INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES CENTRE VAL DE LOIRE

Java

benjamin.nguyen@insa-cvl.fr

d’après J.-M. Doudoux

But du cours (compétences

- Savoir coder une application complexe en Java (avec des tests unitaires)
- Savoir débugger une application complexe en Java
- Savoir gérer les fichiers en Java
- Savoir gérer la persistence en Java avec la sérialisation
- Savoir utiliser les threads en Java
- Savoir créer une interface graphique simple en Java (approche modèle MVC)

Si le temps le permet …

- Connaître les bibliothèques cryptographiques en Java
- Comprendre et avoir utilisé le mécanisme d'introspection en Java
- Savoir gérer la persistence en Java avec une base de données

Programme du cours

Cours 1

- Rappels (Classes, Collections)

Cours 2

- Fonctionnement de la JVM

Cours 3

- Tests Unitaires

Cours 4

- Fichiers et Sérialisation

Cours 5

- Les Threads

Cours 6

- Interfaces graphiques

Cours 7

- Cryptographie

Cours 8

- Introspection

Cours 9

- Objets et bases de données

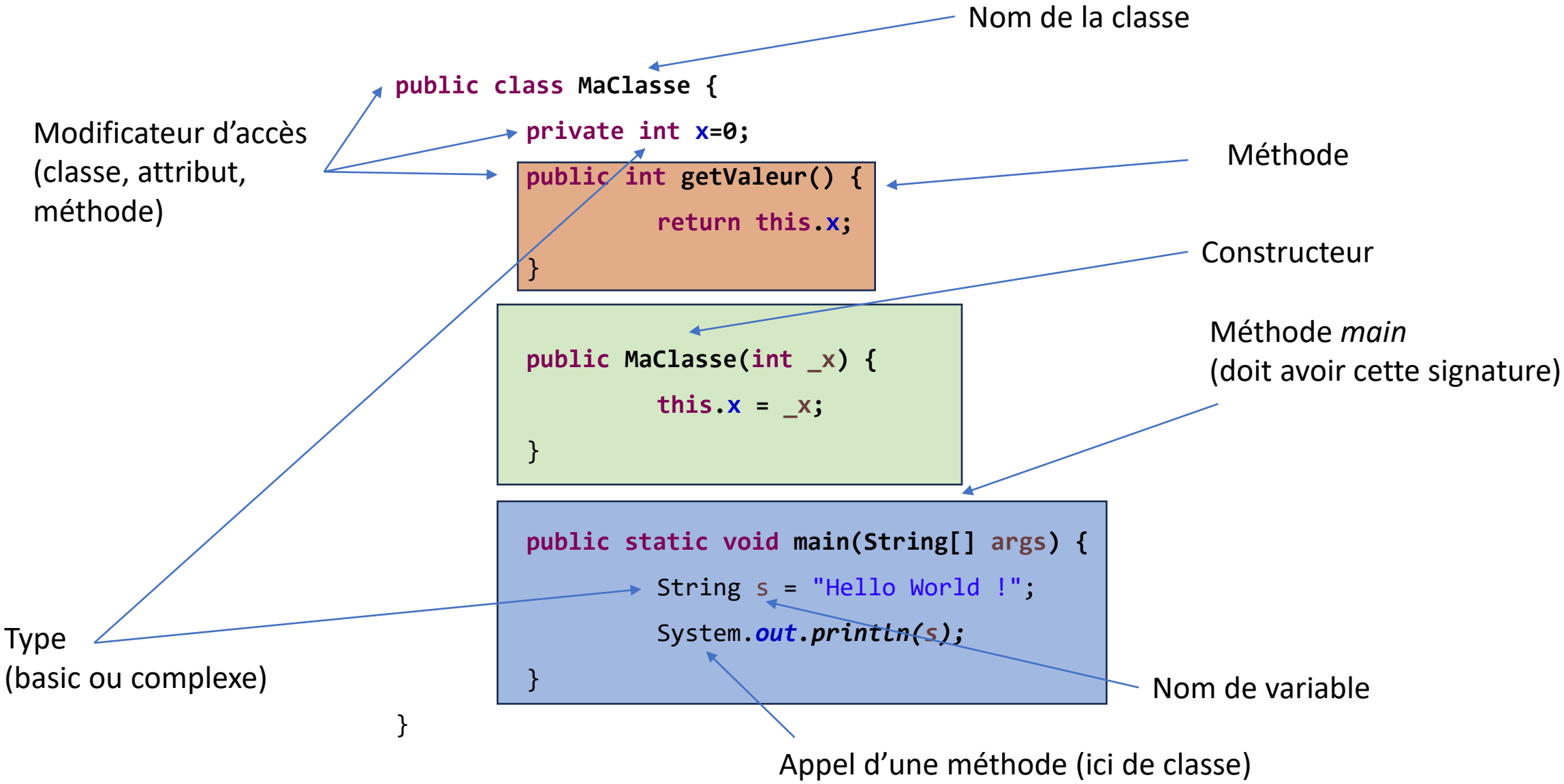
Ressources

- Site de Jean-Michel Doudoux : cours de 4000 pages https://www.jmdoudoux.fr/accueil_java.htm#dej

Rappels

Classe et objets

Une classe en java



Accessibilité

	private		protected	Public
La classe	X	X	X	X
Même package et sous classe		X	X	X
Même package et pas sous classe		X	X	X
Différent package et sous classe			X	X
Différent package et pas sous classe				X

Possibilité de référencer dans notre code les classes, attributs et méthodes

La classe

- Correspond à une *classe* en UML
- Contient des attributs (potentiellement avec des liens vers d'autres classes)
- Contient des méthodes (d'objet ou de classe)
- Contient un *constructeur* ou une *usine* qui permet de créer des objets (pour le constructeur, de la *classe* en question).
