Enumérations et classes imbriquées

Pratique de la programmation orientée-objet

Michel Schinz – 2014-02-24

Enumérations

Cartes à jouer

On désire modéliser les cartes à jouer d'un jeu de 52 cartes. Chacune d'entre-elles est caractérisée par sa couleur (cœurs, carreaux, trèfles ou piques) et sa valeur (2, 3, …, 10, valet, dame, roi, as).

```
public final class Card {
  private final ??? suit; // couleur
  private final ??? rank; // valeur
  public Card(??? suit, ??? rank) {
    this.suit = suit;
    this.rank = rank;
  }
}
```

Question: quel(s) type(s) utiliser pour la couleur (*suit*) et la valeur (*rank*) ?

Solution 1 : chaînes/entiers

Une première solution est d'utiliser le type `String` pour la couleur et le type `int` pour la valeur, et d'introduire des constantes pour les valeurs valides :

```
public final class Card {
  public final static String SPADES =
    "SPADES";
  // … idem pour HEARTS, CLUBS et DIAMONDS
  public final static int JACK = 11;
  // … idem pour QUEEN, KING et ACE
  private final String suit;
  private final int rank;
  public Card(String suit, int rank) { … }
}
```

Question : est-ce satisfaisant ? Si non, pourquoi ?

Solution 1 : chaînes/entiers

La solution basée sur les types `String` et `int` n'est pas satisfaisante, pour plusieurs raisons :

1. Ces types sont trop généraux et incluent énormément de valeurs invalides, p.ex. la chaîne `"bonjour"` pour la couleur, ou l'entier `40` pour la valeur.
2. Ces types ne sont pas assez parlants à la lecture du code.
3. Il n'est pas possible d'attacher des champs ou des méthodes à ces types.

Solution 1 : chaînes/entiers

Il y a une infinité de valeurs de type `String`, mais seules quatre d'entre-elles sont des couleurs valides. De même, il y a 2^{32} valeurs de type `int`, mais seules treize d'entre-elles sont des valeurs de cartes valides…

Cela force à faire constamment des vérifications coûteuses, p.ex. dans le constructeur de la classe `Card` :

```
public Card(String suit, int rank) {
  if (! (suit.equals(SPADES)
         || suit.equals(HEARTS)
         || suit.equals(DIAMONDS)
         || suit.equals(CLUBS)))
    throw new IllegalArgumentException(…);
  …
}
```

Solution 2 : objets

Une solution plus propre et conforme aux principes de la programmation orientée-objets consiste à définir une classe pour les couleurs et une pour les valeurs, et des constantes :

```
public final class Suit {
  public final static Suit SPADES =
    new Suit("SPADES");
  // … idem pour HEARTS, CLUBS et DIAMONDS
  private final String name;
  private Suit(String name) {
    this.name = name;
  }
}
public final class Rank { … }
```

Question : pourquoi le constructeur est-il privé ?

Solution 2 : objets

Une fois les classes `Suit` et `Rank` définies de la sorte, le constructeur de `Card` devient plus simple car il n'a plus besoin de vérifier ses arguments, qui sont valides par construction (si on ignore `null`, en tout cas) !

```
public final class Card {
  private final Suit suit;
  private final Rank rank;
  public Card(Suit suit, Rank rank) {
    this.suit = suit;
    this.rank = rank;
  }
  // … autres méthodes
}
```

Solution 2 : objets

Autre avantage de cette solution orienté-objet : il est possible d'ajouter des méthodes ou des champs aux classes `Suit` et `Rank`. Exemple :

```
public final class Suit {
  // … comme avant
  public String frenchName() {
    if (this == SPADES)   return "piques";
    if (this == DIAMONDS) return "carreaux";
    if (this == CLUBS)    return "trèfles";
    assert this == HEARTS;
    return "cœurs";
  }
}
```

Solution 3 : énumérations

La solution orientée-objets est bonne, mais aussi fastidieuse à mettre en œuvre, car elle demande l'écriture de beaucoup de code répétitif.
Pour éviter de devoir écrire ce code à chaque fois, Java offre la notion d'**énumération**.

Solution 3 : énumération

Au moyen des énumérations, les types `Suit` et `Rank` peuvent se définir simplement ainsi :

```
// fichier Suit.java
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS
};
// fichier Rank.java
public enum Rank {
  TWO, THREE, FOUR, FIVE, SIX, SEVEN,
  EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE
};
```

Java se charge de traduire ces énumérations en classes encore plus complètes que celles que nous avons écrites.

