Généricité

Théorie et pratique de la programmation
Michel Schinz – 2013-02-25

Listes d'éléments arbitraires

Listes arbitraires

L'interface `StringList` et les classes qui la mettent en œuvre ne peuvent représenter que des listes de chaînes de caractères.

Comment faire pour représenter des listes d'un type arbitraire ? Idées :

- écrire autant d'interfaces et de classes qu'il existe de types (spécialisation), ou
- faire des listes de `Object`, ou
- utiliser la généricité.

Spécialisation

La spécialisation consiste à écrire une interface par type d'éléments que la liste peut contenir. Cela devient vite irréaliste...

```
interface StringList {
  … void add(String newElem); … }
interface BooleanList {
  … void add(boolean newElem); … }
interface IntList {
  … void add(int newElem); … }
interface IntListList {
  … void add(IntList newElem); … }
interface IntListListList {
  … void add(IntListList newElem); … }
```

Listes de type `Object`

Une solution plus réaliste que la spécialisation est l'utilisation du type `Object`, qui en Java est un super-type de tous les types, sauf les types de base :

```
interface List {
  boolean isEmpty();
  int size();
  void add(Object newElem);
  void remove(int index);
  Object get(int index);
  void set(int index, Object elem);
}
```

(Solution utilisée avant l'introduction de la généricité en Java).

Listes de type `Object`

Malheureusement, l'utilisation de telles listes implique une grande quantité de transtypages (*casts*) :

```
List l = new …;
l.add("hello");
char c = ((String)l.get(0)).charAt(0);
```

et comporte des risques…

```
List l = new …;
l.add(new Object());
char c = ((String)l.get(0)).charAt(0);
```

Généricité

En raison des problèmes posés par la spécialisation et la solution basée sur le type `Object`, la notion de **généricité** (*genericity*), aussi appelée **polymorphisme paramétrique** (*parametric polymorphism*) a été introduite dans la version 5 de Java.

Au moyen de la généricité, il est possible de définir des listes génériques, c'est-à-dire capables de contenir des éléments d'un type arbitraire.

Généricité

Listes génériques

Une interface pour les listes de valeurs d'un type arbitraire peut se définir ainsi :

```
interface List<E> {
  boolean isEmpty();
  int size();
  void add(E newElem);
  void remove(int index);
  E get(int index);
  void set(int index, E elem);
}
```

Cette interface est **générique**, et E est un **paramètre de type** de cette interface. Il s'agit d'une variable (de type !) représentant le type des éléments de la liste.

Instanciation

Pour utiliser un type générique comme `List`, il faut spécifier le type concret à utiliser pour le paramètre de type.
Exemples :

- `List<Object>`
- `List<String>`
- `List<List<String>>`

Tous ces types sont des **instanciations** du type générique `List`.

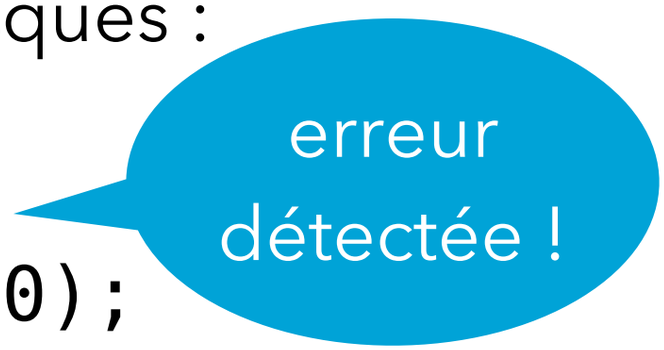
Utilisation

Contrairement aux listes basées sur `Object`, les listes génériques n'impliquent aucun transtypage :

```
List<String> l = new …;
l.add("hello");
char c = l.get(0).charAt(0);
```

et ne comportent pas les mêmes risques :

```
List<String> l = new …;
l.add(new Object());
char c = l.get(0).charAt(0);
```



Listes chaînées génériques

La classe pour les listes chaînées de valeurs arbitraires doit bien entendu également être générique :

```
class LinkedList<E> implements List<E> {
  private LLNode<E> head = null, tail = null;
  private int size = 0;

  public boolean isEmpty() {
    return size == 0;
  }
  public int size() {
    return size;
  }
  …
}
```

Nœuds génériques

Il en va de même pour la classe des nœuds :

```
class LLNode<E> {
  E value;
  LLNode<E> next;

  public LLNode(E value) {
    this.value = value;
    this.next = null;
  }
}
```

