Cours 3 :
Les Listes / Méthode Equals

Le parcours des collections
Interfaces & Classes abstraites pour les listes
Les listes & UML
Itérateur spécial liste
Méthode equals
Les classes concrètes : ArrayList / LinkedList

Auteur : CHAUDET Christelle – Intervenants : BODEVEIX Jean-Paul, MIGEON Frédéric

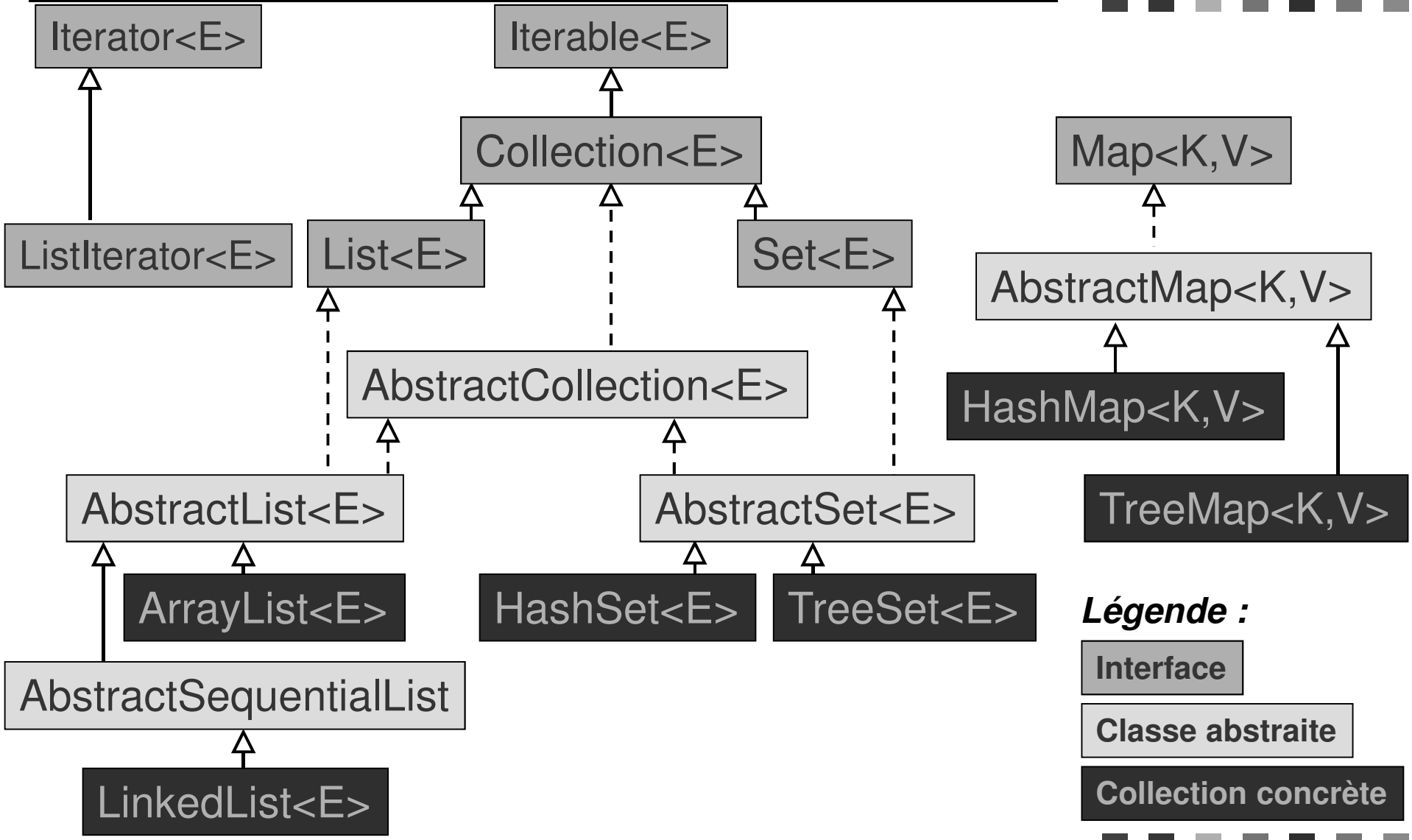
Préambule

- La programmation orientée objet encapsule les données dans des classes.
- Utilisation de structures de données.
- Le choix de la structure dépend du problème à résoudre.
- Exemple :
  - Recherche parmi des milliers de données ?
  - Données triées ?
  - Suppression et insertion dans un ensemble trié ?
  - Tableau donnant un accès à une donnée permettant d'accéder à un ensemble de données encore plus grand ?
- Attention à la structure de données et aux opérations utilisées pour garantir une performance correcte.
- Les collections de Java.

