IUT Lyon 1

l'excellence technologique

Collections

LP IEM

Collections

- Une collection est un objet qui regroupe plusieurs éléments.
- Les collections sont utilisées pour stocker, extraire, manipuler et communiquer des données agrégées.
- Typiquement elles représentent des données qui vont naturellement ensemble
  - une main au poker (une collection de cartes),
  - un dossier de courrier (une collection de lettres),
  - un répertoire téléphonique (des associations nom/numéro de téléphone).

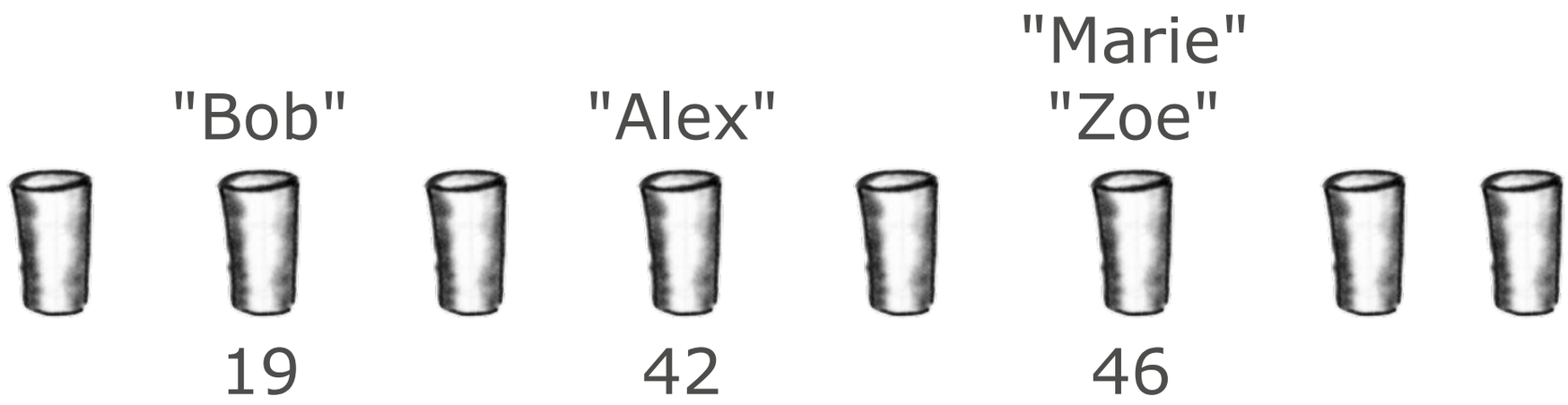
- Collections
  - Les collections remplacent avantageusement les tableaux lorsque le nombre d'éléments n'est pas connu à l'avance
  - Les ensembles gérés par les classes implémentant l'interface **List** peuvent contenir des éléments égaux. Chaque élément a une position déterminée (un peu comme un tableau mais non limité en taille)
  - Les ensembles gérés par des classes implémentant l'interface **Set** ne peuvent pas contenir d'éléments égaux.
  - Les ensembles gérés par les classes implémentant l'interface **Map** utilisent une clé pour accéder aux éléments stockés au lieu d'un indice entier.
    - La clé peut être n'importe quel objet (par exemple une chaîne correspondant au nom d'un contact, la valeur stockée étant un numéro de téléphone)

Collections - hashcode() et equals()

