

POO - Collections

Exemple : ArrayList

1. Collections (chapitre sera abordé dans le module d'algo avancée au S3).
2. Parcourir une collection
3. Comment créer une collection de primitifs ?

1. Collections

Une collection est une classe permettant de stocker et d'organiser des objets

Java distingue (principalement) 2 catégories de collections :

- les ensembles :

ensemble = groupe d'éléments sans duplication

classes : *HashSet*, *TreeSet* (voir vues publiques de ces classes)

- les listes :

liste = groupe ordonné d'éléments (contenant d'éventuelles duplications)

classes : *ArrayList*, *Vector* (ancienne classe)

Caractéristiques d'une collection

Le nombre d'éléments d'une collection varie en fonction des ajouts et des retraits (il ne peut donc être défini à l'avance).

Note : le nombre d'éléments d'un tableau est déterminé par l'instruction **new** à la création du tableau et demeure invariable.

Une collection "classique" ne contient que des références de la classe *Object* :

on peut ajouter une référence sur un objet de n'importe quelle classe dans une collection (toutes les classes sont compatibles avec la classe *Object*)

On peut utiliser les collections "génériques" (java 5)

Méthodes élémentaires d'une collection

chaque type de collection possède les méthodes suivantes :

- **constructeur par défaut initialisant une collection vide**

ex) `HashSet ensemble = new HashSet(), ArrayList liste = new ArrayList()`

- **public boolean add(Object o)**

// ajoute la référence o à la collection courante (retourne true si l'ajout est effectué).

- **public boolean contains(Object o)**

// retourne true si une référence de la collection courante désigne un objet égal (au sens de equals()) à l'objet référencé par o.

- **public boolean remove(Object o)**

// cherche dans la collection courante une référence sur un objet égal (au sens de equals()) à l'objet référencé par o. Si une telle référence est trouvée elle est retirée et on retourne true.

- **public int size()**

//retourne le nombre de références dans la collection

Caractéristiques d'une liste

Chaque élément d'une liste est identifié par un indice.

La liste est ordonnée selon ces indices.

Comportement spécifique de la méthode *add* :

- **public boolean add(Object o)**

// ajoute la référence o en dernière position de la liste courante (retourne true si l'ajout est effectué, sinon false)

Méthodes spécifiques aux listes

Elles utilisent l'indexation des éléments

- **void add(int i, Object o)**

// insère la référence o au rang i de la liste courante (les références de rang supérieur ou égal à i sont décalées d'un cran vers la droite)

- **Object remove(int i)**

// retire de la liste la référence de rang i et la retourne

(les références de rang supérieur ou égal à i sont décalées d'un cran vers la gauche)