Traduction de `enum`

Une énumération Java est traduite en une classe et chaque élément de l'énumération est traduit en une instance de cette classe. Ces instances sont des champs statiques non modifiables de la classe de l'énumération, exactement comme dans notre version.
Java définit de plus quelques méthodes utiles sur les énumérations et leurs éléments…

Méthode `values`

La méthode statique `values`, définie sur l'énumération elle-même, retourne un tableau contenant la totalité des éléments de l'énumération, dans l'ordre de déclaration.

Par exemple, pour l'énumération `Suit` suivante :

```
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS
};
```

l'appel `Suit.values()` retourne un tableau de taille 4 contenant `SPADES` à la position 0, `DIAMONDS` à la position 1, et ainsi de suite.

Méthode `ordinal`

La méthode `ordinal` retourne un entier indiquant la position de l'élément dans l'énumération (à partir de zéro).

Par exemple, étant donnée l'énumération `Suit` suivante :

```
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS
};
```

`SPADES.ordinal()` retourne 0, `DIAMONDS.ordinal()` retourne 1, et ainsi de suite.

Les méthodes `ordinal` et `values` sont donc inverses l'une de l'autre.

Autres méthodes

Les éléments des énumérations sont également équipés d'autres méthodes, parmi lesquelles :

- `toString`, héritée de `Object` mais redéfinie pour retourner le nom de l'élément (p.ex. `"SPADES"`),
- `compareTo`, qui compare l'élément avec un autre élément de la *même* énumération et retourne une valeur négative si le premier est inférieur au second, nulle si les deux sont égaux, et positive sinon,
- `equals`, héritée de `Object`, qui compare l'élément avec un objet quelconque,
- `hashCode`, qui retourne le code de hachage de l'élément (concept que nous verrons plus tard).

Enoncé `switch`

Les éléments d'une énumération sont utilisables avec l'énoncé `switch`. Par exemple :

```
public final class Card {
  private final Suit suit;
  // … autres champs, constructeur, etc.
  public String frenchSuitName() {
    switch (suit) {
      case SPADES:   return "piques";
      case DIAMONDS: return "carreaux";
      case CLUBS:    return "trèfles";
      case HEARTS:   return "cœurs";
      default:       throw new Error();
    }
  }
}
```

obligatoire en raison d'une limitation de Java

Ajout de membres

Il est possible d'ajouter des membres (champs ou méthodes) aux énumérations, p.ex. pour mettre la méthode `frenchName` dans `Suit` plutôt que dans `Card` :

```
public enum Suit {
  SPADES, DIAMONDS, CLUBS, HEARTS;
  public String frenchName() {
    switch (this) {
      case SPADES:   return "piques";
      case DIAMONDS: return "carreaux";
      case CLUBS:    return "trèfles";
      case HEARTS:   return "cœurs";
      default:       throw new Error();
    }
  }
}
```

Constructeur

Il est aussi possible de définir un constructeur pour l'énumération, mais il ne peut être public. Exemple :

```
public enum Suit {
  SPADES("piques"), DIAMONDS("carreaux"),
  CLUBS("trèfles"), HEARTS("cœurs");

  private final String frenchName;
  public String frenchName() {
    return frenchName;
  }
  private Suit(String frenchName) {
    this.frenchName = frenchName;
  }
}
```

Enumérations imbriquées

Les énumérations `Suit` et `Rank` ne sont pas utiles en dehors de la classe `Card`. Il est préférable de les imbriquer dans celle-ci, ce qui est tout à fait possible :

```
public final class Card {
  public enum Suit { SPADES, …, HEARTS };
  public enum Rank { TWO, …, ACE };
  private final Suit suit;
  private final Rank rank;
  public Card(Suit suit, Rank rank) { … };
}
```

Depuis l'extérieur de la classe `Card`, les noms des différentes entités sont préfixés, p.ex. `Card.Suit`, `Card.Rank` ou encore `Card.Suit.SPADES`.

Enumérations imbriquées

Etant donné que :

- Java autorise l'imbrication des énumérations dans des classes, et que
- les énumérations sont traduites en classes,

on pourrait en conclure, avec raison, que l'imbrication de classes est également autorisé !

