Introduction à la programmation en Java

UFR Sciences de Nice

Licence Math-Info 2006-2007

Module L1I1

Frédéric MALLET

Jean-Paul ROY

Où en sommes-nous ?

- Nous savons rédiger le texte d'une classe d'objets, avec dans l'ordre : ses **champs**, ses **constructeurs**, ses **méthodes**.
- Nous pouvons exprimer qu'une méthode a ou n'a pas de résultat.
- L'utilisation d'une conditionnelle **if** n'a plus de secrets pour nous.
- Nous commençons à acquérir des capacités d'abstraction et de modularité.
- Nous savons faire la différence entre une méthode d'instance et une méthode de classe (*statique*).
- Nous avons fait connaissance avec les 3 boucles :

  `for, while, do…while`

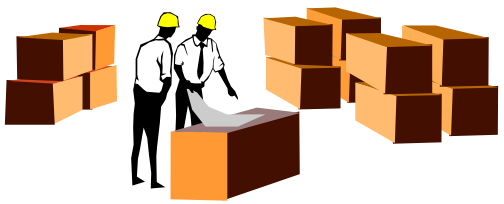
  qui nous permettent de programmer des répétitions.
- Nous savons piloter une tortue pour réaliser des programmes graphiques.
- Nous savons manipuler les tableaux : collections de taille fixe !

  `int[] tab = new tab[6];` // 6 est choisi **une fois pour toute !**

COURS 10

Collections d'objets

(de taille variable)



Que nous manque-t-il ?

- Beaucoup de choses en vérité !
- Pour l'instant :
  - construire de vastes **collections** d'objets
  - **parcourir** et analyser les collections créées.
- Mais :
  - Les collections seront-elles *ordonnées* ou *en vrac* ?
  - Leur nombre d'éléments sera-t-il *fixe* ou *variable* ?

Pourquoi des collections ?

♦ Les programmeurs ont besoin de stocker de nombreuses données dans des *collections* : bibliothèques, sécurité sociale, cartes d'étudiants, albums de photos, etc.

♦ Souvent des milliers d'éléments, parfois des millions !

♦ La **taille** (nombre d'éléments) d'une collection peut être :

- **fixe** : l'ensemble des lettres de l'alphabet (tableaux)
- **variable** : ma collection de DVD

♦ Nous allons modéliser un *agenda électronique* destiné à conserver des notes (petits textes sous forme de chaînes de caractères) saisies une à une.

Le projet **Notebook**

♦ Une instance de la classe **Notebook** est un *agenda* qui permet de stocker un *nombre variable* de notes.

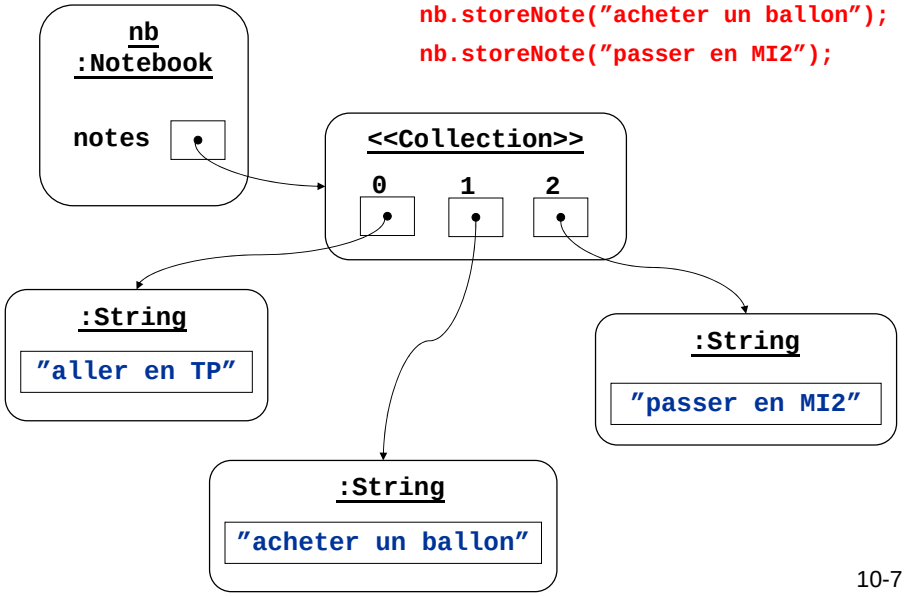
♦ Un agenda doit savoir en plus :

