history of drawing requests. We can maintain the history in a static variable inside class `Shape`, and have `drawAll()` store its incoming argument into the history field.

```
static List<List<? extends Shape>> history =
new ArrayList<List<? extends Shape>>();
public void drawAll(List<? extends Shape> shapes) {
    history.addLast(shapes);
    for (Shape s: shapes) {
        s.draw(this);
    }
}
```

Finally, again let's take note of the naming convention used for the type parameters. We use **T** for type, whenever there isn't anything more specific about the type to distinguish it. This is often the case in generic methods. If there are multiple type parameters, we might use letters that neighbor **T** in the alphabet, such as **S**. If a generic method appears inside a generic class, it's a good idea to avoid using the same names for the type parameters of the method and class, to avoid confusion. The same applies to nested generic classes.

## 6 Interoperating with Legacy Code

Until now, all our examples have assumed an idealized world, where everyone is using the latest version of the Java programming language, which supports generics.

Alas, in reality this isn't the case. Millions of lines of code have been written in earlier versions of the language, and they won't all be converted overnight.

Later, in section 10, we will tackle the problem of converting your old code to use generics. In this section, we'll focus on a simpler problem: how can legacy code and generic code interoperate? This question has two parts: using legacy code from within generic code, and using generic code within legacy code.

### 6.1 Using Legacy Code in Generic Code

How can you use old code, while still enjoying the benefits of generics in your own code?

As an example, assume you want to use the package `com.Fooblibar.widgets`. The folks at Fooblibar.com<sup>2</sup> market a system for inventory control, highlights of which are shown below:

```
package com.Fooblibar.widgets;
public interface Part { ... }
public class Inventory {
    /**
     * Adds a new Assembly to the inventory database.
     * The assembly is given the name name, and consists of a set
     * parts specified by parts. All elements of the collection parts
     * must support the Part interface.
    */
    public static void addAssembly(String name, Collection parts) {...}
    public static Assembly getAssembly(String name) {...}
}
public interface Assembly {
    Collection getParts(); // Returns a collection of Parts
}
```

Now, you'd like to add new code that uses the API above. It would be nice to ensure that you always called `addAssembly()` with the proper arguments - that is, that

---

<sup>2</sup>Fooblibar.com is a purely fictional company, used for illustration purposes. Any relation to any real company or institution, or any persons living or dead, is purely coincidental.

the collection you pass in is indeed a `Collection` of `Part`. Of course, generics are tailor made for this:

```
package com.mycompany.inventory;
import com.Fooblibar.widgets.*;
public class Blade implements Part {
}
public class Guillotine implements Part {
}
public class Main {
    public static void main(String[] args) {
        Collection<Part> c = new ArrayList<Part>();
        c.add(new Guillotine());
        c.add(new Blade());
        Inventory.addAssembly("thingee", c);
        Collection<Part> k = Inventory.getAssembly("thingee").getParts();
    }
}
```

When we call `addAssembly`, it expects the second parameter to be of type `Collection`. The actual argument is of type `Collection<Part>`. This works, but why? After all, most collections don't contain `Part` objects, and so in general, the compiler has no way of knowing what kind of collection the type `Collection` refers to.

In proper generic code, `Collection` would always be accompanied by a type parameter. When a generic type like `Collection` is used without a type parameter, it's called a *raw type*.

Most people's first instinct is that `Collection` really means `Collection<Object>`. However, as we saw earlier, it isn't safe to pass a `Collection<Part>` in a place where a `Collection<Object>` is required. It's more accurate to say that the type `Collection` denotes a collection of some unknown type, just like `Collection<?>`.

But wait, that can't be right either! Consider the call to `getParts()`, which returns a `Collection`. This is then assigned to `k`, which is a `Collection<Part>`. If the result of the call is a `Collection<?>`, the assignment would be an error.

In reality, the assignment is legal, but it generates an *unchecked warning*. The warning is needed, because the fact is that the compiler can't guarantee its correctness. We have no way of checking the legacy code in `getAssembly()` to ensure that indeed the collection being returned is a collection of `Parts`. The type used in the code is `Collection`, and one could legally insert all kinds of objects into such a collection.

So, shouldn't this be an error? Theoretically speaking, yes; but practically speaking, if generic code is going to call legacy code, this has to be allowed. It's up to you, the programmer, to satisfy yourself that in this case, the assignment is safe because the contract of `getAssembly()` says it returns a collection of `Parts`, even though the type signature doesn't show this.

So raw types are very much like wildcard types, but they are not typechecked as stringently. This is a deliberate design decision, to allow generics to interoperate with pre-existing legacy code.

Calling legacy code from generic code is inherently dangerous; once you mix generic code with non-generic legacy code, all the safety guarantees that the generic

type system usually provides are void. However, you are still better off than you were without using generics at all. At least you know the code on your end is consistent.

At the moment there's a lot more non-generic code out there than there is generic code, and there will inevitably be situations where they have to mix.

If you find that you must intermix legacy and generic code, pay close attention to the unchecked warnings. Think carefully how you can justify the safety of the code that gives rise to the warning.

What happens if you still made a mistake, and the code that caused a warning is indeed not type safe? Let's take a look at such a situation. In the process, we'll get some insight into the workings of the compiler.

## 6.2 Erasure and Translation

```
public String loophole(Integer x) {
    List<String> ys = new LinkedList<String>();
    List xs = ys;
    xs.add(x); // compile-time unchecked warning
    return ys.iterator().next();
}
```

Here, we've aliased a list of strings and a plain old list. We insert an `Integer` into the list, and attempt to extract a `String`. This is clearly wrong. If we ignore the warning and try to execute this code, it will fail exactly at the point where we try to use the wrong type. At run time, this code behaves like:

```
public String loophole(Integer x) {
    List ys = new LinkedList;
    List xs = ys;
    xs.add(x);
    return (String) ys.iterator().next(); // run time error
}
```

When we extract an element from the list, and attempt to treat it as a string by casting it to `String`, we will get a `ClassCastException`. The exact same thing happens with the generic version of `loophole()`.

The reason for this is, that generics are implemented by the Java compiler as a front-end conversion called *erasure*. You can (almost) think of it as a source-to-source translation, whereby the generic version of `loophole()` is converted to the non-generic version.

As a result, **the type safety and integrity of the Java virtual machine are never at risk, even in the presence of unchecked warnings.**

Basically, erasure gets rid of (or *erases*) all generic type information. All the type information between angle brackets is thrown out, so, for example, a parameterized type like `List<String>` is converted into `List`. All remaining uses of type variables are replaced by the upper bound of the type variable (usually `Object`). And, whenever the resulting code isn't type-correct, a cast to the appropriate type is inserted, as in the last line of `loophole`.

The full details of erasure are beyond the scope of this tutorial, but the simple description we just gave isn't far from the truth. It's good to know a bit about this, especially if you want to do more sophisticated things like converting existing APIs to use generics (see section 10), or just want to understand why things are the way they are.

### 6.3 Using Generic Code in Legacy Code

Now let's consider the inverse case. Imagine that Fooblibar.com chose to convert their API to use generics, but that some of their clients haven't yet. So now the code looks like:

```
package com.Fooblibar.widgets;
public interface Part { ...}
public class Inventory {
    /**
     * Adds a new Assembly to the inventory database.
     * The assembly is given the name name, and consists of a set
     * parts specified by parts. All elements of the collection parts
     * must support the Part interface.
    */
    public static void addAssembly(String name, Collection<Part> parts) {...}
    public static Assembly getAssembly(String name) {...}
}
public interface Assembly {
    Collection<Part> getParts(); // Returns a collection of Parts
}
```

and the client code looks like:

```
package com.mycompany.inventory;
import com.Fooblibar.widgets.*;
public class Blade implements Part {
}
public class Guillotine implements Part {
}
public class Main {
    public static void main(String[] args) {
        Collection c = new ArrayList();
        c.add(new Guillotine());
        c.add(new Blade());
        Inventory.addAssembly("thingee", c); // 1: unchecked warning
        Collection k = Inventory.getAssembly("thingee").getParts();
    }
}
```

The client code was written before generics were introduced, but it uses the package `com.Fooblibar.widgets` and the collection library, both of which are using generic types. All the uses of generic type declarations in the client code are raw types.

Line 1 generates an unchecked warning, because a raw Collection is being passed in where a Collection of Parts is expected, and the compiler cannot ensure that the raw Collection really is a Collection of Parts.

As an alternative, you can compile the client code using the source 1.4 flag, ensuring that no warnings are generated. However, in that case you won't be able to use any of the new language features introduced in JDK 1.5.

## 7 The Fine Print

### 7.1 A Generic Class is Shared by all its Invocations

What does the following code fragment print?

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

You might be tempted to say `false`, but you'd be wrong. It prints `true`, because all instances of a generic class have the same run-time class, regardless of their actual type parameters.

Indeed, what makes a class generic is the fact that it has the same behavior for all of its possible type parameters; the same class can be viewed as having many different types.

As consequence, the static variables and methods of a class are also shared among all the instances. That is why it is illegal to refer to the type parameters of a type declaration in a static method or initializer, or in the declaration or initializer of a static variable.

### 7.2 Casts and InstanceOf

Another implication of the fact that a generic class is shared among all its instances, is that it usually makes no sense to ask an instance if it is an instance of a particular invocation of a generic type:

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ...} // illegal
```

similarly, a cast such as

```
Collection<String> cstr = (Collection<String>) cs; // unchecked warning
```

gives an unchecked warning, since this isn't something the run time system is going to check for you.

The same is true of type variables

```
<T> T badCast(T t, Object o) {return (T) o; // unchecked warning
}
```

Type variables don't exist at run time. This means that they entail no performance overhead in either time nor space, which is nice. Unfortunately, it also means that you can't reliably use them in casts.

### 7.3 Arrays

The component type of an array object may not be a type variable or a parameterized type, unless it is an (unbounded) wildcard type. You can declare array *types* whose element type is a type variable or a parameterized type, but not array *objects*.

This is annoying, to be sure. This restriction is necessary to avoid situations like:

```
List<String>[] lsa = new List<String>[10]; // not really allowed
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // unsound, but passes run time store check
String s = lsa[1].get(0); // run-time error - ClassCastException
```

If arrays of parameterized type were allowed, the example above would compile without any unchecked warnings, and yet fail at run-time. We've had type-safety as a primary design goal of generics. In particular, the language is designed to guarantee that **if your entire application has been compiled without unchecked warnings using javac -source 1.5, it is type safe**.

However, you can still use wildcard arrays. Here are two variations on the code above. The first forgoes the use of both array objects and array types whose element type is parameterized. As a result, we have to cast explicitly to get a `String` out of the array.

```
List<?>[] lsa = new List<?>[10]; // ok, array of unbounded wildcard type
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = (String) lsa[1].get(0); // run time error, but cast is explicit
```

In the next variation, we refrain from creating an array object whose element type is parameterized, but still use an array type with a parameterized element type. This is legal, but generates an unchecked warning. Indeed, the code is unsafe, and eventually an error occurs.

```
List<String>[] lsa = new List<?>[10]; // unchecked warning - this is unsafe!
Object o = lsa;
Object[] oa = (Object[]) o;
List<Integer> li = new ArrayList<Integer>();
li.add(new Integer(3));
oa[1] = li; // correct
String s = lsa[1].get(0); // run time error, but we were warned
```

Similarly, attempting to create an array object whose element type is a type variable causes a compile-time error:

```
<T> T[] makeArray(T t) {  
    return new T[100]; // error  
}
```

Since type variables don't exist at run time, there is no way to determine what the actual array type would be.

The way to work around these kinds of limitations is to use class literals as run time type tokens, as described in section 8.

## 8 Class Literals as Run-time Type Tokens

One of the changes in JDK 1.5 is that the class `java.lang.Class` is generic. It's an interesting example of using genericity for something other than a container class.

Now that `Class` has a type parameter `T`, you might well ask, what does `T` stand for? It stands for the type that the `Class` object is representing.

For example, the type of `String.class` is `Class<String>`, and the type of `Serializable.class` is `Class<Serializable>`. This can be used to improve the type safety of your reflection code.

In particular, since the `newInstance()` method in `Class` now returns a `T`, you can get more precise types when creating objects reflectively.

For example, suppose you need to write a utility method that performs a database query, given as a string of SQL, and returns a collection of objects in the database that match that query.

One way is to pass in a factory object explicitly, writing code like:

```
interface Factory<T> { T make();}  
public <T> Collection<T> select(Factory<T> factory, String statement) {  
    Collection<T> result = new ArrayList<T>();  
    /* run sql query using jdbc */  
    for (* iterate over jdbc results *) {  
        T item = factory.make();  
        /* use reflection and set all of item's fields from sql results */  
        result.add(item);  
    }  
    return result;  
}
```

You can call this either as

```
select(new Factory<EmplInfo>(){ public EmplInfo make() {  
    return new EmplInfo();  
}}  
, "selection string");
```

or you can declare a class `EmplInfoFactory` to support the `Factory` interface

```

class EmpInfoFactory implements Factory<EmpInfo> {
    ...
    public EmpInfo make() { return new EmpInfo(); }
}

```

and call it

```
select(getMyEmpInfoFactory(), "selection string");
```

The downside of this solution is that it requires either:

- the use of verbose anonymous factory classes at the call site, or
- declaring a factory class for every type used and passing a factory instance at the call site, which is somewhat unnatural.

It is very natural to use the class literal as a factory object, which can then be used by reflection. Today (without generics) the code might be written:

```

Collection emps = sqlUtility.select(EmpInfo.class, "select * from emps");
...
public static Collection select(Class c, String sqlStatement) {
    Collection result = new ArrayList();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        Object item = c.newInstance();
        /* use reflection and set all of item's fields from sql results */
        result.add(item);
    }
    return result;
}

```

However, this would not give us a collection of the precise type we desire. Now that **Class** is generic, we can instead write

```

Collection<EmpInfo> emps =
    sqlUtility.select(EmpInfo.class, "select * from emps");
...
public static <T> Collection<T> select(Class<T>c, String sqlStatement) {
    Collection<T> result = new ArrayList<T>();
    /* run sql query using jdbc */
    for ( /* iterate over jdbc results */ ) {
        T item = c.newInstance();
        /* use reflection and set all of item's fields from sql results */
        result.add(item);
    }
    return result;
}

```

giving us the precise type of collection in a type safe way.

This technique of using class literals as run time type tokens is a very useful trick to know. It's an idiom that's used extensively in the new APIs for manipulating annotations, for example.

## 9 More Fun with Wildcards

In this section, we'll consider some of the more advanced uses of wildcards. We've seen several examples where bounded wildcards were useful when reading from a data structure. Now consider the inverse, a write-only data structure.

The interface `Sink` is a simple example of this sort.

```
interface Sink<T> {
    flush(T t);
}
```

We can imagine using it as demonstrated by the code below. The method `writeAll()` is designed to flush all elements of the collection `coll` to the sink `snk`, and return the last element flushed.

```
public static <T> T writeAll(Collection<T> coll, Sink<T> snk){
    T last;
    for (T t : coll) {
        last = t;
        snk.flush(last);
    }
    return last;
}
Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s); // illegal call
```

As written, the call to `writeAll()` is illegal, as no valid type argument can be inferred; neither `String` nor `Object` are appropriate types for `T`, because the `Collection` element and the `Sink` element must be of the same type.

We can fix this by modifying the signature of `writeAll()` as shown below, using a wildcard.

```
public static <T> T writeAll(Collection<? extends T>, Sink<T>){...}
String str = writeAll(cs, s); // call ok, but wrong return type
```

The call is now legal, but the assignment is erroneous, since the return type inferred is `Object` because `T` matches the element type of `s`, which is `Object`.

The solution is to use a form of bounded wildcard we haven't seen yet: wildcards with a *lower bound*. The syntax `? super T` denotes an unknown type that is a supertype of `T`<sup>3</sup>. It is the dual of the bounded wildcards we've been using, where we use `? extends T` to denote an unknown type that is a subtype of `T`.

