ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 20 mars 2011

Résumé

- Iterator (Partie 2)
  - Classe interne
  - Implémentation ≪fail-fast≫
  - Ajout des méthodes **add, remove**
  - Nouvelle syntaxe 1.5

∗. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Implémentation -2- : classe interne non static

Nous utiliserons une **classe interne** (imbriquée et non static, inner class) pour créer un itérateur :

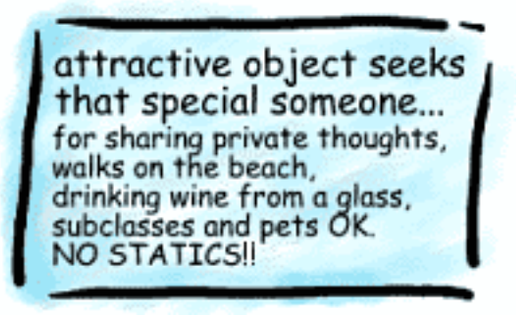
- L'instance d'une **classe interne** ne peut exister en l'absence d'un objet de la classe externe (le contraire est faux) ;
- L'instance d'une **classe interne** a accès aux variables et méthodes d'instance de l'objet qui l'a créé.

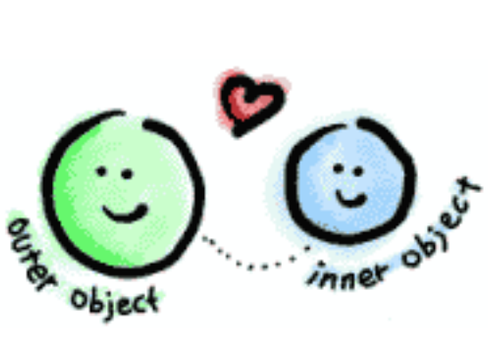
```
class Outer {
  class Inner {

  }
  Inner newInner() {
     return new Inner();
  }
}
```

$\Rightarrow$ Pensez aux variables et méthodes d'instance.

“Getting in Touch with your Inner Class”

www.javaranch.com/campfire/StoryInner.jsp





Classe interne non static

Une **classe interne non static** (**inner** class) est une classe imbriquée non ≪static≫.

- Un objet d'une classe interne non static requière un objet de la classe externe (mais un objet de la classe externe ne requière pas un objet de la classe interne) ;
- **Un objet d'une classe interne non static a accès aux variables et méthodes de l'objet de la classe externe à partir duquel il a été créé.**

```
class Outer {
  class Inner {
  }
}
```

⇒ Pensez aux variables et méthodes d'instance.

```
public class Outer {
    private class Inner {
        private Inner() {
            System.out.println( "* new instance of Inner class *" );
        }
    }
    public Outer() {
        System.out.println( "* new instance of Outer class *" );
    }
}
class Test {
    public static void main( String[] args ) {
        Outer o = new Outer();
    }
}
```

`java Test` imprimera "* new instance of Outer class *", ainsi **l'objet de la classe Outer existe sans qu'un objet de la classe Inner n'ait été créé (rien d'automatique ici)**.

```
public class Outer {
    private class Inner {
        private Inner() {
            System.out.println( "* new instance of Inner class *" );
        }
    }
    public Outer() {
        System.out.println( "* new instance of Outer class *" );
    }
    public static void doesNotWork() {
        Inner i = new Inner();
    }
}
Outer.java:11:
non-static variable this cannot be referenced from a static context
        Inner i = new Inner ();
                  ^
```

$\Rightarrow$ **Un objet d'une classe interne non static doit être créé par un objet de la classe externe**.

```
public class Outer {
    private class Inner {
        private Inner() {
            System.out.println( "* new instance of Inner class *" );
        }
    }
    private Inner i;
    public Outer() {
        System.out.println( "* new instance of Outer class *" );
        i = new Inner();
    }
}
class Test {
    public static void main( String[] args ) {
        Outer o = new Outer();
    }
}
> java Test
* new instance of Outer class *
* new instance of Inner class *
```

```
public class Outer {
    private class Inner {
        private Inner() {
            System.out.println( "* new instance of Inner class *" );
            System.out.println( "value = " + value ); // <---
        }
    }
    private Inner o;
    private String value; // <---
    public Outer() {
        System.out.println( "* new instance of Outer class *" );
        value = "from the instance variable of the outer object"; // <---
        o = new Inner();
    }
}
class Test {
    public static void main( String[] args ) {
        new Outer();
    }
}
```

```
> java Test
* new instance of Outer class *
* new instance of Inner class *
value = from the instance variable of the outer object
```

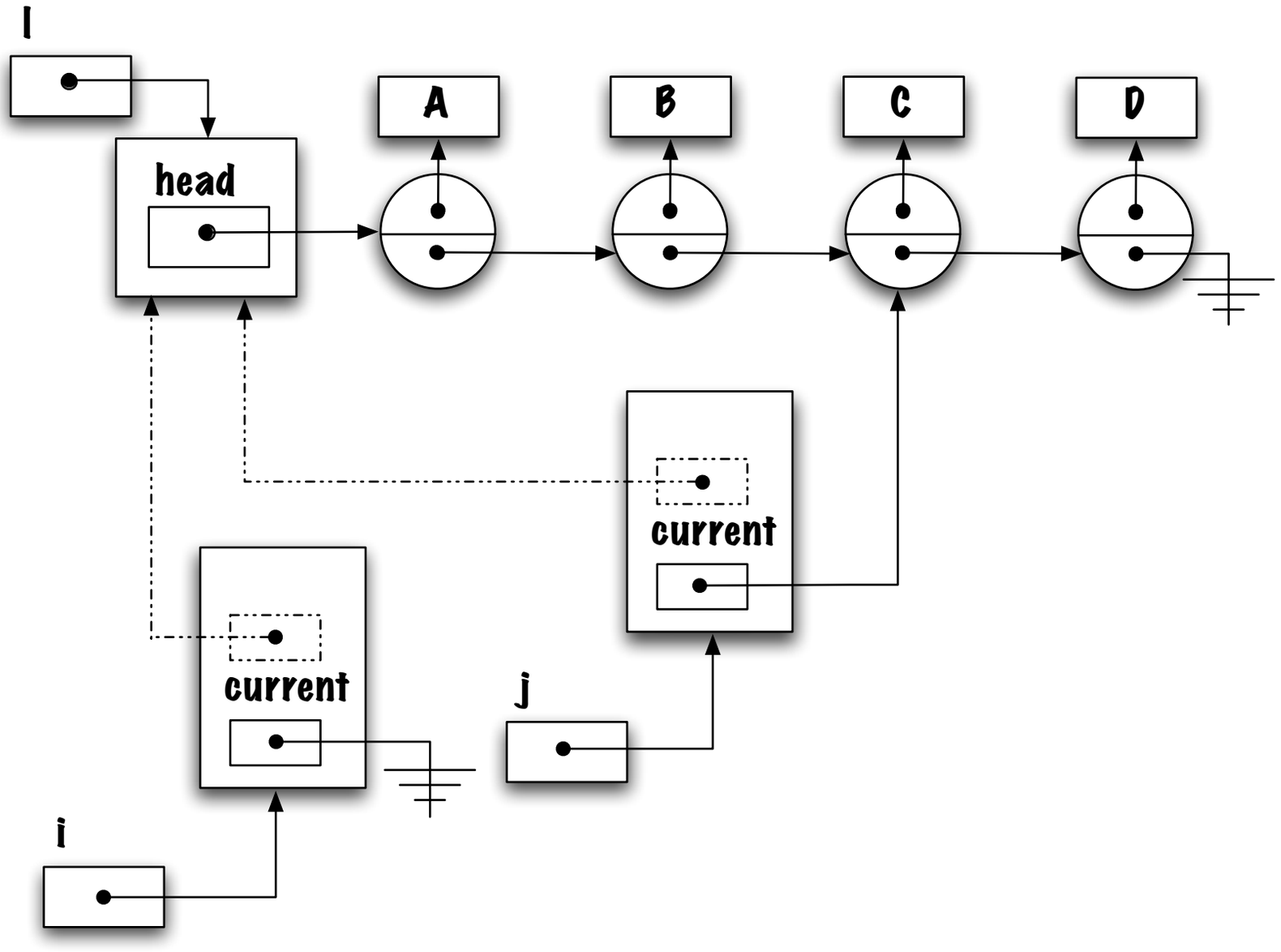
Un objet d'une classe interne non static a accès aux variables et méthodes de son objet externe !

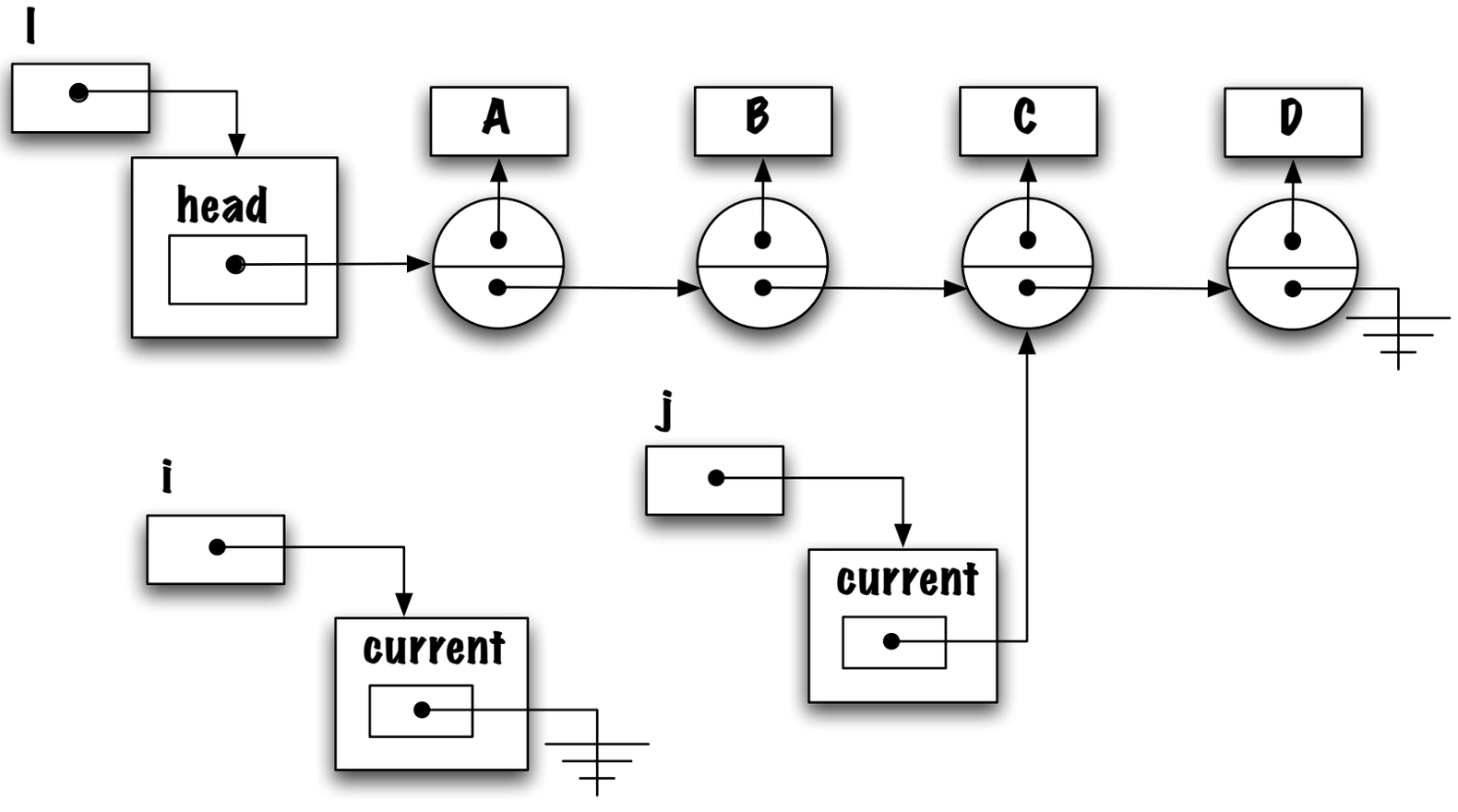
Une classe imbriquée ≪static≫ peut simuler cette relation ≪objet interne≫-≪objet extérieur≫ comme suit.

```
public class Outer {
    private static class Inner {
        private Outer parent; // <--
        private Inner( Outer parent ) {
            this.parent = parent;  // <--
            System.out.println( "* new instance of Inner class *" );
            System.out.println( "value = " + parent.value ); // <--
        }
    }
    private Inner o;
    private String value;
    public Outer() {
        System.out.println( "* new instance of Outer class *" );
        value = "from the instance variable of the outer object"; // <--
        o = new Inner( this );
    }
}
```

