Collections en Java

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-31

Table des matières

1 L'intérêt des collections 2
1.1 Types de structures de données 2
1.2 Ressources supplémentaires 3
2 Les collections 3
3 Une hiérarchie d'Interfaces 3
4 Des classes abstraites 4
4.1 Hierarchie 4
4.2 Structures de Données Classiques 4
5 Un premier exemple 5
6 Administration des collections 5
7 Les standards et les exceptions 5
8 Des collections « génériques » ? 6
9 Un exemple de transtypage 1/2 6
10 Un exemple de transtypage 2/2 7
11 Les types génériques (1/2) 7
12 Les types génériques (2/2) 8
12.1 Les types bornés (bounded types) 8
12.1.a Syntaxe 8
12.2 Les wildcards (1/2) 9
12.3 Les wildcards (2/2) 9
13 Les collections sont génériques 9
13.1 Une liste polymorphe: 9
13.2 Une liste d'entiers: 9
14 Le parcours d'une collection (1/2) 10
15 Le parcours d'une collection - Exemple (2/2) 10
16 La conversion vers un tableau 10
16.1 Conversion correcte 11
17 Trier une collection 11

18 Egalité et Comparaison 11
18.1 Exemple 11
18.2 Utilisation 12
19 Recherche dans une collection 12
20 Utilisation des `Set` avec des arbres (`TreeSet`) 13
20.0.a TreeSet 13
21 Utilisation des Set avec des tables de hachage (HashSet) 13
21.1 HashSet 13
21.1.a Avantages 13
22 Redéfinition de hashCode() et equals() 13
23 Redéfinition de hashCode() et equals() 13
23.1 Règles importantes 14
23.2 Bonnes pratiques 14
24 L'utilisation des Maps 14
25 L'interface Map<K,V> 15
26 Fonctionnement de Map 15
27 Exemple de Map 15
27.1 Utilisation 16
28 Parcourir une Map 16
29 Les critères de choix de classe 16
30 Arguments variables 16
31 Implémentations Spécialisées de Set et Map pour les Enums 17
31.1 Exemple 17
31.2 Exemple 18
32 Eclipse Collections 18
32.1 Exemple 18
32.2 Exemple 18

1 L'intérêt des collections

Lors d'un développement, l'un des points cruciaux est le choix de structures de données adaptées. Il est souvent nécessaire de développer des solutions pour stocker des objets, les parcourir et les retrouver, avec ou sans clé. Ces structures peuvent prendre la forme de tableaux, de listes, d'arbres, de tables de hachage, etc.

1.1 Types de structures de données

- **Tableaux** : Utilisés pour stocker des éléments de même type dans un ordre séquentiel.
- **Listes** : Permettent de stocker des éléments de manière ordonnée, avec des opérations d'insertion et de suppression flexibles.

- **Arbres** : Structures hiérarchiques permettant des recherches rapides et des opérations de tri.
- **Tables de hachage** : Utilisées pour stocker des paires clé-valeur, permettant des recherches rapides basées sur des clés.

1.2 Ressources supplémentaires

Pour en savoir plus, vous pouvez consulter : - Un tutoriel - La javadoc

2 Les collections

En Java, une collection est un objet qui permet de contenir des références vers d'autres objets. Nous avons déjà vu une structure qui contient des références : les tableaux. Cependant, les collections offrent plus de flexibilité et de fonctionnalités.

Le JDK propose plusieurs éléments pour travailler avec les collections :

- **Interfaces** : Elles définissent les grands types de collections, comme List, Set, et Map.
- **Classes abstraites** : Elles implantent partiellement ces interfaces, fournissant une base pour les classes concrètes.
- **Classes concrètes** : Elles précisent les implantations des interfaces et des classes abstraites, comme ArrayList, HashSet, et HashMap.
- **Paquetage** `java.util` : Toutes les interfaces et classes de collections sont définies dans ce paquetage.
- **Itérateurs** : Un mécanisme de parcours général est proposé via les itérateurs, permettant de parcourir les éléments d'une collection de manière uniforme.
- **Outils de tri et de recherche** : Le JDK fournit également des méthodes utilitaires pour trier et rechercher dans les collections, facilitant ainsi la manipulation des données.

