# Ing39 – Bibliothèque Collections (1)

Virginia Aponte  Pierre Courtieu

# Dans ce cours

- ⋆ Philosophie et organisation de la bibliothèque
- ⋆ Hiérarchie `Collection<E>` et ses parcours
- ⋆ Un mot sur la généricité
- ⋆ Les listes `List<E>` et une implantation : `ArrayList<E>`
- ⋆ Les ensembles non triés `Set<E>` et une implantation.
- ⋆ Les classes utilitaires `Collections` et `Arrays`

Attention :

- ⋆ 1 cours indispensable si vos collections contiennent des types non standard : « Égalité »

# Approche générale



Bibliothèque des Collection : fournir des structures des données à objets, avec même approche « fonctionnalités »

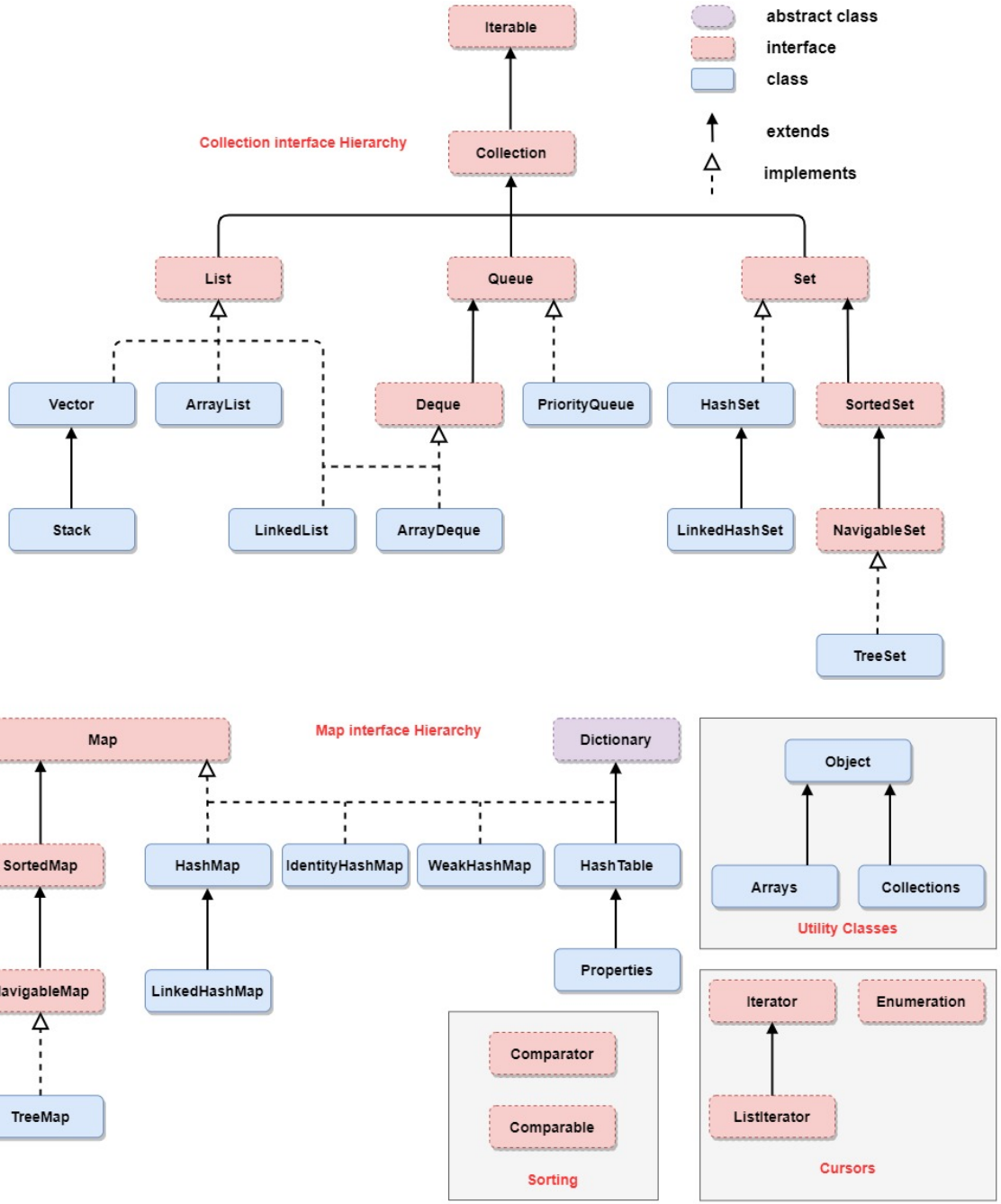
- ⋆ listes, piles, ensembles, files, dictionnaires associatifs
- ⋆ assez riches en opérations ; avec implantations efficaces
- ⋆ génériques, i.e., reutilisables sur tout type d'objet
- ⋆ organisation d'interfaces en sous familles de structures
  - ⋆ sous-ensemble opérations communes
  - ⋆ même sorte de fonctionnalités : interface parcours, vues, bulk operations ...
- ⋆ certaines variantes triées et plusieurs manières spécifier ordre tris

