Java 101 - Magistère BFA

Lesson 4: Generic Type and Collections

Stéphane Airiau

Université Paris-Dauphine

Linked List

```java
public class Node {
   String value;
   Node next;

   public Node(String val) {
     value = val;
   }

   public void setNext(Node next) {
     this.next = next;
   }
}
```

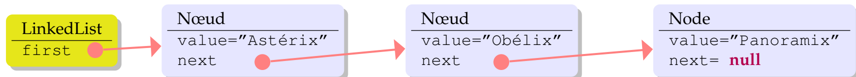
```java
public class LinkedList {
   Node first;

   public LinkedList() {
     premier = null;
   }

   public void add(String val) {
     Node newNode = new Node(val);
     if (first == null)
       first = newNode;
     else {
       Node last = first;
       while (last.next != null)
         last = last.next;
       last.next = newNode;
     }
   }
}
```

Exemple

{"Astérix","Obélix","Panoramix"},



Generic Type

Let us build a list of `Characters`.

- Create two classes : one for a node containing a `Character`, the other for a linked list of `Character`.

Generic Type

Let us build a list of `Character`s.

- Create two classes : one for a node containing a `Character`, the other for a linked list of `Character`.
- Modify our `Node` class by replacing `String` with `Object`.

Generic Type

Let us build a list of `Characters`.

- Create two classes : one for a node containing a `Character`, the other for a linked list of `Character`.
- Modify our `Node` class by replacing `String` with `Object`.

➡ this is possible (it was done until version 5 of `Java`), but we will need to use **cast**

Generic Type

Let us build a list of `Characters`.

- Create two classes : one for a node containing a `Character`, the other for a linked list of `Character`.
- Modify our `Node` class by replacing `String` with `Object`.
- ⇨ this is possible (it was done until version 5 of `Java`), but we will need to use **cast**
- and what if we can put a **type parameter** ?

Generic Type

Let us build a list of `Characters`.

- Create two classes : one for a node containing a `Character`, the other for a linked list of `Character`.
- Modify our `Node` class by replacing `String` with `Object`.
- ➡ this is possible (it was done until version 5 of `Java`), but we will need to use **cast**
- and what if we can put a **type parameter** ?

```
1  public class Node<E> {
2     E value;
3     Node<E> next;
4
5     public Node(E val){
6       value = val;
7     }
8
9     public void setNext(Node<E> next){
10      this.next = next;
11    }
12 }
```

```
public class LinkedList<E> {
  Node<E> first;

  public LinkedList(){
    first = null;
  }

  public void add(E val){
    Node<E> newNode = new Node<E>(val);
    if (first == null)
      first = newNode;
    else {
      Node<E> last = first;
      while(last.next != null)
          last = last.next;
      last.next = newNode;
    }
  }

  public E get(int index){
    int i=0;
    Node<E> current=first;
    while(current.next != null && i<index){
      i++;
      current = current.next;
    }
    if(index == i) // we found ith element
      return current;
    else
      return null;
    }
  }
}
```

Use

```
1  IndomitableGaul asterix =
2                  new IndomitableGaul("Astérix");
3  IndomitableGaul obelix =
4                  new IndomitableGaul("Obélix");
5  Gaul Informatix = new Gaul("Informatix");
6  LinkedList<Gaul> list = new LinkedList<Gaul>();
7  list.add(asterix);
8  list.add(obelix);
9  list.add(informatix);
```

- The type parameter can **not** be a primitive type (ex **int**, **char**, **double**, etc...)
  The parameter can only be an **object**
  ex : `Node<int>` is not allowed.
- When calling the constructor, one does not have to repeat the parameters (but you must use <>).
  ex :`LinkedList<Gaul> list = new LinkedList<>();`
  `Java` will infer the parameter type

Autoboxing

Java can now perform some automatic changes

```
LinkedList<Integer> myList = new LinkedList<Integer>();
//old style
myList.add(new Integer(7));
Integer seven = myList.get(1);
System.out.println(seven.intValue());
//new style
myList.add(6);
int six = myList.get(2);
```

Type parameter & inheritance

One class with a parameter can inherit from a class with a parameter

```
1 class <class name> < parameter 1>
2       extends <super class> < parameter 1>
3  { ... }
```

```
1 class Tuple<T,U> { ... }
2 class ApprenticeMentor<T,U> extends Tuple<T,U> { ... }
```

Inheritance of the parameters - use as bounds

```
1 class <class name> < parameter 1 extends <super class name> >
2
3  { ... }
```

```
1 class Distribution<E extends Character>{ ...}
```

We specify that the type parameter `E` must be a subclass of `Character`.

Some subtleties

```
LinkedList<Gaul> lg = new LinkedList<Gaul>;
LinkedList<Character> lp = lg;
```

On line 2, we feel like writing that a `Gaul` list is also a `Character` list. Is this correct ?