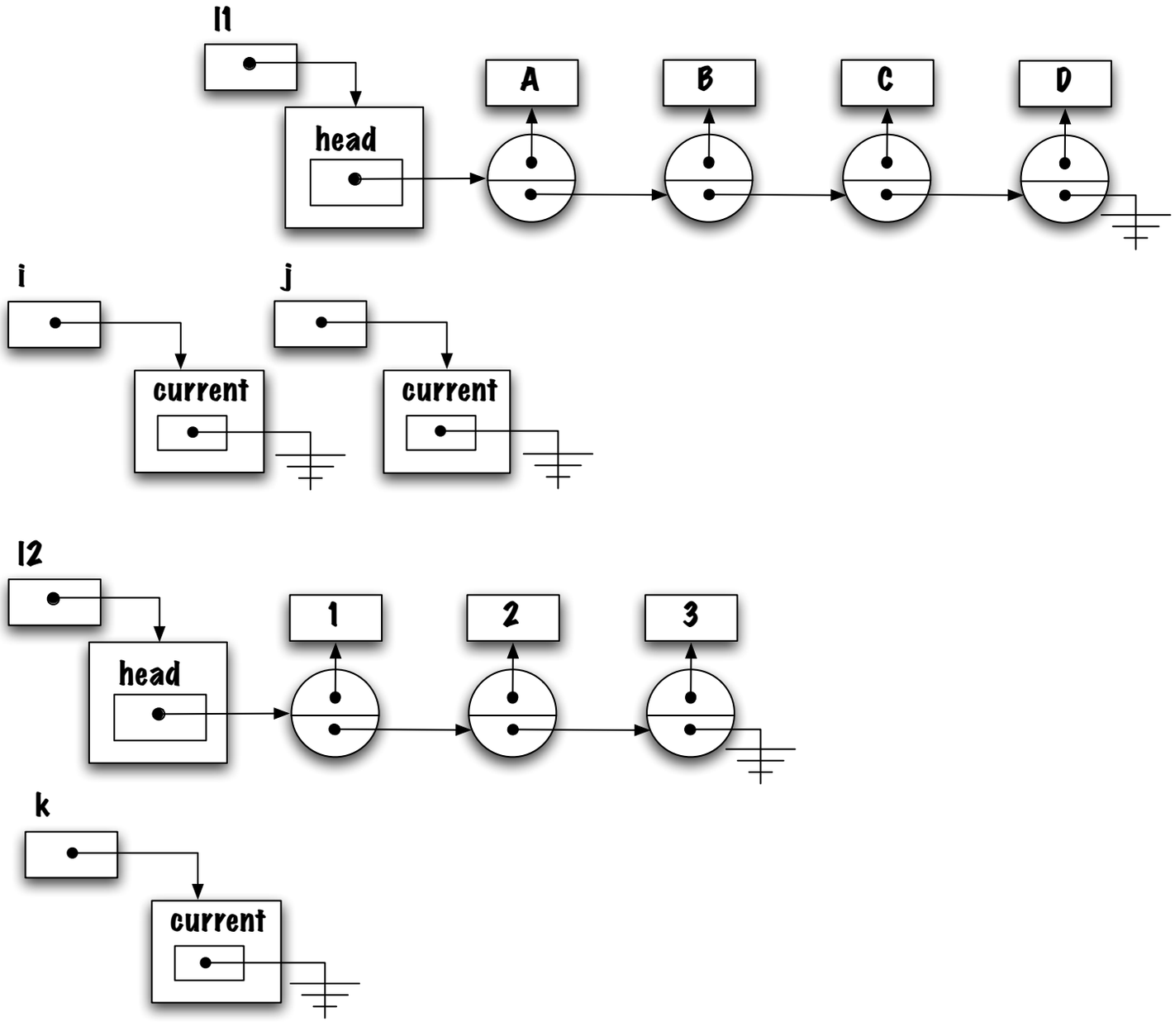
Classe interne : résumé

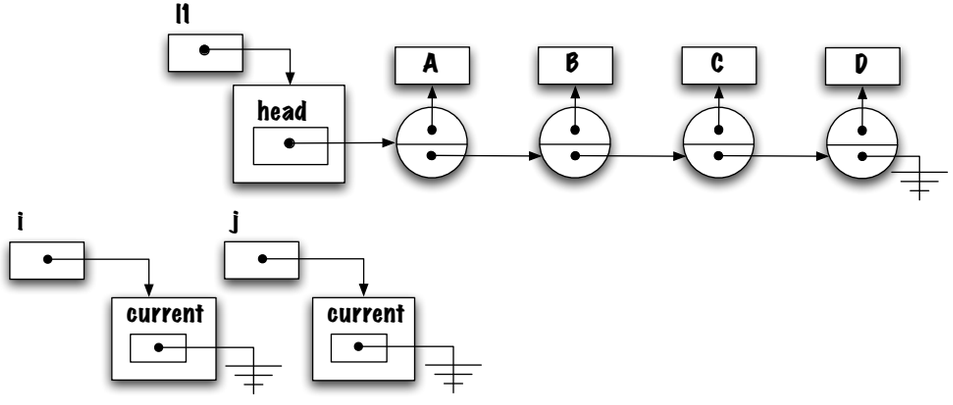
- Une *classe interne non static* est une classe imbriquée dont la déclaration n'est pas ≪static≫ ;
- Un objet de la classe externe crée une instance de la classe interne :
  - L'objet de la classe interne non static est lié à une et une seule **instance** de la classe externe (celle qui l'a créée) ;
  - L'objet de la class externe peut créer **zéro**, **une**, **plusieurs** instances de sa classe interne non static ;
  - **Un objet d'une classe interne non static a accès aux variables et méthodes de son objet externe**.

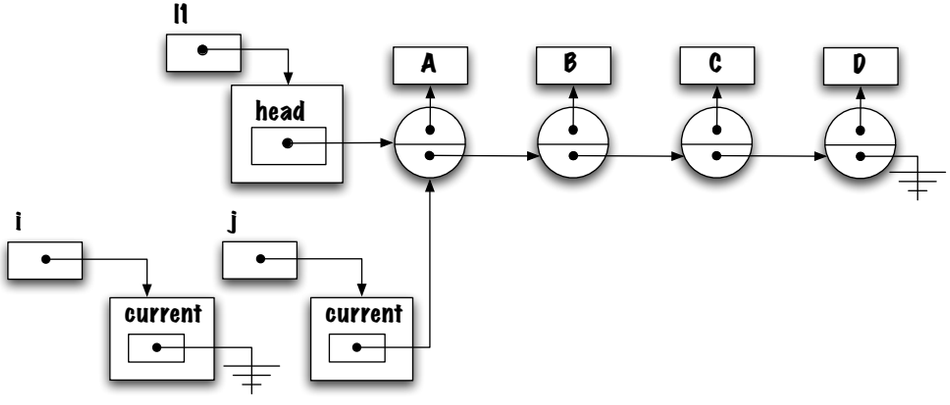

l
head
A
B
C
D
current
current
j
i

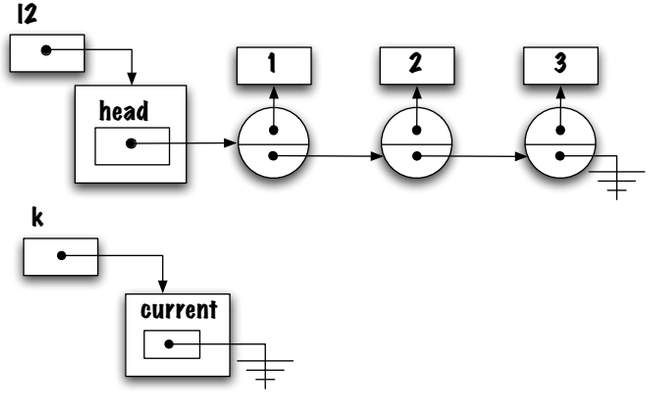

l
head
A
B
C
D
i
current
j
current

```
LinkedList l1 = new LinkedList();
l1.addLast( "A" );
l1.addLast( "B" );
l1.addLast( "C" );
l1.addLast( "D" );

LinkedList l2 = new LinkedList();
l2.addLast( new Integer( 1 ) );
l2.addLast( new Integer( 2 ) );
l2.addLast( new Integer( 3 ) );

Iterator i, j, k;

i = l1.iterator();
j = l1.iterator();
k = l2.iterator();
```