Classes imbriquées statiques

Pile d'entiers

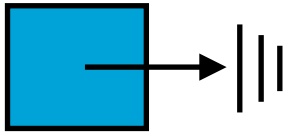
On désire modéliser une pile d'entiers (non bornée) offrant l'interface très simple suivante :

```
public interface IntStack {
  // Ajoute l'élément au sommet de la pile.
  public void addFirst(int value);

  // Enlève et retourne l'élément au sommet
  // de la pile. Lève IllegalStateException
  // si la pile est vide.
  public int removeFirst();

  // Retourne vrai ssi la pile est vide.
  public boolean isEmpty();
}
```

Pile d'entiers chaînée

Une manière de mettre en œuvre les piles d'entiers est de chaîner des nœuds contenant chacun un entier.

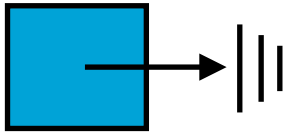
```
new LinkedIntStack()
```



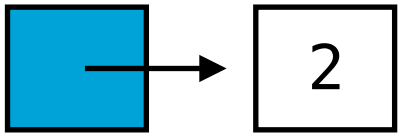
Pile d'entiers chaînée

Une manière de mettre en œuvre les piles d'entiers est de chaîner des nœuds contenant chacun un entier.

```
new LinkedIntStack()
```

