CENTRALE NANTES

Programmation Orientée Objet - Chapitre 3 – Structures de données et Généricité en Java

Jean-Marie Normand
Bâtiment A - Bureau 114
jean-marie.normand@ec-nantes.fr

Plan du cours I

Plan

Plan de la section : Structures de données, Généricité en Java, Collections Java

Dans cette section, nous allons parler de :

- Structures de données : structure logique destinée à stocker de l'information
- Généricité en Java
- Conteneurs et collections de l'API Java

Structures de données

Une structure de données est :

- une structure logique destinée à contenir des données, afin de leur donner une organisation et de simplifier leur traitement
- implémente un type abstrait de données (TAD)

Structures de données

Une structure de données est :

- une structure logique destinée à contenir des données, afin de leur donner une organisation et de simplifier leur traitement
- implémente un type abstrait de données (TAD)

Vous en connaissez déjà des exemples :

- les enregistrements en C/C++
- les variables
- les classes !

Structures de données

Une structure de données est :

- une structure logique destinée à contenir des données, afin de leur donner une organisation et de simplifier leur traitement
- implémente un type abstrait de données (TAD)

Vous en connaissez déjà des exemples :

- les enregistrements en C/C++
- les variables
- les classes !

Mais ces exemples représentent des structures de données finies !

Structures de données

Une structure de données est :

- une structure logique destinée à contenir des données, afin de leur donner une organisation et de simplifier leur traitement
- implémente un type abstrait de données (TAD)

Vous en connaissez déjà des exemples :

- les enregistrements en C/C++
- les variables
- les classes !

Mais ces exemples représentent des structures de données finies !

Nous allons nous concentrer ici sur les structures de données permettant de **stocker** et de **manipuler** un **ensemble de données**

Pourquoi des structures de données ?

Parce qu'elles permettent d'organiser les données afin de les traiter dans leur ensemble, et de réaliser par exemple :

- le **tri** de données
- la **recherche** d'une donnée particulière
- la **suppression** d'une donnée particulière
- l'**insertion** d'une nouvelle donnée (ou de plusieurs) dans la structure

Pourquoi des structures de données ?

La notion de structure de donnée est indépendante à tout langage de programmation, seule la manière de les utiliser dépend du langage cible, mais le concept sous-jacent est indépendant du langage.

Nous nous intéresserons principalement à deux types de structures :

- les structures **linéaires** (aussi appelées structures séquentielles)
- les structures **arborescentes**

Structures linéaires

Les structures de données linéaires (ou séquentielles) sont utiles lorsque l'on souhaite **traiter tous les éléments d'un ensemble de manière séquentielle et de façon identique**.
Les **principales** structures de données linéaires sont les suivantes (nous mentionnons le nom des implémentations Java correspondantes que nous aborderons un peu plus tard) :

- le tableau (`Array` et `ArrayList`)
- la liste chaînée (`LinkedList`)
- la pile (`Stack`)
- la file (`Queue`)
- l'ensemble (`Set`)
- la table de hachage (`HashMap`)

Le vecteur ou tableau

Structure linéaire basique correspondant à un ensemble de cases contigües :

- l'accès aux cases se fait via un indice numérique
- l'indice représente l'emplacement de l'élément dans la structure
- généralement cet indice $\in [0, N]$ pour un tableau de taille $N + 1$

Un tableau peut-être à :

- une dimension

Indices	0	1	2	3	4
Valeurs	6	5	4	3	1

Figure: Tableau simple d'entiers.

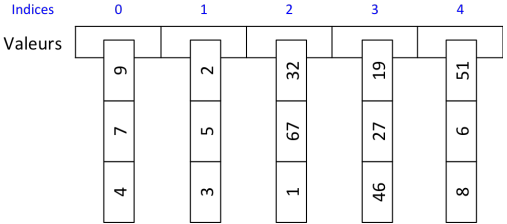
Le vecteur ou tableau

Structure linéaire basique correspondant à un ensemble de cases contigües :

- l'accès aux cases se fait via un indice numérique
- l'indice représente l'emplacement de l'élément dans la structure
- généralement cet indice $\in [0, N]$ pour un tableau de taille $N + 1$

Un tableau peut-être à :

- plusieurs dimensions : vecteur de vecteur



Caractéristiques tableau I

Remarques sur l'efficacité d'opérations sur les tableaux/vecteurs

- permet l'**accès direct à un élément** ! (si l'on connait son indice)
- l'**insertion d'un élément** :
  - en tête de tableau : très mauvais car nécessite un décalage de tous les éléments suivants
  - en milieu de tableau : mauvais car nécessite un décalage de tous les éléments suivants
  - en fin de tableau : très efficace

Caractéristiques tableau II

Remarques sur l'efficacité d'opérations sur les tableaux/vecteurs

- la **suppression d'un élément** :
  - en tête de tableau : très mauvais car nécessite un décalage de tous les éléments suivants
  - en milieu de tableau : mauvais car nécessite un décalage de tous les éléments suivants
  - en fin de tableau : très efficace
- **recherche d'un élément** sans connaitre son indice : parcours de toutes les cases jusqu'à trouver l'élément

La liste chaînée

Une liste chaînée est ensemble fini d'éléments non-contigus mais liés par une relation de séquentialité (on dit aussi chaînés entre eux) :

- le premier élément de la liste n'a pas de prédécesseur
- le dernier élément n'a pas de successeur
- tous les autres éléments ont un prédécesseur et un successeur
- on parle de **liste simplement chaînée** si un **élément possède uniquement un lien vers son successeur**
- on parle de **liste doublement chaînée** si un **élément possède un lien vers son successeur et son prédécesseur**