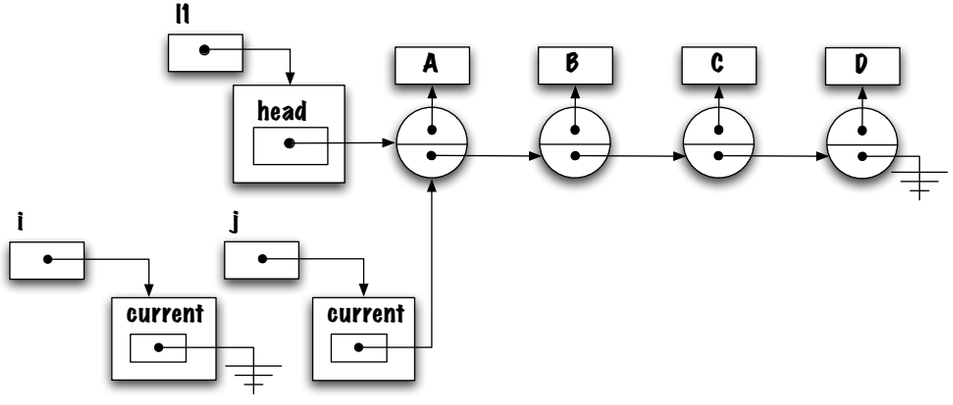

l1
head
A
B
C
D
i
current
j
current
l2
head
1
2
3
k
current

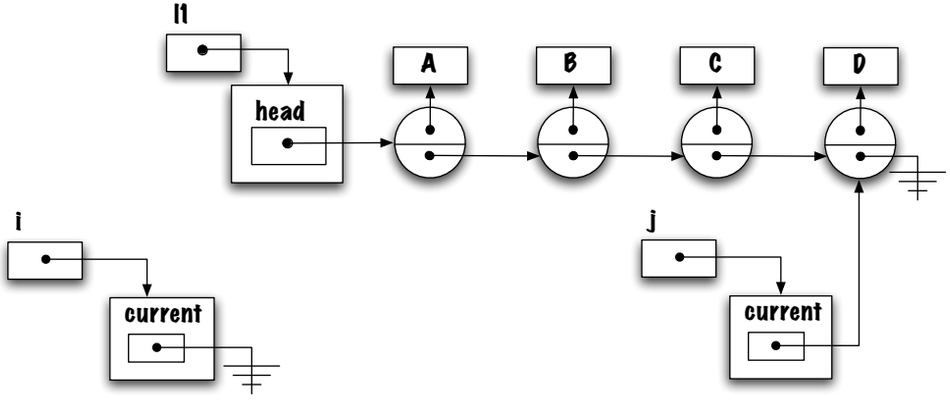


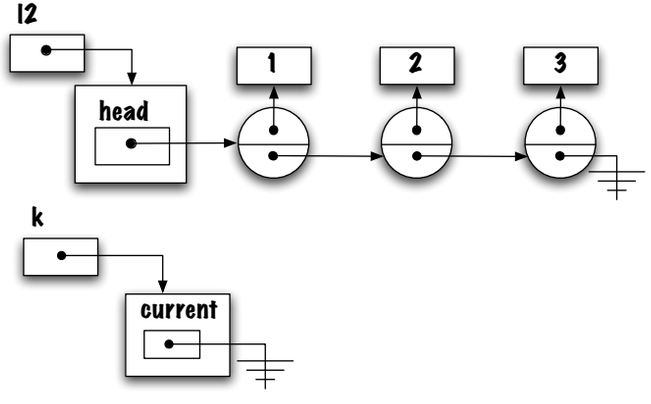




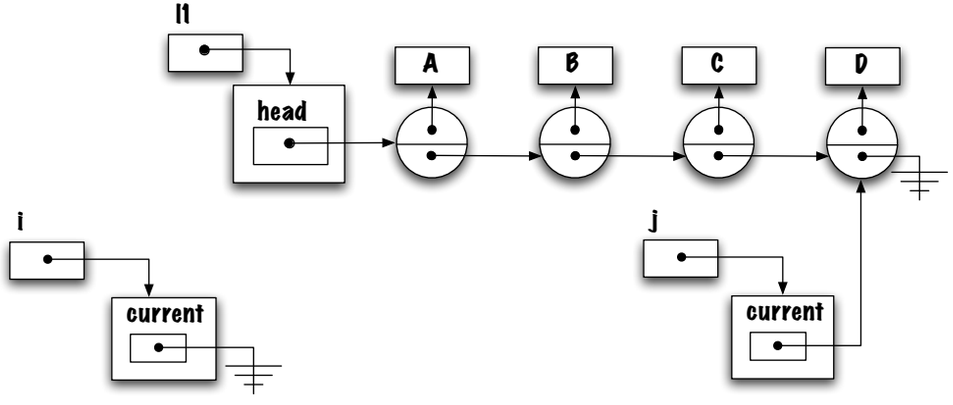

```
if ( j.hasNext() ) {
    String o = j.next();
}
```

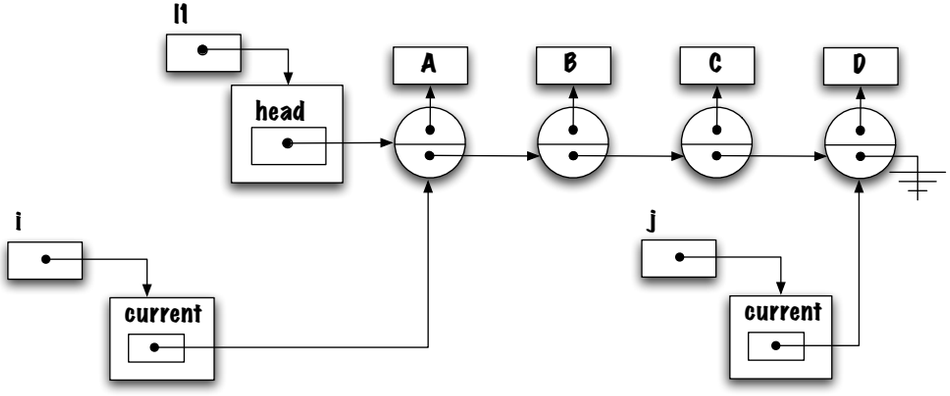


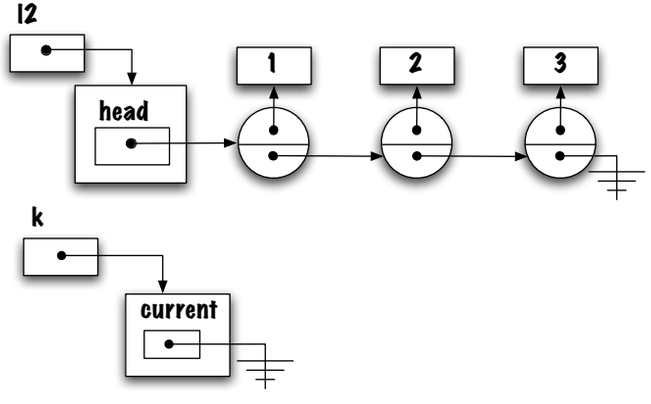




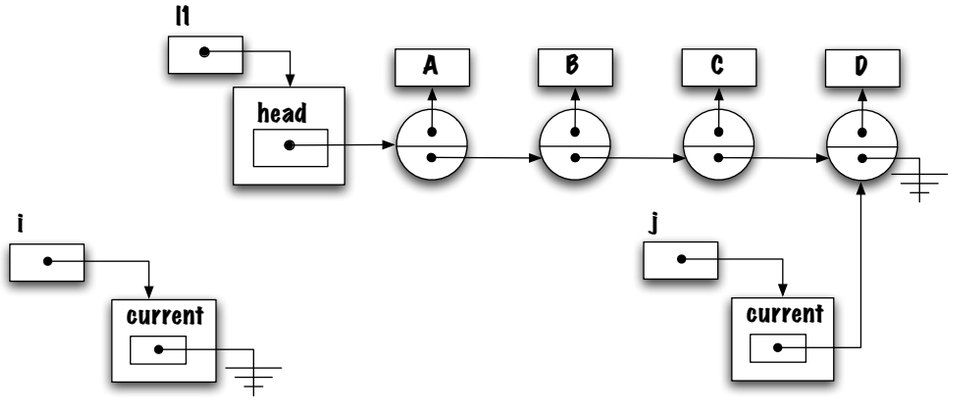

```
while ( j.hasNext() ) {
    String o = j.next();
}
```

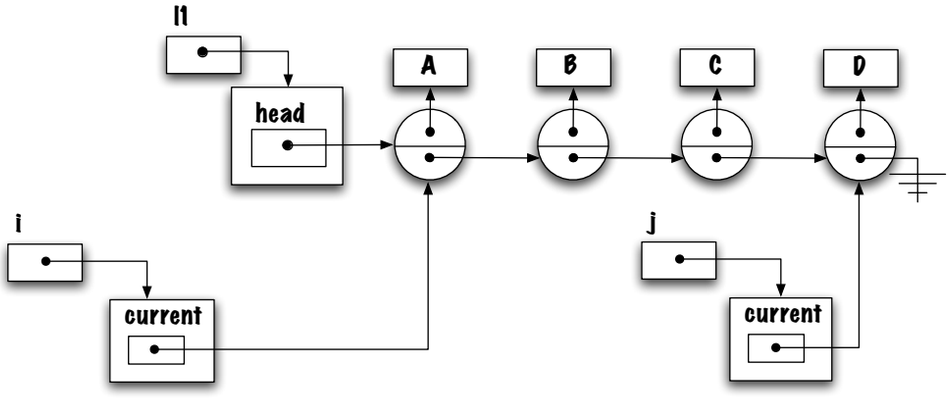


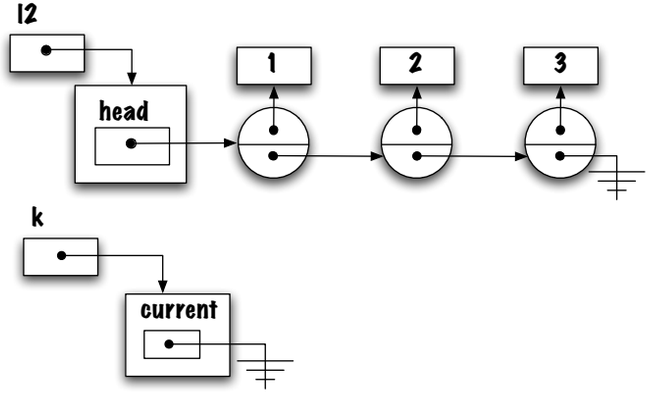


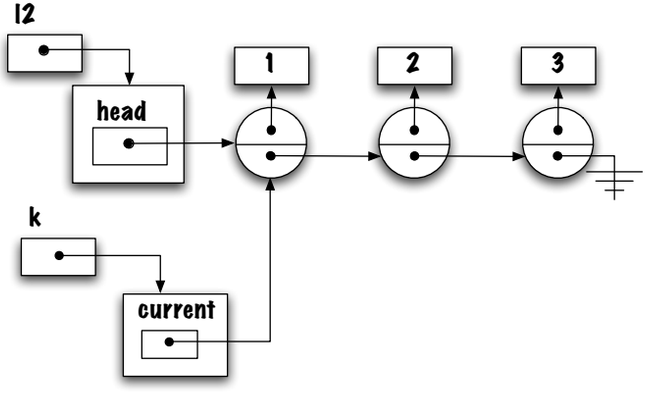



```
if ( i.hasNext() ) {
    Object o = i.next();
}
```

