Eléments de génie logiciel

Pratique de la programmation orientée-objet

Michel Schinz – 2014-02-17

Génie logiciel

Le développement de programmes conséquents comme le projet requiert des techniques d'organisation et des méthodes de travail spécifiques afin de pouvoir gérer la complexité.

Le **génie logiciel** est la discipline qui étudie ces méthodes de travail. Cette leçon se contente d'examiner trois techniques d'organisation et de travail et leur mise en œuvre en Java :

1. les paquetages,
2. les assertions,
3. le test unitaire.

Paquetages

Nom unique des entités

Java utilise des **noms** pour identifier les entités du programme : classes, interfaces, etc.
Ces noms doivent être uniques, faute de quoi il y a ambiguïté. En effet, si plusieurs entités ont le même nom, comment savoir laquelle utiliser à l'apparition de ce nom ?
(A noter que la surcharge permet de donner le même nom à plusieurs méthodes, leur type étant utilisé pour les différencier. Mais la surcharge est limitée aux méthodes.)

Nom unique des entités

Lorsqu'on écrit un programme de A à Z, garantir l'unicité des noms est possible.
Mais que faire lorsqu'on écrit des bibliothèques ayant pour but d'être utilisées par des milliers de programmeurs, dans autant de programmes différents ?
Par exemple, la bibliothèque standard de Java 7 comporte plus de 3000 classes et interfaces publiques, donc autant de noms. Comment garantir l'absence de conflit avec les noms choisis par tous les programmeurs utilisant cette bibliothèque ?

Exemple

Exemple plausible de conflit de nom : la **table associative** (*map* en anglais) est une des collections fondamentales en informatique. La bibliothèque Java possède ainsi une interface `Map`, utilisée par la quasi-totalité des programmes non triviaux.

Mais le mot *map* désigne aussi une carte en anglais. Un programme cartographique a donc de très grandes chances de comporter une classe ou une interface nommée également `Map`.

Comment distinguer ces deux utilisations du même nom si un seul programme a besoin des deux ?

Mauvaise solution : préfixes

Idée : préfixer tous les noms par une chaîne qu'on espère globalement unique.

Par exemple, l'interface `Map` pour les tables associatives pourrait être nommée `CollMap`, tandis que la classe `Map` des cartes pourrait être nommée `CartoMap`.

De tels préfixes ont plusieurs problèmes :

- ils devraient être longs pour être uniques, mais
- ils devraient être courts pour ne pas gêner, et
- ils doivent être utilisés même en l'absence de conflit puisqu'ils font partie du nom.

Paquetage

Pour tenter de résoudre le problème des noms uniques, Java offre la notion de **paquetage** (*package*).

Un paquetage est une entité nommée qui contient un certain nombre de types (classes et/ou interfaces).

Au début de chaque fichier source, il est possible de spécifier le paquetage dans lequel placer les classes et interfaces qu'il contient au moyen de l'énoncé `package`.

Exemple :

```
package collections;
public interface Map { … }
```

Noms qualifiés

Le **nom complet** ou **nom complètement qualifié** (*fully-qualified name*) d'un type déclaré à l'intérieur d'un paquetage inclut le nom du paquetage.

Ainsi, avec la déclaration suivante :

```
package collections;
public interface Map { … }
```

le nom complètement qualifié de l'interface `Map` est `collections.Map`.

Les paquetages jouent donc un rôle similaire aux préfixes, avec l'avantage de pouvoir être omis dans la majorité des cas, comme nous allons le voir.

Utilisation des noms

Tous les noms d'un paquetage sont utilisables sans préfixe à l'intérieur de ce même paquetage. Cela reste vrai même si un paquetage est réparti sur plusieurs fichiers ! Exemple :

```
package collections;
public class HashMap implements Map { … }
```

A l'extérieur d'un paquetage donné, les noms publics de ce dernier sont utilisables en version totalement qualifiée.