La liste chaînée

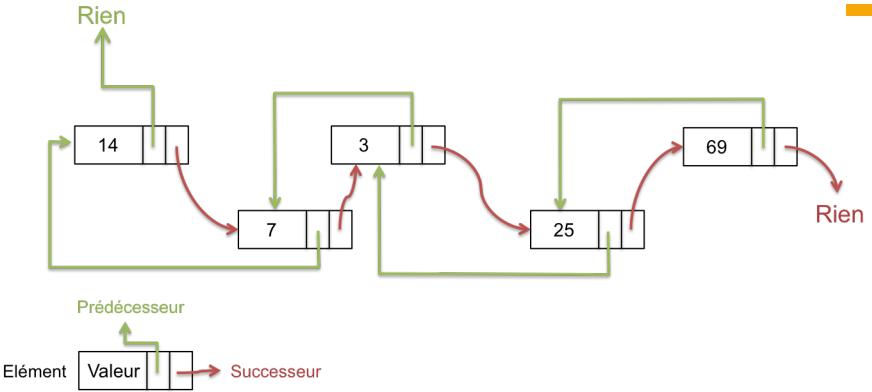


Figure: Une liste d'entiers doublement chainée.

Caractéristiques liste chaînée I

Remarques sur l'efficacité d'opérations sur les listes chaînées

- ne permet **pas d'accès direct à un élément** (même si on connait sa position dans la liste) !
- l'**insertion d'un élément** :
  - en tête de liste : très efficace (on créé une nouvelle cellule et l'on met à jour les liens)
  - en milieu de liste : mauvais (nécessité de parcourir la liste jusqu'à trouver la position où l'on souhaite insérer l'élément)
  - en fin de liste :
    - si l'on possède un **lien direct vers le dernier élément** de la liste : très efficace
    - **sinon** : très mauvais (on doit parcourir toute la liste pour arriver au dernier élément)

Caractéristiques liste chaînée II

Remarques sur l'efficacité d'opérations sur les listes chaînées

- la **suppression d'un élément** :
  - en tête de liste : très efficace (on met à jour les liens puis on supprime la cellule)
  - en milieu de liste : mauvais (nécessité de parcourir la liste jusqu'à trouver la position de l'élément à supprimer)
  - en fin de liste :
    - si l'on possède un **lien direct vers le dernier élément** de la liste : très efficace
    - **sinon** : très mauvais (on doit parcourir toute la liste pour arriver au dernier élément)
- **recherche d'un élément** sans connaitre sa position : parcours de toute la liste jusqu'à trouver l'élément

Comparaison "simpliste" liste chaînée vs. tableau I

Quel choix faire ? Liste chaînée ou tableau ?

- **parcours de tous les éléments** de la structure de donnée : équivalent
- l'**insertion d'un élément** :
  - en fin de structure : avantage tableau (sauf si on stocke le dernier élément de la liste chaînée en + du premier, dans ce cas équivalent)
  - en tête de structure : avantage liste chaînée
  - en milieu de structure : avantage liste chaînée (les décalages sont plus couteux à effectuer que la recherche)
- la **suppression d'un élément** :
  - en fin de structure : avantage tableau (sauf si on stocke le dernier élément de la liste chaînée en + du premier, dans ce cas équivalent)
  - en tête de structure : avantage liste chaînée
  - en milieu de structure : avantage liste chaînée (à cause des décalages)
- **recherche d'un élément** sans connaitre sa position : équivalent

CENTRALE NANTES

Comparaison “simpliste” liste chaînée vs. tableau II

Réponse simpliste :

- si beaucoup d’insertions/suppressions dans la structure de données $\Rightarrow$ **liste chaînée**
- si peu d’insertions/suppressions dans la structure de données ou beaucoup d’accès à des données connaissant leurs indices $\Rightarrow$ **tableau**

Insertion dans une liste chaînée

Illustration

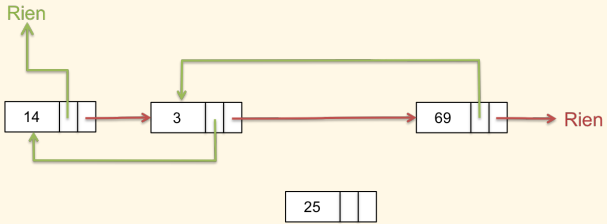


Figure: Insertion dans une liste chainée.

Insertion dans une liste chaînée

Illustration

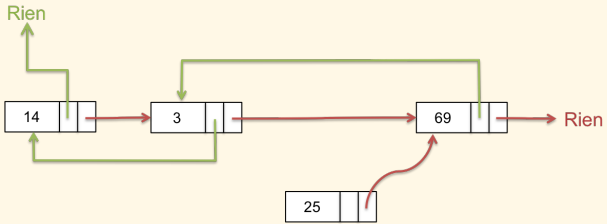


Figure: Insertion dans une liste chainée.

Insertion dans une liste chaînée

Illustration

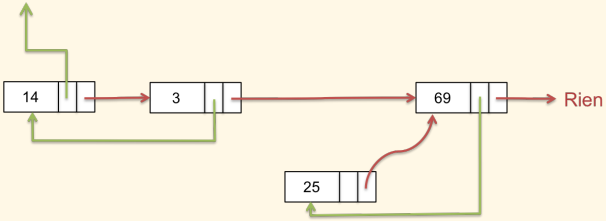


Figure: Insertion dans une liste chainée.