Itérateurs génériques

Exercice : en vous inspirant de l'interface des itérateurs de chaînes de caractères, rappelée ci-dessous, définissez une interface pour un itérateur générique.

```
interface StringIterator {
  boolean hasNext();
  String next();
}
```

Paramètres multiples

Un type générique peut bien entendu avoir plusieurs paramètres de type. Par exemple, une classe modélisant les paires de valeurs quelconque pourrait être définie ainsi :

```
public class Pair<F, S> {
  private final F fst;
  private final S snd;

  public Pair(F fst, S snd) { … }

  public F fst() { return fst; }
  public S snd() { return snd; }
}
```

Méthodes génériques

Méthodes génériques

Il n'y a pas que les classes et les interfaces qui puissent être génériques : les méthodes le peuvent aussi. Exemple :

```
class Lists {
  static <E> List<E> newLinkedList1(E e) {
    List<E> l = new LinkedList<E>();
    l.add(e);
    return l;
  }
}
```

Ici, `E` est le paramètre de type de la méthode générique `newLinkedList1`. Il est déclaré *avant* le type de retour de la méthode, car ce dernier peut y faire référence – comme c'est le cas ici.

Inférence des types

Lorsqu'on appelle une méthode générique, il n'est généralement pas nécessaire de spécifier explicitement les arguments de type car ils sont **inférés** (c-à-d devinés).
Par exemple, dans l'appel ci-dessous, l'information que `E` vaut `String` est déterminée automatiquement en fonction de la valeur passée :

```
Lists.newLinkedList1("hello")
```

mais il est également possible – et parfois nécessaire – de le spécifier explicitement, au moyen de la syntaxe suivante :

```
Lists.<String>newLinkedList1("hello")
```

Classes et méthodes gén.

Il est bien entendu possible de placer des méthodes génériques à l'intérieur de classes qui le sont également :

```
class LinkedList<E> implements List<E> {
  …
  <F> List<Pair<E,F>> zip(List<F> that) {
    Iterator<E> i1 = this.iterator();
    Iterator<F> i2 = that.iterator();
    List<Pair<E,F>> r =
      new LinkedList<Pair<E,F>>();
    while (i1.hasNext() && i2.hasNext())
      r.add(new Pair<E,F>(i1.next(),
                          i2.next()));
    return r;
  }
}
```

Inférence des types (bis)

Depuis la version 7 de Java, l'inférence des types peut aussi être demandée lors de la construction d'une valeur d'une classe générique. On utilise pour cela les crochets vides **<>**, aussi appelés « le diamant » (*diamond*).
La méthode `zip` peut p.ex. se simplifier ainsi :

```
<F> List<Pair<E,F>> zip(List<F> that) {
  Iterator<E> i1 = this.iterator();
  Iterator<F> i2 = that.iterator();
  List<Pair<E,F>> r = new LinkedList<>();
  while (i1.hasNext() && i2.hasNext())
    r.add(new Pair<>(i1.next(),i2.next()));
  return r;
}
```

ici…

…et là

Généricité et types de base

Types de base

Rappel : Java possède 8 types dits *de base* qui ne sont pas des objets (`boolean`, `byte`, `short`, `int`, `long`, `char`, `float`, `double`).

Malheureusement, les types de base ne peuvent pas être utilisés comme paramètres de type d'un type générique.

Dès lors, le code suivant est erroné :

```
List<int> l = …; // interdit !
l.add(2);
l.add(3);
```

Que faire si l'on désire créer une liste d'entiers ?

Emballage

Solution : stocker chaque valeur de type `int` dans un objet de type `java.lang.Integer`, et créer une liste de ce type.
L'exemple devient :

```
List<Integer> l = …;
l.add(new Integer(2));
l.add(new Integer(3));
```

On dit alors que les entiers ont été **emballés** (*wrapped* ou *boxed*) dans des objets de type `Integer`.
Le paquetage `java.lang` contient une classe d'emballage par type de base (`Boolean` pour `boolean`, `Char` pour `char`, etc.)

Déballage

Bien entendu, lorsqu'on ressort les valeurs emballées de la liste, il faut les déballer (*unwrap* ou *unbox*) avant de pouvoir les utiliser.

