Licence MIASHS

INF F5 — Collections et Maps

UGA UFR SHS Université Grenoble Alpes

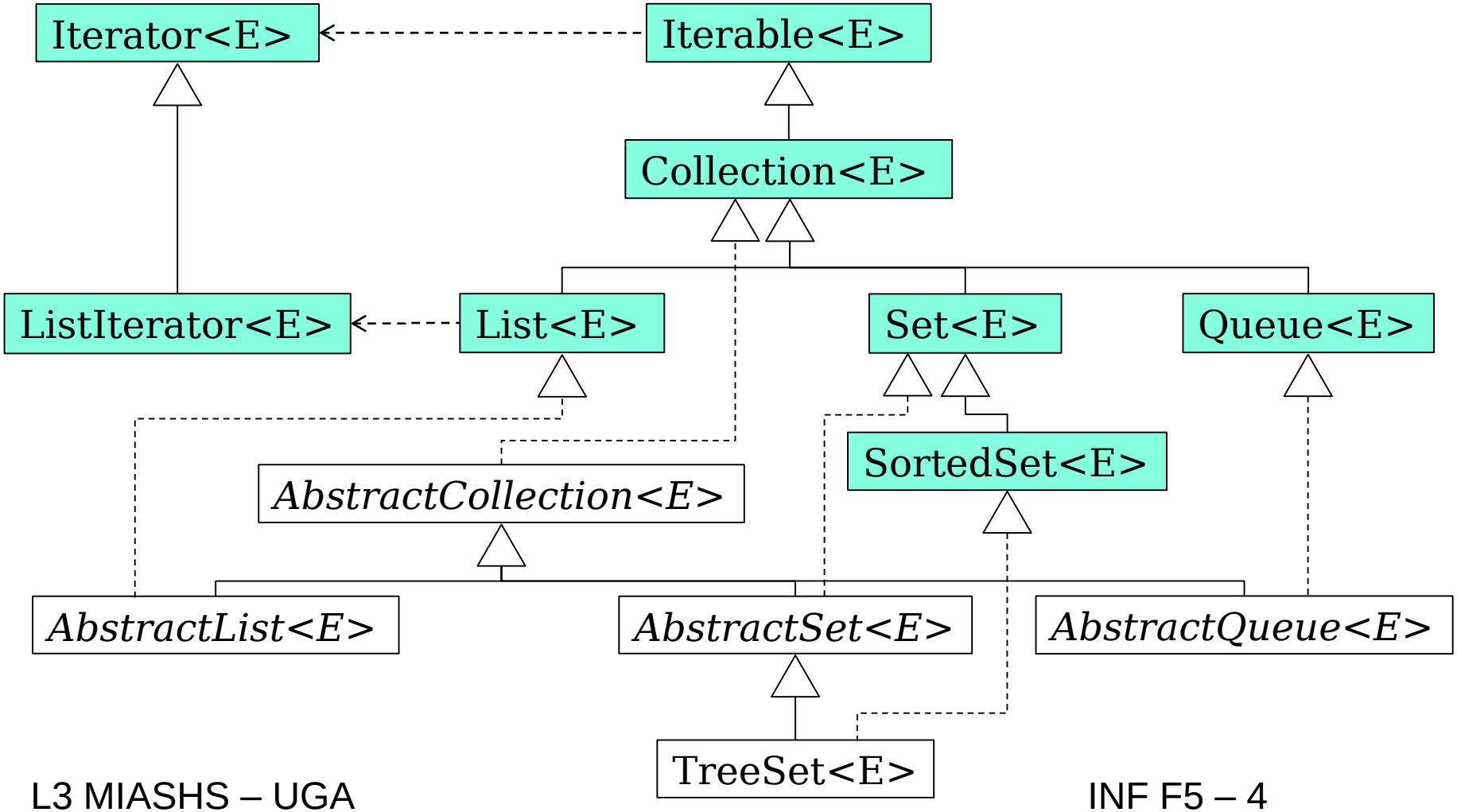
Daniel.Bardou@univ-grenoble-alpes.fr

Sommaire

- Collections
  - interfaces
  - classes prédéfinies
  - ArrayList
  - LinkedList
- Maps
  - interfaces et clases prédéfinies
  - HashMap
  - TreeMap

Collections

Interfaces


Iterator<E>
Iterable<E>
Collection<E>
ListIterator<E>
List<E>
Set<E>
Queue<E>
SortedSet<E>
AbstractCollection<E>
AbstractList<E>
AbstractSet<E>
AbstractQueue<E>
TreeSet<E>

Iterator<E>

- Un Iterator est un objet permettant d'effectuer un parcours des éléments d'une Collection.
- Méthodes
  - **boolean** hasNext()
    retourne **true** si le parcours n'est pas terminé.
  - E next()
    retourne le prochain élément dans le parcours. Lève une NoSuchElementException si le parcours est terminé.
  - **void** remove()
    supprime le dernier élément retourné par next(). Lève une IllegalStateException si la méthode next() n'a jamais été appelée ou s'il y a déjà eu un remove() depuis le dernier next(). **Méthode « optionnelle »** : levée possible d'une UnsupportedOperationException.

ListIterator<E>

- Un ListIterator est un Iterator permettant de parcourir une List. Le parcours avant-arrière est possible, de même que la possibilité d'obtenir des indices.
- Méthodes de parcours additionnelles (hasNext() et next() sont héritées pour le parcours avant)
  - **int** nextIndex()
    retourne l'indice de l'élément qui serait retourné par un appel à next() ou -1 si fin du parcours avant.
  - **boolean** hasPrevious()
    retourne **true** si le parcours arrière n'est pas terminé.
  - E previous()
    retourne le prochain élément dans un parcours arrière. Lève une NoSuchElementException si le parcours arrière est terminé.
  - **int** previousIndex()
    retourne l'indice de l'élément qui serait retourné par un appel à previous() ou -1 si fin du parcours arrière.