Insertion dans une liste chaînée

Illustration

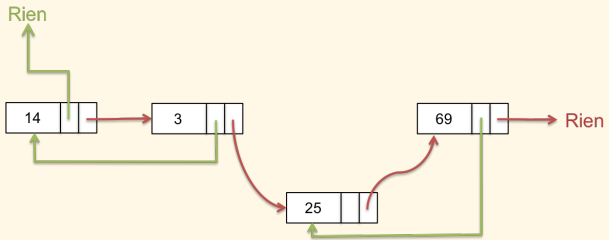


Figure: Insertion dans une liste chainée.

Insertion dans une liste chaînée

Illustration

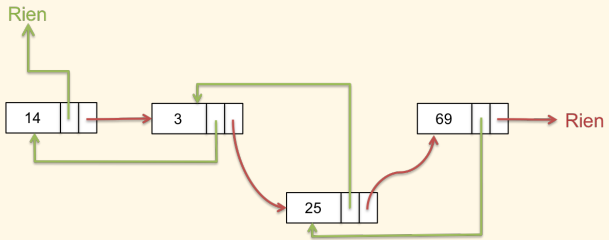


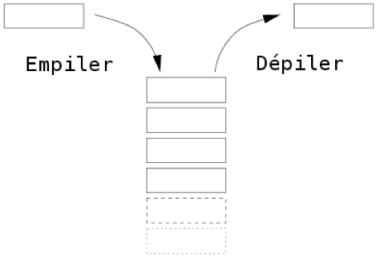
Figure: Insertion dans une liste chainée.

La pile

Une pile est une structure (qui peut être implémentée sous la forme d'une liste chaînée) dont seul le dernier élément est accessible. Les piles sont appelées des structures "LIFO" : *Last In First Out* car lorsqu'on empile un élément il devient le seul accessible (le **sommet** de la pile).

Seules trois opérations sont disponibles sur une pile :

- empiler un élément
- consultation du sommet de la pile
- dépiler le sommet de la pile

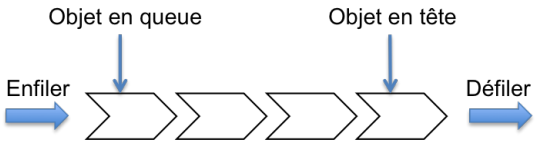


La file

Une file est une structure (qui peut être implémentée sous la forme d'une liste chaînée) dont seuls le premier et le dernier éléments sont accessibles. Les files sont appelées des structures "FIFO" : *First In First Out* car c'est forcément le premier élément inséré qui sera traité.

Seules quatre opérations sont disponibles sur une pile :

- enfiler un élément
- défiler le sommet de la pile
- consultation du premier élément de la file (tête)
- consultation du dernier élément de la file (queue)



L'ensemble

Un ensemble est une structure de donnée qui permet de stocker des éléments de manière unique, **sans doublon**, comme un ensemble mathématique.
Les ensembles bénéficient en plus des opérations "basiques" d'opérations plus spécifiques :

- l'union de deux ensembles
- l'intersection de deux ensembles
- la différence entre deux ensembles

La table de hachage

Un structure de donnée relativement importante est la **table de hachage** qui associe une clé à chaque élément stocké, afin de le retrouver plus rapidement. Une table peut être créée avec un type arbitraire pour les clés (par exemple des chaines de caractères). **Plusieurs éléments peuvent être associés à une même clé**.
L'accès à un élément de la table de hachage se fait via la transformation de la clé en une valeur de hachage (ou simplement hachage) via une **fonction de hachage**.

La table de hachage

Le hachage est un nombre qui permet la localisation des éléments dans le tableau (typiquement il correspond à l'indice de l'élément dans le tableau).
Les tables de hachage bénéficient de **performances excellentes** pour :

- l'ajout d'un élément dans la table de hachage (si la fonction de hachage est rapide)
- la recherche d'un élément dans la table de hachage

cela est particulièrement vrai lorsque l'on a énormément de données à traiter.

La table de hachage : exemple de l'annuaire

Problème

On souhaite implémenter un annuaire qui permette de retrouver facilement le numéro de téléphone en fonction du nom d'une personne (pour simplifier, on suppose qu'il n'y a pas d'homonymes)

La table de hachage : exemple de l'annuaire

Problème

On souhaite implémenter un annuaire qui permette de retrouver facilement le numéro de téléphone en fonction du nom d'une personne (pour simplifier, on suppose qu'il n'y a pas d'homonymes)

Solution proposée

Avoir un tableau indicé par les noms (chaînes de caractères) et y stocker le numéro de téléphone

La table de hachage : exemple de l'annuaire

Problème

On souhaite implémenter un annuaire qui permette de retrouver facilement le numéro de téléphone en fonction du nom d'une personne (pour simplifier, on suppose qu'il n'y a pas d'homonymes)

Solution proposée

Avoir un tableau indicé par les noms (chaînes de caractères) et y stocker le numéro de téléphone

Difficulté technique

On ne sait pas créer des tableaux indicés par des chaînes de caractères

La table de hachage : exemple de l'annuaire

Problème

On souhaite implémenter un annuaire qui permette de retrouver facilement le numéro de téléphone en fonction du nom d'une personne (pour simplifier, on suppose qu'il n'y a pas d'homonymes)

Solution proposée

Avoir un tableau indicé par les noms (chaînes de caractères) et y stocker le numéro de téléphone