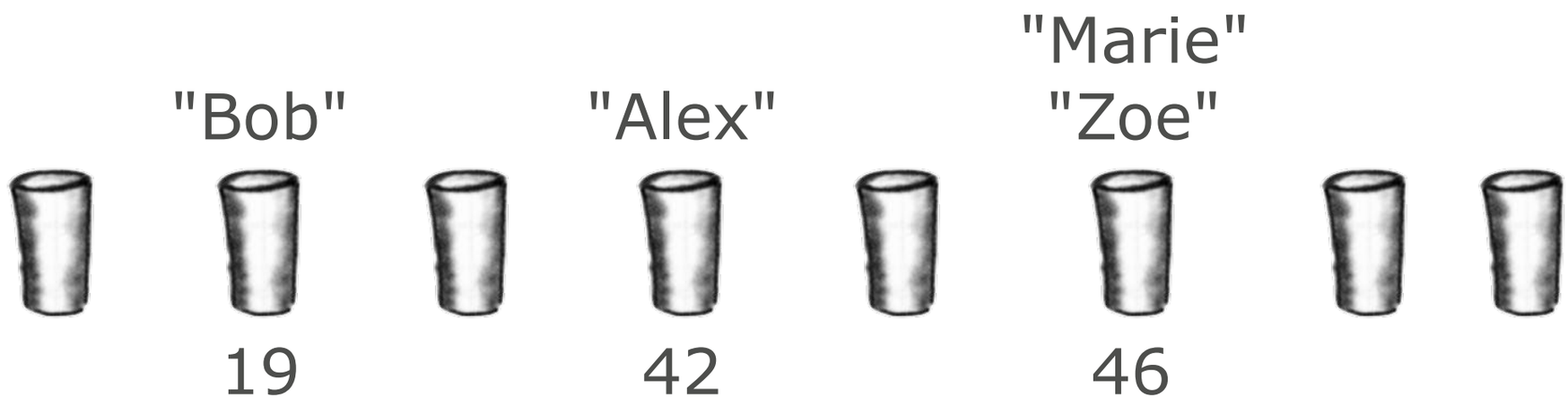
- Pour faire bon ménage avec les classes de collections les objets doivent souvent redéfinir 2 méthodes de la classe Object : hashcode() et equals()
- Pour bien comprendre le contrat que doivent remplir ces méthodes, il faut regarder comment certaines classes de collections utilisent les codes de hachage
  - Imaginez des récipients alignés sur le sol (*bucket*)
  - Quelqu'un vous donne un papier avec un nom dessus
  - A partir du nom, vous calculez un code : nombre entier obtenu en faisant la somme des valeurs numériques des lettres (A : 1, B : 2, …).
    ➙ A un nom donné correspondra toujours la même valeur

Collections - hashcode() et equals()

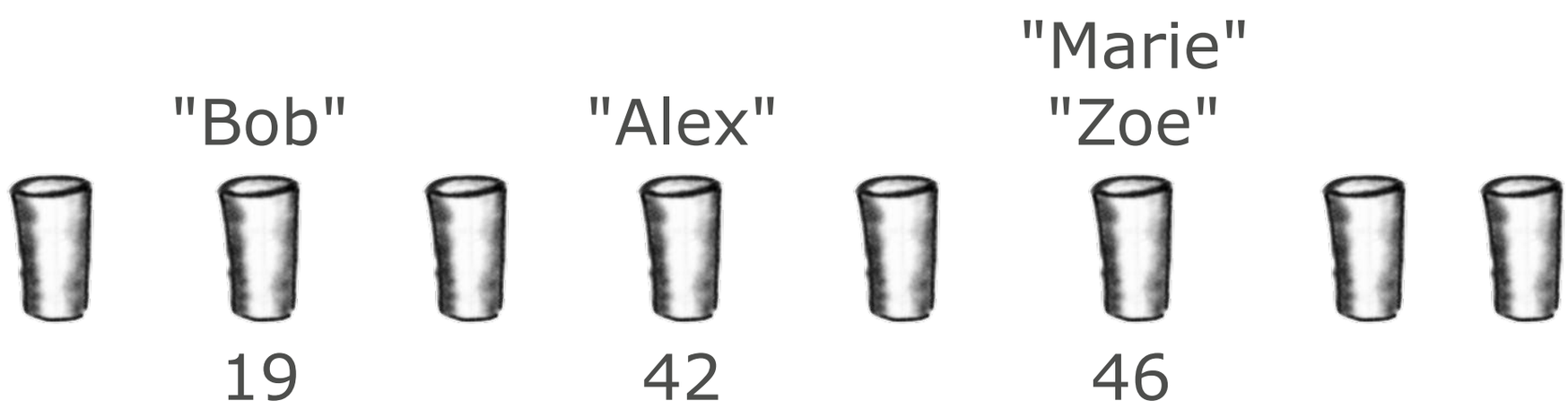
Clé	Algorithme de hachage	Code de hachage
Alex	A(1)+L(12)+E(5)+X(24)	42
Marie	M(13)+A(1)+R(18)+I(9)+E(5)	46
Bob	B(2)+O(15)+B(2)	19
Zoe	Z(26)+O(15)+E(5)	46