- Peut hériter d'une autre classe (**extends / implements**)
- Peut être abstraite (pas complètement implémentée) ou pas du tout implémentée (**interface**)

Création d’objets

- Pour créer un objet, on dit qu’on *l’instancie* (on crée une nouvelle *instance* de la classe)
- Se fait en appelant le constructeur (sous réserve d’accessabilité)

Démonstration

- Introduction à Eclipse
- Exemple1
- Utilisation du debugger

Héritage

Concept

L'héritage est un mécanisme qui facilite la réutilisation du code et la gestion de son évolution. Elle définit une relation entre deux classes :

- une classe mère ou super-classe
- une classe fille ou sous-classe qui hérite de sa classe mère

Intérêt

Grâce à l'héritage, les objets d'une classe fille ont accès aux données et aux méthodes de la classe parente et peuvent les étendre.

Les sous-classes peuvent redéfinir les variables et les méthodes héritées.

- Pour les variables, il suffit de les redéclarer sous le même nom avec un type différent.
- Les méthodes sont redéfinies avec le même nom, les mêmes types et le même nombre d'arguments, sinon il s'agit d'une surcharge.

Intérêt

- L'héritage successif de classes permet de définir une hiérarchie de classe qui se compose de super-classes et de sous-classes. Une classe qui hérite d'une autre est une sous-classe et celle dont elle hérite est une super-classe. Une classe peut avoir plusieurs sous-classes.
- Une classe ne peut avoir qu'une seule classe mère : **il n'y a pas d'héritage multiple en Java. /!\ Interfaces**
- **Object** est la classe parente de toutes les classes en Java. Toutes les variables et méthodes contenues dans Object sont accessibles à partir de n'importe quelle classe car par héritages successifs toutes les classes héritent d'Object.