ListIterator<E>

- Méthodes de modification (nouvelle spécification pour `remove()` qui est héritée)
  - **void** `remove()`
    supprime dans la liste parcourue le dernier élément retourné par `next()` ou `previous()`. Lève une `IllegalStateException` s'il n'y a pas encore eu d'appel ni de `next()`, ni de `previous()`, ou s'il y a déjà eu un `remove()` ou un `add()` depuis le dernier `next()` ou `previous()`. **Méthode « optionnelle »**.
  - **void** `set(E e)`
    remplace par `e` dans la liste parcourue le dernier élément retourné par `next()` ou `previous()`. Lève une `IllegalStateException` s'il n'y a pas encore eu d'appel ni de `next()`, ni de `previous()`, ou s'il y a déjà eu un `remove()` ou un `add()` depuis le dernier `next()` ou `previous()`. **Méthode « optionnelle »**.
  - **void** `add(E e)`
    insère `e` dans la liste parcourue juste l'avant l'élément d'indice `nextIndex()` s'il existe, et juste après l'élément d'indice `previousIndex()` s'il existe. Si la liste est vide `e` devient son unique élément. **Méthode « optionnelle »**.

Iterable<E>

- Un Iterable est un objet que l'on peut parcourir à l'aide d'un Iterator.
- Méthode
  - Iterator<E> iterator()
    retourne un Iterator permettant de parcourir **this**.
- Un parcours total sur un Iterable (et aussi sur les tableaux) peut être réalisé grâce à un « foreach ».
  - Exemple avec une List :

```
List<Object> l;
// ...
for(Object obj : l)
  System.out.println(obj);
```

  - Exemple avec un tableau :

```
int[] tab;
int somme = 0;
// ...
for(int v : tab)
  somme += v;
```

Collection<E>

- Une Collection est un objet pouvant contenir plusieurs éléments.
- Méthodes générales (iterator() est hérité d'Iterable)
  - **int** size()
    retourne le nombre d'éléments de **this**.
  - **boolean** isEmpty()
    retourne **true** si **this** n'a aucun élément.
  - **boolean** contains(Object obj)
    retourne **true** si **this** contient au moins un élément équivalent à obj.
    Lève une ClassCastException si le type d'obj est incompatible avec E.
    Lève une NullPointerException si obj est **null** et que **this** n'autorise pas d'élément **null**.
  - **void** clear()
    supprime tous les éléments de **this**.

Collection<E>

- Méthodes de modification
  - **boolean** add(E e)
    ajoute e à **this** et retourne **true** si **this** a effectivement été modifié. **Méthode « optionnelle »**.
  - **boolean** remove(Object obj)
    supprime dans **this** un élément équivalent à obj et retourne **true** si **this** a effectivement été modifié. **Méthode « optionnelle »**.

Collection<E>

- Méthodes « ensemblistes »
  - **boolean** containsAll(Collection<?> c)
    retourne **true** si **this** contient tous les éléments de c.
  - **boolean** addAll(Collection<? **extends** E> c)
    ajoute tous les éléments de c à **this** et retourne **true** si **this** a effectivement été modifié. **Méthode « optionnelle »**.
  - **boolean** removeAll(Collection<?> c)
    supprime dans **this** tous les éléments équivalents à un élément de c et retourne **true** si **this** a effectivement été modifié. **Méthode « optionnelle »**.
  - **boolean** retainAll(Collection<?> c)
    supprime dans **this** tout élément qui n'est pas équivalent à au moins un élément de c et retourne **true** si **this** a effectivement été modifié. **Méthode « optionnelle »**.

Collection<E>

- Méthodes d'export vers un tableau
  - `Object[] toArray()`
    retourne un tableau contenant tous les éléments de **this**. Le tableau retourné est indépendant de **this**.
  - `<T> T[] toArray(T[] tab)`
    si `tab` a une taille suffisante, retourne `tab` après y avoir rangé (en début de tableau) tous les éléments de **this**.
    Si `tab` n'a pas une taille suffisante, retourne un tableau du même type que `tab` contenant exactement tous les éléments de **this**.
    Lève une `ArrayStoreException` si `E` est incompatible avec `T`.
    Lève une `NullPointerException` si `tab` vaut **null**.

List<E>

- Une List est une Collection permettant l'accès indicé à ses éléments.
- Méthodes générales additionnelles (List hérite de Collection)
  - ListIterator<E> listIterator()
    retourne un ListIterator pour parcourir **this**.
  - ListIterator<E> listIterator(**int** i)
    retourne un ListIterator pour parcourir **this**, avec un curseur de lecture positionné juste avant l'élément d'indice i.
  - List<E> subList(**int** from, **int** to)
    retourne une « vue » de **this** limitée à ses éléments d'indice from inclus à to exclus. Toute modification de **this** indépendamment de la vue retournée peut rendre cette vue incohérente.
    Lève une IndexOutOfBoundsException si les valeurs de from et/ou to sont incorrectes.

List<E>

- Méthodes pour l'accès indicé. Toutes ces méthodes lèvent une IndexOutOfBoundsException si la valeur de i est incorrecte.
  - E get(**int** i)
    retourne l'élément d'indice i.
  - E set(**int** i, E e)
    retourne l'élément d'indice i après l'avoir remplacé par e dans **this**. **Méthode « optionnelle »**.
  - **void** add(**int** i, E e)
    insère e dans **this** comme nouvel élément à l'indice i. **Méthode « optionnelle »**.
  - E remove(**int** i)
    retourne l'élément d'indice i après l'avoir supprimé de **this**. **Méthode « optionnelle »**.

List<E>

- Méthodes de recherche.
  - **int** indexOf(Object obj)
    retourne l'indice du premier élément de **this** équivalent à obj, ou -1 si aucun élément de **this** est équivalent à obj.
  - **int** lastIndexOf(Object obj)
    retourne l'indice du dernier élément de **this** équivalent à obj, ou -1 si aucun élément de **this** est équivalent à obj.