Dans le cas des entiers, cela se fait au moyen de la méthode `intValue` de la classe `Integer` :

```
List<Integer> l = …;
…
int sum = l.get(0).intValue() +
  l.get(1).intValue();
```

Emballage automatique

L'emballage et le déballage manuels étant lourds à l'usage, le code nécessaire peut être produit automatiquement. On appelle cela l'emballage – et le déballage – automatique (*autoboxing*).

L'exemple précédent peut donc également s'écrire ainsi :

```
List<Integer> l = …;
l.add(2);
l.add(3);
int sum = l.get(0) + l.get(1);
```

et est automatiquement transformé afin que les entiers soient emballés avant d'être passés à `add` puis déballés avant l'addition.

Limitations de la généricité en Java

Limitations de la généricité

Pour des raisons historiques, la généricité en Java possède les limitations suivantes :

- la création de tableaux dont les éléments ont un type générique est interdite,
- les tests d'instance impliquant des types génériques sont interdits,
- les transtypages (*casts*) sur des types génériques ne sont pas sûrs, c-à-d qu'ils produisent un avertissement lors de la compilation et un résultat éventuellement incorrect à l'exécution,
- la définition d'exceptions génériques est interdite.

Généricité et tableaux

Limitation n°1 : la création de tableaux dont les éléments ont un type générique est interdite.
Par exemple, le code suivant est refusé :

```
static <T> T[] newArray(T x) {
   return new T[]{ x };
}
```

Test d'instance générique

Limitation n°2 : les tests d'instance impliquant des types génériques sont interdits.

Par exemple, le code suivant est refusé :

```
<T> void clearIntList(List<T> l) {
  if (l instanceof List<Integer>)
    while (l.size() > 0)
      l.remove(0);
}
```

Transtypage générique

Limitation n°3 : les transtypages impliquant des types génériques ne sont pas sûrs.
Par exemple, le code suivant ne lève pas d'exception à l'exécution, alors qu'il devrait en lever une :

```
List<Integer> l = new …;
Object o = l;
List<String> l2 = (List<String>)o;
```

aucune exception levée

Un avertissement est toutefois produit.

Exceptions génériques

Limitation n°4 : une classe définissant une exception ne peut pas être générique. En d'autres termes, aucune sous-classes de `Throwable` ne peut avoir de paramètres de type.

Par exemple, la définition suivante est refusée :

```
class BadException<T> extends Exception {}
```