Ces éléments permettent de créer des structures de données flexibles et efficaces pour stocker, parcourir et manipuler des objets en Java.

3 Une hiérarchie d'Interfaces

En Java, les collections sont organisées en une hiérarchie d'interfaces qui permettent de structurer et de manipuler des groupes d'objets de manière efficace. La principale interface est `Collection`, qui se décline en plusieurs sous-types spécialisés :

- **Set** : Représente des ensembles d'éléments uniques.
- **SortedSet** : Maintient les éléments dans un ordre spécifique.
- **List** : Permet de gérer des séquences ordonnées d'éléments.
- **Queue** : Facilite la gestion des files d'attente.
- **Map** : Utilisée pour associer des clés à des valeurs.

- **SortedMap** : Maintient les paires clé-valeur dans un ordre déterminé.

Toutes ces interfaces et leurs implémentations sont définies dans le paquetage `java.util`, offrant ainsi un cadre robuste et flexible pour le développement d'applications Java.

4 Des classes abstraites

En Java, un ensemble de classes abstraites implémente les interfaces des collections, en prenant en charge les parties communes de leur fonctionnement. Parmi ces classes abstraites, on trouve :

- `AbstractCollection`
- `AbstractList`
- `AbstractQueue`
- `AbstractSequentialList`
- `AbstractSet`
- `AbstractMap`

Ces classes fournissent une base solide pour les collections en définissant des comportements génériques, ce qui permet de réduire la duplication de code. Les classes concrètes, telles que `ArrayList`, `TreeSet`, `HashMap` et `TreeMap`, héritent de ces classes abstraites et implémentent les fonctionnalités spécifiques nécessaires pour chaque type de collection. Cela permet une grande flexibilité et réutilisabilité du code, tout en assurant une cohérence dans le comportement des différentes collections.

4.1 Hierarchie

4.2 Structures de Données Classiques

En Java, les structures de données classiques sont organisées dans une hiérarchie de classes et implémentent des interfaces communes, offrant ainsi une grande flexibilité et réutilisabilité du code. Parmi les interfaces principales, on trouve `List`, `Set` et `Map`. Les `List` incluent des implémentations comme `ArrayList`, qui hérite de `AbstractList` et `AbstractCollection`, et `LinkedList`, qui permet de gérer des listes chaînées. Les `Set` comprennent des structures comme `HashSet` pour les tables de hachage et `TreeSet` pour les arbres à balance équilibrée. Enfin, les `Map` incluent `HashMap` et `TreeMap`, qui permettent de gérer des associations clé-valeur avec des tables de hachage et des arbres équilibrés, respectivement. Ces structures permettent de stocker, parcourir et manipuler des collections d'objets de manière efficace et optimisée.

5 Un premier exemple

Le programme suivant ajoute trois chaînes de caractères (« Medor », « Rex » et « Brutus ») à la liste maListe et l'affiche.

```java
import java.util.*; // Importation de toutes les classes du paquetage java.util
List maListe = new ArrayList(); // Création d'une nouvelle liste de type ArrayList
maListe.add("Medor"); // Ajout de l'élément "Medor" à la liste
maListe.add("Rex"); // Ajout de l'élément "Rex" à la liste
maListe.add("Brutus"); // Ajout de l'élément "Brutus" à la liste

System.out.println(maListe); // Affichage du contenu de la liste
```

6 Administration des collections

L'administration des collections en Java, telles que le tri, la recherche, et la détermination des valeurs minimales et maximales, est assurée par des méthodes statiques des classes suivantes :

- **Collections** :
  - Cette classe fournit des méthodes pour opérer sur ou retourner des collections.
  - Elle inclut des algorithmes polymorphes pour manipuler les collections, tels que le tri (`sort`), la recherche (`binarySearch`), et la détermination des valeurs minimales et maximales (`min`, `max`).
  - Elle propose également des « wrappers » qui retournent une nouvelle collection basée sur une collection spécifiée, comme `unmodifiableList`, `synchronizedList`, etc.
- **Arrays** :
  - Cette classe contient diverses méthodes pour manipuler les tableaux, comme le tri (`sort`) et la recherche (`binarySearch`).
  - Elle inclut également une fabrique statique (`asList`) qui permet de voir les tableaux comme des listes, facilitant ainsi leur manipulation avec les méthodes des collections.