Set<E>

- Un Set est une Collection dans laquelle l'unicité des éléments est garantie.
  - Il ne peut pas y avoir dans un Set deux éléments e1 et e2 tels que e1.equals(e2).
  - Si le Set accepte **null** comme élément, un seul de ses éléments peut être **null**.
  - Si le Set accepte l'ajout de nouveaux éléments, la méthode add(E e) retourne **false** si le Set contient déjà un élément équivalent à e.
  - Attention aux objets mutables utilisés comme éléments.
  - Pas d'autres méthodes que celles héritées de Collection.

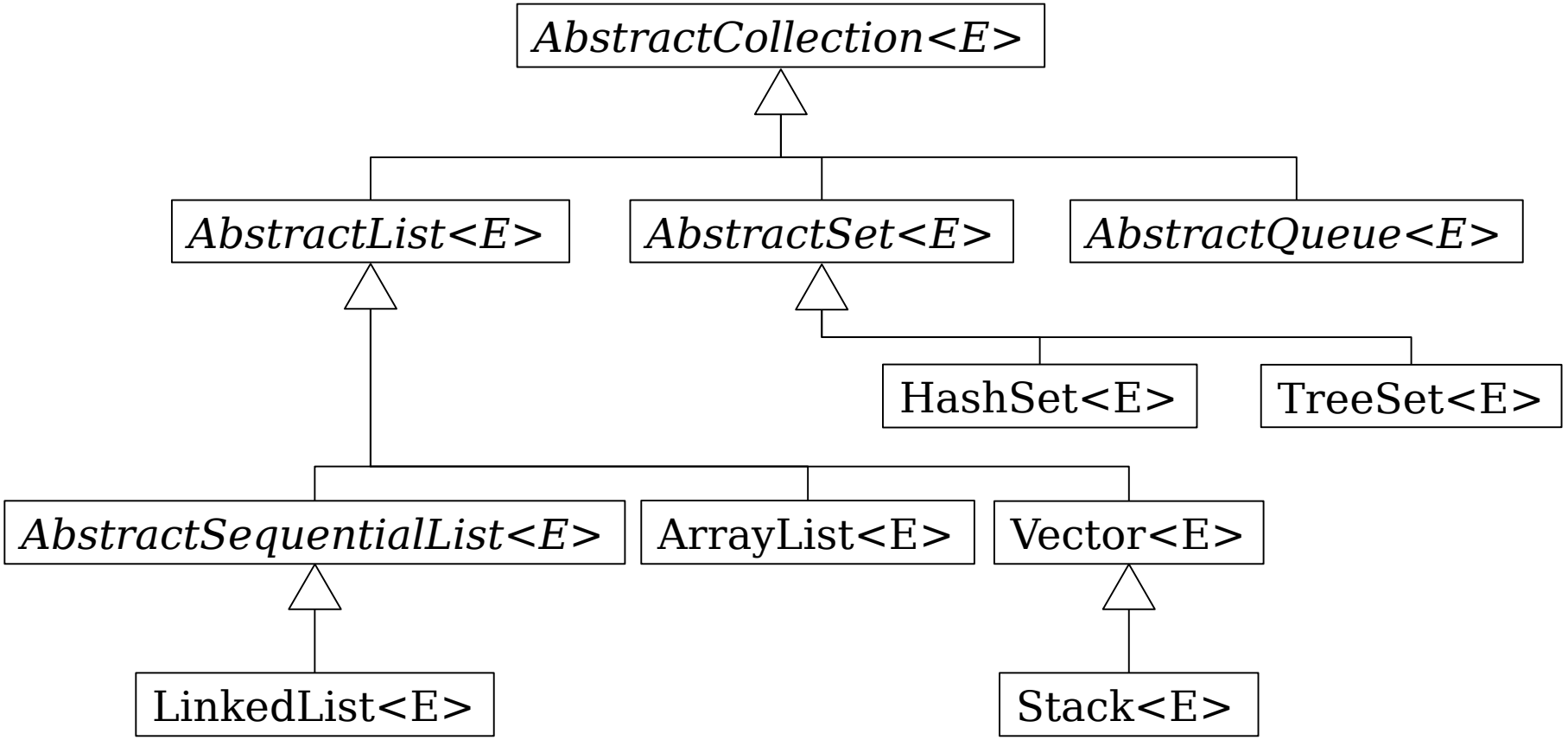
SortedSet<E>

- Un SortedSet est un Set qui garantit que le parcours de ses éléments se fait dans l'ordre de ceux-ci.
- L'ordre des éléments est déterminé à la création du SortedSet :
  - ordre « naturel » si les éléments sont des Comparable.
  - ordre spécifique si un Comparator est fourni.

SortedSet<E>

- Méthodes additionnelles (SortedSet hérite de Set)
  - Comparator<? **super** E> comparator()
    retourne le Comparator associé à **this** ou **null** si **this** n'est pas muni d'un Comparator.
  - SortedSet<E> subSet(E from, E to)
    retourne une vue de **this** limitée aux éléments compris entre from inclus et to exclus. Levée d'une IllegalArgumentException si les valeurs de from et/ou to sont incorrectes.
  - SortedSet<E> headSet(E to)
    retourne une vue de **this** limitée aux éléments strictement inférieurs à to.
  - SortedSet<E> tailSet(E from)
    retourne une vue de **this** limitée aux éléments supérieurs ou équivalents à from.
  - E first()
    retourne le premier (le plus petit) élément de **this**. Levée d'une NoSuchElementException si **this** est vide.
  - E last()
    retourne le dernier (le plus grand) élément de **this**. Levée d'une NoSuchElementException si **this** est vide.