Mise en oeuvre

- On utilise le mot clé **extends** pour indiquer qu'une classe hérite d'une autre. En l'absence de ce mot réservé associé à une classe, le compilateur considère la classe Object comme classe mère.
- Pour invoquer une méthode d'une classe mère, il suffit d'indiquer la méthode préfixée par **super**. Pour appeler le constructeur de la classe mère, il suffit d'écrire **super(paramètres)** avec les paramètres adéquats.
- Le lien entre une classe fille et une classe mère est géré par la plate-forme : une évolution des règles de gestion de la classe mère conduit à modifier automatiquement la classe fille dès que cette dernière est recompilée.
- En Java, il est obligatoire dans un constructeur d'une classe fille de faire appel **explicitement ou implicitement** au constructeur de la classe mère.

Accès aux attributs et méthodes héritées

- Les variables et méthodes définies avec le modificateur d'accès **public** restent publiques à travers l'héritage et toutes les autres classes.
- Une variable d'instance définie avec le modificateur **private** est bien héritée mais elle **n'est pas accessible directement** mais par les méthodes héritées.
- Une variable définie avec le modificateur **protected** sera héritée dans toutes les classes filles qui pourront y accéder librement ainsi que les classes du même package.

Redéfinition / surcharge de méthodes héritée

- La **redéfinition** d'une méthode héritée doit impérativement conserver la déclaration de la méthode parente (type et nombre de paramètres, la valeur de retour et les exceptions propagées doivent être identiques).
- Si la signature de la méthode change, ce n'est plus une **redéfinition** mais une **surcharge**. Cette nouvelle méthode n'est pas héritée : la classe mère ne possède pas de méthode possédant cette signature.
- Dans ce cas, la forme de la méthode à exécuter est choisie en fonction des paramètres associés à l'appel.

Transtypage

L'héritage définit un cast implicite de la classe fille vers la classe mère : on peut affecter à une référence d'une classe n'importe quel objet d'une de ses sous-classes.

Supposons qu'Employe soit une sous-classe de Personne

```
Personne p = new Personne ("Dupond", "Jean");
Employe e = new Employe("Durand", "Julien", 10000);
p = e ; // ok : Employe est une sous-classe de Personne
Objet obj;
obj = e ; // ok : Employe hérite de Personne qui elle même hérite de Object
Personne[] tab = new Personne[10];
tab[0] = new Personne("Dupond","Jean");
tab[1] = new Employe("Durand", "Julien", 10000);
```

Interfaces et héritage multiple

- Héritage multiple en Java ? Possible à travers les interfaces /!\ pas de possibilité de faire de l'héritage multiple sur des attributs.
- Une **interface** est un ensemble de **constantes** et de **déclarations de méthodes** correspondant un peu à une classe abstraite. C'est une sorte de standard auquel une classe peut répondre. Tous les objets qui se conforment à cette interface (qui implémentent cette interface) possèdent les méthodes et les constantes déclarées dans celle-ci. Plusieurs interfaces peuvent être implémentées dans une même classe.
- Les seules variables que l'on peut définir dans une interface sont des **variables de classe qui doivent être constantes** : elles sont donc implicitement déclarées avec le modificateur **static** et **final** même si elles sont définies avec d'autres modificateurs.
- Les **interfaces** se déclarent avec le mot clé **interface** et sont intégrées aux autres classes avec le mot clé **implements**.
- Une interface est implicitement déclarée avec le modificateur abstract.

Méthodes par défaut

- Problème : si on modifie une interface, il faut réimplémenter toutes les classes
- Proposition : méthodes par défaut
- Une méthode par défaut est déclarée en utilisant le mot clé default. Le corps de la méthode contient l'implémentation des traitements.

```
public interface MonInterface {
  default void maMethode() {
    System.out.println("Implementation par defaut");
  }
}
```

Les méthodes par défaut devraient surtout être utilisées pour maintenir une compatibilité ascendante afin de permettre l'ajout d'une méthode à une interface existante sans avoir à modifier les classes qui l'implémentent.

