



# LSINF1121: Restructuration 1

## Structure Linéaires Chaînées + Complexité

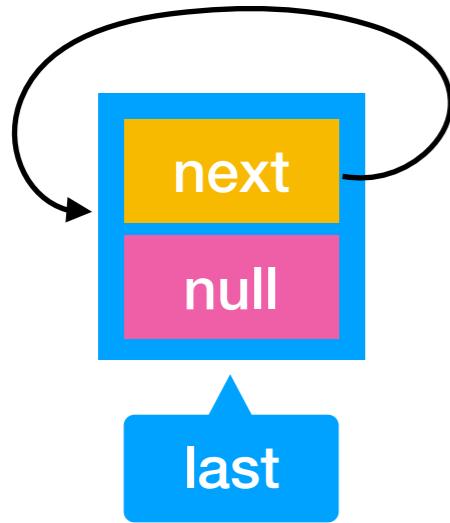
*Pierre Schaus*

# CircularLinkedList Implem



Ajouter un « dummy » element permet d'éviter les cas particulier

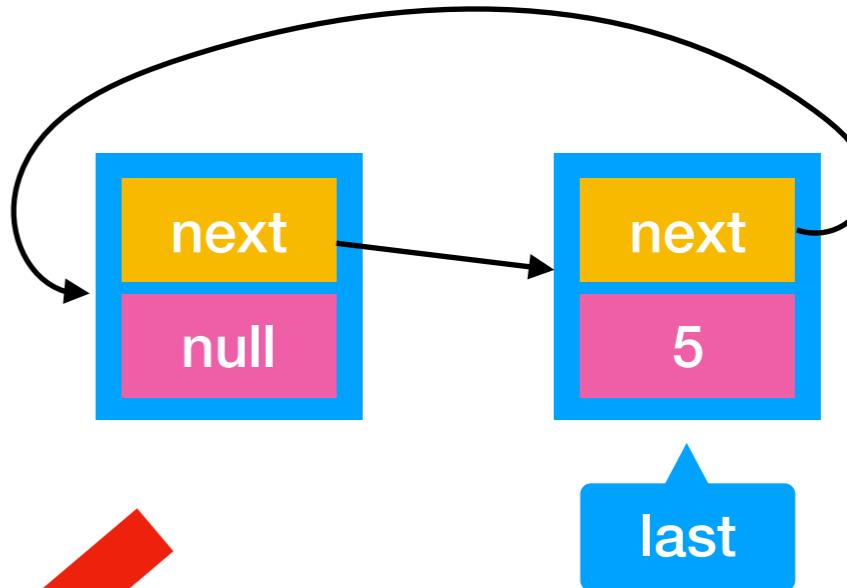
Empty list



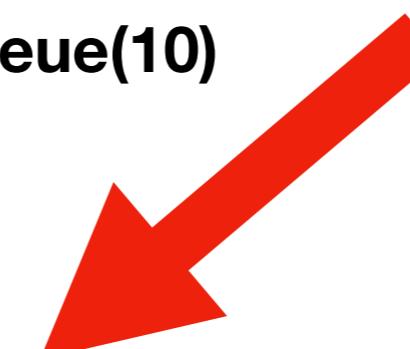
enqueue(5)