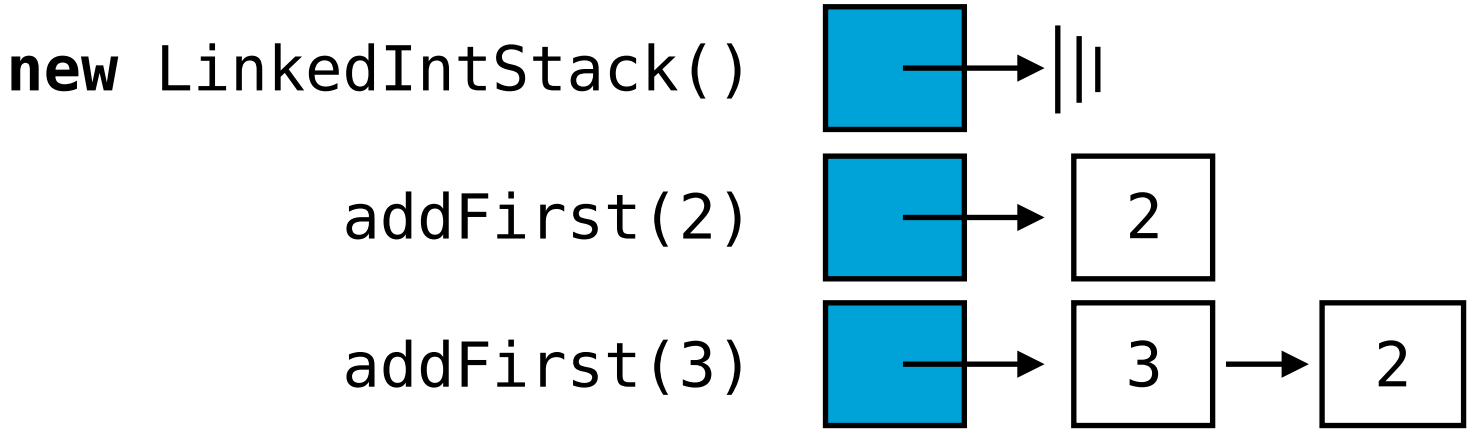


```
addFirst(2)
```



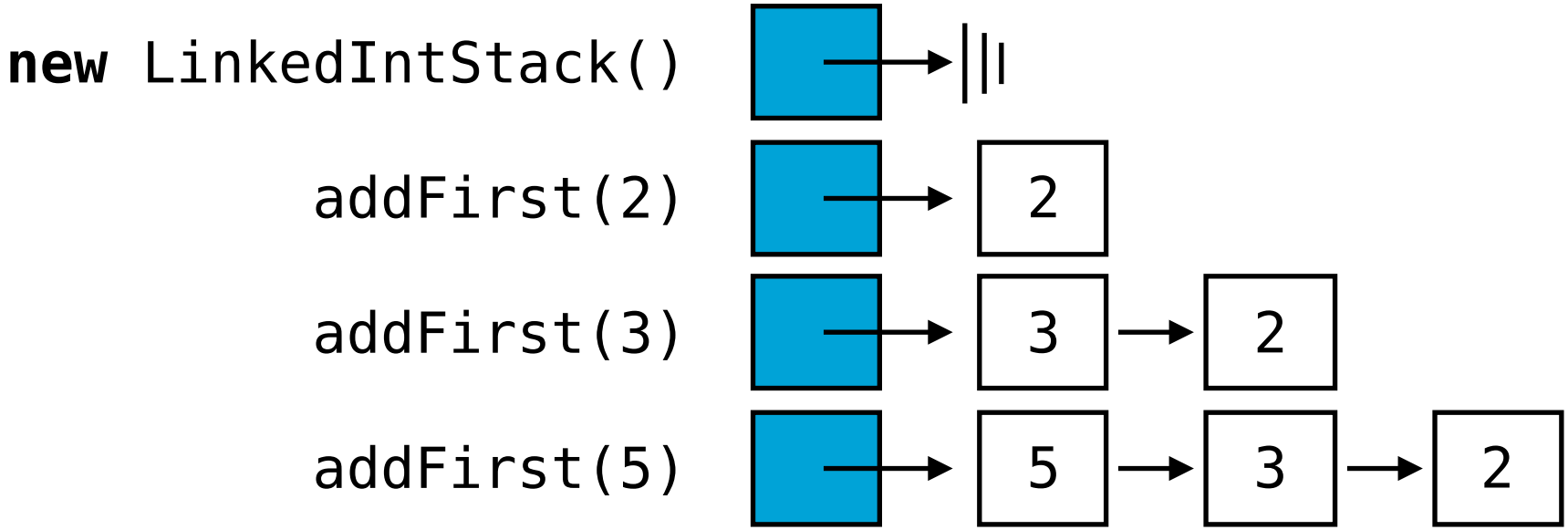
Pile d'entiers chaînée

Une manière de mettre en œuvre les piles d'entiers est de chaîner des nœuds contenant chacun un entier.



