Les collections dans Java 2

...

et quelques nouveautés de Java 5.0

Les collections

- Collection :
  - *objet qui regroupe de multiples éléments dans une seule entité.*
- Utilisation de collections pour
  - *stocker, retrouver et manipuler des données*
  - *transmettre des données d ’une méthode à une autre*
- Exemples :
  - *un dossier de courrier : collection de mails*
  - *un répertoire téléphonique : collection d ’associations noms numéros de téléphone.*

Les collections dans Java

- Dans les premières versions de Java quelques implémentations pour différents types de collections :
  - *tableaux*
  - ***`Vector`*** *: tableaux dynamiques (listes)*
  - ***`Hastable`*** *: tables associatives*

- Une **modification majeure** de Java 2 a été d'inclure un véritable « **framework** » pour la gestion des collections (package **`java.util`**)
  - *architecture unifiée pour représenter et manipuler les collections*
  - *composée de trois parties*
    - *une hiérarchie d'interfaces permettant de représenter les collections sous forme de types abstraits*
    - *des implémentations de ces interfaces*
    - *des algorithmes réalisant des opérations fréquentes sur les collections (recherche, tri…)*
  - *similaire à STL (Standard Template Library) en C++ et au collection framework de Smalltalk.*

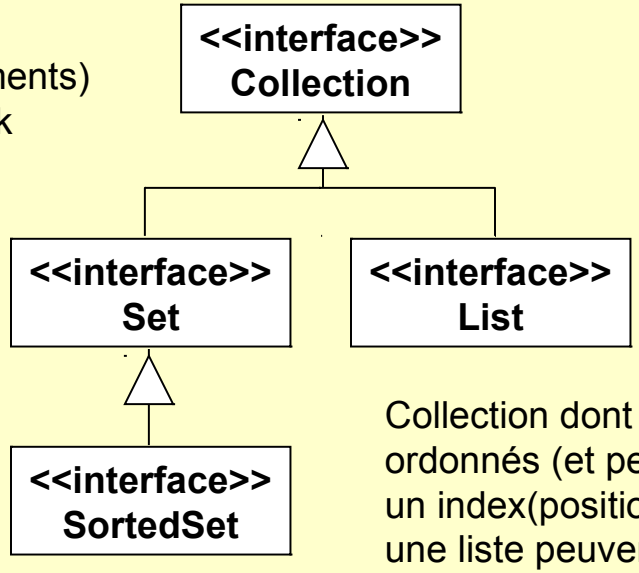
Les collections dans Java 2

- Avantages d'un « framewok » pour les collections :
  - *réduction de l'effort de programmation*
  - *amélioration de la qualité et de la performance des programmes*
  - *permet interoperabilité entre des API non directement liées*
  - *facilite l'apprentissage et l'utilisation de nouvelles API*
  - *réduit l 'effort de conception de nouvelles API*
  - *encourage la réutilisation du logiciel*

- Peut être perçu à premier abord comme étant plutôt complexe
  - *le framework proposé pour java ne l'est pas tant que cela (dixit les auteurs du « Java Tutorial », on n 'est pas obligés d 'être d'accord…)*
  - *le retour sur investissement est important*

Les interfaces

Racine de la hiérarchie
- représente un groupe d 'objets (les éléments)
- pas d 'implémentation directe dans le jdk



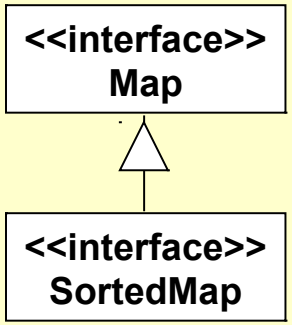
Collection dont les éléments ne peuvent être dupliqués (abstraction de la notion d'ensemble en mathématiques)

Ensemble trié

Collection dont les éléments sont ordonnés (et peuvent être accédés via un index(position)). Les éléments dans une liste peuvent être dupliqués.

Associe un objet à une clé (à une clé ne peut être associé qu'un seul objet)

Une « Map » qui maintient ses clés triées selon un ordre ascendant (par exemple une dictionnaire, un annuaire téléphonique)



Interface Collection

```
public interface Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

Une implémentation donnée de cette interface n'est pas obligée de supporter ces opérations.
Si il n'y a pas de support pour une méthode une `UnsupportedOperationException` sera lancée

basée sur `equals`

`true` si l'opération a modifié la collection

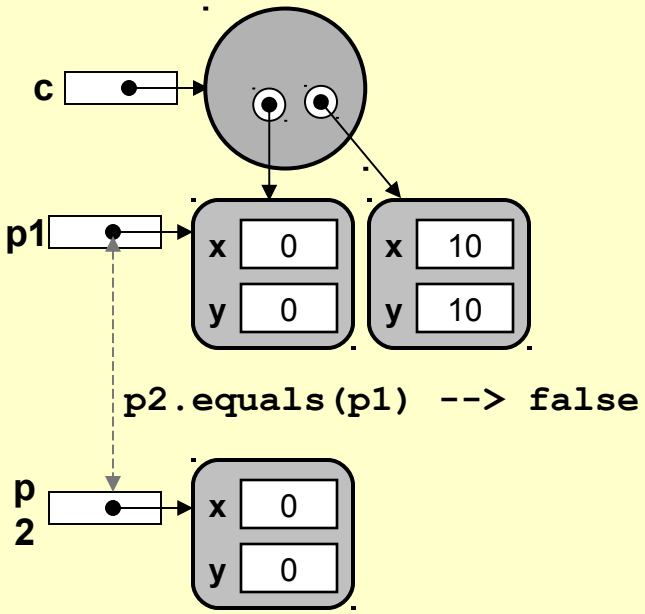
parcourt séquentiel des éléments de la collection

Interface Collection

- `boolean contains(Object o)`
  - *renvoie* `true` *si la Collection contient l'objet* `o`, `false` *sinon*
    - *renvoie true si cette collection contient au moins un élément* `e` *tel que* `(o==null ? e==null : o.equals(e))`.

```
public class Point {
  private int x;
  private int y;

  public Point(int x, int y) {
     this.x = x;
     this.y = y;
  }

  ...



}
```

```
// c une reference vers
// une collection vide

Point p1 = new Point(0,0);

c.add(p1);

c.add(new Point(10,10));

c.contains(p1)   --> true

Point p2 = new Point(0,0);

c.contains(p2)   --> false
```



La méthode `equals` exécutée est la méthode héritée de `Object`, définie par :

```
public boolean equals(Object o) {
        return this == o;
}
```

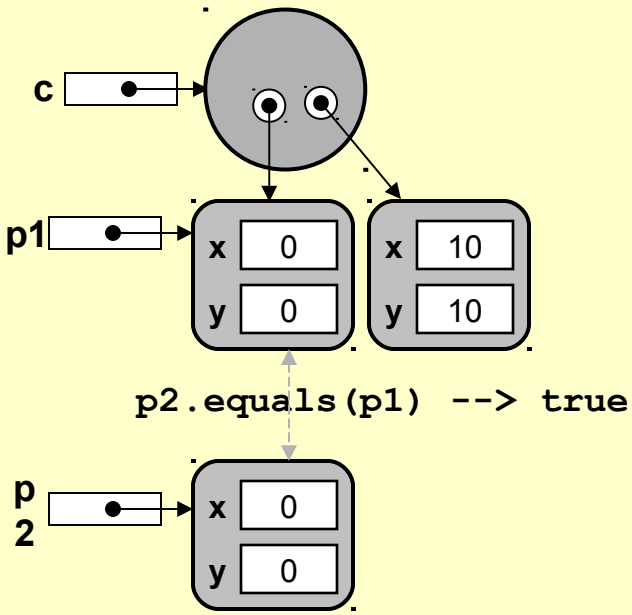
Interface Collection

- **Redéfinition de la méthode `boolean equals(Object o)`**

```
public boolean equals(Object o) {
      if (!(o instanceof Point)
        return false;
      Point p = (Point) o;
      return this.x == p.x &&
                this.y == p.y;
}
```

```
public class Point {
  private int x;
  private int y;

  public Point(int x, int y) {
     this.x = x;
     this.y = y;
  }

  ...
  public boolean equals(Object o)
  {
     ...
  }
}
```

```
// c une reference vers
// une collection vide

Point p1 = new Point(0,0);

c.add(p1);

c.add(new Point(10,10));

c.contains(p1)  --> true

Point p2 = new Point(0,0);

c.contains(p2)  --> true
```



A propos de `equals`

- **Pourquoi ne pas définir equals avec la signature `boolean equals(Point o)` ?**

```
public class Point {
  private int x;
  private int y;

  public Point(int x, int y) {
     this.x = x;
     this.y = y;
  }
```

Surcharge de equals

```
  public boolean equals(Point o) {
      return this.x == o.x && this.y == o.y;
  }
```

```
  public boolean equals(Object o) {
       return this == o;
  }
```

Méthode equals héritée de Object

```
}
```

```
Point p1 = new Point(0,0);
Point p2 = new Point(0,0);
p1.equals(p2)   --> true

Object o = p2;
p1.equals(o)    --> false
```

Interface Collection

- Méthodes **`add`** et **`remove`** définies de manière suffisamment générale pour avoir un sens aussi bien pour les collections autorisant la duplication des éléments (**`List`**) que pour celles ne l'autorisant pas (**`Set`**).

- **`boolean add(Object o)`**
  - *garantit que la Collection contiendra `o` après exécution de cette méthode*
  - *renvoie **`true`** si la Collection a été modifiée par l'appel*
  - *renvoie **`false`** si la Collection n'a pas été modifiée*

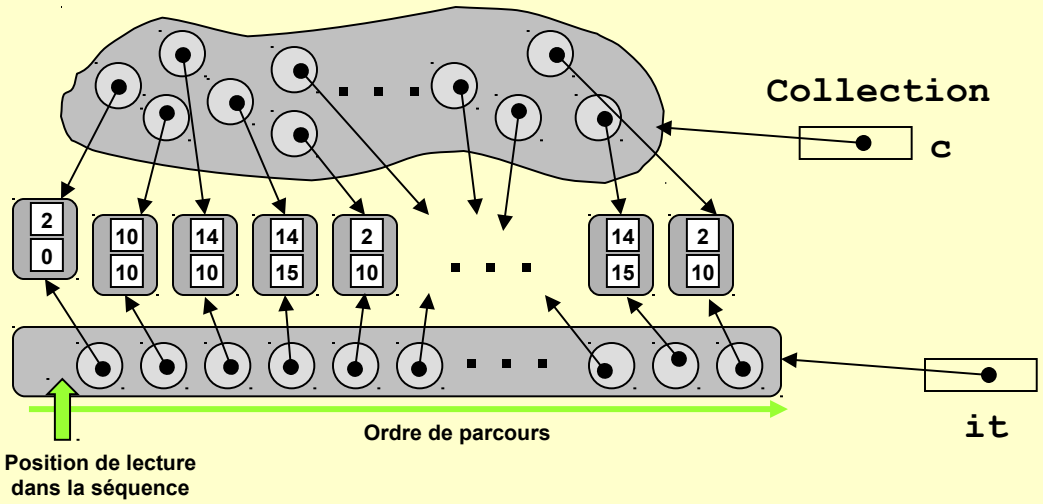
- **`boolean remove(Object o)`**
  - *retire une instance d'un élément `e` tel que :*

    `(o==null ? e==null : o.equals(e))`
  - *renvoie **`true`** si la Collection a été modifiée par l'appel*
  - *renvoie **`false`** si la Collection n'a pas été modifiée*

Iterator

- `Iterator iterator()`
  - *cette méthode renvoie un objet « iterateur » qui permet le parcours séquentiel des éléments d'une collection.*
    - *Pas de garantie concernant l'ordre dans lequel les éléments sont retournés*
- `Iterator` est une interface définie dans le package **java.util**

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();    // Optional
}
```




```
Iterator it = c.iterator();
```