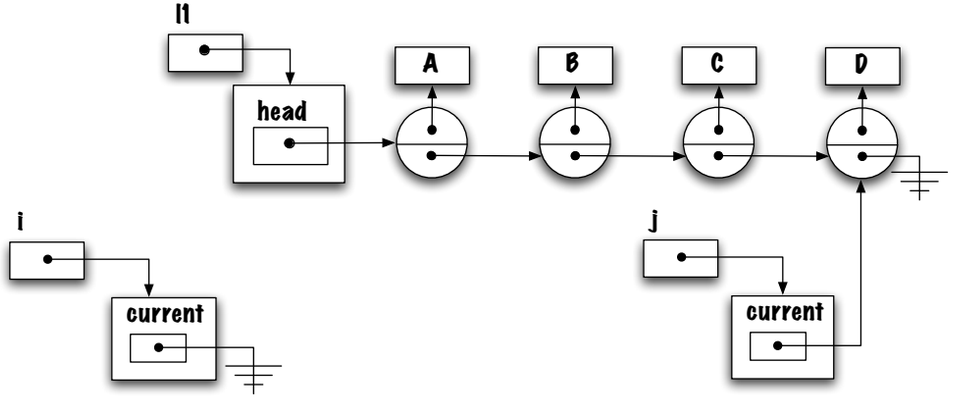


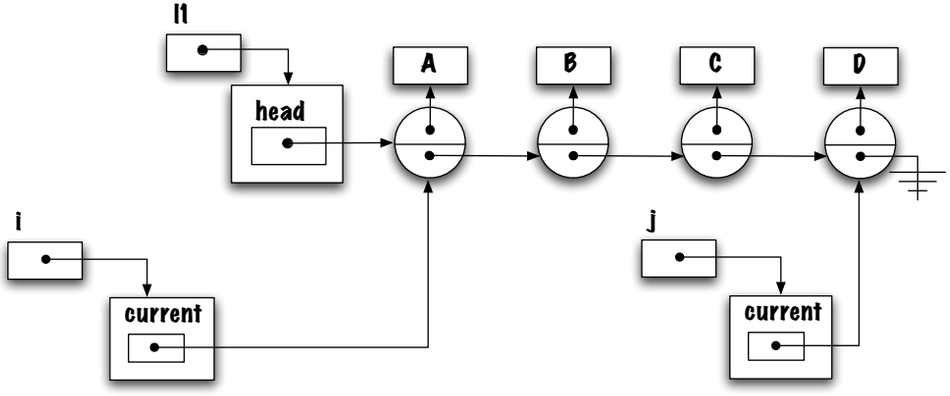


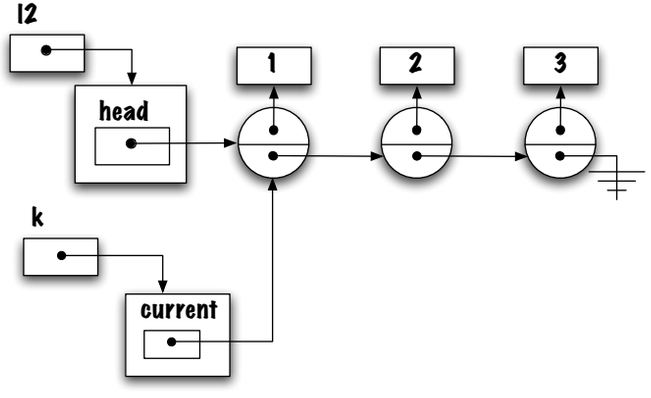


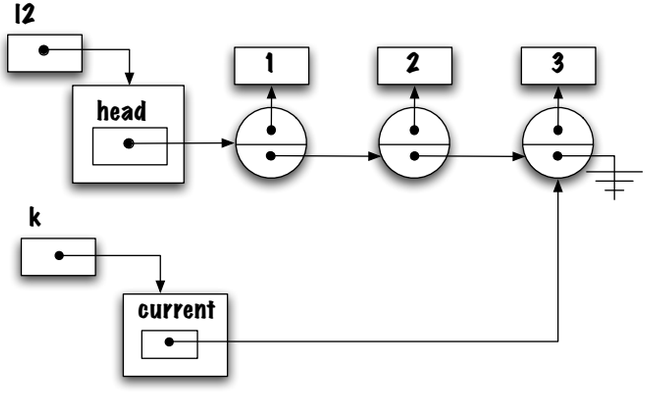



```
if ( k.hasNext() ) {
    Integer o = k.next();
}
```










```
while ( k.hasNext() ) {
    Integer o = k.next();
}
```

```
public class LinkedList<E> implements List<E> {
    private static class Node<E> { ... }
    private Node<E> head;
    private class ListIterator implements Iterator<E> {
       // ...
    }
}
```

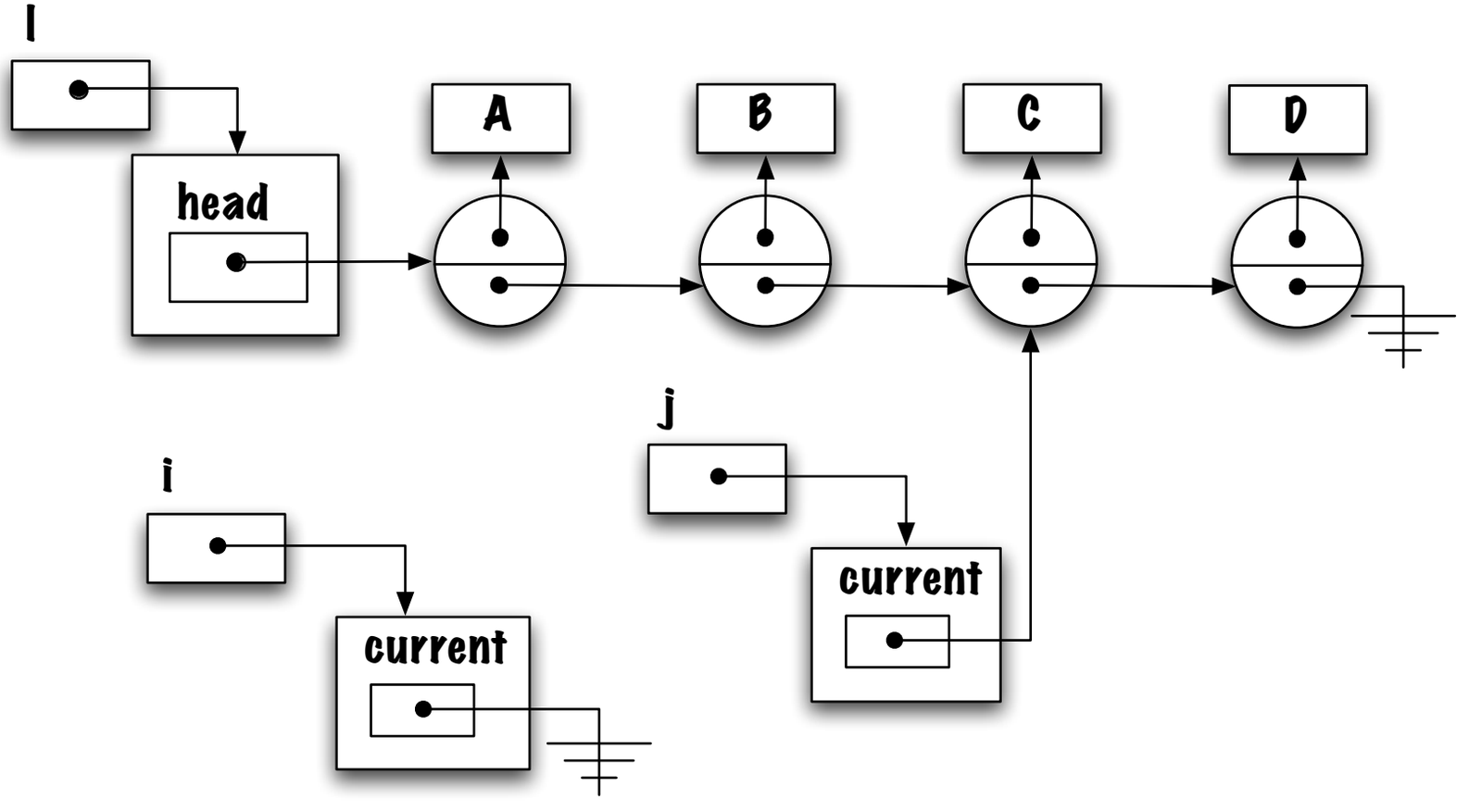
- **ListIterator** est une classe interne non static ;
- **ListIterator** réalise **Iterator** ;
- **LinkedList** n'implémente pas **Iterator**.

⇒ Quelles sont les variables instances de la classe **ListIterator** ?

Chaque objet doit posséder une variable afin de désigner un élément de la liste : **current**.

```
public class LinkedList<E> implements List<E> {

    private static class Node<E> { ... }

    private Node<E> head;

    private class ListIterator implements Iterator<E> {
        private Node<E> current; // <---
        ...
    }
}
```

Cette nouvelle implémentation permet à plusieurs itérateurs de co-exister.



Le constructeur initialise la variable **current** avec la valeur **null** ; désignant ainsi la situation où l'itérateur se trouve à la gauche de la liste.

```
public class LinkedList<E> implements List<E> {
    private static class Node<E> { ... }

    private Node<E> head;

    private class ListIterator implements Iterator<E> {
        private Node<E> current;
        private ListIterator() {
            current = null;
        }
        ...
    }
    ...
}
```

Comment crée-t-on un itérateur ?

```
public class LinkedList<E> implements List<E> {
    private static class Node<E> { ... }

    private Node<E> head;

    private class ListIterator implements Iterator<E> {
        private Node<E> current;
        private ListIterator() {
            current = null;
        } ...
    }
    public Iterator<E> iterator() {
        return new ListIterator();
    } ...
}
```

⇒ **iterator()** : est une méthode d'instance de la classe **LinkedList**.

```
public class LinkedList<E> implements List<E> {
    private static class Node<E> { ... }
    private Node<E> head;
    private class ListIterator implements Iterator<E> {
        private Node<E> current;
        private ListIterator() {
            current = null;
        }
        public E next() {
            if ( current == null ) {
                current = head;
            } else {
                current = current.next;
            }
            return current.value;
        }
        public boolean hasNext() { ... }
    }
    public Iterator<E> iterator() {
        return new ListIterator();
    } ...
}
```

Usage

```
List<Double> doubles = new LinkedList<Double>();

doubles.add( new Double( 5.1 ) );
doubles.add( new Double( 3.2 ) );

double somme = 0.0;

Iterator<Double> i = doubles.iterator();

while ( i.hasNext() ) {
    Double v = i.next();
    somme = somme + v.doubleValue();
}
```

$\Rightarrow$ Exercise : tracez l'exécution de ces énoncés.

Est-ce rapide ?

Nous avons fait tout ce travaille afin fournir un mécanisme permettant de traverser la liste de l'extérieur, est-ce que ça vaut la peine ?

# noeuds	.next (ms)	itérateur (ms)
10,000	2	6
20,000	4	5
40,000	8	11
80,000	14	19
160,000	23	48

Multiples itérateurs

```
List<Double> doubles = new LinkedList<Double>();

for ( int c=0; c<5; c++ ) {
    doubles.add( new Double( c ) );
}
Iterator<Double> i = doubles.iterator();
while ( i.hasNext() ) {
    Double iVal = i.next();
    Iterator<Double> j = doubles.iterator();
    while ( j.hasNext() ) {
        Double jVal = j.next();
        System.out.println( "("+iVal+","+jVal+")" );
    }
}
```

$\Rightarrow$ (0.0,0.0), (0.0,1.0), . . ., (0.0,4.0), . . ., (4.0,4.0).