- **Object get(int i)**

// retourne la référence de rang i de la liste courante

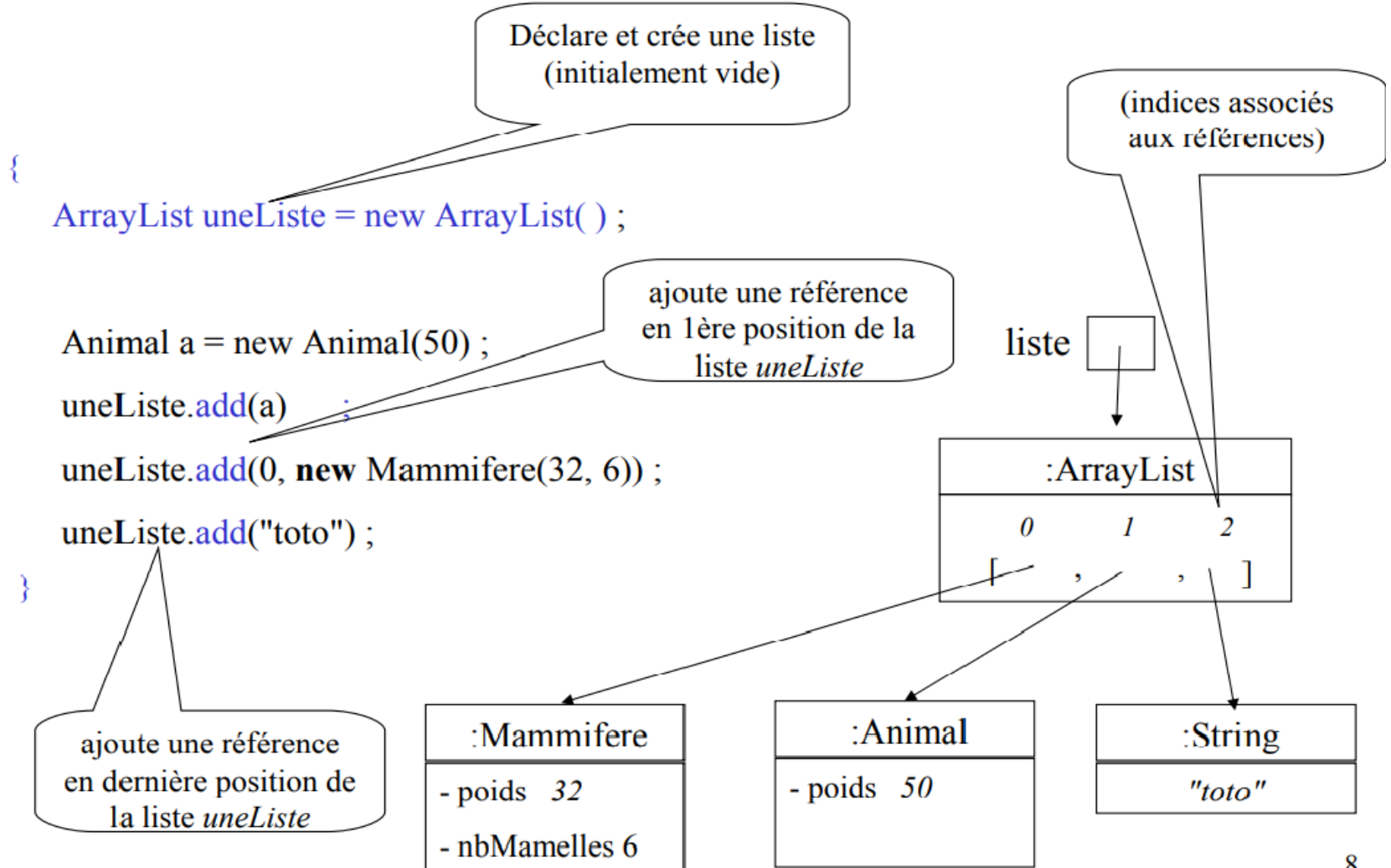
- **Object set(int i, Object o)**

// remplace la référence de rang i (et la retourne) par la référence o

- **int indexOf(Object o)**

// cherche dans la liste courante une référence désignant un objet égal (au sens de equals) à l'objet référencé par o et retourne le rang de cette référence si elle est trouvée, sinon retourne -1

Exemple de création d'une liste



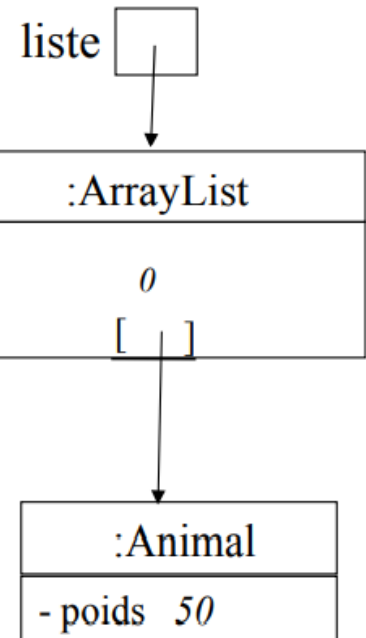
Le type initial d'un élément inséré dans une collection est perdu

Un exemple avec une liste

```
{  
    ArrayList liste = new ArrayList( ) ;  
    Animal a = new Animal(50) ;  
    liste.add(a) ;  
    Animal b ;  
    b = liste.get(0) ;  
    b = (Animal) liste.get(0) ;  
}
```

Erreur de compilation : la méthode *get* retourne un *Object* incompatible avec *Animal*

Ok
le *cast* est nécessaire car pour le compilateur toutes les références au sein de la liste sont des références sur des *Object*