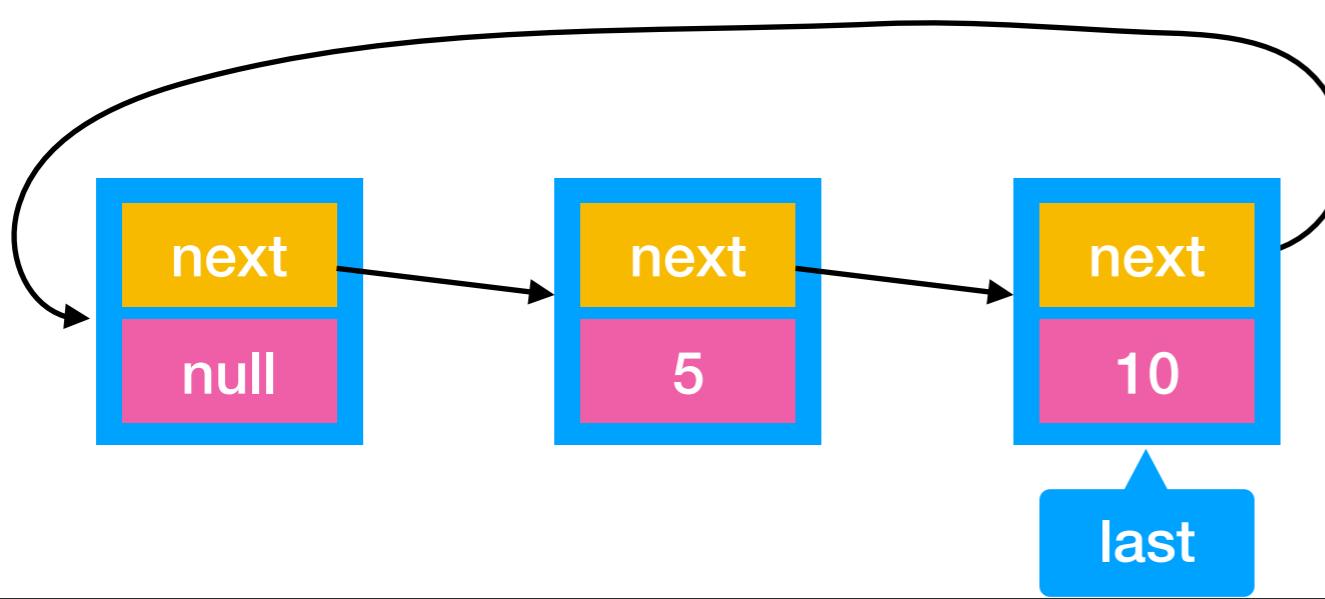
One element



enqueue(10)



Two elements



# CircularLinkedList

- Complexité de:
  - `public void enqueue(Item item) ?`
  - \*  $O(1)$
  - \*  $O(n)$  où  $n$  est le nombre d'entrées dans la liste
  - \*  $\Theta(n)$  où  $n$  est le nombre d'entrées dans la liste
  - \*  $O(1)$  amorti

# CircularLinkedList

- Complexité de:
  - \* public Item remove(int index) ?
  - \*  $O(1)$
  - \*  $O(n)$  où n est le nombre d'entrées dans la liste
  - \*  $\Theta(n)$  où n est le nombre d'entrées dans la liste
  - \*  $O(1)$  amorti



# Iterable et Iterator: rappel

- *Iterable* = interface avec une méthode *iterator()* pour générer un *Iterator*:

Modifier and Type	Method and Description
boolean	<a href="#">hasNext()</a> Returns true if the iteration has more elements.
E	<a href="#">next()</a> Returns the next element in the iteration.
void	<a href="#">remove()</a> Removes from the underlying collection the last element returned by this iterator ( <b>optional operation</b> ).

La méthode « delete » est optionnelle et généralement pas implémentée. Si pas implémentée, il faut lancer une « `NotImplementedException` » pour éviter de ne rien faire silencieusement.

- Un *Iterator* permet d'itérer sur des collections.
- Une collection qui implémente *Iterable* peut être utilisée dans les boucles `for`:

```
for (Integer i: collection) {  
    System.out.println(i);  
}
```

vs

```
Iterator<Integer> iterator = collection.iterator();  
while (iterator.hasNext()) {  
    System.out.println(iterator.next());  
}
```

# Concurrent Modification

- Une collection ne peut généralement pas être utilisée alors qu'un iterator est utilisé sur celle-ci.
- Si entre deux « hasNext/next » la collection est modifiée, il faudra lancer un « ConcurrentModificationException ».

```
List<Integer> collection = new LinkedList<>();  
collection.add(1);  
collection.add(2);  
for (Integer i: collection) {  
    collection.add(i);  
}
```