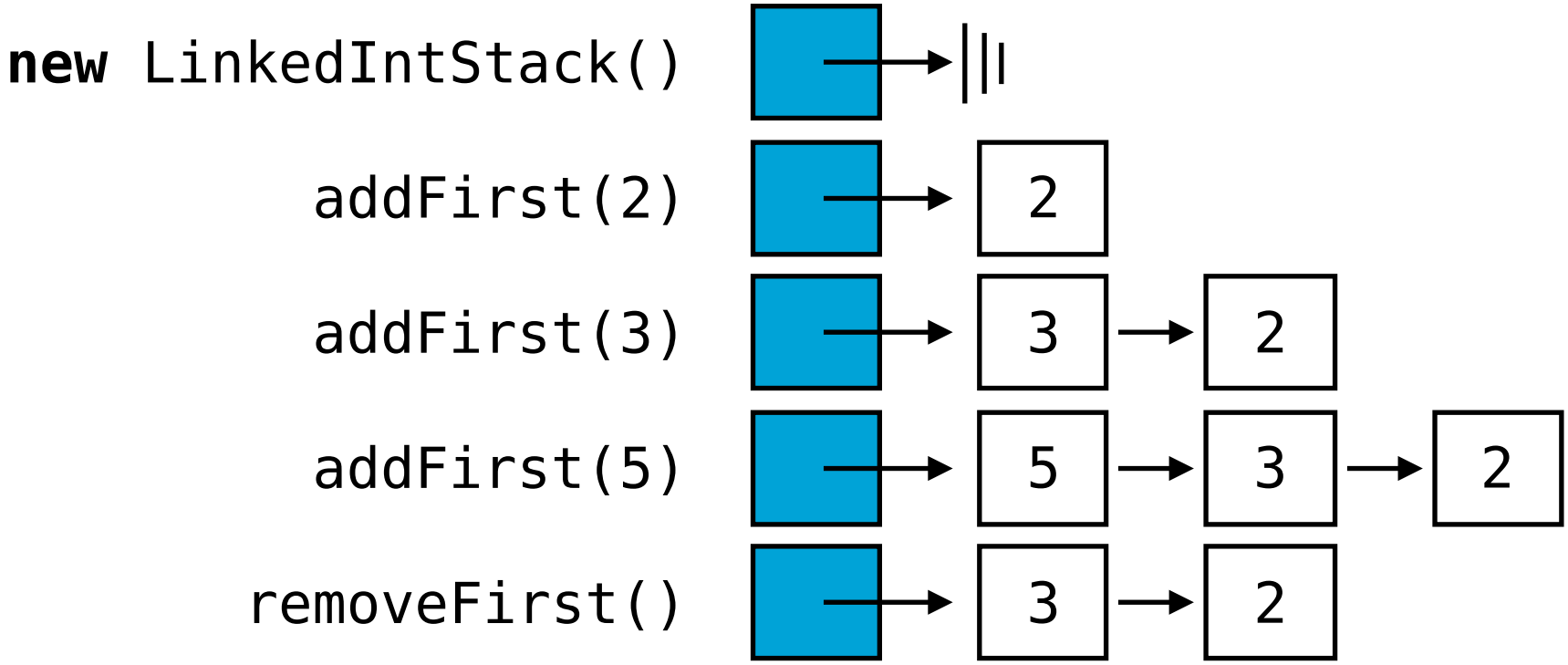
Pile d'entiers chaînée

Une manière de mettre en œuvre les piles d'entiers est de chaîner des nœuds contenant chacun un entier.