```
public static <T> T writeAll(Collection<T> coll, Sink<? super T> snk){...}
String str = writeAll(cs, s); // Yes!
```

Using this syntax, the call is legal, and the inferred type is `String`, as desired.

---

<sup>3</sup>Or `T` itself. Remember, the supertype relation is reflexive.

Now let's turn to a more realistic example. A `java.util.TreeSet<E>` represents a tree of elements of type `E` that are ordered. One way to construct a `TreeSet` is to pass a `Comparator` object to the constructor. That comparator will be used to sort the elements of the `TreeSet` according to a desired ordering.

```
TreeSet(Comparator<E> c)
```

The `Comparator` interface is essentially:

```
interface Comparator<T> {
    int compare(T fst, T snd);
}
```

Suppose we want to create a `TreeSet<String>` and pass in a suitable comparator, We need to pass it a `Comparator` that can compare `Strings`. This can be done by a `Comparator<String>`, but a `Comparator<Object>` will do just as well. However, we won't be able to invoke the constructor given above on a `Comparator<Object>`. We can use a lower bounded wildcard to get the flexibility we want:

```
TreeSet(Comparator<? super E> c)
```

This allows any applicable comparator to be used.

As a final example of using lower bounded wildcards, lets look at the method `Collections.max()`, which returns the maximal element in a collection passed to it as an argument.

Now, in order for `max()` to work, all elements of the collection being passed in must implement `Comparable`. Furthermore, they must all be comparable to *each other*.

A first attempt at generifying this method signature yields

```
public static <T extends Comparable<T>>
    T max(Collection<T> coll)
```

That is, the method takes a collection of some type `T` that is comparable to itself, and returns an element of that type. This turns out to be too restrictive.

To see why, consider a type that is comparable to arbitrary objects

```
class Foo implements Comparable<Object> {...}
Collection<Foo> cf = ...;
Collections.max(cf); // should work
```

Every element of `cf` is comparable to every other element in `cf`, since every such element is a `Foo`, which is comparable to any object, and in particular to another `Foo`. However, using the signature above, we find that the call is rejected. The inferred type must be `Foo`, but `Foo` does not implement `Comparable<Foo>`.

It isn't necessary that `T` be comparable to **exactly** itself. All that's required is that `T` be comparable to one of its supertypes. This give us:<sup>4</sup>

---

<sup>4</sup>The actual signature of `Collections.max()` is more involved. We return to it in section 10

```
public static <T extends Comparable<? super T>>
    T max(Collection<T> coll)
```

This reasoning applies to almost any usage of `Comparable` that is intended to work for arbitrary types: You always want to use `Comparable<? super T>`.

In general, if you have an API that only uses a type parameter `T` as an argument, its uses should take advantage of lower bounded wildcards (`? super T`). Conversely, if the API only returns `T`, you'll give your clients more flexibility by using upper bounded wildcards (`? extends T`).

## 9.1 Wildcard Capture

It should be pretty clear by now that given

```
Set<?> unknownSet = new HashSet<String>();
/** Add an element t to a Set s */
public static <T> void addToSet(Set<T> s, T t) { ... }
```

The call below is illegal.

```
addToSet(unknownSet, "abc"); // illegal
```

It makes no difference that the actual set being passed is a set of strings; what matters is that the expression being passed as an argument is a set of an unknown type, which cannot be guaranteed to be a set of strings, or of any type in particular.

Now, consider

```
class Collections {
    ...
    <T> public static Set<T> unmodifiableSet(Set<T> set) { ... }
}
Set<?> s = Collections.unmodifiableSet(unknownSet); // this works! Why?
```

It seems this should not be allowed; yet, looking at this specific call, it is perfectly safe to permit it. After all, `unmodifiableSet()` does work for any kind of `Set`, regardless of its element type.

Because this situation arises relatively frequently, there is a special rule that allows such code under very specific circumstances in which the code can be proven to be safe. This rule, known as *wildcard capture*, allows the compiler to infer the unknown type of a wildcard as a type argument to a generic method.

## 10 Converting Legacy Code to Use Generics

Earlier, we showed how new and legacy code can interoperate. Now, it's time to look at the harder problem of "generifying" old code.

If you decide to convert old code to use generics, you need to think carefully about how you modify the API.

You need to make certain that the generic API is not unduly restrictive; it must continue to support the original contract of the API. Consider again some examples from `java.util.Collection`. The pre-generic API looks like:

```
interface Collection {  
    public boolean containsAll(Collection c);  
    public boolean addAll(Collection c);  
}
```

A naive attempt to generify it is:

```
interface Collection<E> {  
    public boolean containsAll(Collection<E> c);  
    public boolean addAll(Collection<E> c);  
}
```

While this is certainly type safe, it doesn't live up to the API's original contract. The `containsAll()` method works with any kind of incoming collection. It will only succeed if the incoming collection really contains only instances of `E`, but:

- The static type of the incoming collection might differ, perhaps because the caller doesn't know the precise type of the collection being passed in, or perhaps because it is a `Collection<S>`, where `S` is a subtype of `E`.
- It's perfectly legitimate to call `containsAll()` with a collection of a different type. The routine should work, returning `false`.

In the case of `addAll()`, we should be able to add any collection that consists of instances of a subtype of `E`. We saw how to handle this situation correctly in section 5.

You also need to ensure that the revised API retains binary compatibility with old clients. This implies that the erasure of the API must be the same as the original, ungenerified API. In most cases, this falls out naturally, but there are some subtle cases. We'll examine one of the subtlest cases we've encountered, the method `Collections.max()`. As we saw in section 9, a plausible signature for `max()` is:

```
public static <T extends Comparable<? super T>>  
    T max(Collection<T> coll)
```

This is fine, except that the erasure of this signature is

```
public static Comparable max(Collection coll)
```

which is different than the original signature of `max()`:

```
public static Object max(Collection coll)
```

One could certainly have specified this signature for `max()`, but it was not done, and all the old binary class files that call `Collections.max()` depend on a signature that returns `Object`.

We can force the erasure to be different, by explicitly specifying a superclass in the bound for the formal type parameter T.

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<T> coll)
```

This is an example of giving *multiple bounds* for a type parameter, using the syntax  $T_1 \& T_2 \dots \& T_n$ . A type variable with multiple bounds is known to be a subtype of all of the types listed in the bound. When a multiple bound is used, the first type mentioned in the bound is used as the erasure of the type variable.

Finally, we should recall that `max` only reads from its input collection, and so is applicable to collections of any subtype of T.

This brings us to the actual signature used in the JDK:

```
public static <T extends Object & Comparable<? super T>>
    T max(Collection<? extends T> coll)
```

It's very rare that anything so involved comes up in practice, but expert library designers should be prepared to think very carefully when converting existing APIs.

Another issue to watch out for is *covariant returns*, that is, refining the return type of a method in a subclass. You should not take advantage of this feature in an old API. To see why, let's look at an example.

Assume your original API was of the form

```
public class Foo {
    public Foo create() {...} // Factory, should create an instance of whatever class it is declared in
}
public class Bar extends Foo {
    public Foo create() {...} // actually creates a Bar
}
```

Taking advantage of covariant returns, you modify it to:

```
public class Foo {
    public Foo create() {...} // Factory, should create an instance of whatever class it is declared in
}
public class Bar extends Foo {
    public Bar create() {...} // actually creates a Bar
}
```

Now, assume a third party client of your code wrote

```
public class Baz extends Bar {
    public Foo create() {...} // actually creates a Baz
}
```

The Java virtual machine does not directly support overriding of methods with different return types. This feature is supported by the compiler. Consequently, unless

the class `Baz` is recompiled, it will not properly override the `create()` method of `Bar`. Furthermore, `Baz` will have to be modified, since the code will be rejected as written - the return type of `create()` in `Baz` is not a subtype of the return type of `create()` in `Bar`.

## 11 Acknowledgements

Erik Ernst, Christian Plesner Hansen, Jeff Norton, Mads Torgersen, Peter von der Ahé and Philip Wadler contributed material to this tutorial.

Thanks to David Biesack, Bruce Chapman, David Flanagan, Neal Gafter, Örjan Petersson, Scott Seligman, Yoshiki Shibata and Kresten Krab Thorup for valuable feedback on earlier versions of this tutorial. Apologies to anyone whom I've forgotten.

# Interfaces/langage en Java

*GLIN505*

# Motivation

- Types **plus abstraits** que les classes
  - plus réutilisables
- Technique pour **masquer l'implémentation**
  - découplage public/privé : type/implémentation
- Favorise l'écriture de **code plus général**
  - écrit sur des types plus abstraits
- Relations de **spécialisation multiple**
  - entre les interfaces
  - entre les classes et les interfaces
  - Meilleure organisation des types

# Définition

- Interface
  - méthodes d'instances publiques et abstraites
    - public abstract
  - variables de classes constantes et publiques
    - public final static

# Syntaxe

```
public interface Iquadrilater {  
    public static final int nbCotes = 4;  
    public abstract float perimetre();  
}
```

*obligatoires ...peuvent être omis*

Écriture plus courante :

```
public interface Iquadrilater {  
    int nbCotes = 4;  
    float perimetre();  
}
```

# Syntaxe

```
public interface Iquadrilatere {  
    int nbCotes = 4;  
    float perimetre();  
}
```

Remarque : pas de constructeur

- ce n'est pas un oubli !
- Il n'y aura pas de constructeur par défaut généré

# Spécialisation (*extends*)

```
interface Irectangle extends Iquadrilatere
{
    float angle = 90;
    float angle();
    float largeur();
    float hauteur();
}
```

# Implémentation - classe concrète

toutes les opérations sont implémentées

```
interface Iquadrilatere{...}
```

```
interface Irectangle extends Iquadrilatere{...}
```

```
public class Rectangle implements Irectangle {  
    private float largeur,hauteur;  
    public Rectangle(){}
    public Rectangle(float l, float h){largeur=l;hauteur=h;}  
    public float périmètre(){return2*largeur()+2*hauteur();}  
    public float angle(){return Irectangle.angle;}  
    public float largeur(){return largeur;}  
    public float hauteur(){return hauteur;}  
}
```

# Implémentation - classe abstraite

certaines opérations ne sont pas implémentées

```
interface Iquadrilatere{...}
```

```
interface Irectangle extends Iquadrilatere{...}
```

```
public abstract class Rectangle implements Irectangle {  
    private float largeur,hauteur;  
    public Rectangle(){}
    public Rectangle(float l, float h){largeur=l;hauteur=h;}  
    // public float perimetre(){return2*largeur()+2*hauteur();}  
    public float angle(){return Irectangle.angle;}  
    // public float largeur(){return largeur;}  
    public float hauteur(){return hauteur;}  
}
```

# Ecriture de code plus général

```
public class StockRectangle {  
    Vector<Irectangle> listeRectangle = new Vector<Irectangle>();  
    public void ajoute(Irectangle r){listeRectangle.add(r);}  
    public float sommePerimetres()  
    { float sp=0;  
        for (int i=0; i<listeRectangle.size(); i++)  
            {sp+=listeRectangle.get(i).perimetre();}  
        return sp; }  
    }
```

- Fonctionne pour toute classe implémentant l' interface Irectangle
- Ne peut s' appuyer sur aucune partie d' implémentation

# Ecriture de code plus général

StockRectangle fonctionne pour des objets de Rectangle mais aussi pour des objets d'une autre implémentation de Irectangle

```
public class RectangleTab implements Irectangle
{ private float tab[] = new float[2];
public RectangleTab(){}
public RectangleTab(float l, float h){tab[0]=l; tab[1]=h;}
public float perimetre() {return 2*largeur() + 2*hauteur();}
public float angle() {return Irectangle.angle;}
public float largeur() {return tab[0];}
public float hauteur() {return tab[1];} }
```

# Interfaces et généricité paramétrique

Les interfaces se paramètrent comme les classes

```
public interface Pile<T> {  
    void empile(T t);  
    void depile();  
    T sommet();  
    boolean estVide();  
}
```

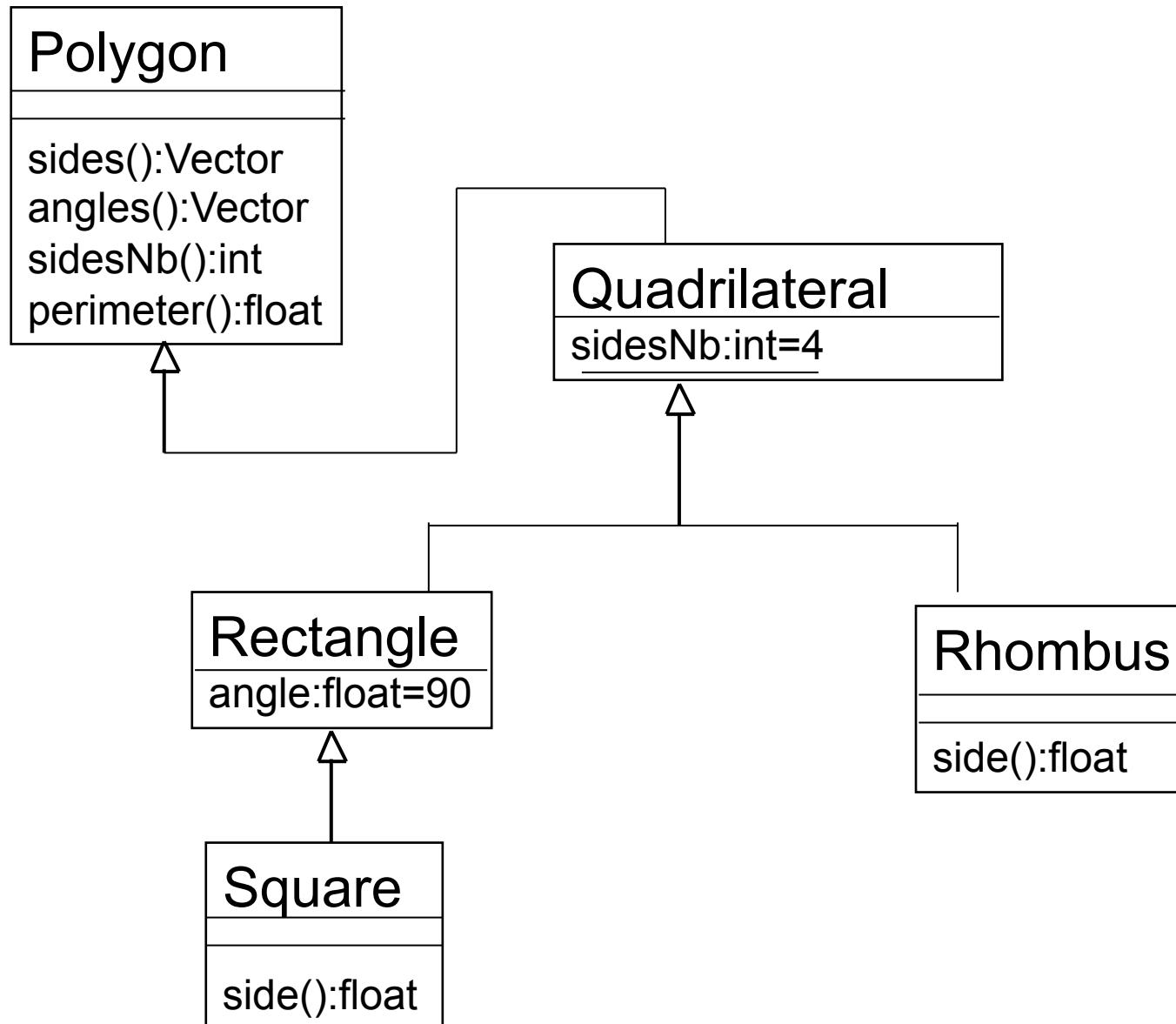
# Interfaces et généricité paramétrique

Les interfaces sont plus indiquées que les classes pour écrire les contraintes

```
public interface Comparable<A>
{public abstract boolean infStrict(A a);}
```

```
public class orderedSet<A extends Comparable<A>>
{....}
```