- Collections - hashcode() et equals()
  - Le code de hachage permet de savoir dans quel récipient on place le papier
  - Si quelqu'un vous demande de retrouver le papier correspondant au nom "Zoe"
    - vous calculez le code de hachage pour Zoe, ce qui vous donne le numéro du récipient dans lequel il se trouve
    - par contre, deux objets différents peuvent donner le même code de hachage (ici Marie et Zoe) : cela implique qu'il faut ensuite rechercher dans le récipient en question et tester l'égalité entre le nom cherché et les noms inscrits sur les papiers contenus dans ce récipient



- Collections - hashcode() et equals()
  - En résumé : recherche dans une table de hachage
    - Recherche du bon récipient en utilisant hashcode()
    - Recherche du bon objet dans le récipient en utilisant equals()
  - Pour améliorer l'efficacité, il faut disposer d'une bonne fonction de hachage qui produit des valeurs bien dispersées pour des objets différents



- Collections - hashcode() et equals()
  - Contrat général de la méthode **equals()**
    - Réflexivité : si x != null alors x.equals(x) doit retourner true
    - Symétrie : si x et y non null alors x.equals(y) doit renvoyer true si et seulement si y.equals(x) retourne true
    - Transitivité : si x.equals(y) et y.equals(z) retournent true alors x.equals(z) doit retourner true
    - Cohérence : plusieurs invocation de x.equals(y) doivent retourner le même résultat si les objets n'ont pas été modifiés
    - si x est non null, x.equals(null) doit retourner false
  - Définition un peu mathématique mais son non respect peut conduire à des problèmes de fonctionnement importants si les objets sont utilisés avec des classes comme les classes de collections

Collections - hashcode() et equals()

Contrat général de la méthode equals(), exemple de non respect du contrat de symétrie (1/2)

```
// Broken - violates symmetry!
public final class CaseInsensitiveString {
     private final String s;

     public CaseInsensitiveString(String s) {
          if (s == null)
               throw new NullPointerException();
          this.s = s;
     }

     // Broken - violates symmetry!
     @Override
     public boolean equals(Object o) {
          if (o instanceof CaseInsensitiveString)
               return s.equalsIgnoreCase(((CaseInsensitiveString) o).s);
          if (o instanceof String) // One-way interoperability!
               return s.equalsIgnoreCase((String) o);
          return false;
     }
}
```

Collections - hashcode() et equals()

Contrat général de la méthode equals(), exemple de non respect du contrat de symétrie (2/2)

```
CaseInsensitiveString cis = new CaseInsensitiveString("Toto");
String s = "toto";
```

- `cis.equals(s)` retourne true comme attendu mais `s.equals(cis)` retourne false
- Solution :
  - Ne pas essayer d'interopérer avec la classe String

```
@Override
    public boolean equals(Object obj) {
        return (obj instanceof CaseInsensitiveString) &&
                ((CaseInsensitiveString) obj).s.equalsIgnoreCase(this.s);
    }
```

Collections - hashcode() et equals()

Contrat général de la méthode equals(), exemple de non respect du contrat de transitivité (1/5)