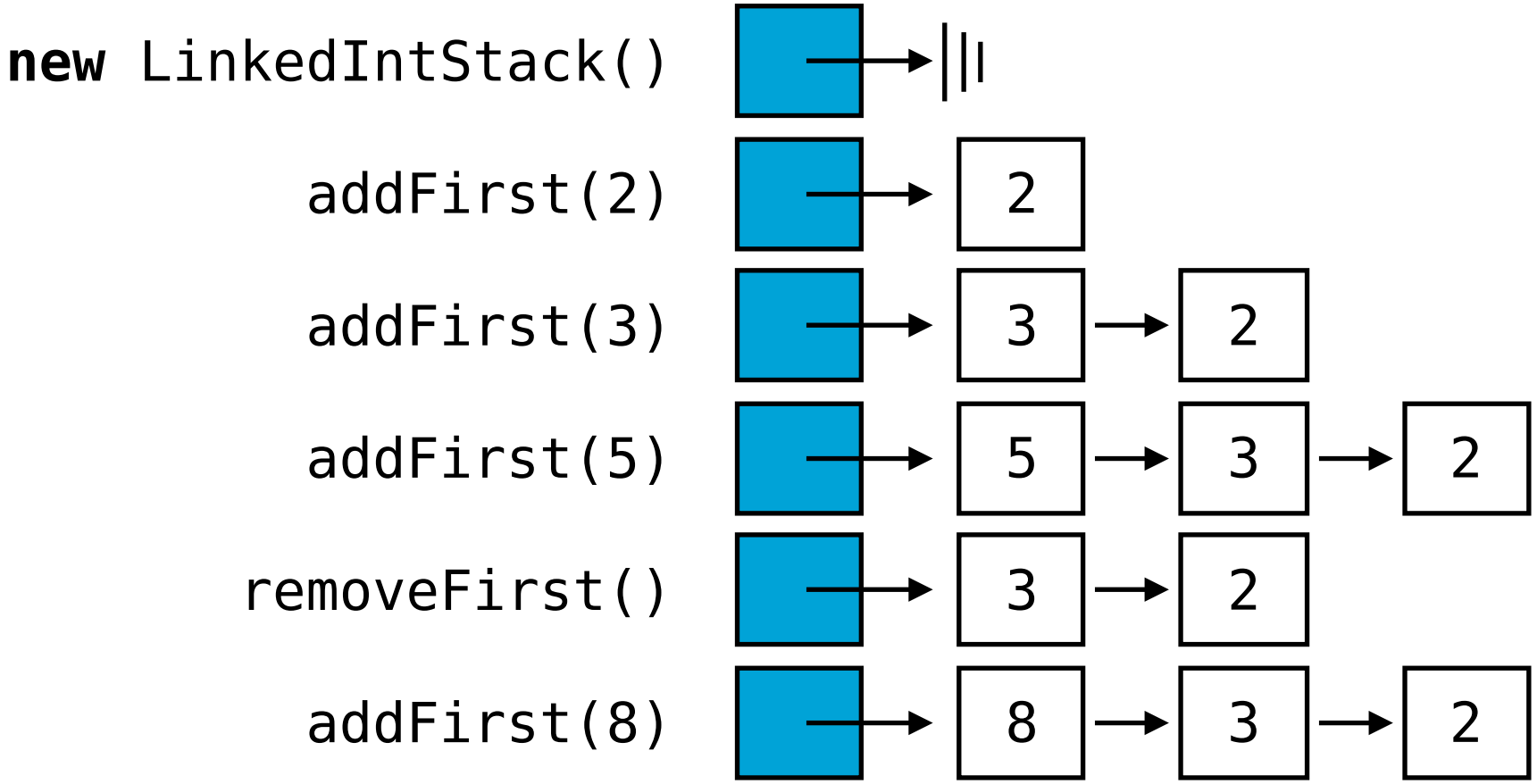
Pile d'entiers chaînée

Une manière de mettre en œuvre les piles d'entiers est de chaîner des nœuds contenant chacun un entier.



Pile d'entiers chaînée

Une manière de mettre en œuvre les piles d'entiers est de chaîner des nœuds contenant chacun un entier.



Pile d'entiers chaînée

On peut mettre cette idée en œuvre au moyen des deux classes suivantes :

```
public class LinkedIntStack
    implements IntStack {
  private LISNode first = null;
  // … isEmpty, addFirst, removeFirst
}
class LISNode {
  final int value;
  final LISNode next;
  public LISNode(int value, LISNode next) {
    this.value = value;
    this.next = next;
  }
}
```

Exercice

Donnez les définitions des méthodes `isEmpty`, `addFirst` et `removeFirst`.

Classe des nœuds

La classe des nœuds est un détail d'implémentation de la classe des piles. Il ne faudrait donc pas qu'elle soit visible de l'extérieur.
Une solution est de lui donner la visibilité par défaut, comme ici, ce qui la rend invisible en dehors du paquetage dans laquelle elle est définie.
Mais ce n'est pas encore totalement satisfaisant, il faudrait vraiment qu'elle soit invisible en dehors de la classe des piles…
Cela est possible en **imbriquant statiquement** la classe des nœuds dans celle des piles.

Classe nœud imbriquée

```
public class LinkedIntStack
    implements IntStack {
  private Node first = null;
  // … isEmpty, addFirst, removeFirst
  private static class Node {
    final int value;
    final Node next;

    public Node(int value, Node next) {
      this.value = value;
      this.next = next;
    }
  }
}
```

plus besoin de préfixe !

Question: le code des méthodes (`isEmpty`, …) change-t-il ?

Classe imbriquée statique

Une classe imbriquée statique (*static nested class*) est très similaire à une classe non imbriquée, si ce n'est que :

- elle peut être marquée `private` ou `protected`,
- elle a accès à tous les membres *statiques* de la classe englobante, y compris les privés (!),
- de l'extérieur, son nom est préfixé de celui de la classe englobante, comme pour les énumérations.

Attention : une classe imbriquée statique n'a pas accès aux membres non statiques de sa classe englobante !

(Note : les énumérations imbriquées sont toujours implicitement statique, même s'il est autorisé de leur attacher le modificateur `static`).

Utilité de l'imbrication

L'imbrication statique de classes est utile dans les situations suivantes :

1. lorsqu'une classe n'est utile que pour la mise en œuvre d'une autre, comme la classe des nœuds dans notre exemple,
2. lorsqu'une classe est subordonnée à une autre, et n'a pas de signification isolément, comme les bâtisseurs (*builders*) du projet.

Exercice

Si les classes imbriquées statiques avaient le droit d'accéder aux membres non statiques de leur classe englobante, le code ci-dessous serait valide :

```
public class Outer {
  private final int x;
  public Outer(int x) { this.x = x; }
  public static class Inner {
    public Inner() {
      System.out.println("x = " + x);
    }
  }
}
```

En quoi cela serait-il problématique ?

Classes imbriquées non statiques

Classes imbriquées

Java permet également d'imbriquer des classes de manière non statique.
Une telle classe imbriquée (appelée *inner class* en anglais) a les mêmes capacités qu'une classe imbriquée statique, mais peut en plus accéder à tous les membres non statiques de sa classe englobante, y compris les privés.
Les classes imbriquées non statiques sont également très utiles mais dans des situations différentes de celles rencontrées jusqu'à présent. Nous les étudierons plus tard.