2. Parcourir une collection

2.1. Parcours d'une collection sans ajout ni retrait d'éléments

Utiliser une boucle for

Exemple avec un for classique (1/2)

```
public static void afficheListe(ArrayList uneListe)
{
    for (int i = 0 ; i < uneListe.size() ; i++)
    {
        System.out.println(uneListe.get(i)) ;
    }
}
```

Compilateur ok :
la méthode *toString()* implicitement invoquée est définie dans la classe *Object*.
A l'exécution c'est la méthode *toString* de la classe de l'instance référencée qui s'applique (polymorphisme)

Exemple avec un for classique (2/2)

```
public static double moyennePoidsAnimaux(ArrayList uneListe)
{
    double sommePoids = 0 ;

    for (int i = 0 ; i < uneListe.size( ) ; i++)
    {
        Animal a = (Animal) uneListe.get(i) ;

        sommePoids = sommePoids + a.getPoids( ) ;
    }

    return (sommePoids / uneListe.size( ) ) ;
}
```

suppose que la liste ne contient
que des références sur des
instances de la classe *Animal* (et
sous-classes)

Exemple avec un for spécifique aux collections

(Java 5 uniquement)

```
public static double moyennePoidsAnimaux(ArrayList uneListe)
{
```

```
    double sommePoids = 0 ;
```

Pour chaque objet *o* de la liste
uneListe

```
    for (Object o : uneListe)
```

```
    {
```

```
        sommePoids = sommePoids + ((Animal) o).getPoids( ) ;
```

```
    }
```

```
    return (sommePoids / uneListe.size( ) ) ;
```

```
}
```

cast obligatoire (pour le
compilateur) : *getPoids()* n'est
pas définie dans *Object*

2.2. Parcours d'une collection avec ajout(s) ou retrait(s) d'éléments

L'ajout ou le retrait d'un élément durant le parcours d'une collection modifie cette dernière. S'il s'agit d'une liste les éléments restant changent d'indice

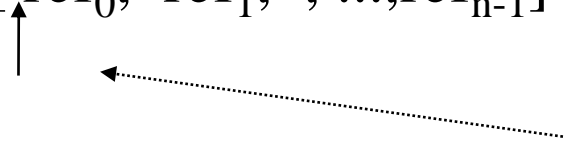
L'unique façon de parcourir sans problème une collection qui se modifie est d'utiliser un *itérateur*

Itérateur

Un *itérateur* est un objet dédié au parcours d'une collection

Un *itérateur* gère un curseur interne qui désigne l'élément de la collection auquel on peut accéder (le curseur est placé devant l'élément)

[ref₀, ref₁, , ...,ref_{n-1}]



Le curseur est initialement devant le premier élément et avance d'un cran à chaque fois qu'on accède à un élément

[ref₀, ref₁, , ...,ref_{n-1}]