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

    public Point(int x, int y) {
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Point) {
            Point p = (Point) obj;
            return p.x == x && p.y == y;
        }
        return false;
    }
}
```

Collections - hashcode() et equals()

Contrat général de la méthode equals(), exemple de non respect du contrat de transitivité (2/5)

```
public class ColorPoint extends Point {

     private int color;

     public ColorPoint(int x, int y, int color) {
          super(x, y);
          this.color = color;
     }

     @Override
     public boolean equals(Object obj) {
          if (!(obj instanceof ColorPoint))
               return false;

          return super.equals(obj) && ((ColorPoint) obj).color == color;
     }

}
```

Collections - hashcode() et equals()

- Contrat général de la méthode equals(), exemple de non respect du contrat de transitivité (3/5)

```
Point p = new Point(1,2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

- `p.equals(cp)` retourne true alors que `cp.equals(p)` retourne false -> non respect de la symétrie

- On serait tenté d'ignorer la couleur lorsqu'on fait une comparaison avec un objet point et de proposer une méthode equals comme celle qui suit

```
@Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Point))
            return false;

        if (!(obj instanceof ColorPoint))
            //obj is a point, do a color-blind comparison
            return obj.equals(this);

        //obj is a ColorPoint : do a full comparison
        return super.equals(obj) && ((ColorPoint) obj).color == color;
    }
```

Collections - hashcode() et equals()

Contrat général de la méthode equals(), exemple de non respect du contrat de transitivité (4/5)

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1,2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

- `p1.equals(p2)` et `p2.equals(p3)` retournent true mais `p1.equals(p3)` retourne false -> non respect de la transitivité
- Solution
  - Utiliser la composition plutôt que l'héritage et fournir une méthode publique qui retourne un point à la même position

Collections - hashcode() et equals()

- Contrat général de la méthode equals(), exemple de non respect du contrat de transitivité (5/5)
- Solution

```
public class ColorPoint{

     private final Color color;
     private final Point point;

     public ColorPoint(int x, int y, Color color) {
          if (color==null)
               throw new NullPointerException();
          point = new Point(x, y);
          this.color = color;
     }

     Point asPoint(){
          return point;
     }

     @Override
     public boolean equals(Object obj) {
          if (!(obj instanceof ColorPoint))
               return false;
          ColorPoint cp = (ColorPoint)obj;
          return cp.point.equals(point) && cp.color.equals(color);
     }
}
```

- Collections - hashcode() et equals()
  - Contrat général de la méthode hashcode()
    - Doit retourner une valeur identique tant qu'aucune des informations testées par la méthode equals() ne change
    - Deux objets égaux (au sens de la méthode equals()) doivent retourner des hashcodes identiques
    - Il n'est pas nécessaire que deux objets non égaux (au sens d'equals()) retournent des codes de hachage différents. C'est par contre souhaitable pour améliorer les performances des collections utilisant des tables de hachage
  - Pour respecter ce contrat, à chaque fois qu'on redéfinit equals() il faut redéfinir hashcode()
    - Pour une bonne implémentation, s'appuyer sur les méthodes générées automatiquement par les IDE ou se référer au livre "Effective Java" de Joshua Bloch (Item 9 - 2nd Edition)