Classes


AbstractCollection<E>
AbstractList<E>
AbstractSet<E>
AbstractQueue<E>
HashSet<E>
TreeSet<E>
AbstractSequentialList<E>
ArrayList<E>
Vector<E>
LinkedList<E>
Stack<E>

AbstractCollection<E>

- Classe abstraite racine de la hiérarchie d'héritage des collections, implémente Collection<E>.
- Aucun attribut localement déclaré
- Constructeur
  - **protected** AbstractCollection()
- 2 méthodes laissées abstraites
  - **public abstract** Iterator<E> iterator()
  - **public abstract int** size()

AbstractCollection<E>

- Implémentation par défaut de toutes les autres méthodes imposées par Collection<E> et redéfinition de toString()
  - **public boolean** contains(Object obj)
  - **public boolean** isEmpty()
  - **public void** clear()
  - **public boolean** add(E e)
  - **public boolean** remove(Object obj)
  - **public boolean** containsAll(Collection<?> c)
  - **public boolean** addAll(Collection<? **extends** E> c)
  - **public boolean** removeAll(Collection<?> c)
  - **public boolean** retainAll(Collection<?> c)
  - **public** Object[] toArray()
  - **public** <T> T[] toArray(T[] t)

AbstractCollection<E>

- La méthode add(E e) est laissée optionnelle …

```
public boolean add(E e) {
    throw new UnsupportedOperationException();
}
```

AbstractCollection<E>

- Mais pas la méthode remove(Object obj) :

```
public boolean remove(Object obj) {
    Iterator<E> it = iterator();
    if (obj == null) {
        while (it.hasNext())
            if (it.next() == null) {
                it.remove();
                return true;
            }
    } else {
        while (it.hasNext())
            if (obj.equals(it.next())) {
                it.remove();
                return true;
            }
    }
    return false;
}
```

AbstractCollection<E>

- Définition de addAll(Collection<? extends E> c) :

```
public boolean addAll(Collection<? extends E> c) {
    boolean modified = false;
    for (E e : c)
        if (add(e))
            modified = true;
    return modified;
}
```

AbstractList<E>

- Hérite d'`AbstractCollection<E>`, propose une implémentation de `List<E>` basique pouvant convenir à une structure sous-jacente à accès indicée (*e.g.* un tableau).
- Un attribut localement déclaré pour la coordination des `Iterator`, mais aucun attribut dirigeant le stockage des éléments
  - `protected transient int modCount = 0;`

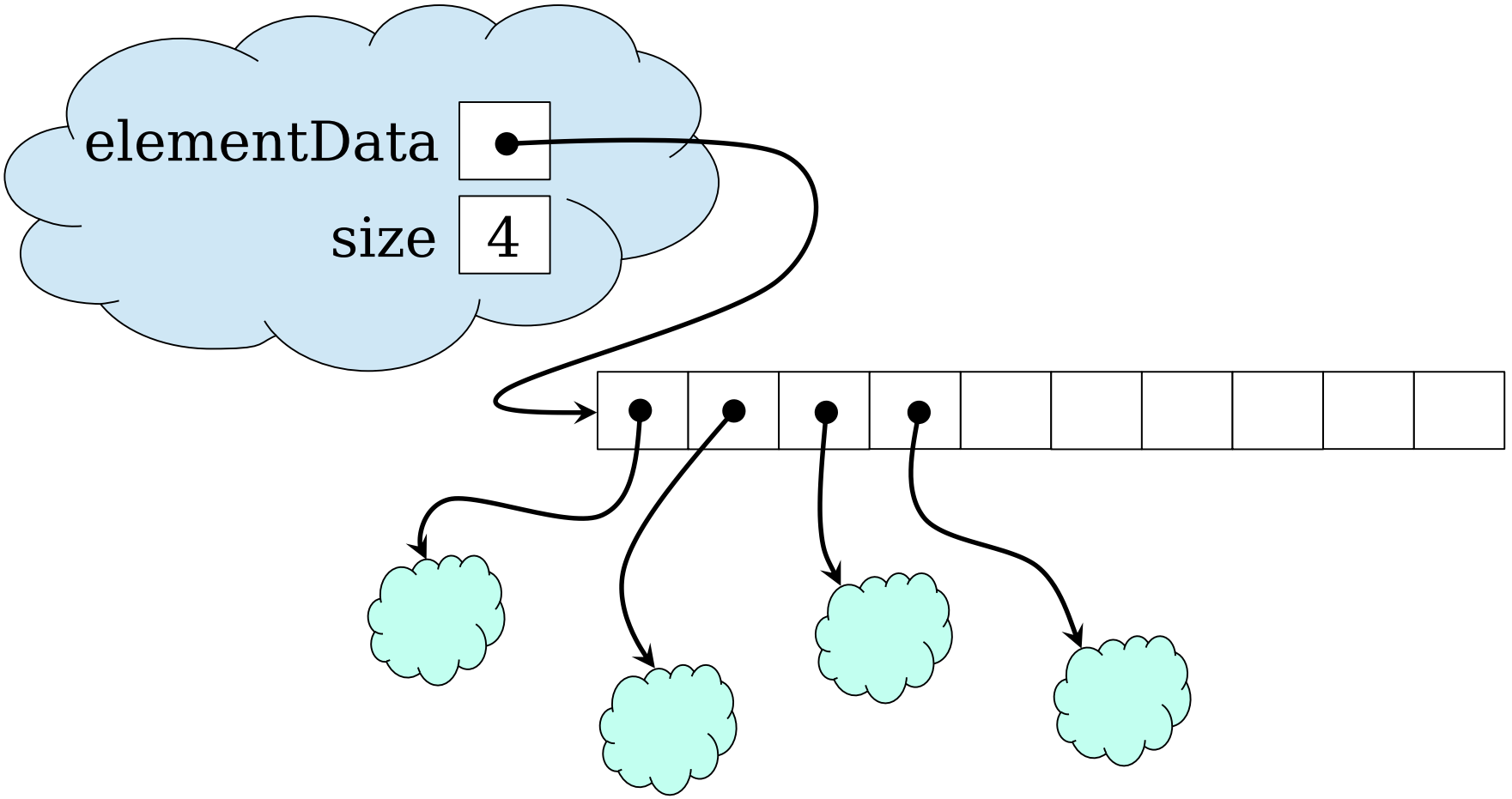
AbstractList<E>

- Méthodes laissées abstraites :
  - **public abstract** E get(**int** i);
  - **public abstract int** size();
- Méthodes optionnelles :
  - **public void** add(**int** i, E e)
  - **public** E set(**int** i, E e)
  - **public** E remove(**int** i)
- Deux classes imbriquées Itr et ListItr proposent des implémentations d'Iterator et ListIterator.