Difficulté technique

On ne sait pas créer des tableaux indicés par des chaînes de caractères

Solution proposée

Transformer chaque nom en nombre et stocker le numéro de téléphone dans un tableau indicé par ces nombres

La table de hachage : exemple de l'annuaire

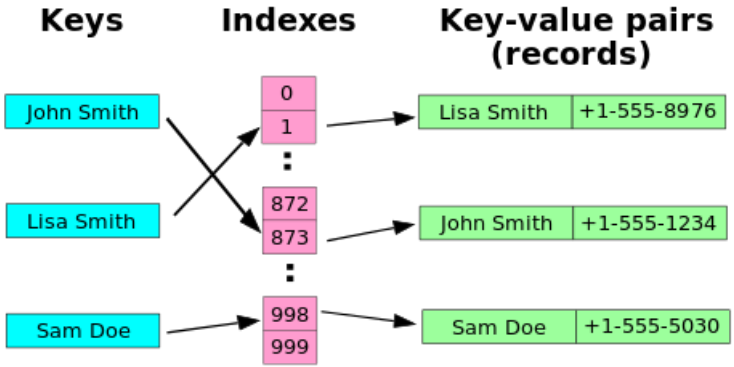


Figure: Table de hachage représentant un annuaire, source Wikipédia.

La table de hachage : exemple de l'annuaire

Problème potentiel : les collisions

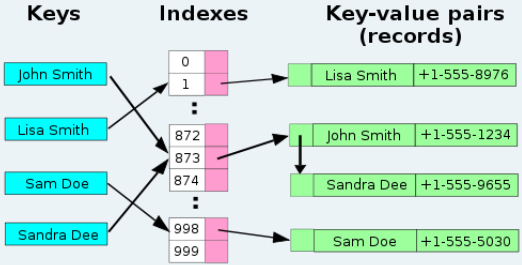
Il est possible que deux entrées (même si le nom est différent) aient la même valeur de hachage, dans ce cas, il existe plusieurs moyens de résoudre ce problème :

La table de hachage : exemple de l'annuaire

Problème potentiel : les collisions

Il est possible que deux entrées (même si le nom est différent) aient la même valeur de hachage, dans ce cas, il existe plusieurs moyens de résoudre ce problème :

Résolution des collisions par chaînage

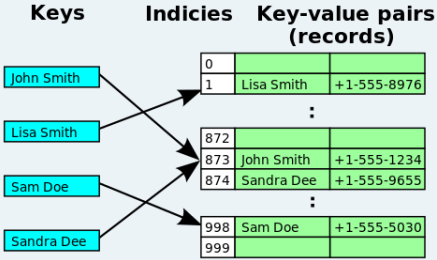


La table de hachage : exemple de l'annuaire

Problème potentiel : les collisions

Il est possible que deux entrées (même si le nom est différent) aient la même valeur de hachage, dans ce cas, il existe plusieurs moyens de résoudre ce problème :

Résolution des collisions par adressage ouvert et sondage linéaire



Les structures arborescentes, les arbres

Les arbres constituent une manière de relier des données de manière plus complexes que par des liens de séquentialité.

Exemples de relations d'arborescence :

- arbre généalogique
- organisation des dossiers et fichiers dans un ordinateur
- relations d'héritage entre classes !

Les arbres peuvent être :

- binaires : chaque nœud de l'arbre a au + 2 fils
- plus généralement n-aires : chaque nœud de l'arbre a au plus n fils

Reportez vous à vos cours d'ALGPR pour plus d'informations, en Java, nous verrons qu'il existe les classes `TreeMap`, `TreeSet`

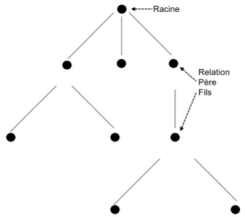


Figure: Un arbre ternaire.

Mini Quiz

Votons à mains levées !

Quelle(s) affirmation(s) est/sont correcte(s) ?

1. La notion de structure de données dépend du langage d'implémentation cible
2. Il revient au même d'utiliser une liste simplement chaînée et un vecteur
3. Les ensembles, comme les listes permettent d'utiliser les opérations ensemblistes (union, différence, intersection)
4. Une pile est une file inversée

Quelle(s) affirmation(s) est/sont correcte(s) ?

1. Le mécanisme d'attente d'impression de fichiers est généralement implémenté par une pile
2. Les tables de hachage sont des structures de données particulièrement intéressantes lorsque la masse de données à manipuler est grande
3. Les arbres permettent de représenter des relations de séquentialité entre objets
4. Les cases d'un vecteur sont forcément contiguës

Plan

Généricité en Java

Rappels Structures de données

Lors de la présentation ci dessus des structures de données classiques, un point primordial a été laissé de côté :

- **que peut-on stocker dans ces structures de données ?**
- précédemment nous avons volontairement laisser ce problème de côté
- pour simplifier dans nos exemples nous avons illustré les structures de données avec des types simples, comme des entiers.

Comment faire en Java ?

- utiliser la notion d'`Object` ?
- utiliser la notion d'encapsulation et de classe ?
- autre solution ?