Exemple :

```
package wordprocessor;
class Dictionary
  implements collections.Map { … }
```

Importation

Pour éviter de devoir utiliser la version complètement qualifiée des noms définis dans un autre paquetage, il est possible d'**importer** les noms utilisés, au moyen de l'énoncé `import`.

L'exemple précédent peut se récrire ainsi :

```
package wordprocessor;
import collections.Map;
class Dictionary implements Map { … }
```

≡ collections.Map

Il est interdit d'importer deux noms identiques, ou de définir un nom identique à un nom importé.

Tous les énoncés `import` doivent apparaître juste après l'énoncé `package` (et nulle part ailleurs).

Importation multiple

Il est aussi possible d'importer la totalité des noms d'un paquetage au moyen de l'astérisque. Par exemple,

```
import java.util.*;
```

importe la totalité des noms du paquetage `java.util` (`Map`, `Set`, `Collection`, et beaucoup d'autres).

Cette notation est déconseillée car il devient alors difficile de savoir quel ensemble de noms est importé, et il peut même changer au cours du temps !

L'importation multiple était utile à l'époque où Java a été conçu mais de nos jours les environnements de développement comme Eclipse gèrent automatiquement les importations.

Visibilité des noms

Les paquetages influencent la visibilité des noms :

- Les types des classes ou interfaces qui ne sont pas déclarées `public` sont visibles uniquement dans le paquetage dans lequel ils sont déclarés.
- Les membres qui ne sont déclarés ni `public` ni `private` sont visibles dans le paquetage dans lequel leur propriétaire est déclaré.
- Les membres qui sont déclarés `protected` sont visibles dans le paquetage dans lequel leur propriétaire est déclaré, et dans toutes les sous-classes, indépendamment de leur paquetage.

Hiérarchie

Les paquetages peuvent être organisés en hiérarchie, c-à-d qu'un paquetage peut contenir d'autres paquetages, et ainsi de suite.
Par exemple, il existe un paquetage standard nommé `java`, dans lequel se trouvent plusieurs sous-paquetages comme `java.lang` et `java.util`.
A l'intérieur de ce dernier se trouvent aussi bien des classes et interfaces (p.ex. `List`, `Set`, `Map`) que d'autres sous-paquetages comme `java.util.concurrent`.

Nommage des paquetages

Utiliser des paquetages ne fait que repousser un peu plus loin le problème de l'unicité des noms... Comment éviter que deux programmeurs définissent des paquetages de même nom ?

Idée : utiliser le nom de domaine Internet de l'organisation (unique), inversé, comme préfixe du nom de paquetage.

Exemple : le nom de domaine de l'EPFL est `epfl.ch`. Tous les paquetages développés à l'EPFL peuvent commencer par `ch.epfl`, p.ex. `ch.epfl.collections`.

Problème : les organisations sont renommées, rachetées (Sun par Oracle, p.ex.), disparaissent, etc.

Répertoires et fichiers

Le contenu d'un paquetage peut être réparti sur plusieurs fichiers. Chacun d'entre-eux doit commencer avec un énoncé `package` approprié.

Les fichiers doivent être placés dans des répertoires portant le nom du paquetage.

Par exemple, si l'interface `Map` est déclarée à l'intérieur d'un fichier débutant ainsi :

```
package java.util;
public interface Map { … }
```

alors ce fichier doit être stocké dans un répertoire nommé `util`, lui-même placé dans un répertoire nommé `java`.

Et bien entendu, le fichier doit être nommé `Map.java`.

Importation statique

Le mot-clef `import` peut aussi être suivi du mot-clef `static` pour importer les noms des membres statiques de classes (ou interfaces) sans devoir nommer à chaque fois la classe (ou l'interface). On nomme cela **importation statique**.
Attention, ce type d'importation n'a rien à voir avec les paquetages !
L'astérisque peut également être utilisée pour importer la totalité des membres statiques.

Importation statique

Par exemple, la classe `Math` (du paquetage `java.lang`) possède un champ statique nommé `PI` contenant la constante du même nom, et des méthodes comme `sin`, `cos`, `tan`, etc.