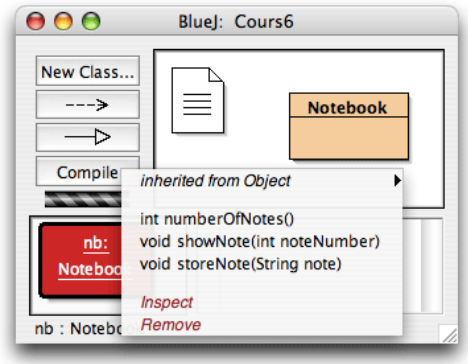
- afficher chacune des notes à la demande
- indiquer à tout moment combien de notes il contient.

```
Notebook nb = new Notebook();
nb.storeNote("aller en TP");
nb.storeNote("acheter un ballon");
nb.storeNote("passer en MI2");
```

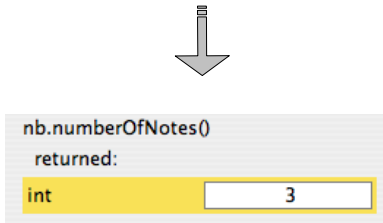

♦ Structure d'un agenda

```
Notebook nb = new Notebook();
nb.storeNote("aller en TP");
nb.storeNote("acheter un ballon");
nb.storeNote("passer en MI2");
```

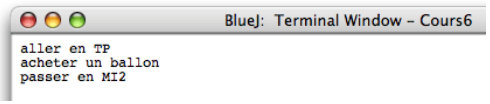




nb.numberOfNotes()




```
nb.showNote(0);
nb.showNote(1);
nb.showNote(2);
```



Utilisons l'API !

♦ Pour stocker les notes, nous utiliserons la **classe de bibliothèque** ArrayList. L'API de Java en comporte des centaines pour accélérer le travail des programmeurs !

***API** = **A**pplication **P**rogramming **I**nterface*

♦ La bibliothèque (API) des classes utilitaires est organisée en **groupes de classes** (paquetages) qui ont des utilisations proches. La classe ArrayList appartient au paquetage java.util

♦ Pour utiliser les classes d'un paquetage, on peut :

- Soit utiliser le *nom complet* : java.util.ArrayList
- Soit *importer le nom* : import java.util.ArrayList;

L'utilisation de import

```
class NoteBook {
  java.util.ArrayList notes;

  NoteBook() {
    notes = new java.util.ArrayList();
  }
}
```

OU

```
import java.util.ArrayList;
class NoteBook
{
  ArrayList notes;

  NoteBook() {
    notes = new ArrayList();
  }
}
```


La classe `ArrayList` de l'API

♦ C'est une collection à **nombre variable d'éléments**

- Cette collection est vide après l'appel du constructeur ;
- On peut ajouter autant d'éléments que l'on veut ;
- On a des moyens de connaître le nombre courant d'éléments (sa taille).

♦ Les éléments sont **numérotés** (cf. **`Notebook`**)

- Les notes sont maintenues dans l'ordre d'insertion (par défaut) ;
- Un *indice* est associé à chaque élément ;
- Le premier élément est à l'indice 0.

Notes personnelles

♦ Il est de bonne pratique (*depuis le JDK 1.5*) d'indiquer le type des éléments contenus dans une **`ArrayList`**. Ceci n'est pas obligatoire mais lève de nombreuses difficultés…

♦ Par exemple, une liste dont les éléments sont tous des chaînes de caractères se note **`ArrayList<String>`**.

♦ Ce qui est le cas d'un agenda !