7 Les standards et les exceptions

En Java, la plupart des classes de collections offrent des constructeurs sans paramètres ainsi que des constructeurs prenant une collection en paramètre, facilitant ainsi l'initialisation et la conversion entre différentes implémentations de collections. Depuis Java 9, des factories sont également disponibles via la classe `Arrays` et des méthodes statiques des interfaces, permettant de créer facilement des collections. Cependant, les interfaces ne peuvent pas garantir la présence de constructeurs, car elles ne peuvent pas en définir. Toutes les méthodes définies par une interface

doivent être implémentées par les classes concrètes, même si certaines méthodes peuvent ne pas avoir de sens pour certaines spécialisations. Par exemple, les collections en lecture seule ne peuvent pas implémenter les méthodes `add()` ou `put()`, et retournent donc une exception `UnsupportedOperationException` lorsqu'elles sont appelées.

```
// Création d'une liste à partir d'un tableau
List list1 = Arrays.asList("Pierre", "Paul","Paul");
// Création d'un ensemble immuable
Set set1 = Set.of("Marie","Denise","Victoire");

// Affichage des collections
System.out.println("List: "+list1);
System.out.println("Set: "+set1);
```

8 Des collections « génériques » ?

Les collections en Java représentent des ensembles d'objets, ce qui signifie que le type de retour des méthodes de l'interface est généralement `Object`. En conséquence, les objets extraits d'une collection doivent être transtypés pour être utilisés dans leur type spécifique. De plus, les fonctions qui manipulent ces collections prennent souvent des instances de `Object` en paramètre, ce qui complique le contrôle de la consistance des types. Cette approche peut entraîner des erreurs de type à l'exécution et rendre le code plus difficile à maintenir.

9 Un exemple de transtypage 1/2

```
// Classe abstraite représentant un animal
public abstract class Animal {
    // Méthode abstraite que chaque animal doit implémenter pour définir son cri
    public abstract String crier();
}

public class Chien extends Animal {
    // Implémentation de la méthode crier pour les chiens
    @Override
    public String crier() {
        return System.identityHashCode(this) + " aboie.";
    }

    // Méthode spécifique aux chiens
    public void sePromenerEnLaisse() {
```

```
        // Logique pour promener le chien en laisse
    }
}

public class Chat extends Animal {
    // Implémentation de la méthode crier pour les chats
    @Override
    public String crier() {
        return System.identityHashCode(this) + " miaule.";
    }
}
```

10 Un exemple de transtypage 2/2

```
import java.util.*;
List l = new ArrayList();
l.add(new Chien());
l.add(new Chien());
l.add(new Chat());
System.out.println("Taille de la liste: "+l.size());
```

```
l.get(0).crier();
```

```
((Animal)l.get(2)).crier();
```

```
((Chien)l.get(2)).sePromenerEnLaisse();
```

11 Les types génériques (1/2)

Un type générique (ou type paramétré) permet de définir des classes, interfaces et méthodes avec des types variables. Cela signifie que vous pouvez créer des structures de données et des algorithmes qui fonctionnent avec n'importe quel type de données, sans avoir à spécifier le type exact à l'avance.

Les avantages des types génériques sont nombreux. Tout d'abord, ils permettent une réutilisabilité du code. En utilisant des types génériques, vous pouvez écrire des méthodes et des classes qui fonctionnent avec différents types de données, ce qui réduit la duplication de code et facilite la maintenance.