Exception in thread "main"  
java.util.ConcurrentModificationException

# Inner vs Static Nested classe en Java

```
public class OuterClass {  
  
    public int a = 0;  
  
    class InnerClass {  
        public void foo() {  
            a = 2;  
        }  
    }  
    static class StaticNestedClass {  
        public void foo() {  
            // cannot touch a;  
        }  
    }  
    public static void main(String[] args) {  
        StaticNestedClass nested = new StaticNestedClass();  
        OuterClass outer = new OuterClass();  
        OuterClass.InnerClass inner = outer.new InnerClass();  
        inner.foo(); // will change the value a in outer  
        System.out.println(outer.a);  
    }  
}
```



Les private inner classes sont très utiles pour implémenter des iterateurs car celles-ci ont accès à l'état de l'outer class

# CircularLinkedList

```
public class CircularLinkedList<Item> implements Iterable<Item> {  
  
    private int n;          // size of the stack  
    private Node last;      // trailer of the list  
  
    private class Node {  
        private Item item;  
        private Node next;  
    }  
  
    public CircularLinkedList() {  
        last = new Node();  
        last.next = last;  
        n = 1;  
    }  
    public boolean isEmpty() { return n == 1; }  
    public int size() {  
        return n-1;  
    }  
    public void enqueue(Item item) { ... }  
    public Item remove(int index) { ... }  
  
    public Iterator<Item> iterator() {  
        return new ListIterator();  
    }  
  
    private class ListIterator implements Iterator<Item> {  
        private ListIterator() { ... }  
        public boolean hasNext() { ... }  
        public void remove() {  
            throw new UnsupportedOperationException();  
        }  
        public Item next() { ... }  
    }  
}
```

Inner classes

# Et si on utilise deux iterates en même temps ?

```
CircularLinkedList<Integer> list = new CircularLinkedList<>();  
list.enqueue(1);  
list.enqueue(2);  
list.enqueue(3);  
  
for (int i : list) {  
    for (int j : list) {  
        System.out.println(i+","+j);  
    }  
}
```