Et aussi, des classes utilitaires

- ⋆ `Arrays` et `Collections` : nombreux algorithmes classiques et efficaces pour manipuler collections et tableaux

# Architecture Collections (https://vidvaan.com/java-collection-framework-overview/)





# Organisation de la bibliothèque



Notion de Collection : conteneur d'objets mettant en œuvre *systématiquement*

- ⋆ 1 organisation de ses éléments + opérations adaptées
- ⋆ opérations parcours du contenu
- ⋆ vues modifiables du contenu pour son parcours/modification
- ⋆ génériques : utilisables sur tous types d'objets

2 hiérarchies indépendantes

- ⋆ `Collection<E>` : collections proprement dites, éléments de type E
  - ⋆ interfaces filles : listes, ensembles, files+ classes implantations
  - ⋆ étend `Iterable<E>` ⇒ toute collection est itérable.
- ⋆ `Map<K,V>` : dictionnaires associatifs (clé(`K`) ↦ valeur (`V`))

# Autres éléments importants de la bibliotèque



- 2 interfaces pour parcours :
  - `Iterable<E>` et `Iterator<E>`
  - implantées par **`Collection`**, mais pas par **`Map`**
- 2 interfaces pour comparer éléments :
  - `Comparable<E>` ou `Comparator<E>`
  - à implanter si conteneur trié (sauf si `E` type standard)
  - c.f. cours « Collections triées »
- 2 opérations à redéfinir dans `E` (si `E` non standard)
  - **`equals()`**, **`hashcode()`** : nécessaires pour fonctionnement correct
  - c.f. cours « egalité »
- 2 classes utilitaires `Collections` + `Arrays` : méthodes statiques de tri, recherche, copie, intéropérabilité avec tableaux ...

# Bref intermède : la généricité

## Généricité

Un code (classes, méthodes) est écrit en termes d'un type quelconque, désigné par une lettre entourée des symboles <>.

- `<E>` : signifie «n'importe quel type objet»
- Ex : la classe `ArrayList<E>` est déclarée de manière à pouvoir stocker des éléments de *type quelconque* E.
- à la création et utilisation ⇒ on remplace E par n'importe quel type objet

Utilisation :

- `new ArrayList<Compte>()` ⇒ E=Compte.
- `new ArrayList<Integer>()` ⇒ E=String, ...

# Généricité en Java

- ⋆ A la déclaration : on peut paramétrer classes, interfaces ...par 1 ou + types génériques
- ⋆ ces paramètres peuvent apparaître dans leurs corps
- ⋆ A l'utilisation : instanciés en fournissant n'importe quel type concret
- ⋆ +clair, +sûr, efficace (pas de cast, erreurs de type détectées compilation).

```
public interface Liste<T> {
    public int size();
    public void add(T el);
    public T get(int index);
}
Liste<Integer> liste1 = ...      // T= Integer
Liste<Compte> liste2 =...        // T= Compte
liste1.add("un texte");          // ne compile pas !!
```

- ⋆ `T` : type quelconque des éléments de la liste ;
- ⋆ les méthodes ont des types génériques :
  ⇒ `add(T el)` ajoute un élément de type `T` dans la liste ;

# Interface `Collection<E>`



`Collection<E>` : aggrégat d'objets **E** avec doublons possibles ; pas d'ordre d'insertion ni de parcours. Pas d'accès sur 1 élément précis.