Généricité sans typage

- En java, toutes les classes dérivent de la classe `Object`
- une *Collection* d'`Object`s peut contenir des objets hétérogènes

Collection hétérogène

```
ArrayList l = new ArrayList<>(); // mauvaise idee voir plus loin
l.add(new Integer(34));
l.add(new String("essai"));
l.add(new Float(34.4));
String c = (String) l.get(1); // marche car on sait que l'element d'indice 1 est de type
    String
for (int i=0 ; i<l.size() ; i++) {
  // on reviendra plus tard sur ces methodes ''bizarres''
  System.out.println("l'element ("+i+") est de type "+l.get(i).getClass().getName());
}
```

Exécution

```
l'element (0) est de type java.lang.Integer
l'element (1) est de type java.lang.String
l'element (2) est de type java.lang.Float
```

Exemple : une classe Couple de chaînes de caractères

Classe représentant un couple de String

```
// classe couple generique
public class CoupleString {
 private String a;
 private String b;
 public CoupleString(String a,String b) {
   this.a = a;
   this.b = b;
 }
```

```
// classe couple generique
 public void echange() {
   String local = a;
   a = b;
   b = local;
 }
}
```

Utilisation de notre classe

```
 public static void main(String[] args) {
   CoupleString c = new CoupleString(new String("2a2"), new String("243"));
   System.out.println(c.getA());
   c.echange();
   System.out.println(c.getA());
 }
```

Exemple : une classe Couple de chaînes de caractères

Classe représentant un couple de `String`

```
// classe couple generique
public class CoupleString {
 private String a;
 private String b;
 public CoupleString(String a,String b) {
   this.a = a;
   this.b = b;
 }
```

```
// classe couple generique
 public void echange() {
   String local = a;
   a = b;
   b = local;
 }
}
```

Fonctionne très bien, mais...

- pour faire une autre classe Couple ?
  - copier/coller du code + renommage (bof)
  - `CoupleObject` et on peut mettre ce qu'on veut dedans

```
  System.out.println(c.getA());
 }
```

Exemple : une classe CoupleObject "générique"

Classe CoupleObject

```
// classe couple generique
public class CoupleObject {
  private Object a;
  private Object b;

  public CoupleObject(Object a,Object b) {
    this.a = a;
    this.b = b;
  }
   ...

  public static void main(Object[] args) {
    CoupleObject c = new CoupleObject(new String("2a2"), new Champignon());
    System.out.println(c.getA());
    c.echange();
    System.out.println(c.getA());
  }
}
```

Bilan

- Bien : ça marche, mais
- on peut faire des couples non-homogènes (pas forcément le but recherché)
- on est obligés de faire des cast à chaque fois pour réutiliser les membres du couple :

Necessité de transtypage !

```
CoupleObject C = new CoupleObject(new Personne(...),new Personne(...));
Personne a = (Personne) C.getA();
```

Généricité en Java

Rappels Structures de données

Lors de la présentation ci dessus des structures de données classiques, un point primordial a été laissé de côté :

- que peut-on stocker dans ces structures de données ?
- précédemment nous avons volontairement laisser ce problème de côté
- pour simplifier dans nos exemples nous avons illustré les structures de données avec des types simples, comme des entiers.

En Java :

Les structures de données sont conçues pour pouvoir contenir des **objets tous du même type, et ce quel que soit ce type** $\Rightarrow$ c'est la **généricité** !!

Généricité en Java II

Il existe une notation spéciale pour les classes génériques en Java :

- le nom de la classe est suivi de la notation `<E>` (parfois noté `<T>`)
- on représente ainsi que cette classe peut stocker n'importe quel **objet** de type `E` (on l'appelle **type paramétrique**), celui-ci :
  - **est inconnu** lors de la déclaration de la classe !
  - mais sera **instancié par un type réel (effectif)** lors de l'utilisation de la classe générique
- par exemple `public class ArrayList<E>` : allez voir la Javadoc en ligne pour voir d'autres exemples !
- nous verrons qu'il est possible qu'une classe soit paramétrée par **plusieurs types génériques**

Exemple `ArrayList<E>`

La notation `ArrayList<E>` signifie que la classe `ArrayList` (qui implémente en Java une liste-tableau) peut stocker des objets de types `String`, `Integer`, ou n'importe quel autre type, mais ce type est inconnu lorsque l'on déclare la classe !

Généricité : exemple I  I

Une classe générique en Java :

```
// Classe adaptee depuis le cours
// d'OpenClassRooms
// Apprenez a programmer en Java
public class ValeurGenerique<T> {
   // Variable d'instance (attribut)
   private T valeur;
   // Constructeur par defaut
   public ValeurGenerique(){
      this.valeur = null;
   }
   // Constructeur avec 1 param.
   // de type inconnu pour l'instant
   public ValeurGenerique(T val){
      this.valeur = val;
```

Généricité : exemple I II

```
    }

    // Definit la valeur avec le parametre
    public void setValeur(T val){
        this.valeur = val;
    }
    // Retourne la valeur avec le bon type !
    public T getValeur(){
        return this.valeur;
    }
}
```

Généricité : exemple I (suite) I

Utilisation d'une classe générique en Java :

```
// Illustration du fonctionnement d'une classe generique
public class TestValeurGenerique {
  // Main
  public static void main(String[] args) {
    // Une 'ValeurGenerique' instanciee avec la classe 'Integer'
    ValeurGenerique<Integer> v1 = new ValeurGenerique<Integer>(23);

    // Une 'ValeurGenerique' instanciee avec la classe 'String'
    ValeurGenerique<String> v2 = new ValeurGenerique<String>("Ma Valeur Generique");

    // Utilisation des methodes
    System.out.println("La valeur generique entiere est : "+v1.getValeur());
    System.out.println("La valeur generique chaine de caracteres est :
    "+v2.getValeur());
```