Pour utiliser ces membres statiques, on peut bien entendu les préfixer du nom de la classe : `Math.PI`, `Math.sin(…)`, etc. Mais il est aussi possible de les importer statiquement puis de les utiliser ensuite sans préfixe :

```
import static Math.sin;
class MyClass {
  double sin2(double x) {
    return sin(x) * sin(x);
  }
}
```

Assertions

Propriétés des programmes

Lorsqu'on écrit un programme, on sait souvent que certaines propriétés sont (ou devraient être) vraies, sans que cela ne soit totalement évident à la lecture du code.

Par exemple, on peut savoir qu'à la fin d'un calcul arithmétique complexe, le résultat est dans un intervalle donné.

Connaître ce genre de propriétés à propos d'un morceau de code facilite souvent sa compréhension. De plus, le fait qu'une propriété qui devrait être vraie ne le soit pas signale la présence d'un problème.

Il est donc intéressant de pouvoir les exprimer.

Expression des propriétés

Comment exprimer les propriétés (non triviales) que l'on sait être vraies ? On peut :

- les décrire dans des commentaires — mais les commentaires ne sont pas vérifiables automatiquement, donc ils peuvent être faux et induire en erreur, ou
- faire des tests explicites au moyen d'un `if` et lever une exception si la propriété n'est pas vraie — mais le coût du test peut être important, et la vérification d'une propriété prend au moins deux lignes de code, ou
- utiliser des assertions !

Assertions

Java offre l'énoncé `assert` qui permet d'affirmer qu'une expression booléenne — l'assertion — est toujours vraie. Par exemple, l'énoncé

```
assert 0.5 <= x && x <= 1.0;
```

est équivalent à :

```
if (! (0.5 <= x && x <= 1.0))
  throw new AssertionError();
```

mais possède deux avantages :

1. il est plus concis,
2. il peut être totalement supprimé au moment du lancement du programme, sans devoir re-compiler quoi que ce soit.

Activation des assertions

Par défaut, la vérification des assertions est *désactivée*. C'est-à-dire que les conditions des assertions ne sont pas évaluées et ne coûtent donc rien en temps d'exécution.

Pour activer la vérification des assertions, il faut passer l'option `-enableassertions` (ou `-ea`) à la machine virtuelle Java lors du démarrage du programme.

Recherche dichotomique

Pour illustrer l'utilisation des assertions, écrivons une méthode qui recherche un élément dans un tableau d'entiers trié, en utilisant la technique de la **recherche dichotomique** (*binary search*).
La recherche dichotomique permet de trouver un élément dans un tableau trié en O(log n). C'est mieux que la recherche linéaire, qui est en O(n).
L'idée consiste à comparer l'élément recherché avec celui se trouvant au milieu du tableau, et à éventuellement poursuivre la recherche dans l'un des deux sous-tableaux délimités par cet élément.

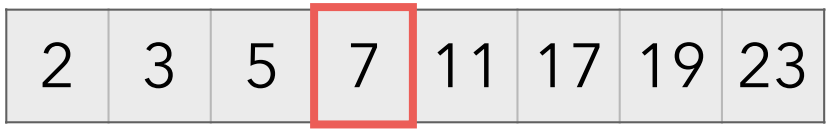
Recherche dichotomique

Exemple : recherche de 19 dans le tableau trié des huit premiers nombres premiers (2, 3, 5, 7, 11, 17, 19, 23).

2	3	5	7	11	17	19	23

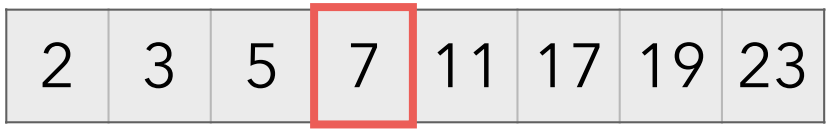
Recherche dichotomique

Exemple : recherche de 19 dans le tableau trié des huit premiers nombres premiers (2, 3, 5, 7, 11, 17, 19, 23).