Ensuite, les types génériques offrent une sécurité de type à la compilation. Cela signifie que les erreurs de type sont détectées lors de la compilation plutôt qu'à l'exécution, ce qui rend le code plus fiable et réduit les bugs.

Enfin, les types génériques éliminent les conversions explicites. Sans types génériques, vous devez souvent convertir des types de données, ce qui peut être source d'erreurs et de complexité. Avec les types génériques, ces conversions sont implicites et gérées par le compilateur, ce qui simplifie le code et améliore sa lisibilité.

12 Les types génériques (2/2)

```java
public class Couple<T1, T2> {
    private final T1 e1;
    private final T2 e2;

    public Couple(T1 p1, T2 p2) {
        this.e1 = p1;
        this.e2 = p2;
    }

    public T1 getFirst() {
        return e1;
    }

    public T2 getSecond() {
        return e2;
    }
}

// Exemple d'utilisation
Couple<Chien, Chat> c = new Couple<>(new Chien(), new Chat());

c.getFirst().crier();
```

12.1 Les types bornés (bounded types)

Les types bornés permettent de restreindre les types génériques à des sous-types spécifiques. Cela signifie que vous pouvez spécifier que le type générique doit être un sous-type d'une certaine classe ou implémenter une certaine interface.

12.1.a Syntaxe

La syntaxe pour définir un type borné est `<T extends SuperType>`. Par exemple, si vous voulez créer une méthode générique qui accepte uniquement des instances de Number ou de ses sous-classes, vous pouvez utiliser la syntaxe suivante :

Si vous voulez créer une interface générique qui accepte uniquement des instances de Runnable ou de ses sous-classes, vous pouvez utiliser la syntaxe suivante :

- Avantages :

- Permet d'utiliser des méthodes spécifiques du supertype.
- Offre plus de flexibilité tout en maintenant la sécurité de type.

12.2 Les wildcards (1/2)

Les wildcards (`?`) permettent de travailler avec des types inconnus dans les génériques. Elles sont particulièrement utiles lorsque vous travaillez avec des collections de types variés.

Il existe deux types principaux de wildcards : - `? extends T` : Utilisé pour les bornes supérieures, ce qui signifie que le type inconnu doit être un sous-type de `T`. - `? super T` : Utilisé pour les bornes inférieures, ce qui signifie que le type inconnu doit être un supertype de `T`.

12.3 Les wildcards (2/2)

Voici un exemple d'utilisation de wildcard avec une borne supérieure :

Et un exemple avec une borne inférieure :

13 Les collections sont génériques

```
List<Chien> listeDeChiens = new ArrayList<>();
listeDeChiens.add(new Chien());
listeDeChiens.add(new Chien());
for(Chien chien:listeDeChiens)
    System.out.println(chien.crier());
```

```
listeDeChiens.add(new Chat());
```

13.1 Une liste polymorphe:

```
List<Animal> animaux = new ArrayList<>(listeDeChiens);
animaux.add(new Chat());
System.out.println("Taille de la liste d'animaux: "+animaux.size());
```

```
for (Animal a:animaux)
    System.out.println(a.crier());
```

13.2 Une liste d'entiers:

```
List<Integer> l = new ArrayList<>();
l.add(3); // Utilisation de l'autoboxing
l.add(4);
```

```java
int sum = l.get(0) + l.get(1);
System.out.println("La somme est " + sum);
```

14 Le parcours d'une collection (1/2)

Les boucles traditionnelles permettent de parcourir les collections indexées (comme les listes) avec une boucle. Cependant, cette méthode présente une limitation importante : elle n'est pas idéale pour l'évolutivité car elle supprime l'encapsulation des collections. En revanche, l'utilisation d'un itérateur, qui est une instance de l'interface Iterator, permet d'énumérer les éléments d'une collection tout en maintenant l'encapsulation. Toutes les collections possèdent la méthode `iterator()` qui retourne un itérateur, et ces itérateurs peuvent être spécialisés en fonction des sous-classes de Collection, comme par exemple `ListIterator`. Les itérateurs offrent plusieurs avantages, notamment le maintien de l'encapsulation des collections et la possibilité de modifier la collection en cours de parcours.