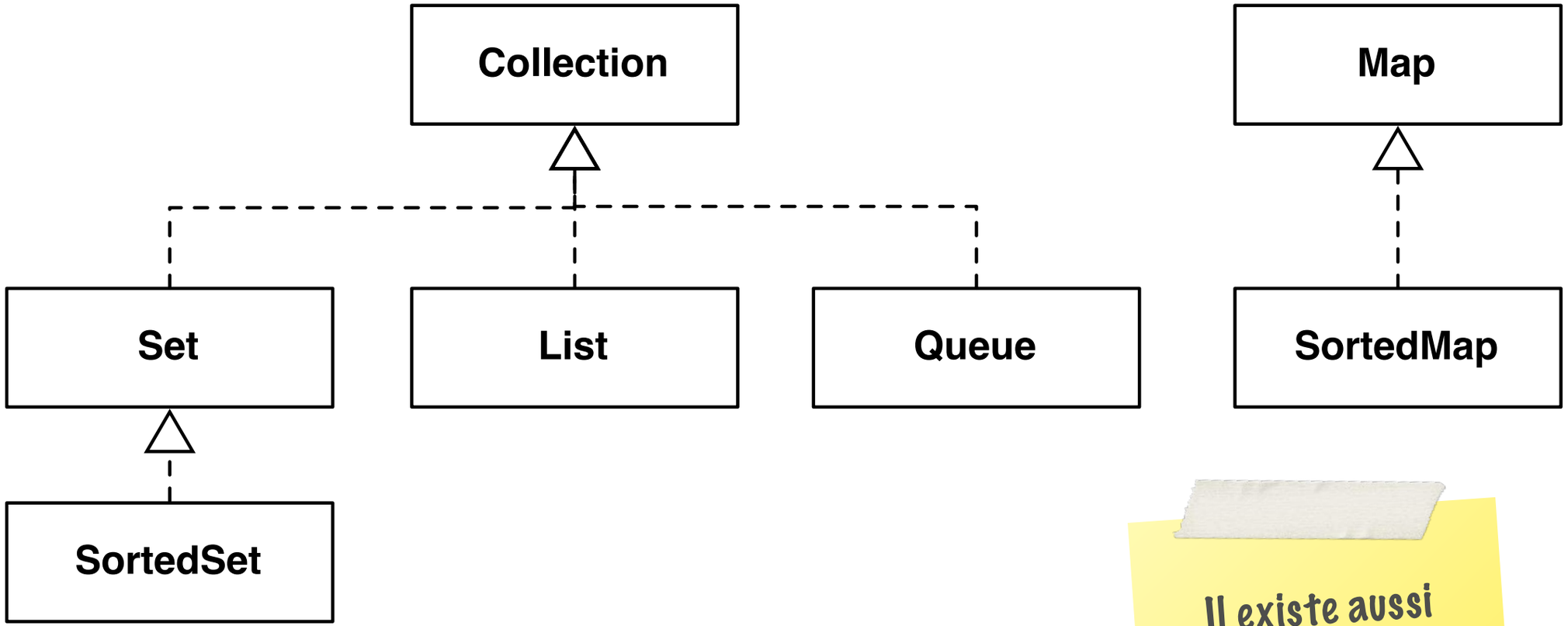
- Exemple
  - equals()
  - hashcode()
    - Implémentation par défaut IntelliJ Idea (avant Java 7)

```java
public class Player {
    private final String name;
    private final int score;
    private final int health;

    public Player(String name, int score, int health) {
        this.name = name;
        this.score = score;
        this.health = health;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

        Player player = (Player) o;

        if (score != player.score) return false;
        if (health != player.health) return false;
        return name.equals(player.name);
    }

    @Override
    public int hashCode() {
        int result = name.hashCode();
        result = 31 * result + score;
        result = 31 * result + health;
        return result;
    }
}
```

- Exemple
  - equals()
  - hashcode()
    - Implémentation (Java 7+)

```java
import java.util.Objects;

public class Player {
    private final String name;
    private final int score;
    private final int health;

    public Player(String name, int score, int health) {
        this.name = name;
        this.score = score;
        this.health = health;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Player player = (Player) o;
        return score == player.score &&
                health == player.health &&
                Objects.equals(name, player.name);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, score, health);
    }
```