♦ Nous allons donc construire la classe **`Notebook`** en exprimant que le stockage se fera dans un champ **`notes`** qui sera un objet de type **`ArrayList<String>`** :

```
ArrayList<String> notes;
```


Un aperçu de la classe ***`Notebook`*** *:*

```
import java.util.ArrayList;

class Notebook {
    ArrayList<String> notes;            // la collection

    Notebook()                          // le constructeur
    { // construit une liste de chaînes de caractères
      notes = new ArrayList<String>();
    }

    int numberOfNotes()                 // les méthodes
    {  ...  }

    void storeNote(String note)
    {  ...  }

    void showNote(int noteNumber)
    {  ...  }
}
```


♦ Le constructeur **`Notebook( )`** de la classe **`Notebook`** initialise le champ **`notes`** en faisant appel au constructeur sans paramètre de la classe paramétrée **`ArrayList<String>`** :

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

♦ La méthode **`numberOfNotes( )`** retourne le nombre d'éléments contenus dans la collection courante :

```
int numberOfNotes( )
{
   return notes.size( );
}
```

♦ Elle utilise la méthode **`size( )`** de la classe **`ArrayList`** :

```
public int size( )
```

classe que l'on trouve documentée dans l'API.

♦ La méthode **`showNote(int n)`** affiche la note numéro **`n`** ≥ 0 :

```
void showNote(int n)
{
   if (n >= 0 && n < this.numberOfNotes())
   {
      System.out.println(notes.get(n));
   }
   // pas de else : sinon on ne fait rien !
}
```

♦ La méthode **`get()`** de la classe **`ArrayList<String>`** :

```
public String get(int index)
```

renvoie l'objet numéro *index* de la collection, type `String`

N.B. De même la classe ***`ArrayList<Integer>`*** *a une méthode*

```
public Integer get(int index)
```

♦ La méthode **`storeNote(String)`** ajoute une nouvelle note à la collection courante :

```
void storeNote(String note)
{
    notes.add(note);
}
```

♦ Elle utilise la méthode **`add(String obj)`** de la classe paramétrée **`ArrayList<String>`**.

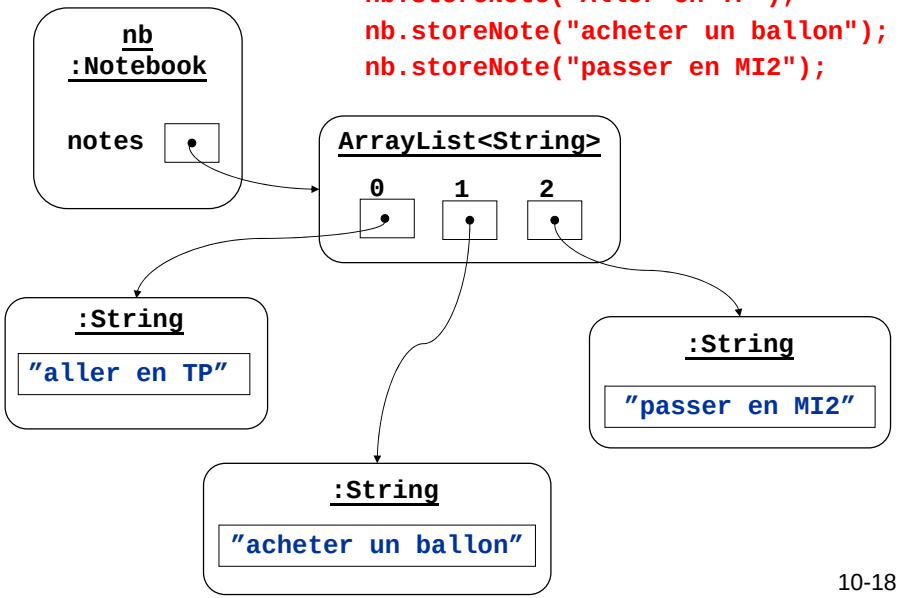
♦ En réalité, la documentation de **`ArrayList`** dans l'API nous indique la signature exacte :

```
public boolean add(String obj)
```