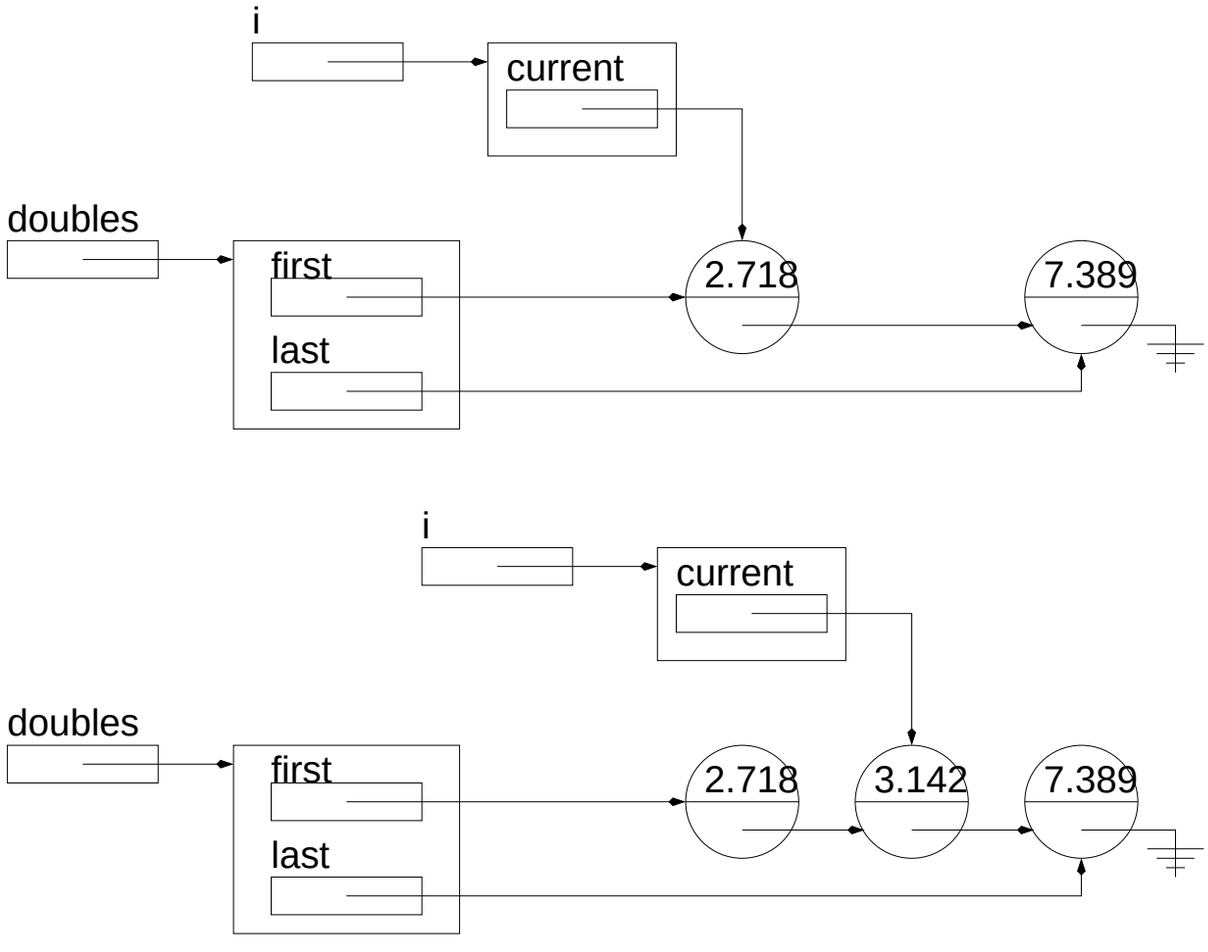
Méthodes add et remove

Ajoutons les méthodes **add(E o)** et **remove()** à l'interface **Iterator**.

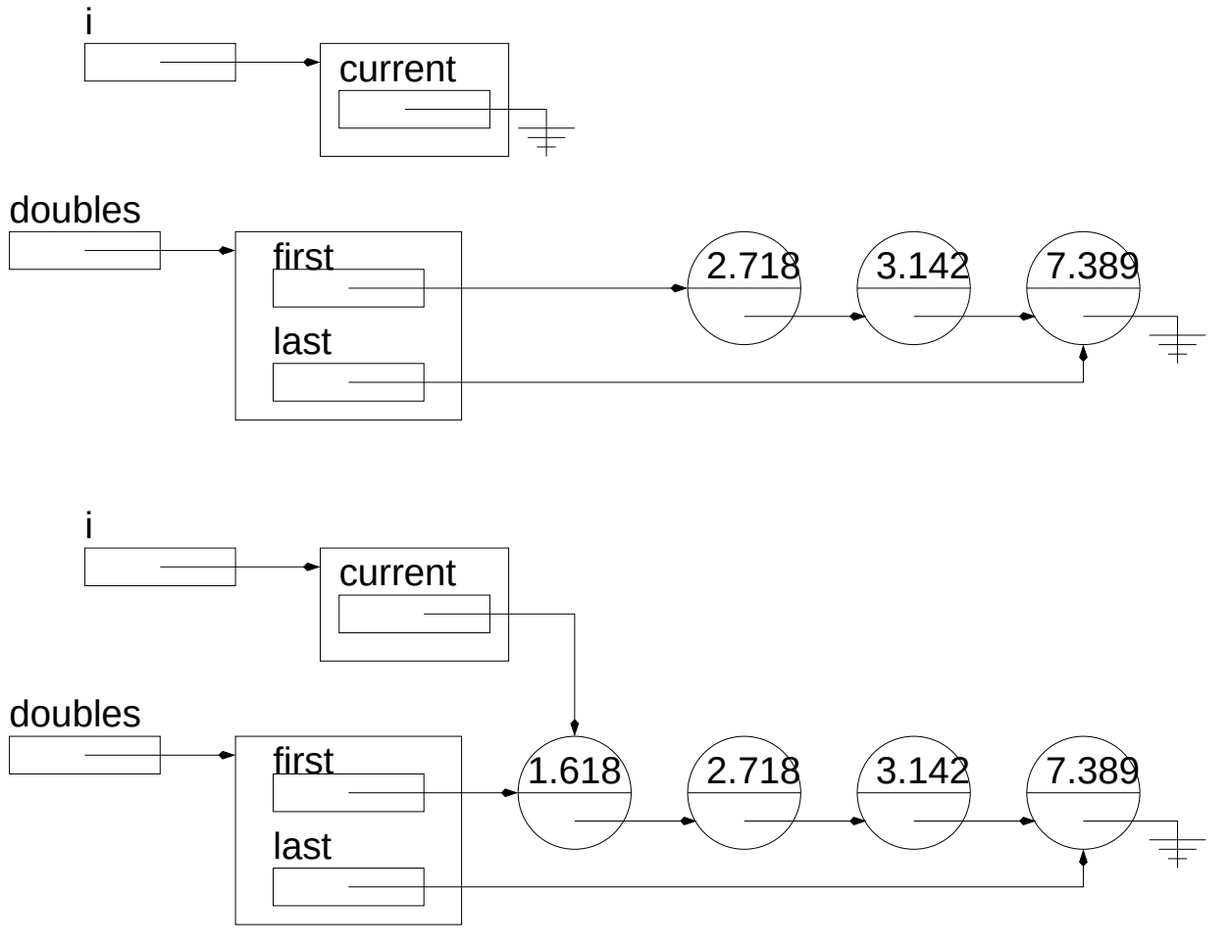
add()

i.add(value) insère la nouvelle valeur avant l'élément retourné par le prochain appel à la méthode **i.next()**.

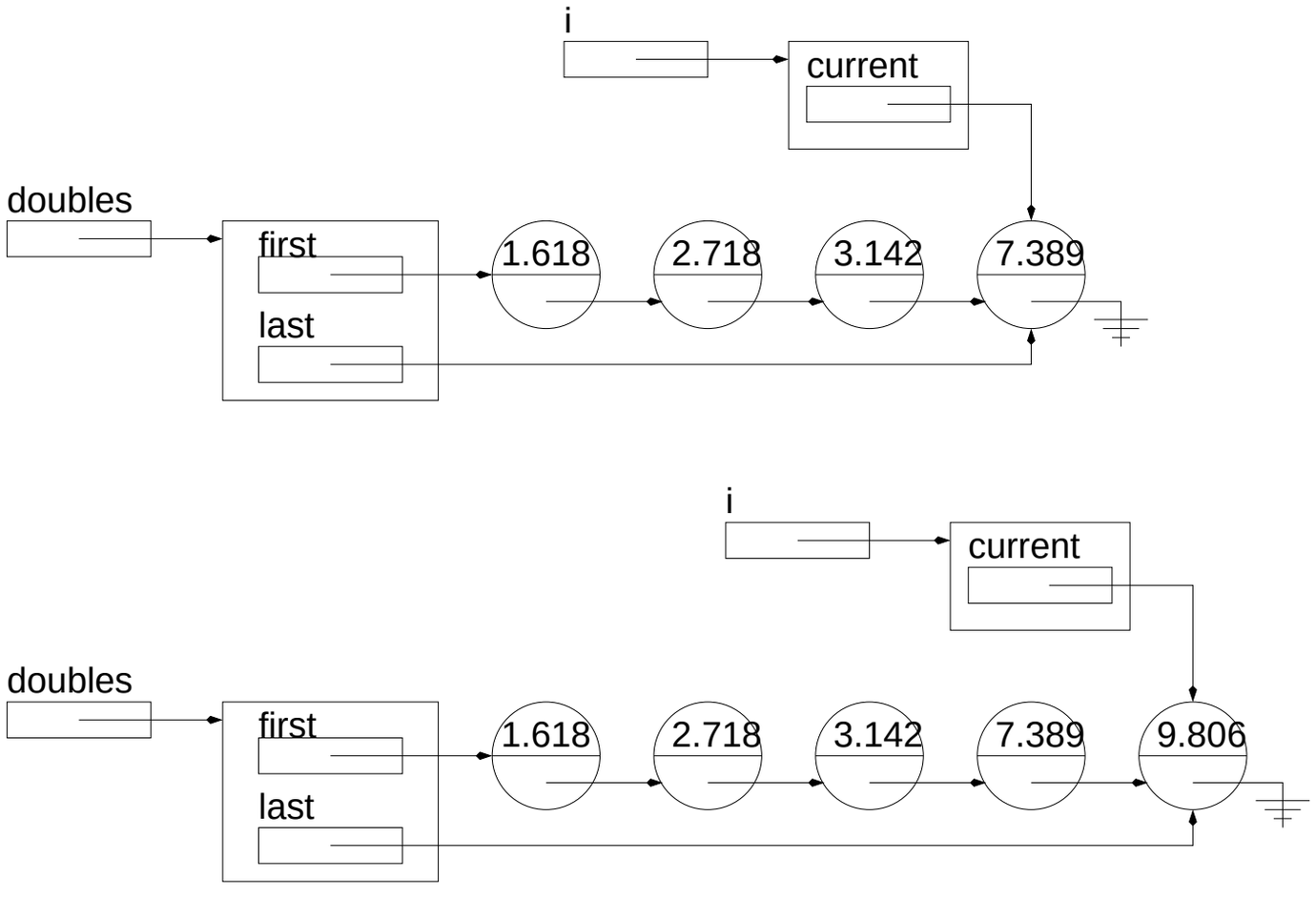
Ainsi, la méthode **i.next()** retourne la même valeur avec ou sans ajout.



$\Rightarrow$ L'appel **i.add(new Double(3.142))**, ajoute à une position intermédiaire dans la liste, remarquez que la méthode **i.next()** retournera la valeur avec ou sans insertion.



$\Rightarrow$ ajout au début de la liste, **i.add(new Double(1.618))**, encore **i.next()** retournera la même valeur avec ou sans insertion.



$\Rightarrow$ Ajout d'un élément à la fin de la liste, **i.add(new Double(1.618))**, avec ou sans insertion, **hasNext()** retourne la même la même valeur.