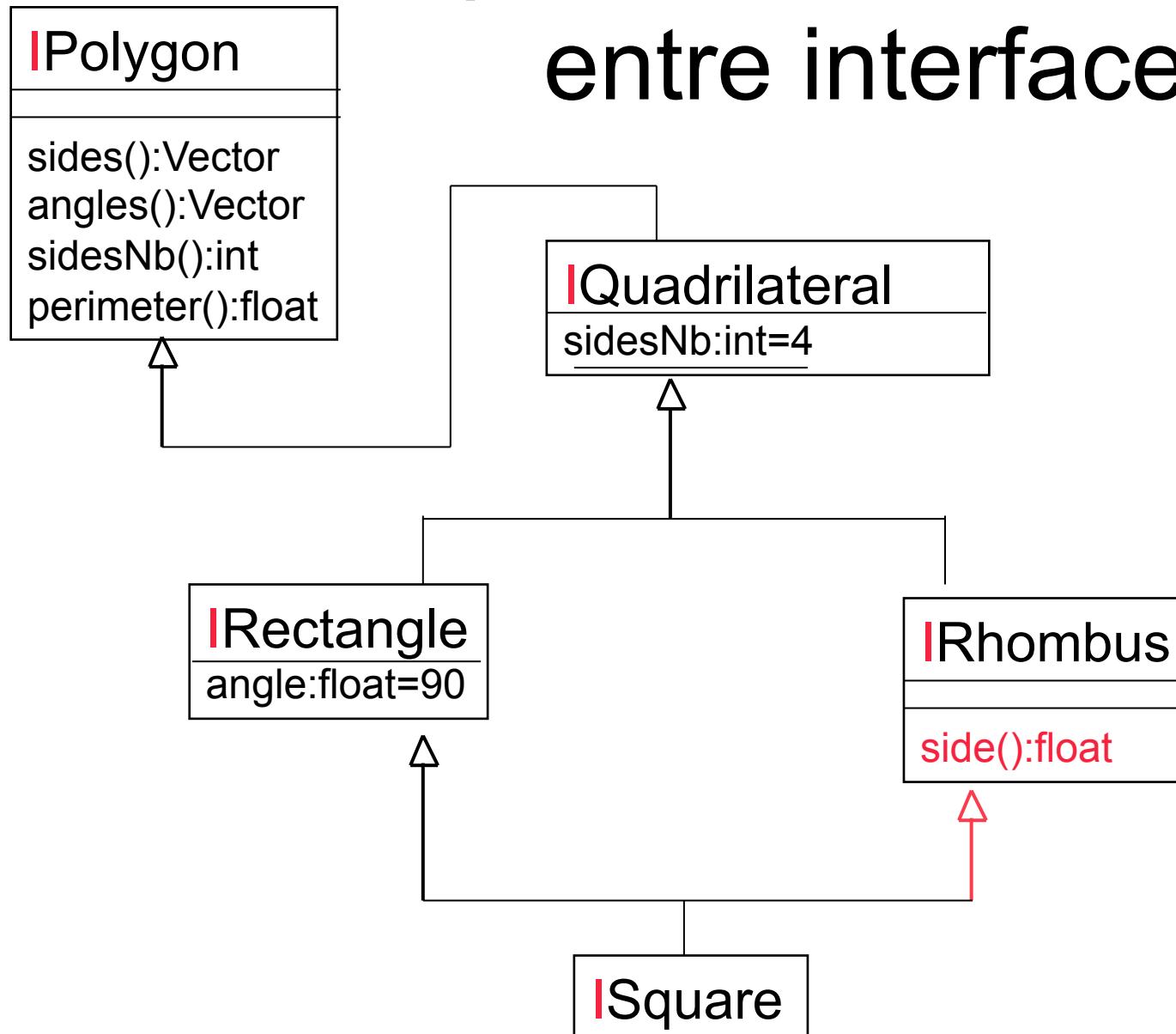
# Interfaces et spécialisation multiple



# Interfaces et spécialisation multiple

- Sans spécialisation multiple :
  - Pb de **polymorphisme** : les carrés ne peuvent être considérés à la fois comme des rectangles et comme des losanges
  - Pb de **redondance** : side()

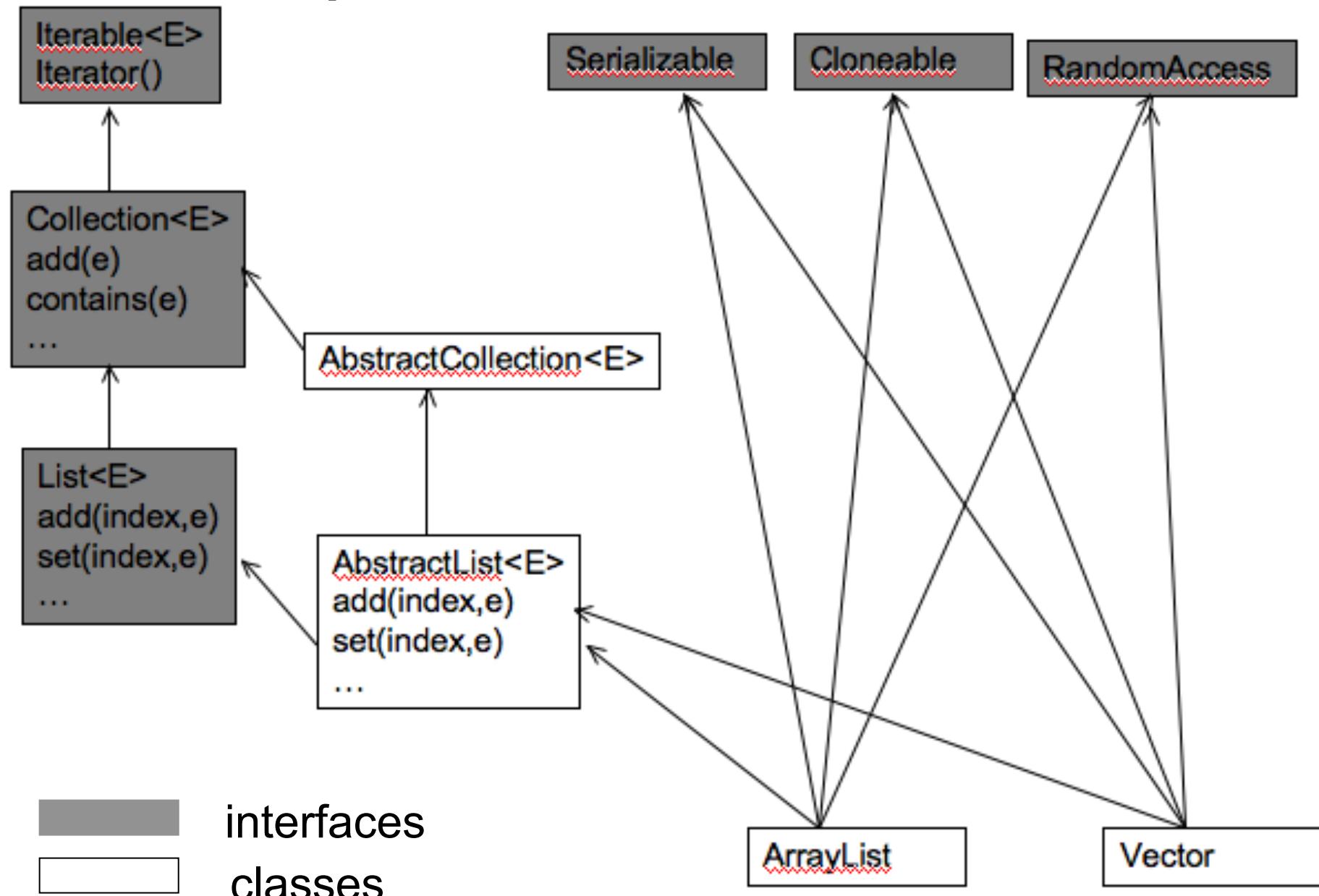
# Spécialisation multiple entre interfaces



# Quelques interfaces de Java les *marqueurs* (vides)

- **Cloneable** : Lorsqu'une classe implémente cette interface, ses objets peuvent être clonés
  - méthode `clone`
  - `protected` dans `Object`,
  - à redéfinir `public` dans la classe concernée
- **Serializable** : les objets de la classe peuvent être écrits dans un flux de données
  - `readObject`, `writeObject`

# Quelques interfaces de Java



# Quelques interfaces de Java

```
public interface Iterable<T>
{ Iterator<T>iterator(); }
```

```
public interface Iterator<T>{
    boolean hasNext(); T next();}
```

*// un code qui ne fait référence qu'à des interfaces*

```
public class StockRectangle{ ....
public float sommePerimetres() {
float sp=0;
Iterator<lrectangle> lt=listeRectangle.iterator();
while (lt.hasNext()){sp+=lt.next().perimetre();}
return sp; } .....
```

# Interface Cloneable

- Par défaut
  - protected Object clone() dans Object
- Pour effectuer une copie profonde
  - redéfinir

```
public Object clone()
```
  - au minimum faire la copie binaire

```
public Object clone() throws
CloneNotSupportedException
{return super.clone();}
```

# Classe Date clonable

```
class Date implements Cloneable
{
    private int jour;private int mois;private int annee;
    // accesseurs ...
    // constructeurs ...
    public Date(int j, int m, int a){jour=j; mois=m;
        annee=a;}

    public Object clone() throws CloneNotSupportedException
    { return super.clone();}

    public String toString()
    {return jour+" "+mois+" "+annee+" ";}
}
```

# Classe Employé clonable

```
public class Employe implements Cloneable
{private String name;
private Date dateEmbauche;
// accesseurs ..
// constructeurs ..
public String toString()
{return name+" "+dateEmbauche+" ";}
public Object clone() throws
CloneNotSupportedException
{ Employe e = (Employe)(super.clone());
e.dateEmbauche =(Date)(dateEmbauche.clone());
return e;}
```

# Un programme

```
Date d = new Date(1,1,1990);
System.out.println(d);
Employe z = new Employe("Zoe",d);
Employe l =(Employe)(z.clone());
l.name = "tutu";
l.dateEmbauche.setAnnee(2009);
System.out.println(z);
System.out.println(l);
```

# **Le mécanisme des exceptions en Java**

*HLIN505 – L3 – 2014*

Qu'est-ce qu'une exception?  
Comment définir et signaler des exceptions?  
Comment récupérer des exceptions?

# Qu'est-ce qu'une exception?

Un objet qui représente une *erreur à l'exécution*

*due à*

- une faute de saisie
- un problème matériel
- une faute de programmation

}

**Causes externes  
au programme**

**Bugs**

`T[i]` avec `i = T.length`

`ArrayIndexOutOfBoundsException`

`o.f()` avec `o = null`

`NullPointerException`

# Hiérarchie des exceptions en Java

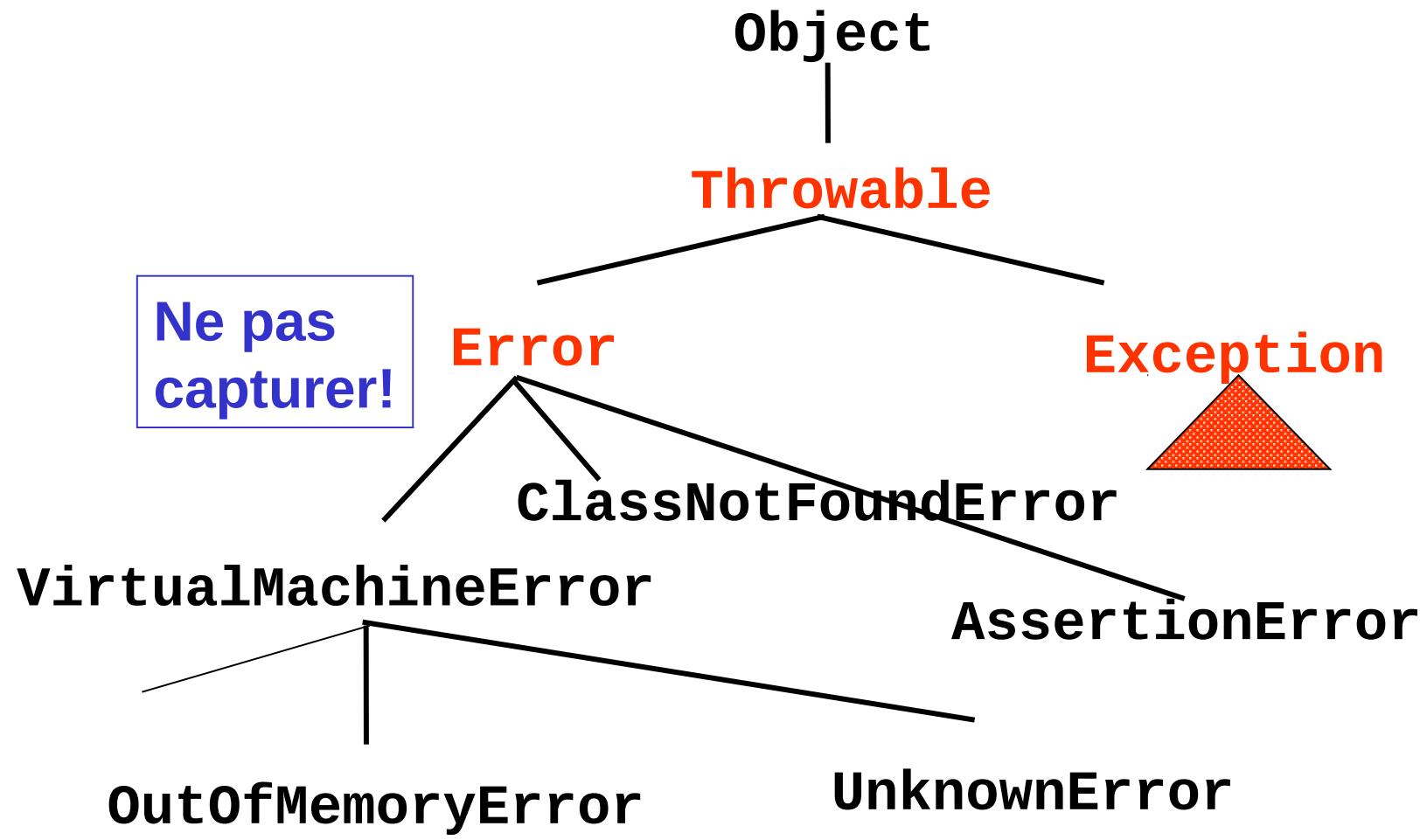


- **Attributs** :

message d'erreur (une **String**)  
état de la pile des appels

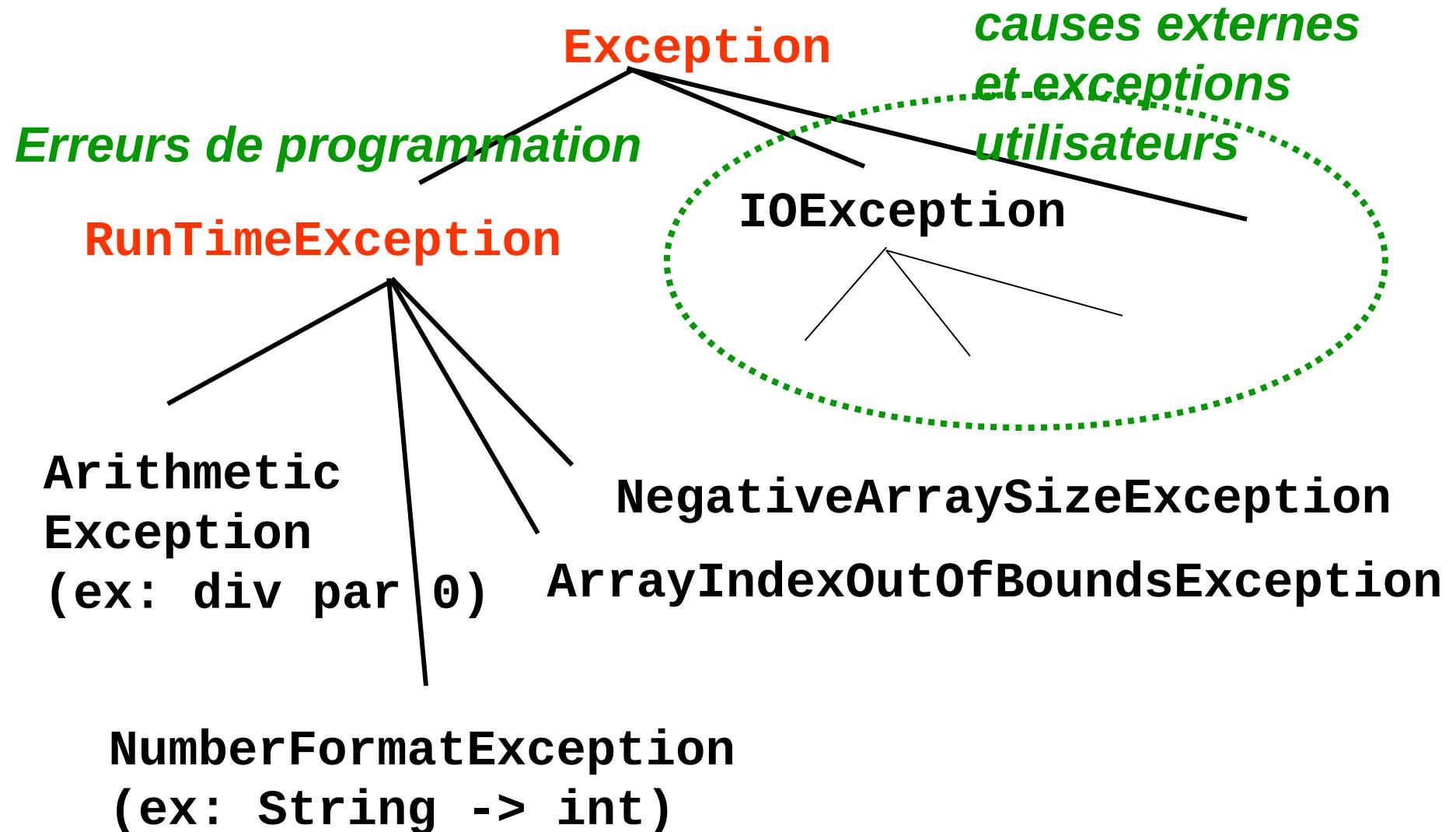
- **Méthodes** :

```
public Throwable()  
public Throwable(String message)  
public String getMessage()  
public void printStackTrace()
```