Elle renvoie donc un ***`boolean`***, mais le résultat ne nous intéresse pas ! Java nous autorise à ne pas utiliser ce résultat et à écrire une *instruction* : **`this.add("Aller en TP");`**

♦ Structure d'un agenda (diagramme d'objets UML) :

```
nb.storeNote("Aller en TP");
nb.storeNote("acheter un ballon");
nb.storeNote("passer en MI2");
```



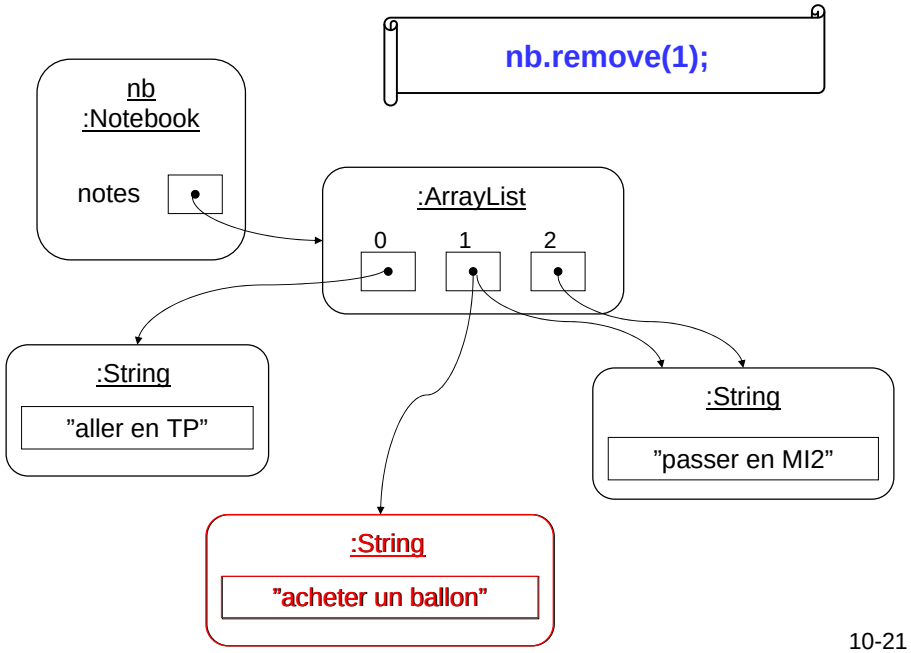
♦ Peut-on supprimer une note de l'agenda, par ex. la note n°1 ?

♦ Plus généralement, peut-on supprimer un élément d'une collection de type **`ArrayList<String>`** ?

```
public String remove(int index)
Removes the element at the specified position
in this list.  Shifts any subsequent elements
to the left (subtracts one from their indices).
```

```
void removeNote (int index)
{
   if (index >= 0 && index < numberOfNotes())
   {
     notes.remove(index);      // on n'utilise pas le résultat !
   }     // pas de else :  sinon ne rien faire…
}
```


Notes personnelles



- Que devient l'objet **"acheter un ballon"** ?
- Cet objet **"acheter un ballon"** devient 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 !
- Le langage Java est muni d'un dispositif de **recyclage automatique des objets inutilisables** ! Ce dispositif se nomme un **GARBAGE COLLECTOR** (GC, 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 !

Traiter *globalement* une collection

- Comment afficher TOUTES les notes ?
- Même s'il y en a 100 ?…

```
showNote(0);
showNote(1);
showNote(2);
...
showNote(98);
showNote(99);
```

Une méthode `listeNotes()`

- On ajoute la méthode `listeNotes()` qui affiche toutes les notes :

```
public void listeNotes() {
  for(int i = 0; i < notes.size(); i+=1)
    System.out.println(notes.get(i));
  }
}
```

```
public void listeNotes() {
  int i = 0;
  while(i < notes.size()) {
    System.out.println(notes.get(i));
    i = i + 1;
  }
}
```

Culture : les notations ++ et --

♦ Il est courant en Java (et en C) d'utiliser l'opérateur **++** qui incrémente la variable qui le précède :

`i++;` équivaut à `i += 1;`

♦ De même pour `i--;`

♦ Mais ATTENTION, en tant qu'expression, `i++` renvoie la valeur ***avant*** l'incrémentation alors que `i += 1` renvoie la valeur ***après*** l'incrémentation !