Quel est le résultat ?

```
public class Test {
    public static void main( String[] args ) {
        List<String> l = new LinkedList<String>();
        Iterator<String> i = l.iterator();

        for ( int c=0; c<5; c++ ) {
            i.add( "element-" + c );
        }
        i = l.iterator();
        while ( i.hasNext() ) {
            System.out.println( i.next() );
        }
    }
}
```

element-0
element-1
element-2
element-3
element-4

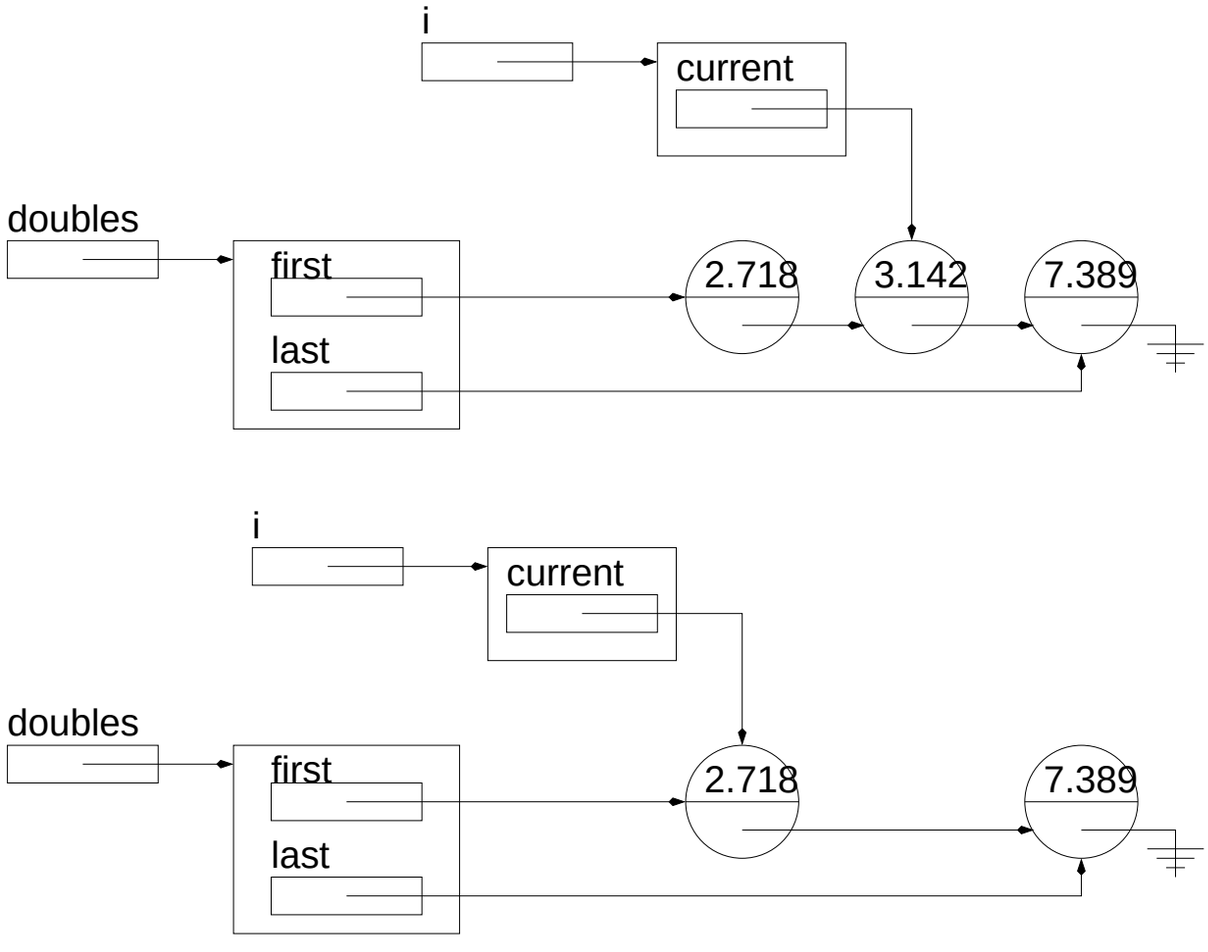
```
public class LinkedList<E> implements List<E> {
    private static class Node<E> { ... }
    private Node<E> head;
    private class ListIterator implements Iterator<E> {
        private Node<E> current;
        private ListIterator() { current = null;  }
        public E next() { ... }
        public boolean hasNext() { ... }
        public void add( E o ) {
            if ( current == null ) {
                head = new Node<E>( o, head );
                current = head;
            } else {
                current.next = new Node<E>( o, current.next );
                current = current.next;
            }
        }
        public void remove() { ... }
    }
    public Iterator<E> iterator() { return new ListIterator(); }
    // ... all the usual methods of LinkedList
}
```

remove()

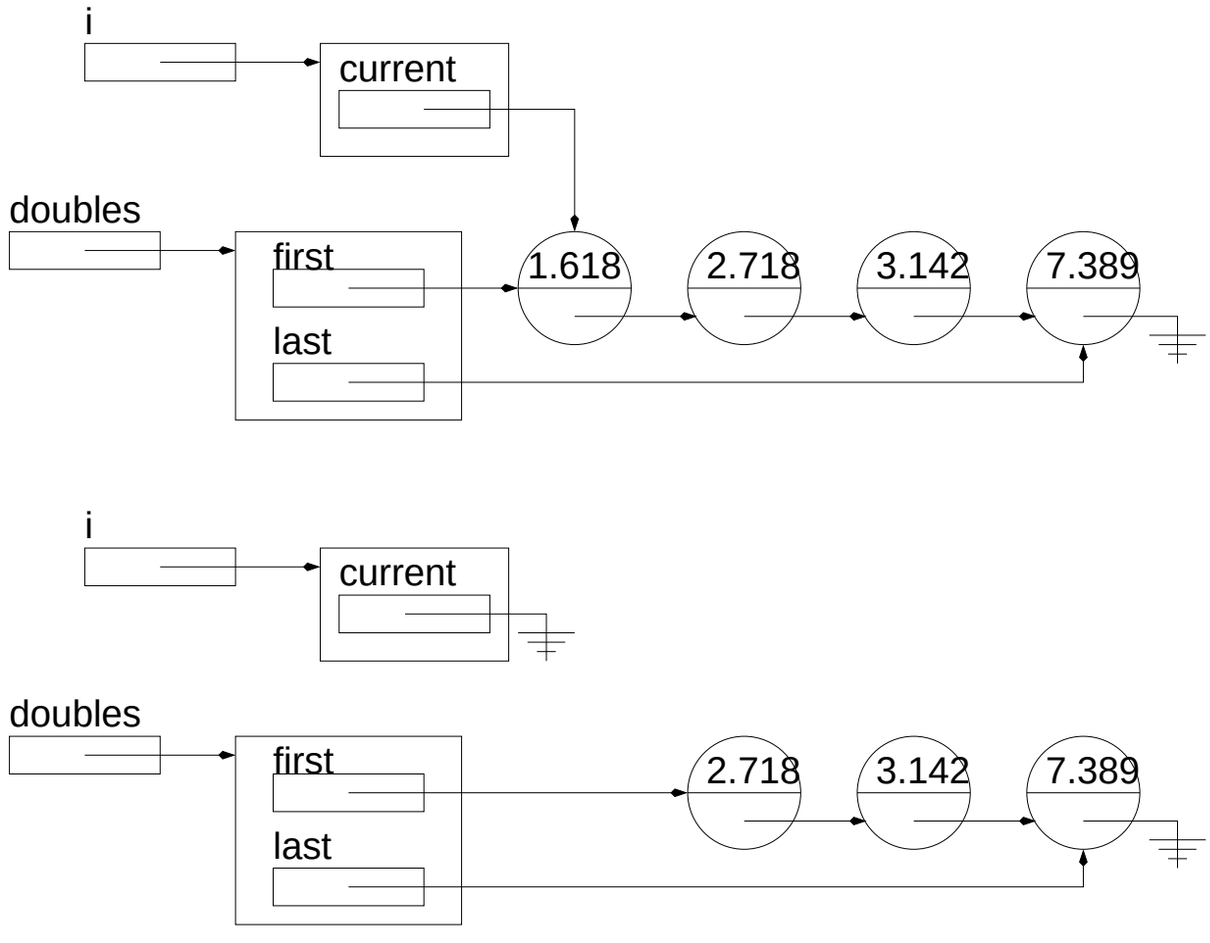
Retire le dernier élément retourné par la méthode **next()**.

Un appel à la méthode **add(value)** suivi d'un appel à méthode **remove()** garde la liste inchangée.

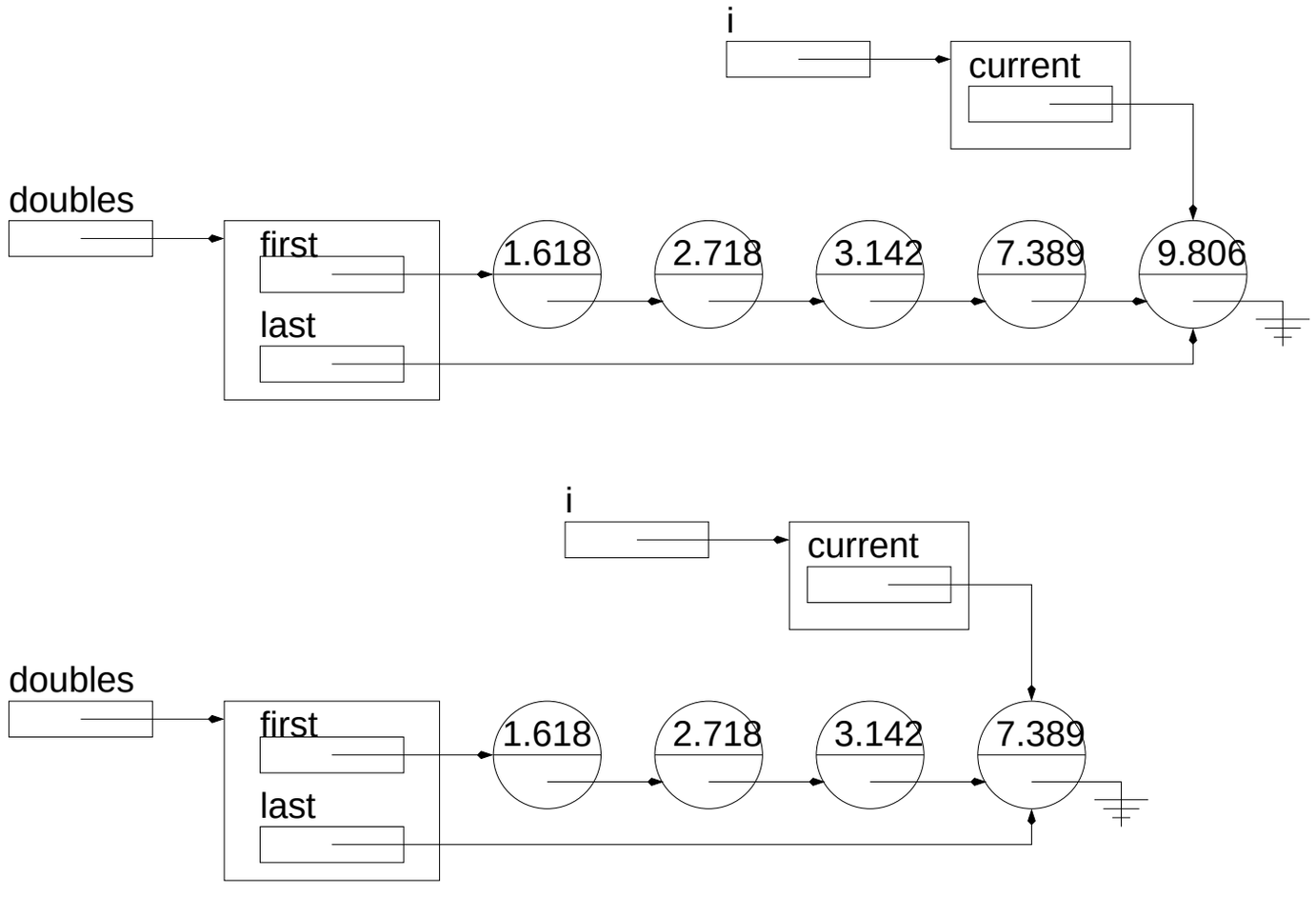
$\Rightarrow$ Notez que la méthode **remove()** ne retourne pas l'élément qu'elle retire, celui-ci a déjà été retourné par le dernier appel à la méthode **next**.



