JAVA
Types de classes, Interfaces et Collections

Lydia YATAGHENE
lydia.yataghene@junia.com

Classes abstraites

Classe abstraite

- L'héritage est un moyen de mutualiser du code dans une classe parente. Parfois cette classe représente une abstraction pour laquelle il n'y a pas vraiment de sens de créer une instance.
- Les classes abstraites sont des modèles de classes non instantiables, utilisées pour centraliser des méthodes et attributs.
- On utilise le mot-clé abstract.

```
public abstract class Personnage {

Private string nom;

}
```

```
Personnage personne= new Personnage(); // erreur
```

Classes et méthodes abstraites

- Une méthode abstraite est une méthode dont seul le prototype est défini.
- Ses séquences d'instructions seront définies par héritage.
- Créer une méthode abstraite impose de définir la classe associée comme abstraite.

```
public abstract class Personnage {
 public Personnage(String name) ;
 public abstract presentation() ;
 }
```

Classe abstraite

- Regrouper des prototypes de méthodes.
- Ne pas implémenter chaque méthode (méthodes abstraites).
- Définir au besoin des méthodes par défaut.

Interfaces

Interfaces

- Une interface est une classe regroupant un ensemble de prototypes.
- Une interface introduit une abstraction pure qui permet un découplage maximal entre un service et son implémentation.
- On retrouve ainsi les interfaces au cœur de l'implémentation de beaucoup de bibliothèques et de frameworks.
- Le mécanisme des interfaces permet d'introduire également une forme simplifiée d'héritage multiple.
- Elles ne peuvent être instanciées.
- Elles permettent:
  - D'imposer un modèle à des classes héritées
  - De définir des standards de développement
  - De pré-implémenter des méthodes de classe

Interfaces

- Une interface contient un ensemble de méthodes.
- On la définit en utilisant le mot-clé interface.
- Comme pour une classe, une interface a une portée, un nom et un bloc de déclaration. Une interface est déclarée dans son propre fichier qui porte le même nom que l'interface.
- Une interface introduit un nouveau type d'abstraction qui définit à travers ces méthodes un ensemble d'interactions autorisées. Une classe peut ensuite implémenter une ou plusieurs interfaces.

```
public interface Combattant {
 public void attack(Personnage p) ;
 public void defense(Combattant c) ;
 public default void runAway(){goHome() ;}
 }
```

Interfaces

- Les méthodes d'une interface sont par défaut public et abstract.
- Une classe implémente les interfaces grâce au mot-clé **implements**.
- Une classe concrète doit fournir une implémentation pour toutes les méthodes d'une interface, soit dans sa déclaration, soit parce qu'elle en hérite.
- Attributs et méthodes statiques.

Interfaces
Héritage d'interface

- Une interface peut hériter d'autres interfaces. Contrairement aux classes qui ne peuvent avoir qu'une classe parente, une interface peut avoir autant d'interfaces parentes que nécessaire. Pour déclarer un héritage, on utilise le mot-clé extends.

```
public interface <nom interface> [extends <nom interface 1> <nom interface 2> : : : ] {

 // méthodes ou des attributs static

 }
```

- Une classe concrète qui implémente une interface doit donc disposer d'une implémentation pour les méthodes de cette interface mais également pour toutes les méthodes des interfaces dont cette dernière hérite.

- L'héritage d'interface permet d'introduire de nouveaux types par agrégat.

Interfaces
Implémentation par défaut

Il est parfois difficile de faire évoluer une application qui utilise intensivement les interfaces.

- En ajoutant une nouvelle méthode à notre interface, nous devons fournir une implémentation pour cette méthode dans toutes les classes que nous avons créées pour qu'elles continuent à compiler. Mais si d'autres équipes de développement utilisent notre code et ont, elles-aussi, créé des implémentations pour l'interface Compte, alors elles devront adapter leur code au moment d'intégrer la dernière version de notre interface.
- Comme les interfaces servent précisément à découpler deux implémentations, elles sont très souvent utilisées dans les bibliothèques et les frameworks. D'un côté, les interfaces introduisent une meilleure souplesse mais, d'un autre côté, elles entraînent une grande rigidité car il peut être difficile de les faire évoluer sans risquer de casser des implémentations existantes.
- Pour palier partiellement à ce problème, une interface peut fournir une implémentation par défaut de ses méthodes. Ainsi, si une classe concrète qui implémente cette interface n'implémente pas une méthode par défaut, c'est le code de l'interface qui s'exécutera. Une méthode par défaut doit obligatoirement avoir le mot-clé default dans sa signature.