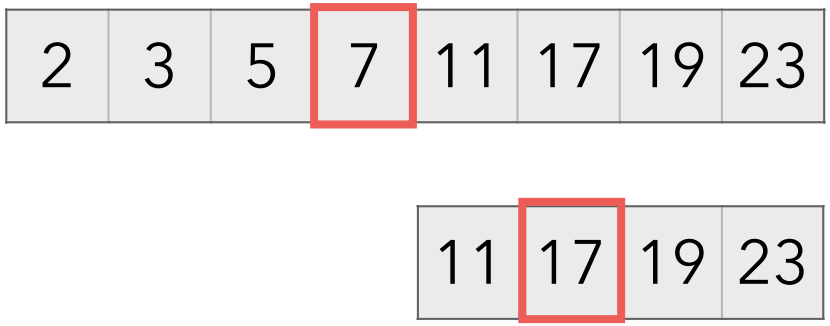
Recherche dichotomique

Exemple : recherche de 19 dans le tableau trié des huit premiers nombres premiers (2, 3, 5, 7, 11, 17, 19, 23).



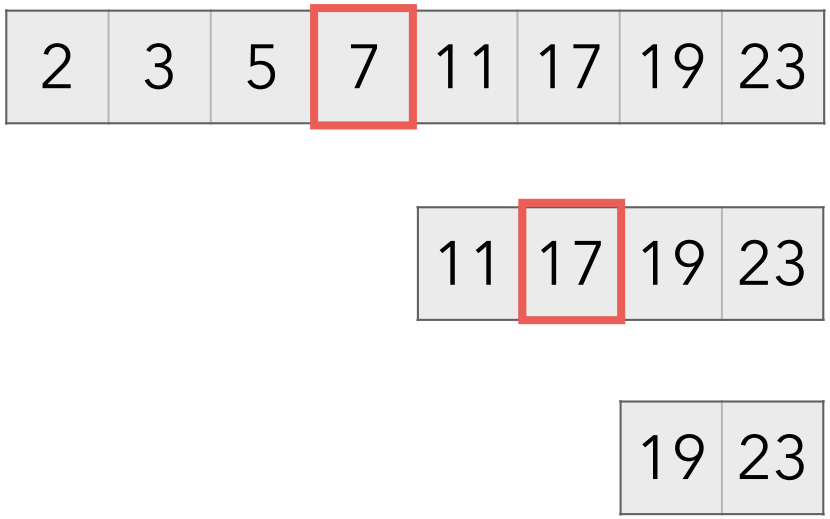
Recherche dichotomique

Exemple : recherche de 19 dans le tableau trié des huit premiers nombres premiers (2, 3, 5, 7, 11, 17, 19, 23).