Les différentes versions de JAVA

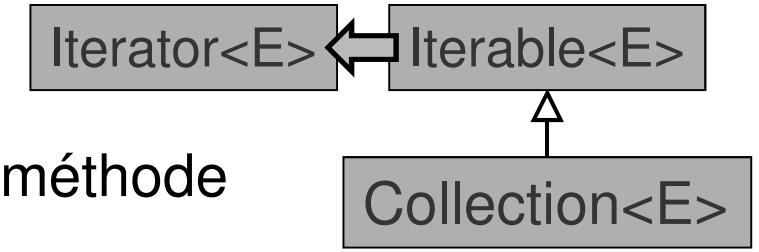
- Même si les collections sont apparues dès la version 1.2, nous utiliserons les interfaces et les classes apparues dans la version 1.4.
  - List liste = **new** ArrayList();
- Pour leur utilisation nous utiliserons leur version générique (à partir de la version 1.5).
  - List<Integer> liste = **new** ArrayList<Integer>();
- Enfin par rapport à la facilité d'écriture nous utiliserons la version 1.7 de Java.
  - List<Integer> liste = **new** ArrayList<>();
- La version 8 de Java n'apporte aucune modification si ce n'est toujours plus de types de collections.

Les collections Java



Le parcours des collections

- Toutes les collections implémentent l'interface **Iterable**, c'est-à-dire que chacune d'entre elles dispose d'une méthode *iterator()*.



java.lang

Interface Iterable<T>

Methods	
Modifier and Type	**Method and Description**
`Iterator<T>`	`iterator()` Returns an iterator over a set of elements of type T.

java.util

Interface Iterator<E>

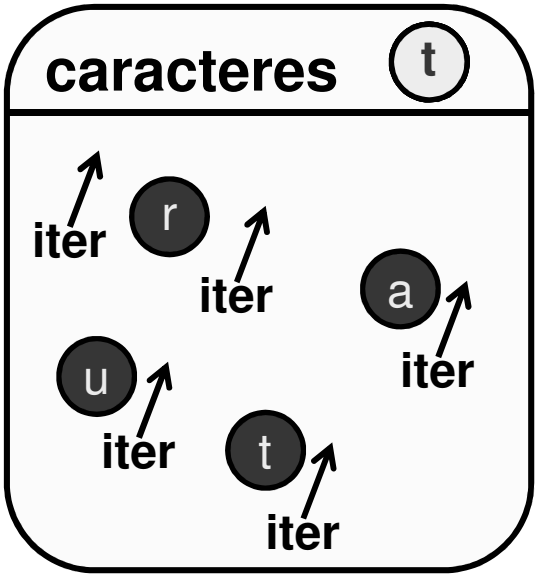
Methods	
Modifier and Type	**Method and Description**
`boolean`	`hasNext()` Returns `true` if the iteration has more elements.
`E`	`next()` Returns the next element in the iteration.
`void`	`remove()` Removes from the underlying collection the last element returned by this iterator (optional operation).

Rappel : Un itérateur permet de parcourir l'ensemble des éléments d'une collection

Le parcours des collections

- Il peut être effectué dans une boucle for en initialisant la variable de boucle avec l'itérateur sur la collection.

```
List<Character> caracteres =  new ArrayList<>();
Collections.addAll(caracteres, 'r', 'a', 'u', 't');
for (Iterator<Character> iterator = caracteres.iterator();
      iterator.hasNext();) {
   Character caractere = iterator.next();
   System.out.println(caractere);
   if (caractere.equals('u')) iterator.remove();
}
```



- Ce même parcours peut être simplifié dans les **cas de lecture** (sans modification de la collection) avec la **boucle foreach**

```
List<Character> caracteres =  new ArrayList<>();
Collections.addAll(caracteres, 'r', 'a', 'u', 't');
for (Character caractere : caracteres) {
   System.out.println(caractere);
}
```

De l'interface à la classe abstraite

- L'interface *Collection* déclare les méthodes les plus courantes que chaque classe implémentée doit fournir.

java.util

Class AbstractCollection<E>

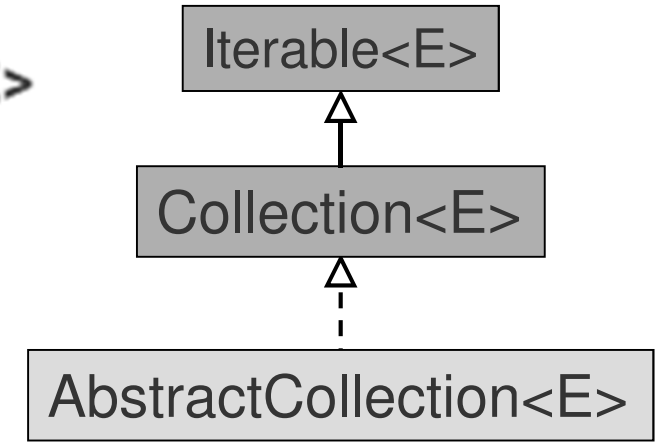
java.lang.Object
    java.util.AbstractCollection<E>

All Implemented Interfaces:

Iterable<E>, Collection<E>

Direct Known Subclasses:

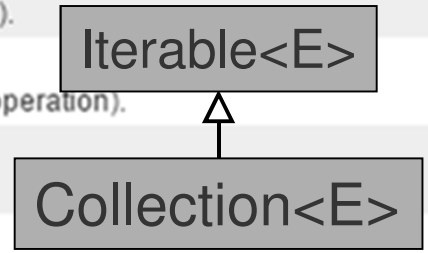
AbstractList, AbstractQueue, AbstractSet, ArrayDeque, ConcurrentLinkedDeque



- Chaque classe concrète de collection étend la classe *AbstractCollection*, et redéfinit éventuellement certaines méthodes.