Choix de la méthode

- L'héritage multiple de comportement permet par exemple que de mêmes méthodes par défaut, avec des signatures identiques, soient définies dans plusieurs interfaces héritées par une interface fille.
- Il est probable que chaque implémentation de ces méthodes soit différente : le compilateur a besoin de règles pour déterminer quelle implémentation il doit utiliser :
  - la redéfinition d'une méthode par une classe ou une super-classe est toujours prioritaire par rapport à une méthode par défaut
  - l'implémentation choisie est celle par défaut de l'interface la plus spécifique
- Ainsi une interface peut être modifiée en ajoutant une méthode sans compromettre sa compatibilité ascendante sous réserve qu'elle implémente cette méthode en tant que méthode par défaut.

Conseils sur l’héritage

Lors de la création d'une classe « mère » il faut tenir compte des points suivants :

- la définition des accès aux variables d'instances, très souvent privées, doit être réfléchie entre protected et private
- pour empêcher la redéfinition d'une méthode ou sa surcharge, il faut la déclarer avec le modificateur final

Lors de la création d'une classe fille, pour chaque méthode héritée qui n'est pas final, il faut envisager les cas suivants :

- la méthode héritée convient à la classe fille : on ne doit pas la redéfinir
- la méthode héritée convient mais partiellement du fait de la spécialisation apportée par la classe fille : il faut la redéfinir voire la surcharger. La plupart du temps une redéfinition commencera par appeler la méthode héritée (en utilisant le mot clé super) pour garantir l'évolution du code
- la méthode héritée ne convient pas : il faut redéfinir ou surcharger la méthode sans appeler la méthode héritée lors de la redéfinition.

Collections

Avant les collections …

- Utilisation de tableaux : []
- Limitations (en particulier gestion de la mémoire)
- Manque de méthodes génériques implémentées sur cette structure (e.g. recherche)

Types de collections

- java.util.Collection : gestion de groupes d’objets
- java.util.Map : gestion de paires (clé, valeur)

- Caractéristiques :
  - Recherche et parcours (interface Iterator et ListIterator)
  - Comparaison (interface Comparable et classe Comparator)
  - Concurrence
- Définition de classes abstraites / interfaces
- Plusieurs possibilités d’implémentation

Interfaces à utiliser

Collection : interface implémentée par la plupart des objets collection

Map : clé/valeur

Set : si pas de doublons

List : doublons et accès direct

SortedSet : étend l'interface Set et permet d'ordonner l'ensemble

SortedMap : étend l'interface Map et permet d'ordonner l'ensemble

Classes concrêtes

Le framework propose plusieurs objets qui implémentent ces interfaces et qui peuvent être directement utilisés :

- **HashSet** : Hashtable qui implémente l'interface Set
- **TreeSet** : arbre qui implémente l'interface SortedSet
- **ArrayList** : tableau dynamique qui implémente l'interface List
- **LinkedList** : liste doublement chaînée (parcours de la liste dans les deux sens) qui implémente l'interface List
- **HashMap** : Hashtable qui implémente l'interface Map
- **TreeMap** : arbre qui implémente l'interface SortedMap
- ***Vector*** *: tableau à taille variable qui implémente maintenant l'interface List*
- ***Hashtable*** *: table de hachage qui implémente maintenant l'interface Map*

Anciennes classes de Java 2 mises à jour

Classes complémentaires

Le framework définit aussi des interfaces pour faciliter le parcours des collections et leur tri :

- **Iterator** : interface pour le parcours des collections
- **ListIterator** : interface pour le parcours des listes dans les deux sens et pour modifier les éléments lors de ce parcours
- **Comparable** : interface pour définir un ordre de tri naturel pour un objet
- **Comparator** : interface pour définir un ordre de tri quelconque

