# ITI 1121. Introduction to Computing II *

Guy-Vincent Jourdan
(based on Marcel Turcotte's slides)
School of Electrical Engineering and Computer Science

Version of January 4, 2018

## Abstract

- Iterator[1]

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

[1] A great opportunity to discuss time complexity (asymptotic analysis, big-O), encapsulation, object-oriented programming, interface, and more.

# Motivation

Given a (singly) linked list implementation, called **LinkedList**, of the interface **List**, defined as follows.

```
public interface List<E> {
    public abstract boolean add( E obj );
    public abstract E get( int index );
    public abstract boolean remove( E obj );
    public abstract int size();
}
```

**Consider writing statements to traverse the list. Compare writing the code inside and outside of the implementation.**

$\Rightarrow$ The difficulties would be the same for a doubly linked list.

# Traversing a list inside/outside of the class

```
List<String> colors;
colors = new LinkedList<String>();

colors.add( "bleu" );
colors.add( "blanc" );
colors.add( "rouge" );
colors.add( "jaune" );
colors.add( "vert" );
colors.add( "orange" );
```

# A — traversing a linked list inside of the class

Being inside of the implementation, the nodes can be accessed directly.

```
Node<E> p = head;
while ( p != null ) {
    System.out.println( p.value );
    p = p.next;
}
```

# B — traversing a linked list outside of the class

Traversing a list without having access to the underlying implementation requires using **E get( int pos )**.

```
for ( int i=0; i < colors.size(); i++ ) {
    System.out.println( colors.get( i ) );
}
```

$\Rightarrow$ How efficient, w.r.t. time, this code is compared to writing it within the class **LinkedList**? Much faster, faster, same, slower, much slower?

# Timing results

Here are the timing results for $20,000$, $40,000$ and $80,000$ nodes lists, averaged over 5 runs (times in milliseconds).

| # nodes | A | B |
|---|---|---|
| 20,000 | 2 | 20,644 |
| 40,000 | 5 | 94,751 |
| 80,000 | 12 | 407,059 |

$\Rightarrow$ For $80,000$ nodes it takes 6.5 minutes traversing the list using **get( pos )** vs 12 ms for the other approach. What can explain this huge difference?

```
for ( int i=0; i< names.size(); i++ ) {
    System.out.println( names.get( i ) );
}
```

| Call | Number of nodes visited |
|---|---|
| get( 0 ) | 1 |
| get( 1 ) | 2 |
| get( 2 ) | 3 |
| get( 3 ) | 4 |
| ... | ... |
| get( n-1 ) | n |

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \approx n^2$$

Implementation A visits $n$ nodes, implemenation B visits $n^2$ nodes!

| # nodes | A | B |
|---|---|---|
| 20,000 | 2 | 20,644 |
| 40,000 | 5 | 94,751 |
| 80,000 | 12 | 407,059 |

Executing the following statements (implementation B).

```
for ( int i=0; i< names.size(); i++ ) {
    System.out.println( names.get( i ) );
}
```

It is as if implementation A had been written like this.

```
for ( int i=0; i< size(); i++ ) {
    Node p = head;
    for ( int j=0; j<i; j++ ) {
        p = p.next;
    }
    System.out.println( p.value );
}
```

Instead of this.

```
Node p = head;
while ( p != null ) {
    System.out.println( p.value );
    p = p.next;
}
```

What mechanism could the implementation provide that would allow to efficiently traverse the list?

We are outside of the class definition, therefore, we do not have access to the implementation (**p.next** and such), yet the need to traverse a data structure is frequent.

(!) The concept that we are introducing today will apply to a specific context, TRAVERSING A LIST.

WE WILL NOT PROVIDE A GENERAL MECHANISM TO MAKE **get( i )** EFFICIENT ALWAYS.

# Iterator: introduction

- A uniform mechanism to traverse a variety of data structures, such as lists, but also trees and other data structures to be seen in CSI 2114;
- Gives access to all the elements, one a time.
- It's a mechanism that is used by Java's own collections.

# Concept

```
‘‘create an iterator’’ (* positioned before the start of the list *)

while ( ‘‘has more elements?’’ ) {
  ‘‘move the iterator forward and return the value’’
  ‘‘process the value’’
}
```

# Concept

# Iterator interface

The implementation of the **Iterator** is separated from its usage by creating an interface for it.

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

```
/**
 * An iterator for lists that allows the programmer to traverse the
 * list.
 */
public interface Iterator<E> {
    /**
     * Returns the next element in the list.  This method may be called
     * repeatedly to iterate through the list.
     *
     * @return the next element in the list.
     * @exception NoSuchElementException if the iteration has no next element.
     */

    public abstract E next();

    /**
     * Returns <tt>true</tt> if this list iterator has more elements when
     * traversing the list in the forward direction. (In other words, returns
     * <tt>true</tt> if <tt>next</tt> would return an element rather than
     * throwing an exception.)
     *
     * @return <tt>true</tt> if the list iterator has more elements when
     *          traversing the list in the forward direction.
     */

    public abstract boolean hasNext();
}
```

# Issues

- Which class will be implementing the interface?
- How to create and initialize a new iterator?
- How to move the iterator forward?
- How to detect the end?

$\Rightarrow$ This process is modeled on traversing an array and should be familiar.

# Implementation -1-

In our first implementation, the class **LinkedList** developed earlier is modified to implement the interface **Iterator**. This involves modifying the header of the class, adding "implements Iterator", and to provide an implementation for each of the methods defined in the interface.

```
public class LinkedList<E> implements List<E>, Iterator<E> {
   private static class Node<E> { ... }
   private Node head<E>;
   public E next() { ... }
   public boolean hasNext() { ... }
   public Iterator iterator() { ... }
   // ...
}
```

# Usage

```
LinkedList l = new LinkedList();

// Operations that add elements to the list ...

// Position the iterator to the left of the list

Iterator i = l.iterator();
while ( i.hasNext() ) {
    Object o = i.next();
    // do something with o
}
```

# Remarks

1. Why is the iterator positioned to the left of the list and not on the first element?
   - To correctly process the case of the empty list; positioning the iterator onto the first element implies there will always be a first element.
2. Functional definition of "**next()**":
   (a) **Iterator moves one position forward**;
   (b) **then returns the value of the current element**.
3. A danger awaits you when calling "**next**", what is it?
   - Look at the functional definition of next, the iterator moves forward one position then returns a value, it's possible that there is no next element!
   - Give two examples where that would be the case:
     – The empty list;
     – The iterator was already positioned on the last element of the list.

```
LinkedList l = new LinkedList();
Iterator i = l.iterator();
while ( i.hasNext() ) {
    Object o = i.next();
    // ...
}
```

- Is the empty list case handled correctly?
  - Yes. The body of the loop is never executed when the list is empty, and therefore no call to next() is made.
- Is the end of the list handled correctly?
  - Yes. Consider a singleton list. Initially, the iterator is positioned to the left of the list, **hasNext()** is true, and the execution of the body of the loop can proceed. The method **next()** is called, which moves the iterator one position forward and returns the value of the current element. The remaining statements of the body of the loop are executed. The end-test is checked again, this time **hasNext()** returns false and the execution resumes with the statement that follows the loop.
    Notice how each call to **next()** is preceded by a call to **hasNext()**.

# Example: sum of the elements of a list

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

        LinkedList<Integer> l = new LinkedList<Integer>();
        for ( int i=0; i<5; i++ ) {
            l.add( new Integer( i ) );
        }
        int sum = 0;

        Iterator<Integer> i = l.iterator();
        while ( i.hasNext() ) {
            Integer v = i.next();
            sum += v.intValue();
        }
        System.out.println( "sum = " + sum );
    }
}
```

# Implementation -1- (contd)

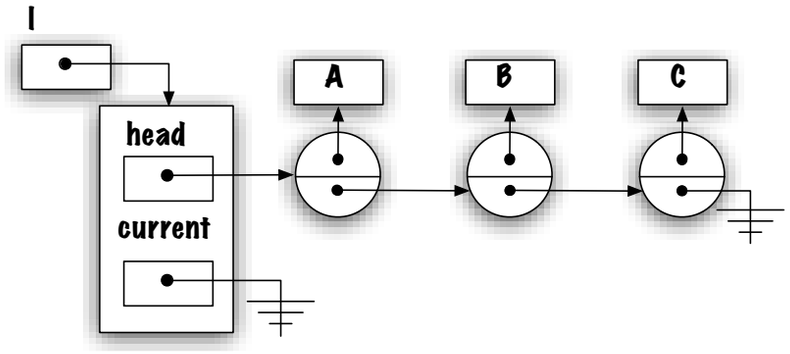
What are the necessary changes to the class **LinkedList** in order to implement the interface **Iterator**?

Let's add an instance variable called **current** to **LinkedList**.

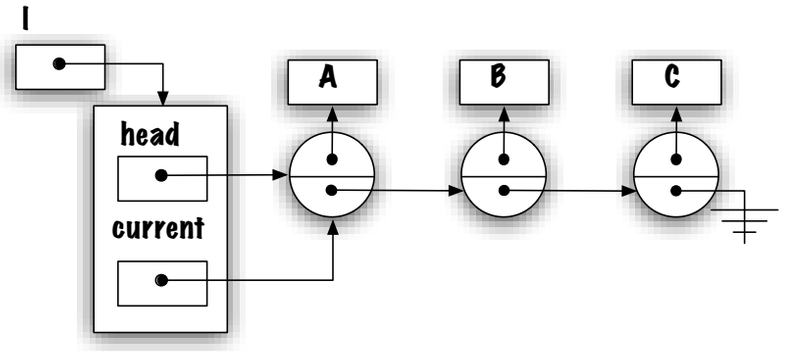
We'll use the value **null** to denote the situation where the iterator is positioned to the left of the list.

The first time **next()** is called, we'll move **current** to the first node and return its value.

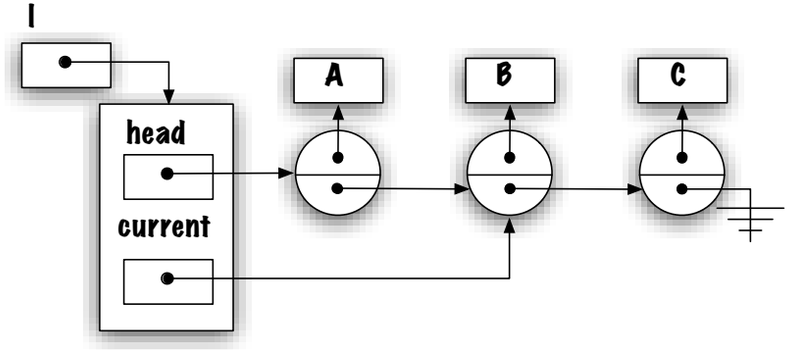
For each successive call, **current** is moved one position forward and the value found at that node is returned.



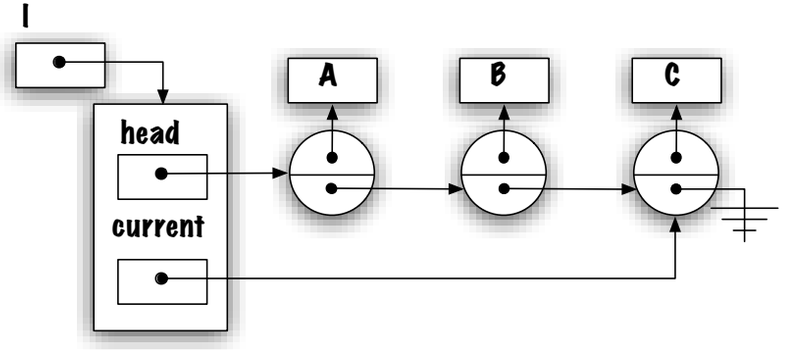

Iterator i = l.iterator();




after i.next();




after i.next();




after i.next();

Implementing iterator().

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

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

    private Node<E> head;
    private Node<E> tail;
    private Node<E> current; // <---

    public Iterator<E> iterator() {
        current = null;
        return this;
    }
}
```

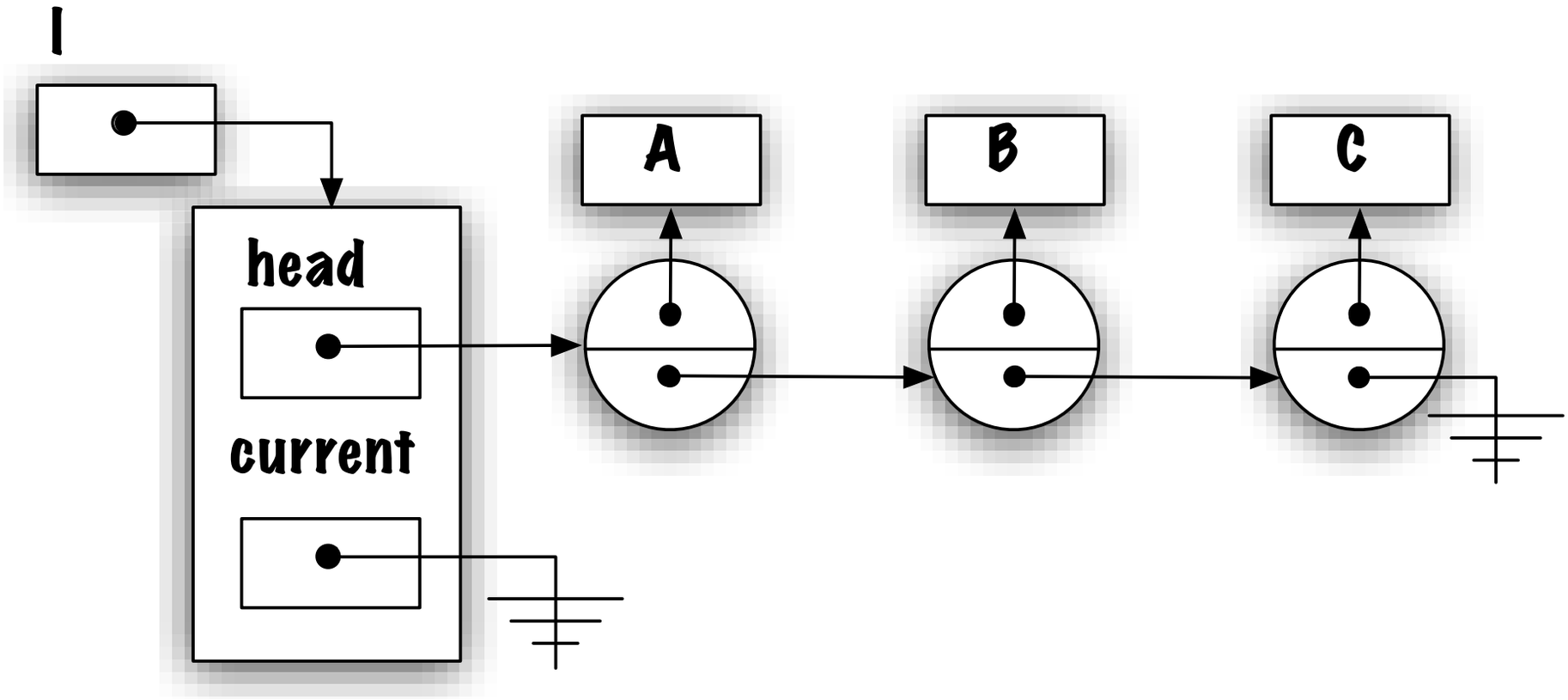
But do we need **iterator()**? Couldn't we simply assign *null* in the constructor of the list?