Error = problème de la machine virtuelle :

- Erreurs internes
- manque de ressources



# Méthodes génératrices d'exceptions

- Toute méthode doit **déclarer** les exceptions qu'elle est susceptible de lancer/transmettre

*classe java.io.BufferedReader*

```
public final String readLine()  
    throws IOException  
{.....}
```

- ... sauf si ce sont des **RuntimeException** ou **Error**

*classe java.lang.Integer*

```
public static int parseInt(String s)  
    throws NumberFormatException  
{.....}
```

**Pas obligatoire**

# Philosophie générale

## Exceptions hors contrôle

- les **Error** car leur traitement ne nous est pas accessible
- les **Runtime** car on n'aurait pas dû les laisser survenir

## Exceptions sous contrôle

- Toutes les autres !

# Méthodes génératrices d'exceptions

## Signalement

### *Exemple*

```
public final String readLine()  
    throws IOException  
{  
    if (...) throw new IOException();  
}
```



- crée un objet d'une certaine classe d'exception (ici IOException)
- signale (lève) cette exception : **throw**

## Exceptions dans une classe utilisateur : Point

```
public class Point
{ private int x, y; //coordonnées

    public Point(int x, int y)
    {.....}
    public String toString()
    { return x + " " + y; }
    public void deplace (int dx, int dy)
        // ajoute dx et dy à x et y
        {.....}
} // fin classe
```

On ajoute la contrainte : un Point doit avoir des coordonnées **positives** ou **nulles**.

**Comment assurer le respect de cette contrainte?**

```
public Point(int x, int y)
{
    // si x < 0 ou y < 0, que faire?

    this.x = x;
    this.y = y;
}
```

```
public void deplace (int dx, int dy)
{
    // si (x + dx) < 0 ou (y + dy) < 0,
    // que faire?

    x += dx;
    y += dy;
}
```

# Une classe d'exception pour Point

```
public class PointCoordException  
    extends Exception  
{  
    public PointCoordException()  
    { super(); }  
  
    public PointCoordException(String s)  
    { super(s); }  
}
```

On pourrait aussi créer une hiérarchie de classes d'exception pour Point

```
public Point(int x, int y)  
  
{  
    // si x < 0 ou y < 0,  
    // générer une PointCoordException  
  
    this.x = x;  
    this.y = y;  
}
```

```
public Point(int x, int y)
            throws PointCoordException

{
    if ((x < 0) || (y < 0))
        throw new PointCoordException
            ("création pt invalide "+ x + ' ' + y);

    this.x = x;
    this.y = y;
}
```

```
public void deplace (int dx, int dy)
{
    // si (x + dx) < 0 ou (y + dy) < 0,
    // générer une PointCoordException

    x += dx;
    y += dy;
}
```

```
public void deplace (int dx, int dy)
                     throws PointCoordException

{
    if ((x + dx < 0) || (y + dy < 0))
        throw new PointCoordException
            ("déplacement invalide "+dx+' '+dy);

    x += dx;
    y += dy;
}
```

# Capture versus transmission d'exception

```
public static int lireEntier()
{
    BufferedReader clavier = new BufferedReader
        (new InputStreamReader(System.in));
    String s = clavier.readLine();
    int ilu = Integer.parseInt(s);
    return ilu;
}
```

Erreur de compilation : lireEntier() doit *capturer* l'exception susceptible d'être transmise par readLine() ou *déclarer* qu'elle peut transmettre une exception (la laisser passer)

# Solution 1 : la laisser passer

```
public static int lireEntier() *  
{.....} ajout à l'entête
```

- \* throws IOException
- \* throws IOException, NumberFormatException
- \* throws Exception *pas très informant !*

La clause throws doit "**englober**" tous les types d'exception à déclaration obligatoire susceptibles d'être transmis de la manière la plus spécifique possible

## Solution 2 : la capturer (et la traiter)

- 1) surveiller l'exécution d'un bloc d'instructions : **try**
- 2) capturer *des exceptions* survenues dans ce bloc : **catch**

```
public static int lireEntier()
{ BufferedReader clavier = .....;
try
{ String s = clavier.readLine();
  int ilu = Integer.parseInt(s);
}
catch (IOException e) {.....}
return ilu; /* *
}
* Erreur de compilation : ilu
inconnu
```

```
public static int lireEntier()
{ BufferedReader clavier = .....;
  int ilu = 0;

  try
  { String s = clavier.readLine();
    ilu = Integer.parseInt(s);
  }
  catch (IOException e) {}
  // on capture e mais traitement = rien
  return ilu;
}
```

Que se passe-t-il si :

- une exception est générée par **readLine** ? (**IOException**)
- par **parseInt** ? (**NumberFormatException**)

```
public static int lireEntier()
{ BufferedReader clavier = .....;
int ilu = 0;

try
{ String s = clavier.readLine(); // 1
ilu = Integer.parseInt(s); } // 2
catch (IOException e) {}
// on capture e mais traitement = rien

return ilu; // 3
}
```

Si une exception est générée par **readLine** :

**2** n'est pas exécuté ;

clause **catch** capture l'exception ;

l'exécution continue en **3** : ilu *retourné (avec valeur 0)*<sup>20</sup>

```
public static int lireEntier()
{ BufferedReader clavier = .....;
int ilu = 0;

try
{ String s = clavier.readLine(); // 1
    ilu = Integer.parseInt(s); } // 2
catch (IOException e) {}
// on capture e mais traitement = rien

return ilu; // 3
}
```

Si une exception est générée par **parseInt** (**NumberFormatException**) :  
aucune clause **catch** ne capture l'exception ;  
elle est donc transmise à l'**appelant** ;  
**3** n'est pas exécuté

# **try + une (ou plusieurs) clause(s) catch**

Si une exception est générée dans un bloc **try**,

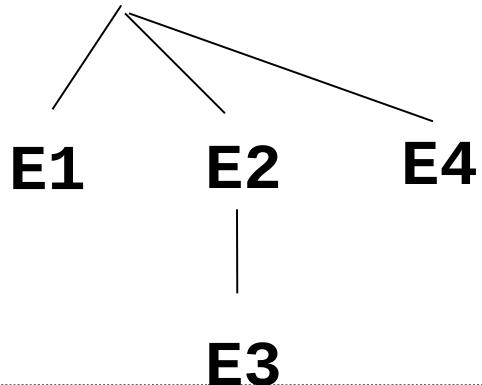
- l'exécution **s'interrompt**
- les clauses **catch** sont examinées dans l'ordre, jusqu'à en trouver une qui "englobe" la classe de l'exception
- s'il en existe une : le bloc du catch est exécuté, et l'exécution **reprend** juste après les clauses catch
- sinon : l'exception n'est pas capturée ; elle est donc **transmise** à l'appelant, et le reste de la méthode n'est pas exécuté

[On peut ajouter une clause **finally**  
par laquelle on passe toujours]

Nom :

Prénom :

## Exception



Quelles instructions sont exécutées si la partie (1) signale une exception de type :

-E1 :

-E2 :

-E3 :

-E4 :

méthode f(...)**throws E4**

{

**try**

{

**(1)** pouvant générer  
des E1, E2, E3 et E4

}

**catch(E1)**

{**(2)**}

**catch(E2)**

{**(3)**}

**finally**

{**(4)**}

**(5)**

}

## Try avec ressource (depuis Java 7)

- Constat avant Java 7 :
  - Certaines ressources (fichiers, flux, etc) doivent être fermées explicitement pour les libérer
- Parade avant Java 7 :
  - utilisation du bloc finally pour fermer le flux même si une exception est levée
- Ce qui impliquait :
  - Déclaration de la ressource en dehors du bloc try
  - La méthode close() de la ressource peut lever une IOException à gérer (autre bloc try/catch ou propagation)

## Try avec ressource (depuis Java 7)

- Depuis Java 7, définition possible d'une ressource avec l'instruction try
- La ressource sera automatiquement fermée à la fin de l'exécution du bloc try.

## Try avec ressource (depuis Java 7)

```
try {  
    try (BufferedReader bufferedReader = new  
        BufferedReader(new FileReader("myFile.txt"))) {  
        String line=null;  
        while ((line = bufferedReader.readLine()) != null) {  
            System.out.println(line);  
        }  
    }  
} catch (IOException ioe) {  
    ioe.printStackTrace();  
}
```

bufferedReader sera fermé proprement à la fin normale ou anormale des traitements → appel automatique à close.

## Try avec ressource (depuis Java 7)

- Ce qu'on peut mettre comme ressource dans le try :
  - Un `java.lang.AutoCloseable`
- `java.lang.AutoCloseable` définit une seule méthode : `close()` qui lève une exception de type `Exception`
- `java.io.Closable` extends `AutoCloseable`
- Le `close` de `Closable` lève une `IOException`

## Try avec ressource (depuis Java 7)

- Il est possible de créer de nouveaux types de ressources autoclosable → implémenter AutoClosable
- La ressource doit être déclarée et définie dans le try
- Possibilité de spécifier plusieurs ressources : try (Ressource a= ... ; Ressource b= ... ; Ressource c=...) → fermeture automatique de c, puis b, puis a

## Attraper plusieurs types d'exceptions (depuis Java 7)

```
try {  
    ...  
}  
catch(IOException | SQLException ex) {  
    ex.printStackTrace();  
}
```

# Retour à lireEntier()

Essayons de trouver une bonne  
façon de gérer les erreurs

```
public static int lireEntier()
{ BufferedReader clavier = .....;
int ilu = 0;

try
{ String s = clavier.readLine();
ilu = Integer.parseInt(s);
}

catch (Exception e) {}
// on capture e mais traitement = RIEN

return ilu;
}
```

Qu'en penser...?

## Problème...

L'erreur n'est pas **vraiment** réparée:

si 0 est retourné,

l'appelant ne peut pas savoir que ça ne correspond pas **forcément** à une **valeur saisie**

**Si on ne sait pas comment traiter une exception, il vaut mieux ne pas l'intercepter**

**Trouvons un traitement plus approprié ...**

```
public static int lireEntier()throws IOException
{ BufferedReader clavier = ....;
  int ilu = 0;
  boolean succes = false ;
  while (! succes)
  {
    try
    { String s = clavier.readLine();
      ilu = Integer.parseInt(s);
      succes = true;
    }
    catch (NumberFormatException e)
    { System.out.println("Erreur : " + e.getMessage());
      System.out.println("Veuillez recommencer ... ");
    }
  } // end while

  return ilu;
}
```

## **lireEntier()**

## **Changer de niveau d'abstraction**

```
public static int lireEntier()  
  
throws IOException, MauvaisFormatEntierException  
{ BufferedReader clavier = .....;  
int ilu = 0;  
  
try  
{ String s = clavier.readLine();  
ilu = Integer.parseInt(s);  
}  
catch (NumberFormatException e) {  
    throw new MauvaisFormatEntier();}  
  
return ilu;}
```

**L'erreur de bas-niveau retournée est interceptée et transformée en erreur du niveau de lireEntier()**

A ne pas faire

## Remplacer un test par la génération d'une exception

tableau d'entiers Tab de taille t

Problème : calculer l'indice i du premier 0 de Tab s'il existe (sinon i est affecté de -1)

**NON !**

```
int i = 0;  
try  
{ while (Tab[i] != 0) i++; }  
  
catch(ArrayIndexOutOfBoundsException e)  
{ i = -1; }
```

# A ne pas faire

## Cacher les exceptions pour éviter des erreurs de compilation

```
public void f()                                NON !
{
    try
    { ..... // ici le corps normal de f
    }

    catch (Exception e) {}

}
```

# A ne pas faire

## Chercher à traiter des exceptions à tout prix

```
public void f (String nomFichier)  
{  
    On essaye d'ouvrir le fichier dont le  
    nom est passé en paramètre  
    Si une FileNotFoundException surgit,  
    que faire?  
}
```

**A faire :** transmettre l'exception à l'appelant, jusqu'à arriver à la méthode qui a *décidé* du nom de fichier

**Exercice** : soit une méthode qui utilise la classe Point

```
public Rectangle CreerRect(int x1, int y1,  
                           int x2, int y2)  
{  
    Point p1 = new Point(x1, y1);  
    Point p2 = new Point(x2, y2);  
  
    Rectangle r = new Rectangle(p1, p2);  
  
    return r;  
}
```

Quelle(s) attitude(s) cette méthode peut-elle adopter face aux PointCoordException susceptibles d'être générées?