Interface Collection – Liste des méthodes

boolean	add(E e) Ensures that this collection contains the specified element (optional operation).
boolean	addAll(Collection<? extends E> c) Adds all of the elements in the specified collection to this collection (optional operation).
void	clear() Removes all of the elements from this collection (optional operation).
boolean	contains(Object o) Returns true if this collection contains the specified element.
boolean	containsAll(Collection<?> c) Returns true if this collection contains all of the elements in the specified collection.
boolean	isEmpty() Returns true if this collection contains no elements.
abstract Iterator<E>	iterator() Returns an iterator over the elements contained in this collection.
boolean	remove(Object o) Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	removeAll(Collection<?> c) Removes all of this collection's elements that are also contained in the specified collection (optional operation).
boolean	retainAll(Collection<?> c) Retains only the elements in this collection that are contained in the specified collection (optional operation).
abstract int	size() Returns the number of elements in this collection.
Object[]	toArray() Returns an array containing all of the elements in this collection.
<T> T[]	toArray(T[] a) Returns an array containing all of the elements in this collection; the runtime type of the returned array is that of the specified array.
String	toString() Returns a string representation of this collection.



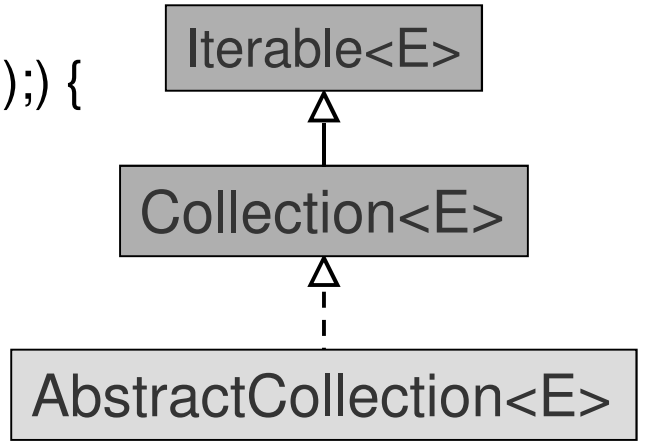
Classe abstraite AbstractCollection

- Exemple :

`void`	`clear()` Removes all of the elements from this collection (optional operation).
`abstract Iterator<E>`	`iterator()` Returns an iterator over the elements contained in this collection.

```
public class AbstractCollection<E> implements Collection<E> {
                    …
  public void clear() {
     for (Iterator<E> e= iterator(); e.hasNext();) {
        e.next ();
        e.remove();
     }
  }

  public abstract Iterator<E> iterator() ;
                    …
}
```



Collection concrètes : ArrayList

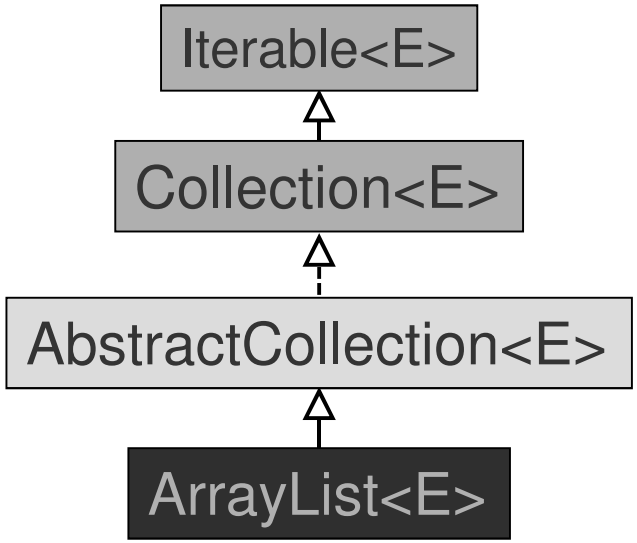
java.util

Class ArrayList<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractList<E>
      java.util.ArrayList<E>

All Implemented Interfaces:

Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess



- Implémente les interfaces *Iterable* et *Collection*, hérite de la classe « AbstractCollection »
- List<String> maListe = **new** ArrayList<>();
  maListe.add("Odralfabétix");
  maListe.add("Astérix");
  maListe.add("Obélix");
  System.*out*.println(maListe);
  // affichage : [Ordralfabétix, Astérix, Obélix]

Utilisation de la méthode toString() héritée de la classe « AbstractCollection »

Ajout dans une ArrayList

- La fonction add() est intelligente : elle assure que l'ensemble des index utilisés est contigu et commence à 0.
- Décalage automatique des éléments déjà présents si nécessaire.
- Exemples d'**utilisation correcte**

```
maListe.clear();                    // vide la liste
maListe.add("Odralfabétix");        // ["Odralfabétix"]
maListe.add("Astérix");             // ["Odralfabétix", "Astérix"]
maListe.add("Obélix");              // ["Odralfabétix", "Astérix", "Obélix"]


maListe.clear();                    // vide la liste
maListe.add(0, "Odralfabétix");     // ["Odralfabétix"]
maListe.add(0, "Astérix");          // ["Astérix", "Odralfabétix"]
maListe.add(0, "Obélix");           // ["Obélix", "Astérix", "Odralfabétix"]
```