A

1,1

1,2

1,3

2,1

2,2

2,3

3,1

3,2

3,3

B

1,2

1,3

2,3

C

1,2

1,3

D

**ConcurrentModificationException**

WOOCCLAP

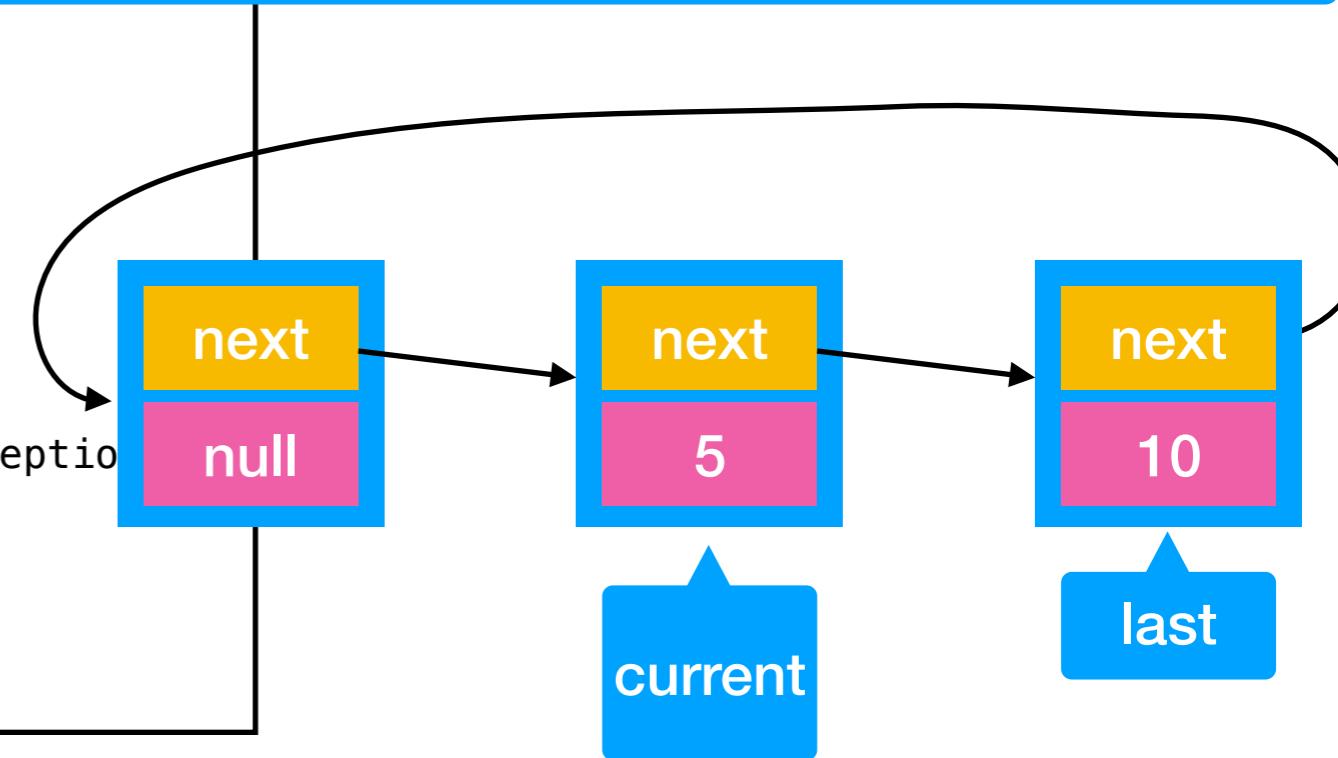
# ListIterator

```
public class CircularLinkedList<Item> implements Iterable<Item> {  
  
    private long nOp = 0; // count the number of operations  
    private int n; // size of the stack  
    private Node last; // trailer of the list  
  
    public Iterator<Item> iterator() {  
        return new ListIterator();  
    }  
}
```

```
private class ListIterator implements Iterator<Item> {
```

```
    private Node current;  
  
    private ListIterator() {  
        current = last.next.next;  
    }  
  
    public boolean hasNext() {  
        return current != last.next;  
    }  
  
    public void remove() {  
        throw new UnsupportedOperationException();  
    }  
  
    public Item next() {  
        if (!hasNext()) throw new NoSuchElementException();  
        Item item = current.item;  
        current = current.next;  
        return item;  
    }  
}
```

L'itérateur a donc son propre état interne (variable d'instance `current` propre à l'instance de itérateur au sein d'une instance particulière d'une `CircularLinkedList`) qui est le noeuds avec le prochain item à retourner



# Et le ConcurrentModificationException ?

- On va stocker un compteur interne à la liste qui compte les opérations « enqueue » et « remove »
- A chaque opération « enqueue » ou « remove » on va augmenter ce compteur.
- Au moment de créer l'itérateur, on va y stocker (variable d'instance) la variable du compteur.
- Si lorsqu'on fait « hasNext() » ou « next() », on réalise que la valeur du « compteur » enregistrée dans l'itérateur est plus petite que celle du compteur de la liste, cela signifie que l'utilisateur a modifié la liste alors qu'il est en train d'utiliser l'itérateur => ConcurrentModificationException



# Iterateur avec détection de modification

```
public class CircularLinkedList<Item> implements Iterable<Item> {  
  
    private long nOp = 0;      // count the number of operations  
    private int n;             // size of the stack  
    private Node last;        // trailer of the list  
  
    ...  
  
    private long nOp() {  
        return nOp;  
    }  
  
    public void enqueue(Item item) {  
        nOp++;  
        ...  
    }  
    public Iterator<Item> iterator() {  
        return new ListIterator();  
    }  
}
```