Héritage et interfaces

- On n'hérite pas d'une interface, on l'implémente.

```
public class Gaulois implements Combattant {
public void attaque(Personnage p){
 gourdePotionMagique.bois() ;
 while(p.isDebout())
 coupsDePoing(p) ;
 }
 public void defend(Combattant c){
 esquive() ;
 attaque(c) ;
 }
 }
```

Interfaces

- Pas de constructeur
- Toutes les méthodes d’une interface sont définies implicitement comme abstraites
- Pas d’attributs non statiques
- Non instantiable
- Peut être étendue par une autre interface

Enumérations

Énumération

- Une énumération définit un ensemble de valeurs finies que peut prendre un objet.
- Cela permet de définir des types dont la valeur est imposée.
- On les définit avec le mot-clé **enum**.
- Ensemble d'entiers statiques

```
public enum CardColor {
CARREAU ,TREFLE ,PIQUE , COEUR
}
```

Énumération

- Il est possible de créer des éléments du type de l'énumération.

```
public enum CardColor {
CARREAU ,TREFLE ,PIQUE , COEUR
}
 CardColor color ;
 if ( color == CardColor . CARREAU ) {}
 color = CardColor . COEUR ; {}
```

Énumération

- ordinal : Renvoie le nombre de valeurs possibles
- toString : Renvoie la conversion en String de la valeur
- values : Renvoie un tableau contenant toutes les valeurs
- valueOf : Renvoie la valeur entière associée à l'élément

```
public enum CardColor {
CARREAU ,TREFLE ,PIQUE , COEUR
}
CardColor color = COEUR ;


System.out.println ( color.valueOf ());
```

Énumération

- Chaque élément d'un énumération n'existe qu'une fois en mémoire. L'énumération garantit que l'unicité de la valeur est équivalente à l'unicité en mémoire. Cela signifie que l'on peut utiliser l'opérateur **==** pour comparer des variables, des attributs et des paramètres du type d'énumération. L'utilisation de l'opérateur **==** est même considérée comme la bonne façon de comparer les énumérations.

Classe interne

Classe interne

- Une classe interne est une classe définie au sein d'une autre classe.
- Les classes internes permettent de définir une classe dans un contexte.
- C'est une alternative à l'héritage.
- Ensemble d'attributs ou de constantes.
- La déclaration des classes internes peut se faire dans l'ordre que l'on souhaite à l'intérieur du bloc de déclaration de la classe englobante.
- Les classes internes peuvent être ou non déclarées static. Ces deux cas correspondent à deux usages particuliers des classes internes.

Classe interne

```
public class ClasseEnglobante {
  public static class ClasseInterneStatic {
  }
  public class ClasseInterne {
public int getValue ();
 }
  }
ClasseEnglobante a = new ClasseEnglobante();
ClasseEnglobante.ClasseInterne b = a.new ClasseInterne();
```

- Deux fichiers de bytecode: ClasseEnglobante.class, ClasseEnglobante$ClasseInterne.class

Classes internes

- Scope d’une classe
  - public : Tous les packages
  - protected : package + sous-classes
  - private : package uniquement

Classe static

- Une classe interne static est indépendante de sa classe englobante
- . Il est possible d'accéder à une classe statique sans instancier la classe externe, en utilisant d'autres membres statiques.
- Une classe static n'a pas accès aux variables d'instance et aux méthodes de la classe externe.

```
public class ClasseEnglobante {
  public static class ClasseInterneStatic {
  }
  public class ClasseInterne {
public int getValue ();
 }
  }
}
```

Classes internes

- Une classe interne qui n'est pas déclarée avec le mot-clé **static** est liée au contexte d'exécution d'une instance de la classe englobante.
- La classe englobante et la classe interne partage le même espace privé. Cela signifie que les attributs et les méthodes privés déclarés dans la classe englobante sont accessibles à la classe interne. Réciproquement, la classe englobante peut avoir accès aux éléments privés de la classe interne.
- Une classe interne static est souvent utilisée pour éviter de séparer dans des fichiers différents de petites classes utilitaires et ainsi de faciliter la lecture du code.
- Le nom complet de classe interne inclut celui de la classe englobante et les deux classes partagent le même espace privé. Mais surtout, une classe interne maintient une référence implicite sur un objet de la classe englobante. Cela signifie que :
  - Une instance d'une classe interne ne peut être créée que par un objet de classe englobante : c'est-à-dire dans le corps d'une méthode ou dans le corps d'un constructeur de la classe englobante.
  - Une instance d'une classe interne a accès directement aux attributs de l'instance dans le contexte de laquelle elle a été créée.