ArrayList<E>

- Hérite d'AbstractList<E>, propose une implémentation de List<E> basée sur un tableau et réalise aussi les interfaces RandomAccess, Cloneable et java.io.Serializable.
- Le tableau est agrandi automatiquement si besoin (technique du buffer).
- Deux attributs localement déclarés :
  - **transient** Object[] elementData;
  - **private int** size;

ArrayList<E>



ArrayList<E>

- Constructeurs
  - **public** ArrayList(**int** capInitiale)
    permet d'obtenir une ArrayList vide de capacité initiale capInitiale.
  - **public** ArrayList()
    équivalent à ArrayList(10).
  - **public** ArrayList(Collection<? **extends** E> c)
    permet d'obtenir une ArrayList contenant initialement tous les éléments de c.

ArrayList<E>

- Trois classes imbriquées Itr, ListItr et SubList. Les deux premières proposent des implémentations d'Iterator et ListIterator.
- La plupart des méthodes héritées sont redéfinies pour assurer de meilleures performances.
- Définition de size() et get(**int** i)
  - ```
    public int size() {
        return size;
    }
    ```
  - ```
    public E get(int i) { // simplifiée
        if (i < 0 || i >= size)
            throw new IndexOutOfBoundsException();
        return (E) elementData[i];
    }
    ```

ArrayList<E>

- Définition de add(E e) et add(**int** i, E e)
  - ```
    private void ensureCapacity(int minCap) { // simplifiée
        modCount ++;
        if (minCap – elementData.length > 0)
            grow(minCap);
    }
    ```
  - ```
    public boolean add(E e) {
        ensureCapacity(size + 1);
        elementData[size ++] = e;
        return true;
    }
    ```
  - ```
    public void add(int i, E e) { // simplifiée
        if (i < 0 || i > size)
            throw new IndexOutOfBoundsException();
        ensureCapacity(size + 1);
        System.arraycopy(elementData, i, elementData, i + 1, size – i);
        elementData[i] = e;
        size ++;
    }
    ```
    ```

ArrayList<E>

- Une méthode spécifique
  - ```
    private void trimToSize() { // simplifiée
        modCount ++;
        if (size < elementData.length)
            elementData = (size == 0)
            ? new Object[0]
            : Arrays.copyOf(elementData, size);
    }
    ```
    ```

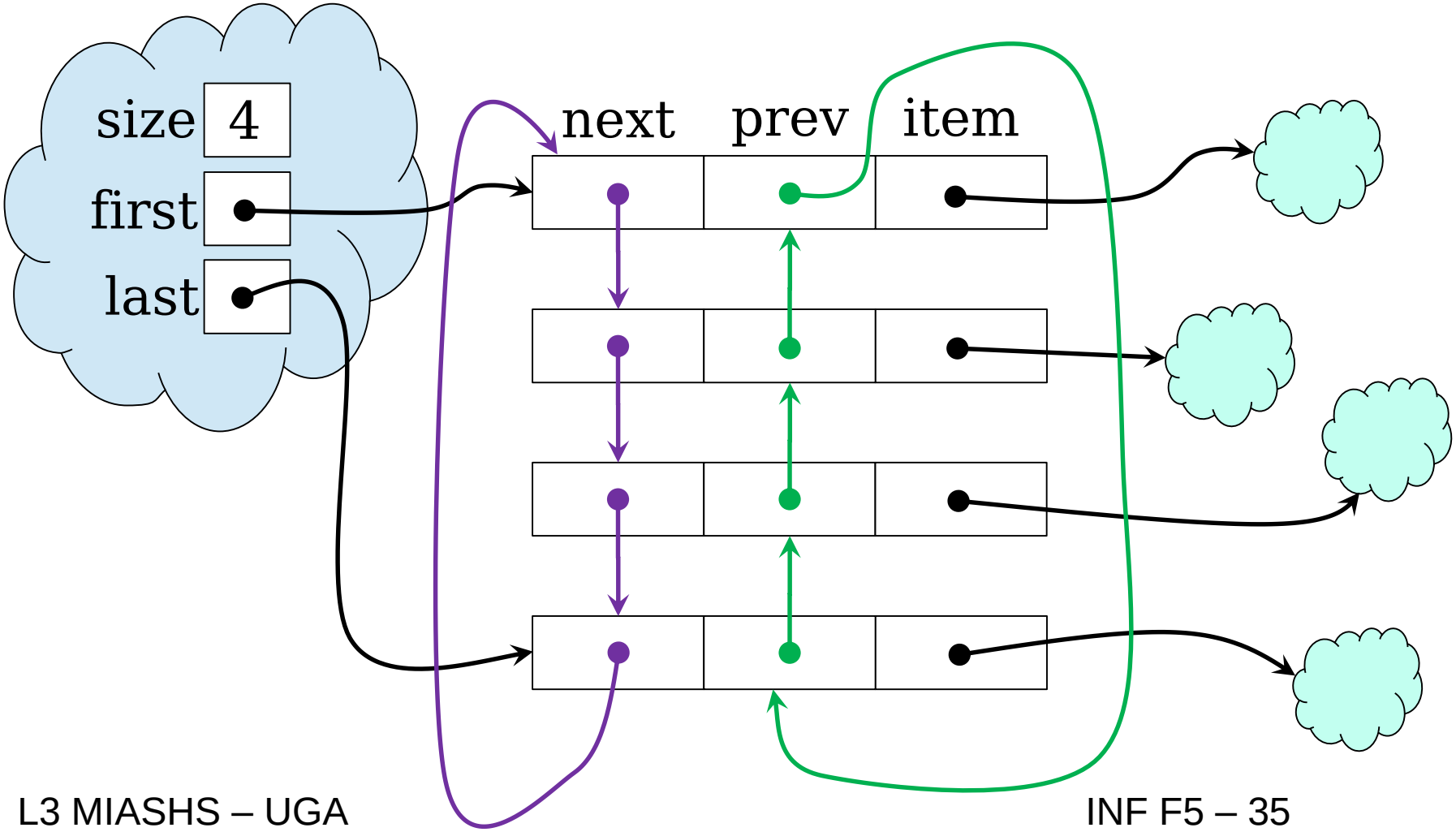
AbstractSequentialList<E>

- Hérite d'`AbstractList<E>`, propose une implémentation de `List<E>` basique pouvant convenir à une structure sous-jacente à accès séquentiel.
- Pas d'attribut localement déclaré.
- Quelques méthodes redéfinies pour fonctionner grâce à un `ListIterator` plutôt qu'un accès indicé.
- La méthode `size()` est héritée abstraite, la méthode `listIterator(int i)` est redéfinie abstraite, la méthode `get(int i)` est définie.

