

Java 101 - Magistère BFA

Lesson 4: Generic Type and Collections

Stéphane Airiau

Université Paris-Dauphine

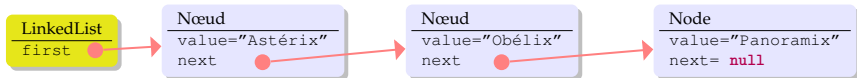
Linked List

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

```
1 public class LinkedList {
2     Node first;
3
4     public LinkedList () {
5         premier = null;
6     }
7
8     public void add (String val) {
9         Node newNode = new Node (val);
10        if (first == null)
11            first = newNode;
12        else {
13            Node last = first;
14            while (last.next != null)
15                last = last.next;
16            last.next = newNode;
17        }
18    }
19 }
```

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 Characters.

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 Characters.
- Modify our Node class by replacing String with Object.

Let us build a list of Characters.

- Create two classes : one for a node containing a Character, the other for a linked list of Characters.
- 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 Characters.
- 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 Characters.
- 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     private E value;
3     private 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 }
```



```
1 public class LinkedList<E> {
2     private Node<E> first;
3
4     public LinkedList () {
5         first = null;
6     }
7
8     public void add(E val) {
9         Node<E> newNode = new Node<E>(val);
10        if (first == null)
11            first = newNode;
12        else {
13            Node<E> last = first;
14            while (last.getNext () != null)
15                last = last.getNext ();
16            last.setNext (newNode);
17        }
18    }
19
20    public E get (int index) {
21        int i=0;
22        Node<E> current=first;
23        while (current.getNext () != null && i<index) {
24            i++;
25            current = current.getNext ();
26        }
27        if (index == i) // we found ith element
28            return current;
29        else
30            return null;
31    }
32 }
```

Use

```
1 Gaul asterix =
2         new IndomitableGaul ("Astérix");
3 Gaul 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

```
1 | LinkedList<Integer> myList = new LinkedList<Integer> ();
2 | //old style
3 | myList.add(new Integer(7));
4 | Integer seven = myList.get(1);
5 | System.out.println(seven.intValue());
6 | //new style
7 | myList.add(6);
8 | 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

```
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?

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 $C<F>$

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 valueDefault;  
3         error!!  
4     public static void setDefault (V value) {valueDefault=value;}  
5         error!!  
6 }
```

- 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 before the return type and after 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

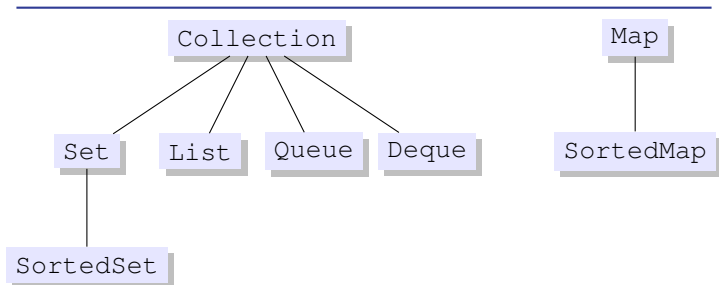
lists, sets, queues are “things” that gather together different objects in one entity

- 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 ?

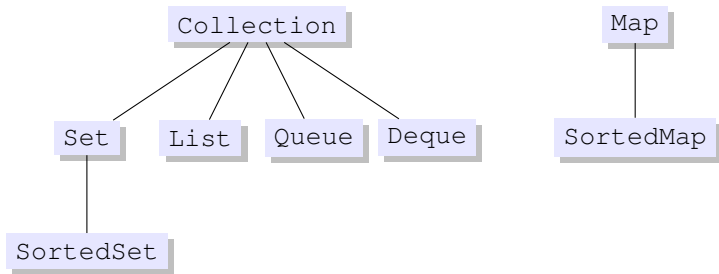
R : ➡ use an interface hierarchy

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.

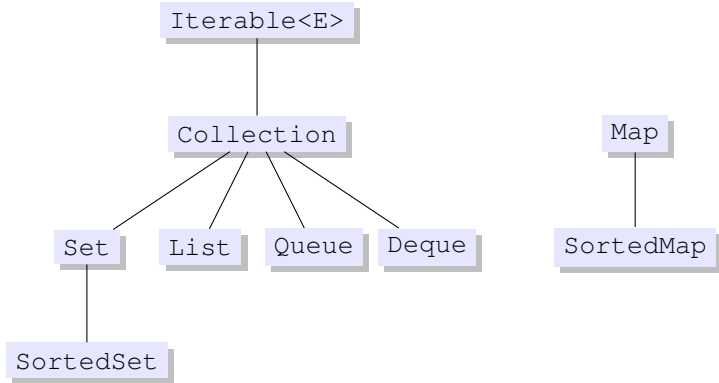
Hierarchie d'interfaces



- **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 Gauls, Integers, Strings, 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> villagers = 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 whether 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 :

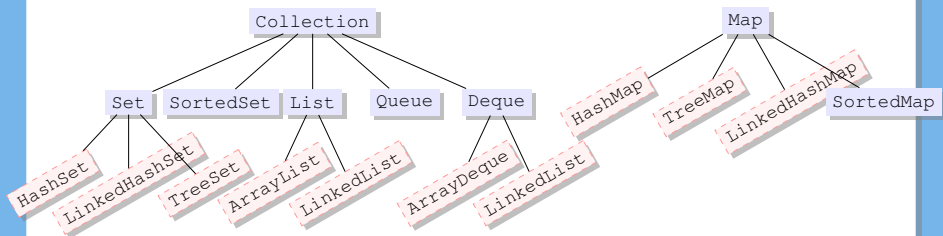
- remove elements
- going over several collections in parallel.

Iterate : second solution