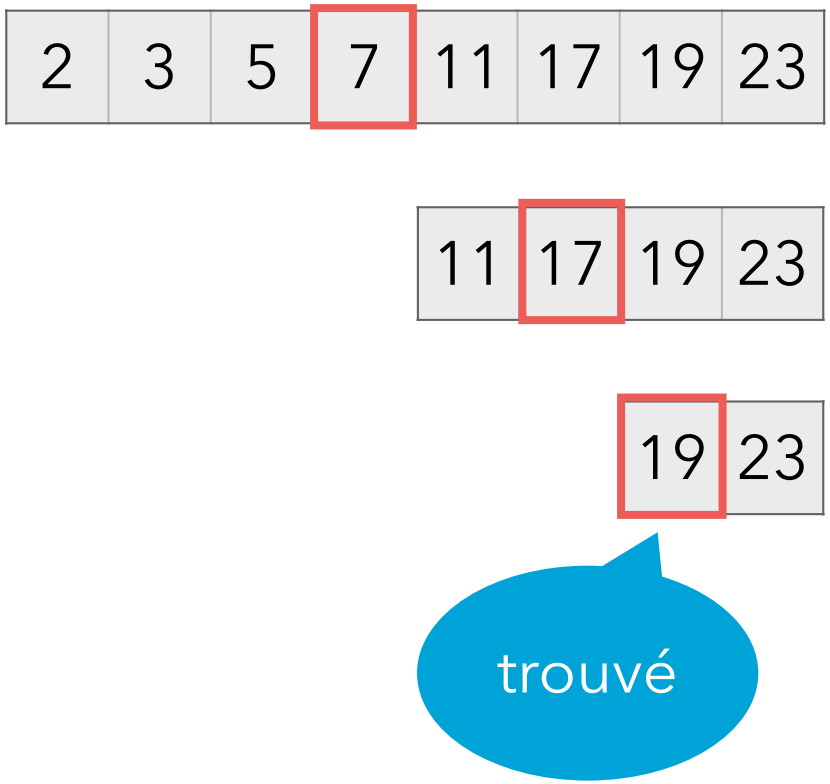
Recherche dichotomique

Exemple : recherche de 19 dans le tableau trié des huit premiers nombres premiers (2, 3, 5, 7, 11, 17, 19, 23).



Recherche dichotomique

Exemple : recherche de 19 dans le tableau trié des huit premiers nombres premiers (2, 3, 5, 7, 11, 17, 19, 23).



Recherche dichotomique

La recherche dichotomique peut se programmer au moyen d'une boucle qui réduit à chaque itération l'intervalle de recherche [l;h].

```
int binarySearch(int[] a, int k) {
  int l = 0, h = a.length - 1;
  while (l <= h) {
    int m = (l + h) / 2;
    if (a[m] < k) l = m;
    else if (a[m] > k) h = m;
    else return m; // trouvé à la position m
  }
  return -1;          // pas trouvé
}
```

Propriétés de la recherche

Une propriété fondamentale de la recherche dichotomique est qu'elle doit **progresser**, c-à-d qu'à chaque itération l'intervalle de recherche doit être (strictement) inclus dans celui de l'itération précédente.

Si ce n'est pas le cas, la recherche ne termine pas…

Cette propriété n'est actuellement pas exprimée de façon explicite dans le code de la méthode `binarySearch`.

Recherche avec assertions

On peut introduire une assertion exprimant le progrès dans la méthode `binarySearch` :

```
int binarySearch(int[] a, int k) {
  int l = 0, h = a.length - 1;
  while (l <= h) {
    int m = (l + h) / 2, l0 = l, h0 = h;
    if (a[m] < k) l = m;
    else if (a[m] > k) h = m;
    else return m;
    assert l > l0 || h < h0;
  }
  return -1;
}
```

intervalle précédent

progrès

Test de la recherche

On peut tester cette méthode `binarySearch` en faisant quelques appels. Initialement, tout se passe bien :

```
int[] a = new int[] {
  2, 3, 5, 7, 11, 17, 19, 23 };
binarySearch(a, 2);  // retourne 0, OK
binarySearch(a, 11); // retourne 4, OK
```

mais les choses se gâtent avec l'appel suivant :

```
binarySearch(a, 23);
```

dont l'exécution provoque la levée d'une exception :

```
Exception in thread "main"
java.lang.AssertionError
  at binarySearch(BinarySearch.java:8)
```

Visiblement, notre méthode `binarySearch` est incorrecte!

Recherche dichotomique

Pour mieux comprendre le problème, il est possible d'ajouter un message (une chaîne de caractères) à l'assertion, qui est attaché à l'exception:

```
int binarySearch(int[] a, int k) {
  int l = 0, h = a.length - 1;
  while (l <= h) {
    int m = (l + h) / 2, l0 = l, h0 = h;
    if (a[m] < k) l = m;
    else if (a[m] > k) h = m;
    else return m;
    assert l > l0 || h < h0: "l="+l+" h="+h;
  }
  return -1;
}
```

Assertion avec message

En reexécutant l'appel problématique :

```
int[] a = new int[] {
  2, 3, 5, 7, 11, 17, 19, 23 };
binarySearch(a, 23);
```

on obtient maintenant un message d'erreur qui nous permet de comprendre le problème :

```
Exception in thread "main"
java.lang.AssertionError: l=6 h=7
  at binarySearch(BinarySearch.java:8)
```

Lorsqu'il n'y a pas plus de deux éléments dans l'intervalle, `m` est égal à `l`. Or si l'élément recherché n'est pas à la position `l`, le « nouvel » intervalle de recherche est identique au précédent. En d'autres termes, il n'y a plus de progrès !