Augmente le nombre d'opération

```
private class ListIterator implements Iterator<Item> {  
  
    private Node current;  
    private long nOp;  
  
    private ListIterator() {  
        nOp = nOp();  
        current = last.next.next;  
    }  
  
    public boolean hasNext() {  
        if (nOp() != nOp) throw new ConcurrentModificationException();  
        return current != last.next;  
    }  
}
```

Capture du nombre d'opération sur la liste au moment de la création de l'itérateur

Nombre d'opérations sur la liste au temps présent

Nombre d'opérations sur la liste à la création de l'itérateur

# CircularLinkedList

- Complexité de:
  - d'une séquence d'opérations qui consistent à créer un itérateur et ensuite itérer sur les k-premiers éléments ?
    - \*  $O(n)$  où n est le nombre d'entrées dans la liste
    - \*  $\Theta(n)$  où n est le nombre d'entrées dans la liste
    - \*  $O(k)$
    - \*  $\Theta(k)$



# Evaluation d'expression post-fixe

- Par exemple "2 3 1 \* + 9 \* »
- Ces expression peuvent être évaluées facilement à l'aide d'une:
  - ▶ Queue (FIFO) ?
  - ▶ Stack (LIFO) ?
  - ▶ Un arbre ?



# Post-fix evaluation

- input 

4	20	+	3	5	1	*	*
---	----	---	---	---	---	---	---
- algo:
  - ▶ stack = new Stack()
  - ▶ i = 0
  - ▶ While (i < input.length) :
    - \* input[i] = a number => stack.push(input[i])
    - \* input[i] = an operator => stack.push(stack.pop() input[i] stack.pop())
    - \* i++
  - ▶ return input[0]

# Complexité temporelle d'une éval post-fix ?

- En supposant un push/pop en  $O(1)$  et  $n$  la taille de l'input:
  - ▶  $\Theta(n)$
  - ▶  $\Theta(n^2)$
  - ▶  $O(n)$
  - ▶  $O(n^2)$



# Les listes fonctionnelles

- Un peu comme dans Oz {Browse {List.append Xs [2 3 4]}}
- Les structures fonctionnelles sont immuables

```
public abstract class FList<A> implements Iterable<A> {
    // creates an empty list

    public static <A> FList<A> nil();
    // prepend a to the list and return the new list
    public final FList<A> cons(final A a);

    public final boolean isEmpty();
    public final int length();
    // return the head element of the list
    public abstract A head();
    // return the tail of the list
    public abstract FList<A> tail();
    // return a list on which each element has been applied function f
    public final <B> FList<B> map(Function<A,B> f);
    // return a list on which only the elements that satisfies predicate are kept
    public final FList<A> filter(Predicate<A> f);
    // return an iterator on the element of the list
    public Iterator<A> iterator();

}
```

La liste vide

Ajoute un élément à la liste et retourne la nouvelle liste.