# Retour sur la conception

Comment déterminer les exceptions/assertions :

- invariants de classe
  - l'âge d'une personne est compris entre 0 et 140
  - une personne mariée est majeure
- préconditions
  - dépiler() seulement si pile non vide
- postcondition
  - après empiler(a), l'élément est dans la pile
- abstraction/encapsulation des exceptions des parties ou des éléments de l'implémentation
  - Point mal formé --> Rectangle mal formé
  - tableau interne de pile plein --> impossible d'empiler

# Retour sur la conception

## Assertions

Erreurs de logique du programme

Pour la mise au point du programme

## Exceptions

Erreurs imprévisibles,

Hors du contrôle du programmeur

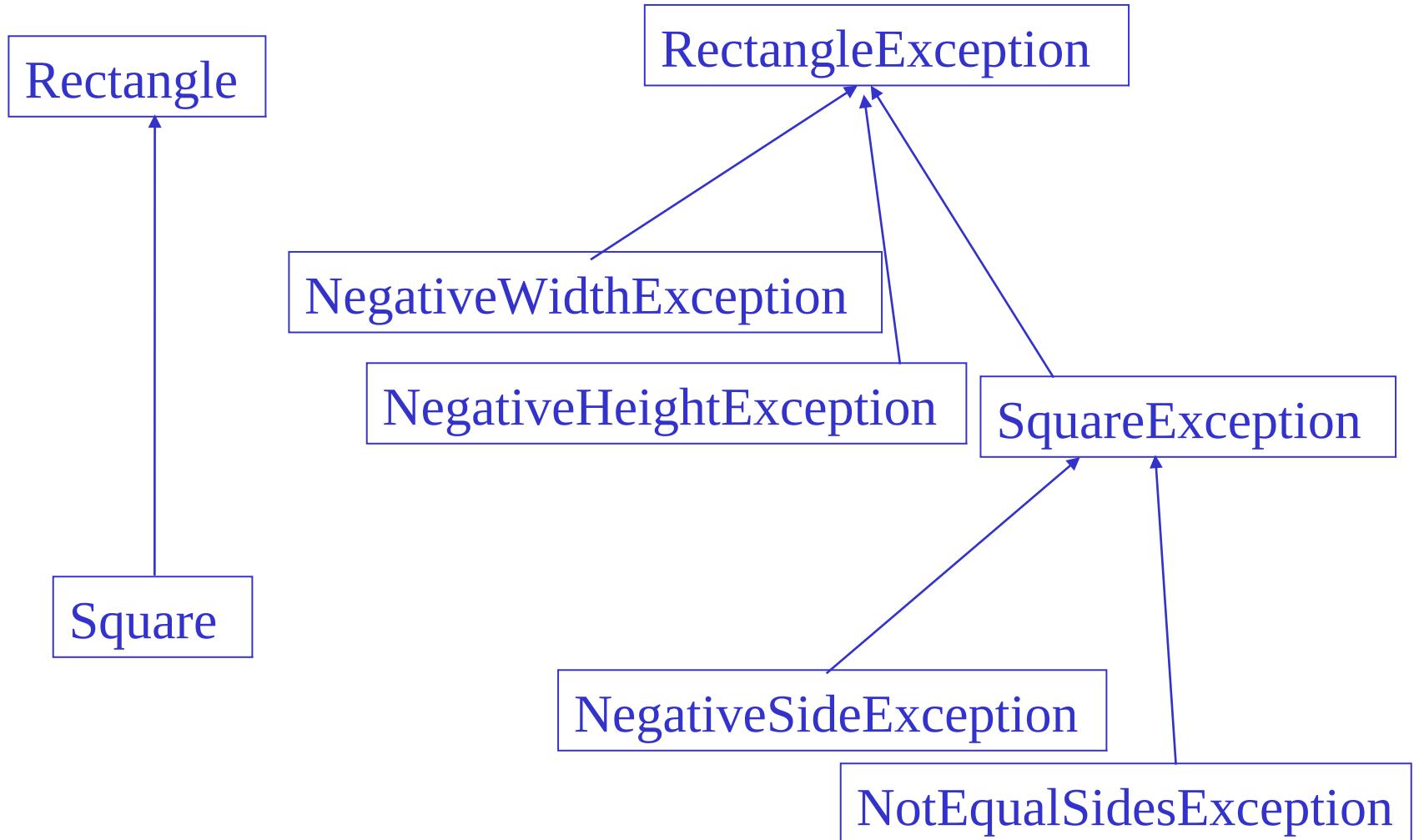
que l'on doit donc accepter

et gérer à l'exécution

# Retour sur la conception

Comment organiser les exceptions :

- une racine pour les exceptions associées à une classe
  - RectangleException, SquareException
- sous la racine, exceptions pour
  - les invariants de classes
  - les pré- et post-conditions
- les exceptions des sous-classes s'organisent sous les exceptions de la classe



# Redéfinition de méthodes

## (règles assurant la substituabilité)

Dans la redéfinition:

- Ajouter une déclaration n'est pas possible
- Retirer une exception est possible
- Spécialiser une exception est possible

Dessiner() throws RectangleException, IOException

Peut être redéfinie en

Dessiner () throws NegativeWidthException

# Introspection et Annotations

Manipulation  
de code source Java

# Introspection

# Principe

- Classes et méthodes permettant
  - Accès à l'information sur les classes
    - attributs
    - méthodes
    - constructeurs
  - Manipulation des objets de ces classes
    - modification d'attributs
    - appel de constructeurs
    - appel de méthodes
- Pendant l'exécution
- Limites en Java : pas de modification des classes, ex. ajout d'attributs ou de méthodes

# Utilisation *réalisation de*

- Débogueurs
- Interprètes
- Inspecteurs d'objets
- Navigateur de classes (class browsers)
- Services particuliers, ex.
  - Sérialization (sauvegarde d'objets)
  - Editeurs d'objets
  - Intercession (interception d'appels)

# Principales classes

- `java.lang`
  - `Class<T>` classe
- `java.lang.reflect`
  - `Field` attribut
  - `Constructor<T>` constructeur
  - `Method` méthode

# Principales classes

- **Class<T>**
  - le type de String.class est Class<String>
  - ses fields, constructors, methods, interfaces, classes, ..
- **Field**
  - son type, sa classe, sa valeur pour un objet, ..
- **Constructor<T>**
  - ses paramètres, exceptions, ..
- **Method**
  - ses paramètres, type de retour, exceptions, ..

# Contextes d'utilisation

Sans volonté d'exhaustivité, quelques exemples classiques d'utilisation ...

- Inspection des méthodes
- Inspection d'objets
- Création d'objets selon des types non connus au préalable
- Appel de méthodes

# Inspection des méthodes

- Eléments utilisés
  - Class, Method
    - String `getName()`
  - Class
    - static Class `forName(String c)`
      - retourne l'instance représentant la classe nommée c
    - Method[] `getMethods()`
      - retourne les méthodes publiques de la classe
  - Method
    - Class `getReturnType()`
    - Class[] `getParameterTypes()`

# Inspection des méthodes

```
abstract class Produit{
    private String reference,designation;
    private double prixHT;
    public Produit(){}
    public Produit(String r,String d,double p)
    {reference=r; designation=d; prixHT=p;}
    public String getReference(){return reference;}
    public void setReference(String r){reference=r;}
    public String getDesignation(){return designation;}
    public void setDesignation(String d){designation=d;}
    public double getPrixHT(){return prixHT;}
    public void setPrixHT(double p){prixHT=p;}
    abstract public double leprixTTC();
    public String infos(){return getReference()+" "+getDesignation()+" "+leprixTTC();}
}
```

# Inspection des méthodes

```
class ProduitTNormal extends Produit
{ public ProduitTNormal(){}
    public ProduitTNormal(String r,String d,double p)
        {super(r,d,p);}
    public double leprixTTC()
        {return getPrixHT() * 1.196;}
}
class Livre extends ProduitTNormal
{ private String editeur;
    public Livre(){}
    public Livre(String r,String d,double p,String e)
        {super(r,d,p);editeur=e;}
    public String getEditeur(){return editeur;}
    public void setEditeur(String e){editeur=e;}
    public String infos(){return super.infos()+
        "+getEditeur();}
}
```

# Inspection des méthodes

```
package Exemples;
```

```
import java.lang.reflect.*;
```

```
// Class est dans java.lang
```

```
// Method est dans java.lang.reflect
```

# Inspection des méthodes

```
public class TestReflexion
{
    public static void afficheMethodesPubliques(Class cl)
    {
        Method[] methodes = cl.getMethods();
        for (int i=0; i<methodes.length; i++)
            {Method m = methodes[i];
             String m_name = m.getName();
             Class m_returnType = m.getReturnType();
             Class[] m_paramTypes = m.getParameterTypes();
             System.out.print(" "+m_returnType.getName()+
                               " "+m_name + "(");
             for (int j=0; j<m_paramTypes.length; j++)
                 System.out.print(" "+m_paramTypes[j].getName());
             System.out.println(")");
    }
.... }
```

# Inspection des méthodes

```
public class TestReflexion
{
    ....
    public static void main(String[] argv)
        throws java.lang.ClassNotFoundException
    {
        System.out.println("Saisir un nom de classe");
        Scanner s=new Scanner(System.in);
        String nomClasse = s.nextLine();
        Class c = Class.forName(nomClasse);
        TestReflexion.afficheMethodesPubliques(c);
    }
}//fin TestReflexion
```

# Inspection des méthodes

Saisir un nom de classe

```
<< Exemples.Livre  
>> java.lang.String getEditeur()           Livre  
>> void setEditeur( java.lang.String)  
>> java.lang.String infos()  
>> double leprixTTC()                     ProduitTNormal  
>> double getPrixHT()                      Produit  
>> java.lang.String getReference() ....  
>> int hashCode()                          Object  
>> java.lang.Class getClass()  
>> boolean equals( java.lang.Object)  
>> java.lang.String toString() .....
```

# Inspection des objets

- Eléments utilisés
  - Object
    - Class `getClass()`
      - retourne la classe de l'objet
  - Class
    - String `getName()`
    - Field  `getField(String n)`
      - retourne l'attribut nommé *n*
  - Field
    - Object `get(Object o)`
      - retourne la valeur de l'attribut pour l'objet *o*

# Inspection des objets

```
Produit p = new Livre("X23", "Paroles de  
Prévert", 25, "Folio");  
System.out.println(p.getClass().getName());
```

>> Exemples.Livre

```
p = new Aliment("A21", "Pain d'épices",  
                12, "BonMiel");  
System.out.println(p.getClass().getName());
```

>> Exemples.Aliment

# Accès aux attributs *public*

```
// éditeur et prixHT ont été déclarés public pour cette partie  
Livre p = new Livre("X23","Paroles de
```

```
    Prévert",25,"Folio");
```

```
Class p_class = p.getClass();
```

```
Field f1_p = p_class.getField("éditeur");
```

```
Object v_f1_p = f1_p.get(p);
```

```
Field f2_p = p_class.getField("prixHT");
```

```
Object v_f2_p = f2_p.get(p);
```

```
System.out.println("v_f1_p="+v_f1_p+  
                    v_f2_p)+"v_f2_p);
```

>> v\_f1\_p=Folio v\_f2\_p=25.0

# Accès aux attributs privés

```
f1_p.setAccessible(true);
```

- méthode héritée de  
AccessibleObject

# Créer des objets

- Eléments utilisés
  - Class
    - static Class `forName(String)`
    - Constructor `getConstructor()`;
      - retourne le constructeur sans paramètres
  - Constructor
    - Object `newInstance()`
      - retourne un objet construit avec le constructeur

# Créer des objets

```
System.out.println("Livre ou Aliment ?");
```

```
Scanner s=new Scanner(System.in);
```

```
String nomClasse = s.next();
```

```
Object np;
```

```
// et maintenant on voudrait créer
```

```
// un livre ou un aliment
```

# Créer des objets

## code classique

```
System.out.println("Livre ou Aliment ?");  
Scanner s=new Scanner(System.in);  
String nomClasse = s.nextLine();Object np;
```

```
if (nomClasse.equals("Exemples.Livre"))  
    np = new Livre();  
else if (nomClasse.equals ("Exemples.Aliment"))  
    np = new Aliment();  
else if ...
```

- Pb extensibilité : ajout de classe, modification de nom de classe implique :  
**modification de code**

# Créer des objets avec la réflexion

```
System.out.println("Livre ou Aliment ?");
```

```
Scanner s=new Scanner(System.in);
```

```
String nomClasse = s.nextLine();
```

```
Object np;
```

```
Class c = Class.forName(nomClasse);
```

```
Constructor constructeur=c.getConstructor();
```

```
np = constructeur.newInstance();
```

```
np.saisie(..); ...
```

# Créer des objets avec la réflexion

- Pour appeler un constructeur prenant des paramètres

Constructor constructeur =

```
c.getConstructor(String.class,  
                String.class,  
                double.class,  
                String.class);
```

np = constructeur.newInstance

```
("xx","Paroles",12,"Folio");
```

# Appeler des méthodes

- Eléments utilisés
  - Class
    - Method `getMethod(String n)`
      - retourne la méthode nommée *n*
    - Method
      - Object `invoke(Object)`
        - appelle la méthode sur l' objet o

# Appeler des méthodes

```
System.out.println("Méthodes existantes sur np");
```

```
TestReflexion.afficheMethodesPubliques(c);
```

```
>> ...
```

```
>> java.lang.String infos()
```

```
>> double leprixTTC()
```

```
>> double getPrixHT()
```

```
>> ....
```

```
System.out.println("Quelle méthode sans argument voulez-vous appeler ?");
```

```
String nomMeth = s.next();
```

```
<< leprixTTC
```

```
Method meth = c.getMethod(nomMeth);
```

```
Object resultat = meth.invoke(np);
```

```
System.out.println("resultat = "+resultat);
```

```
>> resultat = 14.352
```

# Appeler des méthodes avec des paramètres

```
meth = c.getMethod("setEditeur",  
                  String.class);
```

```
resultat = meth.invoke(np,"Gallimard");
```

```
System.out.println("nouvel objet = "+np);
```

# On peut aussi ...

- accéder aux modifiés
- connaître les super-classes, les interfaces
- créer et manipuler des tableaux
- créer des proxys de classes ou d'instances pour intercepter des appels et ajouter du comportement (ex. tracer automatiquement)

# ANNOTATIONS

# Annotations

- **informations** pour les programmes traités par des outils
  - éditeurs,
  - débogueurs,
  - outils de test, documentation, statistiques, refactoring, etc.
- **tags javadoc** : annotations spécialisées pour la documentation
- d'autres formes de tags peuvent être définis par les utilisateurs (programmeurs d'outils notamment)

# Utilisation dans javadoc

```
/**  
 * @deprecated As of JDK version 1.1,  
 * replaced by Calendar.get(Calendar.MONTH)  
 * as shown in {@link java.util.Calendar#get(int) get}  
 */
```

Produit dans la documentation :

**getMonth**

**public int getMonth()**

**Deprecated.**

***As of JDK version 1.1, replaced by***

***Calendar.get(Calendar.MONTH) as shown in [get](#)***

# Utilisation dans javadoc

```
/**  
 * @deprecated As of JDK version 1.1,  
 * replaced by Calendar.get(Calendar.MONTH)  
 * as shown in {@link java.util.Calendar#get(int) get}  
 */
```

- block tag `@deprecated` en début de ligne
- inline tag `{@link }` en milieu de ligne
- utilisé par le programme javadoc pour créer les pages html de la documentation

# Utilisation dans Eclipse

Une méthode privée inutilisée génère un warning

```
private void crypter()  
{/* à écrire plus tard*/ System.out.println("cryptage");}
```

Solution proposée par Eclipse

ajouter un tag pour faire disparaître ce warning

```
@suppresswarnings("unused")  
private void crypter()  
{/* à écrire plus tard*/ System.out.println("cryptage");}
```

Eclipse n'affichera plus de warning !

# Déclaration d'un type d'annotation

- type d' annotation = interface
- mot-clef *interface* précédé par @
- les méthodes
  - définissent des éléments
    - quand il est unique, l'élément s'appelle *value*
  - pas de paramètres
  - pas de clause throws
  - type de retour possible
    - TRP = types primitifs, String, Class, enums,
    - arrays de TRP
  - valeurs par défaut

# Définition d'un type d'annotation

```
/**  
 * Request-For-Enhancement(RFE)  
 * annoter un élément à améliorer  
 * dans la version suivante  
 */  
  
public @interface RequestForEnhancement  
{  
    int id();  
    String synopsis();  
    String engineer() default "[unassigned]";  
    String date() default "[unimplemented]";  
}
```

# Utilisation de l'annotation

~ se place comme un *modifier*

```
@RequestForEnhancement(  
    id = 23777,  
    synopsis = "Improve time complexity",  
    engineer = "Jack",  
    date = "31 oct 2009")  
  
public static  
    <T extends Comparable<? super T>>  
void sort(List<T> list)  
{ ... }
```

# interface **Annotation**

- C'est l'interface spécialisée par les annotations
- Ne pas l'étendre manuellement
- Méthodes
  - `Class<? extends Annotation> annotationType()`  
retourne le type d'annotation de cette annotation
  - `boolean equals(Object obj)`
  - `int hashCode()`
  - `String toString()`

# Types d'Annotation de l'API

- Annotations
  - Deprecated
  - Override
  - SuppressWarnings
- Certaines portent sur d'autres annotations
  - Inherited
  - Documented
  - Retention : décrit la portée (SOURCE, CLASS, RUNTIME)
  - Target : décrit la cible (TYPE, FIELD, METHOD, etc.)