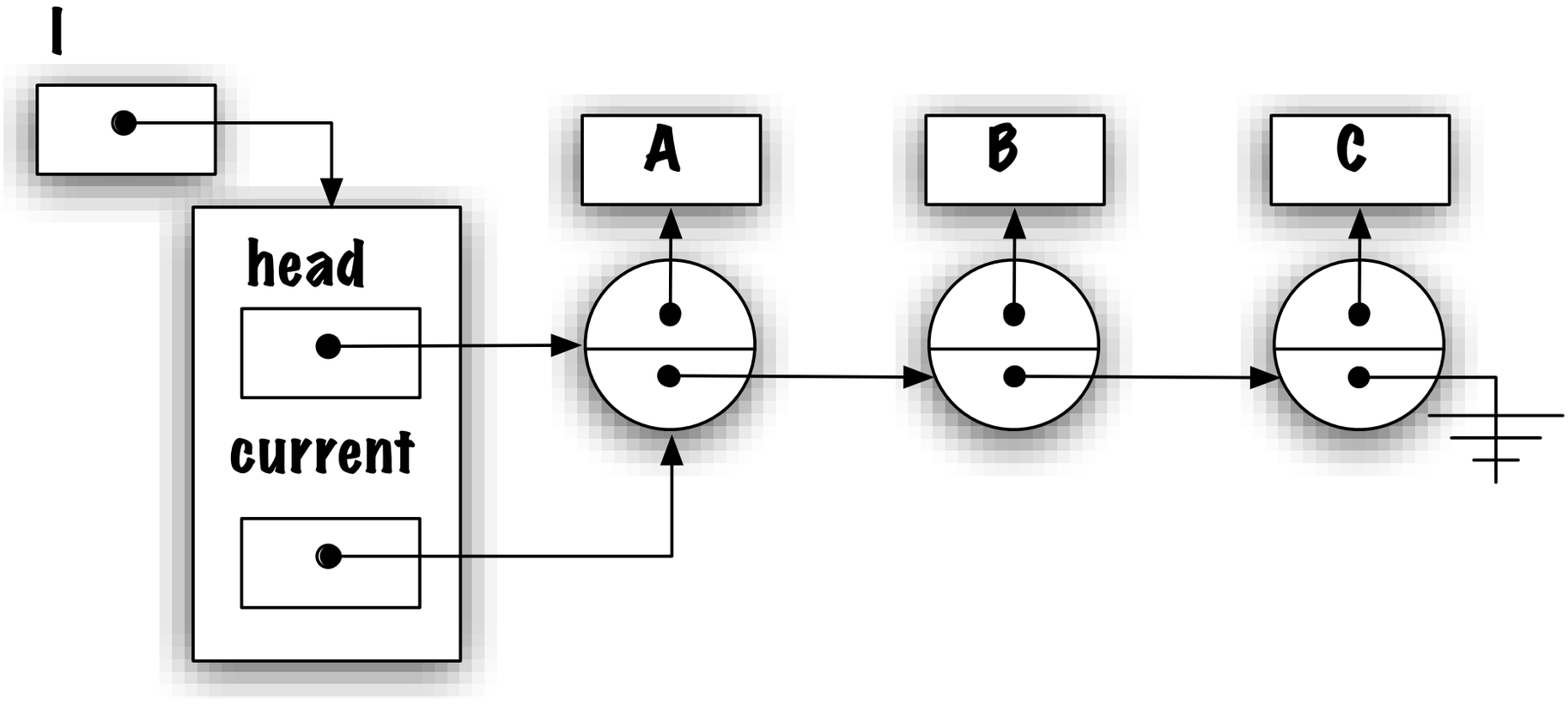
No, the method is necessary to traverse the list more than one time.

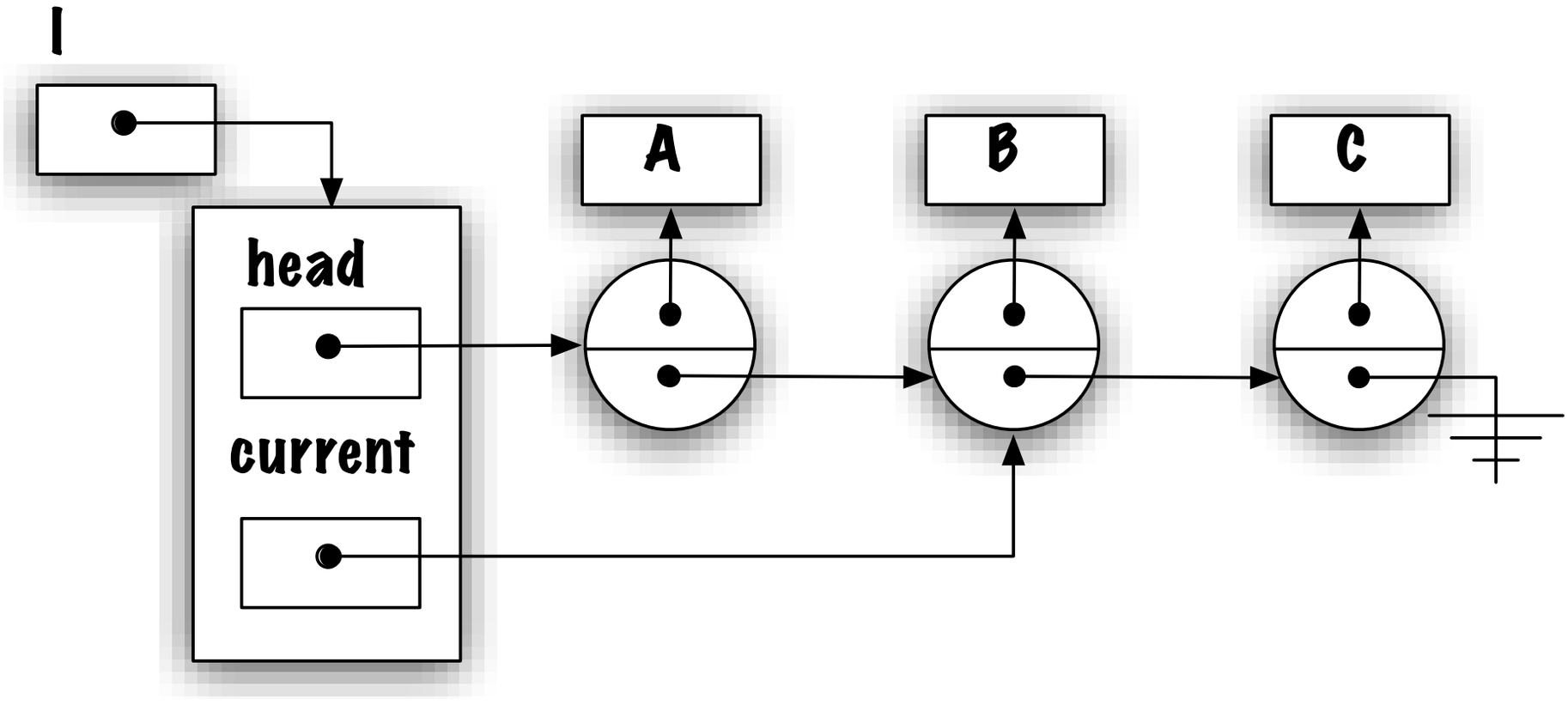
```
int sum = 0; int n = 0;
Iterator<Integer> i = l.iterator();
while ( i.hasNext() ) {
    Integer v = i.next();
    sum += v.intValue();
    n++;
}

i = l.iterator();
while ( i.hasNext() ) {
    Integer v = i.next();
    if ( v.intValue() > ( sum / n ) ) {
        System.out.println( "is greater than average" );
    }
}
```

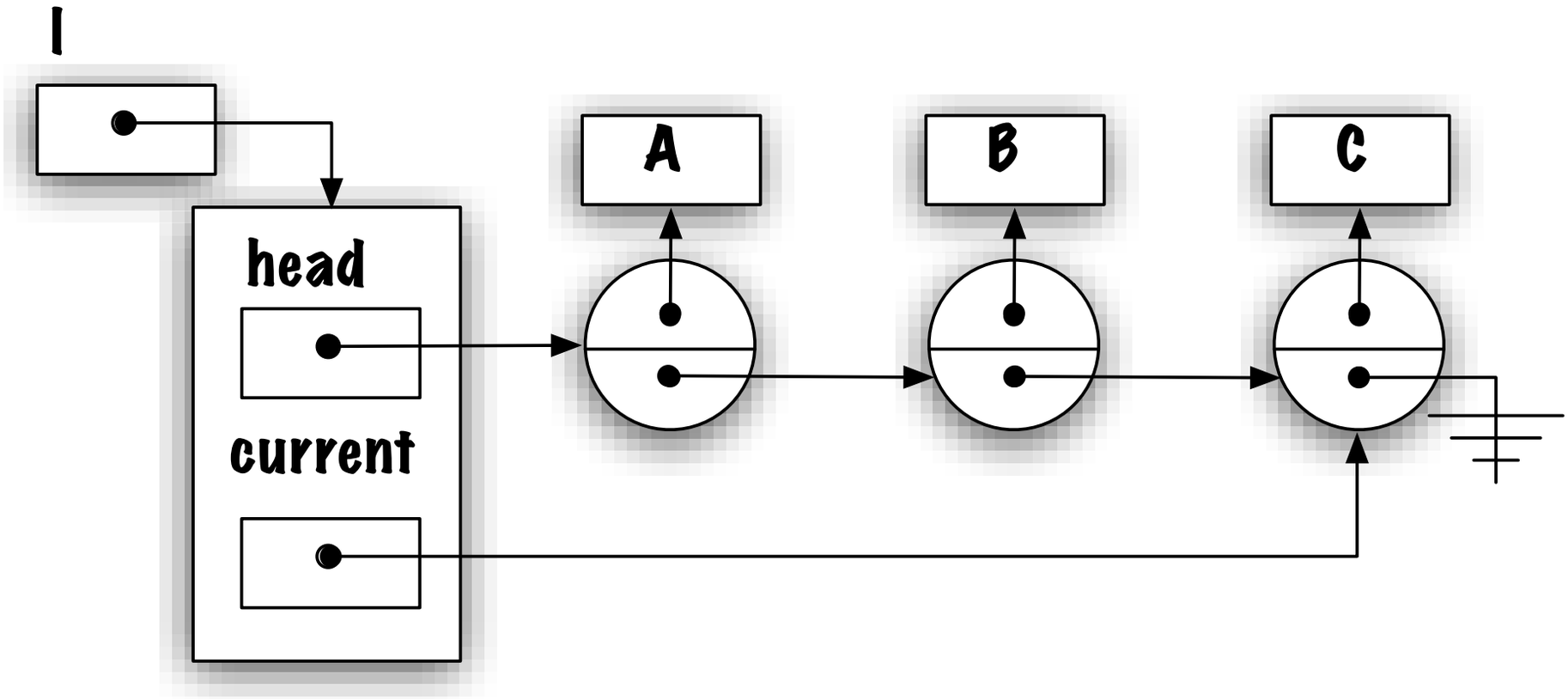
Implementing **next**.

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


l
head
current
A
B
C


l
head
current
A
B
C


l
head
current
A
B
C


l
head
current
A
B
C

# Current limitation

Our current implementation does not allow for multiple iterators!

```
while ( i.hasNext() ) {
    oi = i.next();
    while ( j.hasNext() ) {
        oj = i.next();
        // process oi and oj
    }
}
```

# What's needed then?

- We need as many references (variables) as iterators;
- **An iterator has to have access to the implementation (Node)**;
- **An iterator needs access to the instance variables of the class LinkedList**.

