Programmation en Java 2008-2009
Frédéric Mallet & Jean-Paul Roy
L1 Info&Math - Faculté des Sciences de Nice
http://deptinfo.unice.fr/~roy

Une collection de taille variable

Les listes

Rappel : les collections de type **tableau**

- Les **tableaux** sont des collections d'éléments **numérotés**, **de même type**, **en nombre fixé** une fois pour toutes !

`int[]`	`double[]`	`Point[]`
tableau d'entiers	*tableau de réels*	*tableau de points*

- Les **tableaux** sont des objets mais avec une syntaxe à part :

```
int[] notes = new int[10];
```

- Outre l'accès au champ `length`, les seules opérations possibles sont :
  - **consulter** directement l'élément numéro k
  - **modifier** directement l'élément numéro k

COMPLEXITE : Il est garanti que le **temps d'accès à l'élément numéro k** du tableau ne dépend pas de k (il a lieu **en temps constant**).

Pourquoi veut-on autre chose ?

- Parce que deux opérations importantes ne sont pas disponibles avec le type tableau :
  - **ajouter** un nouvel élément
  - **supprimer** un élément

- On voudrait en quelque sorte des tableaux de longueur variable !

- La classe `ArrayList` de l'API Java remplit cette mission.

- En fait, l'idée de base est simple : on part d'un tableau et on lui ajoute des éléments un à un. Lorsqu'il est plein, on demande un nouveau tableau deux fois plus long, et on recopie l'ancien. Pour supprimer un élément, on procède par décalage à gauche...

- On pourrait presque le faire à la main si l'API n'existait pas ! Mais une API existe justement pour simplifier la vie du programmeur...

Le projet Calepin

• Une instance de la classe `Calepin` est un agenda qui permet de stocker un nombre variable de textes (chaînes de caractères).

• On voudra pouvoir ***ajouter*** un nouveau texte, ***effacer*** un texte, connaître le ***nombre*** de textes, et ***afficher*** un texte.

```
Welcome to DrJava.
> Calepin cal = new Calepin();
> cal.ajouter("aller en TP");
> cal.ajouter("acheter un ballon");
> cal.ajouter("passer en L2");
> cal.afficher();
Texte 0 : aller en TP
Texte 1 : acheter un ballon
Texte 2 : passer en L2
> cal.supprimer(1);
> cal.afficher();
Texte 0 : aller en TP
Texte 1 : passer en L2
> cal.nbTextes()
2
```

Les importations

- Il faut importer la classe `ArrayList` qui se trouve dans le package `java.util` de l'API, tout comme la classe `Random`.

- On importe donc la classe :

```
import java.util.ArrayList;
```

`ArrayList` est une classe paramétrée !

- On pourrait utiliser la classe `ArrayList` telle quelle, mais ceci conduit à des *difficultés de typage*. En réalité, on précise le type des objets qui seront contenus dans la liste, ici des objets de type `String`.

- Une **liste de chaînes de caractères** est un objet de type :

```
ArrayList<String>
```

N.B. Les éléments d'une `ArrayList` sont nécessairement des **objets** !

• Mais après tout, les tableaux aussi étaient déjà des sortes de classes paramétrées :

`String[]`	`ArrayList<String>`
tableau de chaînes	*liste de chaînes*

Une liste de chaînes en champ privé

• L'unique champ (privé) sera une liste `textes` d'objets de type `String` :

```
private ArrayList<String> textes;
```

Le constructeur initialise le champ

```
public Calepin() {
   textes = new ArrayList<String>();
}
```

Au début, la liste est vide...

Constructor Summary

`ArrayList()`
Constructs an empty list with an initial capacity of ten.

La méthode pour connaître le nombre d'éléments

- On utilise la méthode d'instance `size()` de la classe `ArrayList` :

```
public int nbTextes() {
  return textes.size();
}
```

int	**size**() Returns the number of elements in this list.

La méthode pour ajouter un élément

- On utilise la méthode d'instance `add(E obj)` de la classe paramétrée `ArrayList<E>` :

```
public void ajouter(String str) {
  textes.add(str);
}
```

boolean	**add**(E o) Appends the specified element to the end of this list.