♦ Morale : utiliser `++;` comme instruction isolée, et **non** au sein d'une expression arithmétique comme dans :

```
i = 4; b = 2 * (i++) - i;
```


Les classes enveloppantes

♦ Rappel : Java distingue deux sortes de données :

- les données de type primitif : **`int`**, **`double`**, **`boolean`**, …

  `456`, `-23.105`, `true`

- les objets, dont le type est une *classe* :

  `nb` de type **`Notebook`**, `"Hello !"` de type **`String`**

♦ Puis-je faire une liste de notes entières ???

```
ArrayList<int> notes;
```

Erreur à la compilation :

`unexpected type : int`

♦ Les **`ArrayList<E>`** ne peuvent contenir que des **objets** ! Pas des valeurs primitives !

- Par exemple **ArrayList<String>** ou **ArrayList<Notebook>**…

♦ Java associe à chaque type primitif (**`int`**, **`double`**, **`boolean`**…) une « **classe enveloppante** » (**`Integer`**, **`Double`**, **`Boolean`**…) qui permet de représenter une donnée primitive comme un objet.

♦ Il faut donc savoir :

- transformer un **int** en **Integer** (« *boxing* ») :

  Integer obj = new Integer(5);

- récupérer l' **int** caché dans un **Integer** (« *unboxing* ») :

  int x = obj.intValue();

Notes personnelles

Les listes d'entiers

- **Integer**, contrairement à **int**, est une classe
- On peut faire des listes de **Integer**.

```
ArrayList<Integer> notes;
```

- On sait donc créer une classe **Etudiant** qui conserve toutes les notes obtenues :

```
class Etudiant {
  ArrayList<Integer> notes;

  Etudiant() {
    notes = new ArrayList<Integer>();
  }
}
```


Auto-boxing (depuis JDK 1.5)

- Pour ajouter une nouvelle note à l'étudiant, on peut utiliser la méthode **add** :

```
public void add(Integer v)
```

```
void storeNote(int note) {
  notes.add(new Integer(note));
}
```

- Heureusement, le JDK 1.5 nous simplifie la vie avec l'auto-boxing

```
void storeNote(int note) {
  notes.add(note);
}
```

note *est automatiquement transformé en* ***Integer***

Auto-unboxing (depuis JDK 1.5)

- On récupére une note par la méthode **get** :

```
public Integer get(int index)
```

```
int sommeNotes() {
  int somme = 0;
  for(int i=0; i<notes.size(); i++) {
    Integer v = notes.get(i);
    somme = somme + v.intValue();
  }
  return somme;
}
```

ou

```
somme = somme + notes.get(i);
```

Le résultat de ***notes.get(i)*** *est un* ***Integer*** *automatiquement transformé en* ***int***

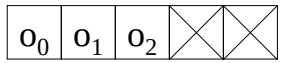
La classe ArrayList et les tableaux

- Une liste ArrayList peut se construire en réalité de deux manières :

public ArrayList()
Constructs an empty list with an initial capacity of 10.

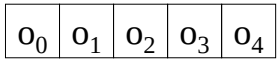
public ArrayList(int initialCapacity)
Constructs an empty list with the specified initial capacity.

- Le principal champ d'une **ArrayList** est un *tableau d'objets* dont certaines cases seulement sont occupées. Les cases libres sont à **null**.



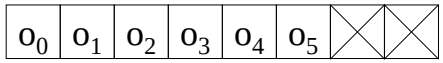
capacity == 5 && size == 3

♦ A force d'ajouter des éléments, on va atteindre la capacité maximale !



capacity == 5 && size == 5

♦ Un autre ajout exige l'*agrandissement du tableau* ! C'est IMPOSSIBLE ! Un nouveau tableau plus long est alloué, l'ancien y est recopié puis est détruit, l'ajout devient possible !