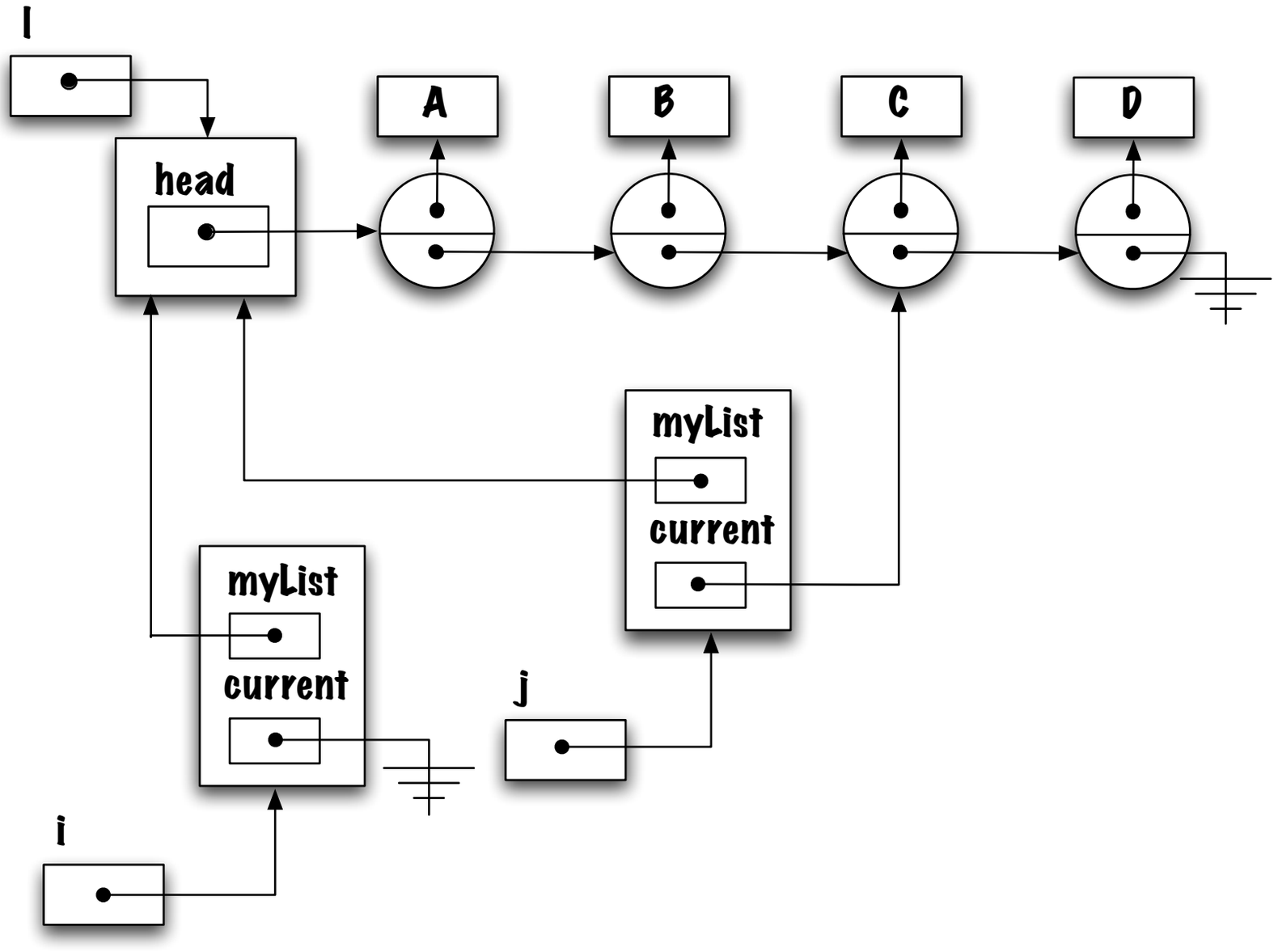
# Allowing for multiple iterators

$\Rightarrow$ Suggestions?

# Implementation -1.5-

Let's create the following top-level class (in the same package as **LinkedList**), this will require that i) **Node** is also a top-level class and ii) the variable **head** should be **protected**.

```
class LinkedListIterator<E> implements Iterator<E> {

    private Node<E> current;
    private LinkedList<E> myList;

    LinkedListIterator( LinkedList<E> myList ) {
        this.myList = myList;
        current = null;
    }
    public boolean hasNext() {
        return ( ( current == null && myList.head != null ) ||
                 ( current != null && current.next != null) );
    }
    public E next() {
        if ( current == null ) {
            current = myList.head;
        } else {
            current = current.next;
        }
        return current.value;
    }
}
```


l
head
A
B
C
D
myList
current
myList
current
j
i

The method **iterator()** of the class **LinkedList** is defined as follows.

```
public Iterator<E> iterator() {
    return new LinkedListIterator<E>( this );
}
```

# Implementation -1.75-

**LinkedListIterator** is now a static nested class of **LinkedList**; which allows us to change the visibility of the variable **head** back to **private**.

```
private static class LinkedListIterator<E> implements Iterator<E> {
    private Node<E> current;
    private LinkedList<E> myList;

    private LinkedListIterator( LinkedList<E> myList ) {
        this.myList = myList;
        current = null;
    }
    public boolean hasNext() {
        return ( ( current == null && myList.head != null ) ||
                 ( current != null && current.next != null ) );
    }
    public E next() {
        if ( current == null ) {
            current = myList.head;
        } else {
            current = current.next;
        }
        return current.value;
    }
} // end of LinkedListIterator
```

```
public Iterator<E> iterator() {
    return new LinkedListIterator( this );
}
```

# Implementation -2-: using an inner class

We'll use an inner class to create the iterators:

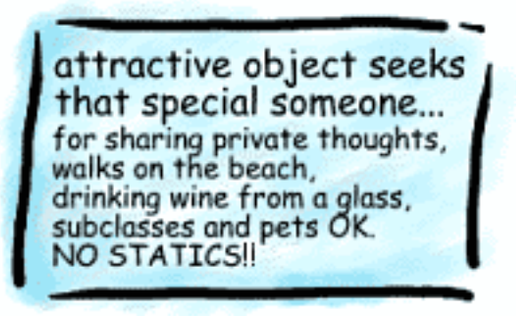
- An instance of an inner class cannot exist on its own, it requires an instance of the outer class (the converse is not true);
- **An instance of an inner class has access to the instance variables (and methods) of the (outer) object that created it;**

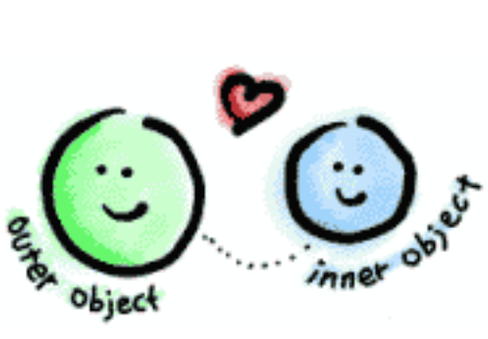
```
class Outer {
  class Inner {

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

$\Rightarrow$ think about instance variables and methods.

# Getting in Touch with your Inner Class (web resource)

www.javaranch.com/campfire/StoryInner.jsp




# Inner class

Let's call a non-static nested class an **inner** class.

- An instance of an inner class cannot exist on its own, it requires an instance of the outer class (but an instance of the outer class does not necessarily require an instance of an inner class);

- **An instance of an inner class has access to the instance variables (and methods) of the (outer) object that created it;**

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

$\Rightarrow$ Think about instance variables and methods.

```
public class Outer {
    class Inner {
        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** simply prints "* new instance of Outer class *", **i.e an instance of the Outer class can exist without the need to create an instance of the inner class** (this is not automatic).

```
public class Outer {
    class Inner {
        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 ();
                  ^
```

⇒ **An instance of an inner class can only be created by an instance of the outer class!**

```
public class Outer {
    class Inner {
        Inner() {
            System.out.println( "* new instance of Inner class *" );
        }
    }
    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 instance variable of the outer object
```

**An instance of an inner class has access to the variables and methods of its outer class!**

There are other subtle details about inner classes but our use will be limited to what has been described in those slides.

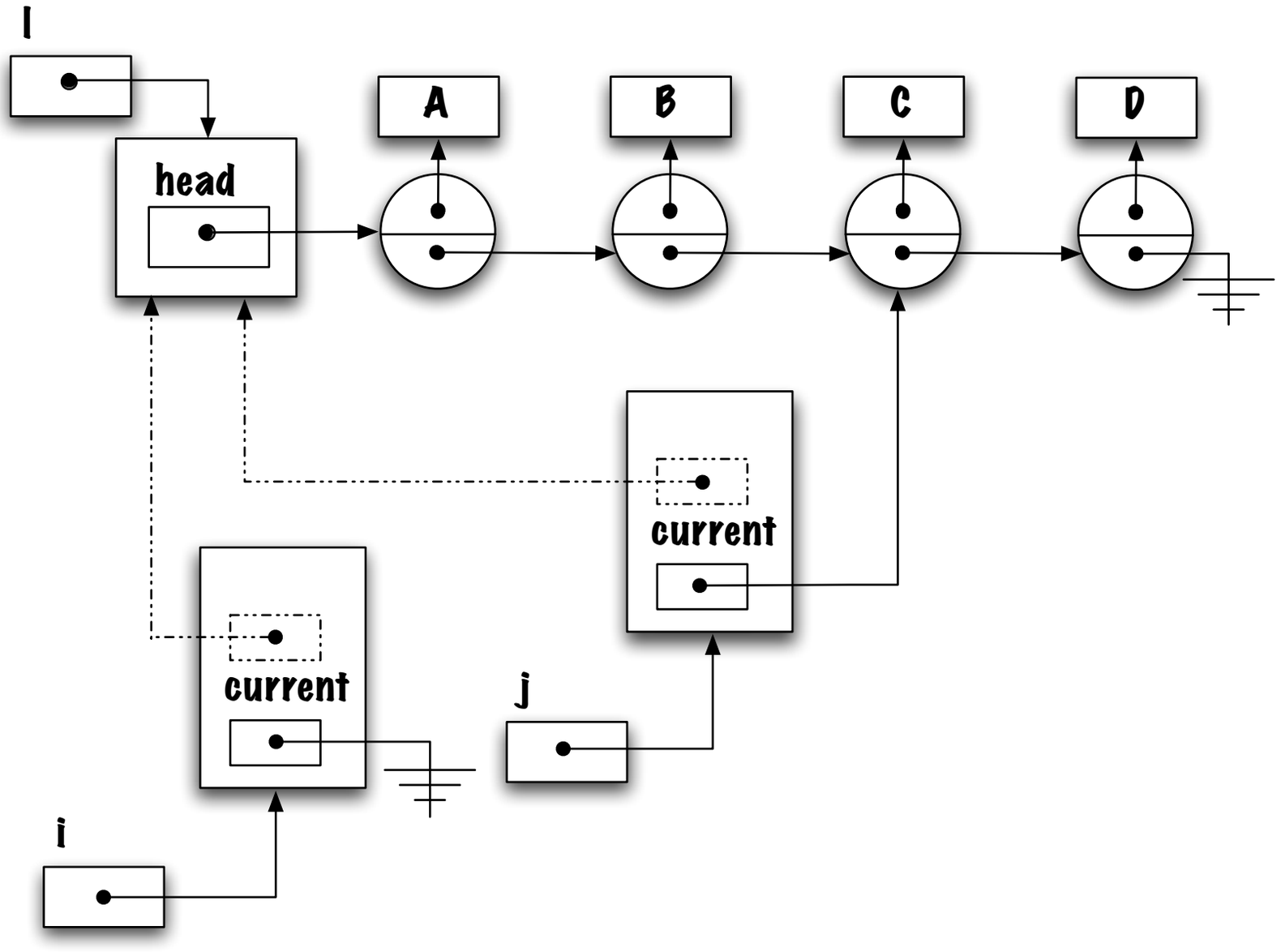
**A static nested class needs to make the relationship inner/outer class explicit!**