Généricité : exemple I (suite) II

```
// Modification
v1.setValeur(58);
v2.setValeur("Je change!");

// Verification
System.out.println("La valeur generique entiere est : "+v1.getValeur());
System.out.println("La valeur generique chaine de caracteres est :
"+v2.getValeur());


// Interdit !
v1.setValeur(2.5); // 2.5 n'est pas de type 'Integer' (c'est un 'double')
```

Généricité : exemple I (suite) III

```
    v2.setValeur('a'); // 'a' n'est pas de type 'String' (c'est un 'char')
  }

}
```

Généricité : exemple II I

Une classe doublement générique en Java :

```
public class Couple<T, S> {
   // Variable d'instance de type T
   private T valeur1;
   // Variable d'instance de type S
   private S valeur2;
   // Constructeur par defaut
   public Couple(){
      this.valeur1 = null;
      this.valeur2 = null;
   }
   // Constructeur avec parametres
   public Couple(T val1, S val2){
      this.valeur1 = val1;
      this.valeur2 = val2;
   }
```

Généricité : exemple II  II

```
// Initialisation des deux valeurs
public void setValeurs(T val1, S val2){
   this.valeur1 = val1;
   this.valeur2 = val2;
}

// Retourne la valeur T
public T getValeur1() {
   return valeur1;
}

// Definit la valeur T
public void setValeur1(T valeur1) {
```

Généricité : exemple II III

```
      this.valeur1 = valeur1;
   }


   // Retourne la valeur S
   public S getValeur2() {
      return valeur2;
   }
   // Definit la valeur S
   public void setValeur2(S valeur2) {
      this.valeur2 = valeur2;
   }
}
```

Généricité : exemple II (suite)

Utilisation d'un classe doublement générique en Java

```
// Illustration du fonctionnement d'une classe doublement generique
public class TestCouple {
   // Main
   public static void main(String[] args) {

      Couple<String, Boolean> dual = new Couple<String, Boolean>("toto", true);

      System.out.println("Valeur de l'objet dual : val1 = " + dual.getValeur1() + ", val2 =
     " + dual.getValeur2());

      Couple<Double, Character> dual2 = new Couple<Double, Character>(12.2585, 'C');

      System.out.println("Valeur de l'objet dual2 : val1 = " + dual2.getValeur1() + ", val2
     = " + dual2.getValeur2());
   }
}
```

Héritage de type et conséquences sur la généricité

Nous avons vu précédemment que :

Dans une hiérarchie de classe :

- un objet d'une sous-classe hérite du type de sa super-classe
- l'héritage est transitif !
- un objet peut donc avoir **plusieurs types** !

Héritage de type et conséquences sur la généricité

Nous avons vu précédemment que :

Dans une hiérarchie de classe :

- un objet d'une sous-classe hérite du type de sa super-classe
- l'héritage est transitif !
- un objet peut donc avoir **plusieurs types** !

Conséquences sur la généricité :

Il est effectivement possible de stocker dans une structure de données générique des éléments de types différents mais compatibles !
Par exemple, nous pourrons donc stocker dans une liste des objets des classes `Guerrier`, `Magicien`, `Demoniste`, etc. si nous déclarons notre liste comme étant une liste de `<Personnage>` !

Mini Quiz

Votons à mains levées !

Quelle(s) affirmation(s) est/sont correcte(s) ?

1. En Java il est recommandé d'utiliser la classe `Object` pour gérer des ensembles d'objets
2. Les méthodes d'une classe génériques prennent souvent en paramètre d'entrée des objets du type paramétrique `E`
3. La généricité est pratique pour gérer des collections d'objets hétérogènes
4. La généricité est pratique pour gérer des collections d'objets homogènes

Quelle(s) affirmation(s) est/sont correcte(s) ?

1. Dans une structure de données générique on utilise généralement comme **type générique** (i.e. entre `<>`) la classe la plus basse dans la hiérarchie
2. La généricité alliée au polymorphisme nous permet de gérer facilement des collections de types compatibles
3. Il est conseillé de ré-implémenter vos propres structures de données génériques plutôt que d'utiliser celles de l'API Java

Collections de l'API Java I

Java propose un ensemble de classes permettant de :

- créer des structures de données
- manipuler des structures de données
- stocker des objets dans des structures de données

ces objets Java s'appellent les **conteneurs** !

En Java il existe deux grands types de conteneurs :

- les `Collection` : tableaux, listes et ensembles
- les `Map` : tables de hachage, structures de données utilisant un système "clé-valeur"

Conteneurs Java :

L'ensemble des classes, interfaces (nous y reviendrons) et types définis dans les conteneurs Java vous seront d'une **grande utilité** car ils vous permettent de **manipuler** et **gérer** de **grands ensembles de données** de manière très efficace sans que vous ayez à ré-implémenter des choses par vous mêmes !