# Annotation annotée

```
@Documented  
@Retention(value=RUNTIME)  
@Target(value=ANNOTATION_TYPE)  
public @interface Retention  
{ RetentionPolicy value(); }
```

# Interface AnnotatedElement

- Implémentée par AccessibleObject, Class, Constructor, Field, Method, Package

## Méthodes

<T extends Annotation> **getAnnotation**  
(Class<T> annotationType)

retourne l'annotation du type passé en paramètre (ou null)

Annotation[ ] **getAnnotations**( )

retourne les annotations attachées à l'élément (incluant héritées)

Annotation[ ] **getDeclaredAnnotations**( )

retourne toutes les annotations attachées à l'élément (propres)

boolean **isAnnotationPresent**

(Class<? extends Annotation>annotationType)

retourne vrai ssi une annotation du type passé en paramètre  
est attachée à l' élément

# Eléments pour un outil de test

- Objectif :
  - embarquer dans les classes des méthodes de test unitaire
  - annotation par les programmeurs de ces méthodes de test (pour les distinguer des autres)
  - l'outil de test utilise les annotations pour tester la classe

# Type d'annotation pour les méthodes de test

```
import java.lang.annotation.*;  
enum NiveauRisque {faible, moyen, eleve;}  
/**  
 * indique qu'une méthode est une méthode de test  
 * à utiliser sur des méthodes sans paramètre  
 */  
@Retention(RetentionPolicy.RUNTIME)  
@Target(ElementType.METHOD)  
public @interface Test  
    {NiveauRisque risque();}
```

# Classe en développement

```
class Foo {  
    @Test(risque=NiveauRisque.faible)  
        public static void m1()  
            {System.out.println("m1");}  
    public static void m2() {System.out.println("m2");}  
    @Test(risque=NiveauRisque.moyen)  
        public static void m3()  
            {throw new RuntimeException("Boom");}  
    public static void m4() {System.out.println("m4");}  
    @Test(risque=NiveauRisque.moyen)  
        public static void m5()  
            {System.out.println("m5");}  
    public static void m6() {System.out.println("m6");}  
    @Test(risque=NiveauRisque.eleve)  
        public static void m7()  
            {throw new RuntimeException("Crash");}  
    public static void m8() {System.out.println("m7");}  
}
```

# Une classe de l'outil de test

```
import java.lang.annotation.*;
import java.lang.reflect.*;
public class TestAnnotations
{ public static void main(String[] args) throws Exception

{int passed = 0, failed = 0;
for (Method m :
    Class.forName(args[0]).getMethods())
{if (m.isAnnotationPresent(Test.class) &&
    (m.getAnnotation(Test.class)).risque() !=NiveauRisque.faible)
{try {m.invoke(null); passed++;}
catch (Throwable ex)
    {System.out.println("Test "+m+" failed:" +ex.getCause());
     failed++;}
}
System.out.println("Passed: "+passed+" Failed "+failed);
}
```

# Exécution

```
>>> java TestAnnotations Foo
```

```
Test public static void Foo.m3() failed: java.lang.RuntimeException: Boom  
m5
```

```
Test public static void Foo.m7() failed: java.lang.RuntimeException: Crash  
Passed: 1 Failed 2
```

# Pour aller plus loin avec les annotations

- Annotation Processing Tool (commande apt)
- s'utilise indépendamment d'une exécution
- exécute des traitements d'annotations sur un ensemble de fichiers source annotés
- Plus complexe (patron de conception Visiteur)
- Pas encore stabilisé (API mirror)
- <http://gfx.developpez.com/tutoriel/java/annotation/>
  - ramasser les Todo d'une classe
  - créer un fichier les contenant
- <http://www.javalobby.org/java/forums/t17876>

## Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

# Diagrammes dynamiques en UML

## diagrammes de séquence, de collaboration, d'état-transition, d'activité

LIRMM / Université de Montpellier 2

19 octobre 2014

# Introduction

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités

- Diagrammes statiques (diagrammes d'instances et diagrammes de classe)
  - Structure d'un système
  - Signatures de méthodes
- Diagrammes dynamiques
  - Dynamique du système

# Sommaire

Diagrammes  
dynamiques

Les  
diagrammes  
de séquence

La ligne de  
vie

Les messages

Composition  
de fragments  
de  
diagrammes  
de  
séquence

Les machines  
à états

États et  
transitions

États initial  
et final

États  
composites

Pseudo-états

Les  
diagrammes  
d'activités

## 1 Les diagrammes de séquence

- La ligne de vie
- Les messages
- Composition de fragments de diagrammes de séquence

## 2 Les machines à états

- États et transitions
- États initial et final
- États composites
- Pseudo-états

## 3 Les diagrammes d'activités

# Sommaire

Diagrammes  
dynamiques

Les  
diagrammes  
de séquence

La ligne de  
vie

Les messages

Composition  
de fragments  
de  
diagrammes  
de séquence

Les machines  
à états

États et  
transitions

États initial  
et final

États  
composites

Pseudo-états

Les  
diagrammes  
d'activités

## 1 Les diagrammes de séquence

- La ligne de vie
- Les messages
- Composition de fragments de diagrammes de séquence

## 2 Les machines à états

- États et transitions
- États initial et final
- États composites
- Pseudo-états

## 3 Les diagrammes d'activités

# Les diagrammes de séquence

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

- Les diagrammes de séquence permettent de représenter les interactions entre des instances particulières. Un diagramme met en jeu :
  - des instances, et éventuellement des acteurs,
  - des messages échangés par ces instances. Un message définit une communication entre instances. Ce peut être par exemple l'émission d'un signal, ou l'appel d'une opération.
- Le diagramme de séquence permet d'insister sur la chronologie des interactions : le temps s'écoule grossièrement du haut vers le bas.
- Les diagrammes de séquence ont été profondément modifiés lors du passage d'UML1.x à UML2.0, et à l'heure actuelle, peu de gens utilisent la nouvelle notation

# Exemple

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités

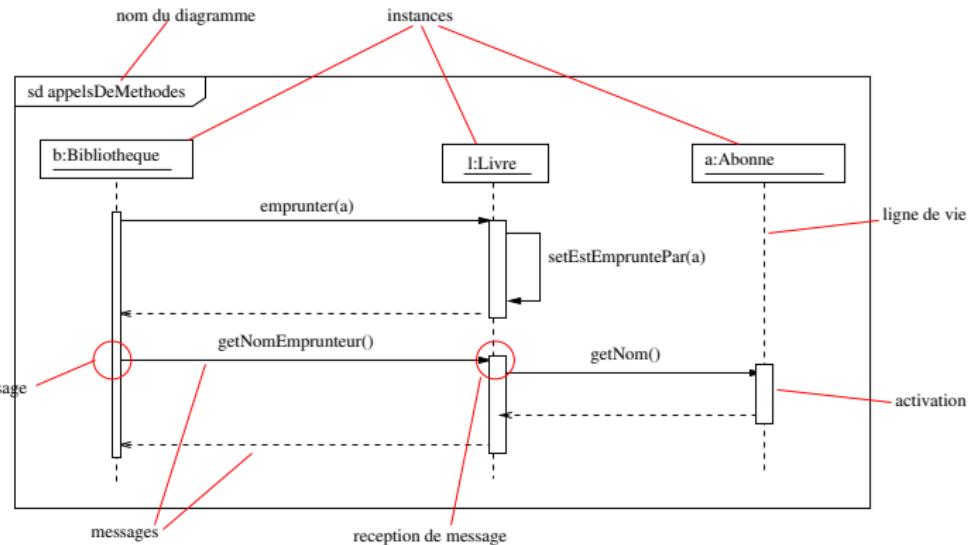


Figure: Premier exemple de diagramme de séquence

# La ligne de vie

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

- À chaque instance est associée une ligne de vie, qui représente la vie de l'objet.
- Les événements survenant sur une ligne de vie (réception de message ou envoi de message) sont ordonnés chronologiquement.
- La ligne de vie est représentée par une ligne pointillée quand l'instance est inactive, et par une boîte blanche ou grisée quand l'instance est active.
- Quand une instance est détruite, on stoppe la ligne de vie par une croix.

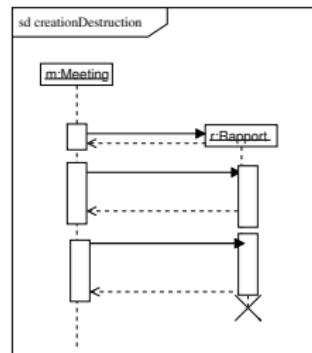


Figure: Ligne de vie

# Les messages

Diagrammes dynamiques

Les diagrammes de séquence  
La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités

- Les messages sont représentés par des lignes fléchées.
- À chaque extrémité de la ligne fléchée correspond un événement (réception ou envoi). Le sens de la flèche permet de déterminer dans quel sens va le message.
- Messages synchrones et asynchrones (voir Figure 3).

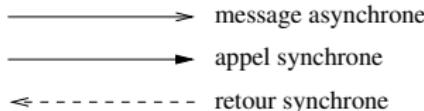


Figure: Messages

# Exemple

## Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

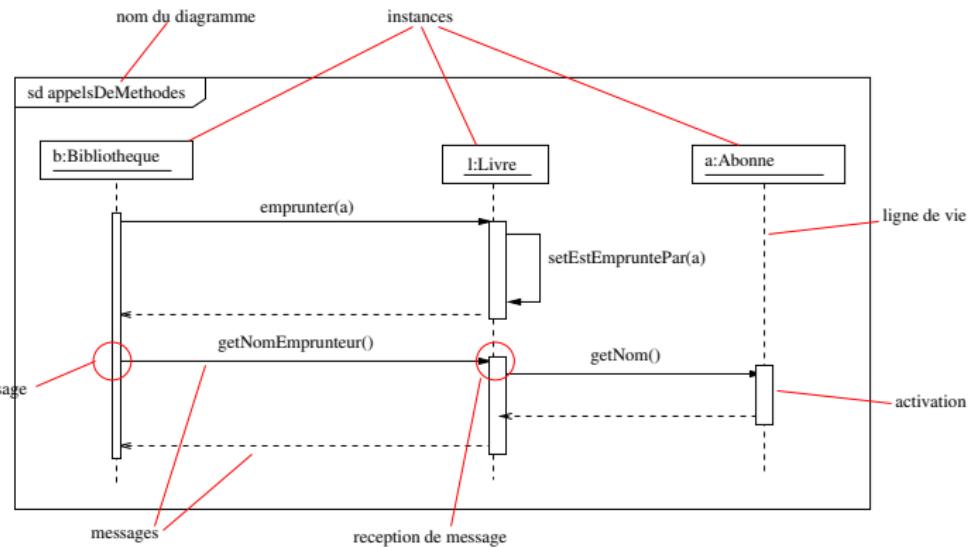


Figure: Premier exemple de diagramme de séquence

# Syntaxe des noms de message

Diagrammes dynamiques

Les diagrammes de séquence  
La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités

La syntaxe pour le nom d'un message est :

([attribut =] signal-ou-NomOpération [(liste-arguments)])[: valeur-retour]) | \*

où la syntaxe pour un argument est :

([nomParam =] valeur-argument) | (attribut = nomParamOut [: valeurArgument]) | -

\* signifie : n'importe quel type de message

- signifie : paramètre indéfini

Par exemple, on peut avoir les noms de message suivants :

- `getAge()`
- `getAge() :12`
- `age=getAge() :12`
- `setAge(age=15)`
- `setAge(-)`

# Exemple d'appel de méthode

## Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

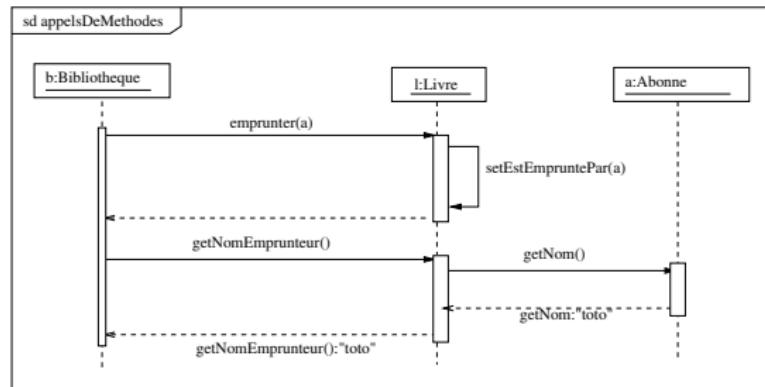


Figure: Appels de méthodes

Les diagrammes de séquence ne sont pas à concevoir indépendamment des autres diagrammes, comme par exemple le diagramme de classes.

# Composition de fragments de diagrammes de séquence

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

- possible depuis la version 2.0 d'UML
- existence de plusieurs opérateurs de composition

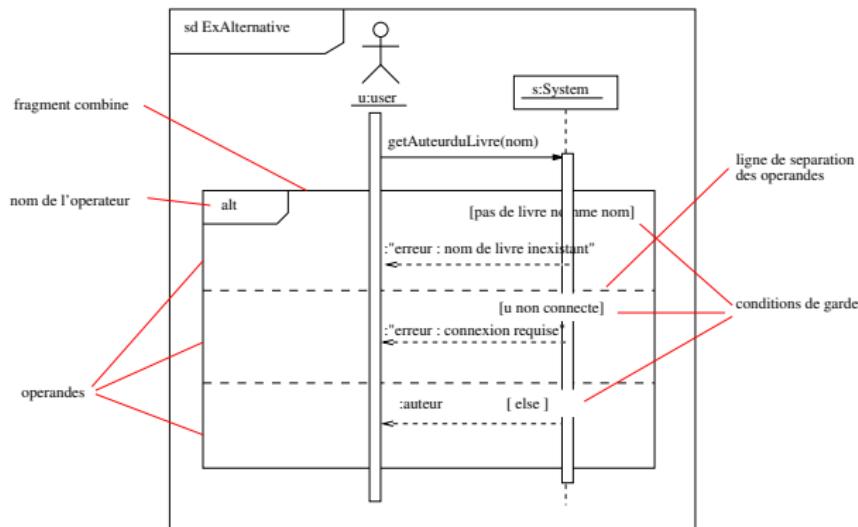


Figure: Opérateurs de composition et fragments combinés

# Alternative et optionnalité

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

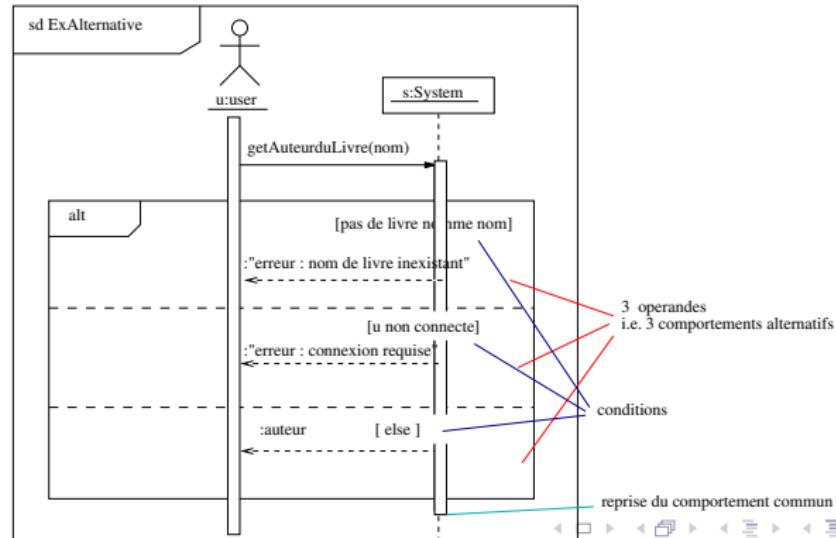
États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

- alternative (noté alt) permet de représenter le choix (exclusif) entre plusieurs comportements
- optionnalité (noté opt) permet de représenter un comportement qui n'a lieu que si une condition de garde est vraie