- ajouter/retirer éléments, par un ou par paquets (*bulk operations*)
- toute collection est itérable (i.e. `Iterable<E>`)
  - accès éléments pendant itération
  - retrait possible pendant itération
  - syntaxe dédiée *for-each*
- étendue par plusieurs sous-hiérarchies :
  - `List<E>`, `Set<E>`, `Queue<E>`
  - où sémantique accès/insertion change
- + autres opérations classiques ...

# Extrait opérations `Collection<E>`



- `int size()`, `boolean isEmpty()`, `boolean contains(Object o)`
- `boolean add(E e)` : false si non ajouté
- `boolean remove(Object e)` : false si non trouvé
- `Iterator<E> iterator()` : objet itérateur pour parcourir/modifier
- `boolean addAll(Collection<? extends E> c)` : ajouter contenu `c`
- `boolean removeAll(Collection<?> c)` : retirer contenu `c`
- `boolean retainAll(Collection<?> c)` : intersection avec contenu `c`
- `<E> E[] toArray(E[] a)` : conversion vers tableau
- `void clear()` : vider contenu

Notez : pas de `get()`

# Exemple d'une collection : ArrayList



```
// création d'une collection de String
Collection<String> coll =  new ArrayList<String>() ;

// ajouts à cette collection
coll.add("un") ;
coll.add("trois") ;
coll.add("un") ;  // doublon

// test appartenance "trois" --> affiche  true
System.out.println(coll.contains("trois"));

// toString() rédéfinit par AbstractCollection
System.out.println(coll.toString());// affiche [un, trois, un]

// ajout collection dans autre (bulk operations)
Collection<String> coll2 =  new ArrayList<String>(coll);

// conversion vers tableau
String [] tab = coll.toArray(new String[] {}) ;
```

# Toute Collection<E> sait fabriquer son itérateur



`Iterable<E>` : étendue par toute **`Collection<E>`** (qui sont toutes itérables)

- ⋆ opération unique
  - ⋆ `Iterator<E> iterator()` : retourne 1 objet itérateur sur éléments collection courante
- ⋆ syntaxe `for-each` : parcours dédiée aux itérables
  - ⋆ utilisable sur collections + tableaux
  - ⋆  pas de modification via cette syntaxe

# Interface Iterator<E>



objet `Iterator<E>` : vue *itérable et modifiable* de la collection courante

- ⋆ contient pointeur vers prochain à visiter
- ⋆ ordre parcours cohérent avec organisation collection sous-jacente
- ⋆ collection courante modifiable pendant parcours (ex : retrait 1 élément )

Opérations :

- ⋆ `boolean` **`hasNext()`** : reste-t-il éléments à visiter ?
- ⋆ **`E next()`** : prochain élément
- ⋆ `void` **`remove()`** : retire de la collection courante dernier obtenu par **`next()`**. Échoue si pas de **`next()`** juste avant.
- ⋆  **`ConcurrentModificationException`** si collection modifiée après création d'un itérateur et avant son utilisation.

# Exemples de parcours d'une collection



```
// Rappel exemple précédent: coll est la liste =  [un, trois, un]

Iterator<String> itColl = coll.iterator();  //  création  iterateur
 // coll.add(1,"deux");  <-- provoquera une erreur + tard
 // si modification de coll APRES création itérateur
 // et avant  son utilisation --> erreur exécution !

// utilisation itérateur: compter les "un" dans coll
int nb=0;
while (itColl.hasNext()) {
    String elem = itColl.next() ;
    if (elem.equals("un")) nb++;
}
System.out.println("Nombre de un: "+n); // affiche: 2

// parcours de coll avec  for each
for (String el : coll) { System.out.print(el+" ") ; }

// for-each utilisable également sur tableaux
int [] tab= {1,2,3};
for (int n: tab){ System.out.print(n+" ") ; }
```

```
Affichages
Nombre de un: 2
un trois un
1 2 3
```