LinkedList<E>

- Hérite d'AbstractSequentialList<E>, propose une implémentation de List<E> basée sur un double chaînage circulaire, et réalise aussi les interfaces RandomAccess, Cloneable et java.io.Serializable.
- Une classe imbriquée ListItr pour fournir les ListIterator.
- Une classe imbriquée privée Node<E> pour définir les chaînons.
- Trois attributs localement déclarés :
  - **transient int** size;
  - **transient** Node<E> first; // premier chaînon
  - **transient** Node<E> last; // dernier chaînon

LinkedList<E>



LinkedList<E>

- Constructeurs
  - **public** LinkedList()
    permet d'obtenir une LinkedList vide.
  - **public** LinkedList(Collection<? **extends** E> c)
    permet d'obtenir une LinkedList contenant initialement tous les éléments de c.

LinkedList<E>

- Définition de size() et listIterator(**int** i)
  - ```
    public int size() {
        return size;
    }
    ```
  - ```
    public ListIterator<E> listIterator(int i) {
        // simplifiée
        if (i < 0 || i > size)
            throw new IndexOutOfBoundsException();
        return new ListItr(i);
    }
    ```

LinkedList<E>

- Définition de get(**int** i)
  - ```
    private E get(int i) { // adaptée
        if (i < 0 || i > size)
            throw new IndexOutOfBoundsException();
        if (i < (size >> 1)) {
            Node<E> x = first;
            for (int idx = 0; idx < i; idx ++)
                x = x.next;
            return x.item;
        } else {
            Node<E> x = last;
            for (int idx = size – 1; idx > i; i --)
                x = x.prev;
            return x.item;
        }
    }
    ```
    ```

LinkedList<E>

- Des méthodes spécifiques
  - **public** E getFirst()
    retourne le premier élément de **this**. Lève une NoSuchElementException si **this** est vide.
  - **public** E getLast()
    retourne le dernier élément de **this**. Lève une NoSuchElementException si **this** est vide.
  - **private** E removeFirst()
    retourne le premier élément après l'avoir supprimé de **this**. Lève une NoSuchElementException si **this** est vide.
  - **private** E removeLast()
    retourne le dernier élément après l'avoir supprimé de **this**. Lève une NoSuchElementException si **this** est vide.
  - **private void** addFirst(E e)
    ajoute e au début de **this**.
  - **private void** addLast(E e)
    ajoute e à la fin de **this**.

AbstractSet<E>

- Hérite d'AbstractCollection<E> et propose une implémentation minimale de Set<E>.
- Pas d'attribut localement déclaré.
- Seule les méthodes suivantes sont redéfinies
  - **public boolean** equals(Object obj)
  - **public int** hashCode()
  - **public boolean** removeAll(Collection<?> c)

HashSet<E>

- Hérite d'AbstractSet<E> et propose une implémentation de Set<E> basée sur une HashMap<E, Object>, et réalise les interfaces Cloneable et java.io.Serializable.
- Un attribut localement déclaré
  - **private transient** HashMap<E, Object> map;
- La plupart des méthodes héritées sont redéfinies par délégation explicite vers map.
- Définition de size()
  - ```
    public int size() {
        return map.size();
    }
    ```
    ```

HashSet<E>

- Constructeurs
  - **public** HashSet(**int** initCap, **float** loadFactor)
    permet d'obtenir un HashSet vide de capacité initiale initCap et de facteur de charge loadFactor.
  - **public** HashSet(**int** initCap)
    équivalent à HashSet(initCap, 0.75f).
  - **public** HashSet()
    équivalent à HashSet(16, 0.75f).
  - **public** HashSet(Collection<? **extends** E> c)
    permet d'obtenir un HashSet de facteur de charge 0.75 et contenant tous les éléments de c.