```
1 List<Gaul> villagers = 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



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<V> 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)

K	<code>getKey ()</code> Returns the key corresponding to this entry.
V	<code>getValue ()</code> Returns the value corresponding to this entry.

Example

```
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.

Notion of Order between Objects

`Comparable` is an interface that contains a unique method :

```
public int compareTo(T o)
```

This method returns

- a negative integer when the current object is "smaller" than the object passed in parameter
- 0 when the two objects are "equally big"
- a positive integer when the current object is "larger" than the object passed in parameter.

Many classes such as `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` implement the interface `Comparable`.

When implementing a class, similarly to wondering whether to code methods such as `equals`, `clone`, `toString`, one can also wonder whether the class should implement the interface `Comparable`.

Example

```
1 public class Gaul extends Character
2     implements Comparable<Gaul>{
3     String name;
4     int numBoarsEaten;
5     ...
6
7     public int compareTo(Gaul ixis) {
8         return this.numBoarsEaten - ixis.numBoarsEaten;
9     }
10 }
```

Note that the interface uses generic types. By writing `Comparable<Gaul>`, we make it clear that we can compare with instances of the type `Gaul` and its subtypes.

Let's sort, actually, let us make java sorts things for us!

There is a java class called `Collections` that contains many methods for manipulating `Collections` (in particular, `Lists`, `Vectors`, etc).

In particular, there is a method for sorting, so one does not need to implement a sort method! Note that is a static method of the class `Collections`.

```
public static <T extends Comparable<? super T> void sort (List<T> list)
```

We want to sort a `List` of `T`. It should not be any `T`, as we need an ordering. So the signature specifies that `T` must implement `Comparable<? super T>`. The use of `? super` of `T` allows using the order of a parent type, for example using the `compare` method of `Gaul` for comparing `IndomitableGaul`.

```
1 List<Gaul> l = new LinkedList<> ();
2 l.add(new Gaul ("Astérix", 52));
3 l.add(new Gaul ("Obélix", 365));
4 l.add(new Gaul ("Getafix", 12));
5 System.out.println(l);
6 Collections.sort(l);
7 System.out.println(l);
```

One order may not be enough! Use the interface `Comparator`

Java proposes another way to make an order. After all, there is no unique way to sort objects! Think about students, sometimes we want to sort them by alphabetical orders, sometimes by grades, etc...). Java proposes another interface.

```
1 public interface Comparator<T> {  
2     int compare(T o1, T o2);  
3 }
```

Here, the idea is to create a class that will represent an order. To make that explicit, the class will need to implement the interface `Comparator` and the type parameter allows to express the type that will be compared!

suppose our `Character` class has an attribute `int height`.

```
1 public class OrderingHeight implements Comparator<Character> {  
2     public int compare(Character left, Character right) {  
3         if (left.height < right.height)  
4             return +1;  
5         else if (left.height == right.height)  
6             return 0;  
7         else  
8             return 1;  
9     }  
10 }
```

Sort, second way

The `Collection`s class has another useful method for sorting that uses the `Comparator` interface :

```
public static <T> void sort (List<T> list, Comparator<? super T> c)
```

It is again a `static` method. Again, the method will sort a `List<T>`, but there are no constraints on the parameter `T`. However, there is a second parameter that is the order between elements of the type `T`! This parameter must implement the interface `Comparator<? super T>` :

- it must implement `Comparator` as it must be an order!
- the instance must compare `<? super T>` so that an ordering notion of a parent class can be used (again, I can use an order for the class `Character` to order `Gaul`).

Example

```
1 public static void main(String[] args) {
2     Character obelix = new IndomitableGaul("Obelix", 1.81);
3     Gaul asterix = new IndomitableGaul("Astérix", 1.60);
4     Character cesar = new Character("César", 1.75);
5
6     List<Character> characters = new ArrayList<Character>();
7
8     characters.add(asterix);
9     characters.add(obelix);
10    characters.add(cesar);
11
12    for (Character p: characters)
13        System.out.println(p.presentation());
14
15    Comparator<Character> orderingHeight = new OrderingHeight();
16    Collections.sort(characters, orderingHeight);
17
18    for (Character p: characters)
19        System.out.println(p.presentation());
```