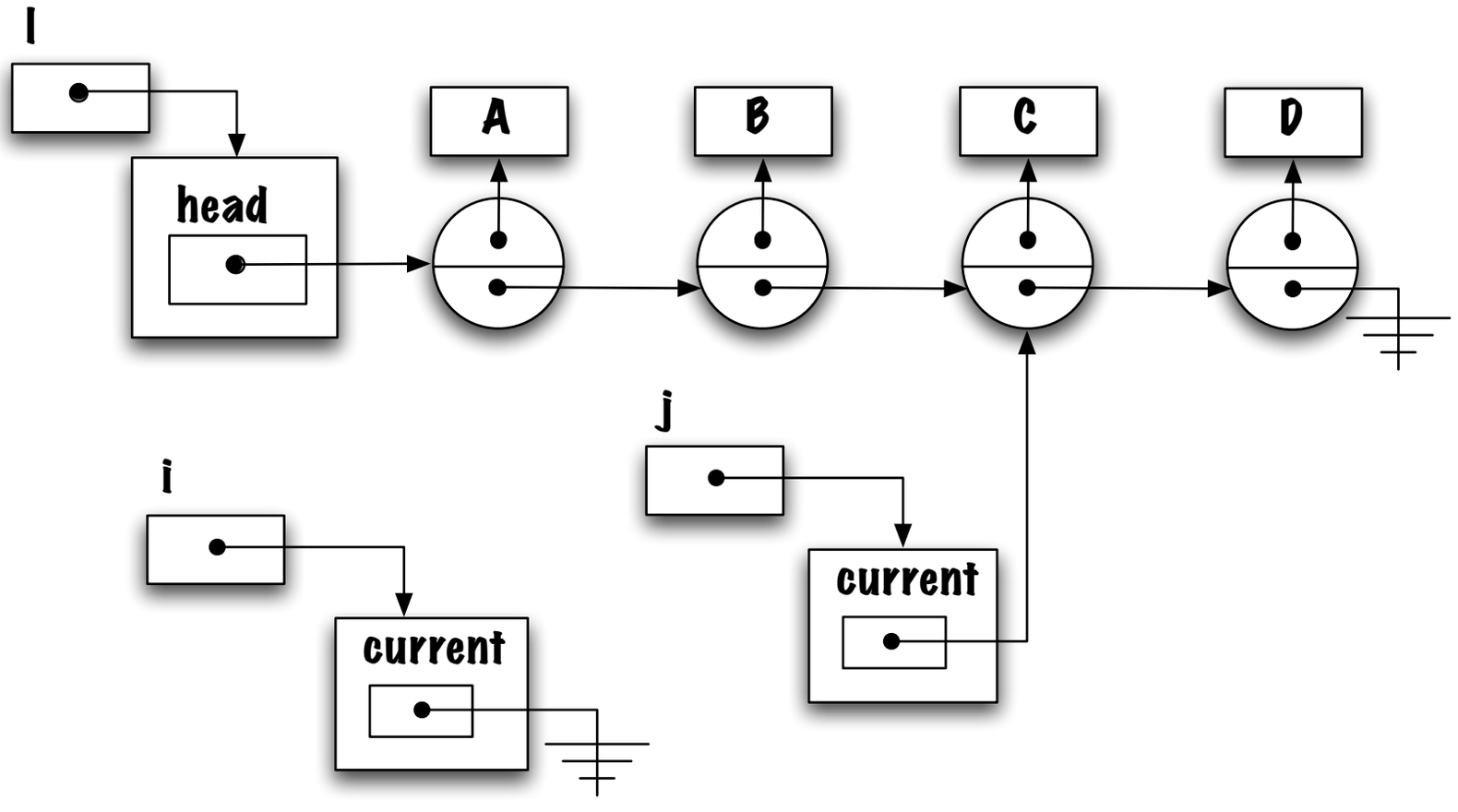
```
public class Outer {
    private static class Inner {
        private Outer parent; // <--
        private Inner( Outer parent ) {
            this.parent = parent;  // <--
            System.out.println( "* new instance of Inner class *" );
            System.out.println( "value = " + parent.value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

# Inner class: summary

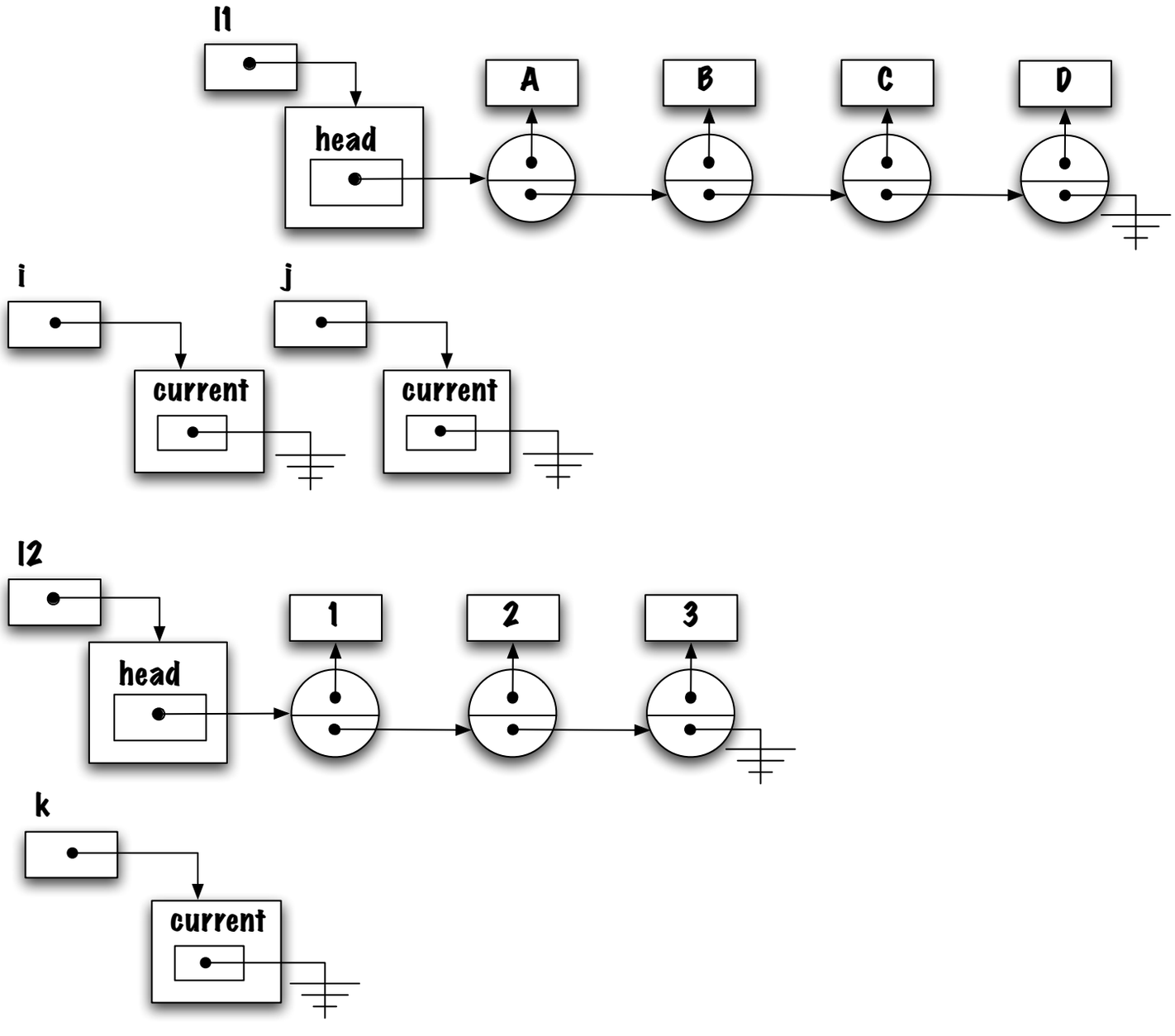
- "An *inner class* is a nested class that is not explicitely or implicitely declared static." The Java Language Specification;

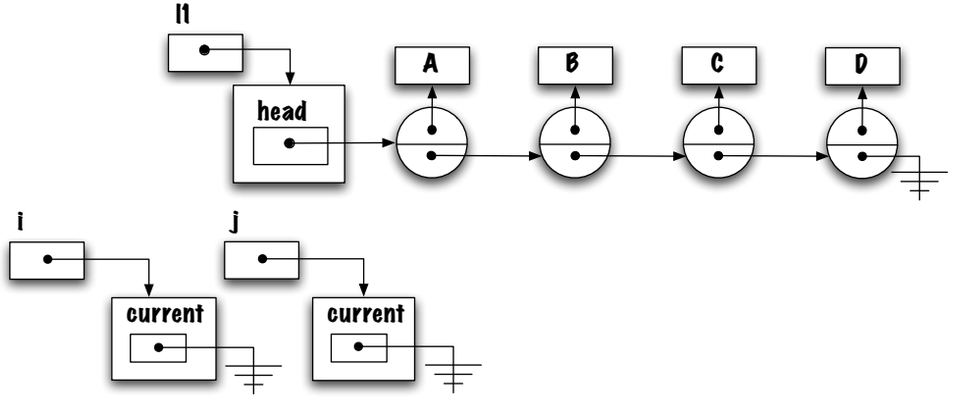
- An object of the outer class creates an instance of an inner class:
  - An object of an inner class has **one** outer object;
  - An object of the outer class may have **zero**, **one** or **several** instances of its inner class;
  - **An object of an inner class has access to the instance variables and methods of its outer object**.

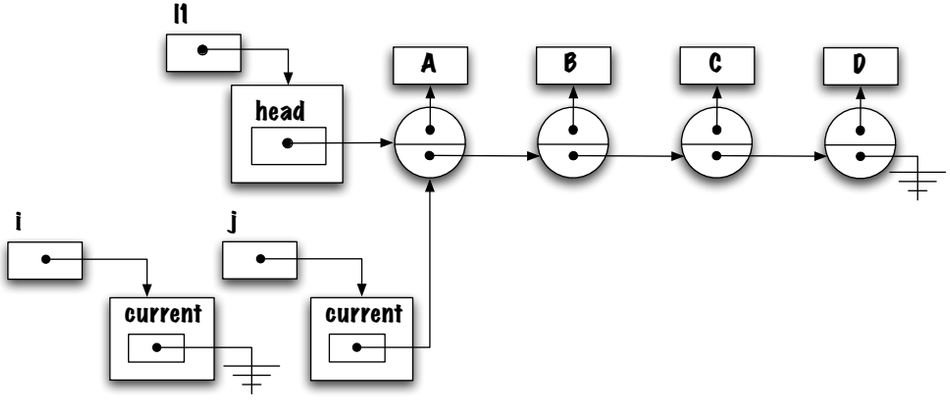

l
head
A
B
C
D
current
current
j
i

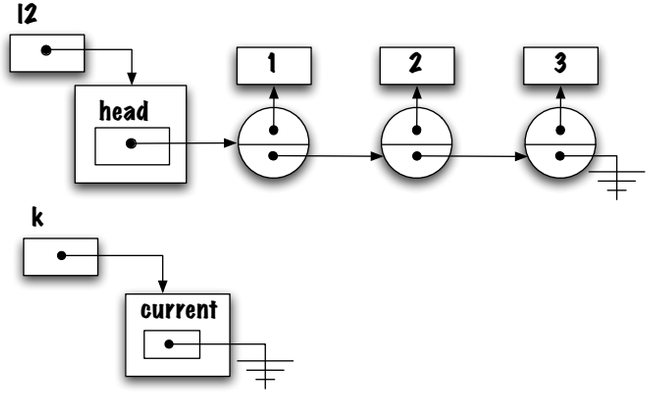

l
head
A
B
C
D
i
current
j
current

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

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

Iterator<String> i, j;
Iterator<Integer> k;

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

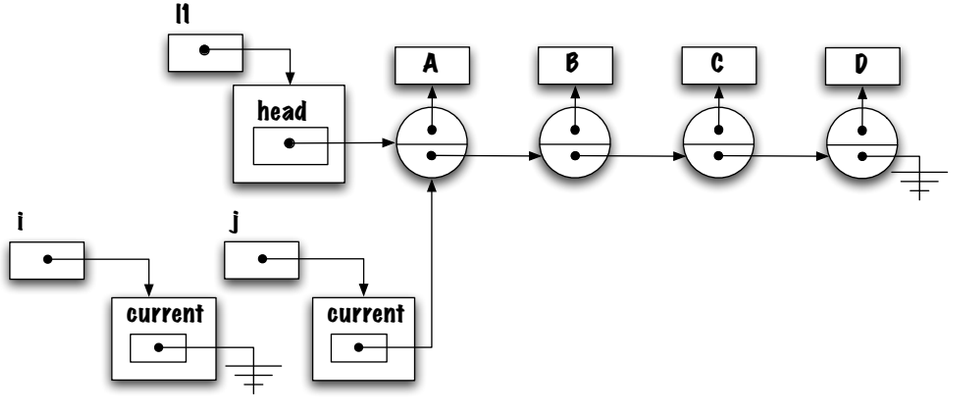

l1
head
A
B
C
D
i
current
j
current
l2
head
1
2
3
k
current

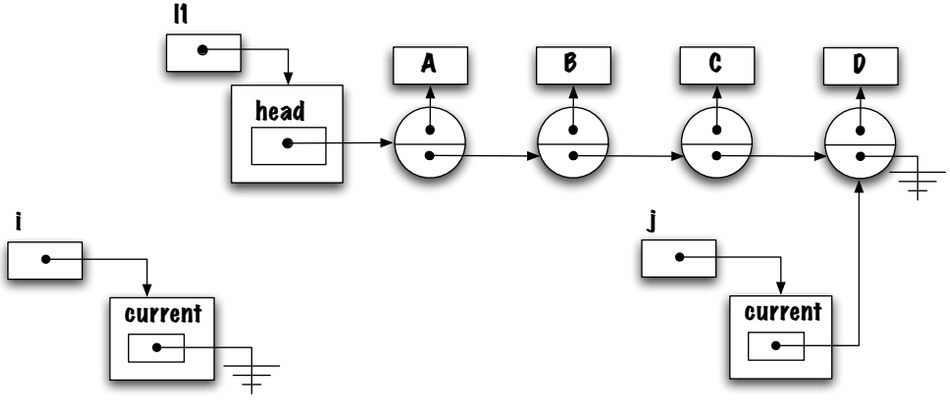


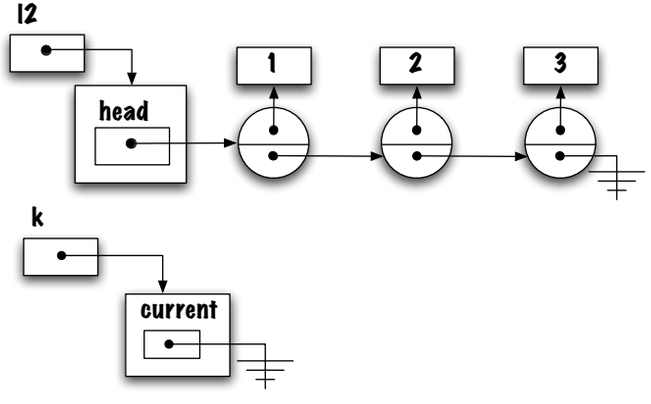



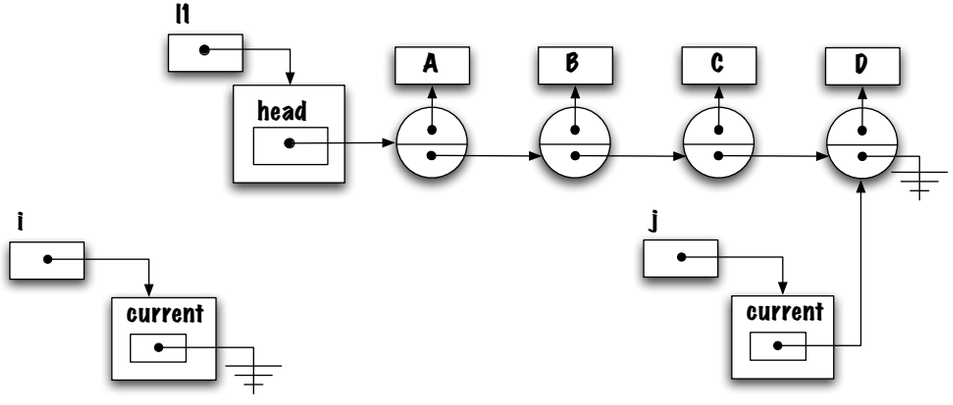