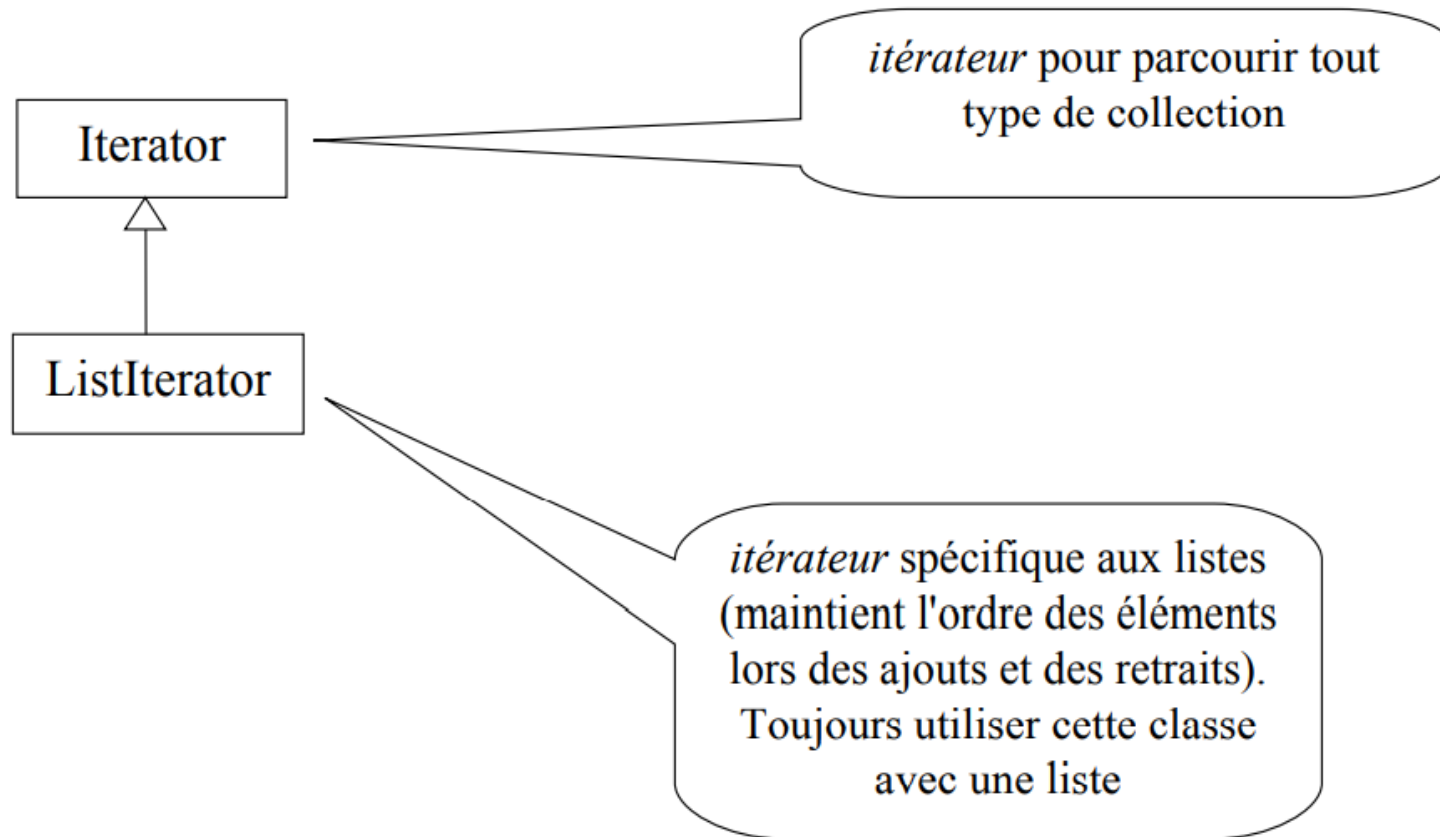


en fin de parcours le curseur se trouve au delà du dernier élément

[ref₀, ref₁, , ...,ref_{n-1}]



Classes d'itérateurs




Principales méthodes d'un itérateur (1/2)

(méthodes communes à *Iterator* et *ListIterator*)


un itérateur gère le parcours de la collection

- **public boolean hasNext()**

// retourne true s'il y a un élément derrière le curseur, false si l'on est en fin de collection

[ref₀,  ref₁, ref₂, ..., ref_{n-1}]



(retourne true)

[ref₀, ref₁, ref₂, ..., ref_{n-1} ]

(retourne false)

- **public Object next()**

// retourne l'élément situé derrière le curseur et avance le curseur d'un cran

Ex : [ref₀,  ref₁, ref₂, ..., ref_{n-1}] \Rightarrow [ref₀, ref₁,  ref₂, ..., ref_{n-1}]

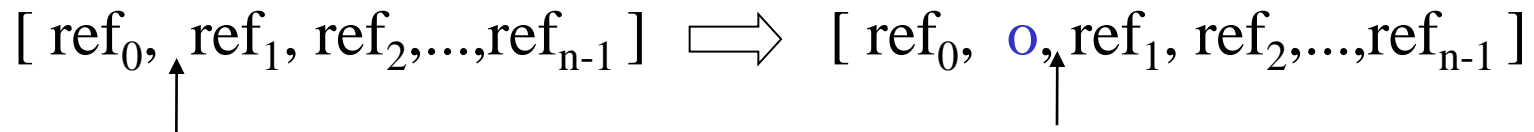
(retourne ref₁)

Principales méthodes d'un itérateur (2/2)

un itérateur gère la modification de la collection pendant le parcours :