- Collections
  - Le framework Collections définit plusieurs interfaces



Il existe aussi l'interface Deque

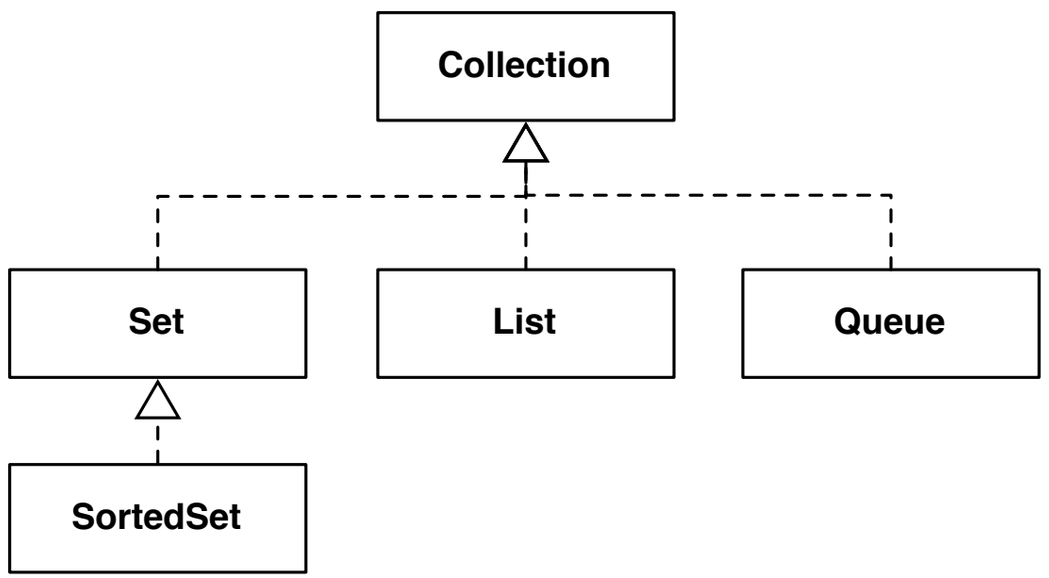
Interface Collection - exemples de méthodes

<<interface>> Collection	
int size()	
boolean isEmpty()	
boolean contains(Object element)	
boolean add(E element)	//optionelle
boolean remove(Object element)	//optionelle
boolean containsAll(Collection<?> c)	
boolean addAll(Collection<? extends E> c)	//optionelle
void clear()	//optionelle

Les méthodes "optionnelles" doivent être implémentées mais elle peuvent ne pas remplir le contrat général.

Ex. : une collection immuable ne supporte pas les opérations pouvant modifier le contenu

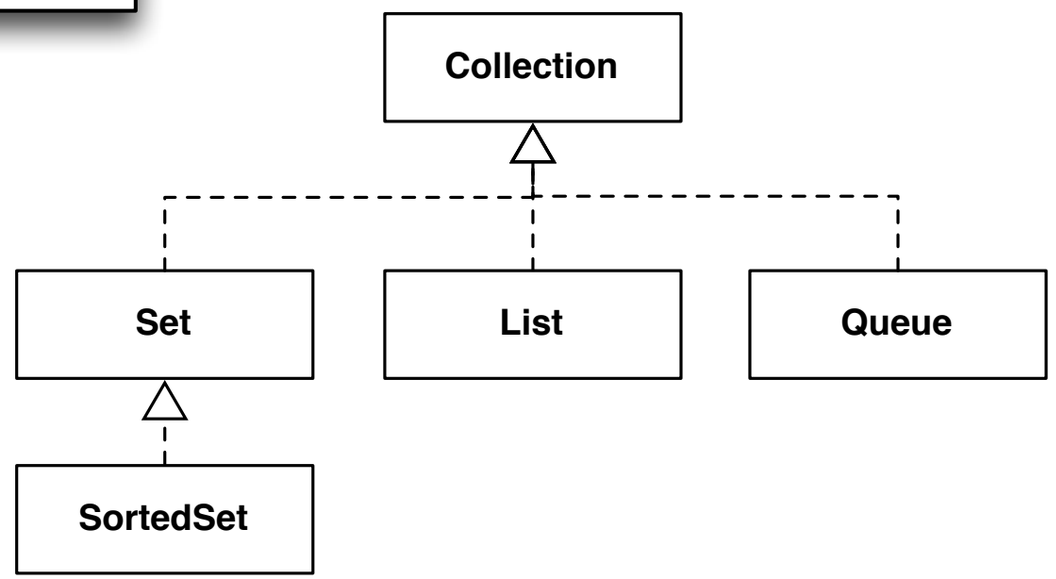
L'implémentation peut choisir de renvoyer une exception UnsupportedOperationException ou de ne rien faire.