15 Le parcours d'une collection - Exemple (2/2)

```java
for (int i=0;i<animaux.size();i++)
  System.out.println(animaux.get(i).crier()) ;
```

```java
Iterator<Animal> itAnimaux = animaux.iterator();
while (itAnimaux.hasNext())
  System.out.println(itAnimaux.next()
                     .crier().toUpperCase());
```

16 La conversion vers un tableau

La méthode `toArray()` retourne un tableau d'objets, ce qui nécessite une attention particulière au type de retour. Il est possible de passer un tableau en paramètre à `toArray(T[] a)`, mais il est déconseillé de simplement transtyper le résultat car cela peut entraîner des erreurs de type à l'exécution. Si le tableau passé en paramètre est trop petit, un nouveau tableau du même type est créé. En cas de collection vide, la méthode retourne un tableau vide et non `null`.

```java
Object[] tObjets = listeDeChiens.toArray();
System.out.println(Arrays.toString(tObjets));
for (Object c:tObjets) {System.out.println(((Chien)c).crier());};
```

16.1 Conversion correcte

```
Chien[] desChiens = new Chien[listeDeChiens.size()];
listeDeChiens.toArray(desChiens);
for (Chien c:desChiens) {System.out.println(c.crier());};
```

17 Trier une collection

Les classes `Collections` et `Arrays` proposent des méthodes de classe pour traiter les collections et les tableaux. Elles permettent en particulier de trier et de rechercher dans des listes, de faire des copies, etc. Nous avons déjà vu que l'on pouvait trier une collection de `String`. En réalité, pour qu'une collection puisse être triée, ses éléments doivent implémenter l'interface `Comparable` :

```
int compareTo(T o)
```

Cette méthode compare cet objet avec l'objet spécifié pour l'ordre. Elle retourne un entier négatif, zéro ou un entier positif selon que cet objet est respectivement moins que, égal à, ou supérieur à l'objet spécifié. Il est important de veiller à la consistance entre `Comparable` et `equals()`.

18 Egalité et Comparaison

L'interface `Comparable` et sa méthode `compareTo` définissent l'ordre naturel des objets de la classe `Animal`. En complément, la classe `ComparateurPoidsAnimal` utilise l'interface `Comparator` pour trier des objets `Animal` par leur poids, offrant ainsi une méthode de comparaison personnalisée et flexible. L'utilisation de `Comparator` permet de séparer la logique de comparaison de la classe elle-même, ce qui améliore la modularité et la flexibilité du code.

18.1 Exemple

```
public class Animal implements Comparable<Animal> {
  public final String nom ;
  public final int age ;
  public final int poids ;
  public Animal(String nom) {this(nom, -1, -1);}
  public Animal(String nom, int age, int poids)
    {this.nom=nom; this.age=age; this.poids=poids;}
  public int compareTo (Animal a) {
    return nom.compareTo(a.nom);
  }
  public boolean equals (Object o) {
    if (this == o) return true;
```

```
    if (o == null || getClass() != o.getClass()) return false;
    Animal animal = (Animal) o;
    //return nom.equals(animal.nom);
   return age == animal.age && poids == animal.poids && nom.equals(animal.nom);
  }
  public int hashCode () {
    return Objects.hash(nom, age, poids);
  }
  public String toString ( ) {
    return "Animal(nom="+nom+",age="+age+",poids="+poids+")";}
}
```

18.2 Utilisation

```
List<Animal> l = new ArrayList<Animal >() ;
Animal figaro=new Animal("Figaro", 3, 2 );
//Nom, age , poids
l.add(new Animal("Medor", 2, 5 ));
l.add(figaro);
l.add(new Animal("Brutus", 1, 15));
System.out.println(l);

Collections.sort(l); // Trier par nom
System.out.println(l);
```

```
public class ComparateurPoidsAnimal implements Comparator<Animal> {
  public int compare (Animal animal1 , Animal animal2) {
    // Integer.compare évite le débordement (overflow) de la soustraction
    return Integer.compare(animal1.poids, animal2.poids);
  }
}
```

```
// Trier par poids
Collections.sort (l , new ComparateurPoidsAnimal());
System.out.println(l);
```