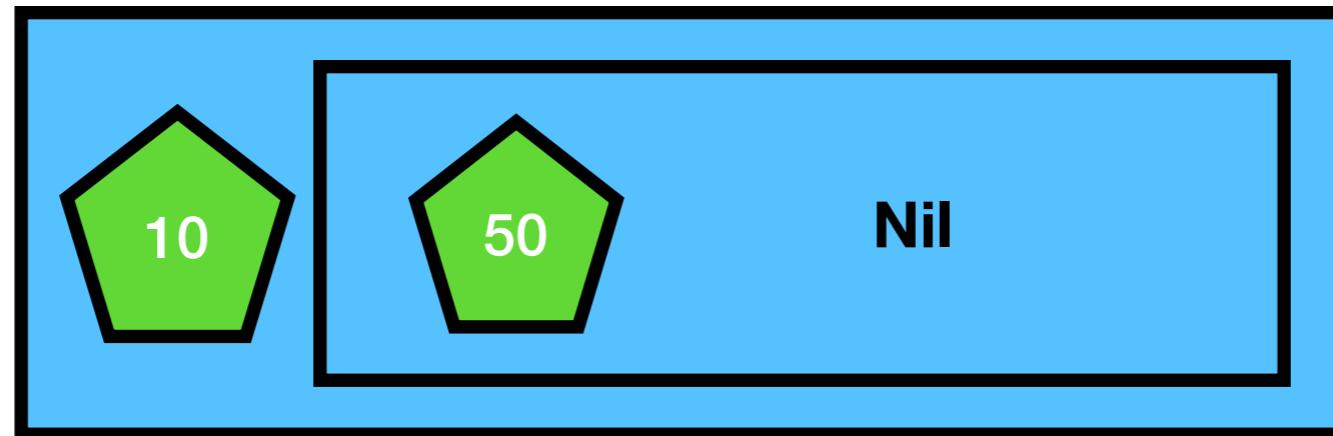
Complexité :

- O(1)
- O(n)
- $\Theta(n)$



# Les listes fonctionnelles: FList

- Définition récursive:
- Une FList est soit:
  - ▶ Une liste vide (Nil)
  - ▶ Un élément suivi d'une FList
- Exemple pour la liste [10,50]



- Notre implémentation va refléter fidèlement cette définition

# Quelle est la complexité spatiale de cette méthode?

- $\Theta(n)$
- $\Theta(n^2)$
- $O(n)$
- $O(n^2)$

```
/*
 * @param n the size of the list
 * @return The list [0,1,...,n-1]
 */
public static FList<Integer> rangeList(int n) {
    FList<Integer> list = FList.nil();
    for (int i = n-1; i >= 0; i--) {
        list = list.cons(i);
    }
    return list;
}
```