Fourni un itérateur sur la collection

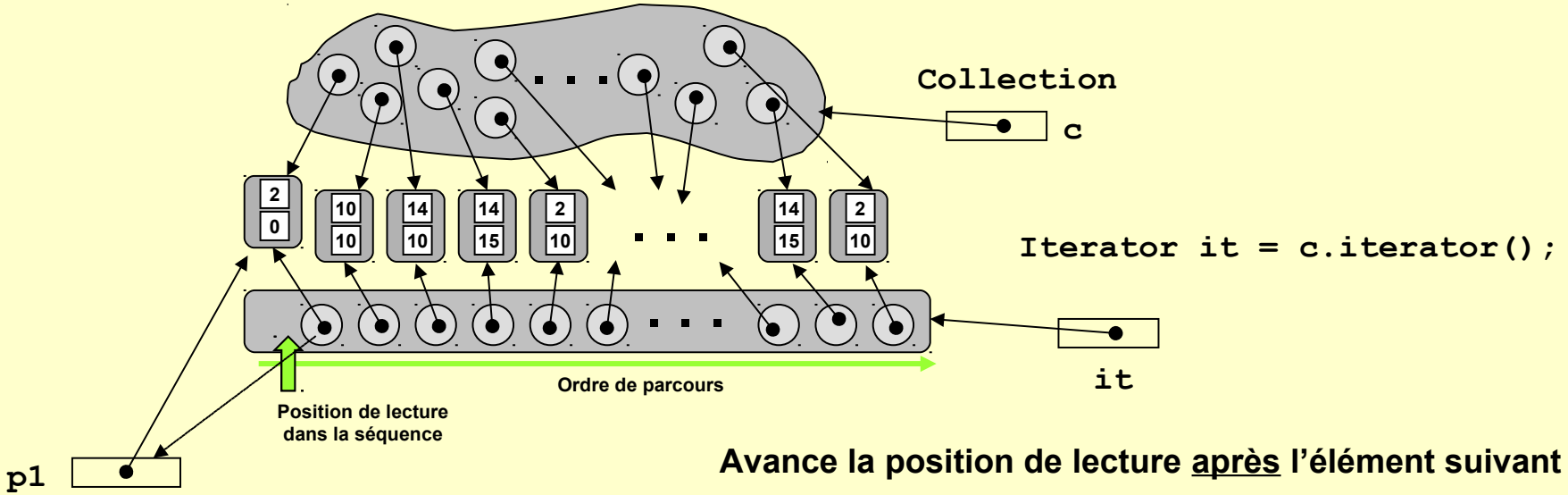
Un ordre de parcours

Une position de parcours

Iterator

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Permet d'effectuer une lecture séquentielle



Avance la position de lecture <u>après</u> l'élément suivant

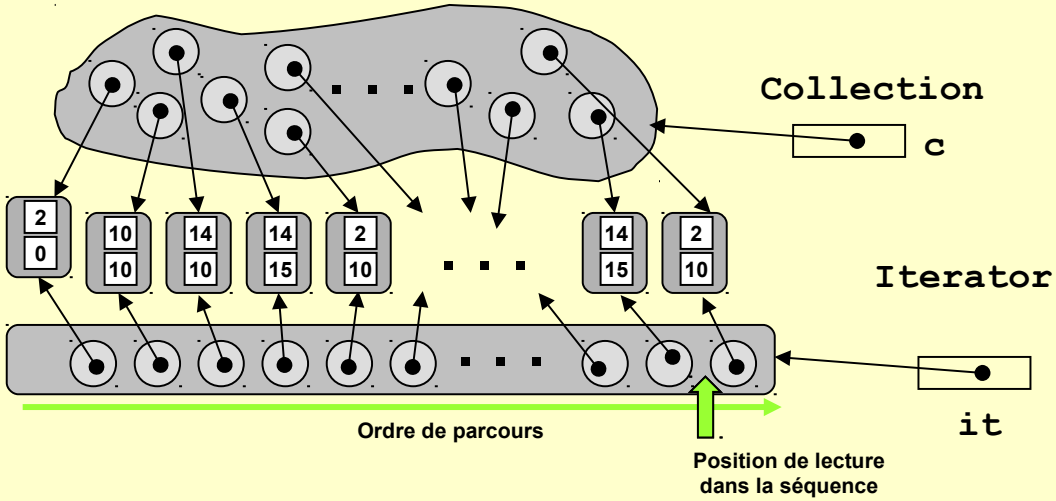
```
Point p1 =(Point) it.next();
```

Permet de récupérer la référence de l'élément lu

Iterator

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Test de fin de parcours



`true` si il y a encore au moins un élément suivant

`it.hasNext()`

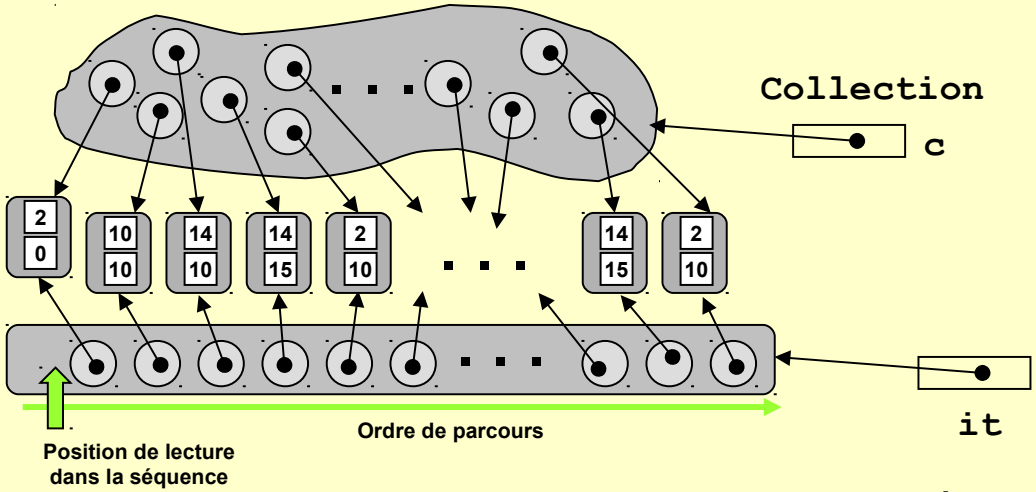
`it.next()`

`it.hasNext()`

`false` si il y a plus d'élément suivant

Iterator

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```



Parcours séquentiel de tous les éléments de la collection

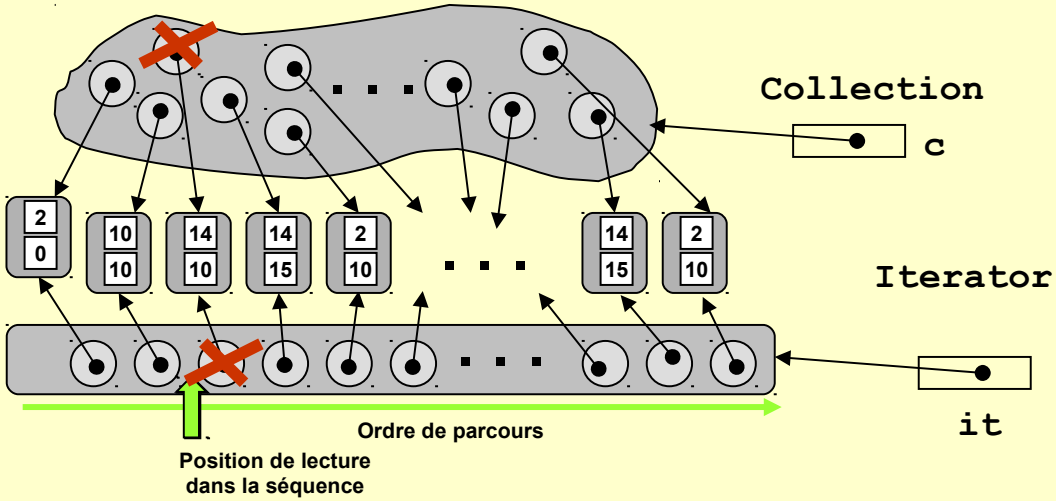
```
Iterator it = c.iterator();

while (it.hasNext()) {
  Point p = (Point) it.next();
  System.out.println(p.distance());
}
```

Iterator

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```

Retire un élément de la collection lors d'un parcours itératif

`Iterator.remove` est la seule manière sûre de modifier une collection lors d'un parcours itératif



retire de la collection le dernier élément « lu » (élément qui précède la position de lecture)

```
it.next()
it.remove()
it.remove()
```

ne peut être invoqué qu'une fois par appel de `next`

Iterator

- Exemple : utilisation d'un itérateur pour filtrer une collection

```
static void filter(Collection c) {
    Iterator it = c.iterator()
    while (it.hasNext()){
        if (!cond(it.next()))
            it.remove();
    }
}
```

```
static void filter(Collection c) {
    for (Iterator it = c.iterator(); it.hasNext(); )
        if (!cond(it.next()))
            it.remove();
}
```

- Le code est polymorphe (polymorphique ?)
  - *fonctionne pour n'importe quelle* `Collection` *qui supporte la suppression d'élément, quelle que soit son implémentation*

Interface Set

```
public interface Set extends Collection {
    // Basic Operations
    int size();
    boolean isEmpty();
    boolean contains(Object element);
    boolean add(Object element);    // Optional
    boolean remove(Object element); // Optional
    Iterator iterator();

    // Bulk Operations
    boolean containsAll(Collection c);
    boolean addAll(Collection c);    // Optional
    boolean removeAll(Collection c); // Optional
    boolean retainAll(Collection c); // Optional
    void clear();                    // Optional

    // Array Operations
    Object[] toArray();
    Object[] toArray(Object a[]);
}
```

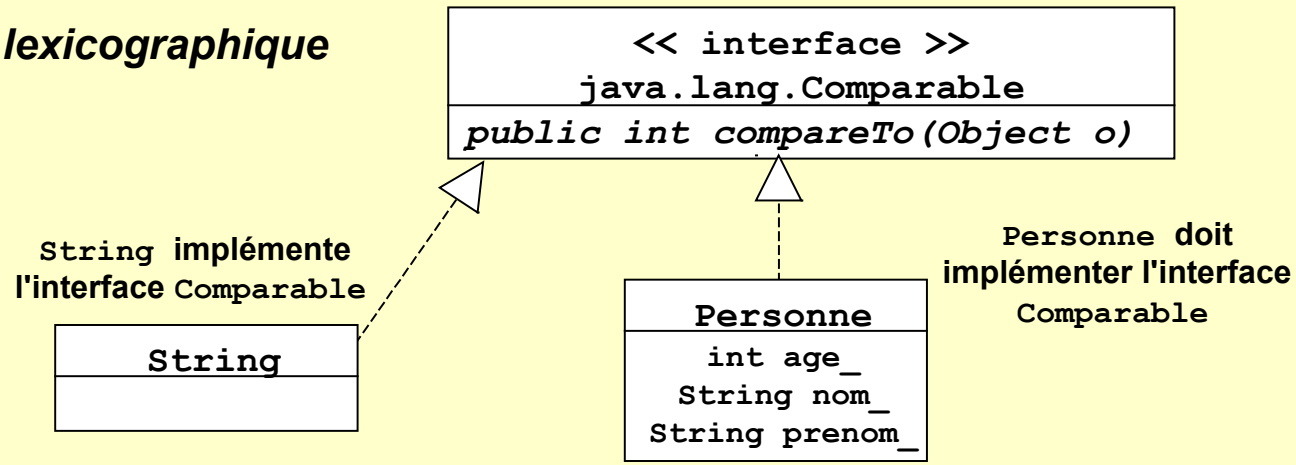
ne définit pas de nouvelles méthodes, Ajoute simplement des restrictions sur les définitions des opérations en interdisant la duplication des éléments

Interface SortedSet

- Ensemble qui garantit que son iterator fera un parcours de ses éléments selon un ordre ascendant
- Comment cet ordre est-il défini ?
  - *lorsque l'on ajoute un élement à un ensemble, l'ensemble doit comparer celui-ci avec les autres éléments déjà présents dans l'ensemble*
  - *c'est réalisé par l'envoi du message compareTo(Object) à l'élément*
  - *cette méthode est définie dans l'interface `java.lang.Comparable`*