# Itération : ce que l'on peut faire/ne pas faire



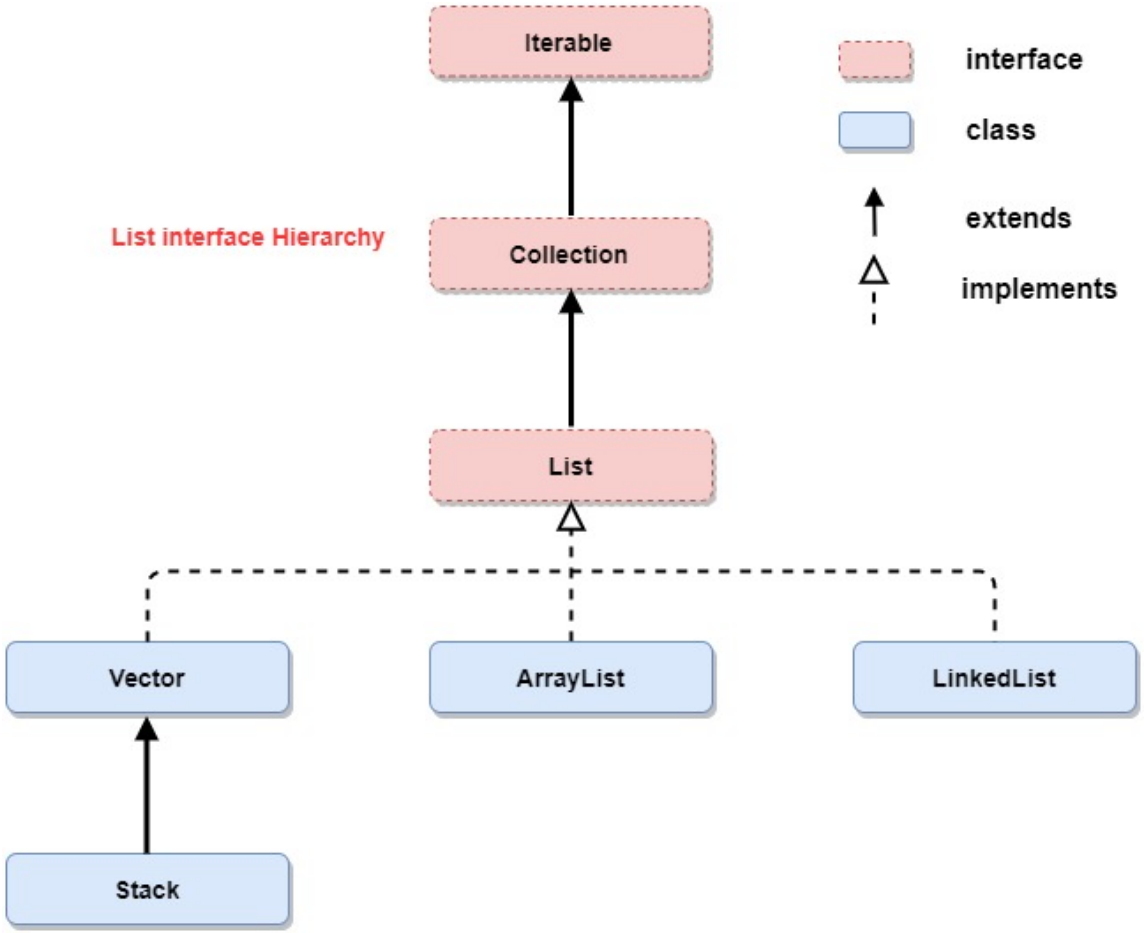
```
// une liste d'entiers
ArrayList<Integer> liste = new ArrayList<Integer>();
Random r = new Random();
for (int i=1; i<=51; i++){ liste.add(r.nextInt(100));  }

// modification avec itérateur: retirer les entiers  entre 10 et 20 : OK!
 Iterator<Integer> it = liste.iterator();
 while(it.hasNext()){
      Integer n = it.next();
      if (n>=10 && n <=20){  it.remove(); }
 }

// Modification avec for-each : erreur à l'exécution
 for (Integer n: liste) {
     if (n>=10 && n <=20) liste.remove(n);
 }
```

\+ modification possible via itérateur : **`next()`** puis **`remove()`**

\- pas de modification tableau/collection via for-each

⋆  **`ConcurrentModificationException`** si collection modifiée après création d'un itérateur et avant son utilisation.

