Introduction à Java

Cours 6: Généricité et Collections

Stéphane Airiau

Université Paris-Dauphine

Liste chainée

```java
public class Noeud {
   String valeur;
   Noeud suivant;

   public Noeud(String val){
     valeur = val;
   }

   public void setSuivant(Noeud next){
     suivant = next;
   }
}
```

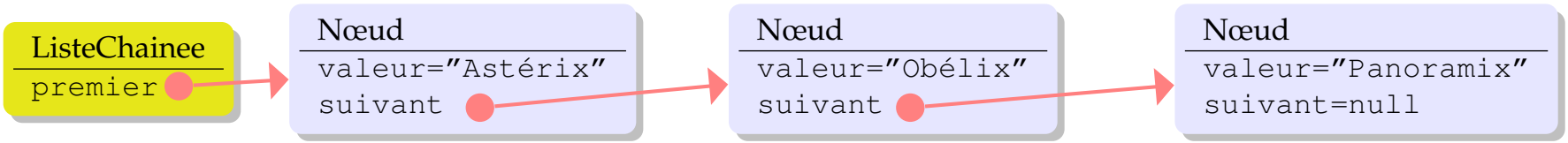
```java
public class ListChainee {
   Noeud premier;

   public ListeChainee(){
     premier = null;
   }

   public void add(String val){
     Noeud nouveau = new Noeud(val);
     if (premier == null)
       premier = nouveau;
     else {
       Noeud dernier = premier;
       while(dernier.suivant != null)
         dernier = dernier.suivant;
       dernier.suivant = nouveau;
     }
   }
}
```

Exemple

```
{"Astérix","Obélix","Panoramix"},
```



Généricité

On voudrait maintenant faire une liste de `Personnages`

- changer la classe `Noeud` et la classe `ListeChainee` pour faire deux classes spécialisées `NoeudPersonnage` et `LCPersonnage`.
- mettre `Object` à la place de `String` et faire une liste chainée d'`Object`.

⇨ possible mais nécessitera des transtypages explicites

- et si on pouvait ajouter un paramètre à la classe `Noeud`... c'est ce que `Java` propose

```
1  public class Noeud<E> {
2     E valeur;
3     Noeud<E> suivant;
4
5     public Noeud(E val){
6       valeur = val;
7     }
8
9     public void setSuivant(Noeud<E> next){
10      suivant = next;
11    }
12 }
```

```java
public class ListChainee<E> {
  Noeud<E> premier;

  public ListeChainee(){
    premier = null;
  }

  public void add(E val){
    Noeud<E> nouveau = new Noeud<E>(val);
    if (premier == null)
      premier = nouveau;
    else {
      Noeud<E> dernier = premier;
      while(dernier.suivant != null)
        dernier = dernier.suivant;
      dernier.suivant = nouveau;
    }
  }

  public E get(int index){
    int i=0;
    Noeud<E> courant=premier;
    while(courant.suivant != nll && i<index){
      i++;
      courant = courant.suivant;
    }
    if(index == i) // on a trouvé l'élément numéro i
      return courant;
    else
      return null;
    }
  }
}
```

Utilisation

```
1 IrreductibleGaulois asterix =
2                 new IrreductibleGaulois("Astérix");
3 IrreductibleGaulois obelix =
4                 new IrreductibleGaulois("Obélix");
5 Gaulois Informatix = new Gaulois("Informatix");
6 ListeChainee<Gaulois> liste = new ListeChainee<Gaulois>();
7 liste.add(asterix);
8 liste.add(obelix);
9 liste.add(informatix);
```

- Attention, le paramètre de type ne peut pas être un type primitif (ex `int`, `char`, `double`, etc...)
  ex : `Noeud<int>` n'est pas permis.
- Lors de l'appel au constructeur, on n'est pas obligé de répéter les paramètres (mais il ne faut pas oublier les chevrons)
  ex :`ListeChainee<Gaulois> liste = new ListeChainee<>();`
  `Java` va inférer les paramètres

Méthodes génériques

On peut aussi créer des méthodes génériques : des méthodes avec un paramètre de type.

Quand on déclare une méthode générique, le paramètre de type est déclaré avant le type de retour et après la portée (`public, private`) et l'indication d'une méthode de classe (`static`).