Recherche dichotomique

Ayant compris le problème, on peut maintenant modifier la méthode `binarySearch` pour qu'elle mette correctement en œuvre l'algorithme de recherche dichotomique :

```
int binarySearch(int[] a, int k) {
  int l = 0, h = a.length - 1;
  while (l <= h) {
    int m = (l + h) / 2, l0 = l, h0 = h;
    if (a[m] < k) l = m + 1;
    else if (a[m] > k) h = m - 1;
    else return m;
    assert l > l0 || h < h0;
  }
  return -1;
}
```

Assertions ou exceptions ?

Quand faut-il utiliser une assertion et quand faut-il utiliser un test avec l'exception `IllegalArgumentException` p.ex. ?
La règle générale est que :

- les assertions sont utilisées pour vérifier des conditions internes du programme qui devraient toujours être vraies sauf en présence d'un bug, alors que
- les exceptions sont utilisées pour vérifier les paramètres passés par du code client.

Cela dit, les assertions peuvent être désactivées, pas les exceptions. Donc si une vérification de paramètre est très chère, on peut utiliser une assertion pour la faire. Exemple : vérifier que le tableau passé à `binarySearch` est bien trié.

Assertion ou exception ?

Pour les exemples suivants, pensez-vous qu'il est préférable d'utiliser une assertion ou un test levant une exception ?

1. Pour vérifier que l'entier passé à une fonction factorielle est positif ou nul.
2. Pour vérifier, lors du transfert d'un montant d'un compte en banque à un autre, que la somme des deux soldes est la même avant et après le transfert.
3. Pour vérifier qu'un tableau de réels passé à une fonction de calcul de moyenne n'est pas vide.
4. Pour vérifier que le (très grand) entier passé à une fonction cryptographique est bien premier.

Test unitaire (avec JUnit)

Unités

Tout programme conséquent est composé de plusieurs **unités**, qui doivent idéalement être aussi indépendantes que possible les unes des autres. Une unité est composée d'un petit nombre de classes indissociables.

Par exemple, dans un programme de jeu de Monopoly, on peut imaginer avoir une unité représentant la banque, une autre représentant le plateau de jeu, etc.

Test unitaire

Le **test unitaire** ou **test par unité** (*unit testing*) consiste à écrire des petits programmes s'assurant que chaque unité se comporte comme elle le devrait.
Ces tests doivent être automatiques, dans le sens où il ne doit pas être nécessaire qu'un humain examine le résultat de leur exécution pour savoir qu'un problème est survenu. Il est alors possible de les lancer très fréquemment – p.ex. après chaque modification – et de détecter les problèmes dès leur apparition.

JUnit

JUnit (`http://junit.org/`) est une bibliothèque Java facilitant l'écriture de tests unitaires.

Elle fournit des méthodes pour tester que certaines conditions sont satisfaites, et signaler une erreur dans le cas contraire. De plus, elle fournit une infrastructure permettant de lancer tous les tests contenus dans une classe donnée, et signaler les éventuelles erreurs.

Nous l'utiliserons pour illustrer les idées du test unitaire, mais ces idées ne sont pas spécifiques à JUnit ou à Java.

Utilisation de JUnit

Pour tester une unité à l'aide de JUnit, il convient de définir une (ou plusieurs) classe de test contenant un certain nombre de méthodes de test. Une telle méthode se reconnaît à l'annotation `@Test` qu'on lui attache. Exemple :

```
@Test
public void addition() {
  assertEquals(2, 1 + 1);
}
```

La méthode de test doit utiliser une ou plusieurs méthodes d'assertion fournies par JUnit (`assertEquals`, `assertTrue`, etc.), vérifiant qu'une condition est vraie.