# Composition parallèle

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

- L'opérateur de composition parallèle (noté par) permet de spécifier des comportements qui peuvent avoir lieu en parallèle les uns des autres. Cet opérateur est n-aire.
- Quand un comportement A est en parallèle avec un comportement B, l'ordre partiel des événements de A et de B est conservé.
- Raccourci syntaxique : *corégion*

# Composition séquentielle faible

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

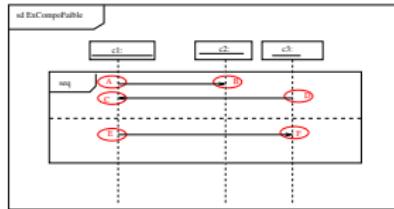


Figure: Composition séquentielle faible (seq)

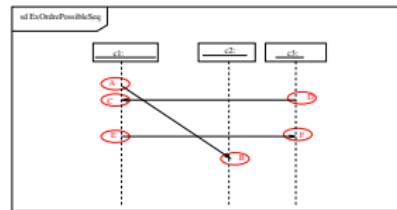


Figure: Un diagramme de séquence pouvant en résulter

- $A \prec B$
- $D \prec C$
- $E \prec F$
- $C \prec E$
- $D \prec F$

# Composition séquentielle forte

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

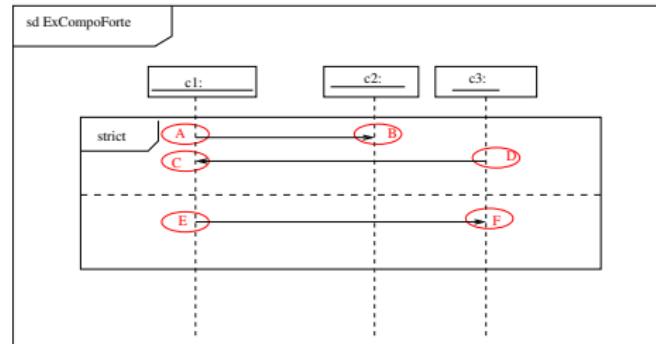


Figure: Composition séquentielle forte (strict)

- A  $\prec$  B
- D  $\prec$  C
- E  $\prec$  F
- C  $\prec$  E
- D  $\prec$  F
- B  $\prec$  E

# Boucle

Diagrammes dynamiques

Les diagrammes de séquence  
La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

L'opérateur `loop` permet d'itérer des comportements. On doit pour cela spécifier :

- le nombre minimum `minInt` de tours de boucles,
- le nombre maximum `maxInt` de tours de boucle (\* signifie infini),
- une condition de garde,
- une unique opérande représentant le comportement sur lequel on boucle.

Syntaxe de la boucle :

`loop[ (minInt [ , maxInt ] ) ]`

Par défaut, `minInt=0` et `maxInt=*`.

# Exemple de boucle

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

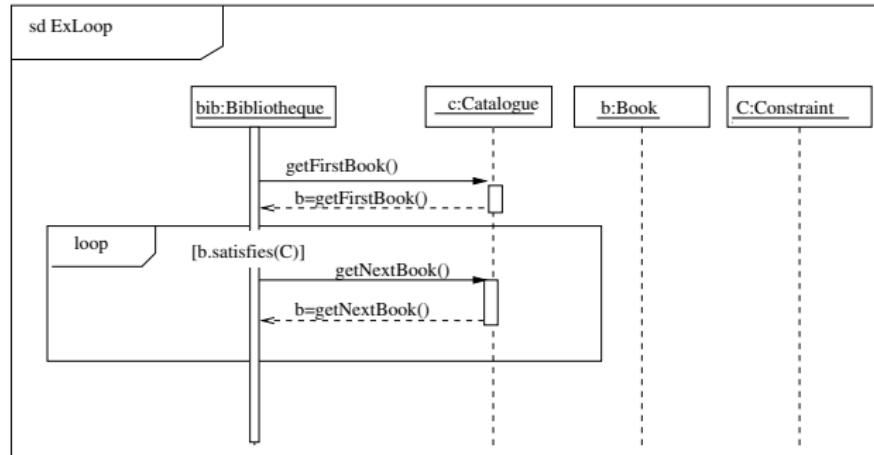


Figure: Boucles dans les diagrammes de séquence

# Question de cours

## Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

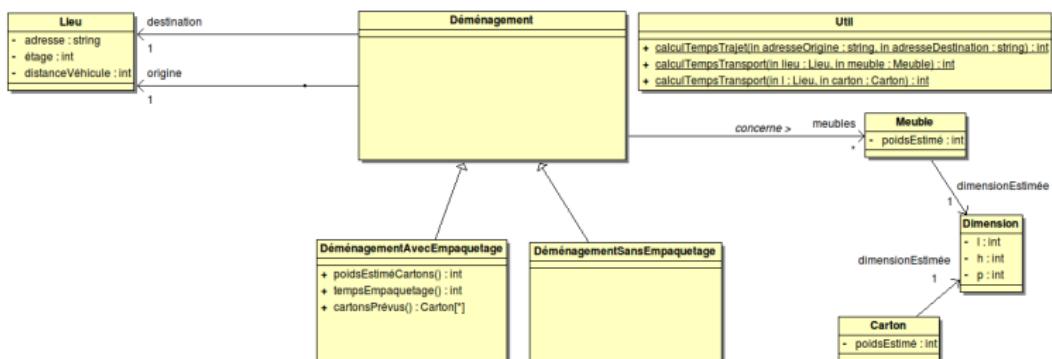
États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités



→ Une méthode qui calcule le temps de déménagement d'un déménagement avec empaquetage : temps de trajet+temps d'empaquetage + distance au véhicule de l'origine et de la destination.

# Sommaire

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

## 1 Les diagrammes de séquence

- La ligne de vie
- Les messages
- Composition de fragments de diagrammes de séquence

## 2 Les machines à états

- États et transitions
- États initial et final
- États composites
- Pseudo-états

## 3 Les diagrammes d'activités

# Les machines à états

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités

Les machines à états, aussi appelés diagrammes d'état-transition, servent à modéliser la dynamique d'un sous-système, souvent d'une classe. Une machine à états décrivant le comportement d'une classe décrit en fait la dynamique de toutes ses instances à la réception ou à l'envoi de messages.

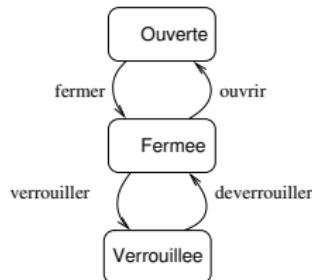


Figure: Diagramme d'état-transition très simple pour une porte

# États et transitions

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

- Un état modélise une situation où un certain invariant (généralement implicite) est maintenu (la porte est fermée, un compte bancaire a un solde positif, ...)
- Transition : passage d'un état à un autre
  - Il peut y avoir plusieurs événements déclencheurs possibles, auquel cas on les liste tous (en les séparant par des virgules).
  - L'action peut être une affectation d'attribut, un appel de méthode, ...
  - Quand aucun événement déclencheur n'est spécifié, la transition est dite spontanée.

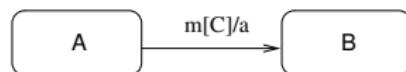


Figure: Une transition

# États initial et final

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

Un pseudo-état initial (noté graphiquement par un petit disque noir) représente un sommet qui est la source d'une seule transition vers l'état "par défaut" d'une machine à état ou d'un état composite. La transition initiale peut être munie d'une action.

L'état final matérialise le fait qu'une région (une machine à état ou une région d'état composite) est "terminée" (voir notation figure 14).



Figure: États initial et Final

# États composites

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie  
Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

Un état composite :

- soit contient une seule région
- soit est décomposé en 2 ou plusieurs régions orthogonales

Un état inclus dans une région d'un état composite est appelé un sous-état de cet état composite. C'est un sous-état direct quand il n'est pas contenu par un autre état, et sinon un sous-état indirect.

# Exemple de machine à état avec état composite

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

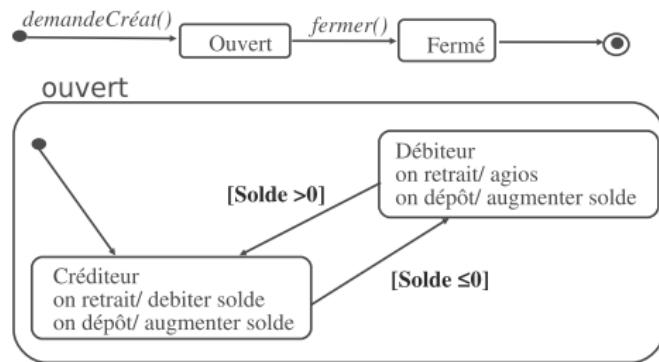


Figure: Exemple de machine à état avec état composite

# Exemple avec état à régions orthogonales

## Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

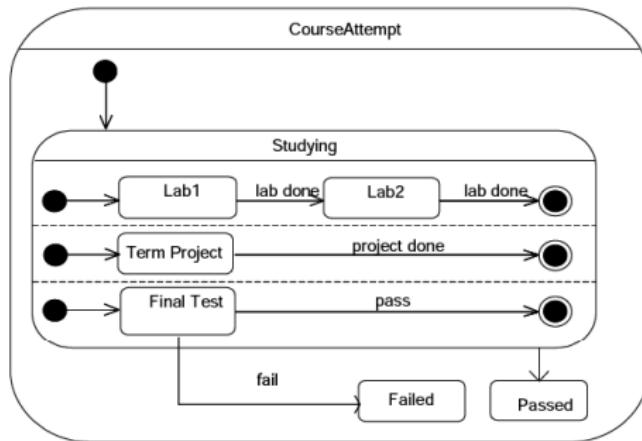


Figure: État composite orthogonal, extrait du document de spécification d'UML 2.0

# Comportement d'entrée et de sortie, comportement dans un état

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

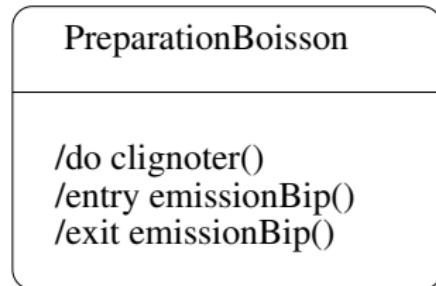


Figure: Actions d'entrée et de sortie des états, comportement dans les états

+ on evenement / action

# États Historiques

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités

Il existe des états dits “mémoire” qui permettent de rentrer dans un état composite dans le même sous-état que quand on en est sorti. Il y a deux états mémoire : historique superficiel et historique profond.

**Historique superficiel (Shallow history)** (noté H). L'historique superficiel représente le sous-état actif le plus récent (mais pas les sous-états de ce sous-état).

**Historique profond (Deep history)** (noté H\*). L'historique profond représente la configuration active la plus récente de l'état composite qui contient directement l'historique profond (c'est-à-dire la configuration active la dernière fois qu'on a quitté l'état composite).



Shallow history



DeepHistory

# Autres pseudo-états

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
**Pseudo-états**

Les diagrammes d'activités

Il existe d'autres pseudo-états comme les jonctions, les choix ou les branchements, nous ne les détaillerons pas ici.

# Question de cours

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

Nous étudions une montre très simple. Elle possède deux boutons : avance et mode. Le mode par défaut est le mode affichage. Quand on appuie une fois sur le bouton mode, la montre passe en mode de modification des heures. Chaque pression sur le bouton avance incrémente l'heure d'une unité. Quand on appuie une nouvelle fois sur le bouton mode, la montre passe en modification des minutes. Chaque pression sur le bouton avance incrémente les minutes d'une unité. Quand on appuie une nouvelle fois sur le bouton mode, la montre repasse en mode affichage.

→ Représentez le diagramme d'états de la montre.

# Sommaire

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

## 1 Les diagrammes de séquence

- La ligne de vie
- Les messages
- Composition de fragments de diagrammes de séquence

## 2 Les machines à états

- États et transitions
- États initial et final
- États composites
- Pseudo-états

## 3 Les diagrammes d'activités

# Les diagrammes d'activité

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages

Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

Les diagrammes d'activités permettent de représenter des flots de contrôle et de données. Ils permettent donc par exemple de représenter le comportement d'une opération ou d'un cas d'utilisation. Les diagrammes d'activité sont des graphes, avec différents types de nœuds et d'arcs. Ils mettent en jeu principalement :

- des nœuds actions
- des nœuds de contrôle permettant de spécifier l'enchaînement des actions (synchronisation, branchement, ...)
- des nœuds d'objet permettant de représenter les objets créés ou utilisés au cours d'une activité
- des arcs de transition permettant de relier les nœuds.

# Exemple

## Diagrammes dynamiques

Les diagrammes de séquence

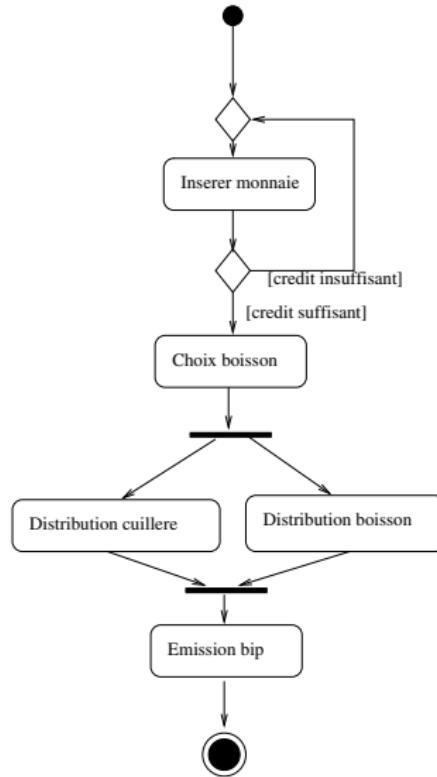
La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités



# Représentation de graphes de flot de contrôle

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités

- Nœud initial (Initial node)  . Point d'entrée pour invoquer une activité. Un jeton de contrôle est placé au nœud initial quand l'activité commence.
- Nœud de fin d'activité (Activity final node)  . Stoppe tous les flots dans une activité. Un jeton atteignant un nœud de fin d'activité fait avorter tous les flots en cours, l'activité est donc terminée et le jeton est détruit (ainsi que tous les jetons circulant dans l'activité).
- Nœud de fin de flot (Flow Final node)  . Termine un flot. Le nœud de flot final détruit les jetons y entrant.

# Représentation de graphes de flot de contrôle

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

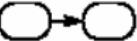
États et transitions

États initial et final

États composites

Pseudo-états

Les diagrammes d'activités

- Nœud d'action (Action node)  . Unité fondamentale de la fonctionnalité exécutable d'une activité. Une action s'exécute quand toutes les contraintes sur ses *flots de contrôle* entrants sont satisfaites (jonction implicite). L'exécution consomme les jetons de contrôle entrants puis présente un jeton sur chaque flot sortant (branchement implicite).
- Flot de contrôle (Control flow)  . Passage des jetons. Les jetons offerts par le nœud source sont offerts au nœud destination.

# Représentation de graphes de flot de contrôle

## Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

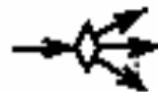
Les machines à états

États et transitions

États initial et final

États composites  
Pseudo-états

Les diagrammes d'activités



- Nœud de décision (Decision node) . Choix parmi les flots sortants. Chaque jeton arrivant sur un nœud de décision ne peut traverser qu'un seul flot sortant. Les jetons ne sont pas dupliqués. Ce sont les gardes sur les flots sortants qui permettent le choix (les gardes doivent assurer le déterminisme du choix).



- Nœud de branchement (Fork node) . Partage d'un flot en flots concurrents. Les jetons arrivant d'un branchement sont dupliqués sur les flots sortants.

# Représentation de graphes de flot de contrôle

Diagrammes dynamiques

Les diagrammes de séquence

La ligne de vie

Les messages  
Composition de fragments de diagrammes de séquence

Les machines à états

États et transitions  
États initial et final  
États composites  
Pseudo-états

Les diagrammes d'activités



- Nœud de jonction (Join node) . Synchronisation de plusieurs flots. Si un jeton de contrôle est offert sur chaque flot entrant, alors un jeton de contrôle est offert sur le flot sortant.