# Hiérarchie `List<E>` (`https://vidvaan.com/java-collection-framework-overview/`)



# Interface `List<E>`



`List<E>` : séquence de E indexée à partir de 0, chaque ajout prend un numéro d'ordre, retrait préserve leur cohérence, itération suit les indices.

Nouvelle opérations sur indices

- `add(int idx, E t)`, `get(int idx)`, `remove(int idx)`, `set(int idx, E t)` …
- indice occurrence : `indexOf(Object o)`, `lastIndexOf(Object o)` …
- vue intervalle d'indices : `subList(int debut, int fin)`
- tri avec un objet Comparator[1] : `sort(Comparator<? super E>)`
- `ListIterator<E>` : itérateur parcours avant/arrière + retrait + ajout + modification éléments.
  - `hasPrevious()`, `previous()`, `hasNext()`, `next()`
  - `nextIndex()`, `previousIndex()`
  - `add(E e)`, `set(E e)`, `remove()`

1. C.f. cours « Collections triées »

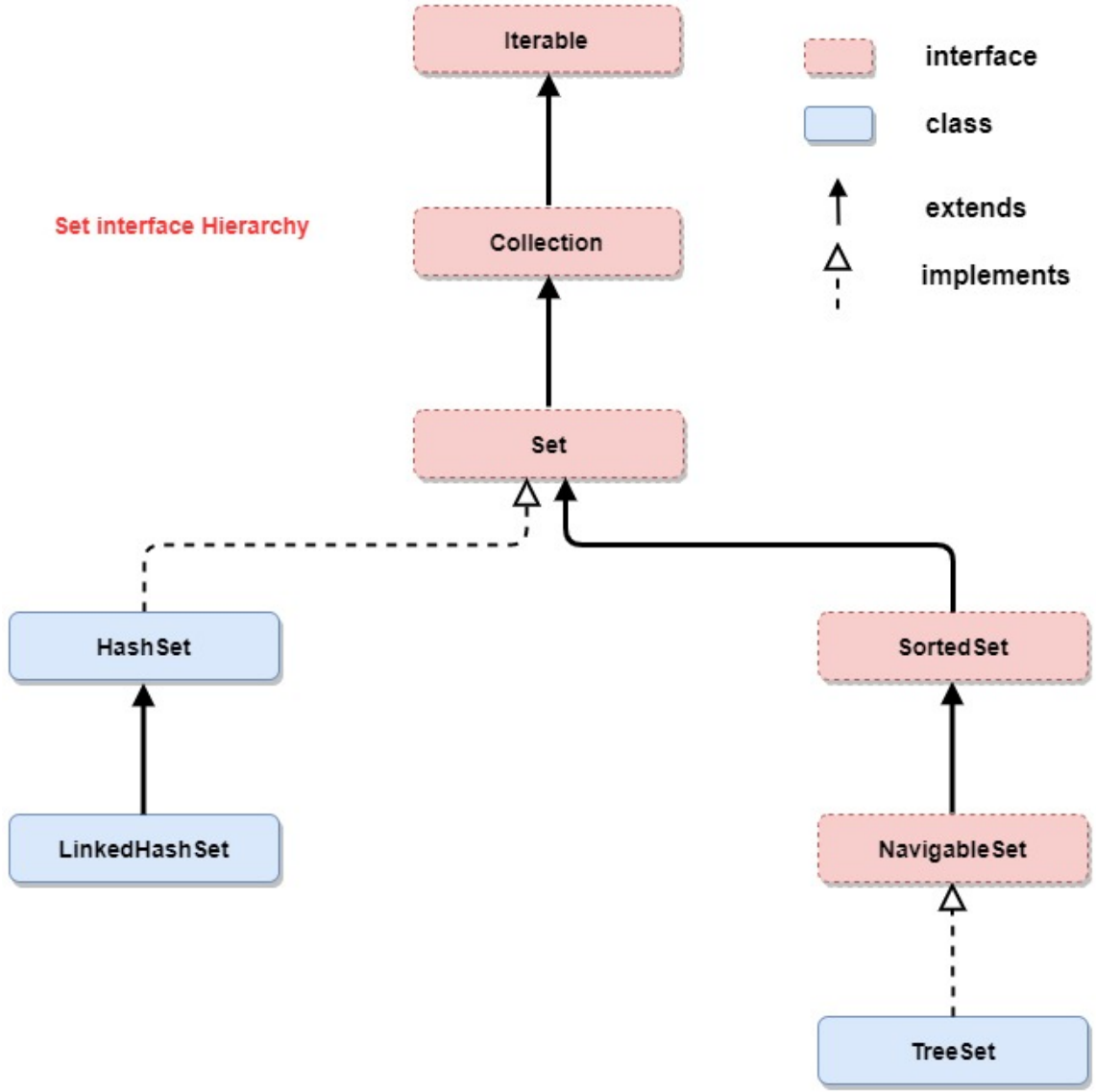
# ArrayList : quelques opérations



- `int size()` : taille de la liste
- `boolean isEmpty()` : true si la liste est vide
- `add(el)` : ajoute el en fin de liste
- `add(i, el)` : ajoute el à l'indice i
- `set(i,el)` : modifie l'élément d'indice i avec el
- `remove(i)` : enlève l'élément d'indice i de la liste
- `remove(obj)` : enlève l'élément obj de la liste
- `get(i)` : renvoie l'élément d'indice i
- `indexOf(obj)` / `lastIndexOf(obj)` : premier/dernier indice de `obj`.
- et beaucoup d'autres (voir la documentation)

# Hiérarchie `Set<E>` (`https://vidvaan.com/java-collection-framework-overview/`)





# Interface Set<E>

**Set<E>** : extension de **Collection<E>** sans doublons, ni ordre pre determiné d'insertion, ni de parcours. Ajout doublon non réalisé et retournant false.

- ⋆ pas d'opérations nouvelles (p.r. à Collection<E>)
