Functional Programming in Java
Part 2

Last week: A list without side effects

```
class Cons {
    public final int value;      // value of the element
    public final Cons next;      // next element, null if end of list

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

    public Cons map(F f) { ... }

    public Cons filter(P p) { ... }
}
```

- Examples:

```
Cons list1 = new Cons(5,new Cons(3,null));   // the list [5,3]
```

Reminder: Exercise on INGInious

- In the INGInious exercises, we asked you to implement a map method and a filter method for an immutable Cons list:

  - The map method takes a function $f: int \rightarrow int$ and applies it to all elements of a list. The result is a new list.

    Example:

    ```
    Cons result = list.map( (i)-> i+3);
    ```

  - The filter method takes a predicate $p: int \rightarrow boolean$ and applies it to all elements of a list. The result is a new list with all elements for which $p(x) = true$.

    Example:

    ```
    Cons result = list.filter( (i)-> i>3);
    ```

Problem of our Cons class

- Imagine we want to filter a list and then increment the elements:

```
Cons list = ... ;
Cons result = list.filter( (i) -> i<5 ).map( (i)-> i+3);
```

Possible NullPointerException!

- We cannot write that!
  - `list.filter(someFilter)` might return an empty list
  - Since we represent empty lists by null, the method `map` would fail
- Correct code:

```
Cons filteredList = list.filter( (i) -> i<5 );
Cons result;
if(filteredList == null)
   result = null;
else
   result = filteredList.map( (i)-> i+3 );
```

Ugly... And we have to do that before every list operation!

Problem of our Cons class (2)

- In last week's lecture, I "cheated" by using a static method in the Cons class:

```
public static Cons increment(Cons list) {
        if(list==null)
            return null;
        else
            return new Cons(list.value+3, increment(list.next));
    }
```

- This method can be used on empty lists, but it's still ugly and every method working with lists would have to be implemented like that: map, filter, length,...

A better list implementation

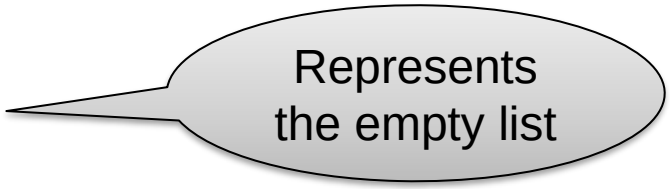
- Instead of using null, we can use an object to represent the empty list:

```
abstract class FList { }

class Nil extends FList { }

class Cons extends FList {
    private FList next;
    private int value;

    public Cons(int value, FList list) {
        this.value=value;
        this.next=list;
    }
}
```



- How to use these classes:
  - An empty list: FList list0 = new Nil();
  - A list with one element: FList list1 = new Cons(3,list0);
  - A list with two elements: FList list2 = new Cons(5,list1);

A better list implementation (2)

- Having no null references makes things easier. Here is a possible implementation of the increment method:

```
abstract class FList {
    public abstract FList increment();
}

class Nil extends FList {
    public FList increment() {
        return new Nil();
    }
}

class Cons extends FList {
    private FList next;
    private int value;

    public Cons(int value, FList list) {
        this.value=value;
        this.next=list;
    }

    public FList increment() {
        return new Cons(value+3), next.increment());
    }
}
```

Incrementing an empty list returns an empty list

No ugly if(list==null) anymore!

A better list implementation (3)

- If we implement the other methods (map, filter,...) like the increment method, we can write without problems:

```
Cons result = list.filter( (i) -> i<5 ).map( (i) -> i+3);
```

- Have you noticed? This line of code is composed of operations that are all functions (in the mathematical sense) without side effects:
  - filter : $FList \times (int \rightarrow boolean) \rightarrow FList$
  - `(i) -> i<5` : $int \rightarrow boolean$
  - map : $FList \times (int \rightarrow int) \rightarrow FList$
  - `(i) -> i+3` : $int \rightarrow int$
- Therefore, we can see the entire line also as a function without side effects: $List \rightarrow List$
- The result of this line of code only depends on the `list` variable. Very easy to read!

The exercise on Inginious

- Two tricks used in the Inginious exercise (where you have to implement an extended version of the FList class):
  1. The Nil object can be implemented as a singleton. No need to have multiple Nil objects!
  2. The Nil and Cons classes have been moved into the FList class as static nested classes. Nothing important, it's just to make the code organization cleaner!

```
public abstract class FList {

    ...

    public static final class Nil extends FList {
        ...
    }
}
```