CENTRALE NANTES

Collections de l'API Java II

Quelques informations utiles :

- toutes les classes, interfaces, etc. des `Collection` et des `Map` se trouvent dans le package : `java.util`
- toutes ces classes sont **génériques** !
- les objets d'un type de **conteneur** sont :
  - **des objets avant tout** : il faut donc les **créer**, etc. comme n'importe quel autre objet
  - qui servent à contenir d'autres objets
- utilisation "classique" : *"Un garage contient des véhicules"* $\Rightarrow$ un objet de type `Garage` va posséder un objet de type conteneur qui va quant à lui contenir des objets de type `Vehicule`
- **LISEZ LA JAVADOC EN LIGNE !!** : http://docs.oracle.com/javase/8/docs
- il faut choisir la bonne structure de donnée en fonction de ce que vous souhaitez en faire

Collections de l'API Java II

Nous allons présenter ici uniquement les structures de données principales, à savoir :

- les tableaux (vecteurs) : classe `ArrayList`
- les listes (doublement) chaînées : classe `LinkedList`
- les ensembles : classe `TreeSet`
- les tables de hachage : classe `HashMap`

Cependant, vous devez utiliser la documentation en ligne pour voir si une structure de donnée particulière n'existe pas déjà avant de vouloir en créer une vous même :

http://docs.oracle.com/javase/8/docs/api/java/util/package-summary.html

La Classe ArrayList<E>

Classe générique permettant de stocker des données sous forme de tableau (chaque élément est indexé par un entier), documentation :
http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Déclaration et création d'un objet de type ArrayList

```
// dans un fichier .java
//...
Personnage persoP = new Personnage(...); // cree un objet Personnage
Magicien persoM = new Magicien(...); // je cree un objet Magicien
ArrayList<Personnage> monTabPerso = new ArrayList<>();
// appel au constructeur sans parametre
// il en existe d'autres, voir la documentation

monTabPerso.add(persoP); // ajout du personnage en fin de tableau
monTabPerso.add(persoM); // ajout du magicien en fin de tableau
```

Principales caractéristiques de `ArrayList<E>`

Principales méthodes

- `ArrayList<E>()` : constructeur sans paramètre qui construit l'objet représentant un "tableau vide"
- `add(E e)` : ajoute l'élément `e` de type inconnu `E` à la fin du tableau
- `get(int index)` : retourne l'élément se trouvant à l'indice `index` dans le tableau
- `size()` : retourne le nombre d'éléments contenus dans le tableau
- `remove(int index)` : supprime l'élément se trouvant dans la case d'indice `index`
- `clear()` : vide le tableau

Pour plus d'informations sur les méthodes proposées par cette classe, allez voir la documentation en ligne !
http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

La Classe LinkedList<E>

Classe générique permettant de stocker des données sous forme de liste chaînée (doublement chaînée), documentation :
http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Déclaration et création d'un objet de type LinkedList

```
// dans un fichier .java
//...

Personnage persoP = new Personnage(...); // je cree un objet Personnage
Magicien persoM = new Magicien(...); // je cree un objet Magicien
LinkedList<Personnage> maListePerso = new LinkedList<>();
// appel au constructeur sans parametre
// il en existe d'autres, voir la documentation

maListePerso.add(persoP); // ajoute du personnage enfin de tableau
maListePerso.add(persoM); // ajout du magicien en fin de tableau
```

Principales caractéristiques de `LinkedList<E>`

Principales méthodes

- `LinkedList<E>()` : constructeur sans paramètre qui construit l'objet représentant une "liste vide"
- `add(E e)` : ajoute l'élément `e` de type inconnu `E` à la fin de la liste
- `get(int index)` : retourne l'élément se trouvant à l'indice `index` dans la liste
- `size()` : retourne le nombre d'éléments contenus dans le tableau
- `remove(int index)` : supprime l'élément se trouvant dans la case d'indice `index`
- `clear()` : vide le tableau

Pour plus d'informations sur les méthodes proposées par cette classe, allez voir la documentation en ligne !
http://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html

Petite remarque sur ArrayList<E> et LinkedList<E>

Leurs principales méthodes portent le même nom !

C'est très pratique pour les développeurs qui peuvent ainsi passer de l'une à l'autre structure de donnée de manière très très simple : en changeant le type des objets conteneurs, et c'est presque tout car ces classes ont beaucoup de méthodes en commun.

Comparaisons opérations ArrayList et LinkedList

Opération (cas moyen)	ArrayList	LinkedList
get(index)	O(1)	O(n/4)
add(E)	O(1) amorti et O(n) pire cas	O(1)
add(E, index)	O(n/2)	O(n/4)
remove(index)	O(n/2)	O(n/4)
Iterator.remove()	O(n/2)	O(1)
Iterator.add(E)	O(n/2)	O(1)

Voir Ce post

StackOverflow

La Classe TreeSet<E>

Classe générique permettant de stocker des données sous forme d'arbre ordonné (les éléments contenus doivent donc pouvoir être ordonnées), stockées sous forme d'arbre binaire documentation :
http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html
Notez qu'il existe également la classe HashSet<E> qui représente un **ensemble non ordonné** !

Déclaration et création d'un objet de type TreeSet

```
// dans un fichier .java
//...
TreeSet<Integer> tree = new TreeSet<>();
tree.add(12);
tree.add(63);
tree.add(34);
tree.add(63);
// Un 'TreeSet', comme tout 'Set' ne contient pas de doublons !
```

Principales caractéristiques de `TreeSet<E>`

Principales méthodes

- `TreeSet<E>()` : constructeur sans paramètre qui construit l'objet représentant un "ensemble vide"
- `add(E e)` : ajoute l'élément `e` de type inconnu `E` dans l'ensemble si celui-ci n'est pas déjà présent
- `size()` : retourne le nombre d'éléments contenus dans le tableau
- `remove(Object o)` : supprime l'élément `o` si celui-ci se trouve dans l'ensemble
- `clear()` : vide l'ensemble

Pour plus d'informations sur les méthodes proposées par cette classe, allez voir la documentation en ligne !
http://docs.oracle.com/javase/8/docs/api/java/util/TreeSet.html