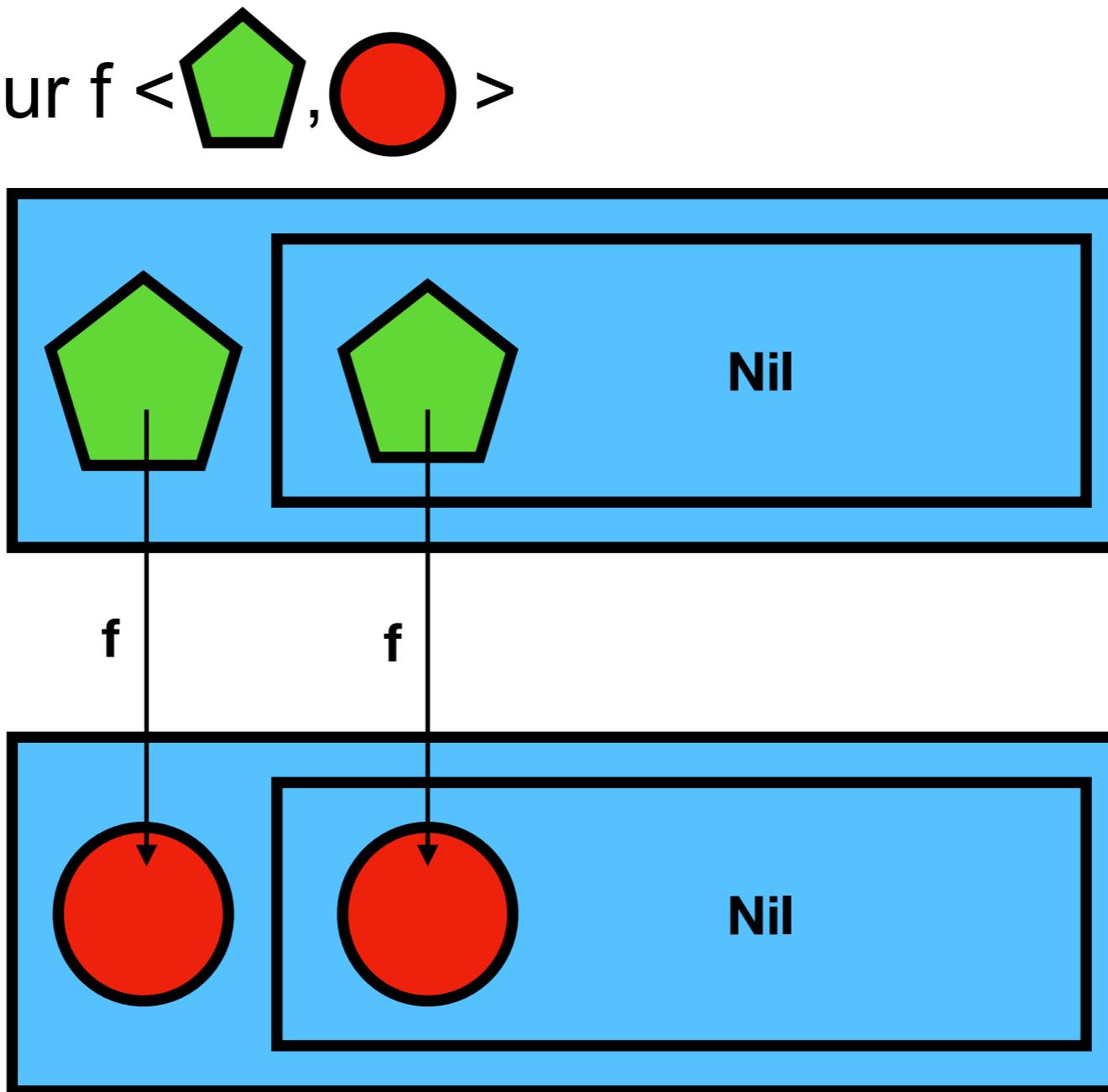
WOOCCLAP

# La méthode « map »

- Appliquée sur une  $FList<A>$ , prend en argument « une fonction »  $f<A,B>$  et reconstruit une liste de type  $FList<B>$

```
public final <B> FList<B> map(Function<A,B> f)
```

- Exemple pour  $f < \text{pentagon}, \text{circle} >$



# La méthode « map »

- Exemple:

```
FList<Integer> list = FList.nil();  
  
for (int i = 9; i >= 0; i--) {  
    list = list.cons(i);  
}  
  
Function<Integer, String> f = new Function<Integer, String>() {  
    @Override  
    public String apply(Integer k) {  
        String res = "";  
        for (int i = 0; i < k; i++) {  
            res += "*";  
        }  
        return res;  
    }  
};  
  
FList<String> rList = list.map(f);  
for (String r: rList) {  
    System.out.println(r);  
}
```

- TimeComplexity ?
  - $\Theta(n)$
  - $\Theta(n^2)$
  - $O(n)$
  - $O(n^2)$

WOOCCLAP

# Sucre syntaxique (Java8) pour les fonctions en arguments

Implémentation implicite de l'unique méthode de l'interface « fonctionnelle »

```
@FunctionalInterface  
public interface Function<T, R> {  
    R apply(T t);  
}
```

```
FList<Integer> list = FList.nil();  
  
for (int i = 9; i >= 0; i--) {  
    list = list.cons(i);  
}  
  
FList<String> rList = list.map(k -> {  
    String res = "";  
    for (int i = 0; i < k; i++) {  
        res += "*";  
    }  
    return res;  
});  
  
for (String r: rList) {  
    System.out.println(r);  
}
```

=

```
FList<String> rList = list.map(  
    new Function<Integer, String>() {  
        @Override  
        public String apply(Integer k) {  
            String res = "";  
            for (int i = 0; i < k; i++) {  
                res += "*";  
            }  
            return res;  
        }  
    } );
```



# LSINF1121: Restructuration 1

## Structure Linéaires Chaînées + Complexité

*Pierre Schaus*