N.B. add renvoie un booléen inutile. Nous nous servons de `add(str)` en tant qu'instruction, en la faisant suivre d'un point-virgule.

La méthode pour supprimer un élément

E	**remove**(int index) Removes the element at the specified position in this list.

- On utilise la méthode d'instance `remove(int i)` de la classe `ArrayList<E>` qui supprime l'élément numéro `i`, avec $0 \leq i < size()$

```
public void supprimer(int numero) {
  if ((numero >= 0) && (numero < textes.size())) {
    textes.remove(numero);
  }
}
```

N.B. `add` renvoie inutilement l'élément supprimé. Nous nous servons de `remove(i)` en tant qu'instruction, en la faisant suivre d'un point-virgule.

Ceci est toujours possible avec une méthode Java qui ne renvoie pas `void`*, on peut faire comme si elle renvoyait* `void`*, en acceptant de perdre le résultat. La réciproque est fausse. Et ceci ne vaut que pour les méthodes, vous ne pouvez pas écrire* `2+3;`

La méthode pour afficher un élément

E	**get**(int index) Returns the element at the specified position in this list.

- On utilise la méthode d'instance `get(int i)` de la classe `ArrayList<E>` qui retourne l'élément numéro `i`, avec $0 \leq i < size()$

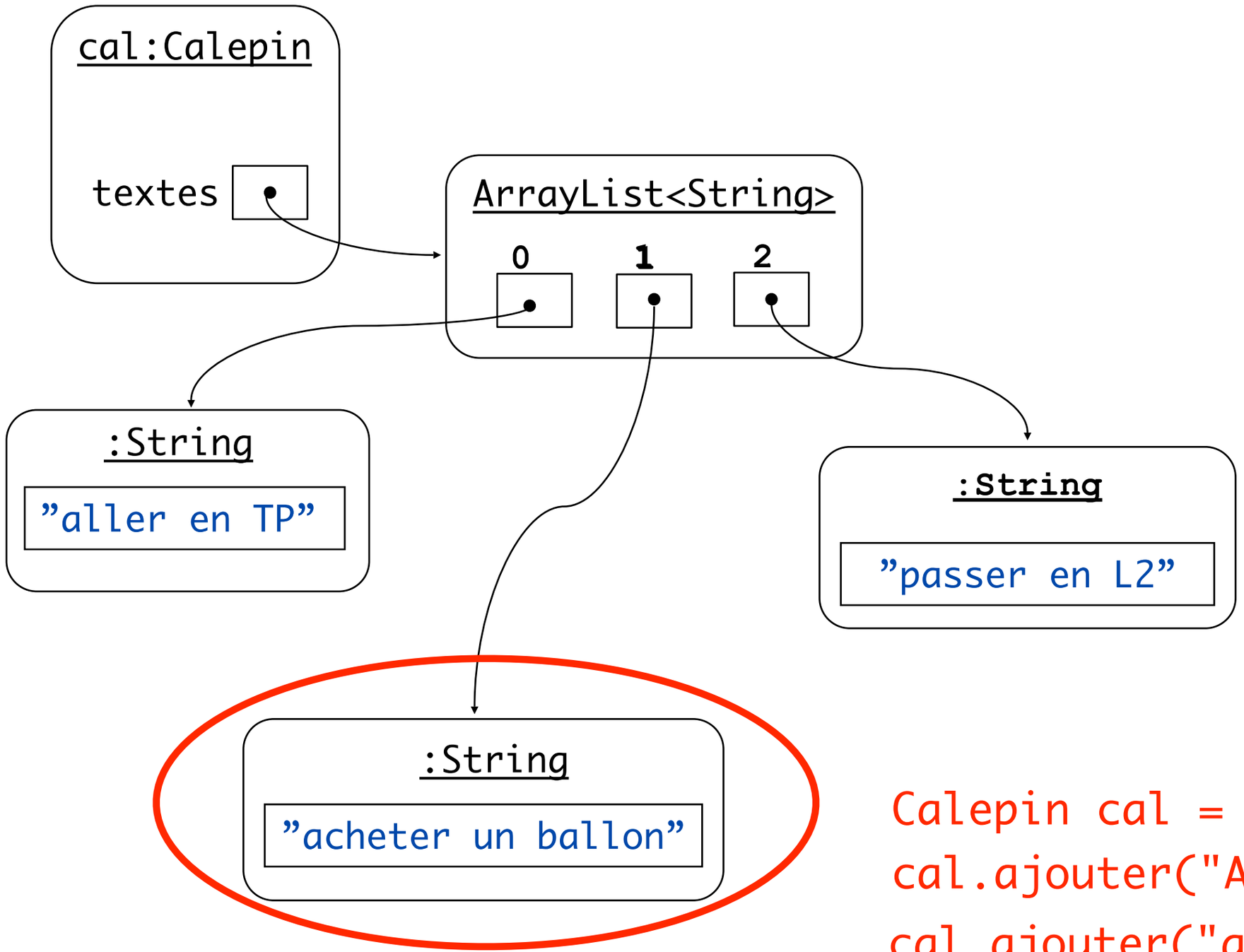
```
public void afficher(int numero) {
  if ((numero >= 0) && (numero < textes.size())) {
    System.out.println("Texte " + numero + " : " + textes.get(numero));
  }
}
```