La Classe HashMap<K,V>

Classe générique permettant de stocker des données sous forme de table de hachage (sous forme de couple **clé-valeur**), documentation :
http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Déclaration et création d'un objet de type HashMap

```
// dans un fichier .java
//...
HashMap<String,Double> balance = new HashMap<String,Double>();

balance.put("Tom Smith", new Double(123.22));
balance.put("Jane Baker", new Double(1378.00));
balance.put("Todd Hall", new Double(99.22));
balance.put("Ralph Smith", new Double(-19.08));
```

Principales caractéristiques de `HashMap<K,V>`

Principales méthodes

- `HashMap<K,V>()` : constructeur sans paramètre qui construit l'objet représentant une "table de hachage vide" avec le type de clé `K` et le type de valeur `V`
- `put(K k, V v)` : associe l'élément de valeur `v` avec la clé `k` dans la table de hachage
- `size()` : retourne le nombre de couples **clé-valeur** contenus dans la table de hachage
- `remove(Object key)` : supprime l'élément dont la clé est `key` si celui-ci se trouve dans la table de hachage
- `clear()` : vide la table de hachage

Pour plus d'informations sur les méthodes proposées par cette classe, allez voir la documentation en ligne !

http://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

Parcours des conteneurs Java

- Il est nécessaire de pouvoir parcourir les conteneurs Java pour accéder aux objets qui y sont stockés !
- Pour les tableaux/listes chaînées : on peut utiliser les indices

Parcours de liste ou de tableau

```
// parcours de la liste maListePerso
for(int i=0; i< maListePerso.size(); i++) {
  System.out.println(maListePerso.get(i));
}
```

Attention avec la méthode `get`

Les performances avec cette méthode pour des objets de type `LinkedList` sont très mauvaises !

- Mais les ensembles et les tables de hachage n'**ont pas d'indices** !
- Java offre un mécanisme commun à tous les conteneurs pour leurs parcours : les

Itérateur

- Chaque classe représentant un **conteneur** Java peut être parcouru par grâce aux **itérateurs** Java
- Un itérateur est un objet Java (de l'interface `Iterator<E>`) qui joue le rôle de "curseur" en désignant une position, avant ou après un objet dans une structure de données
- Un itérateur est créé en appelant la méthode `iterator()` sur l'objet de type conteneur (liste, tableau, etc.) :
  `Iterator<TypeGeneriqueDansLaListe> listIt = list.iterator();`
- Attention l'interface `Iterator` est **générique** ! Si vous souhaitez pouvoir utiliser une méthode de la classe générique stockée dans le conteneur Java (`ArrayList` ou `LinkedList`) il faut bien déclarer l'itérateur comme un objet générique

Itérateur

- À la création d'un itérateur, celui-ci est positionné avant le premier élément de la collection

Figure: Itérateur sur une liste : au départ il est placé avant le premier élément de la liste.

- Deux méthodes sont suffisantes pour gérer les itérateurs :
  - la méthode `hasNext()` : indique si il existe un élément suivant et renvoie 'vrai' dans ce cas
  - la méthode `next()` : se déplace sur le prochain élément et le renvoie

Itérateur : mode d'emploi

Parcours de liste ou de tableau

```
// parcours de la liste avec iterateur
Iterator<Personnage> listIt = maListePerso.iterator();
while(listIt.hasNext()) {
  // je peux recuperer directement un Personnage car
  // l'iterateur est declarer comme Iterator<Personnage>
  Personnage p = listIt.next();
  System.out.println(p);
  // je peux ici appeler une methode de la classe Personnage
  p.affiche();
}
```

Itérateur : mode d'emploi (2)

- Depuis Java 6, il existe une structure de contrôle (boucle `for`) dédiée au parcours de collections
- Cette structure est basée **implicitement sur les itérateurs**
- Il s'agit seulement de "sucre syntaxique" dont le but est de simplifier la vie aux développeurs en leur faisant écrire moins de code

Sucre syntaxique pour le parcours de liste

```
// maListePerso est une LinkedList<Personnage>, parcours avec sucre
    syntaxique
for(Personnage p : maListePerso) {
  System.out.println(p);
}
```

Des exemples avec les différentes structures de données

Vous trouverez sur le serveur pédagogique des exemples de code sur les principales structures de données avec utilisation des itérateurs :

Exemple tableau et liste chaînée `ArrayList` et `LinkedList`

Exemple ensemble `HashSet`

Exemple table de hachage `HashMap`

Bilan généricité et collections

- Les classes génériques (et donc les collections) connaissent un fonctionnement similaire à celui des classes spécifiques pour chaque valeur du type paramétrique `E`
- Dans une classe `ValeurGenerique<E>`, on peut utiliser toute classe dérivant de `E`
- Leur utilisation est transparente pour l'utilisateur

Mini Quiz

Votons à mains levées !

Quelle(s) affirmation(s) est/sont correcte(s) ?

1. En Java il est recommandé d'utiliser les `Collection`s pour gérer des ensembles d'objets
2. Il est relativement simple de passer d'une `ArrayList` à une `LinkedList`
3. Il est toujours plus efficace d'utiliser une `ArrayList`
4. Les conteneurs n'ont pas besoin d'être créés

Quelle(s) affirmation(s) est/sont correcte(s) ?

1. Il n'est pas rare d'avoir des conteneurs comme attributs de classe
2. Il est plus efficace de faire des boucles de parcours basées sur le nombre d'éléments d'un conteneur que d'utiliser les itérateurs
3. L'utilisation de la boucle `for` avec les itérateurs est moins performante que celle utilisant un `while`
4. Attention à l'utilisation de sous-types avec la généricité