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

Quand on appelle une méthode générique, on n'a pas besoin spécifier le paramètre de type, il est inféré par `Java`.
ex : `ArrayUtil.swap(villageois, 2, 6);`

Si on y tient, on peut quand même donner le type (cela permettra un meilleur message d'erreur si quelque chose se passe mal).
ex : `ArrayUtil.<Gaulois>swap(villageois, 2, 6);`

Autoboxing

Maintenant que Java peut connaître le type des objets contenus dans une structure, Java offre des possibilités pour simplifier le code : par exemple la transformation automatique dasn des types primitifs

```
1 ListeChainee<Integer> maListe = new ListeChainee<Integer>();
2 //old style
3 maListe.add(new Integer(7));
4 Integer sept = maListe.get(1);
5 System.out.println(sept.intValue());
6 //new style
7 maListe.add(6);
8 int six = maListe.get(2);
```

Types paramétrés

On peut avoir deux paramètres.

```
class <nom classe> < paramètre 1 [, paramètre 2] >{
   //on peut déclarer des attributs des classes paramètres
  <paramètre 1> attribut;
  //on peut déclarer des méthodes qui retournent l'attribut
  <paramètre 1> <nom méthode>(<liste arguments>){ ... }
   ...
}
```

Paramètre & Héritage

Héritage au niveau de la classe

```
class <nom classe> < paramètre 1 [, paramètre 2] >
      extends <classe mère> < paramètre 1 [, paramètre 2] >
 { ... }
```

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

Héritage au niveau des paramètres (borne de types)

```
class <nom classe> < paramètre 1 extends <classe mère>[,
        paramètre 2] >
 { ... }
```

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

On indique ici que le type `E` doit hérité du type `Personnage`.

On peut avoir plusieurs bornes, mais cela devient plus avancé.

les types génériques sont invariants

```
1 | ListeChainee<Gaulois> lg = new ListeChainee<Gaulois>;
2 | ListeChainee<Personnage> lp = lg;
```

Sur la ligne 2, on a envie de dire qu'une liste de `Gaulois` est une liste de `Personnages`. Cependant...

```
3 | lp.add(new Personnage("Jules César"));
4 | Gaulois g = lg.get(1);
```

quand on récupère un élément via la liste `lg`, on ne récupère pas un `Gaulois` !
Le compilateur `Java` ne permettra pas la ligne 2.

⇨ si `F` est une classe de la descendance de la classe `M`
si `G` est une classe générique,
`G<F>` n'est pas dans la descendance de `G<M>`
Autrement dit, il n'y a pas de relation entre `G<F>` et `G<M>`

Jockers

Java offre la possibilité d'utiliser des « jockers » qui va servir à exprimer un type inconnu.

```
1 | ListeChainee<?> list = new ListeChainee<Gaulois>();
```

- On ne pourra pas utiliser la méthode `add` car on devrait passer un élément de type `?`,
- on peut appeler une méthode comme `get`
  ⇨ transtypage avec `Object`.
  Pour être utile, on va utiliser le jocker avec une borne (inférieure ou supérieure)

Jockers & bornes

avec borne supérieure

`ListChainee<? extends Gaulois>` pour dire que le type inconnu doit être dans la descendance de la classe `Gaulois`.

```
1 | public void presentezVous(ListeChainee<Personnage> liste){
```

✘on ne peut pas utiliser une liste `ListeChainee<Gaulois>`

⇨ on utilise :

```
1 | public void presentezVous(ListeChainee< ? extends Personnage> liste){
```

avec borne inférieure

le jocker est bornée par une borne inférieur, le type inconnu doit être un parent du paramètre T.

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

Dans l'exemple, la borne est un type paramétré (c'est méchant !)

Quelques restrictions

- on ne peut pas utiliser de types primitifs comme paramètres
- on ne peut pas construire un tableau avec des types paramétrés ex : Noeud<Gaulois>[] tableauDeNoeuds = **new** Noeud<Gaulois>[10] ; n'est **pas** permis.
- le paramètre d'une classe ne peut pas être utilisé dans un contexte `static`.