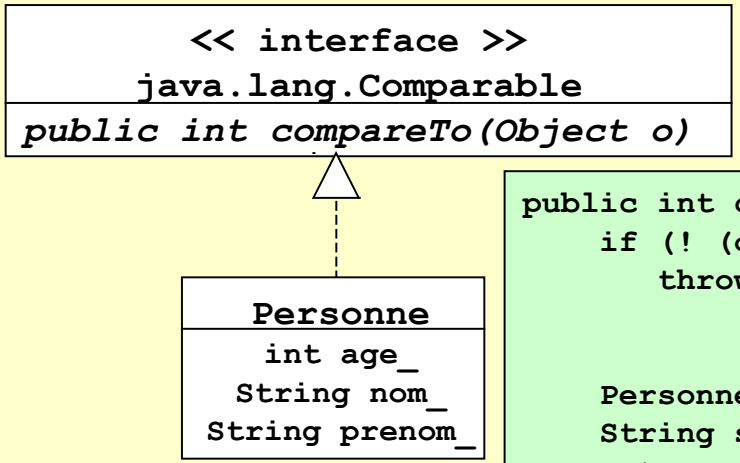
```
public int compareTo(Object o)

    Compares this object with the specified object for order. Returns a negative integer, zero, or a positive
    integer as this object is less than, equal to, or greater than the specified object.
```

  - *String : ordre lexicographique*
  - *Personne : ?*



Interface SortedSet

- **Pour être inséré dans une collection triée un objet doit implémenter l'interface java.lang.Comparable**




```
public int compareTo(Object o) {
    if (! (o instanceof Personne) )
       throw new ClassCastException("impossible de comparer un " +
             o.getClass().getName() + " à une Personne");

    Personne p = (Personne) o;
    String s = this.nom_ + this.prenom_   ;
    return s.compareTo(p.nom_ + p.prenom_);
}
```

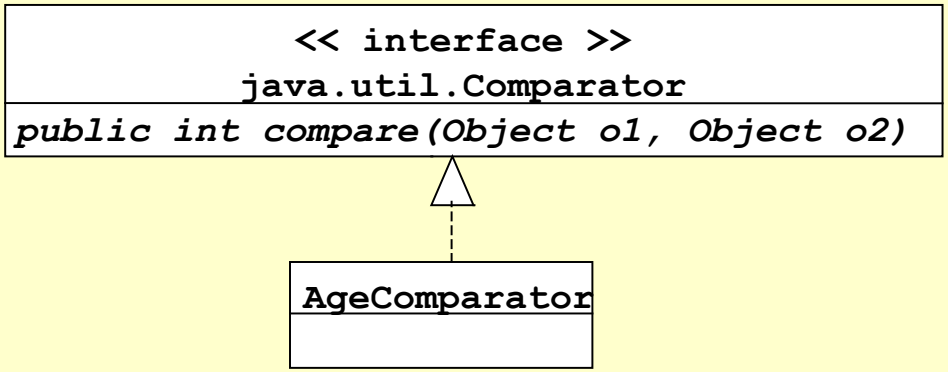
- **Et si on veut un ensemble de personnes triées selon leur âge ?**
  - ***en spécifiant dans le constructeur de l'ensemble un objet qui sera chargé de réaliser les comparaisons***
  - ***cet objet doit implémenter l'interface `java.util.Comparator`***

```
public int compare(Object o1,
                   Object o2)
```

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

Interface SortedSet

- Définition d'un `Comparator` pour comparer les personnes selon leur âge




```
public class AgeComparator implements java.util.Comparator {

  public int compare (Object o1, Object o2) {
    if (! ( (o1 instanceof Personne) && (o2 instanceof Personne) ))
       throw new ClassCastException("o1 et o2 doivent être des personnes");

    int age1 = ((Personne) o1).getAge();
    int age2 = ((Personne) o2).getAge();
    if (age1 < age2)
        return -1;
    else if (aga1 == age2)
        return 0;
    else
        return 1;
  }

}
```

Interface SortedSet

Ensemble qui garantit que son iterator fera un parcours de ses éléments selon un ordre ascendant

```
public interface SortedSet extends Set {
   public Comparator comparator();
         // Returns the comparator associated with this sorted set, or null if it
         // uses its elements' natural ordering.
   public Object first();
         // Returns the first (lowest) element currently in this sorted set.
   public SortedSet headSet(Object toElement);
        // Returns a view of the portion of this sorted set whose elements
        // are strictly less than toElement.
   public Object last();
         // Returns the last (highest) element currently in this sorted set.
   public SortedSet subSet(Object fromElement, Object toElement);
         // Returns a view of the portion of this sorted set whose elements range
        // from fromElement, inclusive, to toElement, exclusive.
   public SortedSet tailSet(Object fromElement);
       //Returns a view of the portion of this sorted set whose
       //elements are greater than or equal to fromElement

}
```

Interface List

Liste : collection ordonnée.

```
public interface List extends Collection {
    // Positional Access
    Object get(int index);
    Object set(int index, Object element);  // Optional
    void add(int index, Object element);    // Optional
    Object remove(int index);               // Optional
    boolean addAll(int index, Collection c);// Optional

    // Search
    int indexOf(Object o);
    int lastIndexOf(Object o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

Les éléments d'une liste sont accessibles par leur position (entier entre 0 et size()-1)

Itérateur plus riche permettant de parcourir la liste dans les deux sens

Interface Map

```
public interface Map {
    // Basic Operations
    Object put(Object key, Object value);
    Object get(Object key);
    Object remove(Object key);
    boolean containsKey(Object key);
    boolean containsValue(Object value);
    int size();
    boolean isEmpty();

    // Bulk Operations
    void putAll(Map t);
    void clear();

    // Collection Views
    public Set keySet();
    public Collection values();
    public Set entrySet();

    // Interface for entrySet elements
    public interface Entry {
        Object getKey();
        Object getValue();
        Object setValue(Object value);
    }
}
```

Associe une valeur à une clé

On accède directement à la valeur à partir de la clé

Les clés forment un ensemble (chaque clé est unique)

Les valeurs constituent une collection

Map.Entry définit un couple clé valeur

Interface Map

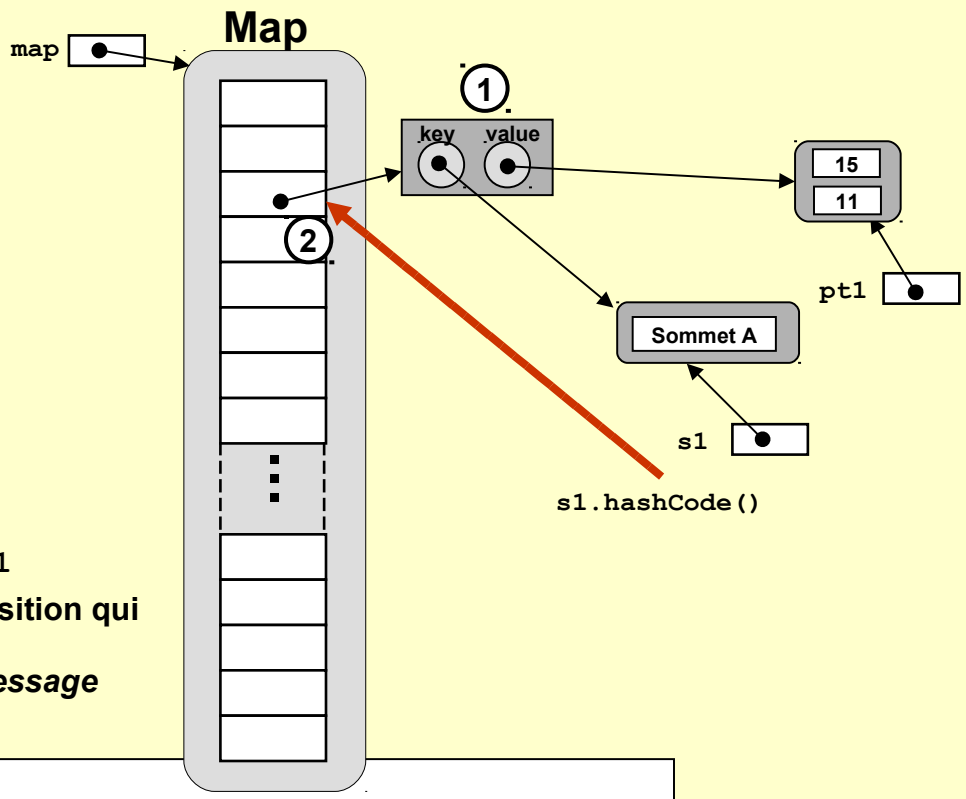
Map : abstraction de structure de données avec adressage dispersé (hashCoding)

La Map peut être vue comme une table de couples clé/valeur (Map.Entry)

```
Map map = new HashMap();
```

Ajouter une entrée à la Map

```
String s1 = new String("Sommet A");
Point pt1 = new Point(15,11);
map.put(s1,pt1);
```



1) Créer un objet `Map.Entry` référençant `s1` et `pt1`

2) Ranger cette `Map.Entry` dans la table à une position qui dépend de la clé (`s1`)

Cette position est calculée en envoyant le message `hashCode()` à l'objet clé (`s1`)

`public int` **`hashCode`**`()`

Returns a hash code for this string. The hash code for a `String` object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

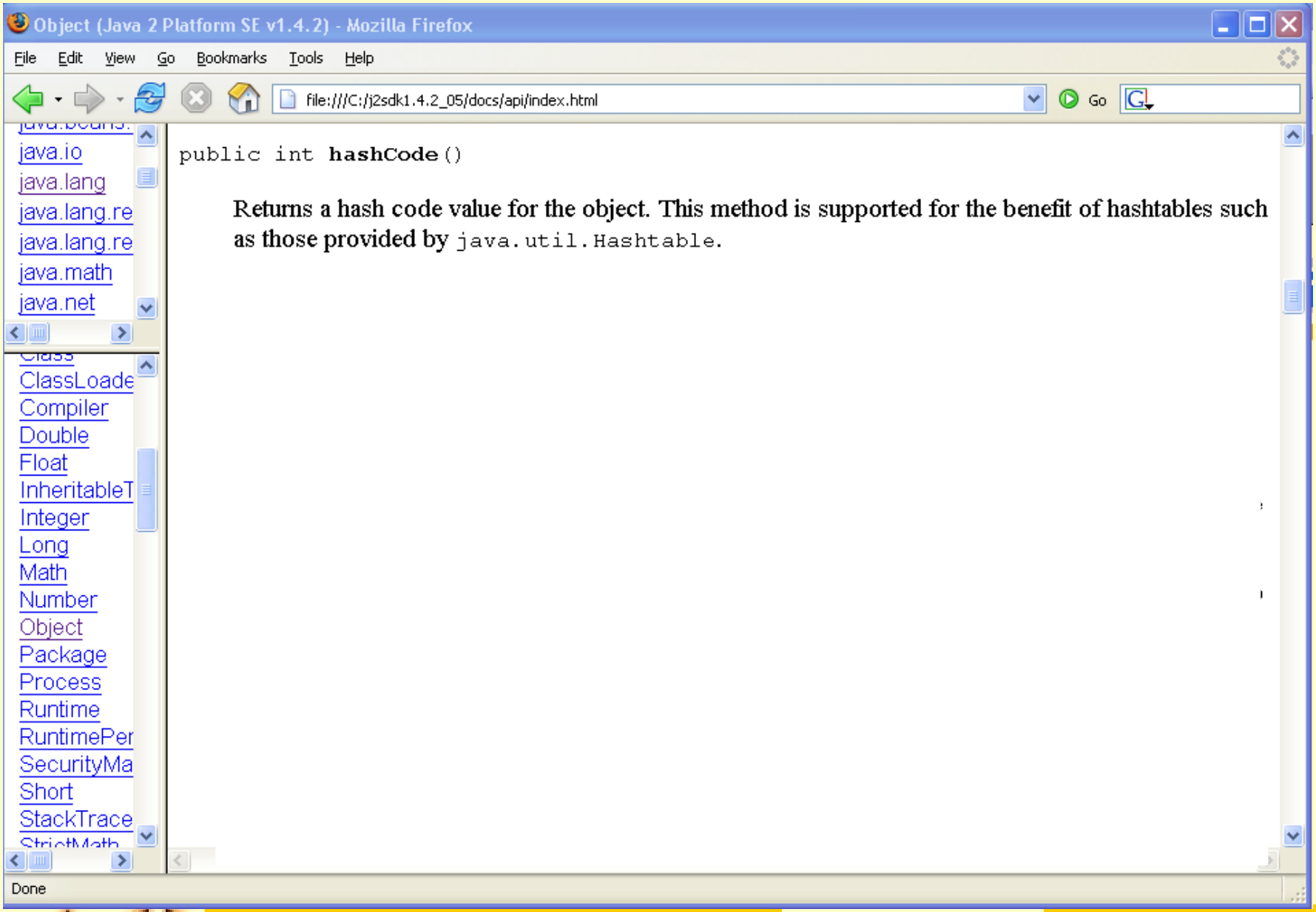
using `int` arithmetic, where `s[i]` is the *i*th character of the string, `n` is the length of the string, and `^` indicates exponentiation. (The hash value of the empty string is zero.)

Overrides:

`hashCode` in class `Object`

Tout objet possède une méthode `hashCode()`

Interface Map



Interface Map