- On **surcharge** cette méthode avec une autre version qui affiche tous les éléments :

```
public void afficher() {
  for(int i = 0; i < nbTextes(); i = i + 1) {
    afficher(i);
  }
}
```

Ce n'est pas une récursivité !

- Structure d'un calepin :




```
Calepin cal = new Calepin();
cal.ajouter("Aller en TP");
cal.ajouter("acheter un ballon");
cal.ajouter("passer en L2");
cal.supprimer(1);
```

Le Garbage Collector (GC)

- Après suppression, que devient l'objet "acheter un ballon" ?

- Cet objet "acheter un ballon" devient inaccessible (donc inutilisable) si personne d'autre ne pointe sur lui ! Par exemple un autre agenda…

- S'il est effectivement devenu inutilisable, on peut considérer qu'il n'existe plus … et le système pourra **récupérer la mémoire** qu'il occupe !

- Comme beaucoup de langages, Java est muni d'un dispositif de **recyclage automatique** des objets inaccessibles ! Ce dispositif se nomme un GARBAGE COLLECTOR (*ramasse-miettes*). À sa charge de prouver que tel ou tel objet est bien devenu inaccessible…

- Certains langages (comme C) ne sont pas pourvus de GC et la gestion de la mémoire doit être manuelle, ce qui peut être fort difficile … et dangereux !

Résumé suffisant de la classe ArrayList<E>

- Importation :

```
java.util.ArrayList;
```

- Constructeur :

```
ArrayList<E>()
```

- Méthodes :

```
int size()
E get(int i)
void add(E obj)
void remove(int i)
```

- En ***Java pur***, n'utilisez JAMAIS la classe `ArrayList` non paramétrée !

- La classe paramétrée `ArrayList<E>` suppose que `E` est une **classe** !

Donc `E` ≠ `int` par exemple !!

Bon, mais si je veux des listes d'entiers ?

- Bien entendu on va vous les fournir ! Il faut pour cela remédier au fait qu'en Java (hélas), **les entiers ne sont pas des objets**, mais des éléments d'un **type primitif**.

- Java propose pour cela une **classe enveloppante** (*wrapper class*) pour chaque type primitif :

type primitif	*classe enveloppante*
`int`	`Integer`
`double`	`Double`
`float`	`Float`
`boolean`	`Boolean`
`char`	`Character`

- On ne peut pas jouer avec des ~~`ArrayList<int>`~~, mais on optera pour des `ArrayList<Integer>` !

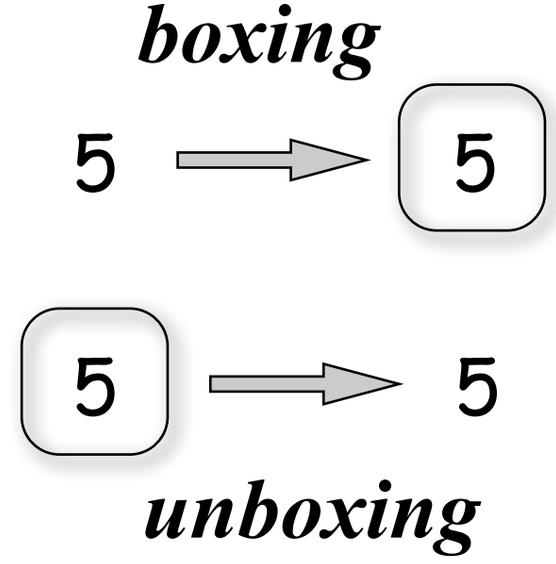
- Il faut savoir passer d'un `int` à un `Integer`, et inversement !

