Le langage Java
- Collections -

Motivations

- Les programmeurs :
  - Font des erreurs
  - Perdent beaucoup de temps à les débugger
  - Ne les trouvent pas toutes -> bugs pouvant être très graves

- Structures de données -> zones à fort risque d'erreur !

- -> fournir des structures de données « prêtes à l'emploi »
  - Fiables
  - Performantes
  - Faciles à utiliser

=> Les collections

Collections = programmation moderne

- Fonction de base des langages actuels
  - Java (depuis JDK 1.2 = 1998)
  - Scala
  - C++
  - C#
  - Python
  - …
- Collections bien utilisées :
  - On n’écrit plus de structures de données (de base)
  - On n’écrit plus de boucles ! (Java 8)

Objectif de ce cours

- Connaître les collections disponibles
- Savoir les utiliser correctement

- Plus tard (pas en PI2!) vous pourrez apprendre:
  - A modifier le comportement de collections existantes
  - A faire vos propres collections

Les collections principales de Java

Interfaces: pas utilisables directement (système proche des classes abstraites)

List xxx = new ArrayList() -> oui
~~List yyy = new List()~~ -> non

- *List* : listes / tableaux dynamiques
  - ArrayList, LinkedList
- Stack : piles d'objets
- *Queue, Deque* : files d'objets
  - ArrayDeque, PriorityQueue, BlockingQueue, BlockingDeque
- *Set* : ensembles d'objets
  - HashSet, LinkedHashSet, TreeSet
- *Map* : tables de correspondances (clé, valeur) d'objets
  - HashMap, LinkedHashMap, TreeMap

Problème : collections de quoi ?

- On veut que nos collections soient utilisables avec n’importe quel type d’objet

- Comment faire ?

Généricité

- Mécanisme qui permet de donner un type comme paramètre
- Permet de faire des collections de « T », où T est un type
- Syntaxe d'utilisation:
  - `NomClasseGenerique < T >`

- Exemple:

```
List<Integer> maListeEntiers = new ArrayList<Integer>();
```

Exemples simples d’utilisation

…faits au tableau…

List<E>

- Représente des séquences d'objets
- Implémentations :
  - ArrayList<E> : tableau dynamique
  - LinkedList<E> : liste (doublement) chaînée
- Méthodes principales :
  - `boolean add(E e)` : ajout d'un élément
  - `int size()` : nombre d'élements de la liste
  - `E get(int index)` : renvoie l'élément à la position index
  - `boolean isEmpty()` : test de liste vide
  - `E remove(int index)` : retire l'élément à index de la liste et le renvoie

Exemples

…au tableau :

création d'une liste

parcours des valeurs

remplacement de l'implémentation

Stack<E>

- Pile d’objets

- Méthodes principales :
  - `boolean empty()` : test de pile vide
  - `E peek()` : renvoie la valeur du haut de pile, sans la dépile
  - `E pop()` : renvoie la valeur du haut de pile, en la dépilant
  - `E push(E e)` : ajoute une valeur en haut de la pile

Set<E>

- Collection d'éléments sans duplicats
- L'ordre dans la collection n'est pas nécessairement celui des ajouts
- Implémentations :
  - HashSet<E> : implémentation par table de hachage
  - LinkedHashSet<E> : table de hachage + liste -> ordre de parcours = ordre des insertions
  - TreeSet<E> : implémentation par arbre
- Méthodes principales :
  - `boolean add(E e) :` ajout d'un élément (s'il n'est pas déjà présent)
  - `boolean contains(Object o)` : test de la présence d'un élément
  - `boolean remove(Object o)` : suppression d'un élément

Map<K, V>

- Collection qui associe des clés à des valeurs
- Au plus une valeur par clé
- Implémentations
  - HashMap<K, V>: table de hachage
  - LinkedHashMap<K, V>: table de hachage + liste pour l'ordre
  - TreeMap<K, V>: arbre
- Méthodes principales :
  - `V put(K key, V value)` : associe key à value. Si l'association existait déjà, l'ancienne valeur est écrasée dans la Map, et est retournée par la méthode
  - `V get(Object key)` : renvoie la valeur associée à la clé key, ou null
  - `Set<K> keySet()` : renvoie l'ensemble des clés de la map

Exemples

…au tableau

- *exemples de Set*
- *exemples de Map*
- *dans les deux cas:*
  - *utilisation basique*
  - *changements d'ordre en changeant l'implémentation*