Méthodes d'assertion

La classe `Assert` de JUnit fournit des méthodes statiques d'assertion, qui vérifient qu'une condition est vraie et signalent une erreur dans le cas contraire. Exemples :

- `assertTrue(boolean b)` vérifie que `b` est vrai,
- `assertEquals(Object o1, Object o2)` vérifie que `o1` et `o2` sont égaux en appelant leur méthode `equals`,
- `assertEquals(long l1, long l2)` vérifie que `l1` et `l2` sont égaux,
- `assertNull(Object o)` vérifie que `o` est nul,
- etc.

Test d'exceptions

Il est souvent utile de tester qu'une exception est bien levée, ce que les méthodes d'assertion de JUnit ne permettent pas de faire.

Par contre, il est possible d'ajouter un argument à l'annotation `@Test`, spécifiant qu'une exception donnée est attendue. Exemple :

```
@Test(expected = ArithmeticException.class)
public void divByZero() {
  int x = 1 / 0;
}
```

à ne pas oublier !

Un tel test échoue si l'exception donnée n'est pas levée lors de son exécution.

Test d'une méthode de tri

Pour illustrer l'écriture de tests unitaires au moyen de JUnit, écrivons un test pour une méthode de tri de tableau entier. Cette méthode a le profil suivant :

```
static void sort(int[] array);
```

Sa documentation spécifie qu'on lui passe en argument un tableau d'entiers et qu'elle trie ce tableau par ordre croissant des éléments.

Le but est d'écrire un test unitaire qui vérifie que c'est bien le cas.

Test d'une méthode de tri

La première méthode de test de notre classe `SortTest` vérifie un cas aux limites : le tri d'un tableau vide.

```
import org.junit.Test;
import static org.junit.Assert.*;
public class SortTest {
  @Test
  public void sortsEmptyArray() {
    int[] a1 = new int[0];
    int[] a2 = new int[0];
    sort(a1);
    assertEquals(a1, a2);
  }
  // Une autre méthode de test suivra...
}
```

Test d'une méthode de tri

La seconde méthode de test vérifie un cas plus général, avec un tableau non vide.

```
boolean isSorted(int[] array) {
  for (int i = 1; i < array.length; ++i)
    if (array[i] < array[i - 1])
      return false;
  return true;
}
@Test
public void sortsNontrivialArray() {
  int[] a = new int[]{ 4,3,6,1,5,6,4,-1 };
  sort(a);
  assertTrue(isSorted(a));
}
```

Exercice

Les deux tests précédents vérifient que :

1. la version triée d'un tableau vide est un tableau vide,
2. après l'appel à la méthode de tri, le tableau est bien trié.

Est-ce suffisant ?

Pouvez-vous écrire une méthode sort qui passe les tests mais qui soit fausse ?

Si oui, quel(s) test(s) devriez-vous encore écrire pour éviter ce problème ?

Limites du test

Comme l'a dit Edsger Dijkstra :

> *Program testing can be used to show the presence of bugs, but never to show their absence!*

c'est-à-dire :

> *Le test peut être utilisé pour démontrer la présence de problèmes, mais jamais pour démontrer leur absence !*

Il est important de toujours garder cela à l'esprit, et de ne surtout pas conclure qu'un programme est correct parce qu'il passe tous les tests que l'on a pensé à écrire !

Résumé

Les **paquetages** permettent d'organiser les noms dans les gros programmes, surtout lorsqu'ils sont écrits par différentes personnes (bibliothèques).

Les **assertions** permettent d'exprimer des propriétés internes des programmes et de les vérifier ou non à l'exécution.

Le **test unitaire** permet de tester les différentes parties d'un programme séparément avant de les assembler, afin de détecter et localiser plus rapidement les problèmes.