  - transformer un `int` en `Integer`

```
Integer obj = new Integer(5);
```

  - récupérer l'`int` caché dans un `Integer`

```
int n = obj.intValue();
```



- Idem pour les `double`, etc. Voir la doc de la classe enveloppante...

- Une **liste d'entiers** (par exemple des notes d'étudiant) serait donc construite ainsi :

```
ArrayList<Integer> notes = new ArrayList<Integer>();
```

Va-t-il falloir jongler sans cesse entre les `int` *et les* `Integer` *?*

Heureusement non depuis seulement trois ans (JDK 1.5)...

Auto-boxing et auto-unboxing

- Prenons l'exemple de la méthode add de `ArrayList<Integer>`. Je veux rajouter la note 13. Je devrais normalement écrire :

```
notes.add(new Integer(13));
```

mais avec le mécanisme d'**auto-boxing**, Java effectue automatiquement la conversion d'un `int` vers un `Integer` si besoin :

```
notes.add(13);
```

- Et inversement, par **auto-unboxing**, il nous évite d'avoir à écrire :

```
Integer obj = notes.get(3);      // l'élément numéro 3
int n = obj.intValue();
```

et nous autorise à récupérer directement un `int` là où se trouve en réalité un `Integer` :

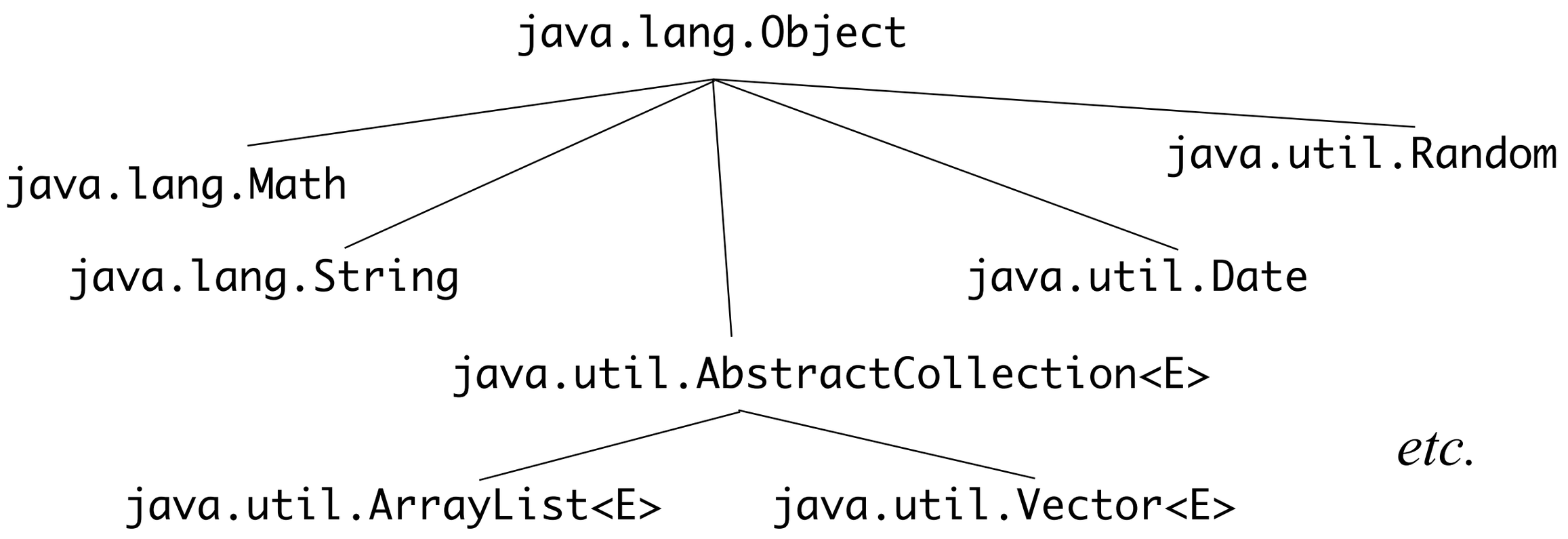
```
int n = notes.get(3);            // l'élément numéro 3
```

Les listes sont des collections

• La classe `ArrayList` n'est qu'une manière de voir les listes en Java, il en existe d'autres que vous verrez au semestre 2, par exemple la classe `Vector`, ou la classe `LinkedList`.