```
map.put(k1,v1);
map.put(k2,v2);
```

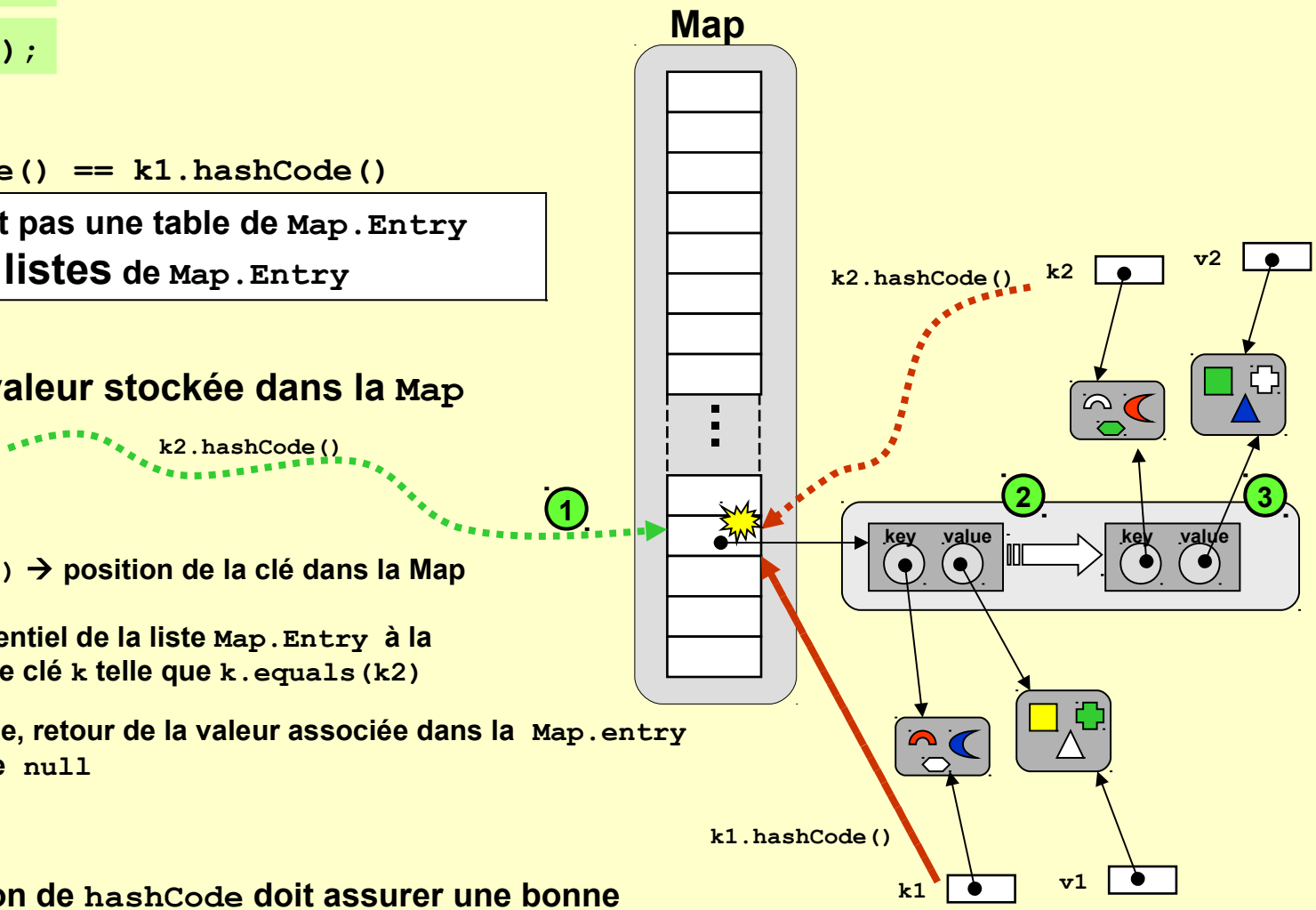
Collision

`k2.hashCode() == k1.hashCode()`

En fait la `Map` n'est pas une table de `Map.Entry` mais une table de **listes** de `Map.Entry`

Retrouver une valeur stockée dans la `Map`

```
map.get(k2);
```



1. `k2.hashCode()` → position de la clé dans la Map
2. Parcours séquentiel de la liste `Map.Entry` à la recherche d'une clé `k` telle que `k.equals(k2)`
3. Si `k` est trouvée, retour de la valeur associée dans la `Map.entry`
   Sinon retour de `null`

Une bonne fonction de `hashCode` doit assurer une bonne dispersion des clés en minimisant les collisions

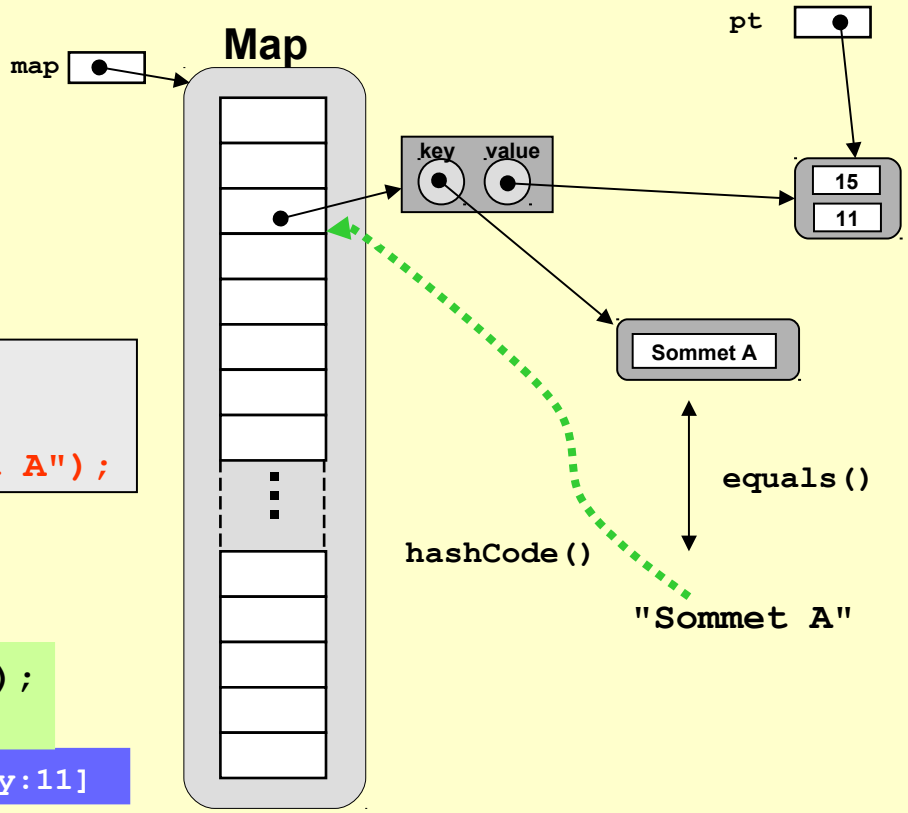
Interface Map

```
Point pt = map.get("Sommet A");
System.out.println(pt);
```

```
found    : java.lang.Object
required: Point
          Point pt = map.get("Sommet A");
```

```
Point pt =  (Point) map.get("Sommet A");
System.out.println(pt);
```

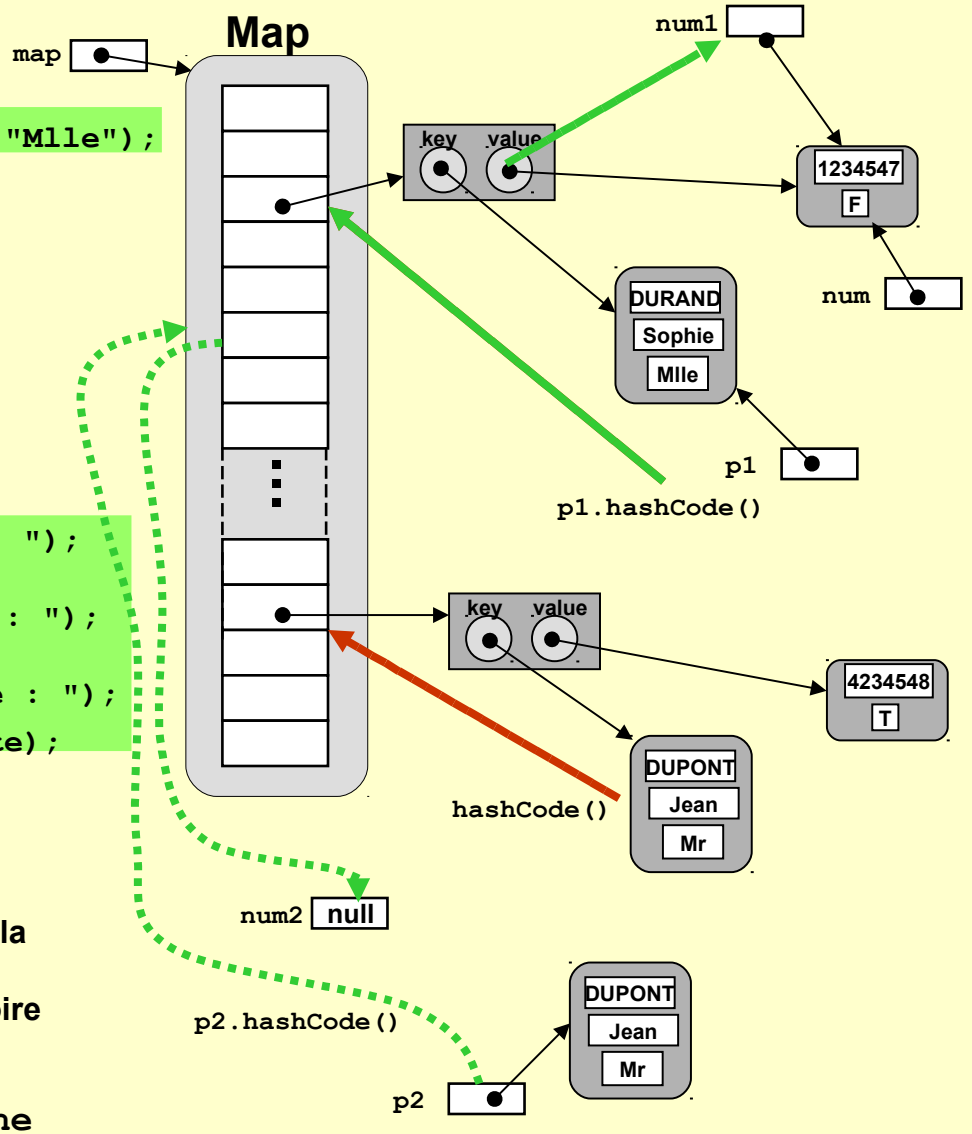
→ `Point[x:15,y:11]`



Interface Map

```
Map map = new HashMap();
Personne p1 = new Personne("DURAND","Sophie","Mlle");
NumTel num = new NumTel("1234547",'F');
map.put(p1,num);
NumTel num1 = map.get(p1);
System.out.println(num1);
```
→ `1234547 (F)`