Interface List - exemples de méthodes supplémentaires

<<interface>> **List**

E get(int index);
E set(int index, E element);
boolean add(E element);
void add(int index, E element);
E remove(int index);
boolean addAll(int index, Collection<? extends E> c);
int indexOf(Object o);
int lastIndexOf(Object o);
List<E> subList(int from, int to);



Implémentations (exemples)

Maps	Sets	Lists	Queues	Utilities
HashMap	HashSet	ArrayList	PriorityQueue	Collections
Hashtable	LinkedHashSet	Vector	LinkedBlocking Queue	Arrays
TreeMap	TreeSet	LinkedList	DelayQueue	
LinkedHashMap				

- Implémentations
  - Implémentations de l'interface List
    - ArrayList
      - "tableau qui peut s'agrandir" avec éléments accessibles par un index
      - itération et accès aléatoire (à un indice donné) rapides
      - à préférer à LinkedList s'il s'agit d'une collection dont on doit parcourir le contenu rapidement et qui ne fait pas l'objet de beaucoup d'insertions/ suppressions
    - LinkedList
      - Liste doublement chaînée dont les éléments sont accessibles par un index
      - Insertion/suppression d’un élément rapide
    - Vector
      - Reste des premières versions de Java (à ne pas utiliser)

- Implémentations
  - Les collections peuvent être ordonnées ou non
    - lors d'une itération sur la collection, l'ordre dans lequel les objets sont parcourus peut être ordonné (ordre d'insertion dans la collection par ex) ou non : l'ordre des éléments dans une HashTable dépendra des valeurs retournées par hashcode() et apparaitra comme aléatoire et dépendant du contenu de la collection
  - Les collections peuvent être triées ou non (Sorted)
    - Certaines collections s'appuient sur les interfaces Comparable ou Comparator pour trier les données et assurer un ordre d'énumération répondant à un critère (prix croissant, ordre alphabétique, dates croissantes,...)
    - Une collection triée est nécessairement ordonnée

- Implémentations
  - Implémentations de l'interface Set (absence de doublons)
    - HashSet
      - Les éléments sont non ordonnés, non triés
      - Temps d'accès à un objet et test de la présence d'un objet rapides
    - LinkedHashSet
      - Idem mais avec des éléments qui sont ordonnées par ordre d'insertion grâce à une liste doublement chaînée
    - TreeSet
      - Les éléments sont triés
      - Temps d'accès et d'insertion plus longs

- Implémentations
  - Implémentations de l'interface Map
    - HashMap
      - La plus rapide
      - Non ordonnée / non triée
    - Hashtable
      - comme Vector : encore là pour des raisons historiques (à oublier !)
    - LinkedHashMap
      - Presque aussi rapide que HashMap mais avec un ordre d'itération déterminé (ordre d'insertion ou de dernier accès -> utile pour des caches)
    - TreeMap
      - Les éléments sont triés mais au prix de performances moins bonnes

Implémentations

Classe	Ordonnée	Triée
HashMap	Non	Non
Hashtable	Non	Non
TreeMap	Triée	Par ordre naturel ou règle de comparaison
LinkedHashMap	Par ordre d'insertion ou de dernier accès	Non
HashSet	Non	Non
TreeSet	Triée	Par ordre naturel ou règle de comparaison
LinkedHashSet	Par ordre d'insertion	Non
ArrayList	Par index	Non
Vector	Par index	Non
LinkedList	Par index	Non