Caractéristiques de chaque type

Collection	**Ordonné**	**Accès direct**	**Clé / valeur**	**Doublons**	**Null**	**Thread Safe**
ArrayList	Oui	Oui	Non	Oui	Oui	Non
LinkedList	Oui	Non	Non	Oui	Oui	Non
HashSet	Non	Non	Non	Non	Oui	Non
TreeSet	Oui	Non	Non	Non	Non	Non
HashMap	Non	Oui	Oui	Non	Oui	Non
TreeMap	Oui	Oui	Oui	Non	Non	Non
Vector	Oui	Oui	Non	Oui	Oui	Oui
Hashtable	Non	Oui	Oui	Non	Non	Oui
Properties	Non	Oui	Oui	Non	Non	Oui
Stack	Oui	Non	Non	Oui	Oui	Oui
CopyOnWriteArrayList	Oui	Oui	Non	Oui	Oui	Oui
ConcurrentHashMap	Non	Oui	Oui	Non	Non	Oui
CopyOnWriteArraySet	Non	Non	Non	Non	Oui	Oui

Il y en a d'autres …

	Utilisation générale	**Utilisation spécifique**	**Gestion des accès concurrents**
List	ArrayList LinkedList	CopyOnWriteArrayList	Vector Stack CopyOnWriteArrayList
Set	HashSet TreeSet LinkedHashSet	CopyOnWriteArraySet EnumSet	CopyOnWriteArraySet ConcurrentSkipListSet
Map	HashMap TreeMap LinkedHashMap	WeakHashMap IdentityHashMap EnumMap	Hashtable ConcurrentHashMap ConcurrentSkipListMap
Queue	LinkedList ArrayDeque PriorityQueue		ConcurrentLinkedQueue LinkedBlockingQueue ArrayBlockingQueue PriorityBlockingQueue DelayQueue SynchronousQueue LinkedBlockingDeque

Méthodes de l’interface Collection

Méthode	Rôle
boolean add(E e)	Ajouter un élément à la collection (optionnelle)
boolean addAll(Collection<? extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection (optionnelle)
void clear()	Supprimer tous les éléments de la collection (optionnelle)
boolean contains(Object o)	Retourner un booléen qui précise si l'élément est présent dans la collection
boolean containsAll(Collection<?> c)	Retourner un booléen qui précise si tous les éléments fournis en paramètres sont présents dans la collection
boolean equals(Object o)	Vérifier l'égalité avec la collection fournie en paramètre
int hashCode()	Retourner la valeur de hachage de la collection
boolean isEmpty()	Retourner un booléen qui précise si la collection est vide
Iterator<E> iterator()	Retourner un Iterator qui permet le parcours des éléments de la collection
boolean remove(Object o)	Supprimer un élément de la collection s'il est présent (optionnelle)
boolean removeAll(Collection<?> c)	Supprimer tous les éléments fournis en paramètres de la collection s'ils sont présents (optionnelle)
boolean retainAll(Collection<?> c)	Ne laisser dans la collection que les éléments fournis en paramètres : les autres éléments sont supprimés (optionnelle). Elle renvoie un booléen qui précise si le contenu de la collection a été modifié
int size()	Retourner le nombre d'éléments contenus dans la collection
Object[] toArray()	Retourner un tableau contenant tous les éléments de la collection
<T> T[] toArray(T[] a)	Retourner un tableau typé de tous les éléments de la collection

Implémentation

- Certaines méthodes de cette interface peuvent lever une exception de type UnsupportedOperationException car leur implémentation est optionnelle : add(), addAll(), remove(), removeAll, retainAll() et clear(). Cette exception peut aussi être levée si l'opération n'a aucune influence sur l'état de la collection.
- Chaque implémentation est libre de :
  - gérer ou non les accès concurrents
  - utiliser l'algorithme de son choix pour tester l'égalité d'un élément (equals(), hashCode()/equals(), ...)
  - utiliser à son avantage les fonctionnalités proposées par les éléments (implémentation de Comparable, ...)