```
map.put(new Personne("DUPONT","Jean","Mr"),
        new NumTel("4234548",'D'));
String nom = LectureClavier.lireChaine("Nom : ");
String prenom =
          LectureClavier.lireChaine("Prénom : ");
String civilite =
         LectureClavier.lireChaine("Civilité : ");
Personne p2 = new Personne(nom,prenom,civilite);
NumTel num2 = map.get(p2);
System.out.println(num2);
```
→ `null` ???



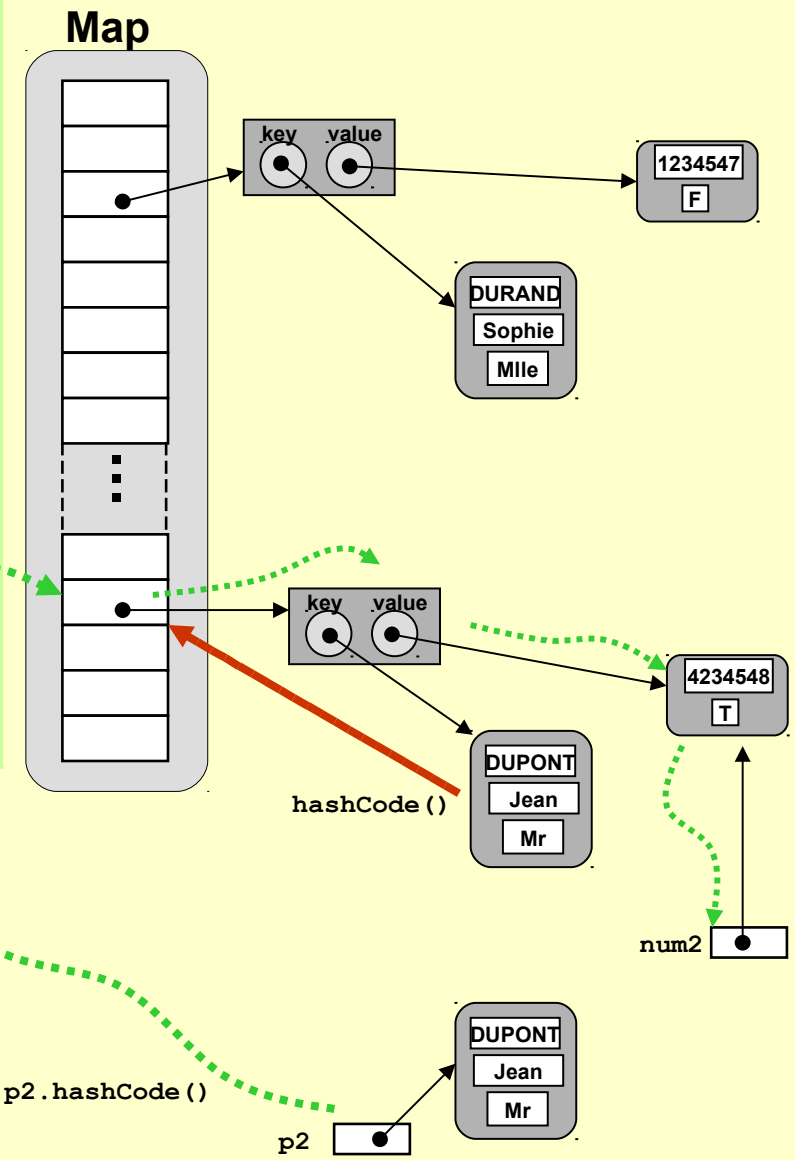
`p2` est un objet distinct de la clé (même si il représente la même personne) et la fonction `hashCode()` héritée de `Object` base son calcul sur la référence (adresse mémoire de cet objet)

→ il faut redéfinir `hashCode()` dans `Personne`

Interface Map

```
public class Personne {
  ...
  public boolean equals(Object o) {
     if (! (o instanceof Personne) )
       return false;

     Personne p = (Personne) o;
     return civilite_ == p.civilite_&&
            nom_.equals(p.nom_) &&
            prenom_.equals(p.prenom_);

 }

  public int hashCode() {
      String nomPlusPrenom = nom_ + prenom_;
      return nomPlusPrenom.hashCode() + civilite_;
  }

}
```

ATTENTION : les `String` sont des objets, bien penser à utiliser `equals` et non pas == pour comparer des chaînes

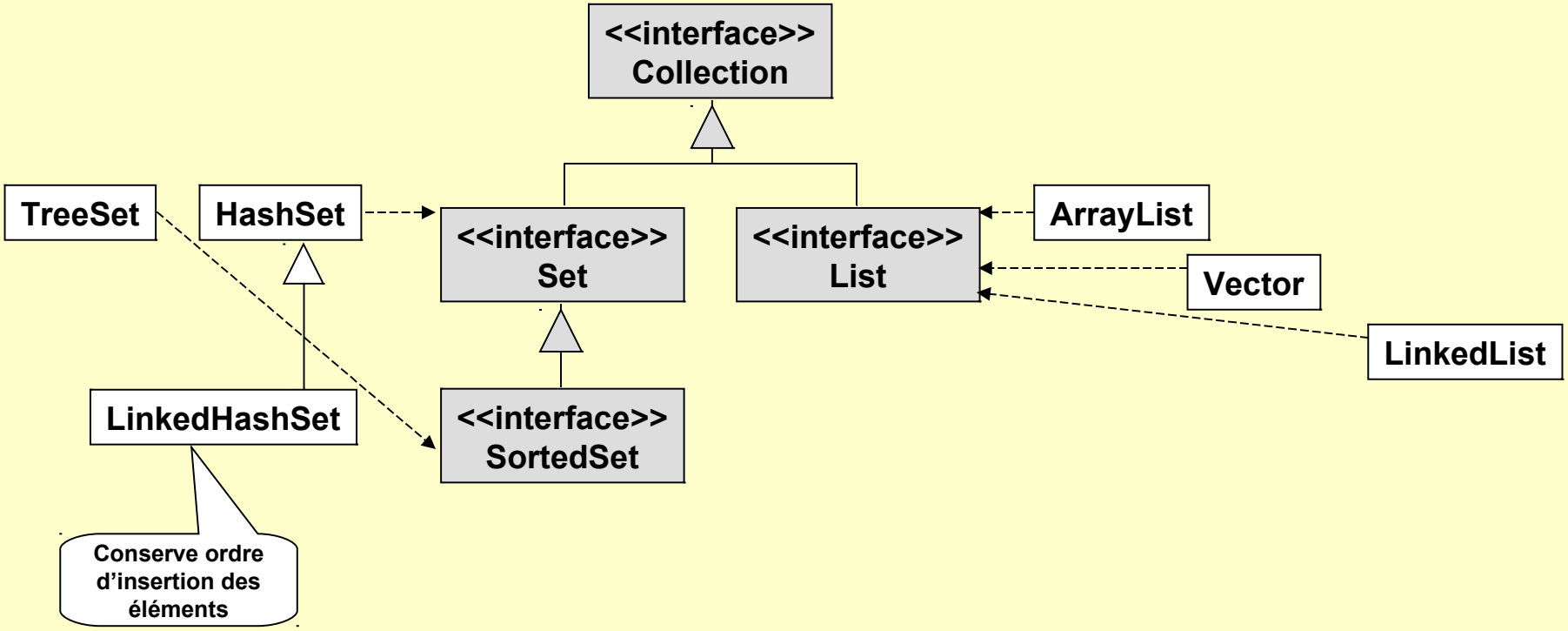



```
Personne p2 = new Personne("DUPONT","Jean","Mr");
NumTel num2 = map.get(p2);
System.out.println(num2); → 4234548 (T)
```

`hashCode()` doit TOUJOURS être redéfinie en cohérence avec `equals()`

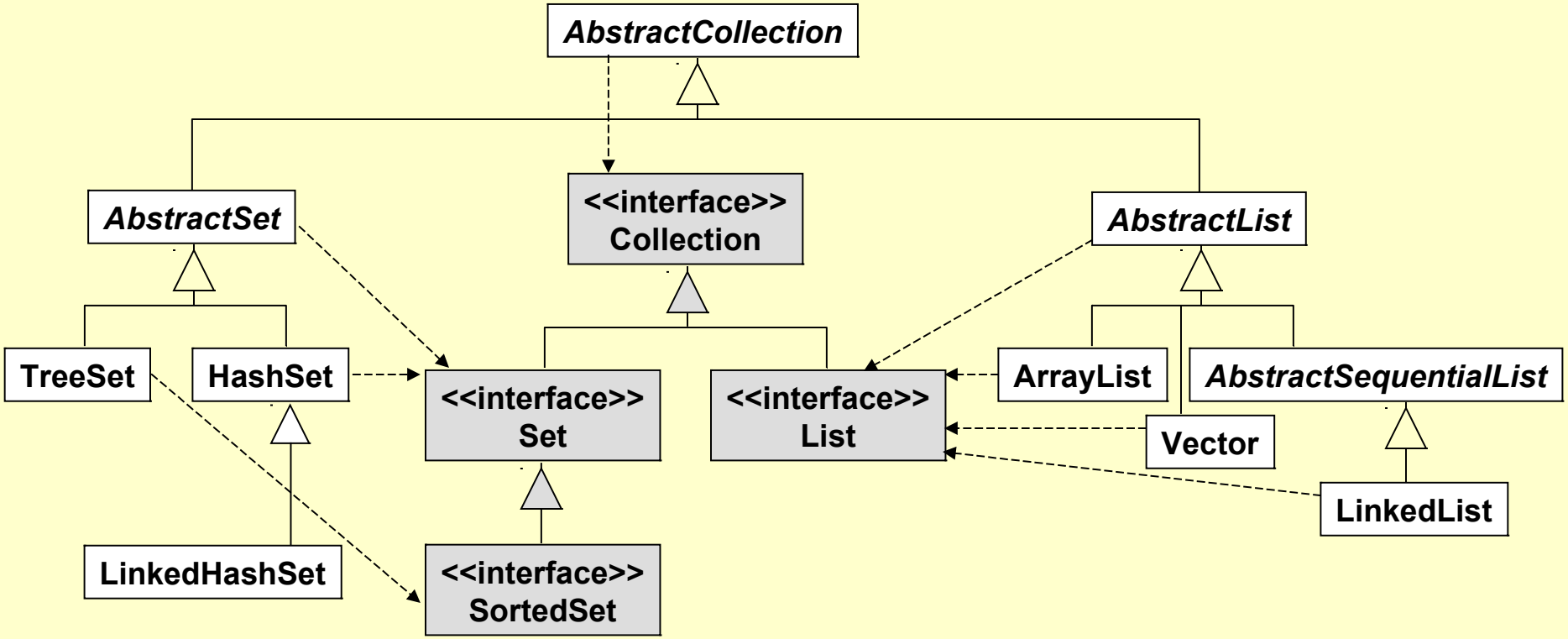
Les implémentations

- **Le jdk propose plusieures implémentations standards pour ces interfaces**
  - *le choix d'une implémentation plutôt qu'une autre peut être guidé par des considérations de performances*



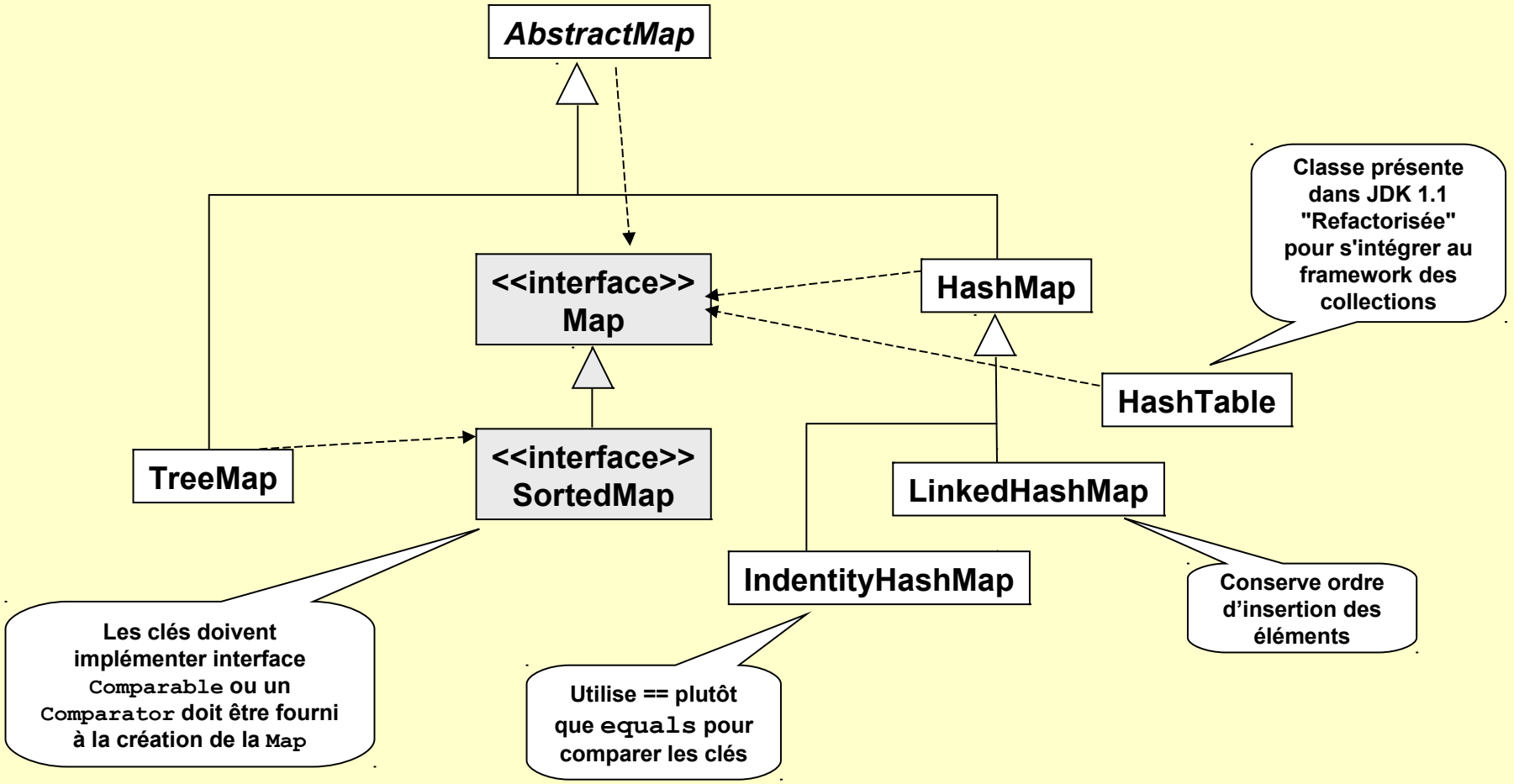
Les implémentations

- **Ces implémentations se situent dans une hiérarchie de classes abstraites**
  - *factorisation de code, réutilisation pour des implémentations spécifiques*



Les implémentations

- Les implémentations de `Map` suivent la même structure



Ok, mais comment cela marche ?

- Les interfaces sont au cœur du « framework »
- Une fois que l'on a compris comment les utiliser on a compris l 'essentiel du package **`java.util`**

- Une règle
  - *penser en terme d'interfaces plutôt que d'implémentations*
- Un style de programmation recommandé
  - *choisir une implémentation lorsqu'une collection est créée*
  - *immédiatement affecter la nouvelle collection à une variable typée avec l'interface correspondante (ou passer la collection à une méthode attendant un argument du type de l'interface). Prendre le type le plus général possible.*
  - *ainsi tout changement d'implémentation sera immédiat*

exemple

- Programme qui prend les mots de sa liste d'arguments, affiche les mots qui apparaissent plusieurs fois, le nombre de mots distincts et une liste des mots où les « duplicatats » ont été supprimés.