- ⋆ **HashSet<E>**, **LinkedHashSet<E>** : basées sur tables hash

```
  Set<String> set =  new HashSet<String>() ;

 // affichons le résultat de plusieurs add
  System.out.println("Ajout un : " + set.add("un")) ;
  System.out.println("Ajout deux : " + set.add("deux")) ;
  System.out.println("Ajout encore un : " + set.add("un")) ; // doublon!!
  System.out.println("Taille du set : " + set.size()) ;

  // Ajout de trois,quatre
  set.add("trois"); set.add("quatre");

  System.out.println(set.toString()); // notez l'ordre d'affichage!

Affichages:
 Ajout un : true
 Ajout deux : true
 Ajoute encore un : false
 Taille du set : 2
 [trois, quatre, un, deux]
```

# Classes utilitaires Arrays et Collections



Méthodes statiques d'opérations classiques avec implantations efficaces.

Arrays : méthodes sur tableaux

- ⋆ tris, recherche dichotomique, copie, initialisation, égalité entre tableaux, ...
- ⋆ Ex : Conversion tableau en liste immuable avec **`Arrays.asList(Object[])`**

Collections : méthodes sur collections

- ⋆ tris, recherche dichotomique, copie, permutation, ...
- ⋆ Comparators en ordre croissant et décroissant, ...

```
Integer [] tab = {1, 2, 3, 4};
List<Integer> list = Arrays.asList(tab);  // conversion tableau -> collection

Object[] elems = list.toArray(); // conversion collection ->  []Object
Integer[] typedElems = list.toArray(new Integer[elems.length]);

 // afficher un tableau
System.out.println("elems=" + elems + "," + Arrays.toString(elems));
System.out.println("typedElems=" + typedElems + "," + Arrays.toString(elems));
```