Implémentations - performances

Structure	get	add	remove	contains
ArrayList	O(1)	O(1)*	O(n)	O(n)
LinkedList	O(n)	O(1)	O(1)	O(n)
HashSet	O(1)	O(1)	O(1)	O(1)
LinkedHashSet	O(1)	O(1)	O(1)	O(1)
TreeSet	O(log n)	O(log n)	O(log n)	O(log n)
HashMap	O(1)**	O(1)	O(1)	O(1)
LinkedHashMap	O(1)	O(1)	O(1)	O(1)
TreeMap	O(log n)	O(log n)	O(log n)	O(log n)

* Temps amorti : ajouter n éléments en O(n) (à cause de l'allocation du tableau sous-jacent)
** Cas idéal (si la fonction de hachage est mauvaise il peut y avoir une longue liste dans le même récipient)

- Types primitifs et collections
  - Les collections s'appuient sur les méthodes equals() et hashcode() de la classe Object et ne peuvent donc stocker que des objets
  - L'ajout de données de type primitifs (int, float, boolean,...) ne peut se faire qu'en utilisant les classes d'encapsulation (wrapper)
    - Ex: Integer pour int, Boolean pour boolean, ...

```
List<Integer> myList = new ArrayList<Integer>();
myList.add(new Integer(42));
myList.add(42);  //En s'appuyant sur l'autoboxing (Java 1.5 et supérieures)
```

- Exemples
  - Toujours déclarer les variables avec le type le plus abstrait possible (généralement une interface)
    - L'implémentation concrète est choisie lors de l'instanciation

```
List<Cocktail> cocktails = new ArrayList<Cocktail>();
cocktails.add(new Cocktail("Mojito"));
cocktails.add(new Cocktail("Diabolo fraise"));
System.out.println(cocktails.get(1));
```

```
Map<String,String> annuaire = new HashMap<String, String>();
annuaire.put("Ledoux Germaine", "06 11 22 33 44");
annuaire.put("Ledoux Robert", "06 55 66 77 88");
System.out.println(annuaire.get("Ledoux Germaine"));
```

- Itérer sur les éléments d'une collection
  - Boucle for each

```
for(Card card : collection){
    System.out.println(card);
}
```

  - Utilisation d'un itérateur

```
for (Iterator<Card> it = collection.iterator(); it.hasNext(); )
{
    Card card = it.next();
    System.out.println(card.toString());
}
```

Utilité des itérateurs

- Permettent d'enlever des éléments de la collection pendant l'énumération
  - La méthode remove est optionnelle (certaines collections ne supportent pas cette opération)

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

- Permettent de parcourir simultanément plusieurs collections

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

Collections - Qu'affiche ce programme ?

```
public class Name {
    private final String first, last;

    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name) o;
        return n.first.equals(first) && n.last.equals(last);
    }

    public static void main(String[] args) {
        Set<Name> s = new HashSet<Name>();
        s.add(new Name("Mickey", "Mouse"));
        System.out.println(s.contains(new Name("Mickey", "Mouse")));
    }
}
```

↳ Corriger le programme

Collections - Qu'affiche ce programme ?

```
public class Name {
    private final String first, last;

    public Name(String first, String last) {
        this.first = first;
        this.last = last;
    }

    public boolean equals(Name n) {
        return n.first.equals(first) && n.last.equals(last);
    }

    public int hashCode() {
        return 31 * first.hashCode() + last.hashCode();
    }

    public static void main(String[] args) {
        Set<Name> s = new HashSet<Name>();
        s.add(new Name("Donald", "Duck"));
        System.out.println(s.contains(new Name("Donald", "Duck")));
    }
}
```

↳ Corriger le programme