- Nœud de fusion (Merge node) . Rassemblement de plusieurs flots. Tous les jetons offerts sur les flots entrants sont offerts sur le flot sortant sans synchronisation.
- Partition d'activité (Activity Partition). Identifie des actions ayant une caractéristique commune. Les partitions



n'affectent pas le flot des jetons.

# Itérateurs et visite des collections (Java 1.7)

## Avant-goût des Streams (Java 1.8)

HLIN505  
Modélisation et programmation par objets 2

Université Montpellier 2

Octobre 2014

# Itérateurs

## Itérateur

Un itérateur est un objet qui permet :

- de visiter les éléments d'une collection un par un,
- plus généralement de visiter les éléments internes d'un autre objet complexe (qui est un composite).

## Patron de conception Iterator

Présenté dans le GOF

**Design Patterns : Elements of Reusable Object-Oriented Software**

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Published Oct 31, 1994 by Addison-Wesley Professional.

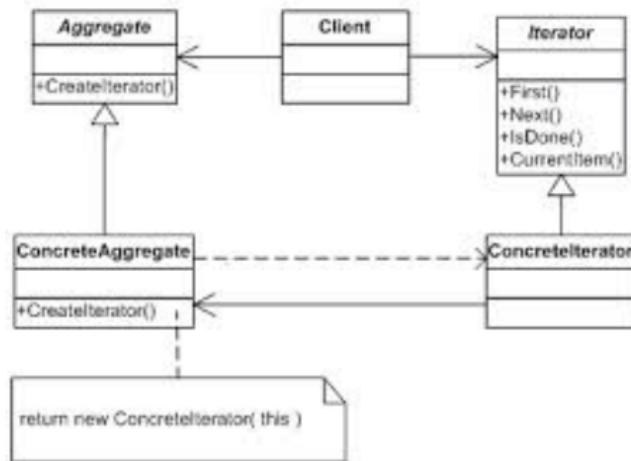
# Patron de conception Iterator

## Problème

- permettre à l'utilisateur d'un objet complexe (collection ou objet composite) de parcourir cette collection ou cet objet composite,
- au travers d'une interface uniforme (opérations de parcours standard),
- sans connaître les détails de l'implémentation,
- la structure interne de l'objet peut changer (ainsi que l'itérateur) sans que le programme utilisateur n'ait à changer.

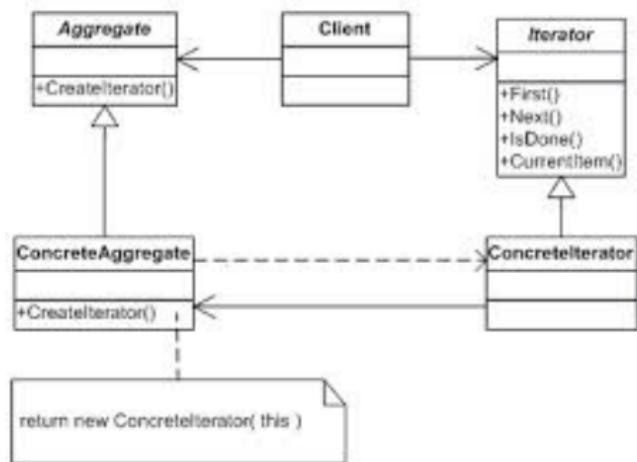
# Patron de conception Iterator

## Solution



Le programme client qui désire accéder à un Aggregate demande à ce dernier de lui procurer un distributeur de ses éléments (Iterator) par l'appel à la méthode CreateIterator.

# Patron de conception Iterator



Ce distributeur d'éléments (Iterator) est créé par instantiation d'une classe `ConcreteIterator`, elle-même conforme à un type `Iterator` fournissant des opérations de parcours et de récupération des éléments.

# Itérateurs

## L'interface Iterator de Java

```
public interface Iterator<T> {
    T next();           // retourne element courant
                        // et passe a l'element suivant
    boolean hasNext(); // teste s'il reste un element
    void remove();     // efface l'element visite
}
```

## Correspondance avec le patron de conception

Java	<i>Patron du GOF</i>
l'itérateur est positionné par défaut au début	First()
next()	CurrentItem() et Next()
hasNext()	isDone()
remove()	pas d'équivalent

# Itérateurs

## Itérable

Un objet *itérable* est un objet sur lequel on dispose d'un iterator  
c'est l'Aggregate du patron de conception Iterator

## L'interface

```
public interface Iterable<T>
{
    Iterator<T> iterator();
}
```

## Correspondance avec le patron de conception

Java	Patron du GOF
iterator()	CreateIterator()

# Illustration avec les collections Java

Toutes les collections sont des objets itérables, notamment les listes.

## Vue très simplifiée d'une liste

```
public class ArrayList<T> implements Iterable<T>{
    Iterator<T> iterator(){ ..... }
}
```

## Vue très simplifiée d'un itérateur de liste

```
public class ArrayListIterator <T>
                    implements Iterator<T>{
    T next(){.....}
    boolean hasNext(){.....}
    void remove(){.....}      .....
}
```

# Exemple de parcours d'une liste d'étudiants

## Création de la liste

```
List<Etudiant> listeEtu = new ArrayList<Etudiant>();  
Etudiant zo =  
    new Etudiant("Zoe", 12, 14, 17, 26, 1, 1);  
Etudiant pa =  
    new Etudiant("Paolo", 27, 1, 2);  
Etudiant je =  
    new Etudiant("Jean", 24, 1, 3);  
listeEtu.add(zo);  
listeEtu.add(pa);  
listeEtu.add(je);
```

# Itérateurs

Parcourir la liste avec une boucle `for` et une variable compteur

```
double moyenne = 0;  
  
for (int i=0; i<listeEtu.size(); i++)  
    moyenne += listeEtu.get(i).moyenne();  
  
moyenne = moyenne/listeEtu.size();
```

# Itérateurs

## Parcourir la liste avec un itérateur

```
double moyenne2 = 0;  
Iterator<Etudiant> ite = listeEtu.iterator();  
  
while (ite.hasNext())  
    moyenne2 += ite.next().moyenne();  
  
moyenne2 = moyenne2/listeEtu.size();
```

# Itérateurs

Parcourir la liste avec *for each* qui est traduit en un itérateur

```
double moyenne3 = 0;  
  
for (Etudiant e : listeEtu)  
    moyenne3 += e.moyenne();  
  
moyenne3 = moyenne3/listeEtu.size();
```

# Itérateurs

## L'opération `remove`

On peut utiliser la méthode `remove` pendant l'itération sans avoir de problème de changement d'indice. Par exemple, si on veut supprimer "Paolo" et "Jean", on peut écrire le code suivant.

```
Iterator<Etudiant> iter = listeEtu.iterator();
while (iter.hasNext())
{
    Etudiant e = iter.next();
    if (e.getNom().equals("Paolo")
        || e.getNom().equals("Jean"))
        iter.remove();
}
```

# Itérateurs

## L'opération remove

Plutôt que ... dans le cas d'une boucle classique où on doit diminuer l'indice après le retrait

```
for (int i=0; i<listeEtu.size(); i++)
    if (listeEtu.get(i).getNom().equals("Paolo")
        || listeEtu.get(i).getNom().equals("Jean"))
        {listeEtu.remove(i); i--;
```

# Itérateurs

Un itérateur spécifique pour les listes

Où les opérations prévues dans l'interface seront spécialement efficaces

```
public interface ListIterator<E>
    extends Iterator<E> {
    boolean hasNext();
    E next();
    boolean hasPrevious();
    E previous();
    int nextIndex();
    int previousIndex();
    void remove();
    void set(E o);
    void add(E o);
}
```

# Un itérateur de Pile

Créer un itérateur pour sa propre structure, par exemple une pile simplifiée

```
public class Pile<T> {  
    private ArrayList<T> elements;  
    public Pile(){initialiser();}  
    public T depiler() {  
        if (this.estVide()) return null;  
        T sommet = elements.get(elements.size()-1);  
        elements.remove(sommet);  
        return sommet;  
    }  
    public void empiler(T t) {  
        elements.add(t);  
    }  
    public boolean estVide() {  
        return elements.isEmpty();  
    }  
}
```

# Un itérateur de Pile

Suite de la définition de la pile

```
public class Pile<T> {  
    .....  
    public void initialiser() {  
        elements = new ArrayList<T>();  
    }  
    public T sommet(){  
        if (! this.estVide())  
            return elements.get(elements.size()-1);  
        else return null;  
    }  
    public String toString(){  
        return "Pile = "+ elements;  
    }  
}
```

# Pour rendre la pile itérable

```
public class Pile<T>
    implements Iterable<T>{
    .....
    public Iterator<T> iterator() {
        return new IteratorPile(elements);
    }
}
```

# Une classe Itérateur de pile

```
public class IteratorPile<T> implements Iterator<T>{  
    private Iterator<T> iterateur_elements;  
    public IteratorPile(ArrayList<T> elements) {  
        this.iterateur_elements = elements.iterator();  
    }  
    public boolean hasNext() {  
        return this.iterateur_elements.hasNext();  
    }  
    public T next() {  
        return this.iterateur_elements.next();  
    }  
    public void remove() {  
        this.iterateur_elements.remove();  
    }  
}
```

# Un main avec l'itérateur utilisé explicitement

```
public static void main( String [] a )
{
    Pile<String> p = new Pile<String>();
    p.empiler("a"); p.empiler("b"); p.empiler("c");

    Iterator<String> it = p.iterator();
    while (it.hasNext())
        System.out.println(it.next());
}
```

# Un main avec foreach

```
public static void main( String[] a )
{
    Pile<String> p = new Pile<String>();
    p.empiler("a"); p.empiler("b"); p.empiler("c");

    for ( String element : p )
        System.out.println(element);
}
```

## Faire différents traitements ...

Introduction d'une nouvelle classe pour représenter des données entreprise

```
public class DossierEntreprise {  
    private String identification;  
    private int anneeCreation;  
    private String emailAddress;  
    .....  
}
```

## Faire différents traitements ...

Introduction d'une classe utilitaire pour les piles avec des traitements spécifiques pour les piles de dossiers d'entreprises

```
public class PileParcours {  
  
    public static<T extends DossierEntreprise>  
        void printMailJeunesEntreprises(Pile<T> p)  
    {  
        for (T element : p)  
            if (element.getAnneeCreation()>=2012)  
                System.out.println(element.getEmailAddress());  
    }  
    ...  
}
```

## Faire différents traitements ...

Manque de généralité du code précédent : à chaque nouveau traitement (imprimer identifications, calculer moyenne âge des jeunes entreprises, etc.) on écrira un nouveau parcours

Solution en Java < 7 : spécifier des traitements dans des classes

### traitement de type test

```
public interface TestDossier {  
    boolean test(DossierEntreprise d);  
}  
  
public class TestJeuneEntreprise implements TestDossier {  
    @Override  
    public boolean test(DossierEntreprise d) {  
        return d.getAnneeCreation() >=2012;  
    }  
}
```

# Faire différents traitements ...

traitement de type action

```
public interface ActionDossier {  
    public void agit(DossierEntreprise d);  
}  
  
public class PrintEmail implements ActionDossier {  
    @Override  
    public void agit(DossierEntreprise d) {  
        System.out.println(d.getEmailAddress());  
    }  
}
```

# Faire différents traitements ...

## Version généralisée du print

```
public class PileParcours {  
    public static<T extends DossierEntreprise>  
        void printEntreprises  
            (Pile<T> p, TestDossier t, ActionDossier a)  
    {  
        for (T element : p)  
            if (t.test(element))  
                a.agit(element);  
    }  
}
```

# Faire différents traitements ...

## Version généralisée du print

```
public static void main(String[] args) {  
    Pile<DossierEntreprise> p = new Pile<>();  
    p.empiler(new DossierEntreprise  
        ("ChezJacques", 2012, "cj@gmail.com"));  
    p.empiler(new DossierEntreprise  
        ("Laforet", 2013, "lf@yahoo.fr"));  
    p.empiler(new DossierEntreprise  
        ("Ast", 2010, "ast@astservice.com"));  
  
    printEntreprises(p,  
        new TestJeuneEntreprise(),  
        new PrintEmail());  
}
```

# Vers Java 8 : lambdas, stream, agrégations

```
public static void main(String[] args) {  
  
    Pile<DossierEntreprise> p = new Pile<>();  
    p.empiler(new DossierEntreprise  
        ("ChezJacques",2012,"cj@gmail.com"));  
    p.empiler(new DossierEntreprise  
        ("Laforet",2013,"lf@yahoo.fr"));  
    p.empiler(new DossierEntreprise  
        ("Ast",2010,"ast@astservice.com"));  
  
    p  
        .stream()  
        .filter(d -> d.getAnneeCreation()>=2012)  
        .map(d -> d.getEmailAddress())  
        .forEach(email -> System.out.println(email));  
  
}
```

# Vers Java 8 : le goût des lambdas

## Fonction anonyme

( liste de paramètres ) → body

## Quelques exemples

`d → d.getAnneeCreation()>=2012`

`(DossierEntreprise d) → d.getAnneeCreation()>=2012`

`d → { return d.getAnneeCreation()>=2012; }`

`(a,b) → a+b`

`(int a, int b) → a+b`

## Autres éléments

Capture, Utilisation de l'environnement, ...

# Vers Java 8 : les Streams et les opérations d'agrégation

## Stream

- Séquence d'éléments avec traitement séquentiel ou parallèle
- Ne stocke pas ses éléments mais décrit (de manière déclarative) sa source et les opérations qui seront effectuées
- Le traitement est pris en charge par l'interprète (et plus efficace)
- L'itération est interne (et non externe comme avec les itérateurs)

```
public static void main(String [] args) {  
    Pile<DossierEntreprise> p = new Pile<>();  
    ...  
    p  
        .stream()  
        ...  
}
```

## Dossiers entreprise

# Vers Java 8 : les Streams et les opérations d'agrégation

## filter

retourne un second stream constitué des éléments du premier stream qui vérifient le prédictat

```
public static void main(String[] args) {  
    Pile<DossierEntreprise> p = new Pile<>();  
    ....  
    p  
        .stream()  
        .filter(d -> d.getAnneeCreation()>=2012)  
    ....  
}
```

Dossiers entreprise dont l'année de création est postérieure à 2012

# Vers Java 8 : les Streams et les opérations d'agrégation

## map

retourne un troisième stream constitué des résultats de l'application de la fonction aux éléments du premier

```
public static void main(String[] args) {  
    Pile<DossierEntreprise> p = new Pile<>();  
    ...  
    p  
        .stream()  
        .filter(d -> d.getAnneeCreation()>=2012)  
        .map(d -> d.getEmailAddress())  
    ...  
}
```

Adresses mails des dossiers entreprise dont l'année de création est postérieure à 2012

# Vers Java 8 : les Streams et les opérations d'agrégation

**foreach**

applique une fonction aux éléments du stream

```
public static void main(String[] args) {  
    Pile<DossierEntreprise> p = new Pile<>();  
    ....  
    p  
        .stream()  
        .filter(d -> d.getAnneeCreation()>=2012)  
        .map(d -> d.getEmailAddress())  
        .forEach(email -> System.out.println(email));  
}
```

Affichage des adresses mails des dossiers entreprise dont l'année de création est postérieure à 2012

# Java 1.7

## Age moyen des jeunes entreprises

```
public static<T extends DossierEntreprise>
            double ageMoyenJeunesEntreprises(Pile<T> p)
{
    double m = 0;  int nbr= 0;
    for (T element : p)
        if (element.getAnneeCreation()>=2012)
            {m += 2014-element.getAnneeCreation();
             nbr++;}
    return m / nbr;
}

//main
System.out.println(ageMoyenJeunesEntreprises(p));
```

# Java 1.8

## Age moyen des jeunes entreprises

```
// main  
  
Pile<DossierEntreprise> p = new Pile<>();  
.....  
System.out.println(  
    p  
        .stream()  
        .filter(d -> d.getAnneeCreation()>=2012)  
        .mapToInt(DossierEntreprise::getAnneeCreation)  
        .map(i -> 2014 - i)  
        .average()  
        .getAsDouble());
```