- **public void add(Object o)**

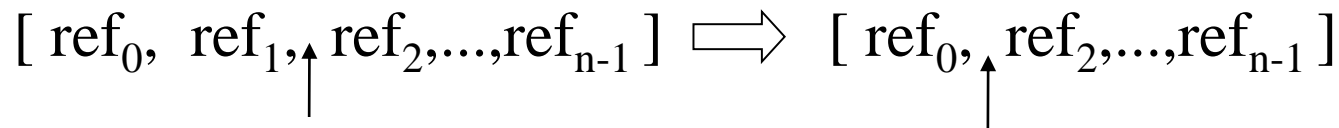
// insère la référence o derrière la position actuelle du curseur, et avance le curseur d'un cran (le curseur est situé derrière la référence insérée)



- **public void remove()**

// retire la référence retournée au dernier appel next() (il faut obligatoirement avoir appelé next() pour utiliser remove())

ex) *next()* a été activée et a retournée ref_1



Créer et utiliser un itérateur

Un exemple sur une liste (ici sans ajout ni retrait)

```
public static void afficheListe(ArrayList uneListe)
{
    LisIterator ite = uneListe.listIterator( ) ;

    while (ite.hasNext( ))
    {
        System.out.println(ite.next( ) ) ;
    }
}
```

la méthode *listIterator()* de la classe *ListArray* créé et retourne un itérateur sur la liste courante (ici *uneListe*). *ite* est une référence sur cet itérateur.

teste s'il y a encore un élément à parcourir.

retourne l'élément situé derrière le curseur et avance le curseur d'un cran

Un exemple avec modification de la collection pendant le parcours

```
public static void elimineAnimauxLourds(ArrayList uneListe)
{
    LisIterator ite = uneListe.listIterator( ) ;

    while (ite.hasNext( ))
    {
        Animal a = (Animal) ite.next( ) ;

        if (a.getPoids( ) > 100)
            ite.remove( ) ;
    }
}
```

Élimine la référence qui vient d'être retournée par *ite.next()*

3. Comment créer une collection de primitifs ?

Théoriquement on ne peut pas
(une collection ne contient que des références)

Pratiquement on peut en utilisant des *classes d'encapsulation*
(ou *adaptateurs*) spécialisées dans la représentation
des valeurs de type primitif.

type primitif	classe associée
<i>byte</i>	<i>Byte</i>
<i>short</i>	<i>Short</i>
<i>int</i>	<u><i>Integer</i></u>
<i>long</i>	<i>Long</i>
<i>float</i>	<i>Float</i>
<i>double</i>	<i>Double</i>
<i>char</i>	<u><i>Character</i></u>
<i>boolean</i>	<i>Boolean</i>

Classe d'encapsulation

Elle dispose d'un constructeur prenant en argument une valeur du type primitif qu'elle représente

Insérer des valeurs primitives au sein d'une collection

Il faut encapsuler chaque valeur de type primitif dans une instance de la classe d'encapsulation associée au type, puis insérer dans la collection une référence sur cette instance