```
% java FindDups i came i saw i left

Duplicate detected: i
Duplicate detected: i
4 distinct words detected: [came, left, saw, i]
```

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new HashSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);

        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

exemple

- Maintenant on voudrait avoir les mots dans l'ordre alphabétique

```
% java FindDups i came i saw i left

Duplicate detected: i
Duplicate detected: i
4 distinct words detected:  [came, i, left, saw]
```

```
import java.util.*;

public class FindDups {
    public static void main(String args[]) {
        Set s = new TreeSet();
        for (int i=0; i<args.length; i++)
            if (!s.add(args[i]))
                System.out.println("Duplicate detected: "+args[i]);

        System.out.println(s.size()+" distinct words detected: "+s);
    }
}
```

String implémente Comparable

La classe `Collections`

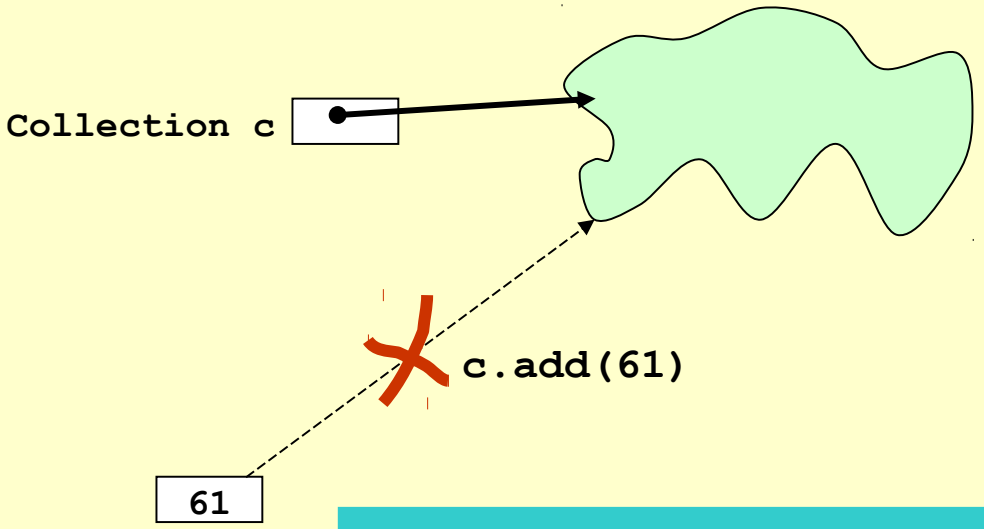
- Classe `Collections` définit des méthodes statiques qui opèrent sur les collections.
  - *Tri*
    - *public static void sort(List list)*
    - *public static void sort(List list,Comparator c)*
  - *Mélange*
    - *public static void shuffle(List list)*
  - *Recherche des valeurs extrêmes*
    - *public static Object max(Collection coll)*
    - *public static Object max(Collection coll, Comparator comp)*
    - *public static Object min(Collection coll)*
    - *public static Object min(Collection coll, Comparator comp)*
  - *Recherche*
    - *public static int binarySearch(List list,Object key)*
    - *public static int binarySearch(List list,Object key, Comparator comp)*

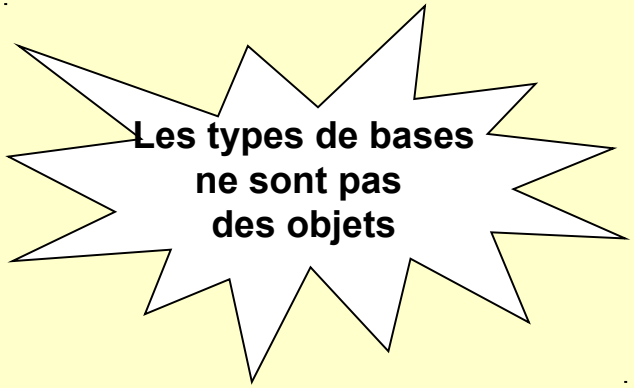
Les éléments de la liste doivent implémenter l'interface `Comparable`

La liste doit être triée dans l'ordre croissant

Les classes « enveloppes »

- Ajout d'un élément à une collection `public boolean add(Object o)`
- Une collection peut contenir n'importe quel type d'objet
- Quid des types de base ?
  `byte, short, int, long, double, float, char, boolean`

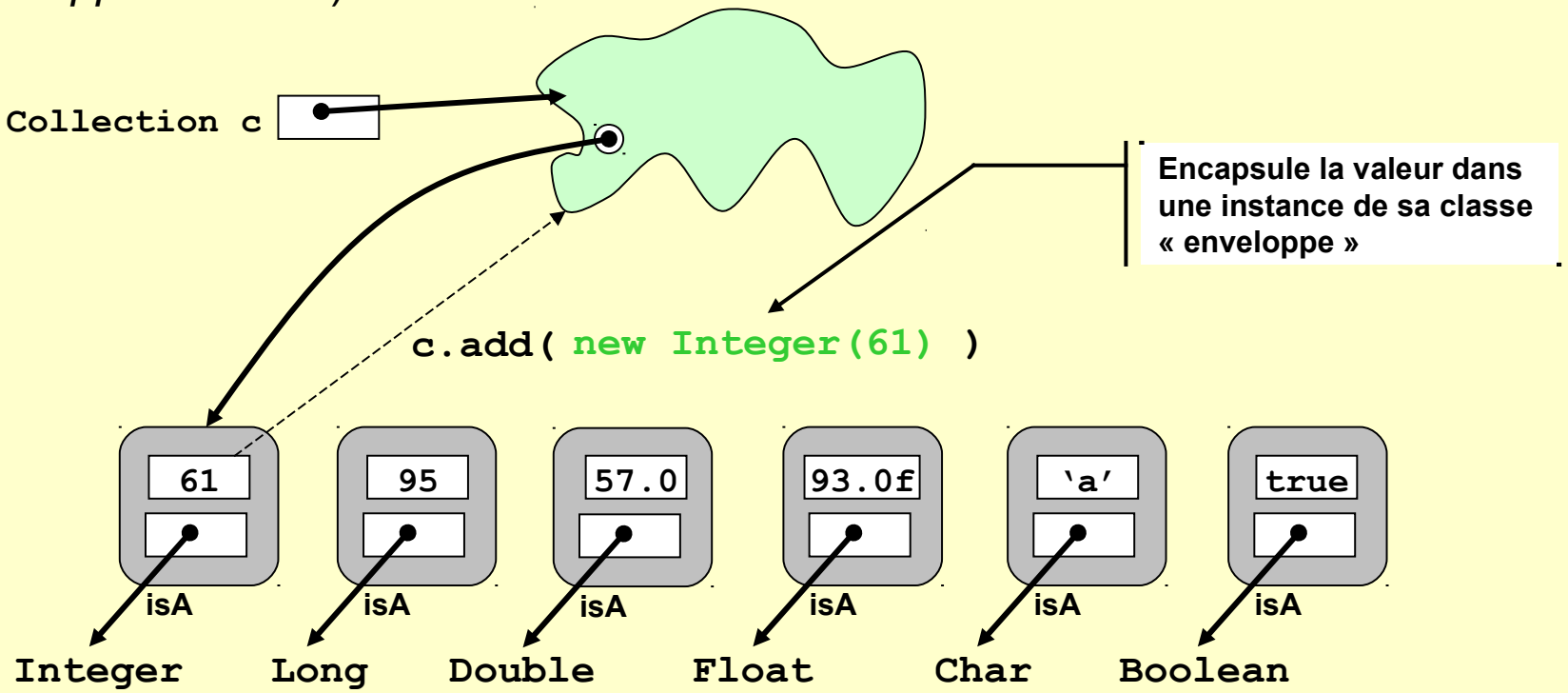





```
TestCollection.java [21:1] add(java.lang.Object) in
java.util.Collection cannot be applied to (int)
```

Les classes « enveloppes »

- Pour permettre l'utilisation de valeurs de types de base en tant qu'objet on « encapsule » la valeur dans un objet à l'aide d'une classe « enveloppe » (*« wrapper class »*)



Pour chacun des types de base il existe une classe enveloppe dont le nom correspond au nom du type de base mais avec une MAJUSCULE

Les classes « enveloppes »

- Principaux services des classe enveloppes

Classe enveloppe	*Création d'une instance (constructeur)*	*Accès à la valeur*	*Méthodes statiques pour conversions de chaînes vers les valeurs du type de base*

Integer	`Integer(int value)`
	`Integer intObj = new Integer(45);`
		`int integerValue()`
		`int i = intObj.integerValue();`
			`Integer valueOf(String s)`
			`int parseInt(String s)`
			`Integer intObj = Integer.valueOf("45");`

Double	`Double(double value)`	`double doubleValue()`	`Double valueOf(String s)` `double parseDouble(String s)`
Boolean	`Boolean(boolean value)`	`boolean booleanValue()`	`Boolean valueOf(String s)`
…			

Wrappers objects / valeurs de type simple

- Avantages et inconvénients des « wrapper » objects*

Avantages

- C'est un objet Java
- Peut être mis à null
- Peut être pointé plusieurs fois
- Peut être stocké dans une collection

Inconvénients

- Espace mémoire supérieur au type simple
- Consomme du temps processeur lors d'une instanciation mémoire par le constructeur
- Peu pratique pour le calcul
- Ne peut changer de valeur sans ré-allocation (pas de set)

* D'après Nicolas Sciara –Bientôt Java 1.5 - Programmez janvier 2004

Collections non typées

- L'utilisation de `Object` comme type pour l'ajout et la récupération d'objets dans les collections permet de créer collections de n'importe quoi….
- **MAIS…**
  - *oblige le programmeur à effectuer un transtypage lorsqu'il accède aux éléments d'une collection*

```
List  lesVisages = new ArrayList();
lesVisages.add(new VisageRond());
...
for (int i = 0; i < lesVisages.size(); i++)

   (VisageRond) (lesVisages.get(i)).dessiner(g);
```

Transtypage — Object

  - *pas de contrôle sur les valeurs rangés dans une collection*
    → risques d'erreurs d'exécution.