Ajout incorrect dans une ArrayList

- Exemples d'**utilisation incorrecte**

  - maListe.clear();
    maListe.add(**1**, "Astérix"); *// Le premier index doit être 0*

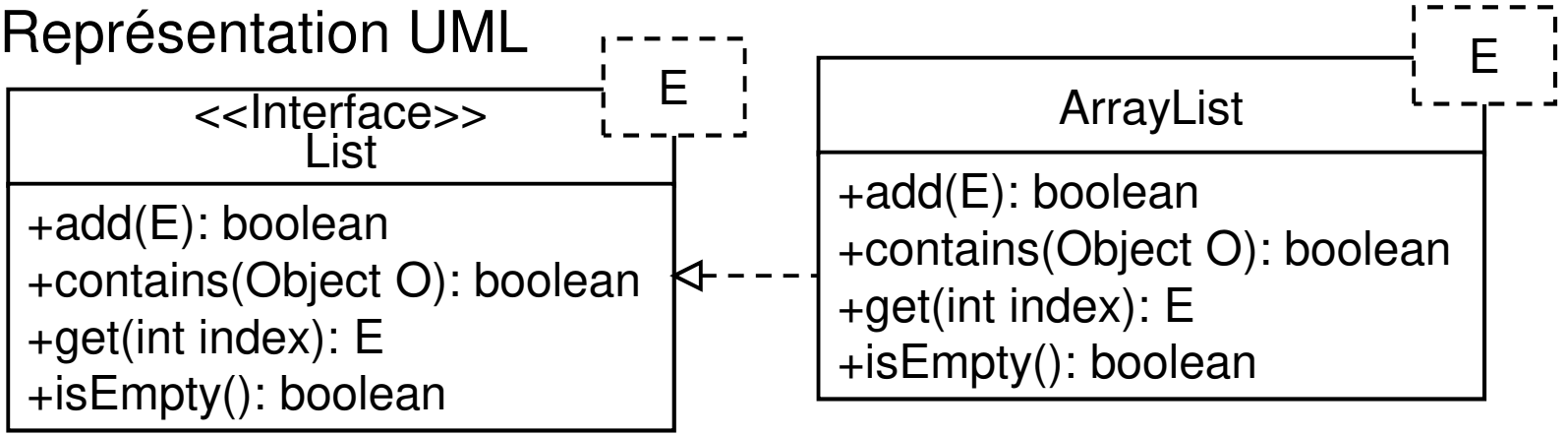
  - maListe.clear();
    maListe.add(0, "Astérix"); *// ["Astérix"]*
    maListe.add(**2**, "Obélix"); *// Il faut 0 ou 1 sinon pas de*
    // contigüité
  - maListe.clear();
    maListe.add(0, "Astérix"); *// ["Astérix"]*
    maListe.add(**-1**, "Obélix"); *// Contigu mais index négatif*
    // interdit

Constructor and Description	Les méthodes utiles pour ce cours. Les autres sont sur votre feuille de TD
`ArrayList()`	Constructs an empty list with an initial capacity of ten.

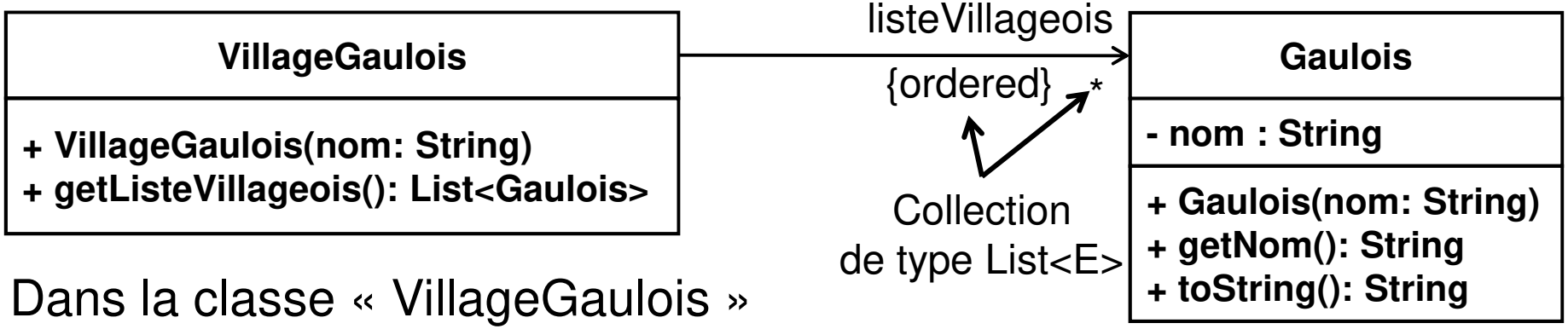
Modifier and Type	Method and Description
`boolean`	`add(E e)` Appends the specified element to the end of this list.
`void`	`add(int index, E element)` Inserts the specified element at the specified position in this
`void`	`clear()` Removes all of the elements from this list.
`boolean`	`contains(Object o)` Returns `true` if this list contains the specified element.
`E`	`get(int index)` **Accès direct en temps constant** Returns the element at the specified position in this list.
`boolean`	`isEmpty()` Returns `true` if this list contains no elements.
`Iterator<E>`	`iterator()` Returns an iterator over the elements in this list in proper sequence.
`int`	`size()` Returns the number of elements in this list.