Annotations Java (digression)

Annotations

Il est souvent intéressant d'attacher de l'information aux entités qui composent un programme (classes, interfaces, méthodes, champs, etc.).
Par exemple, on peut vouloir dire qu'une méthode d'une classe est obsolète et ne devrait plus être utilisée. Placer un commentaire dans la documentation de la méthode est bien, mais idéalement il faudrait que le compilateur détecte et signale toute utilisation d'une telle méthode.
Java offre pour cela la notion d'**annotation**.

Annotations en Java

En Java, les annotations ont un nom et, éventuellement, des paramètres nommés.
Une annotation est toujours attachée à une déclaration (de classe, d'interface, de méthode, de champ, de variable, etc.). Le nom de l'annotation est précédé d'une arobase (@) et suivi des éventuels paramètres, entre parenthèses.
Exemples :

```
public class LinkedIntStackTest {
  @Test
  public void isEmptyInitially() {

  @Test(expected = Exception.class)
  public void removeFirstEmpty() { … }
}
```

sans paramètre

avec paramètre expected

Définition d'annotations

Il est possible de définir de nouvelles annotations, au moyen d'une syntaxe proche de celle utilisée pour définir des interfaces.
Nous n'examinerons pas cette syntaxe ici.
Une annotation est toujours définie dans un paquetage, comme une classe ou une interface. Avant de pouvoir utiliser une annotation, il faut donc importer son nom !
(Exception : les annotations définies dans le paquetage `java.lang`, puisque celui-ci est importé par défaut. On les appelle **annotations prédéfinies**).

@Deprecated

L'annotation prédéfinie `Deprecated` s'attache aux entités (p.ex. les méthodes) qui sont obsolètes et qui ne sont gardées que pour garantir la compatibilité ascendante. Un avertissement est produit lors de l'utilisation d'une telle entité. Exemple :

```
public class Thread { …
  @Deprecated
  void destroy() { … }
}
public class ThreadUse {
  public void m(Thread t) {
    t.destroy();
  }
}
```

@Override

L'annotation prédéfinie `Override` s'attache à une méthode et déclare que celle-ci redéfinit (ou implémente) une méthode héritée. Elle est très utile pour détecter les erreurs bêtes, p.ex.:

```
public class Employee {
  private final String name;
  …
  @Override
  public boolean equals(Employee that) {…}
  @Override
  public int hashCode() {…}
}
```

(Conseil : utilisez *toujours* `Override` lors d'une redéfinition !)

@Override

L'annotation prédéfinie `Override` s'attache à une méthode et déclare que celle-ci redéfinit (ou implémente) une méthode héritée. Elle est très utile pour détecter les erreurs bêtes, p.ex.:

```
public class Employee {
  private final String name;
  …
  @Override
  public boolean equals(Employee that) {…}
  @Override
  public int hashCode() {…}
}
```

(Conseil : utilisez *toujours* `Override` lors d'une redéfinition !)

@SuppressWarnings

L'annotation `SuppressWarnings` permet de supprimer un avertissement produit par le compilateur.

Exemple :

```
@SuppressWarnings("deprecation")
public printYear(java.util.Date date) {
  System.out.println(1900 + date.getYear());
}
```

Avant de supprimer un avertissement, il faut bien entendu comprendre pourquoi il apparaît, et pourquoi il peut être ignoré !

(Conseil : insérez toujours ces annotations via la commande *Quick Fix* d'Eclipse, après mûre réflexion.)

`@Test` (JUnit)

La bibliothèque JUnit définit l'annotation `@Test`, qui s'attache aux méthodes et permet à JUnit de distinguer celles qui représentent des tests (et qui doivent donc être exécutées) des autres.

Elle accepte un paramètre optionnel, nommé `expected`, qui contient la classe de l'exception que le test devrait lever.

Exemple :

```
public class LinkedIntStackTest {
  @Test
  public void isEmptyInitially() { … }
  @Test(expected =
          IllegalStateException.class)
  public void removeFirstEmpty() { … }
}
```

Résumé

Les énumérations Java permettent de décrire très facilement les types dont les valeurs sont énumérables (mois de l'année, jours de la semaine, saisons, couleurs de cartes, etc.).

* * *

La possibilité d'imbriquer *statiquement* des classes Java permet de renforcer l'encapsulation, soit en évitant de rendre visible des classes à usage privé, soit en associant fortement une classe subordonnée à la classe dont elle dépend.