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

- et d'autres subtilités dont on ne parlera pas ici

Collections

Collections

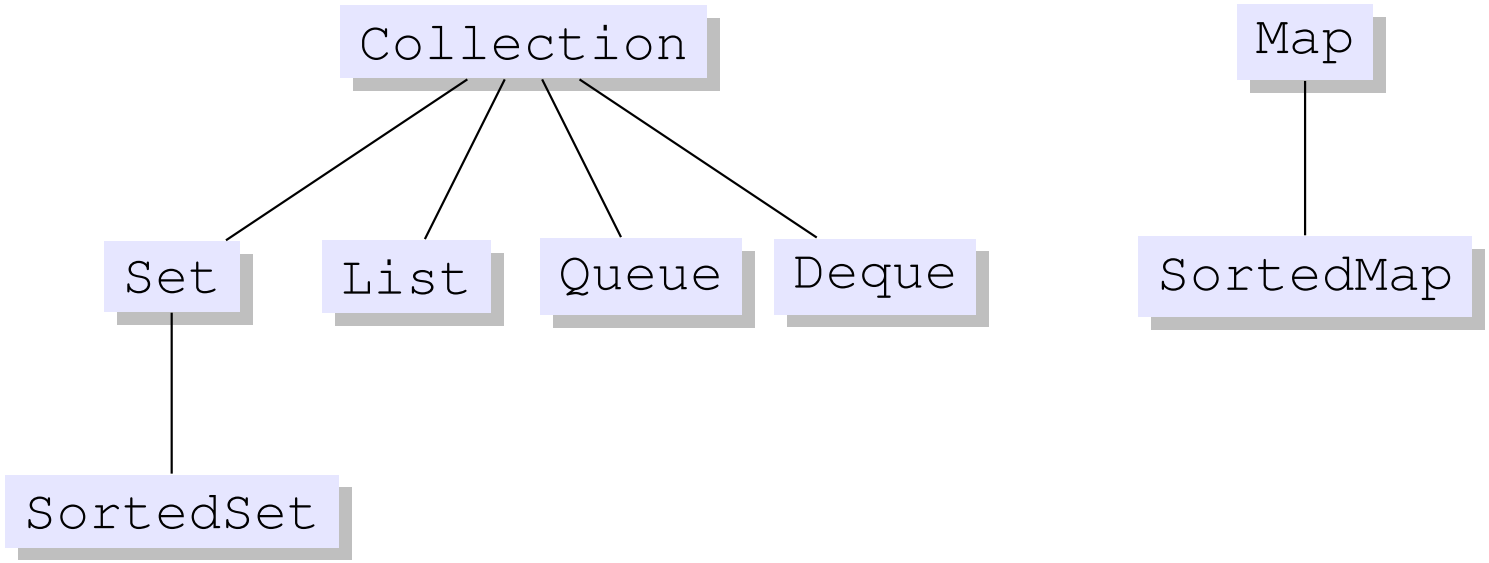
Les listes, les ensembles, les piles, les files d'attente sont des objets qui regroupe plusieurs éléments en une seule entité.

- en commun :
  - mêmes questions : est-ce qu'elles contiennent des éléments ? combien ?
  - mêmes opérations : on peut ajouter ou enlever un élément à la structure, on peut vider la structure. On peut aussi parcourir les éléments contenus dans la structure.
- implémentations différentes

Q : Comment peut-on manipuler toutes ces structures ?

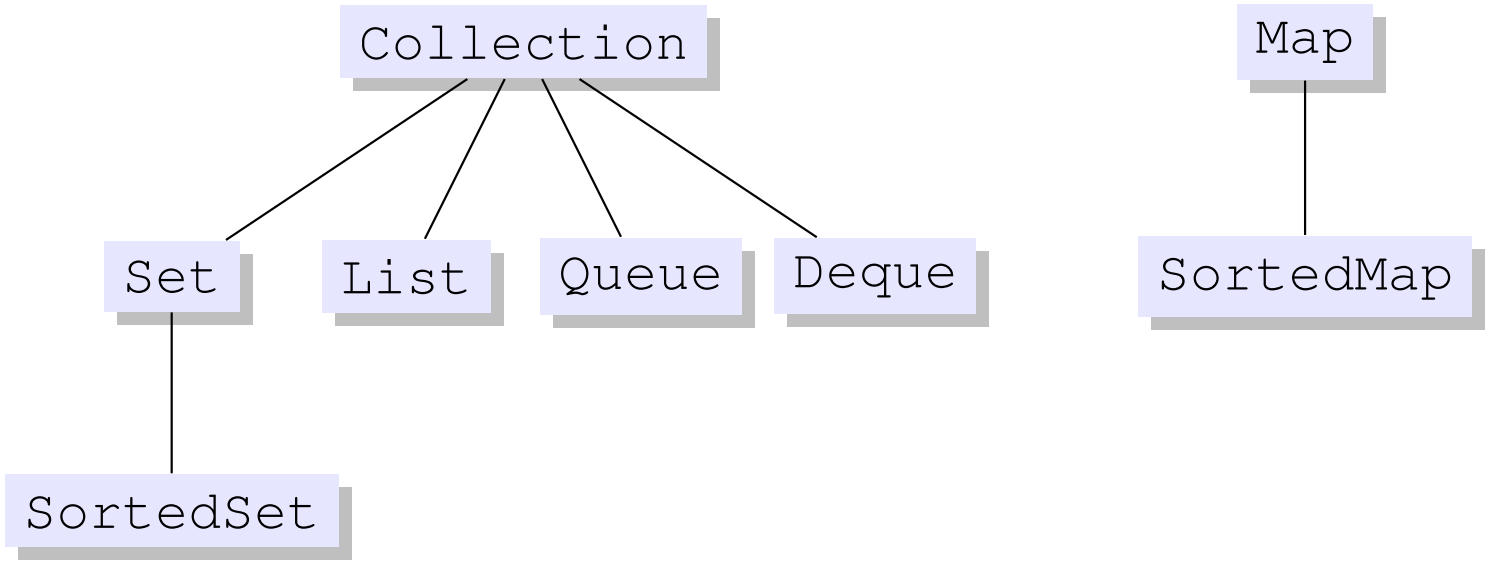
R : ⇨ utiliser une hierarchie d'interfaces