Représentation UML : ArrayList<E>

- Représentation UML



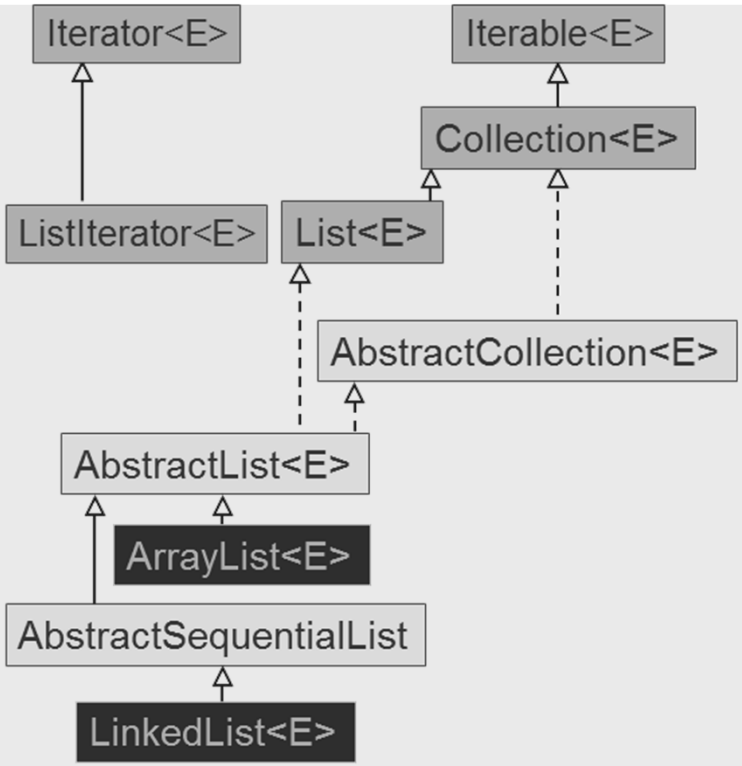
- Exemple d'utilisation



Dans la classe « VillageGaulois »
nous avons donc l'attribut :
List<Gaulois> listeVillageois = **new** ArrayList<>();

Les listes

- L'interface List<E> : apporte des services propres à la liste comme :
  - La création d'un ListIterator depuis un index de la liste
  - Une vue sur un sous-ensemble de la liste
- L'interface ListIterator<E> : apporte un itérateur qui exploite la nature séquentielle de la liste avec :
  - La prise en charge d'un parcours de la liste en sens inverse,
  - L'obtention de la position courante de l'itérateur.
- La classe abstraite AbstractList<E> : optimise certaines méthodes d'AbstractCollection<E> pour une liste et propose une implémentation pour l'ensemble des services de l'interface ListIterator<E>



ListIterator<E>

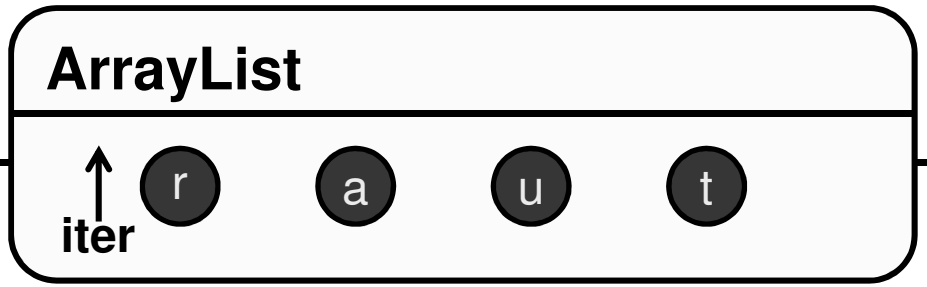
Methods

Modifier and Type	Method and Description
`void`	**`add(E e)`** Inserts the specified element into the list (optional operation).
`boolean`	**`hasNext()`** Returns `true` if this list iterator has more elements when traversing the list in the forward direction.
`boolean`	**`hasPrevious()`** Returns `true` if this list iterator has more elements when traversing the list in the reverse direction.
`E`	**`next()`** Returns the next element in the list and advances the cursor position.
`int`	**`nextIndex()`** Returns the index of the element that would be returned by a subsequent call to **`next()`**.
`E`	**`previous()`** Returns the previous element in the list and moves the cursor position backwards.
`int`	**`previousIndex()`** Returns the index of the element that would be returned by a subsequent call to **`previous()`**.
`void`	**`remove()`** Removes from the list the last element that was returned by **`next()`** or **`previous()`** (optional operation).
`void`	**`set(E e)`** Replaces the last element returned by **`next()`** or **`previous()`** with the specified element (optional operation).