```
if ( j.hasNext() ) {
    String o = j.next();
}
```

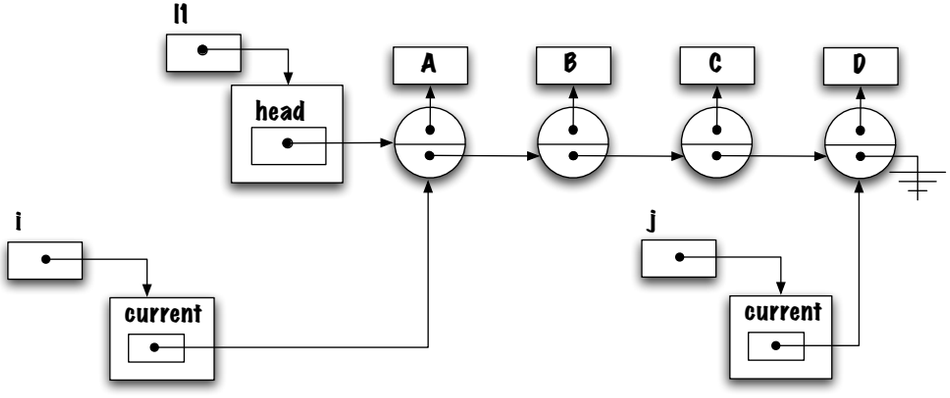


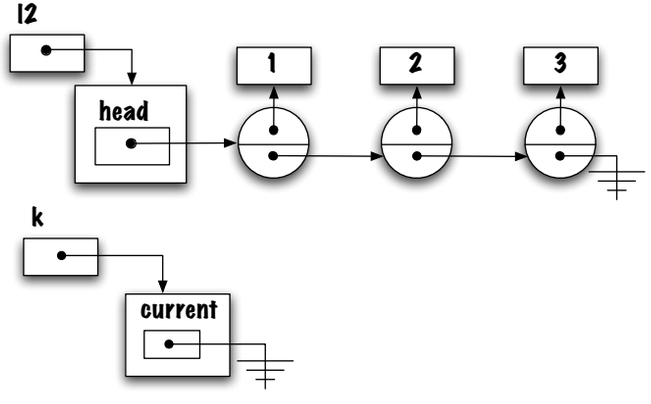



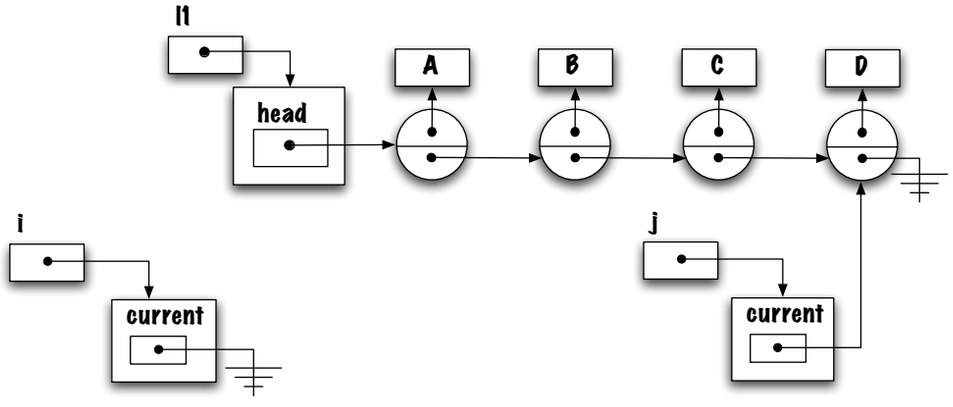

```
while ( j.hasNext() ) {
    String o = j.next();
}
```

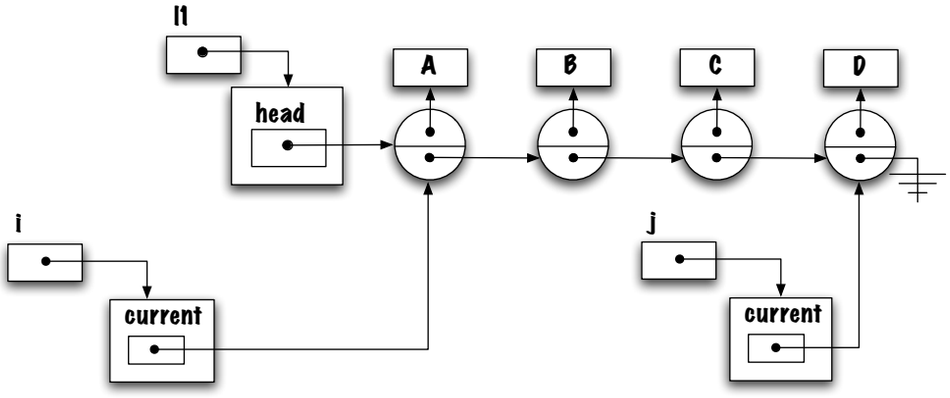


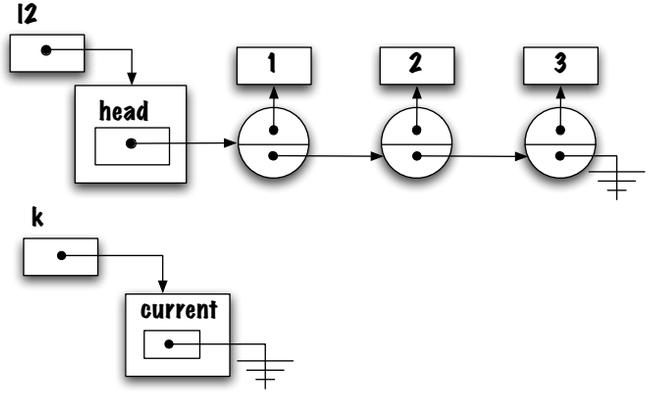



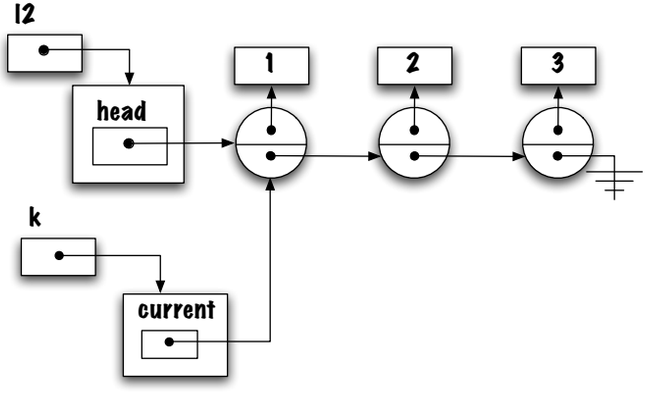




```
if ( i.hasNext() ) {
    String o = i.next();
}
```

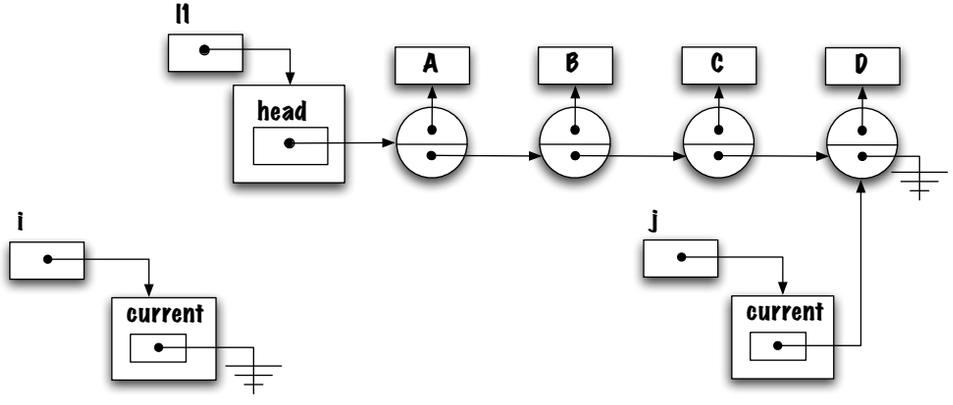


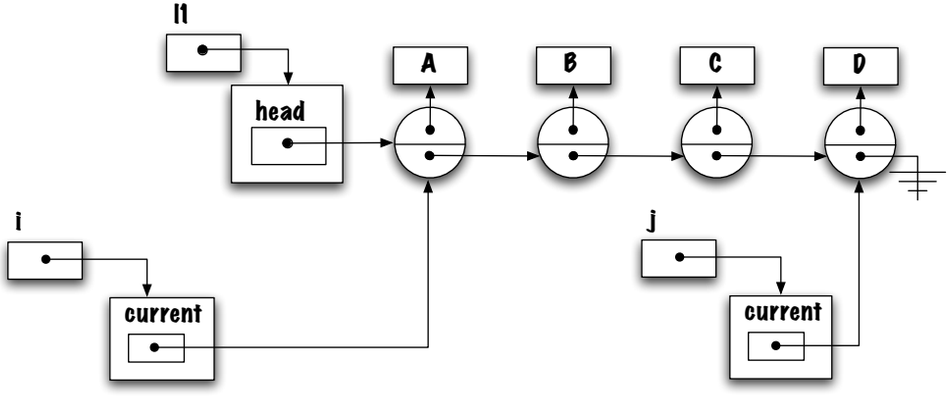



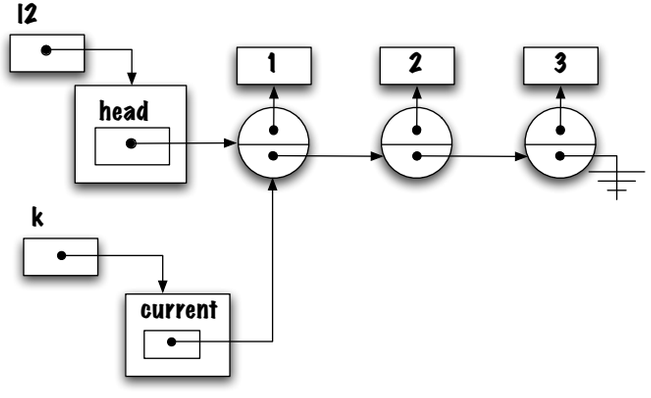



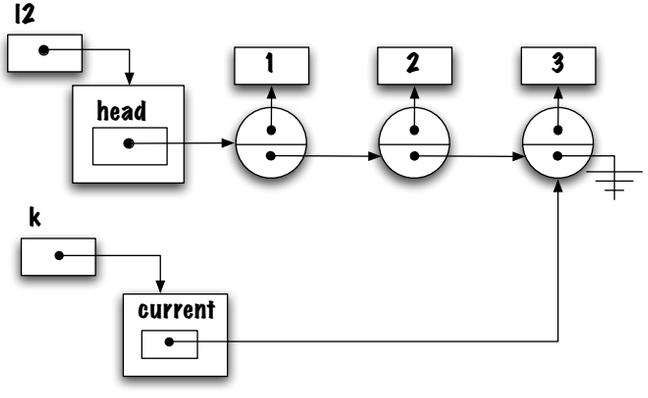




```
if ( k.hasNext() ) {
    Integer o = k.next();
}
```




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

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

    Node<E> head;

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

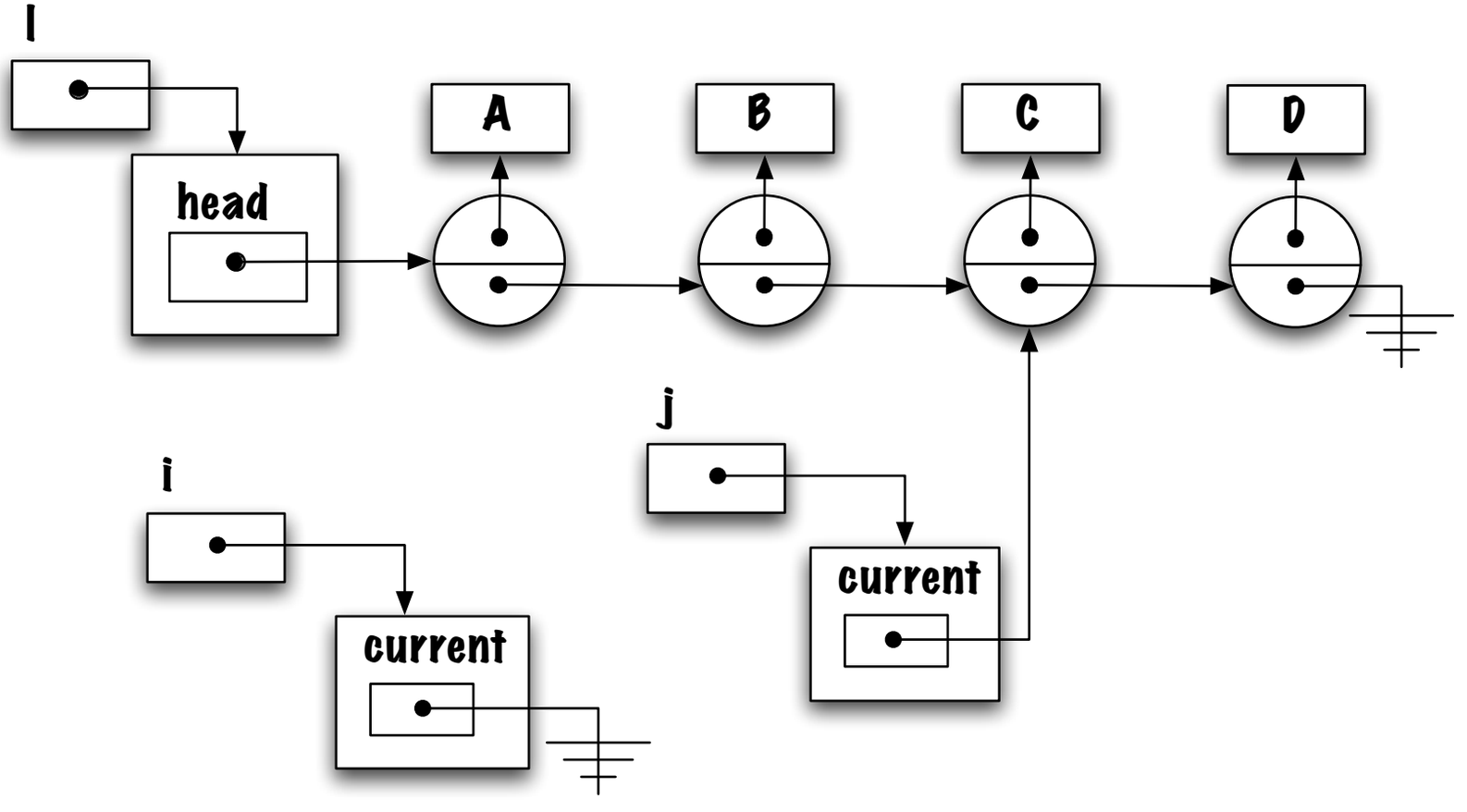
- **LinkedListIterator** is an inner class;
- **LinkedListIterator** implements **Iterator**;
- **LinkedList**
  underlinedoes not implement **Iterator**.

$\Rightarrow$ What are the instance variable of a **LinkedListIterator**?

Each object must have a variable (instance), a reference to the current element that it designates, **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 current; // <---
        ...
    }
}
```

The new implementation allows for several iterators to co-exist.



Its constructor initialises **current** to **null**, a value which, symbolically, designates being before the start of the list.

```
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; // <---
        }
        ...
    }
    ...
}
```

How to create an iterator?

```
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(); // <---
    } ...
}
```

$\Rightarrow$ **iterator()**: is an instance method of an object of the class **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<E>();
    } ...
}
```

# Usage

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

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

double sum = 0.0;

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

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

$\Rightarrow$ Exercise: trace the execution of the above statements.

# Is it fast?

We've worked hard to create a mechanism to traverse the list. What is worth it?

| # nodes | inside (ms) | iterator (ms) |
|---:|---:|---:|
| 10,000 | 2 | 6 |
| 20,000 | 4 | 5 |
| 40,000 | 8 | 11 |
| 80,000 | 14 | 19 |
| 160,000 | 23 | 48 |

# Many iterators

```
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).