Hierarchie d'interfaces



- `Collection` : méthodes de base pour parcourir, ajouter, enlever des éléments.
- `Set` : cette interface représente un ensemble, et donc, ce type de collection n'admet aucun doublon.
- `List` : cette interface représente une séquence d'éléments : l'ordre d'ajout ou de retrait des éléments est important (doublons possibles)
- `Queue` : file d'attente : il y a l'élément en tête et il y a les éléments qui suivent. L'ordre d'ajout ou de retrait des éléments est important (doublons possibles)
- `Deque` : cette interface ressemble aux files d'attente, mais les éléments

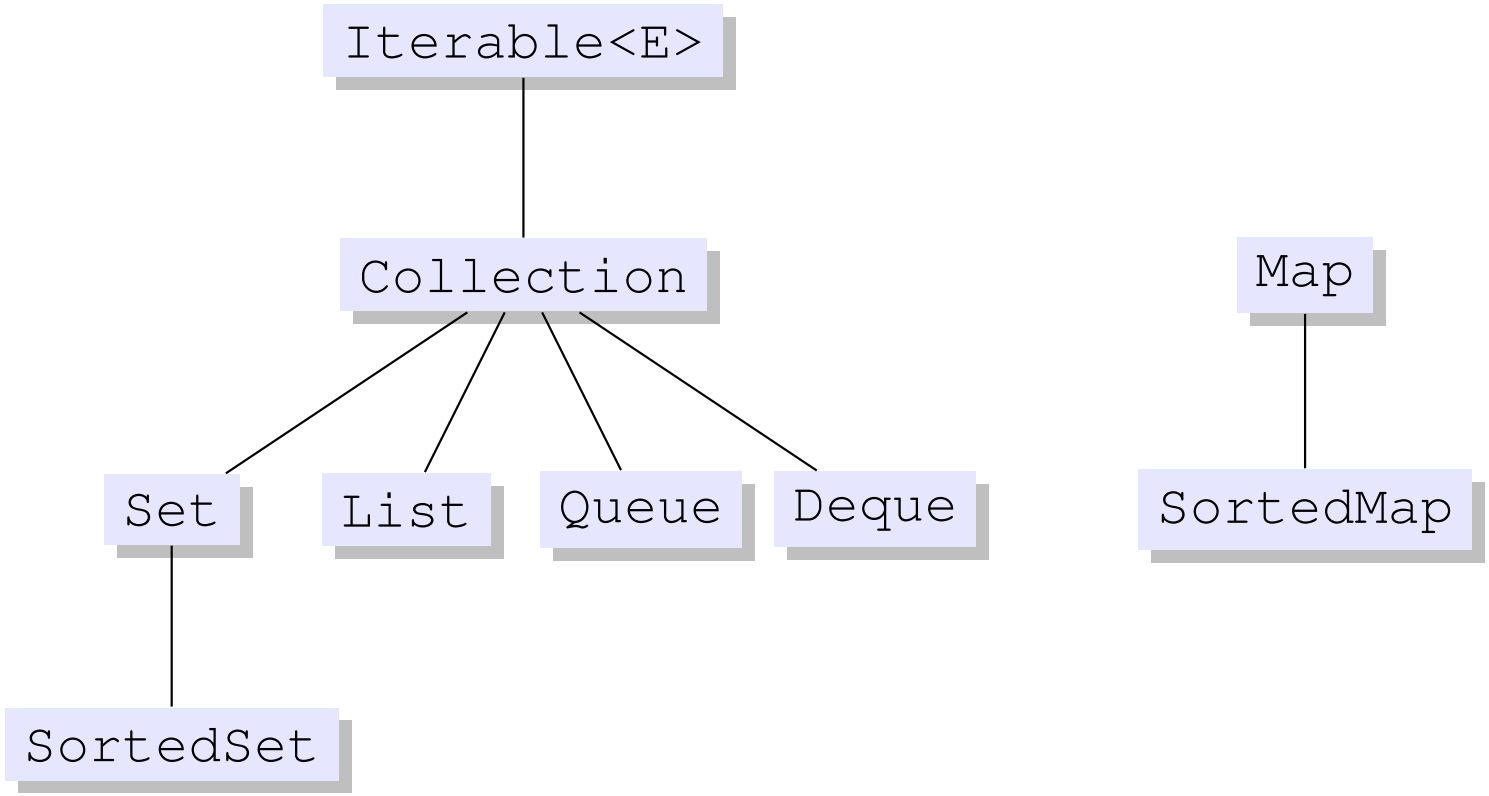
Hierarchie d'interfaces



- `Map` : cette interface représente une relation binaire (surjective) : chaque élément est associé à une clé et chaque clé est unique (mais on peut avoir des doublons pour les éléments).
- `SortedSet` est la version ordonnée d'un ensemble
- `SortedMap` est la version ordonnée d'une relation binaire ou les clés sont ordonnées.

Ces interfaces sont génériques, i.e. on peut leur donner un paramètre pour indiquer qu'on a une collection de `Gaulois`, de `Integer`, de `String`, etc...

Parcourir une collection



On peut utiliser une boucle “**for each**” sut tout objet implémentant l’interface `Iterable`.