Example

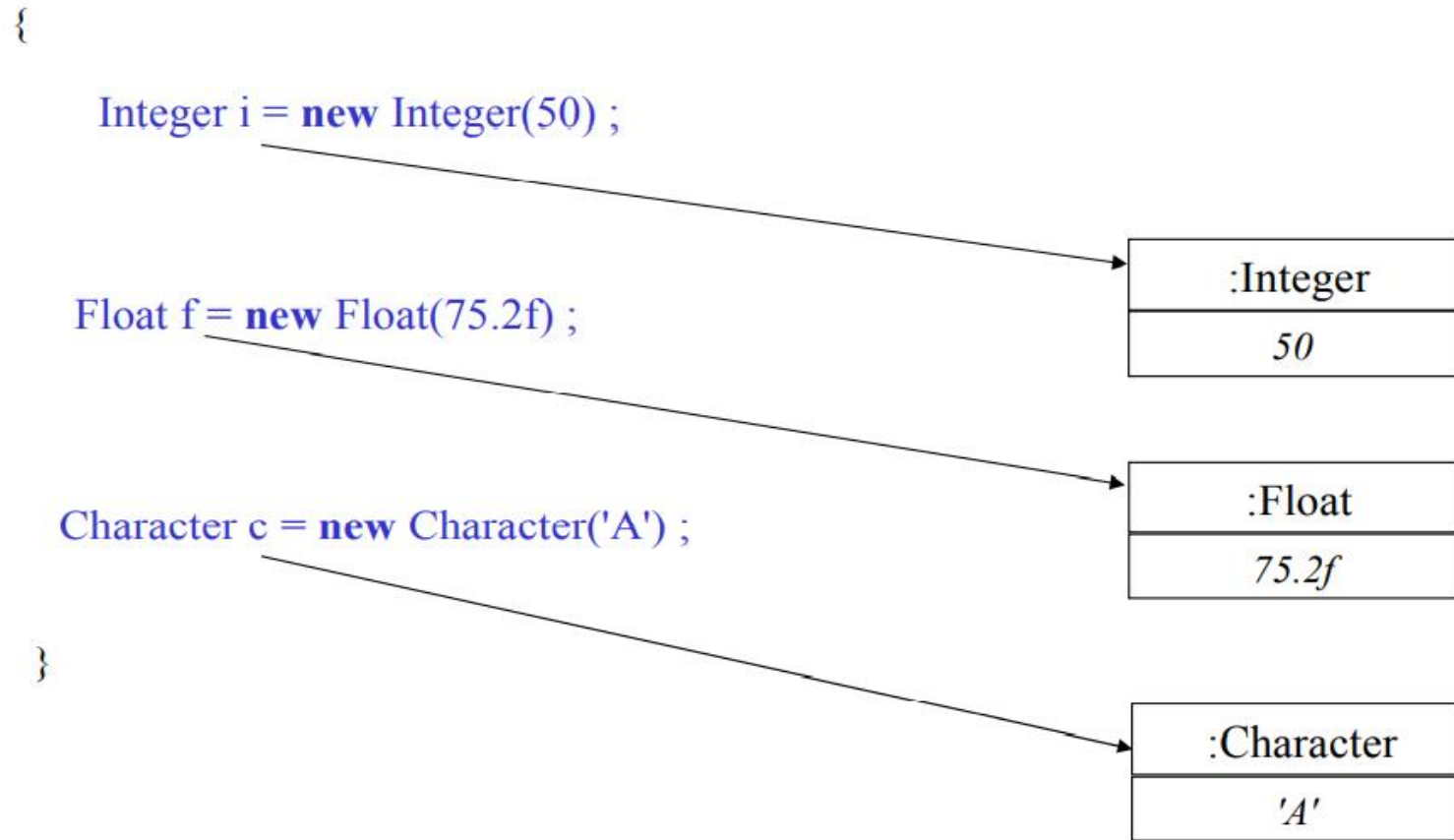
{

Integer i = **new** Integer(50) ;

Float f = **new** Float(75.2f) ;

Character c = **new** Character('A') ;

}



:Integer
50

:Float
75.2f

:Character
'A'

Exemple

Créer une collection contenant 10 Integer

```
ArrayList liste = new ArrayList() ;  
for (int i = 0 ; i < 10 ; i++)  
{  
    liste.add(new Integer(2*i)) ;  
}  
for (int i = 0 ; i < 10 ; i++)  
{  
    Integer ref = (Integer) liste.get(i) ;  
    int v = ref.intValue() ;  
}
```

Encapsulation (boxing) : création d'un objet de type *Integer* encapsulant la valeur primitive $2*i$ (de type *int*)

Désencapsulation (unboxing) : retourne la valeur primitive de type *int* encapsulée dans l' *Integer* valeur

Encapsulation/désencapsulation automatique

(java 5 uniquement)

Exemple

```
ArrayList liste = new ArrayList() ;
```

```
for (int i = 0 ; i < 10 ; i++)
```

```
{
```

```
    liste.add(2*i) ;
```

```
}
```

```
for (int i = 0 ; i < 10 ; i++)
```

```
{
```

```
    Integer ref = (Integer) liste.get(i) ;
```

```
    int v = ref ;
```

```
}
```

Encapsulation automatique : création implicite d'un objet de type *Integer* encapsulant la valeur primitive $2*i$

Désencapsulation automatique : la valeur primitive de type *int* encapsulée dans l' *Integer valeur* est automatiquement extraite (et affectée à *v*).

Collections et généricité

Exemple :

```
ArrayList <String> fichier = new ArrayList <String> ()
```

ainsi :

```
String nomFichier = fichier.get(0) ; //(sans transtypage)
```