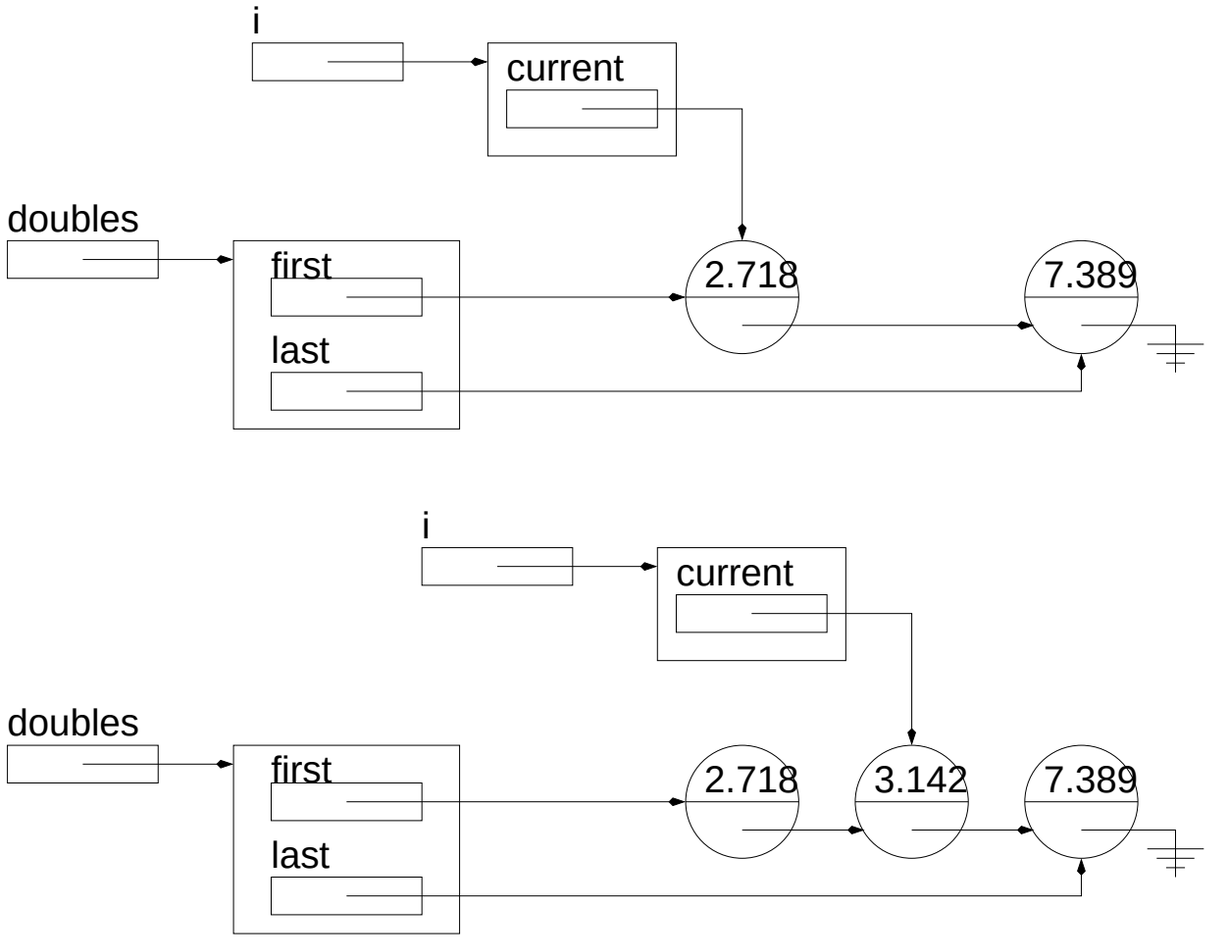
# add( E o ) and remove()

Let's add two methods to the interface **Iterator**, **add( E o )** and **remove()**.

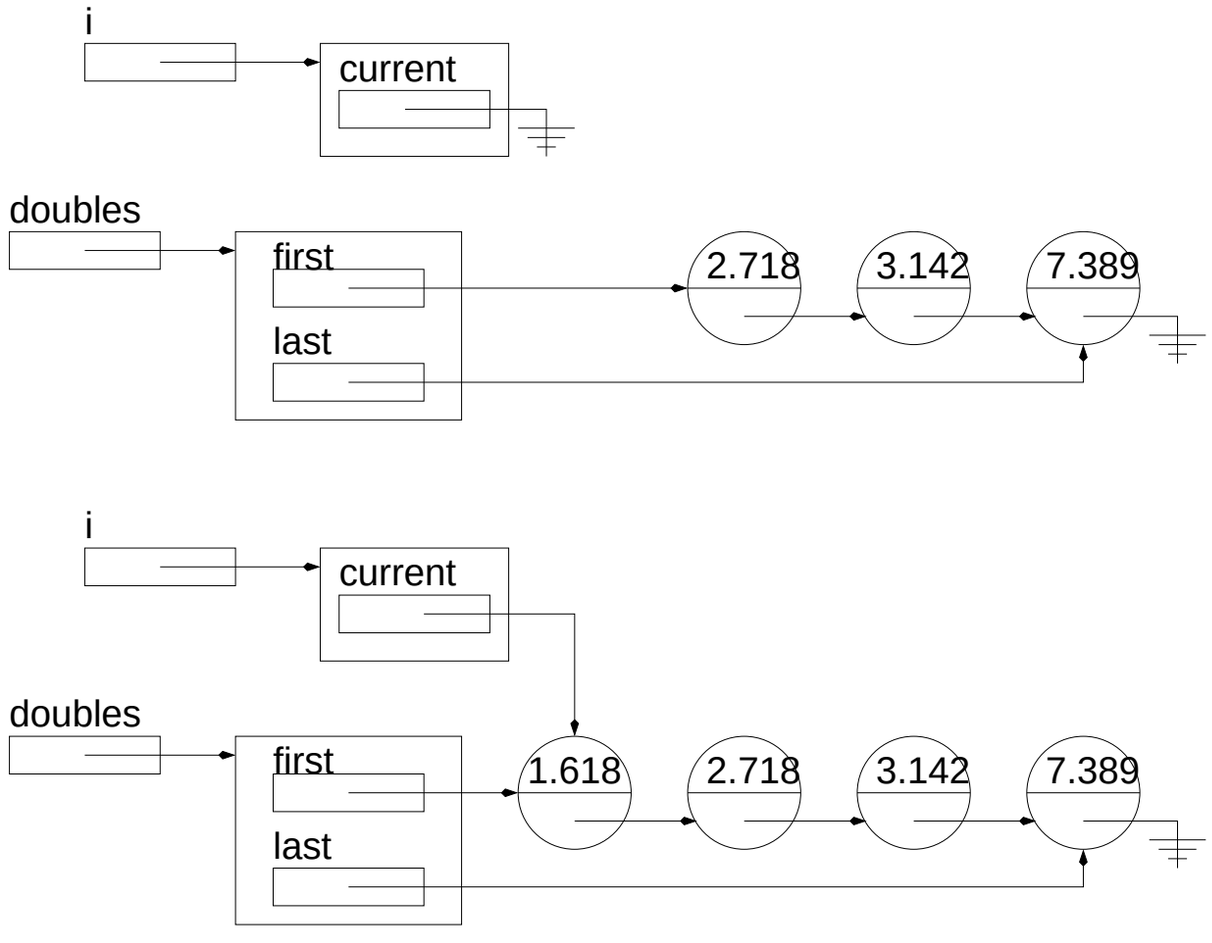
# add()

By definition, **i.add( value )** will insert the new value immediately before the next element that would be returned by **next()**.

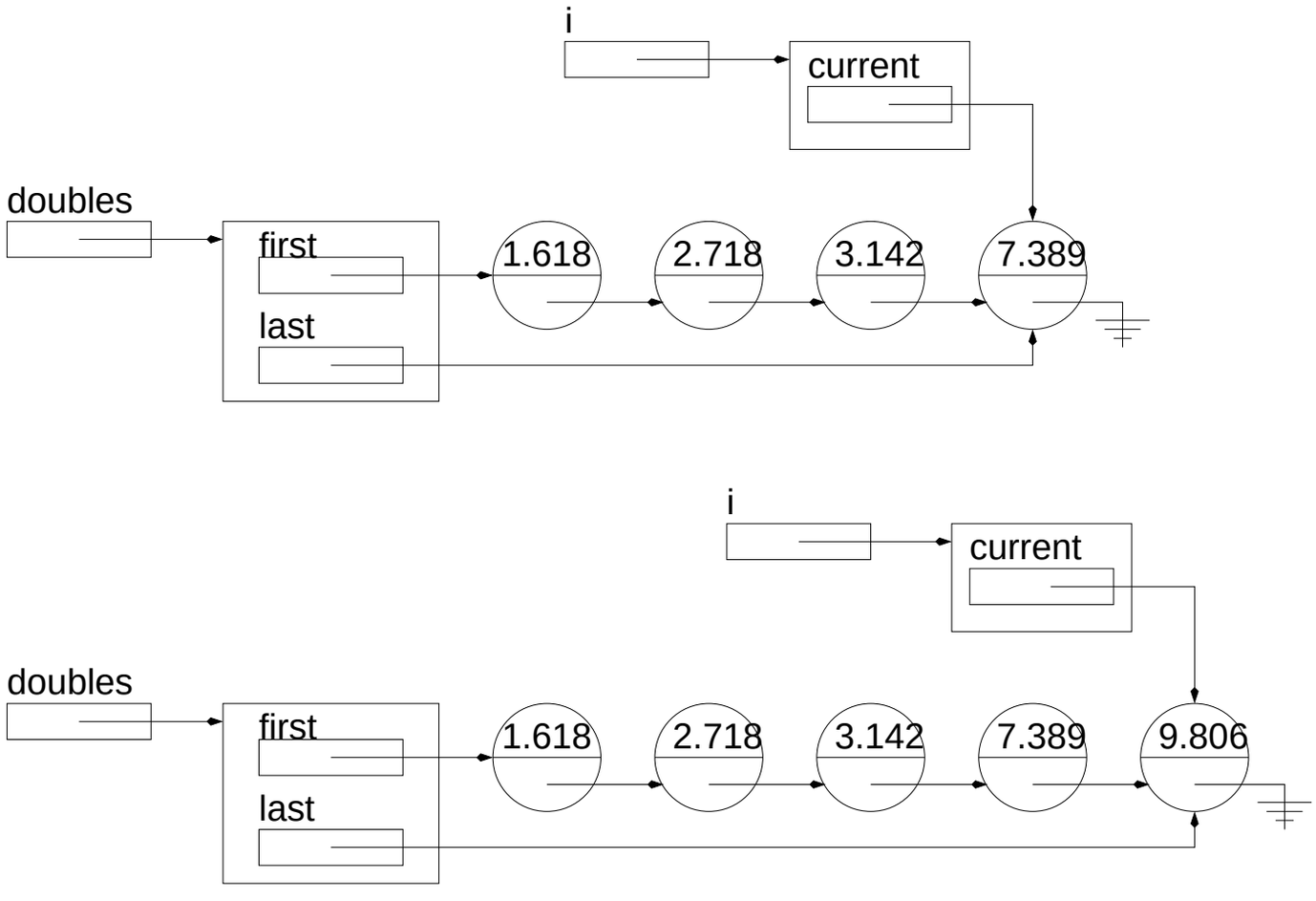
Therefore, the next call to **next()** is unaffected!




$\Rightarrow$ Call to **i.add( new Double( 3.142 ) )**, adding an element at an intermediate position in the list, notice that **i.next()** returns the same value with or without the insertion!




$\Rightarrow$ Adding an element when the iterator is positioned to the left of the list, **i.add( new Double( 1.618 ) )**, again **i.next()** returns the same value with or without insertion.




⇒ Adding an element when positioned at the end of the list, **i.add( new Double( 1.618 ) )**, notice that **hasNext()** returns the same value with or without insertion!

```
public class LinkedList<E> implements List<E> {
    private static class Node<E> { ... }
    private Node<E> first;
    private class ListIterator implements Iterator<E> {
        private Node<E> current;
        private ListIterator() { current = null;  }
        public E next() { ... }
        public boolean hasNext() { ... }
        public boolean add(E o) {
            if ( o == null )
                return false;
            if ( current == null ) {
                first = new Node<E>( o, first );
                current = first;
            } else {
                current.next = new Node<E>( o, current.next );
                current = current.next;
            }
            return true;
        }
        public void remove() { ... }
    }
```

```
    public Iterator<E> iterator() { return new ListIterator(); }
    // ... all the usual methods of LinkedList
}
```

What would the following test program display?

```
public class Test {
    public static void main( String[] args ) {
        LinkedList<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() );
        }
    }
}
```