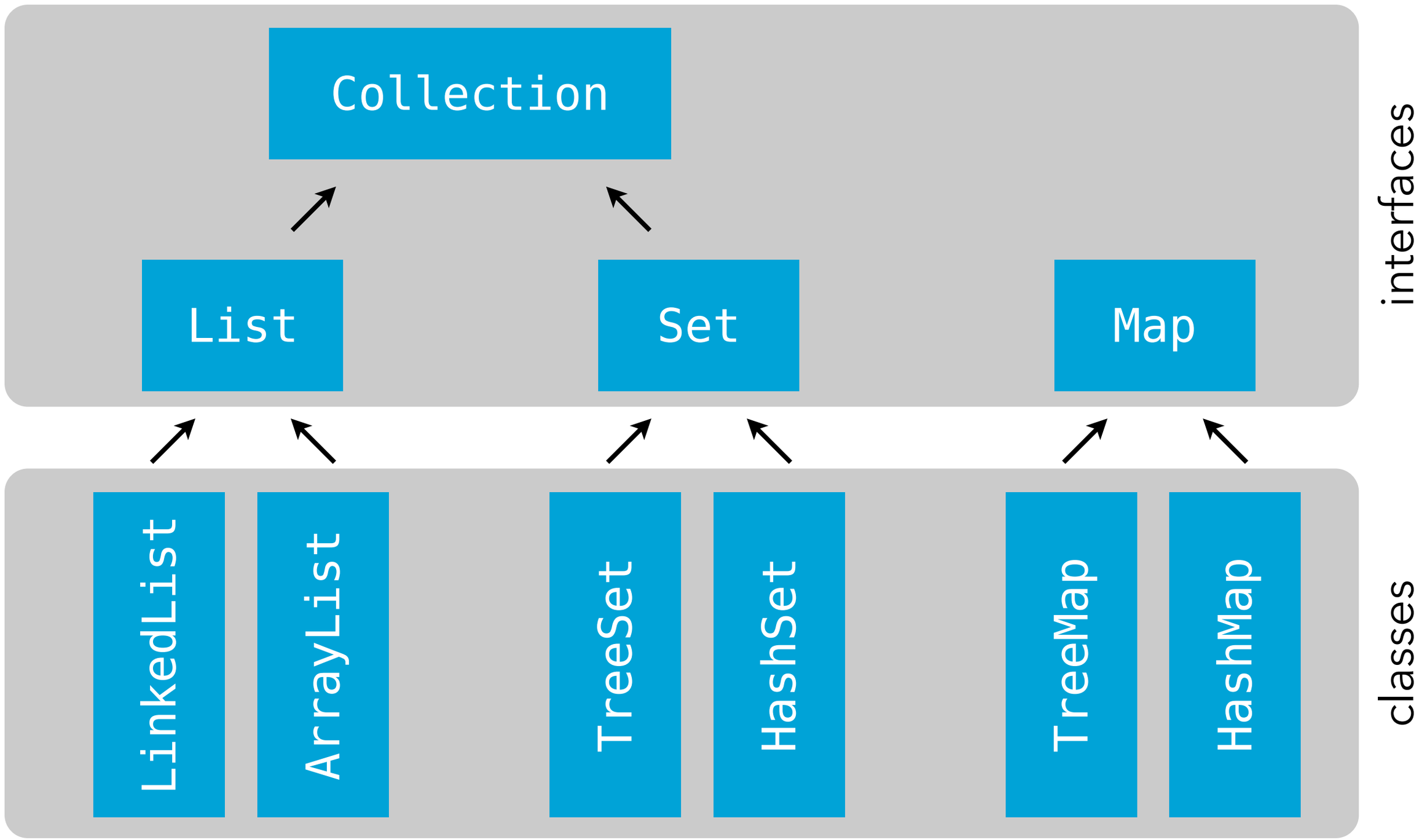
Collections génériques de Java

Collections de l'API Java

L'API Java fournit un certain nombre de collections dans le *Java collections framework*. Tout son contenu se trouve dans le (très mal nommé) paquetage `java.util`.
Pour chaque collection existe :

- une interface,
- une ou plusieurs mises en œuvre, sous la forme de classes.

Collections de l'API Java



(vue partielle)

Exemple d'utilisation

```
import java.util.List;
import java.util.ArrayList;
List<String> workingDays = new ArrayList<>();
workingDays.add("lundi");
workingDays.add("mardi");
workingDays.add("mercredi");
workingDays.add("jeudi");
workingDays.add("vendredi");
System.out.println("nb. de jours ouvrés : " +
                   workingDays.size());
System.out.println("2e jour ouvré : " +
                   workingDays.get(1));
```

Boucle *for-each*

Boucle *for-each*

Rappel : une variante de la boucle `for`, couramment nommée boucle *for-each*, permet de parcourir élégamment les éléments d'un tableau.

Exemple :

```
int[] somePrimes = new int[] {
  2, 3, 5, 7, 11, 13
};
for (int prime: somePrimes)
  System.out.println(prime)
```

Ce type de boucle ne se limite pas aux tableaux !

Boucle *for-each*

La boucle *for-each* est utilisable sur n'importe quel objet dont la classe implémente l'interface générique `Iterable` du paquetage `java.lang`, définie ainsi :

```
interface Iterable<E> {
  Iterator<E> iterator();
}
```

Itérateurs

L'interface `Iterator` du paquetage `java.util` est quasi-identique à celle que nous avons définie :

```
interface Iterator<E> {
  boolean hasNext();
  E next();
  void remove();
}
```

Elle contient une méthode additionnelle, `remove`, permettant de supprimer le dernier élément produit par `next`. (Mais il s'agit d'une opération optionnelle qui peut simplement lever `UnsupportedOperationException`).

Liste itérable

Pour rendre nos listes utilisables avec la boucle *for-each*, il suffit de leur faire implémenter l'interface `Iterable` :

```
interface List<E> extends Iterable<E> {
  …
  Iterator<E> iterator();
}
```

puis d'ajouter la méthode `iterator` à toutes les classes mettant en œuvre nos listes. P.ex. pour les listes chaînées :

```
class LinkedList<E> implements List<E> {
  …
  public Iterator<E> iterator() {
    return new LLIterator<>(head);
  }
}
```

Itérateur de liste chaînée

```
class LLIterator<E> implements Iterator<E> {
  private LLNode<E> nextNode;
  public LLIterator(LLNode<E> head) {
    nextNode = head;
  }
  public boolean hasNext() {
    return nextNode != null;
  }
  public E next() {
    E v = nextNode.value;
    nextNode = nextNode.next;
    return v;
  }
  // remove lève UnsupportedOperationException
}
```

Classes imbriquées (digression)

Visibilité des classes

Problème : les classes `LLNode` et `LLIterator` ne sont d'aucune utilité hors de la classe `LinkedList`. Le fait qu'elles soient néanmoins visibles de l'extérieur viole le principe d'encapsulation.