19 Recherche dans une collection

```
Collections.sort(l);
System.out.println(l);
int positionFigaroNom=Collections.binarySearch(l,figaro);
System.out.println("Figaro position par nom: "+positionFigaroNom);
```

```
Collections.sort (l , new ComparateurPoidsAnimal());
System.out.println(l);
int      positionFigaroPoids      =      Collections.binarySearch(l,figaro,new
ComparateurPoidsAnimal());
System.out.println("Figaro position par poids: "+positionFigaroPoids);
```

20 Utilisation des `Set` avec des arbres (`TreeSet`)

20.0.a TreeSet

TreeSet est une implémentation de l'interface Set basée sur un arbre rouge-noir. Cela signifie que les éléments sont maintenus dans un ordre trié, et les opérations de base (insertion, suppression, recherche) ont des performances logarithmiques (O(log n)).

Avantages Maintien de l'ordre naturel : Les éléments sont automatiquement triés selon leur ordre naturel ou selon un comparateur spécifié. Recherche rapide : Les opérations de recherche sont rapides grâce à la structure de l'arbre rouge-noir.

21 Utilisation des Set avec des tables de hachage (HashSet)

21.1 HashSet

HashSet est une implémentation de l'interface Set basée sur une table de hachage. Contrairement à TreeSet, HashSet ne maintient pas l'ordre des éléments, mais offre des performances moyennes constantes (O(1)) pour les opérations de base (insertion, suppression, recherche).

21.1.a Avantages

- **Performances rapides** : Les opérations de base sont très rapides grâce à la table de hachage.
- **Utilisation efficace de la mémoire** : Idéal pour les ensembles où l'ordre des éléments n'est pas important.

22 Redéfinition de hashCode() et equals()

23 Redéfinition de hashCode() et equals()

La redéfinition correcte de `hashCode()` et `equals()` est essentielle pour utiliser une classe dans une collection basée sur une table de hachage comme `HashSet` ou `HashMap`.

23.1 Règles importantes

- Si deux objets sont égaux selon `equals()`, ils doivent avoir le même `hashCode`
- Si deux objets ont le même `hashCode`, ils ne sont pas nécessairement égaux
- `hashCode()` doit retourner la même valeur pour le même objet pendant toute son existence dans une même exécution

23.2 Bonnes pratiques

- Utiliser `Objects.hash()` pour calculer le hashCode
- Inclure les mêmes champs dans `equals()` et `hashCode()`
- Toujours redéfinir les deux méthodes ensemble

24 L'utilisation des Maps

Une `Map` est un outil essentiel en programmation, permettant d'associer une clé unique à une valeur spécifique. Chaque clé, étant unique, garantit que la méthode `equals()` retourne `false` lorsqu'elle compare deux clés différentes. De plus, pour les tables de hachage, la méthode `hashCode()` est utilisée pour générer un code de hachage pour chaque clé, ce qui permet de distribuer les entrées de manière uniforme et d'améliorer les performances de recherche. Les Maps peuvent également être implémentées à l'aide d'arbres, comme les arbres binaires de recherche, qui permettent de maintenir les clés dans un ordre trié et d'effectuer des opérations de recherche, d'insertion et de suppression en temps logarithmique. Pour ce faire, les clés doivent implémenter l'interface `Comparable`, ce qui permet de définir un ordre naturel entre elles. Les Maps sont couramment utilisées pour diverses opérations telles que l'ajout et la suppression d'entrées, la recherche d'un objet à partir de sa clé, ainsi que la récupération des clés associées à une valeur donnée. Ces fonctionnalités rendent les Maps particulièrement utiles pour organiser et accéder efficacement aux données.