Méthodes de l’interface List

Méthode	Rôle
void add(int index, E e)	Ajouter un élément à la position fournie en paramètre
boolean addAll(int index, Collection<? extends E> c)	Ajouter des éléments à la position fournie en paramètre
E get(int index)	Retourner l'élément à la position fournie en paramètre
int indexOf(Object o)	Retourner la première position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
int lastIndexOf(Object o)	Retourner la dernière position dans la liste du premier élément fourni en paramètre. Elle renvoie -1 si l'élément n'est pas trouvé
ListIterator<E> listIterator()	Renvoyer un Iterator positionné sur le premier élément de la liste
ListIterator<E> listIterator(int indx)	Renvoyer un Iterator positionné sur l'élément dont l'index est fourni en paramètre
E remove(int index)	Supprimer l'élément à la position fournie en paramètre
E set(int index, E e)	Remplacer l'élément à la position fournie en paramètre
List<E> subList(int fromIndex, int toIndex)	Obtenir une liste partielle de la collection contenant les éléments compris entre les index fromIndex inclus et toIndex exclus fournis en paramètres

Classe Vector

La classe Vector, présente depuis Java 1.0 (!), est un tableau dont la taille peut varier selon le nombre d'éléments qu'il contient.

Lors de la création d'une instance de type Vector, il est possible de lui préciser une capacité initiale et une taille d'incrémentation en utilisant la surcharge correspondante du constructeur.

La classe Vector est antérieure à l'API Collections : elle a été mise à jour ultérieurement pour implémenter l'interface Liste. Il y a de ce fait plusieurs méthodes redondantes comme par exemple les méthodes add() et addElement().

Avant l'API Collections la classe Vector était fréquemment utilisée : il est préférable d'utiliser une des implémentations de l'API Collections.

Les éléments sont stockés dans l'ordre dans lequel ils sont ajoutés dans la collection. Un élément peut être ajouté ou supprimé à n'importe qu'elle position dans la collection.

Toutes les méthodes de la classe Vector sont synchronized : elle est donc moins performante que la classe ArrayList car elle est thread-safe.

Classe ArrayList

- C’est un tableau d’objets de taille dynamique
- Utilise un tableau dont la taille s’adapte automatiquement au nombre d’éléments de la collection
- Coût à payer pour chaque redimentionnement (à chaque ajout d’élément)
- **Non thread safe !** (= plus rapide dans un environnement single thread, mais ne fonctionnera pas dans un environnement multi threadé !)

Méthodes de la classe ArrayList

Constructeur	Rôle
ArrayList()	Créer une instance vide de la collection avec une capacité initiale de 10
ArrayList(Collection<? extends E> c)	Créer une instance contenant les éléments de la collection fournie en paramètre dans l'ordre obtenu en utilisant son iterator
ArrayList(int initialCapacity)	Créer une instance vide de la collection avec la capacité initiale fournie en paramètre

Méthodes de la classe ArrayList

Méthode	**Rôle**
boolean add(Object)	Ajouter un élément à la fin du tableau
boolean addAll(Collection)	Ajouter tous les éléments de la collection fournie en paramètre à la fin du tableau
boolean addAll(int, Collection)	Ajouter tous les éléments de la collection fournie en paramètre dans la collection à partir de la position précisée
void clear()	Supprimer tous les éléments du tableau
void ensureCapacity(int)	Augmenter la capacité du tableau pour s'assurer qu'il puisse contenir le nombre d'éléments passé en paramètre
Object get(index)	Renvoyer l'élément du tableau dont la position est précisée
int indexOf(Object)	Renvoyer la position de la première occurrence de l'élément fourni en paramètre
boolean isEmpty()	Indiquer si le tableau est vide
int lastIndexOf(Object)	Renvoyer la position de la dernière occurrence de l'élément fourni en paramètre
Object remove(int)	Supprimer dans le tableau l'élément fourni en paramètre
void removeRange(int, int)	Supprimer tous les éléments du tableau de la première position fournie incluse jusqu'à la dernière position fournie exclue
Object set(int, Object)	Remplacer l'élément à la position indiquée par celui fourni en paramètre
int size()	Renvoyer le nombre d'éléments du tableau
void trimToSize()	Ajuster la capacité du tableau sur sa taille actuelle

Classe LinkedList (liste chaînée)