$\Rightarrow$ Retirer un élément à une position intermédiaire, n'affecte pas le prochain appel à la méthode **next()**.



$\Rightarrow$ Retirer le premier élément.



$\Rightarrow$ Retirer le dernier élément, la valeur de retour de la méthode **hasNext()** n'est pas affectée.

Iterator

```
public interface Iterator<E> {
    public abstract E next();
    public abstract boolean hasNext();
    public abstract void add( E o );
    public abstract void remove();
    public abstract boolean hasPrevious();
}
```

LinkedList

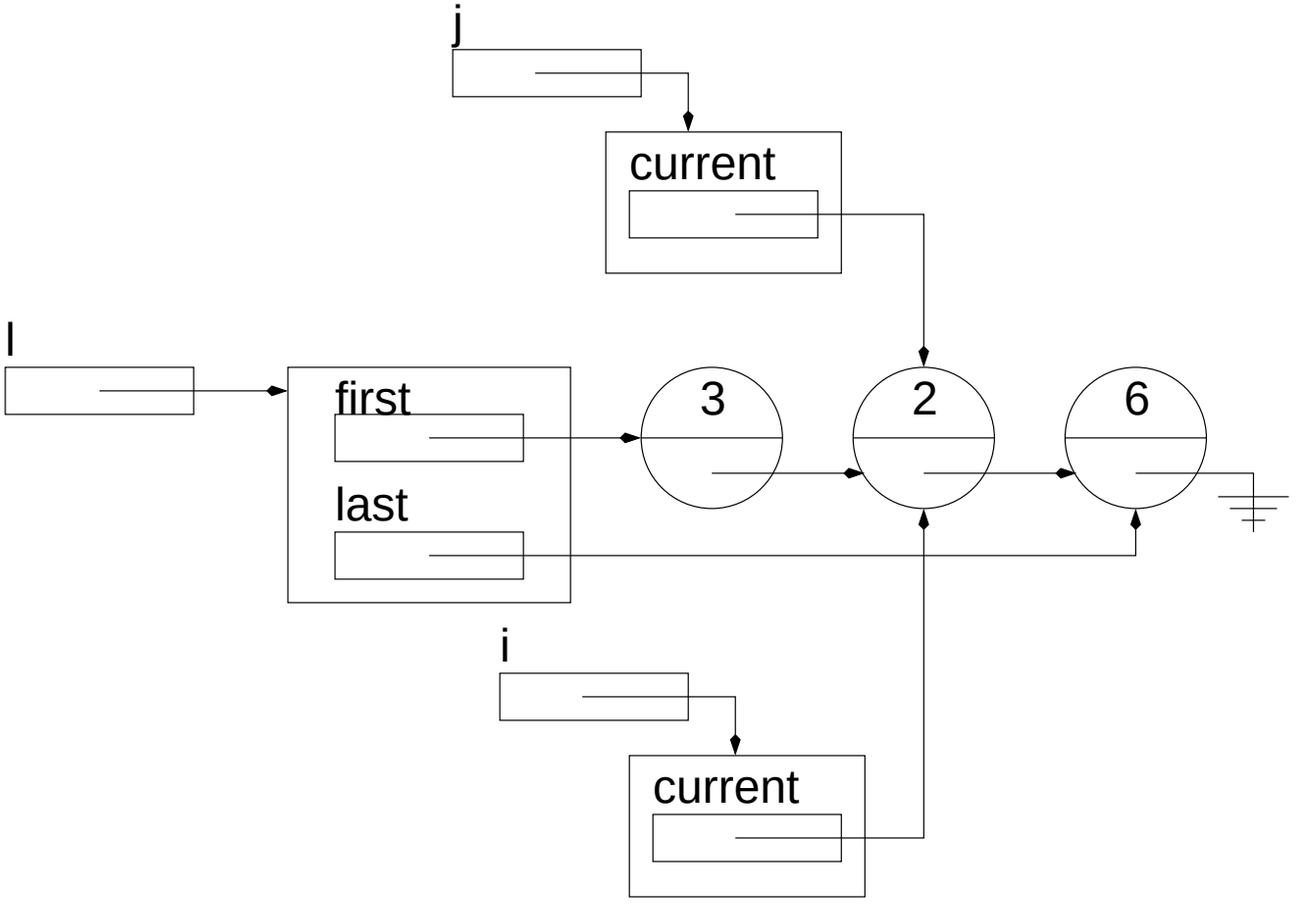
```
int size();
void add( E o );
E get( int pos );
E remove( int pos );
Iterator<E> iterator();
```

Limitations ?

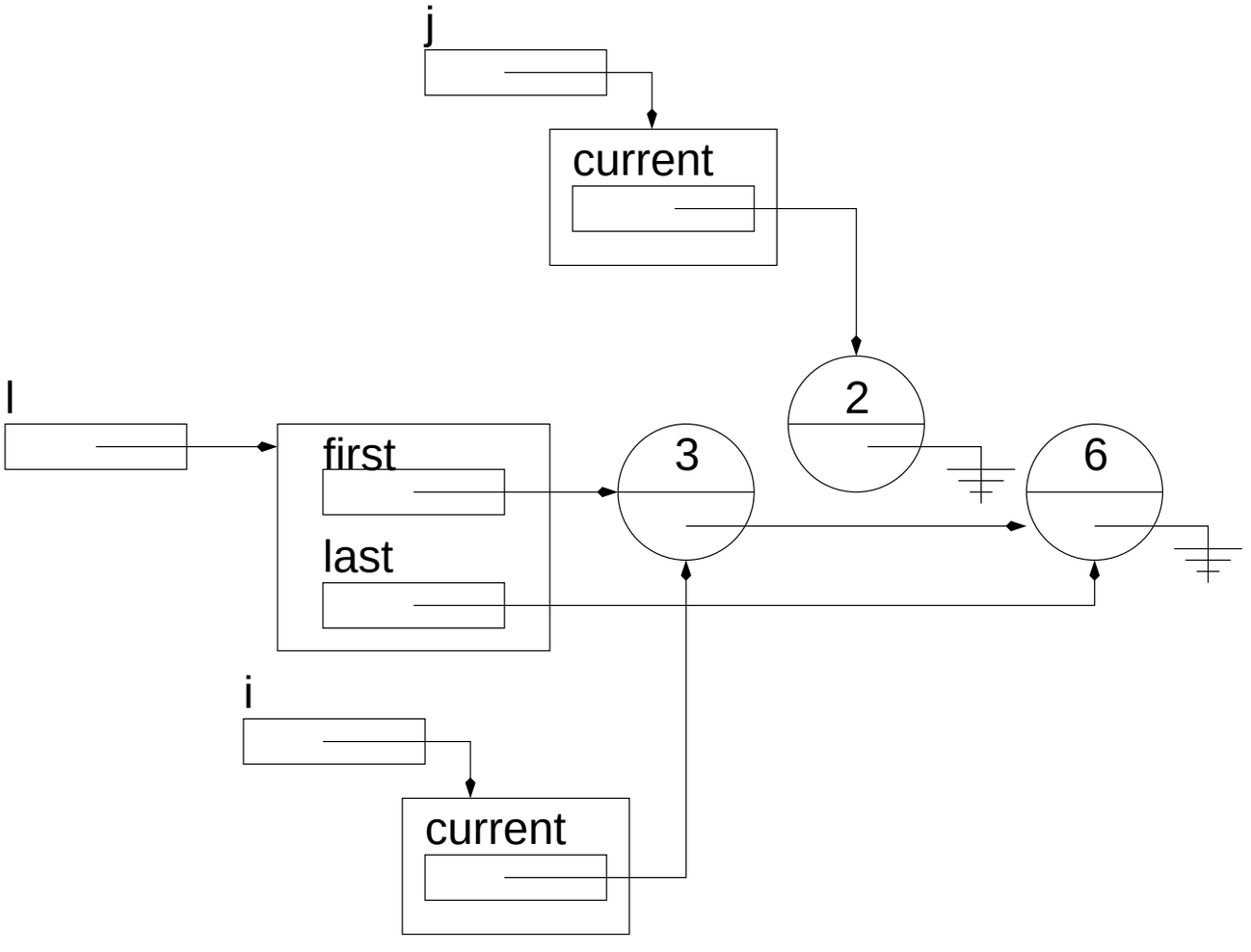
Modifications concourantes.

Avoir plusieurs itérateurs apporte sa part de problèmes. Tout va bien s'il n'y a aucune modification.

Multiples Iterateurs



$\Rightarrow$ **i.remove()**.



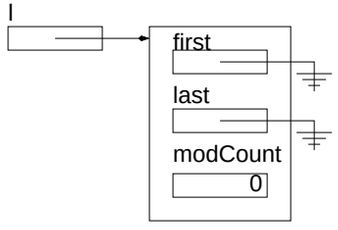
⇒ **j** est maintenant non valide ! Suggestions ?

Implémentation 3 : *fail-fast*

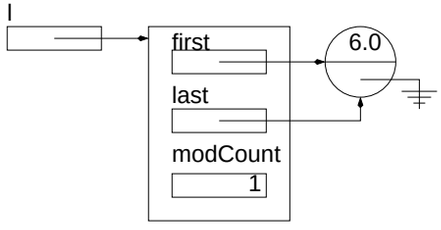
La solution adoptée ici s'appelle ≪**fail-fast**≫ et consiste à rendre un itérateur non valide, dès qu'une modification à la liste est survenue, mais n'a pas été causée par cet itérateur.

Pour implémenter cette solution :

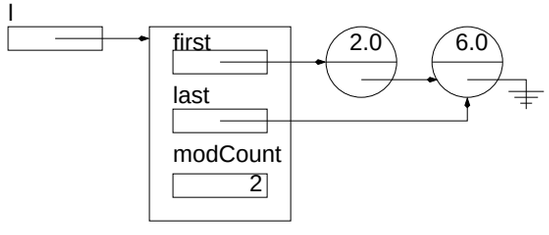
1. Nous ajoutons un compteur de modifications à l'en-tête de la liste (**modCount**) ;
2. Nous ajoutons un compteur de modifications aux itérateurs (**expModCount**) ;



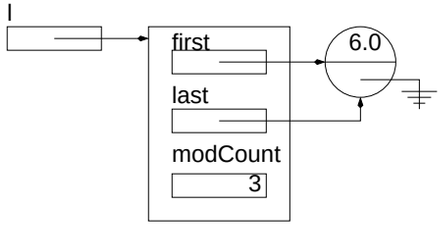
l = new LinkedList() ;



l.addFirst(6.0) ;

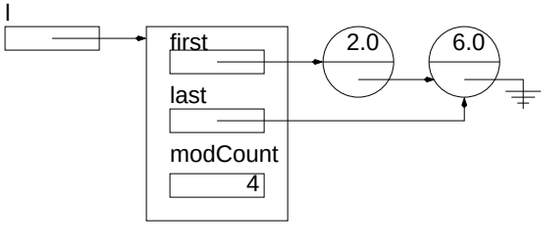


l.addFirst(2.0) ;

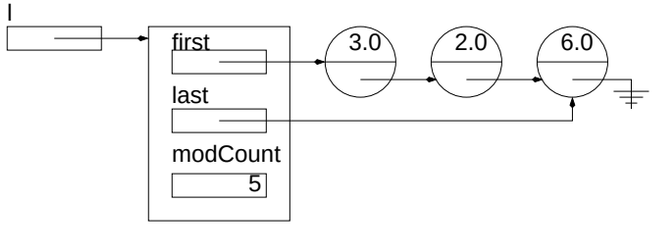


l.removeFirst() ;