En Java, une `Map` est une interface cruciale, implémentée notamment par des classes comme `HashMap` et `TreeMap`. La `HashMap` utilise une table de hachage pour offrir un accès rapide aux éléments, généralement en temps constant $\mathcal{O}(1)$. En revanche, la `TreeMap` utilise un arbre équilibré pour maintenir les clés dans un ordre trié, permettant des opérations de recherche, d'insertion et de suppression en temps logarithmique $\mathcal{O}(\log n)$. Pour que cela fonctionne, les objets clés doivent implémenter l'interface `Comparable`, ou bien un comparateur externe (`Comparator`) peut être fourni pour définir l'ordre des clés. Ces deux implémentations de `Map` sont essentielles pour organiser et accéder efficacement aux données en fonction des besoins spécifiques de performance et d'ordre.

25 L'interface Map<K,V>

```
void clear()
boolean containsKey(Object key)
boolean containsValue(Object value)
Set<Map.Entry<K,V>> entrySet()
boolean equals(Object o)
V get(Object key)
int hashCode()
boolean isEmpty()
Set<K> keySet()
V put(K key, V value)
void putAll(Map<? extends K,? extends V> t)
V remove(Object key)
int size()
Collection<V> values()
```

26 Fonctionnement de Map

En Java, une `Map` stocke par défaut des instances de `Object`, mais il est fortement recommandé d'utiliser des génériques pour spécifier les types de clés et de valeurs. Les couples clé/valeur sont manipulés via l'interface `Map.Entry`, qui fournit les méthodes `getKey()`, `getValue()`, et `setValue()`. La méthode `entrySet()` de `Map` retourne un ensemble (`Set`) de `Map.Entry`, permettant d'itérer sur les entrées de la `Map`. Il est crucial de s'assurer qu'une clé ne peut être remplacée que par une clé égale, conformément à la méthode `equals()`. Pour garantir le bon fonctionnement de la `Map`, il est conseillé de tester les opérations d'ajout et de suppression.

27 Exemple de Map

```
public final class Chien {
 public final String nom;
 public Chien(String nom) {this.nom = nom;}
 public String toString() {return "Chien(){nom="+nom+"})";}
}

Map<String,Chien> map1 = new HashMap<String,Chien>();
map1.put("Ch4", new Chien( "First"));
map1.put("Ch3", new Chien( "Medor"));
map1.put("Ch1", new Chien( "Rex"));
map1.put("Ch2", new Chien( "Medor"));
map1.put("Ch2", new Chien( "Brutus"));
System.out.println("HashMap: "+map1);
```

27.1 Utilisation

```
System.out.println ("Le chien Ch2 est "+map1.get("Ch2").nom);

Map<String,Chien> map2 = new TreeMap<String,Chien>(map1);
System.out.println("TreeMap: "+map2);

SortedMap<String,Chien> map3 = new TreeMap<String,Chien>(map1);
System.out.println("Sub SortedMap: "+map3.subMap("Ch2","Ch3"+"\0"));
```

28 Parcourir une Map

Pour parcourir une `Map` en Java, on commence par récupérer l'ensemble des clés à l'aide de la méthode `keySet()`. Ensuite, on peut parcourir cet ensemble soit avec un itérateur, soit avec une boucle `for each`. Pour chaque entrée, on peut obtenir la clé en utilisant la méthode `getKey()` et la valeur correspondante avec `getValue()`. De plus, la méthode `values()` permet de retourner directement la collection des valeurs contenues dans la `Map`. Cette approche facilite l'itération et la manipulation des données stockées dans la `Map`.

```
Set<Map.Entry<String,Chien>> setChiens = map1.entrySet();

for (Map.Entry<String,Chien> uneEntree : setChiens)
    System.out.print(uneEntree.getKey()+ " :" +uneEntree.getValue());
```

29 Les critères de choix de classe

En Java, l'un des objectifs principaux est de maximiser la réutilisabilité des programmes. Pour y parvenir, il est essentiel de choisir la classe ou l'interface la plus adaptée à chaque situation. Un programme doit être conçu pour être utilisable dans le plus grand nombre de cas possibles. Pour cela, il est recommandé de manipuler les objets via des interfaces aussi générales que possible pour les paramètres, afin de garantir une flexibilité maximale. En revanche, pour les types de retour, il est préférable de choisir des classes spécifiques offrant le plus de fonctionnalités. Cependant, ce choix peut poser des problèmes en cas de changement d'implémentation, d'où l'importance de privilégier également des interfaces plus générales pour maintenir la compatibilité et la flexibilité du code.