Parcourir une collection : première solution

En utilisant la généricité ⇨ le compilateur va connaître le type des éléments qui sont contenus dans la collection.

- **Situation :** on a une collection `maCollection` qui contient des objets de type `E`.
- On va accéder à chaque élément de la collection `maCollection` en utilisant le mot-clé **`for`**,
- chaque élément sera stocké dans une variable `<nom>` de type `E` (évidemment).

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

Parcourir une collection : première solution

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

Parcourir une collection : deuxième solution

Utilisation d'un objet dédié au parcourt d'éléments dans une collection : un objet qui implémente l'interface `Iterator`.

Obtention : appel à la méthode `iterator()` (interface `Iterable`)

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

- `hasNext()` retourne un `boolean` qui indique s'il reste des éléments à visiter,
- `next()` donne accès à l'élément suivant,
- `remove()` permet d'enlever l'élément de la collection.

utilisation :

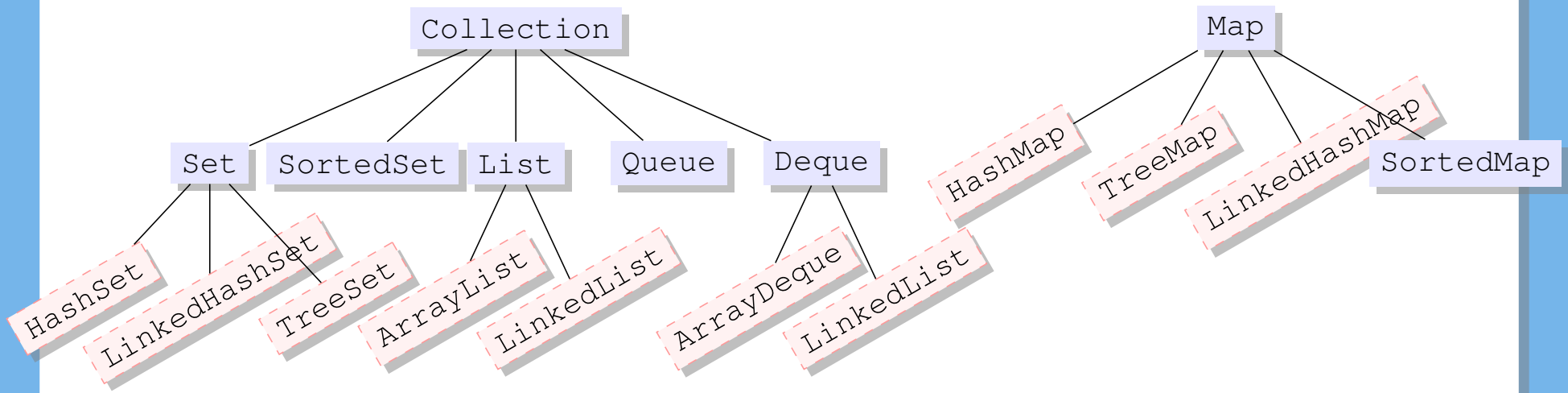
- enlever un élément lors de l'itération
- parcourt de plusieurs collections en parallèle.