```
lesVisages.add(new Compte(…));
...
for (int i = 0; i < lesVisages.size(); i++)
      (VisageRond) (lesVisages.get(i)) .dessiner(g);
```

ClassCastException

Collections non typées

- L'utilisation de `Object` comme type pour l'ajout et la récupération d'objets dans les collections permet de créer collections de n'importe quoi….
- **MAIS…**
  - *oblige le programmeur à effectuer un transtypage lorsqu'il accède aux éléments d'une collection*

```
List  lesVisages = new ArrayList();
lesVisages.add(new VisageRond());
...
for (int i = 0; i < lesVisages.size(); i++)
    (VisageRond) (lesVisages.get(i)) .dessiner(g);
```

avec la version 1.5 de java (Tiger)
cela n'est qu'un mauvais souvenir
grâce à l'introduction (enfin !) de la généricité.

  - *pas de contrôle sur les valeurs rangés dans une collection*
    → risques d'erreurs d'exécution.

```
lesVisages.add(new Compte(…));
...
for (int i = 0; i < lesVisages.size(); i++)
    (VisageRond) (lesVisages.get(i)) .dessiner(g);
```

ClassCastException

Généricité

- Paramétrage d'un classe en fonction d'un type T (une autre classe)
  - *Le type formel T est utilisé comme type d'une ou plusieurs caractéristiques,*
  - *la classe manipule des informations de nature T. Elle ignore tout de ces informations.*

```
public interface List<T>  extends Collection {
    // Positional Access
    T     get(int index);
    T     set(int index,  T  element);  // Optional
    void add(int index,  T  element);   // Optional
    Object remove(int index);           // Optional
    boolean addAll(int index, Collection c);// Optional

    // Search
    int indexOf(  T  o);
    int lastIndexOf(  T  o);

    // Iteration
    ListIterator listIterator();
    ListIterator listIterator(int index);

    // Range-view
    List subList(int from, int to);
}
```

- Lors de l'instanciation le type formel T est remplacé par un type (Classe) existant

```
List<VisageRond> lesVisagesRonds
          = new ArrayList<VisageRond>()
```

```
List<String> lesChaines
          = new ArrayList<String>()
```

Depuis le JDK 7 simplification (diamant) :

List<String> lesChaines = new ArrayList<>();

Généricité

- Dans version 1.5 toutes les collections peuvent être utilisées de manière générique

```
List  lesVisages = new ArrayList();
lesVisages.add(new VisageRond());
...
for (int i = 0; i < lesVisages.size(); i++)
    (VisageRond) (lesVisages.get(i)).dessiner(g);
```

Transtypage obligatoire

```
List<VisageRond>  lesVisages = new ArrayList<VisageRond>();
lesVisages.add(new VisageRond());
...
for (int i = 0; i < lesVisages.size(); i++)
    lesVisages.get(i).dessiner(g);
```

Plus de transtypage

```
lesVisages.add(new Compte(…));
...
for (int i = 0; i < lesVisages.size(); i++)
   (VisageRond) (lesVisages.get(i)) .dessiner(g);
```

ClassCastException

Erreur à l'éxécution

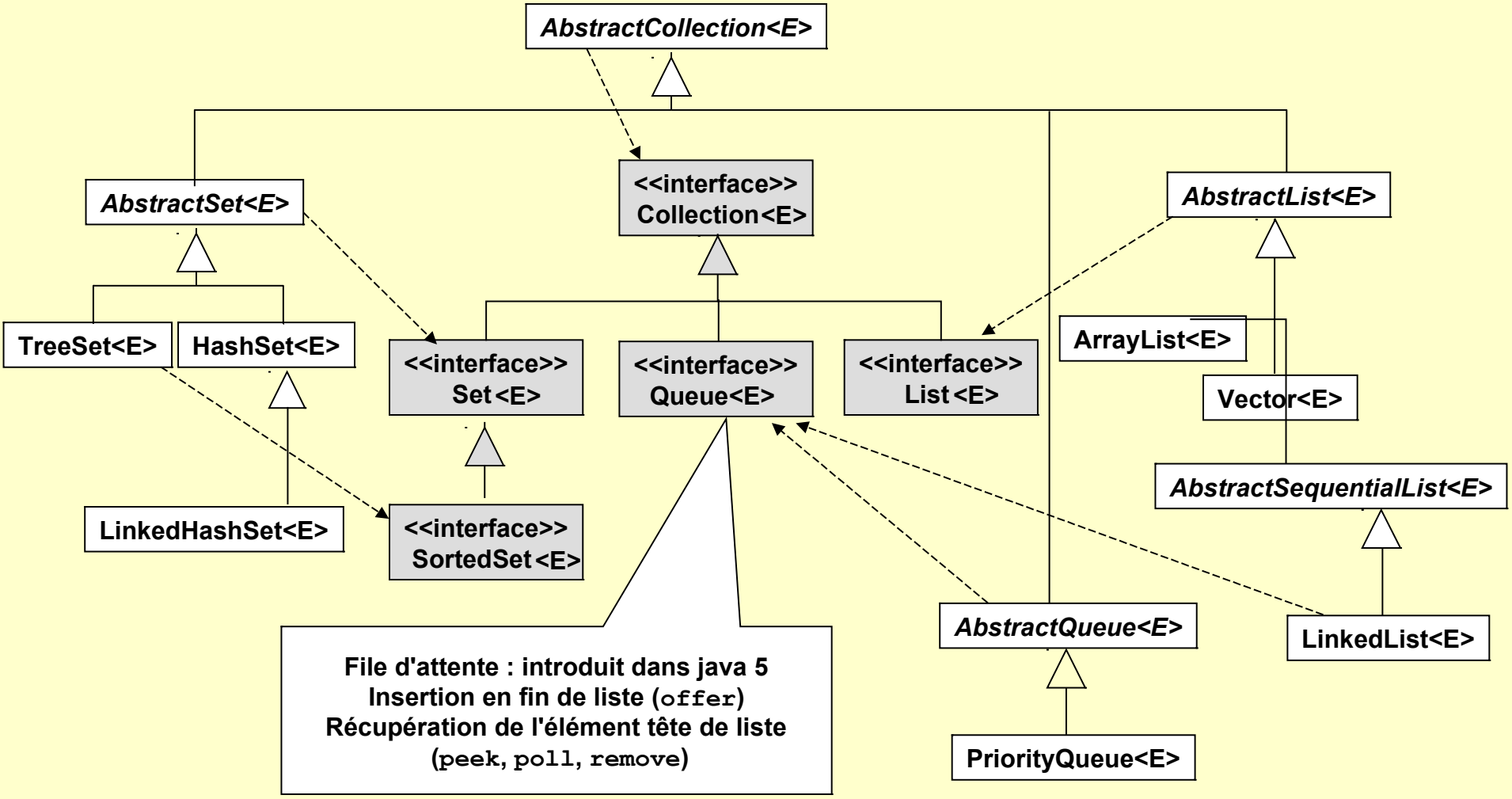
Erreur détectée à la compilation

```
lesVisages.add(new Compte(…));
...
for (int i = 0; i < lesVisages.size(); i++)
   (VisageRond) (lesVisages.get(i)) .dessiner(g);
```

Type incorrect

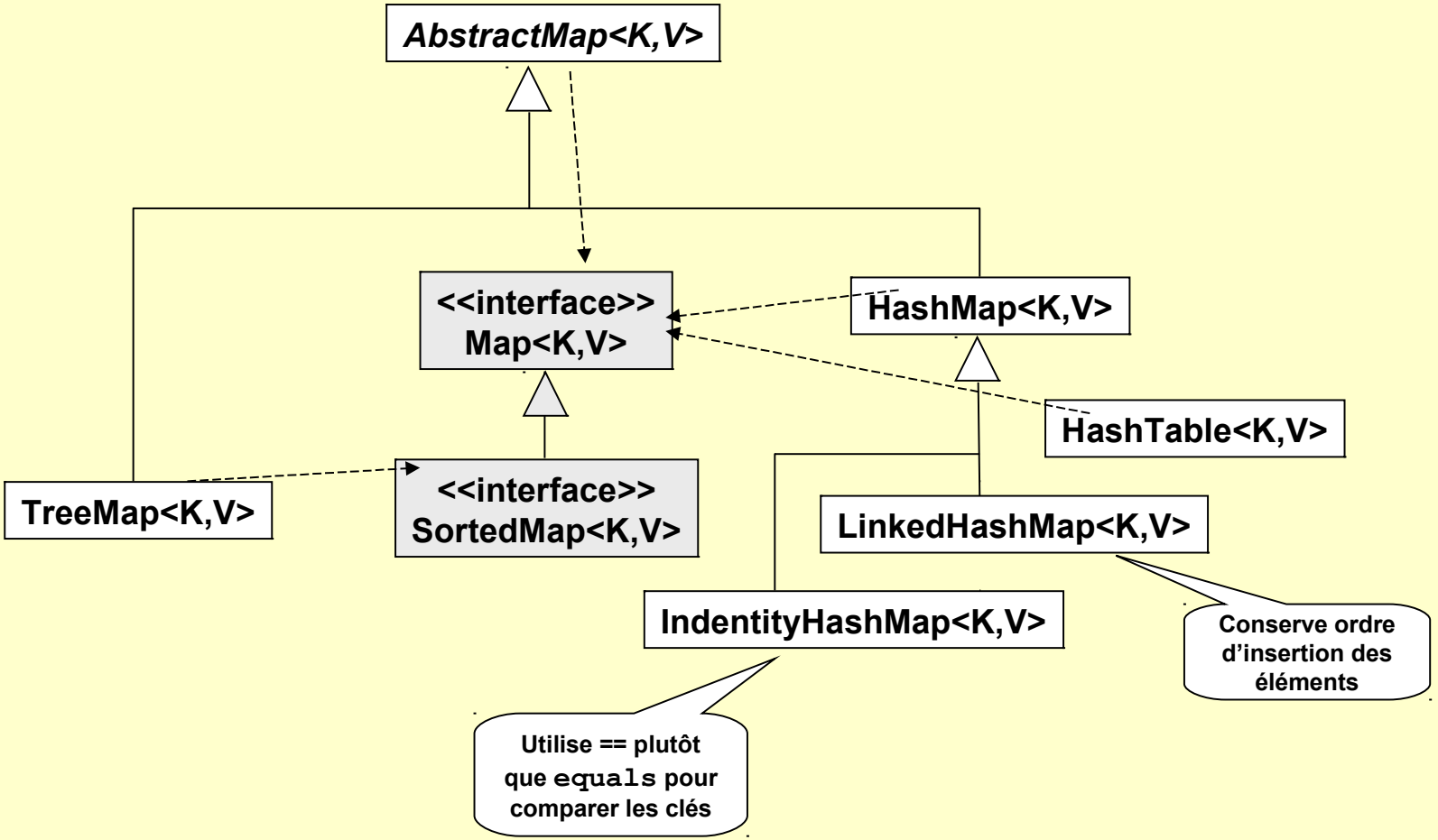
API de java util dans Java 5.0

- Toute l'API des collections s'appuie sur la généricité



API de java util dans Java 5.0

- Les Maps sont également génériques



API de java util dans Java 5.0

- Dans java 5.0 possibilité de continuer à utiliser des collections non typées
  - *Permettre au code antérieur de continuer à compiler et s'exécuter*