ListIterator<E>

- L'interface ListIterator<E>

```
List<Character> caracteres =  new ArrayList<>();
Collections.addAll(caracteres, 'r', 'a', 'u', 't');
ListIterator<Character> iterateur = caracteres.listIterator(2);
Character caractere = iterateur.next();     caractere : u
iterateur.previous();
caractere = iterateur.previous();           caractere : a
iterateur.next();
iterateur.remove();                         liste : [‘r’, ‘u’, ‘t’]
caractere = iterateur.previous();           caractere : r
iterateur.remove();                         liste : [‘u’, ‘t’]
iterateur.next();
iterateur.add('c');                         liste : [‘u’, ‘c’, ‘t’]
iterateur.previous();
iterateur.set('m');                         liste : [‘u’, ‘m’, ‘t’]
```



Types simples et enveloppeur

- Tout type Java est soit :
  - Un type enveloppeur (Wrapper) :
    - Classe, interface, tableau
    - Sous-type de la classe Object
    - Peut avoir la valeur **null**
    - Exemple : Integer
  - Un type primitif :
    - Au nombre de 8
    - Chacun des types possède son enveloppeur
    - Exemple : int

Type primitif	Enveloppeur
byte	Byte
short	Short
int	Integer
long	Long

Type primitif	Enveloppeur
float	Float
double	Double
bool	Boolean
char	Character

Méthode equals (1/6)

- ==
  - sur deux primitifs permet de comparer 2 valeurs

```
boolean egaux = (3==3);
System.out.println(egaux);
```

```
true
```

  - sur deux objets permet de savoir s'il s'agit de la même référence mémoire

```
Gaulois gaulois1 = new Gaulois("Asterix");
Gaulois gaulois2 = new Gaulois("Asterix");
System.out.println("gaulois1 : " + gaulois1);
System.out.println("gaulois2 : " + gaulois2);
```

```
gaulois1 : villageGaulois.Gaulois@74b65a68
gaulois2 : villageGaulois.Gaulois@6fe99db4
```

```
boolean egaux = (gaulois1 == gaulois2);
System.out.println(egaux);
```

```
false
```

Méthode equals (2/6)

- La méthode equals :
  - est une méthode de la classe Object donc héritée par l'ensemble des classes,
  - Retourne vrai si deux objets possèdent la même référence mémoire et sont donc en fait le même objet.

```
Gaulois gaulois1 = new Gaulois("Asterix");
Gaulois gaulois2 = gaulois1;
System.out.println("gaulois1 : " + gaulois1);
System.out.println("gaulois2 : " + gaulois2);
```

```
gaulois1 : villageGaulois.Gaulois@6fe99db4
gaulois2 : villageGaulois.Gaulois@6fe99db4
```

```
boolean egaux = (gaulois1 == gaulois2);
System.out.println(egaux);
```

```
true
```

Méthode equals (3/6)

- La méthode equals telle qu'elle est implémentée dans la classe **Object** est donc équivalente à l'opérateur == (égalité physique).

```
Gaulois gaulois1 = new Gaulois("Asterix");
Gaulois gaulois2 = new Gaulois("Asterix");
System.out.println("gaulois1 : " + gaulois1);
System.out.println("gaulois2 : " + gaulois2);
```

```
gaulois1 : villageGaulois.Gaulois@74b65a68
gaulois2 : villageGaulois.Gaulois@6fe99db4
```

```
boolean egaux = gaulois1.equals(gaulois2);
System.out.println(egaux);
```

```
false
```

- Nous devons donc redéfinir la méthode equals afin de comparer deux objets sur le nom du Gaulois (égalité logique) et non sur leur référence.

Méthode equals (4/6)

- Dans la classe Gaulois nous redéfinissons la méthode equals :

```
public boolean equals(Object obj) {
    if(obj instanceof Gaulois){
        Gaulois gaulois = (Gaulois) obj;
        return nom.equals(gaulois.getNom());
    }
    return false;
}
```

- Refaisons la comparaison :

```
Gaulois gaulois1 = new Gaulois("Asterix");
Gaulois gaulois2 = new Gaulois("Asterix");
System.out.println("gaulois1 : " + gaulois1);
System.out.println("gaulois2 : " + gaulois2);
```

```
gaulois1 : villageGaulois.Gaulois@74b65a68
gaulois2 : villageGaulois.Gaulois@6fe99db4
```

```
boolean egaux = gaulois1.equals(gaulois2);
System.out.println(egaux);
```

```
true
```

Méthode equals (5/6)

- Conclusion : la méthode equals est destinée à être redéfinie afin de comparer deux objets sur leur état et non sur leur référence.