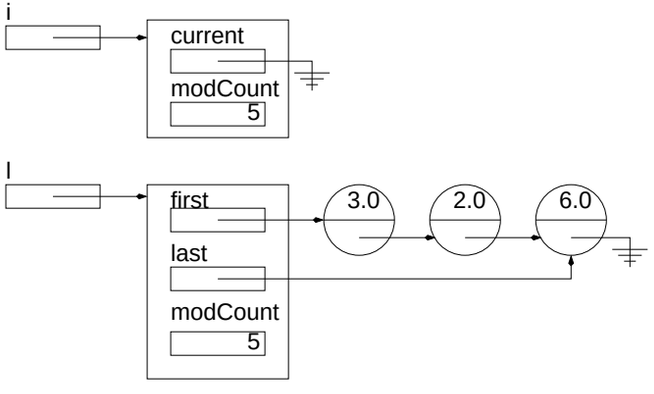
1. Lorsqu'une nouvelle liste est crée, son compteur de modifications est mis à zéro ;
2. Notez que le compteur de modifications dénombre le nombre de modifications et non le nombre d'éléments.



l.addFirst(2.0) ;

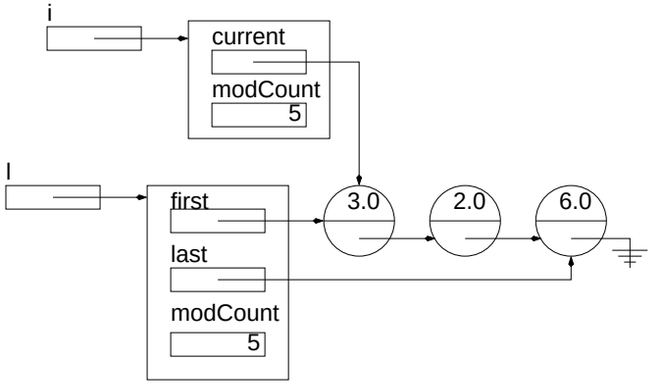


l.addFirst(3.0) ;

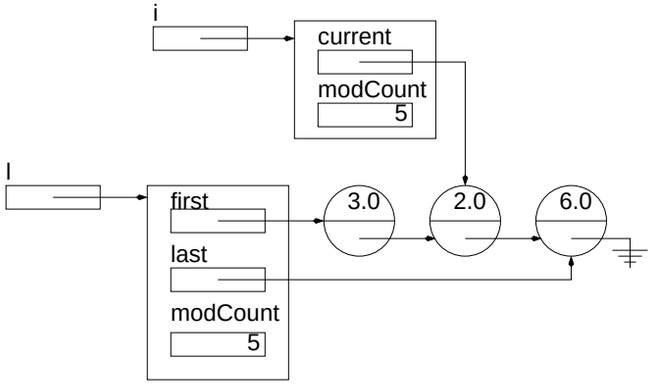


Iterator i = iterator() ;

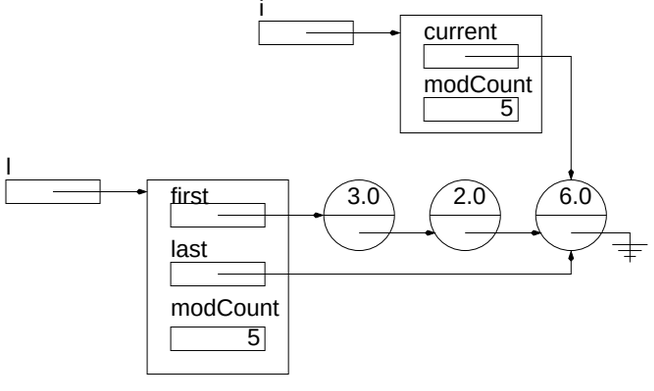
$\Rightarrow$ La création d'un nouvel itérateur n'affecte pas **modCount** !



i.next() ;

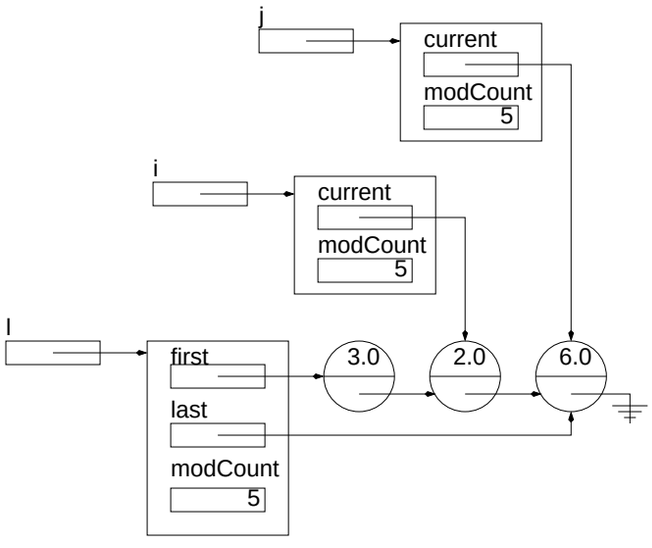


i.next() ;

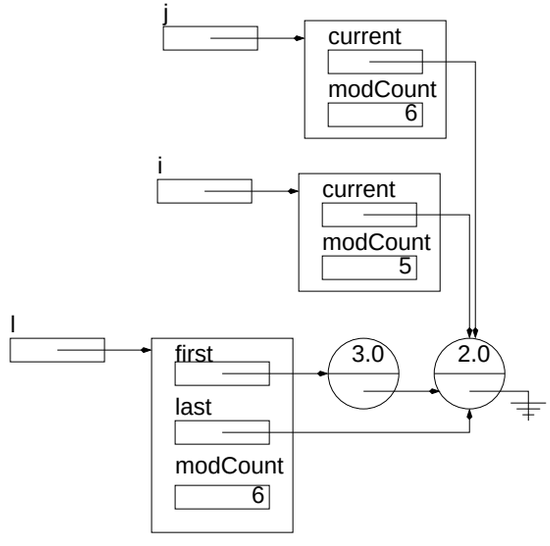


i.next() ;

$\Rightarrow$ De même lorsque la liste est traversée !



Les itérateurs sont valides



j.remove() ;

$\Rightarrow$ L'itérateur dont la valeur de la variable **expectedModCount** est la même que celle de la variable **modCount** de l'en-tête de la liste demeure valide.

Changements apportés à la classe LinkedList

Chaque fois qu'une méthode de **LinkedList** transforme la liste, elle doit mettre à jour **modCount**.

Lorsqu'une nouvelle liste est créée, **modCount** doit être mis à zéro.

Modifications de LinkedListIterator

Lorsqu'un itérateur est créé, la valeur de **modCount** est copiée dans la variable d'instance **expectedModCount**.

Toute méthode transformant la liste (ajoute, retire, etc.) doit mettre à jour **expectedModCount et modCount** !

Toutes les méthodes de l'itérateur ont une précondition qui consiste à vérifier la validité de cet itérateur.

Lorsque la précondition n'est pas satisfaire, la méthode lance une exception de type **ConcurrentModificationException**.

L'itérateur et la boucle ≪for≫

```
List<Integer> ints = Arrays.asList( 1, 2, 3, 4, 5 );

int s = 0;

for ( Iterator<Integer> it = ints.iterator(); it.hasNext(); ) {
    Integer i = it.next();
    s += i.intValue();
}

System.out.println( s );
```

L'itérateur et la boucle ≪for≫

Depuis la version 5, Java a maintenant une nouvelle forme syntaxique de la boucle ≪for≫ qui crée automagiquement un itérateur et l'utilise afin de traverser la liste, et qui s'occupe aussi, si nécessaire, de l'empaquetage et du dépaquetage des valeurs de type primitif à l'aide de classes enveloppantes.

```
List<Integer> ints = Arrays.asList( 1, 2, 3, 4, 5 );

int s = 0;
for ( int i : ints ) {
    s += i;
}

System.out.println( s );
```

L'itérateur et la boucle ≪for≫

```
List<Integer> ints = Arrays.asList( 1, 2, 3, 4, 5 );

int s = 0;
for ( int i : ints ) {
    s += i;
}

System.out.println( s );
```

Quels type s'utilisent avec cette boucle ≪for≫ ? Les types qui possède un itérateur. Quelle est la façon Java de s'en assurer ? Toute classe qui implémente l'interface **Iterable<E>**.

Iterable<E>

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

où

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

L'itérateur et la boucle ≪for≫

```
public class LinkedList<E> implements List<E>, Iterable<E> {
    private static class Node<T> { ... }
    private Node<E> head;
    private class ListIterator implements Iterator<E> {
       ...
    }
    ...
}
```

Iterable

```
public class LinkedList<E> implements List<E>, Iterable<E> {

    private static class Node<T> { ... }

    private Node<E> head;

    private class ListIterator implements Iterator<E> {
       ...
    }

    ...
}
```

Iterable

```
public interface List<E> extends Iterable<E> {
    ...
}

public class LinkedList<E> implements List<E> {

    private static class Node<T> { ... }

    private Node<E> head;

    private class ListIterator implements Iterator<E> {
       ...
    }

    ...
}
```

La nouvelle boucle ≪for≫

Finalement, cette nouvelle forme de la boucle ≪for≫ s'applique aussi aux tableaux.

```
int[] xs = new int[] { 1, 2, 3 };

int sum = 0;

for ( int x : xs ) {
    sum += x;
}
```

Appendice : Accès aux attributs et méthodes de l'objet extérieur

Nous avons donné au compteur de modification de la classe interne (**expectedModCount**) un nom différent de celui de la classe extérieure (**modCount**) afin d'éviter un conflit de nom.

Nous aurions pu utiliser la syntaxe **LinkedList.this.modCount** dans la classe interne (**ListIterator**) afin d'accéder à la variable d'instance de l'objet extérieur (**LinkedList**).

```
public class Outer {
    private int value = 99;
    public class Inner {
        private int value;
        public Inner() {
            this.value = Outer.this.value + 1;
        }
        public int getValue() {
            return value;
        }
    }
    public Inner newInner() {
        return new Inner();
    }
    public int getValue() {
        return value;
    }
}
```

Accès aux attributs et méthodes de l'objet extérieur

```
class Test {
    public static void main( String[] args ) {

        Outer o = new Outer();
        System.out.println( "o.getValue() -> " + o.getValue() );

        Outer.Inner i = o.newInner();
        System.out.println( "i.getValue() -> " + i.getValue() );

        System.out.println( "o.getValue() -> " + o.getValue() );
    }
}
```

Accès aux attributs et méthodes de l'objet extérieur

```
// > java Test
// o.getValue() -> 99
// i.getValue() -> 100
// o.getValue() -> 99
```

Exemple avancé et optionnel

```
public class A {
    private int value = 99;
    public class B {
        private int value;
        public B() { this.value = A.this.value + 1; }
        public class C {
            private int value;
            public C() {
                this.value = B.this.value + 1;
            }
            public int getValue() {
                System.out.println( "A.this.value = " + A.this.value );
                System.out.println( "B.this.value = " + B.this.value );
                System.out.println( "this.value = " + this.value );
                return value;
            }
        }
    }
}
```

```
class D {
    public static void main( String[] args ) {
        A.B.C abc = new A().new B().new C();
        System.out.println( "abc.getValue() -> " + abc.getValue() );
    }
}

// > java D
// A.this.value = 99
// B.this.value = 100
// this.value = 101
// abc.getValue() -> 101
```