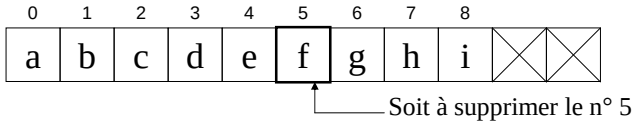


capacity ==8 && size == 6

♦ Le coût pour ajouter un élément est constant sauf lorsque la capacité est maximale. En moyenne, il est donc *constant* !

On paye n pour ajouter n éléments…

♦ Et pour supprimer un élément (méthode **`remove`**) ?

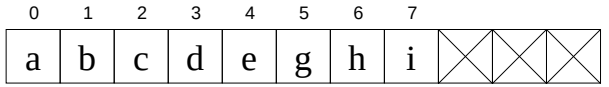


`public E remove(int index)`

Removes the element at the specified position in this list. Shifts any subsequent elements to the left (subtracts one from their indices).

coût ≈ size

```
public E remove(int index)
{
   if (index >= 0 && index < size()) {
  // <<< décaler [index+1, size-1] vers la gauche >>>;
  // <<< mettre null dans la dernière case utilisée >>>;
  // <<< retourner l'objet qui se trouvait au n° index >>>;
   }
}
```



<<< décaler [index+1, size-1] vers la gauche >>>

avec une boucle for de la gauche vers la droite…

<<< mettre null dans la dernière case utilisée >>>

avec une affectation.

<<< retourner l'objet qui se trouvait au n° index >>>

Ah, mais alors il fallait le mettre de côté avant de décaler !

(le sauvegarder)

Notes personnelles

Peut-on paramétrer ses propres classes ?

♦ Il serait manifestement absurde d'avoir à programmer une classe **`Notebook`** et une classe **`IntegerNotebook`** suivant le type des éléments (**`String`** ou **`Integer`**) ! Il suffit de passer ce type **`E`** en paramètre :

Hors programme

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

class GenericNotebook<E> {

    ArrayList<E> notes;

    GenericNotebook() {
        notes = new ArrayList<E>();
    }

    <<< les méthodes >>>
}
```


♦ Les méthodes devront *faire abstraction du type* **`E`** des éléments contenus dans la collection :

Hors programme

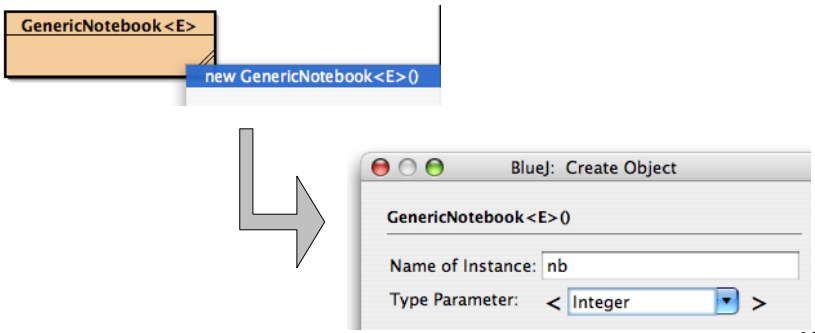
```
void storeNote(E note) {
   notes.add(note);
}

int numberOfNotes() {
   return notes.size();
}

void showNote(int noteNumber) {
   if(noteNumber >= 0 && noteNumber < numberOfNotes()) {
      System.out.println(notes.get(noteNumber));
   }
}

void listeNotesIt() {
   for(E obj : notes) {
      System.out.println(obj);    // en utilisant le toString() de E
   }
```


♦ Comment utiliser cette « classe générique » ?

- dans une autre classe **`TestGenericNotebook`** :

```
GenericNotebook<Integer> nb
    = new GenericNotebook<Integer>();
nb.storeNote(5);
nb.listeNotes();
```

- de manière interactive avec BlueJ :



Notes personnelles