- Tous les enveloppeurs ont leur méthode equals qui a été redéfinie :

```
Integer x = new Integer(3);
Integer y = new Integer(3);
System.out.println(x==y);          ⇨ false
System.out.println(x.equals(y));   ⇨ true
```

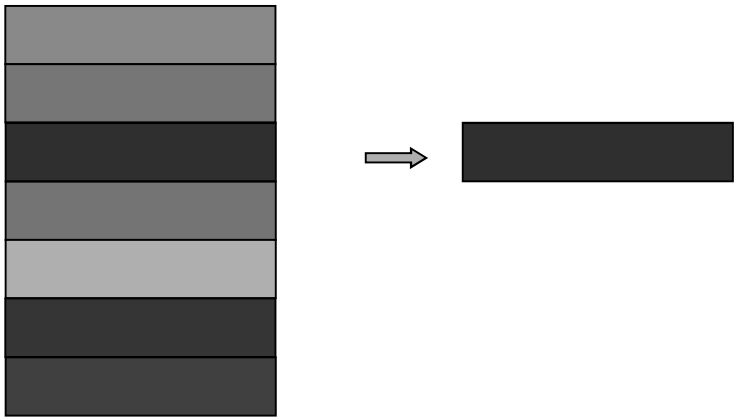
- Il faut donc :
  - utiliser == sur un type primitif (qui ne possède pas de méthode equals !)
  - equals sur un enveloppeur ou tout autre objet (sauf si on veut réellement savoir s'il s'agit de la même référence mémoire !)

Méthode equals (6/6)

- Plusieurs méthodes des collections utilisent la méthode equals comme :
  - contains(Object o)
  - containsAll(Collection<?> c)
  - remove(Object o)
  - removeAll(Collection<?> c)
  - retainAll(Collection<?> c)
- Il ne faut donc pas oublier de redéfinir la méthode equals dans les objets stocker dans les collections.
- **Notez bien** : equals n'implante pas nécessairement l'égalité structurelle entre les objets mais une relation d'équivalence (« avoir le même nom »)

Les limites de ArrayList

- Inconvénient majeur : la suppression et l'insertion d'un élément au milieu d'un tableau nécessite beaucoup de temps machine.
- Exemple : la suppression d'un élément



Les objets d'un tableau occupent des emplacements mémoires successifs : tous les éléments doivent être décalés d'une case.

LinkedList<E>

- Une liste chaînée est une collection **classée**.
- En tant qu'implémentation de List :
  - à éviter si votre application a de nombreux accès aléatoires,
  - à utiliser pour améliorer les performances d'une ArrayList en ajout et en suppression d'éléments.

java.util

Class LinkedList<E>

java.lang.Object
  └ java.util.AbstractCollection<E>
    └ java.util.AbstractList<E>
      └ java.util.AbstractSequentialList<E>
        └ **java.util.LinkedList<E>**

Type Parameters:

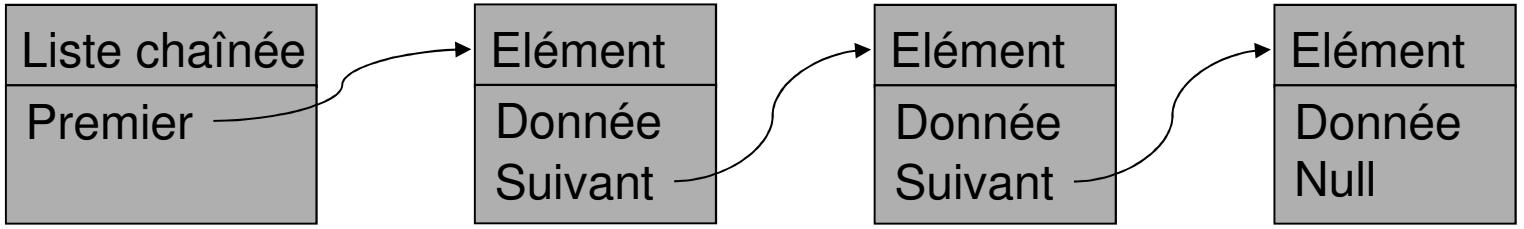
E - the type of elements held in this collection

All Implemented Interfaces:

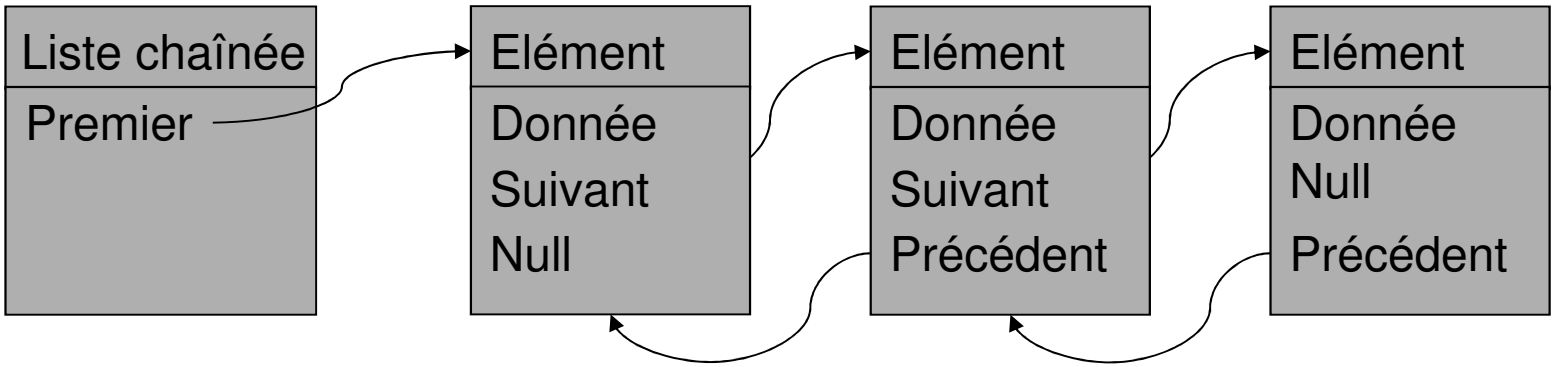
Serializable, Cloneable, Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>