TreeSet<E>

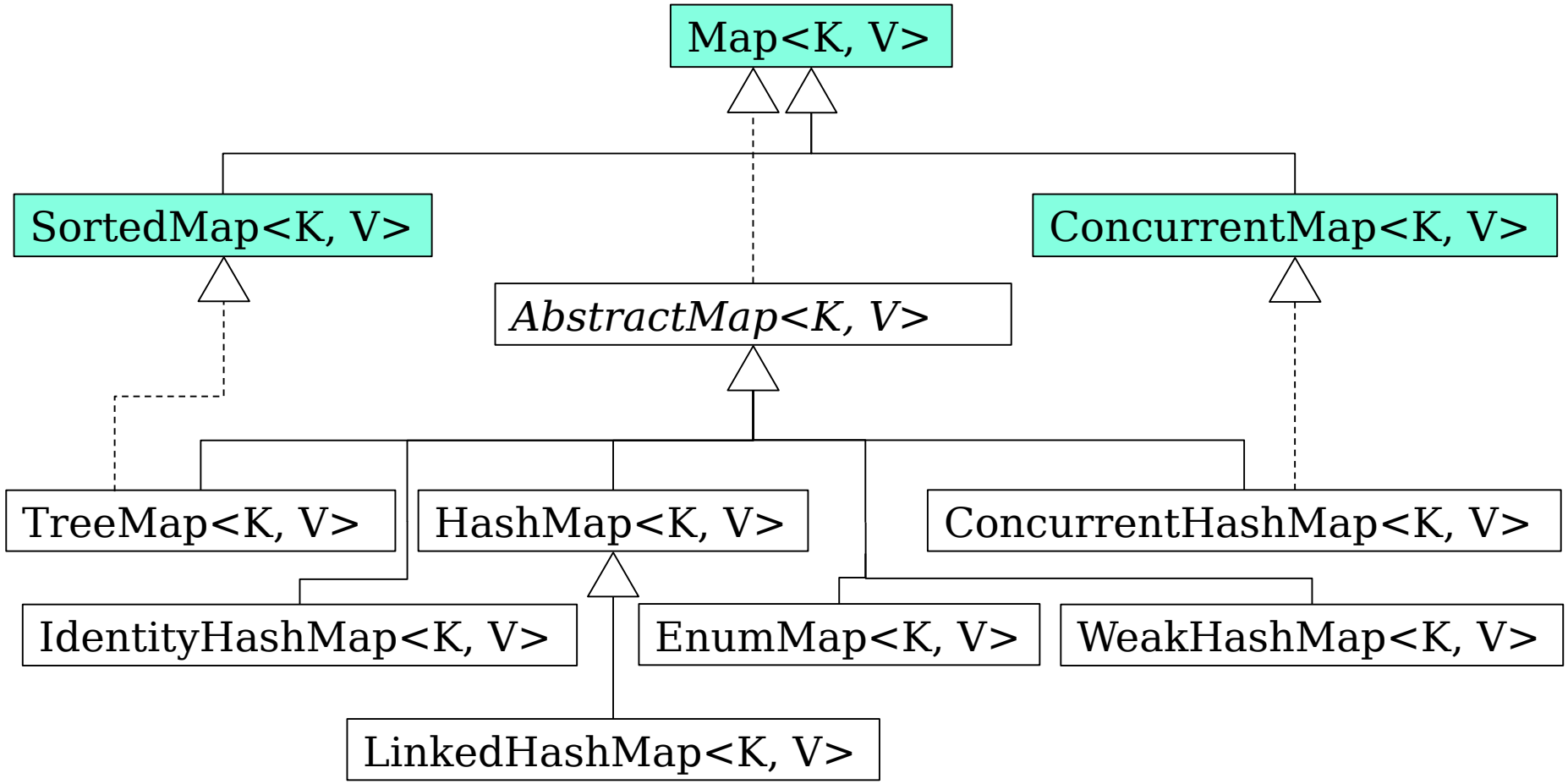
- Hérite d'AbstractSet<E> et propose une implémentation de SortedSet<E> basée sur une TreeMap<E, Object>, et réalise les interfaces Cloneable et java.io.Serializable.
- Un attribut localement déclaré
  - **private transient** TreeMap<E, Object> map;
- La plupart des méthodes héritées sont redéfinies par délégation explicite vers map.
- Définition de size()
  - ```
    public int size() {
        return map.size();
    }
    ```
    ```

TreeSet<E>

- Constructeurs
  - **public** TreeSet()
    permet d'obtenir un TreeSet vide dont l'ordre des éléments est l'ordre « naturel ».
  - **public** TreeSet(Comparator<? **super** E> comp)
    permet d'obtenir un TreeSet vide dont l'ordre des éléments est celui de comp.
  - **public** TreeSet(Collection<? **extends** E> c)
    permet d'obtenir un TreeSet, contenant tous les éléments de c, dans leur ordre « naturel ».
  - **public** TreeSet(SortedSet<E> s)
    permet d'obtenir un TreeSet contenant les mêmes éléments et utilisant le même ordre que s.

Maps

Interfaces et classes



Map<K, V>

- Une `Map` (parfois appelée table d'association ou encore dictionnaire) est un objet pouvant regrouper des associations clef-valeur.
- A l'instar d'un « dictionnaire » pour un langage naturel, la recherche d'une clef (un « mot ») est optimisée, mais pas la recherche d'une valeur (une « définition »).
- L'unicité des clefs est garantie dans une `Map` : il ne peut pas y avoir dans une `Map` deux associations `(k1, v1)` et `(k2, v2)` telles que `k1.equals(k2)` (`v1.equals(v2)` est par contre acceptable).
- Attention à l'utilisation d'objets mutables pour les clefs.
- Chaque association de la `Map` est du type `Entry<K, V>`, une interface imbriquée.

Map<K, V>

- Méthodes générales
  - **int** size()
    retourne le nombre d'associations de **this**.
  - **boolean** isEmpty()
    retourne **true** si **this** est vide.
  - V get(Object obj)
    retourne la valeur associée à la clef équivalente à obj si cette clef est présente dans **this**, **null** sinon.

Map<K, V>

- Méthodes de modification
  - V put(K k, V v)
    associe à la clef k la valeur v dans **this** et retourne la valeur précédemment associée à k. Si k n'était pas présente dans **this**, une association est créée et **null** est retourné. **Méthode « optionnelle »**.
  - V remove(Object obj)
    supprime dans **this** l'association ayant une clef équivalente à obj et retourne la valeur de cette association. Si aucune clef n'est équivalente à obj dans **this**, **null** est retourné. **Méthode « optionnelle »**.
  - **void** putAll(Map<? **extends** K, ? **extends** V> m)
    ajoute toutes les associations de m à **this**. Si des clefs de **this** sont aussi présentes dans m, les associations correspondantes sont écrasées. **Méthode « optionnelle »**.
  - **void** clear()
    supprime toutes les associations de **this**. **Méthode « optionnelle »**.

Map<K, V>

- Méthodes de recherche
  - V get(Object obj)
    retourne la valeur associée à la clef équivalente à obj si cette clef est présente dans **this**, **null** sinon.
  - **boolean** containsKey(Object obj)
    retourne **true** si une clef équivalente à obj est présente dans **this**.
  - **boolean** containsValue(Object obj)
    retourne **true** si au moins une valeur équivalente à obj est présente dans **this**. Attention, la recherche de valeurs n'est pas optimisée.

Map<K, V>

- Méthodes fournissant des vues
  - Set<K> keySet()
    retourne une vue Set des clefs présentes dans **this**.
  - Collection<V> values()
    retourne une vue Collection des valeurs présentes dans **this**.
  - Set<Map.Entry<K, V>> entrySet()
    retourne une vue Set des associations présentes dans **this**.

Map.Entry<K, V>

- Interface imbriquée dans Map pour représenter les associations, avec les méthodes suivantes :
  - `K getKey()`
    retourne la clef de l'association.
  - `V getValue()`
    retourne la valeur de l'association.
  - `V setValue(V v)`
    remplace la valeur de l'association par `v` et retourne la valeur remplacée.

SortedMap<K, V>

- Une SortedMap est une Map dans la laquelle les associations sont ordonnées (l'ordre est bien sûr défini sur les clefs).
- L'ordre des associations est déterminé à la création de la SortedMap :
  - ordre « naturel » si les clefs sont des Comparable.
  - ordre spécifique si un Comparator est fourni.

SortedMap<K, V>

- Méthodes additionnelles (SortedMap hérite de Map)
  - Comparator<? **super** K> comparator()
    retourne le Comparator associé à **this** ou **null** si **this** n'est pas muni d'un Comparator.
  - SortedMap<K> subMap(K from, K to)
    retourne une vue de **this** limitée aux associations dont la clef est comprise entre from inclus et to exclus. Levée d'une IllegalArgumentException si les valeurs de from et/ou to sont incorrectes.
  - SortedMap<K, V> headMap(K to)
    retourne une vue de **this** limitée aux associations de clef strictement inférieure à to.
  - SortedMap<K> tailMap(K from)
    retourne une vue de **this** limitée aux associations de clef supérieure ou équivalente à from.
  - K firstKey()
    retourne la première (la plus petite) clef de **this**. Levée d'une NoSuchElementException si **this** est vide.
  - K lastKey()
    retourne la dernière (la plus grande) clef de **this**. Levée d'une NoSuchElementException si **this** est vide.

AbstractMap<K, V>

- Classe abstraite racine de la hiérarchie d'héritage des Map, implémente Map<K, V>. Elle fournit une implémentation de base pour une Map.
- Aucun choix n'est fixé pour le stockage des associations : les recherches de clefs ne sont donc pas optimisées.
- Constructeur
  - **protected** AbstractMap()
- 1 seule méthode laissée abstraite
  - **public abstract** Set<Entry<K, V>> entrySet()

AbstractMap<K, V>

- Implémentation par défaut de toutes les autres méthodes imposées par Map<K, V> et redéfinition de toString()
  - **public int** size()
  - **public boolean** isEmpty()
  - **public void** clear()
  - **public boolean** containsKey(Object k)
  - **public boolean** containsValue(Object v)
  - **public** V get(Object k)
  - **public** V put(K k, V v)
  - **public** V remove(Object k)
  - **public void** putAll(Map<? **extends** K, ? **extends** V> m)
  - **public** Set<K> keySet()
  - **public** Collection<V> values()

AbstractMap<K, V>

- La méthode put(K k, V v) est laissée optionnelle …