```
public class Test {

    public static void main(String[] args) {

        List l = new ArrayList();
        l.add("bonjour)");
        l.add(new Integer(10));
        Object o = l.get(0);
    }

}
```

```
javac Test.java
Note: Test.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

```
javac -Xlint Test.java
Test.java:32: warning: [unchecked] unchecked call to add(E) as a member of
the raw type java.util.List
        l.add("bonjour)");
Test.java:33: warning: [unchecked] unchecked call to add(E) as a member of
the raw type java.util.List
        l.add(new Integer(10));
2 warnings
```

Autoboxing

- Un autre ajout à java dans la version 1.5 : l'autoboxing
- La possibilité de passer automatiquement d'un type simple (int, float, double, char, …) vers un type objet correspondant (Integer, Float, Double, Char, …) et inversement.
  - *Une technique déjà employée dans C# (bien fait, ils n'avaient qu'à pas copier !)*

```
List ld = new ArrayList();

ld.add(new Double(3.14));

double d1 = ((Double) ld.get(i)).doubleValue();


Double d = new Double(3.14);


double d2 = d.doubleValue();
```

```
List<Double> ld = new ArrayList<Double>();

ld.add(3.14);  // autoboxing

double d1 = (double) ld.get(i);
                // unboxing

Double d = (Double) 3.14;
                // boxing par transtypage

double d2 = (double) new Double(3.14);
                // unboxing par un cast
```

ForEach

- Autre nouveauté dans 1.5 l'instruction « *foreach* »

```
List lv = new ArrayList();

lv.add(new VisageRond(...));
...

for (Iterator it = lv.iterator(); it.hasNext(); ) {
   VisageRond v = (VisageRond) it.next();
   v.deplacer();
   ...
}
```

Idiome de base pour itérer sur des collections : un peu verbeux

```
List<VisageRond> lv = new ArrayList<VisageRond>();

lv.add(new VisageRond(...));
...

for (Iterator<VisageRond> it = lv.iterator(); it.hasNext(); ) {
   VisageRond v = it.next();
   v.deplacer();
   ...
}
```

La situation est un peu améliorée avec la généricité

```
List<VisageRond> lv = new ArrayList<VisageRond>();

lv.add(new VisageRond(...));
...

for (VisageRond v : lv ) {
   v.deplacer();
...
}
```

Un second type de boucles for spécialement conçu pour itérer sur les collections

ForEach

- « *foreach* » peut être aussi utilisé pour itérer sur des tableaux

```
/**
 * calcule somme des éléments d'un tableau d'entiers
 * @param a le tableau d'entier
 * @return la somme des éléments de a
 */
 public int sum(int[] a) {
    int sum = 0;
    for (int i = 0; i < a.length; i++)
       sum += a[i];
    return sum;
}
```

Boucle for traditionnelle

Avec le nouveau type de boucle for

```
public int sum(int[] a) {
    int sum = 0;
    for (int x : a)
       sum += x;
    return sum;
}
```

Autres nouveautés de java 1.5

- **Metadata** This language feature lets you avoid writing boilerplate code under many circumstances by enabling tools to generate it from annotations in the source code. This leads to a "declarative" programming style where the programmer says what should be done and tools emit the code to do it. Also it eliminates the need for maintaining "side files" that must be kept up to date with changes in source files. Instead the information can be maintained *in* the source file. Refer to JSR 175 http://jcp.org/en/jsr/detail?id=175.
- **Static Import** This facility lets you avoid qualifying static members with class names without the shortcomings of the "Constant Interface antipattern." Refer to JSR 201 http://jcp.org/en/jsr/detail?id=201.
- **Typesafe Enums** This flexible object-oriented enumerated type facility allows you to create enumerated types with arbitrary methods and fields. It provides all the benefits of the Typesafe Enum pattern ("Effective Java," Item 21) without the verbosity and the error-proness. Refer to JSR 201 http://jcp.org/en/jsr/detail?id=201.
- **varargs** This facility eliminates the need for manually boxing up argument lists into an array when invoking methods that accept variable-length argument lists. Refer to JSR 201 http://jcp.org/en/jsr/detail?id=201.

Types énumérés

- Avant Java 5.0, définition de type énumérés à l'aide de constantes entières

```
public class Jour {
   public static final int LUNDI = 1;
   public static final int MARDI = 2;
   ...
   public static final int DIMANCHE = 7;
}
```

- **Non « type-safe »** : un Jour étant simplement un `int` il est possible de passer n'importe quelle valeur là où un jour est attendu, ou d'additionner deux jours ensembles
- **Fragilité** : les valeurs énumérées sont des constantes fixées dans les clients qui les utilisent au moment de la compilation. Si une nouvelle constante est ajoutée entre deux constantes ou que l'ordre est changé les clients doivent être recompilés, sinon ils continueront à fonctionner mais leur comportement pourra être imprévisible
- **Non informativité** : lorsque vous imprimez une valeur énumérée vous n'obtenez qu'un entier ce qui ne dit pas ce que cette valeur représente ni quel est son type.
- **Non objet** : pas d'opérations (méthodes) définies sur les valeurs d'un type énuméré

Types énumérés

- Avant java 5.0 (java 1.5.0) nécessité de passer par des patterns objets pour répondre à ces problèmes

```
public class Jour {
   public static final Jour LUNDI = new Jour("Lundi",1);
   public static final Jour MARDI = new Jour("Lundi",2);
   ...
   public static final Jour DIMANCHE = new Jour("Dimanche",7);

   private static Jour[] values = { LUNDI, MARDI, ... , DIMANCHE };

   private String nom;
   private int rang;

   private Jour(String nom, int rang) {
      this.nom = nom;
      this.rang = rang;
   }

   public String toString() {
      return nom;
   }

   public int getRang() {
       return rang;
   }

   public Jour suivant() {
       return values[(this.rang + 1) % 7];
   }
}
```

- *Item 21 dans « Effective Java » de Joshua Bloch*
- *Mais ce n'est pas la panacée*
  - *à la charge du programmeur*
  - *Très verbeux (-> augmente risque d'erreurs)*
  - *Les valeurs de types énumérées ainsi définis ne peuvent être utilisées dans des instructions* `switch`

Types énumérés

- Java 5.0 fournit un type énuméré « type-safe » en standard via le mot clé **enum**

```
public enum Jour {
   LUNDI, MARDI, ..., DIMANCHE ;
}
```

Dans sa forme le plus simple un type énuméré définit un ensemble de valeurs

```
public enum Mois {
   JANVIER, FEVRIER, ..., DECEMBRE ;
}
```

```
public class RendezVous {
   private Jour j;
   private Mois m;
   ...

   public RendezVous(Jour j, Mois m, ...) {
      this.j = j;
      this.m = m;
      ...
   }
   ...
   String toString() {
      return "Rendez-vous : " + j + ... ;
}
```

```
...
RendezVous rv = new RendezVous(
      Jour.LUNDI,Mois.DECEMBRE, ...);
...
System.out.println(rv);
...
```

```
...
Rendez-vous : LUNDI ...
...
```

Les types énumérés étant définis comme des classes il est possible de définir des types beaucoup plus riches

Types énumérés

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    private double mass()   { return mass; }
    private double radius() { return radius; }

    // universal gravitational constant  (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Chaque constante de l'énumération est déclarée avec des paramètres qui seront passés au constructeur

Possibilité **d'ajouter des données** et **un comportement** aux objets d'une énumération

Exemple :
j2sdk1.5.0\docs\guide\language\enums.html

Les planètes du système solaire.

Chaque planète a une masse et un rayon

L'énumération possède un constructeur

Chaque planète peut calculer la gravité à sa surface et le poids d'un objet à sa surface

Types énumérés

- Exemple d'utilisation de l'énumération `Planet`

Un programme qui prend votre poids sur terre (dans n'importe quelle unité) , calcule et affiche votre poids (dans la même unité) sur les différentes planètes du système solaire

```
public static void main(String[] args) {
        double earthWeight = Double.parseDouble(args[0]);
        double mass = earthWeight/EARTH.surfaceGravity();
        for (Planet p : Planet.values())
           System.out.printf("Your weight on %s is %f%n",
                             p, p.surfaceWeight(mass));
    }
```

Boucle « foreach » (java 5.0), simplification d'écriture pour itérer facilement sur les éléments d'une collection

Méthode statique qui pour une énumération retourne un tableau contenant toutes les valeurs énumérées dans l'ordre où elles ont été déclarées

```
Planet[] lesPlanetes = Planet.values();
for (int i = 0; i < lesPlanetes.length; i++) {
    Planet p = lesPlanetes[i];
    System.out.println(…, p,…);
}
```

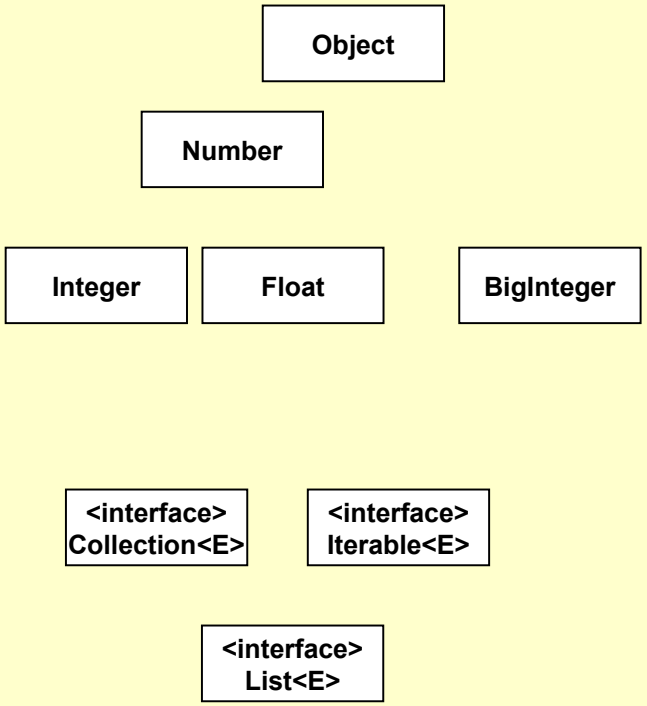
Bien d'autres possibilités avec les types énumérés. Par exemple définir des comportements différents pour les éléments de l'énumération… Voir :

`j2sdk1.5.0\docs\guide\language\enums.html`

```
$ java Planet 175
Your weight on MERCURY is 66.107583
Your weight on VENUS is 158.374842
Your weight on EARTH is 175.000000
Your weight on MARS is 66.279007
Your weight on JUPITER is 442.847567
Your weight on SATURN is 186.552719
Your weight on URANUS is 158.397260
Your weight on NEPTUNE is 199.207413
Your weight on PLUTO is 11.703031
```

Hiérarchie de types paramétrés

- Le types paramétrés forment une hiérarchie de types tout comme les types "normaux"
  - *Cette hiérarchie est basé sur le type de base et non pas sur le type des paramètres*




```
ArrayList<Integer> arrayListInt = new ArrayList<Integer>();
```

Un ArrayList d'Integer est une liste d'Integer

Un ArrayList d'Integer est une collection d'Integer

```
List<Integer> listInt = arrayListInt;
Collection<Integer> colInt = arrayListInt;
```

Les Integer sont des Number. Est-ce qu'un ArrayList d'Integer est une liste de Number ?

```
List<Number> arrayListNumber = arrayListInt;
```

Un ArrayList d'Integer n'est pas une liste de Number !

```
arrayListInt.add(new Integer(10));
arrayListInt.add(new Integer(14));
                    arrayListNumber.add(new Double(3.14159));

Integer anInt1 = arrayListInt.get(0);
Integer anInt2 = arrayListInt.get(1);
  Integer anInt2 = arrayListInt.get(3);
```