30 Arguments variables

Il n'est pas nécessaire d'utiliser une collection pour passer un ensemble d'arguments du même type à une méthode. En Java, vous pouvez utiliser les varargs (arguments variables) pour simplifier ce processus. Les varargs permettent de passer un nombre

variable d'arguments à une méthode sans avoir à créer explicitement un tableau ou une collection. Pour déclarer une méthode avec des varargs, utilisez l'ellipse (...) après le type de l'argument.

```
public static int somme (int... entiers) {
  int total = 0 ;
  for (int e:entiers) total+=e;
  return total ;
}
```

```
System.out.println("1+2 = "+somme(1,2));
```

```
System.out.println("1+2+3+4 = "+somme (1,2,3,4));
```

31 Implémentations Spécialisées de Set et Map pour les Enums

En Java, des implémentations spécialisées de Set et Map sont disponibles pour une utilisation avec les énumérations (enum). EnumSet est une implémentation de Set haute performance, basée sur un bit-vecteur, où tous les éléments doivent appartenir à un seul type d'énumération. Cette structure permet des opérations rapides et efficaces sur les ensembles d'énumérations. De même, EnumMap est une implémentation de Map haute performance, basée sur un tableau, où toutes les clés doivent également appartenir à un seul type d'énumération. Ces implémentations sont particulièrement utiles pour les cas où les performances et l'efficacité sont cruciales, tout en garantissant une gestion optimale des énumérations.

31.1 Exemple

```
public class Person {
    public enum PetType {CAT, DOG};
    private String name;
    private EnumSet<PetType> pets;
        public Person(String name, Set<PetType> petsTypes) {this.name=name;
this.pets=EnumSet.copyOf(petsTypes);}
    public boolean hasPet(PetType petType) {return pets.contains(petType);}
    public String toString() {return name;}
}
```

31.2 Exemple

```
public enum Jour {
  LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE;
}

EnumMap<Jour, Animal> animalParJour = new EnumMap<Jour, Animal>(Jour.class);

animalParJour.put(Jour.MARDI,new Animal("Rex"));
animalParJour.put(Jour.VENDREDI,new Animal("Médor"));
```

```
System.out.println(animalParJour);
System.out.println(animalParJour.get(Jour.VENDREDI));
```

32 Eclipse Collections

Eclipse Collections est une alternative externe au JDK, offrant une bibliothèque beaucoup plus complète pour les développeurs Java. Cette bibliothèque propose de nombreuses structures de données supplémentaires, qu'elles soient mutables ou immutables, permettant ainsi une plus grande flexibilité dans la gestion des collections. De plus, Eclipse Collections intègre pleinement les expressions lambda et les streams, facilitant ainsi une programmation fonctionnelle efficace. Pour en savoir plus, vous pouvez visiter le site officiel https://eclipse.dev/collections/

32.1 Exemple

```
%maven org.eclipse.collections:eclipse-collections-api:11.1.0
%maven org.eclipse.collections:eclipse-collections:11.1.0
```

```
MutableIntList intList = IntLists.mutable.of(1, 2, 3);
intList;
```

32.2 Exemple

```
MutableList<Person> mutableListWith =
  Lists.mutable.with(
    new Person("P1",Set.of(Person.PetType.CAT)),
    new Person("P2",Set.of(Person.PetType.DOG)),
    new Person("P3",Set.of(Person.PetType.CAT, Person.PetType.DOG)));

List<Person> peopleWithCats =
  mutableListWith
    .stream()
```

```
    .filter(person -> person.hasPet(Person.PetType.CAT))
    .collect(Collectors.toList());

System.out.println(peopleWithCats);
```