• Ce sont toutes des exemples de **collections**.

• L'API va en effet organiser ses classes en une hiérarchie, avec des **sous-classes**, comme les ensembles et les sous-ensembles en maths.



COMPLEMENTS SUR L'AUTO-(UN)-BOXING

- Nous avons vu que l'API Java fournit la classe brute `ArrayList`, que l'on doit en principe **paramétrer** par une classe d'objets. Par exemple `ArrayList<Point>`, ou `ArrayList<Integer>`, ou `ArrayList<String>`, etc. Cela permet de fixer une fois pour toutes le type uniforme des objets de la liste.

- Or tout objet est une instance de la classe `Object`. Donc utiliser `ArrayList` non paramétrée revient à utiliser `ArrayList<Object>`. Ce qui permet au passage de déposer dans la liste... divers types d'objets : des *points*, des *entiers*, des *chaînes de caractères*, etc.

En Processing ou au toplevel de DrJava

```
import java.util.ArrayList;
import java.awt.Point;
ArrayList bazar = new ArrayList();
bazar.add(new Point(5,-8));
bazar.add("Hello World !");
bazar.add(new Integer (6));
```

- Jusque là tout va bien. Sauf que nous avons dû écrire :

```
bazar.add(new Integer (6));
```

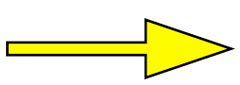
au lieu de :

```
bazar.add(6);
```

puisque, la classe n'étant pas paramétrée par `Integer`, je ne peux pas invoquer l'auto-boxing, et 6 n'est pas un objet !

- Cette petite difficulté s'accroît au moment de la **récupération d'un élément de la liste** par la méthode `get`. En effet, notre liste `bazar` est une liste d'objets (`Object`). J'ai le droit d'écrire :

```
Object e0 = bazar.get(0);
System.out.println("e0 = " + e0);
```



```
e0 = java.awt.Point[x=5,y=-8]
```

• Si j'utilise l'objet e0, il est bien reconnu comme un Point, car on a **tardivement** (*late binding*) mis un point dans cet objet.

• Mais si j'envoie un message à e0 qui a été créé comme un Object (*early binding*), ce n'est pas le **point** qui recevra le message ! Zut !

```
> Object e0 = bazar.get(0);
> e0
java.awt.Point[x=5,y=-8]
> System.out.println("e0 = " + e0);
e0 = java.awt.Point[x=5,y=-8]
> e0.getX()
Error: No 'getX' method in 'java.lang.Object'
```

• Ce **double typage** conduit à une **difficulté** : lorsque je récupère un objet dans ma liste, **je dois connaitre son type de création** (*early*). Si je sais que e0 est un Point, je pourrai écrire directement :

```
> Point e0 = (Point) bazar.get(0);        // cast
> System.out.println("e0 = " + e0);
e0 = java.awt.Point[x=5,y=-8]
> e0.getX()
5.0
```

Solution : le *CAST* ou **transtypage**

```
> Object e0 = bazar.get(0);
> e0.getX()
Error: No 'getX' method in 'java.lang.Object'
> Point p0 = (Point) e0;
> p0.getX()
5.0
```

- Je ne peux procéder à ce **transtypage** (*type cast*) QUE parce que je sais qu'au fond de lui, `e0` est un `Point` !...

```
> String s = (String) e0;       // N'importe quoi !...
ClassCastException: s
```

PROCESSING NE CONNAIT PAS (ENCORE) LES CLASSES PARAMETREES ! Donc :

- en Java pur : ArrayList<E> paramétrée, toujours !

- en Processing : ArrayList + *type cast*. On évitera !