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;
  }
}
```

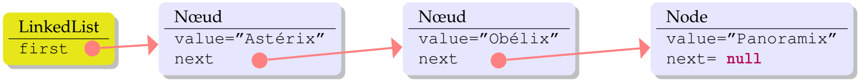
```java
public class LinkedList {
  Node first;

  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 `Characters`.

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`s.
- 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 `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** ?

```java
public class Node<E> {
   E value;
   Node<E> next;

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

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

```java
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

```java
IndomitableGaul asterix =
                new IndomitableGaul("Astérix");
IndomitableGaul obelix =
                new IndomitableGaul("Obélix");
Gaul Informatix = new Gaul("Informatix");
LinkedList<Gaul> list = new LinkedList<Gaul>();
list.add(asterix);
list.add(obelix);
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

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

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

Inheritance of the parameters - use as bounds

```
class <class name> < parameter 1 extends <super class name> >

 { ... }
```

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

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

Some subtleties

```java
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

```java
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 ?

```java
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

```java
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 ?

```java
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.

```java
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

Jockers & bounds

upper bound

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

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

✗we cannot use a `LinkedList<Gaul>`

```
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.

```
public class Collections {
  public static <T> void copy
        (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.

```
public class Paire<C,V>{
  private static V valueDefaut;
      error!!
  public static void setDefaut(V value){valueDefaut=value;}
      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 before the return type and after the visibility (`public, private`) and (`static`).

```
public class ArrayUtil {
  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

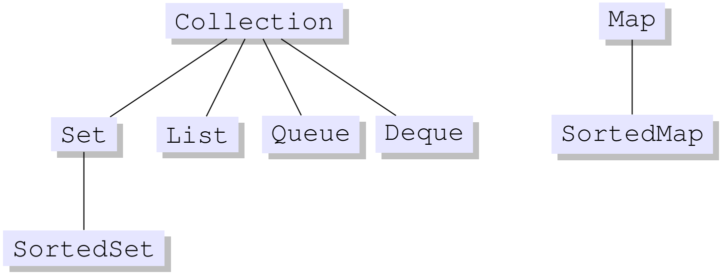
lists, sets, queues are “things” that gather together different objects in one entity

- 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 ?

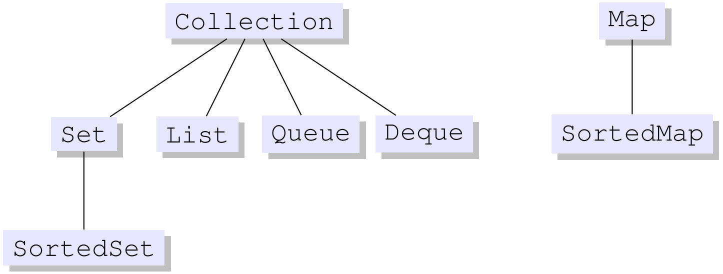
R : ➡ use an interface hierarchy

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.

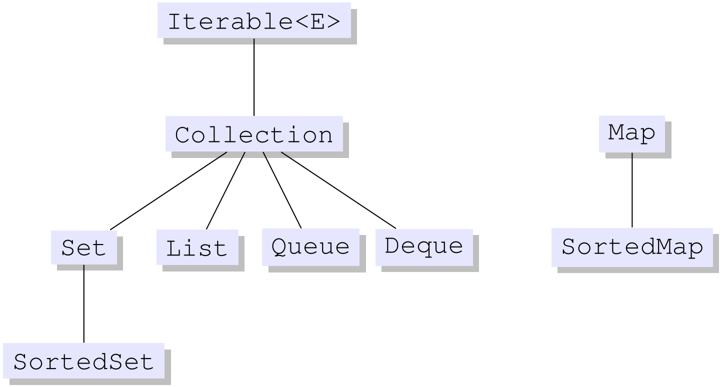
Hierarchie d'interfaces



- `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 !).

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

Iterate : first solution

```java
List<Gaul> villager = new ArrayList<Gaul>();
villagers.add(new Gaul("Asterix"));
villagers.add(new Gaul("Cétaumatix"));
villagers.add(new Gaul("Agecanonix"));
villagers.add(new Gaul("Ordralfabétix"));

for (Gaul g: villagers)
   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

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

- `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

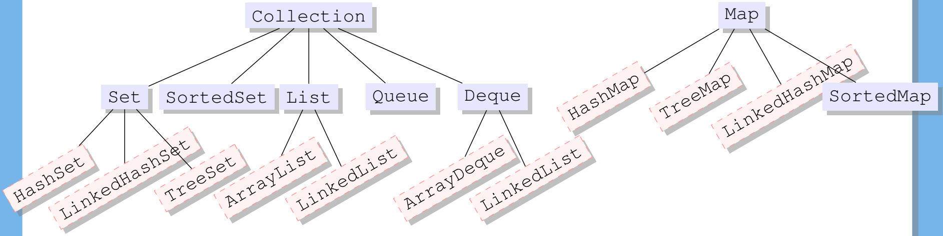
```java
List<Gaul> villager = new ArrayList<Gaul>();
villagers.add(new Gaul("Asterix"));
villagers.add(new Gaul("Cétaumatix"));
villagers.add(new Gaul("Agecanonix"));
villagers.add(new Gaul("Ordralfabétix"));
villagers.add(new Gaul("Bonemine"));

Iterator<Gaul> it = villagers.iterator();
while (it.hasNext()){
   Gaul g = it.next();
   if (g.getName().equals("Asterix"))
       it.remove();
   else
       System.out.println(g);
 }


```

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)

Example

```java
Map<Character,Region> origins = new HashMap<>();
...
for (Map.Entry<Character,Region> pair: origins.entrySet()){
  Character p = pair.getKey();
  Region r = pair.getValue();
  if (r.getName.equals("Iberians"))
    System.out.println(p);
 }
```

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 :

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

This method returns

- a negative interger when the current object is "smaller" than the object passed in parameters paramètre
- 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

```java
public class Gaul extends Character
                implements Comparable<Gaul>{
  String name;
  int numBoarsEaten;
    ...

  public int compareTo(Gaul ixis) {
    return this.numBoarsEaten - ixis. numBoarsEaten;
  }
}
```

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`.

```java
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`.

```java
List<Gaul> l = new LinkedList<>();
l.add(new Gaul("Astérix", 52));
l.add(new Gaul("Obélix", 365));
l.add(new Gaul("Getafix", 12));
System.out.println(l);
Collections.sort(l);
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.

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

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`.

```java
public class OrderingHeight implements Comparator<Character> {
    public int compare(Character left, Character right){
      return left.height < right.height ? -1:
            (left.height== right.height ? 0 : 1);
    }
 }
```

Sort, second way

The `Collections` class has another usefull 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

```java
public static void main(String[] args){
   Character obelix = new IndomitableGaul("Obelix", 1.81);
   Gaul asterix = new IndomitableGaul("Astérix", 1.60);
   Character cesar = new Character("César", 1.75);

   List<Character> characters = new ArrayList<Character>();

   characters.add(asterix);
   characters.add(obelix);
   characters.add(cesar);

   for (Character p: characters)
     System.out.println(p.presentation());

   Comparator<Character> orderingHeight = new OrderingHeight();
   Collections.sort(character, orderingHeight);

   for (Character p: characters)
     System.out.println(p.presentation());
```