Méthode	**Rôle**
void addFirst(Object)	Insèrer l'objet au début de la liste
void addLast(Object)	Insèrer l'objet à la fin de la liste
Object getFirst()	Renvoyer le premier élément de la liste
Object getLast()	Renvoyer le dernier élément de la liste
Object removeFirst()	Supprimer le premier élément de la liste et renvoie l'élément qui est devenu le premier
Object removeLast()	Supprimer le dernier élément de la liste et renvoie l'élément qui est devenu le dernier

Différence ArrayList / LinkedList

- une ArrayList stocke ses éléments en interne dans un tableau à taille fixe alors qu'une LinkedList stocke ses éléments dans une liste doublement chaînée
- une ArrayList permet un accès direct à un élément alors qu'une LinkedList doit parcourir ses éléments pour obtenir celui désiré (mauvaises perfs)
- le coût de variation de la capacité d'une collection de type ArrayList est important car il implique une copie du tableau de stockage interne de ses éléments
- l'ajout d'un élément en début ou en fin d'une collection de type LinkedList est particulièrement performant et son temps d'exécution est constant dans le temps (LinkedList implémente aussi l'interface Deque)

Performances

	get	add	contains	next	remove(0)	iterator.remove
ArrayList	O(1)	O(1)	O(n)	O(1)	O(n)	O(n)
LinkedList	O(n)	O(1)	O(n)	O(1)	O(1)	O(1)

Les Set

- L'interface Set définit les fonctionnalités d'une collection qui ne peut pas contenir de doublons dans ses éléments.
- Les éléments ajoutés dans une collection de type Set doivent réimplémenter leurs méthodes equals() et hashCode(). Ces méthodes sont utilisées lors de l'ajout d'un élément pour déterminer s'il est déjà présent dans la collection. La valeur retournée par hashCode() est recherchée dans la collection :
  - si aucun objet de la collection n'a la même valeur de hachage alors l'objet n'est pas encore dans la collection et peut être ajouté
  - si un ou plusieurs objets de la collection ont la même valeur de hachage alors la méthode equals() de l'objet à ajouter est invoquée sur chacun des objets pour déterminer si l'objet est déjà présent ou non dans la collection

Méthode	Rôle
boolean add(E e)	Ajouter l'élément fourni en paramètre à la collection si celle-ci ne le contient pas déjà et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
boolean addAll(Collection<? extends E> c)	Ajouter tous les éléments de la collection fournie en paramètre à la collection si celle-ci ne les contient pas déjà et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
void clear()	Retirer tous les éléments de la collection (l'implémentation de cette opération est optionnelle)
boolean contains(Object o)	Renvoyer un booléen qui précise si la collection contient l'élément fourni en paramètre
boolean containsAll(Collection<?> c)	Renvoyer un booléen qui précise si tous les éléments de la collection fournie en paramètre sont contenus dans la collection
boolean equals(Object o)	Comparer l'égalité de la collection avec l'objet fourni en paramètre. L'égalité est vérifiée si l'objet est de type Set, que les deux collections ont le même nombre d'éléments et que chaque élément d'une collection est contenu dans l'autre
int hashCode()	Retourner la valeur de hachage de la collection
boolean isEmpty()	Renvoyer un booléen qui précise si la collection est vide
Iterator<E> iterator()	Renvoyer un Iterator sur les éléments de la collection
boolean remove(Object o)	Retirer l'élément fourni en paramètre de la collection si celle-ci le contient et renvoyer un booléen qui précise si la collection a été modifiée (l'implémentation de cette opération est optionnelle)
boolean removeAll(Collection<?> c)	Retirer les éléments fournis en paramètres de la collection si celle-ci les contient et renvoyer un booléen qui précise si la collection a été modifiée. (l'implémentation de cette opération est optionnelle)
boolean retainAll(Collection<?> c)	Retirer tous les éléments de la collection qui ne sont pas dans la collection fournie en paramètre (l'implémentation de cette opération est optionnelle)
int size()	Renvoyer le nombre d'éléments de la collection. Si ce nombre dépasse Integer.MAX_VALUE alors la valeur retournée est MAX_VALUE
Object[] toArray()	Renvoyer un tableau des éléments de la collection
<T> T[] toArray(T[] a)	Renvoyer un tableau des éléments de la collection dont le type est celui fourni en paramètre