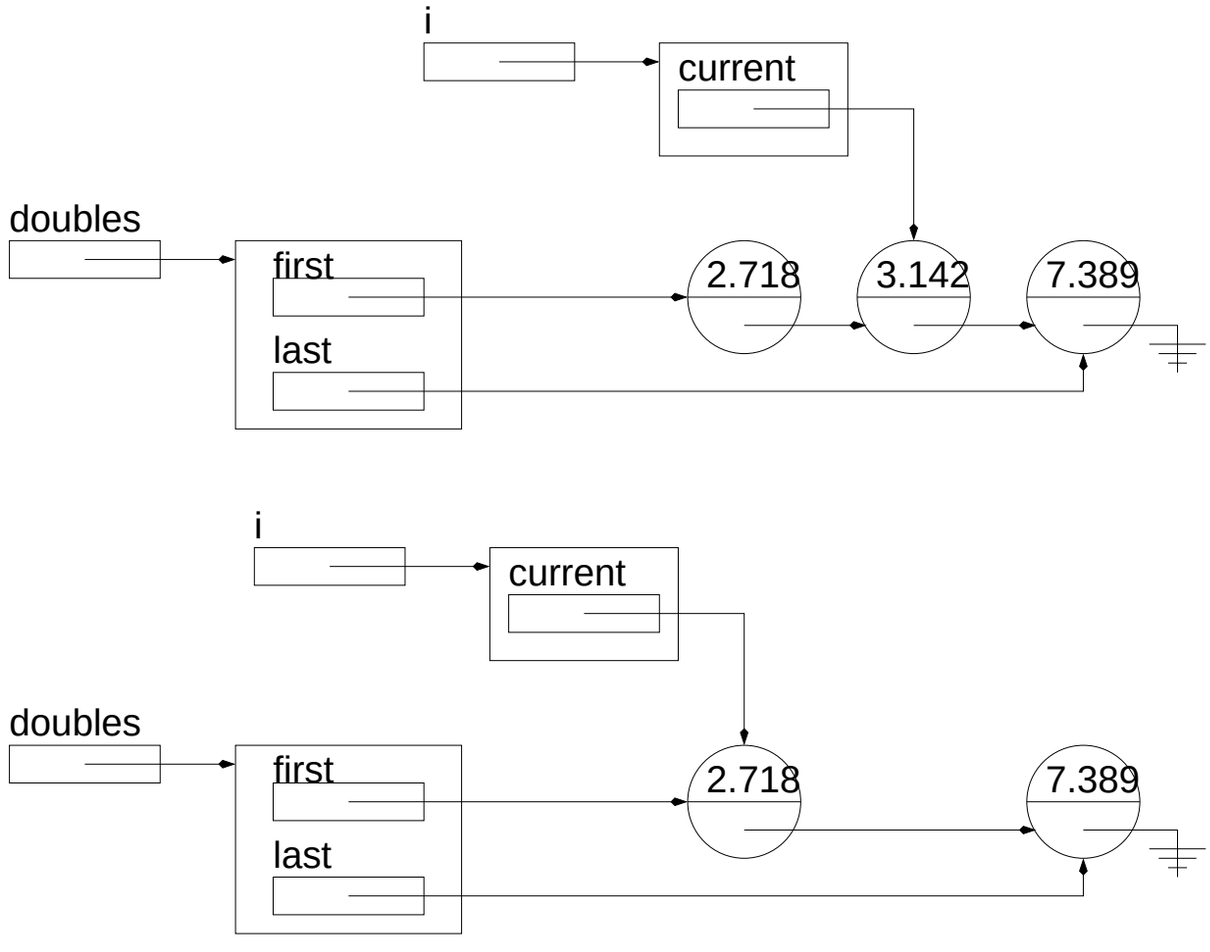
element-0
element-1
element-2
element-3
element-4

# remove()

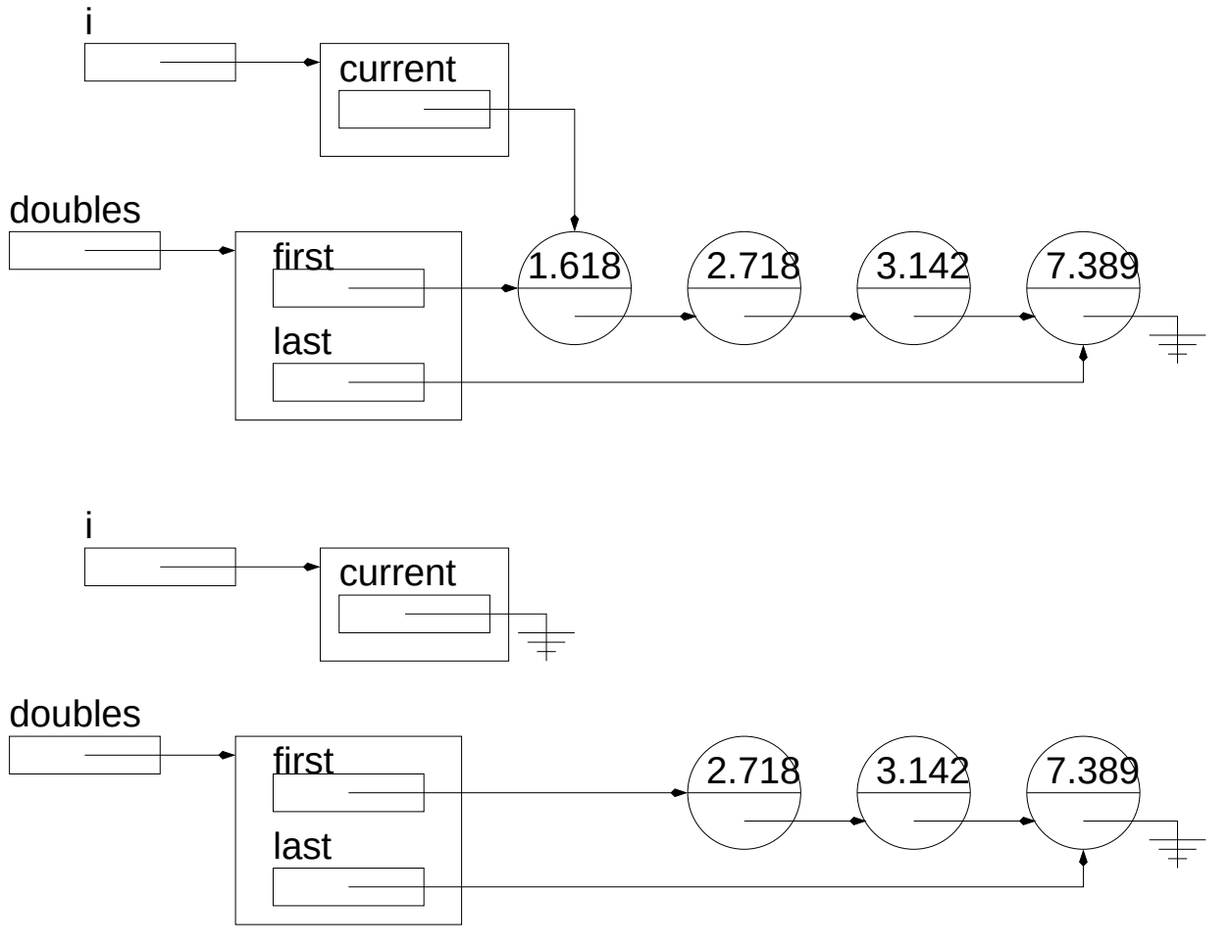
Removes from the list the last element that was returned by **next()**.

A call **i.add( value )** immediately followed by **i.remove()** keeps the list unchanged.

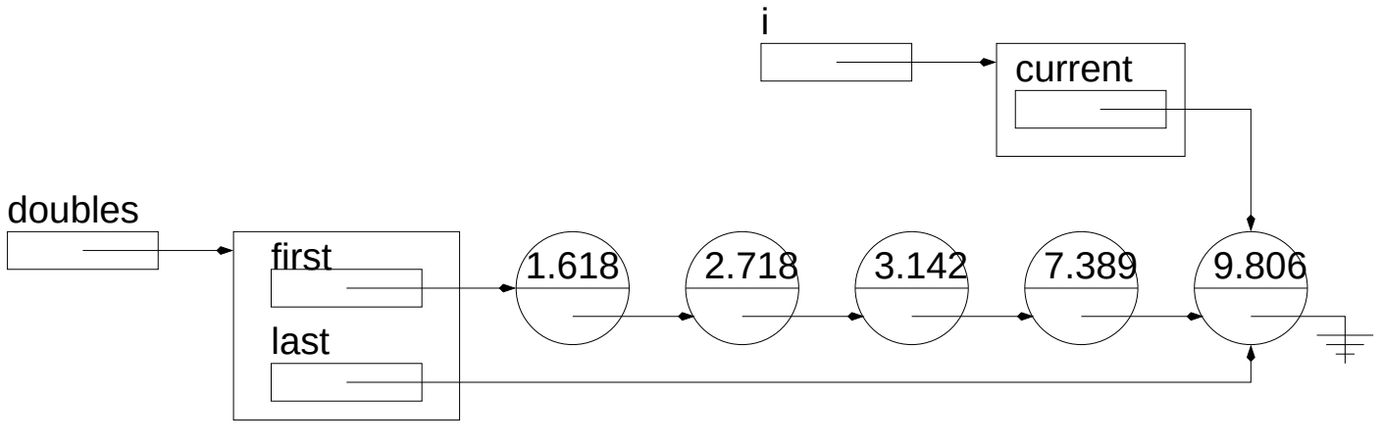
$\Rightarrow$ Notice that **remove()** does return the element that was removed, it is intended that a previous call to **next()** has saved the value.




$\Rightarrow$ Removing an element at an intermediate position, **i.remove()**, notice that the value of **i.next()** is the same with or without removal.




$\Rightarrow$ Removing the first element, **i.remove()**, the value of **i.next()** remains unchanged.




$\Rightarrow$ Removing the last element, **i.remove()**, the value of **hasNext()** is the same with or without removal.

# 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();
}
```

# Public methods of LinkedList

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

# Limitation of the current implementation?

Concurrent modifications to the list!

Having many iterators on the same list poses no problems as long as long no concurrent modifications occur.

# Many iterators and concurrent modifications

Consider:

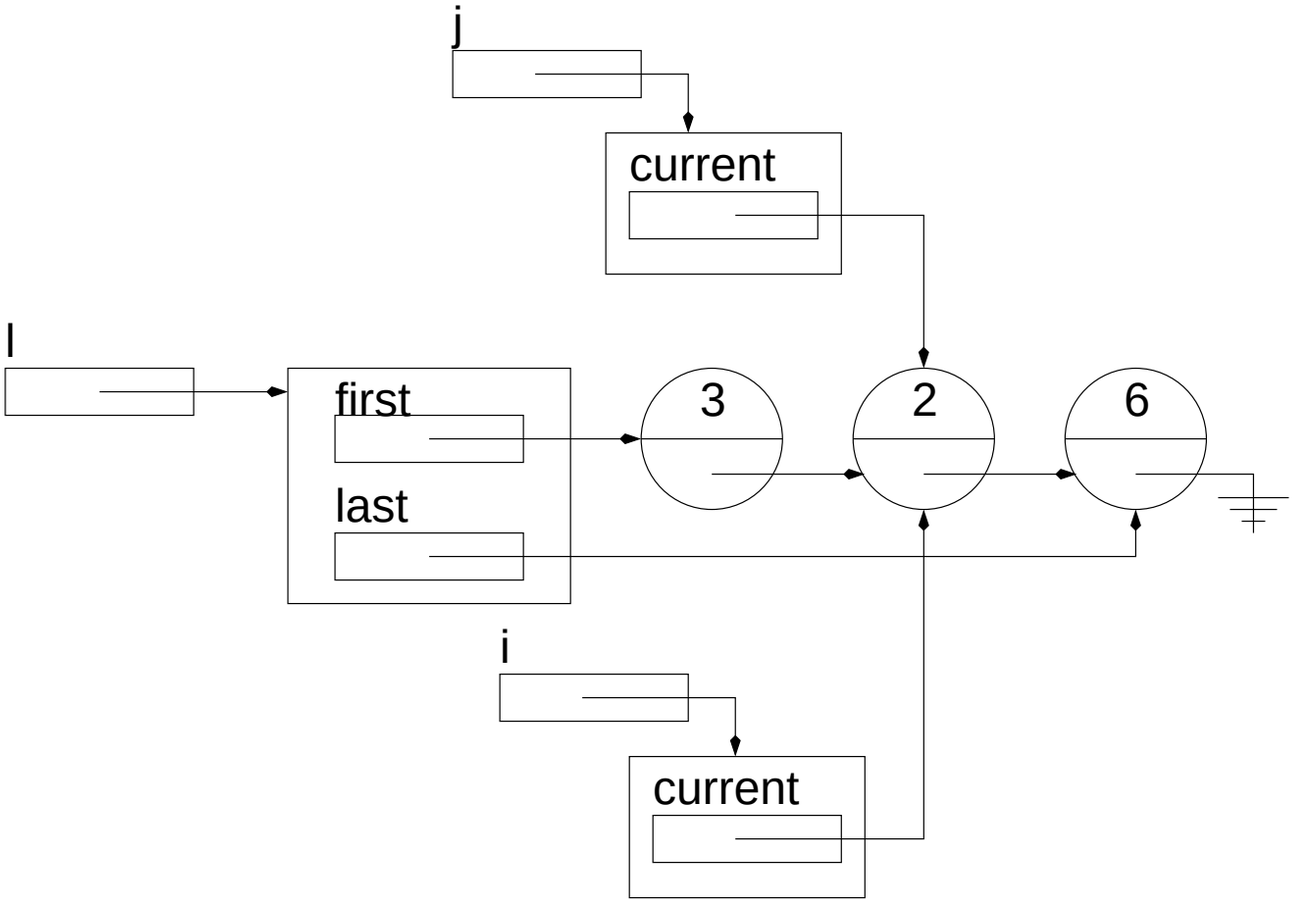
```
[1.618, 2.718, 3.142, 7.389, 9.806]
            i
            j
```

What would occur if a program calls **i.remove()**? What should occur of **j**?

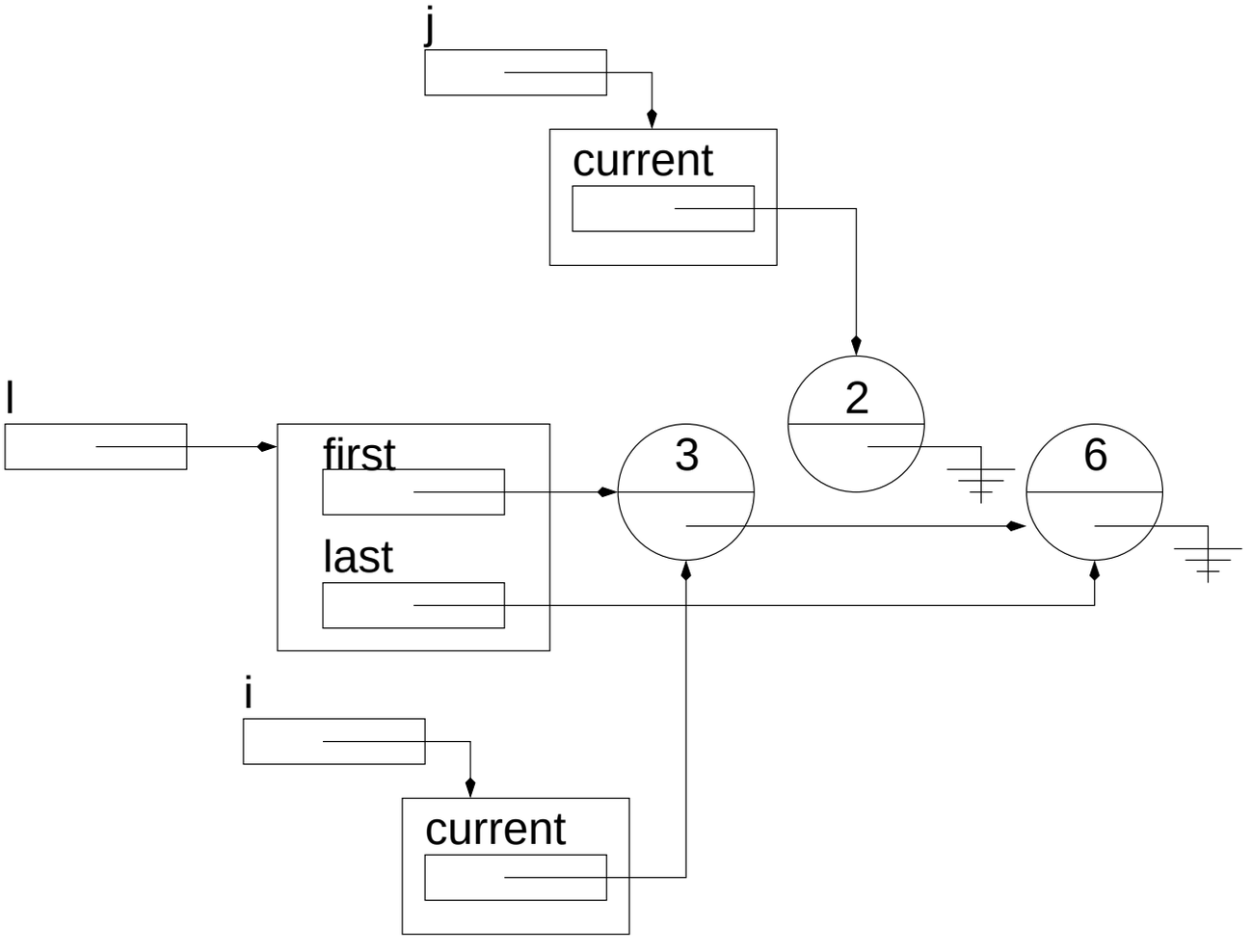
Consider:

```
[1.618, 2.718, 3.142, 7.389, 9.806]
            i
                     j
```

After the call **j.remove()** what should be the value of **i.next()**?




⇒ Consider what would occur after **i.remove()**.




$\Rightarrow$ **j** is now invalid!

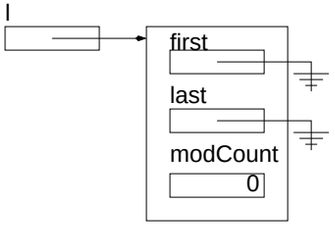
# Implementation 3: *fail-fast*

The solution that we will adopt is called **fail-fast** and consists of making an iterator invalid as soon as a modification to the list has occurred that was not made by the iterator itself; i.e. the modification was made by another iterator or an instance method of the class **LinkedList**.

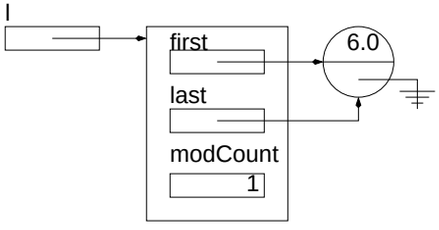
Implementing this solution requires:

1. Adding a counter of modifications to the header of the list (**modCount**);

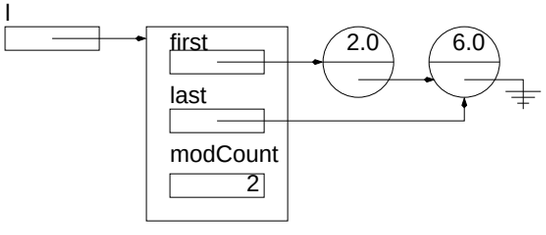
2. Adding a counter of modifications to the iterators (**expectedModCount**);



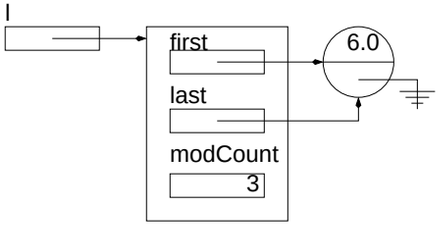

l = new LinkedList();




l.addFirst( new Double( 6.0 ) );

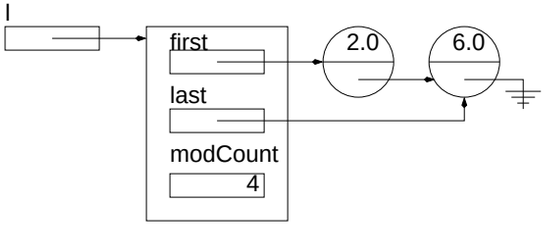



l.addFirst( new Double( 2.0 ) );

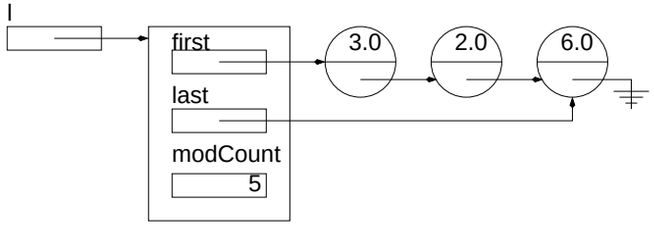



l.deleteFirst();

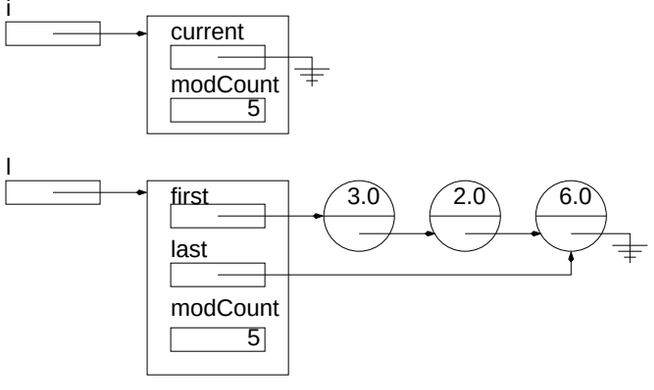
1. When creating a new list, the modification counter is set to 0;
2. Notice that **modCount** counts the number of modifications and **not** the number of elements in the list. Following the call to **deleteFirst()**, **modCount** is 3 and not 1.




l.addFirst( 2.0 );

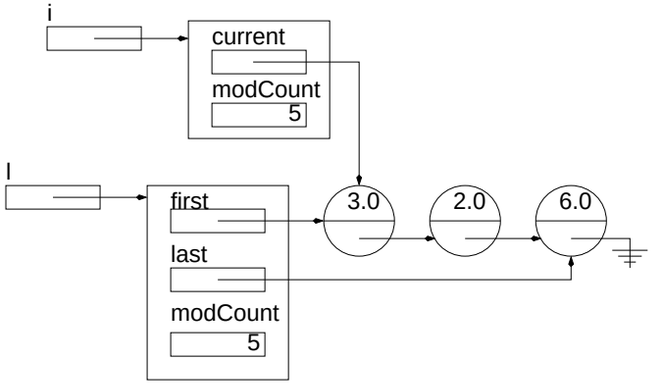



l.addFirst( 3.0 );

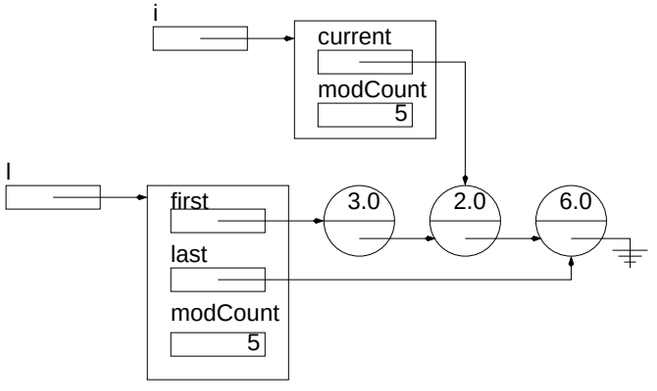



Iterator i = iterator();

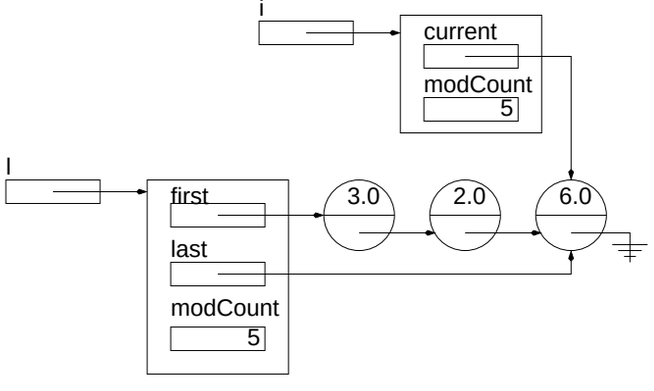
$\Rightarrow$ Creating a new **Iterator** does not change **modCount**!




i.next();

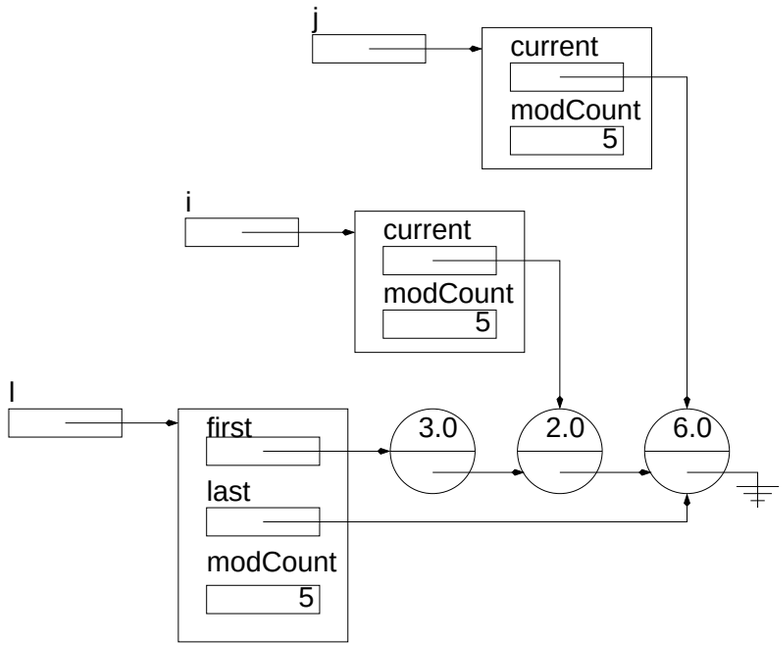



i.next();

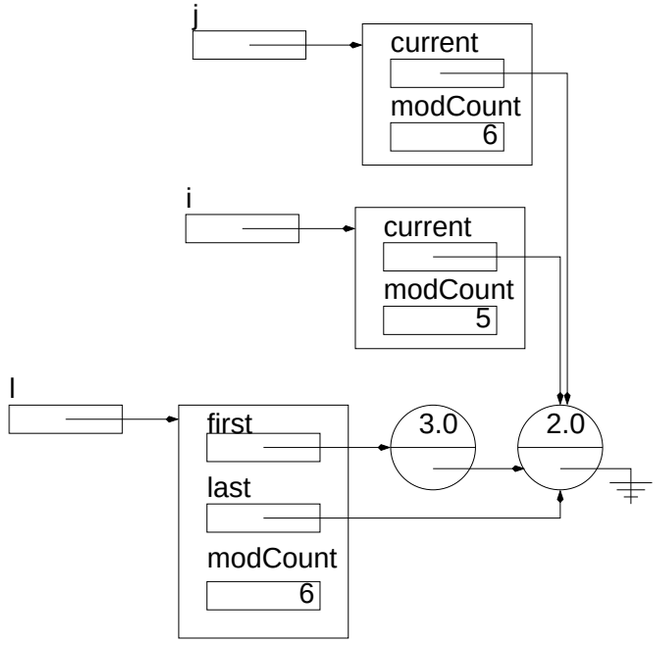



i.next();

$\Rightarrow$ Similarly, traversing the list does not change **modCount**.




Both iterators are valid




j.remove();

$\Rightarrow$ The iterator whose **expectedModCount** has the same value as the one in the header of the list remains valid; all the other ones are not.

# Changes to the class LinkedList

The list operations that modify the list, **addFist**, **deleteFirst**, etc. must now change the value of **modCount**.

When creating a new list, **modCount** must be set to zero.

# Modifications to ListIterator

When creating a new iterator, the value **modCount** from the header of the list must be copied to the instance variable **expectedModCount**.

The methods for adding and removing elements must update both, their own **expectedModCount** and **modCount** in the header of the list.

All the iterator's methods must be changed so that the precondition "is valid?" is true, otherwise they must throw the appropriate exception, **ConcurrentModificationException**.

# isValid()

To facilitate writing the methods of the inner class **ListIterator**, we create a new method called **isValid()**.

The method checks that **expectedModCount == modCount**.

# Iterator and new for loop

```
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 );
```

# Iterator and new for loop

Java 5 introduces a new syntax for the **for** loop that automagically creates and uses an iterator to traverse a list, and unboxes the elements returned by the iteration.

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

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

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

# Iterator and new for loop

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

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

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

What type can be used with the new for loop? Any type that has an iterator. What is the Java way to enforce this? Any class that implements the interface **Iterable<E>**.

# Iterable<E>

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

where

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

# 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> {
       ...
    }

    ...
}
```

# Arrays and new for loop

For the sake of completeness, the new for loop can also be applied to arrays.

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

int sum = 0;

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

# Appendix : Accessing Members of the Outer Object

In the previous example, the modification counter of the inner class (**expectedModCount**) was given a different name than that of the outer class (**modCount**) to avoid a name conflict.

The syntax **LinkedList.this.modCount** could also be used in the inner class (**ListIterator**) to refer to the instance variable of the outer object (**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;
    }
}
```

# Accessing Members of the Outer Object

```
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() );
    }
}
```

# Accessing Members of the Outer Object

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

# Advanced (optional) example

```
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
```