Ne peut-on pas les cacher un peu plus ?

Première idée : on ne les marque pas `public`, et elles ne sont dès lors pas visibles hors du paquetage.

Mais ne peut-on pas faire encore mieux ?

Classe imbriquée

Java permet de déclarer une classe à l'intérieur d'une autre.
On parle alors de **classe imbriquée** (*nested class*).
Une classe imbriquée peut être privée et donc invisible de l'extérieur.
De plus, elle peut être statique ou non, ce qui influence les valeurs auxquelles elles a accès.

Classe imbriquée statique

Une classe imbriquée statique est assez similaire à une classe normale, si ce n'est que :

- elle peut être marquée `private` ou `protected`, ce qui détermine la visibilité de son type selon les règles habituelles,
- lorsqu'elle est visible depuis l'extérieur, son nom est préfixé par le nom de la classe englobante (p.ex. `LinkedList.Node`), ce qui réduit la « pollution » de l'espace de noms,
- elle peut accéder à tous les champs statiques (y compris privés) de sa classe englobante.

Nœud, itérateur imbriqués

```
class LinkedList<E> implements List<E> {
  private Node<E> head = null, tail = null;
  …
  private static class Node<E> {
    private E value;
    private Node<E> next;
    …
  }
  private static class LLIterator<E>
      implements Iterator<E> {
    private Node<E> nextNode;
    …
  }
}
```

plus visible de l'extérieur !

idem

Classe imbriquée non statique

Une classe imbriquée non statique se comporte comme une classe imbriquée statique si ce n'est que :

- elle a accès à tous les champs (même privés) de sa classe englobante,
- elle a accès aux paramètres de type de sa classe englobante.

(Cela a un coût : une telle classe possède un champ qui référence la classe englobante, accessible avec la syntaxe o`.this` où o est le nom de la classe englobante).

Itérateurs non statiques

```
class LinkedList<E> implements List<E> {
  private Node<E> head = null, tail = null;

  …


  private class LLIterator
      implements Iterator<E> {
    private Node<E> nextNode = head;
  }
}
```

plus de paramètre de type !

accès direct au champ privé head !

Classes anonymes

Dans certains cas, on désire uniquement créer une instance d'une classe, sans avoir besoin d'un nom pour son type. Java offre pour cela la possibilité de créer une instance d'une classe anonyme, avec la syntaxe suivante :

```
new n(p1, p2, ...)  {
    … // corps de la classe
}
```

où n désigne une classe (dont on hérite) ou une interface (qu'on implémente, en héritant alors de `Object`) et p_1, … sont les éventuels paramètres du constructeur de la super-classe.

Itérateurs anonymes

```
class LinkedList<E> implements List<E> {
  private Node<E> head = null, tail = null;
  …
  public Iterator<E> iterator() {
    return new Iterator<E>() {
      private Node<E> nextNode = head;

      public boolean hasNext() { … }
      public E next() { … }
      public void remove() { … }
    };
  }
}
```

corps de la classe anonyme (qui implémente `Iterator<E>`)

Accès aux variables locales

Une classe (anonyme) imbriquée dans une méthode ne peut accéder aux variables locales de cette méthode que si ceux-ci sont marqués `final`, et donc non modifiables.

Exemple :

```
class LinkedList<E> implements List<E> {
  private Node<E> head = null, tail = null;
  …
public Iterator<E> iterator() {
  final Node<E> theHead = head;
  return new Iterator<E>() {
    private Node<E> nextNode = theHead;
    …
  };
} … }
```

si `theHead` n'était pas `final`, cet accès serait interdit !

Résumé et références

La généricité permet de paramétrer du code en fonction de types. Cela est très utile dans de nombreux contextes, en particulier les collections, où le type des données stockées sont un paramètre.

Pour aller plus loin :

Java Generics and Collections, Maurice Naftalin et Philip Wadler, O'Reilly Media, 2006

* * *

La possibilité d'imbriquer des classes en Java améliore l'encapsulation.

Les classes anonymes sont utiles pour créer des objets dont l'utilité est très locale.