Implémentation : la classe HashSet

- La classe HashSet présente plusieurs caractéristiques :
  - elle ne propose aucune garantie sur l'ordre de parcours lors de l'itération sur les éléments qu'elle contient
  - elle ne permet pas d'ajouter des doublons mais elle permet l'ajout d'un élément null
- La classe HashSet utilise en interne une HashMap dont la clé est l'élément et dont la valeur est une instance d'Object identique pour tous les éléments.

Comparaison des implémentations de Set

Classe	add()	contains()	next()	thread-safe
HashSet	O(1)	O(1)	O(h/n)	Non
LinkedHashSet	O(1)	O(1)	O(1)	Non
CopyOnWriteArraySet	O(n)	O(n)	O(1)	Non
EnumSet	O(1)	O(1)	O(1)	Oui
TreeSet	O(log n)	O(log n)	O(log n)	Oui
ConcurrentSkipListSet	O(log n)	O(log n)	O(1)	Oui

Interface Map : pour gérer les collections de couples (attribut/valeur)

Méthode	**Rôle**
void clear()	Supprimer tous les éléments de la collection
boolean containsKey(Object)	Indiquer si la clé est contenue dans la collection
boolean containsValue(Object)	Indiquer si la valeur est contenue dans la collection
Set entrySet()	Renvoyer un ensemble contenant les paires clé/valeur de la collection
Object get(Object)	Renvoyer la valeur associée à la clé fournie en paramètre
boolean isEmpty()	Indiquer si la collection est vide
Set keySet()	Renvoyer un ensemble contenant les clés de la collection
Object put(Object, Object)	Insèrer la clé et sa valeur associée fournies en paramètres
void putAll(Map)	Insèrer toutes les clés/valeurs de l'objet fourni en paramètre
Collection values()	Renvoyer une collection qui contient toutes les valeurs des éléments
Object remove(Object)	Supprimer l'élément dont la clé est fournie en paramètre
int size()	Renvoyer le nombre d'éléments de la collection

Note : Une collection de type Map ne propose pas directement d'Iterator sur ses éléments : la collection peut être parcourue de trois manières :

- parcours de l'ensemble des clés
- parcours des valeurs
- parcours d'un ensemble de paires clé/valeur

Classe HashMap

- La classe HashMap utilise un tableau de listes chaînées pour le stockage de ses éléments. L'index d'une clé dans le tableau est déterminé grâce à un algorithme utilisant la valeur de hachage de l'objet.
- Si deux objets possèdent la même valeur de hachage, il y a une collision car les deux objets doivent être insérés dans le même bucket. Pour gérer les problèmes, le bucket contient une liste chaînée : chaque élément (sa clé et sa valeur) est encapsulé dans une instance de type Entry. (voir cours SGBD)

HashMap

- Elle permet l'utilisation de la valeur null comme clé et comme valeur.
- Elle n'est pas thread-safe.
- Elle ne garantit aucun ordre lors du parcours des éléments de la collection.
- La classe HashMap possède deux propriétés :
  - la capacité initiale (toutes les capacités sont des puissances de 2)
  - le facteur de charge (load factor)
- Performance : get() et put() sont O(1) si la fonction de hachage est « équilibrée »

Comparaison des implémentation de Map

	get()	**containsKey()**	**next()**
HashMap	O(1)	O(1)	O(h/n)
LinkedHashMap	O(1)	O(1)	O(1)
IdentityHashMap	O(1)	O(1)	O(h/n)
EnumMap	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)
ConcurrentHashMap	O(1)	O(1)	O(h/n)
ConcurrentSkipListMap	O(log n)	O(log n)	O(1)

Conclusion

A retenir

- L’héritage
- Les collections
- Comment coder en Java en eclipse !