Some subtleties

```
1  LinkedList<Gaul> lg = new LinkedList<Gaul>;
2  LinkedList<Character> lp = lg;
```

On line 2, we feel like writing that a `Gaul` list is also a `Character` list. Is this correct ?

```
3  lp.add(new Character("Jules César"));
4  Gaul g = lg.get(1);
```

But we could obtain a character that is not a `Gaul` !

Some subtleties

```
1  LinkedList<Gaul> lg = new LinkedList<Gaul>;
2  LinkedList<Character> lp = lg;
```

On line 2, we feel like writing that a `Gaul` list is also a `Character` list.
Is this correct ?

```
3  lp.add(new Character("Jules César"));
4  Gaul g = lg.get(1);
```

But we could obtain a character that is not a `Gaul` !

Actually, the `Java` compiler will not allow line 2.

➯if `S` is in the family of `F`, if `C` is a class that uses a parameter, `C<S>` is not in the family of `C<F>`

There is no relationship between `C<S>` and `C<F>`

Jockers

Java allows the use of an unknown type.

```
1 LinkedList<?> list = new LinkedList<Gaul>();
```

- We will **not** be able to use an add method as we should use something of type ?
- however, we **can** use a method such as get
  ➯ but we should use a cast
- to be useful, we will use a bound

upper bound

`LinkedList<? extends Gaul>` the unknown type must be in the family of `Gaul`.

```
1 public void introduce(LinkedList<Character> list){
```

✘we cannot use a `LinkedList<Gaul>`

```
1 public void introduce(LinkedList<? extends Character> list){
```

lower bound

`LinkedList<? super IndomitableGaul>` the unknown type must be a parent, here it must be a parent of indomitable Gaul.

```
1 public class Collections {
2    public static <T> void copy
3         (List<? super T> dest, List<? extends T> src ){ ... }
```

here it is nasty, the bound is a parameter type !

Some restrictions

- we can not use primitive types (`int`, `double`, etc..) as parameter types
- we cannot create an array of parameter types
  ex : `Node<Gaul>[] array = new Node<Gaul>[10];` is **not** allowed.
- the parameter of a class cannot be used in a `static` context.

```
1 public class Paire<C,V>{
2    private static V valueDefaut;
3         error!!
4    public static void setDefaut(V value){valueDefaut=value;}
6         error!!
```

- there are more subtleties that we will not mention here.

Static method with type parameters

We can use a type parameter with a static method.

In the declaration, the type parameter must be declared (so `Java` knows it is a parameter type). it is declared <u>before</u> the return type and <u>after</u> the visibility (`public, private`) and (`static`).

```
1  public class ArrayUtil {
2    public static <T> void swap(T[] array, int i, int j) { ... }
```

When calling such method, `Java` will infer what is the parameter type !
ex : `ArrayUtil.swap(villagers, 2, 6);`

If we really want, we can still specify the type.
ex : `ArrayUtil.<Gaul>swap(villagers, 2, 6);`

Collections

Collections

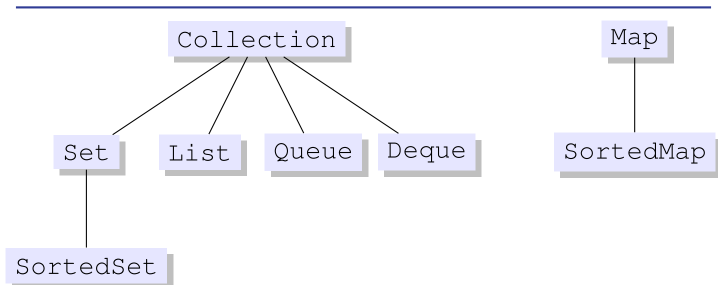
lists, sets, queues are "things" that gather together different objects in one entity

- They share :
  - similar queries : are there any elements, how many
  - same types of operations : add, remove an element, empty it, go over each element
- But the details differ (ex : fifo vs lifo first in first out vs last in first out)

Q : how to manipulate such structures ?

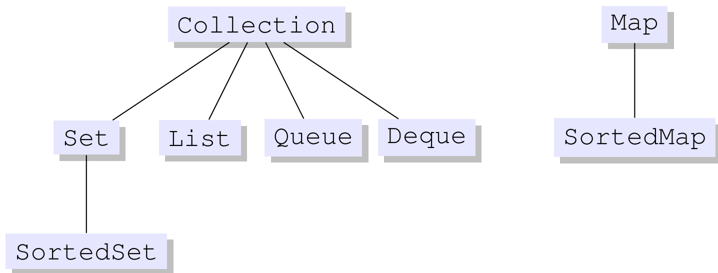
R : ➯ use an interface hierarchy

Interface hierarchy



- `Collection`: all most general methods
- `Set` : as a set in mathematics : cannot have twice the same element. Order of introduction is not important.
- `List` : sequence of elements (order of addition is important). Two (or more) copies of the same object can be members.
- `Queue` : Two (or more) copies of the same object can be members. Order of introduction is not important.