Parcourir une collection : deuxième solution

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

Implémentations

Pour chacune des interfaces, il existe plusieurs implémentations



Map

un `Map` représente une relation binaire : chaque élément d'un `Map` est une paire entre une clé et une valeur.

Dans un `Map`, chaque clé est unique, mais on peut avoir des doublons pour les valeurs.

Attention, `Map` n'est pas une sous interface de `Iterable`, donc on ne peut pas parcourir un `Map` avec une boucle `for each` !
On peut obtenir l'ensemble des clés, l'ensemble des valeurs, et l'ensemble des paires (clé,valeur) grace aux méthodes suivantes :

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

`Map.Entry` désigne une classe `Entry` qui est interne à la classe `Map`. On peut créer des classes à l'intérieur de classes, mais je n'en parlerai pas plus aujourd'hui.

Exemple parcours d'une `Map`

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

Dans cet exemple, on part une `Map` qui associe à chaque personnage sa région d'origine et on affiche seulement les personnages qui sont des ibères.

Ordre

Interface `Comparable` contient une seule méthode :

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

Cette méthode retourne

- un entier négatif si l'objet est plus petit que l'objet passé en paramètre
- zéro s'ils sont égaux
- un entier positif si l'objet est plus grand que l'objet passé en paramètre.

les classes `String`, `Integer`, `Double`, `Date`, `GregorianCalendar` et beaucoup d'autres implémentent toutes l'interface `Comparable`.

Exemple

```
1  public class Gaulois extends Personnage
2                  implements Comparable<Gaulois>{
3    String nom;
4    int quantiteSanglier;
5      ...
6
7    public int compareTo(Gaulis ixis) {
8      return this.quantiteSanglier - ixis. quantiteSanglier;
9    }
10 }
```

interface `Comparator`

ex : trier les éléments d'une collection : utilisation interface `Collections`

```
   // interface Collections
1  public static <T extends Comparable<? super T»
2                  void sort(List<T> list)
3  public static <T> void sort
4                  (List<T> list, Comparator<? super T> c)
```

```
1  public interface Comparator<T> {
2     int compare(T o1, T o2);
3     boolean equals(Object obj);
4  }
```

Pour comparer des `Gaulois`, et même tous les `Personnage` selon leur taille, on peut écrire la classe suivante :

```
1  public class TriHauteur implements Comparator<Personnage> {
3      public int compare(Personnage gauche, Personnage droit){
4        return gauche.hauteur < droite.hauteur ? -1:
5              (gauche.hauteur == droite.hauteur ? 0 : 1);
5      }
6    }
```

Exemple

Ensuite, on peut utiliser cette nouvelle classe pour trier des `Personnages` selon leur taille.

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

Exercice

```
1  public class Pays{
2     private String nom;
3     private long pop;
4
5     public Pays(String nom, long pop){
6       this.nom=nom; this.pop=pop;
7     }
8     public String getNom()return nom;
9     public long getPopulation()return pop;
10  }
```

Créez une classe `InfoPays` qui gère une liste de pays. Implémentez

- `public void affiche(String nom)` : affiche les informations du pays dont le nom est passé en paramètre. Si aucun pays ne correspond dans la liste, écrivez un message.
- `public void ajouterPays(String nomPays, int pop)` : ajoute un pays dont le nom est `nomPays` et la population est `pop`.
- `public void enleverPays(String nom)` : enlève le pays de la liste.
- Afficher les pays en suivant l'ordre alphabétique
- Afficher les pays dans l'ordre de leur population