```
public V put(K k, V v) {
    throw new UnsupportedOperationException();
}
```

- Les autres méthodes de modification (clear(), remove(Object k), putAll(K k, V v)) reposent toutes sur l'utilisation de entrySet().

HashMap<K, V>

- Hérite d'AbstractMap<K, V>, propose une implémentation de Map<K, V> basée sur une table de hachage et réalise aussi les interfaces Cloneable et java.io.Serializable.
- La table de hachage est automatiquement agrandie si besoin.
- Six attributs localement déclarés :
  - **transient** Node<K, V>[] table;
  - **transient** Set<Map.Entry<K, V>> entrySet;
  - **transient int** size;
  - **transient int** modCount;
  - **int** threshold;
  - **final float** loadFactor;

HashMap<K, V>

- Constructeurs
  - **public** HashMap(**int** initialCapacity, **float** loadFactor)
    permet d'obtenir une HashMap vide de capacité initiale initialCapacity et de facteur de charge loadFactor.
  - **public** HashMap(**int** initialCapacity)
    équivalent à HashMap(initialCapacity, 0.75f).
  - **public** HashMap()
    équivalent à HashMap(16, 0.75f).
  - **public** HashMap(Map<? **extends** K, ? **extends** V> m)
    permet d'obtenir une HashMap contenant initialement toutes les associations de m. Son facteur de charge est 0.75.

HashMap<K, V>

- Une classe imbriquée :

```
static class Node<K, V> implements Map.Entry<K, V> {
    final int hash;
    final K key;
    V value;
    Node<K, V> next;

    Node(int hash, K key, V value, Node<K, V> next) {
        //...
     }

    public final K getKey() { return key; }
    public final V getValue() { return value; }
    public final String toString() {
        return key + "=" + value;
    }
    public final V setValue(V newValue) {
        V oldValue = value;
        value = newValue;
        return oldValue;
    }
    // ...
}
```

TreeMap<K, V>

- Hérite d'AbstractMap<K, V>, propose une implémentation de SortedMap<K, V> (plus spécifiquement NavigableMap<K, V> depuis Java 1.6) basée sur un arbre binaire ordonné équilibré et réalise aussi les interfaces Cloneable et java.io.Serializable.
- De nombreuses classes imbriquées dont Entry<K, V> qui implémente Map.Entry<K, V>.
- Quatre attributs localement déclarés :
  - **private final** Comparator<? **super** K> comparator;
  - **private transient** Entry<K, V> root;
  - **private transient int** size;
  - **private transient int** modCount;

TreeMap<K, V>

- Constructeurs
  - **public** TreeMap()
    permet d'obtenir une TreeMap vide basée sur l'ordre « naturel » de ses clefs. Toutes ses clés devront être des Comparable.
  - **public** TreeMap(Comparator<? **super** K> comp)
    permet d'obtenir une TreeMap vide basée sur comp pour l'ordre de ses clefs.
  - **public** TreeMap(Map<? **extends** K, ? **extends** V> m)
    permet d'obtenir une TreeMap contenant initialement toutes les associations de m, et basée sur l'ordre naturel de ses clefs. Toutes ses clés devront être des Comparable.
  - **public** TreeMap(SortedMap<? **extends** K, ? **extends** V> m)
    permet d'obtenir une TreeMap contenant initialement toutes les associations de m, et basée sur le même ordre que m.