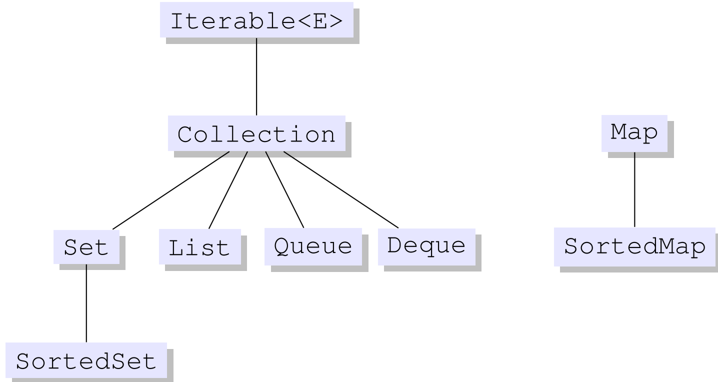
Hierarchie d'interfaces



- `Map` : binary relation (surjection) : mapping (key, value), the key must be unique.
- `SortedSet` is the ordered version of a set
- `SortedMap` is the ordered version of a set where keys are sorted

Each interface has a parameter type : we will have a collection of `Gaul`s, `Integer`s, `String`s, etc...

iterate over a collection



Use a "**for each**" loop on any object that implements the interface `Iterable`.

Iterate : first solution

- **Situation :** we have a collection `myCollection` containing objects of type `E`.
- we iterate using **for**
- each element will be accessible using a variable`<name>` of type `E` (of course !).

```
1  Collection<E> myCollection;
2  ...
3  for (E <nom> : myCollection)
4     // instructions block
```

Iterate : first solution

```
1  List<Gaul> villager = new ArrayList<Gaul>();
2  villagers.add(new Gaul("Asterix"));
3  villagers.add(new Gaul("Cétaumatix"));
4  villagers.add(new Gaul("Agecanonix"));
5  villagers.add(new Gaul("Ordralfabétix"));
6
7  for (Gaul g: villagers)
8    System.out.println(g);
```

Iterate : second solution

Using a dedicated object called an `Iterator`.

we call the `iterator()` method that is part of the `Iterator` interface

```
1  public interface Iterator<E> {
2     boolean hasNext();
3     E next();
4     void remove(); //optional
5  }
```

- `hasNext()` tells wether there are more elements
- `next()` takes the next element (and we cannot go back or ask this element again !)
- `remove()` remove the element from the collection

uses :

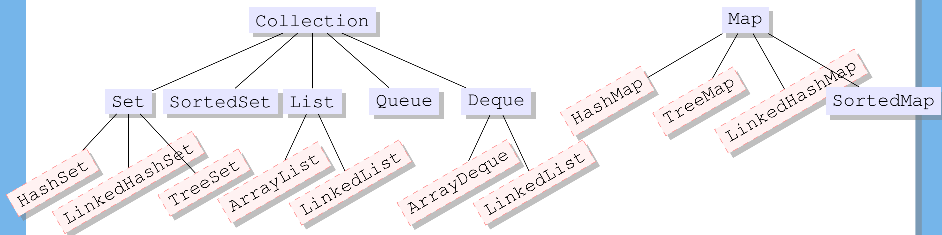
- remove elements
- going over several collections in parallel.

Iterate : second solution

```
1   List<Gaul> villager = new ArrayList<Gaul>();
2   villagers.add(new Gaul("Asterix"));
3   villagers.add(new Gaul("Cétaumatix"));
4   villagers.add(new Gaul("Agecanonix"));
5   villagers.add(new Gaul("Ordralfabétix"));
6   villagers.add(new Gaul("Bonemine"));
7
8   Iterator<Gaul> it = villagers.iterator();
9   while (it.hasNext()){
10    Gaul g = it.next();
11    if (g.getName().equals("Asterix"))
12        it.remove();
13    else
14        System.out.println(g);
15  }
16
17
```

Implementations

There are more than one implementation for each of the interfaces



Map

a map is a binary relation that maps a key to a value.

each key is unique, but a value can be associated to multiple keys.

Warning, `Map` does not implements `Iterable`, so we cannot iterate a `Map` using a `for each` loop !
But we can access the list of keys, values, or pairs as follows :

- `Set<K> keySet()`
- `Set<Map.Entry<K,V» entrySet()`
- `Collection<K> values()`

`Map.Entry` is an inner class (we can define a class inside a class, so as to have a specific tool, but we will not go into the details in this course)

Exemple

```
1  Map<Character,Region> origins = new HashMap<>();
2  ...
3  for (Map.Entry<Character,Region> pair: origins.entrySet()){
4     Character p = pair.getKey();
5     Region r = pair.getValue();
6     if (r.getName.equals("Iberians"))
7       System.out.println(p);
8   }
```

We go over each element of the map, but we print if only if the character is from Portugal or Spain.