Collections concrètes : LinkedList

- Une liste chaînée stocke chaque objet avec un lien qui y fait référence et une référence au lien suivant.



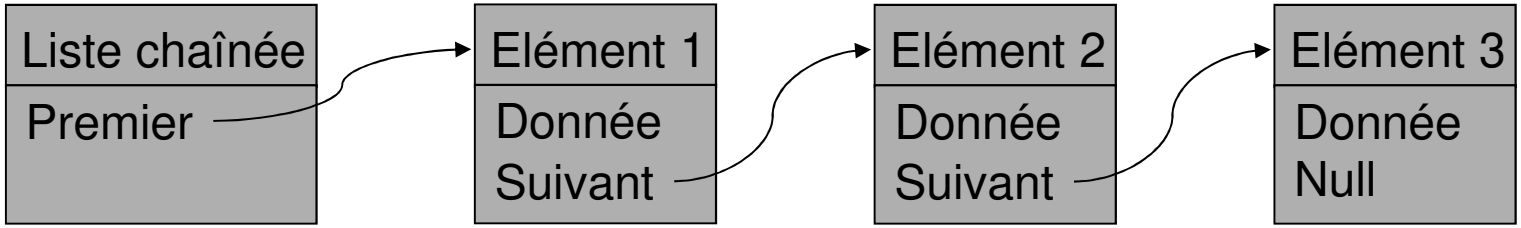
- En Java les listes sont doublement chaînées



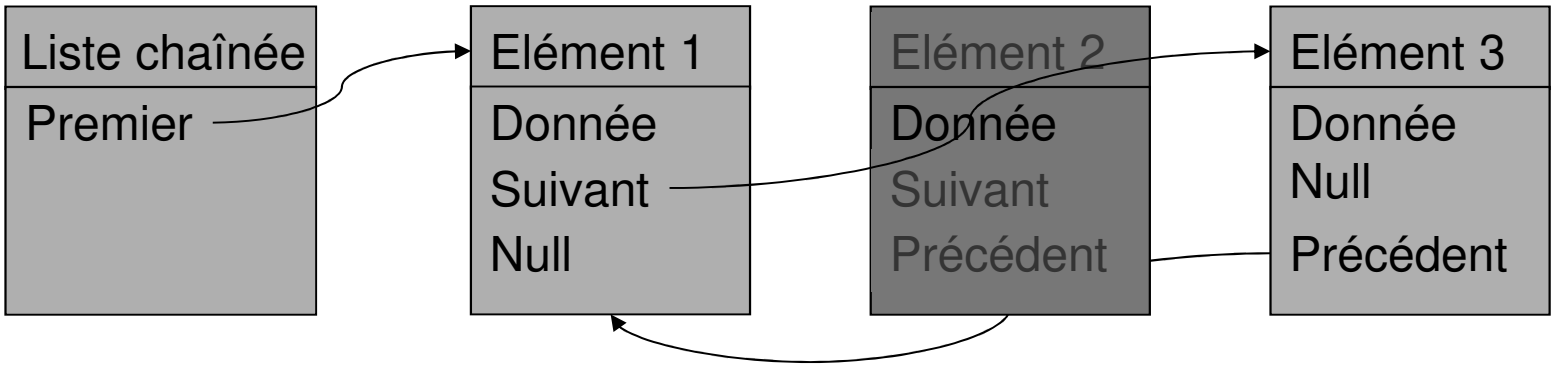
- La suppression ou l'insertion d'un élément ne modifie que les références au lien précédent et au lien suivant

Collections concrètes : LinkedList

- Une liste chaînée stocke chaque objet dans une cellule contenant un lien sur la cellule suivante



- En Java les listes sont doublement chaînées



- La suppression ou l'insertion d'un élément ne modifie que les références au lien précédent et au lien suivant

Les limites de LinkedList

- Inconvénient majeur : les listes chaînées ne permettent pas d'accéder rapidement à un élément : pour connaitre le Xème élément il faut parcourir les X-1premiers (accès séquentiel).

- La méthode get permet d'accéder à un élément particulier :
  Object obj = liste.get(n);
  Technique non efficace ! Si vous l'utilisez c'est que la structure n'est **pas adaptée** à votre problème.

- Exemple inefficace
  for (**int** i = 0 ; i < liste.size() ; i++)
  utilisation de liste.get(i);
  pour chaque nouvel élément la liste est parcourue depuis le début !