Classes anonymes

- Une classe anonyme est une classe qui n'a pas de nom.
- Une classe anonyme est une classe définie à la volée au sein d'une autre classe.
- Elles sont dédiées à une utilisation particulière.
- Elles sont utilisées pour l'implémentation locale d'une interface (handlers, listeners, ...).
- Classes anonymes et méthodes
  - Il est possible de définir une classe en interne d'une méthode.
  - Son scope sera limité à la méthode.
  - Il est nécessaire de redéfinir les variables partagées comme final (avant java8).

Collections

Collections

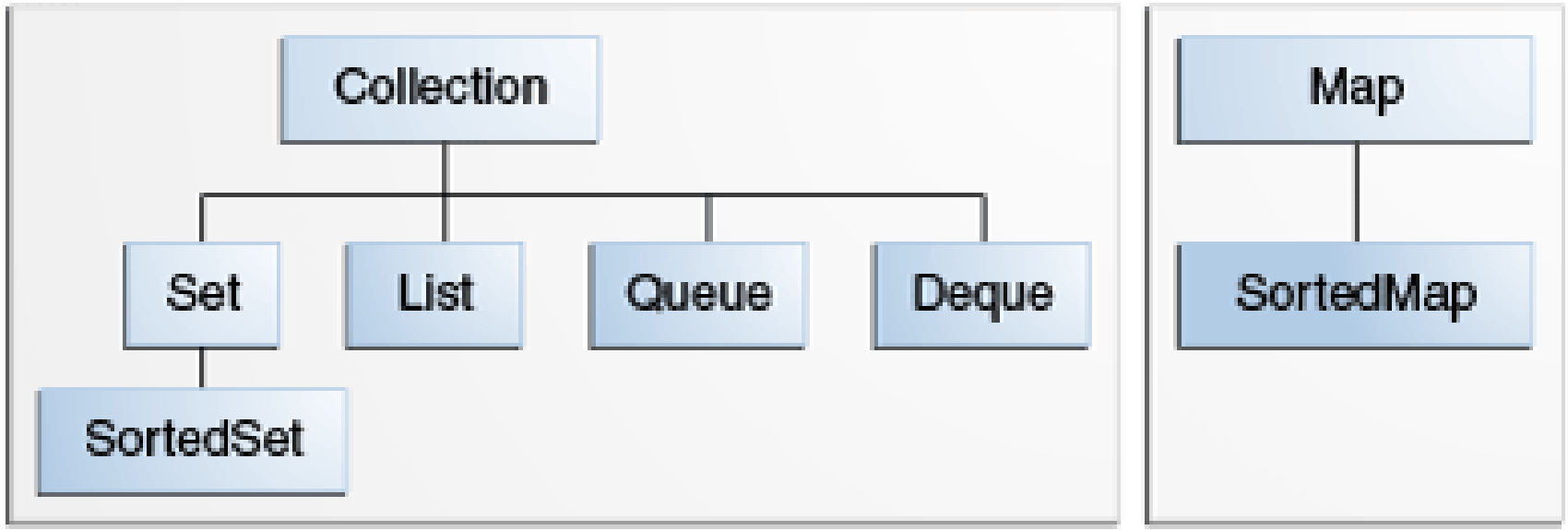
- Collection = Regroupement d'objets (d'éléments).
- Le framework « Collections » est une architecture unifiée pour représenter et manipuler des collections indépendamment de leur représentation.
- Constitué :
  - D'interfaces sous forme de hiérarchie.
  - D'implémentations réutilisables.
  - D'algorithmes préexistants de calculs, de recherche et de tris sur les objets qui implémentent ces interfaces.

Collections

- Bénéfices :
  - Allègement de l'effort de programmation (gain de productivité).
  - Amélioration des temps d'exécution et de la qualité des programmes (algorithmes fiables et optimisés).
  - Meilleure interopérabilité entre différentes API.
  - Réutilisabilité des composants.
- Différences:
  - Méthodes intégrées.
  - Algorithmes de tri, de recherche, de parcours.
  - Gestion des accès concurrents

Collections

- **Hiérarchie des interfaces**



« Collection »
Regroupement d'objets
triés ou non

« Map »
Regroupement d'objets
par paire **Clé/Valeur**

Collections

- **Interface « List »**

SPÉCIFICITÉS :

- Autorise les doublons
- Autorise l'utilisation d'index
- Autorise les éléments « null »

MÉTHODES :

- add(…) / addAll(…)
- get(…)
- indexOf(…)
- remove(…)
- set(…)
- subList(…)

Collections
Interface « List »

- **Classe ArrayList**
  - Spécificités :
    - Remplace avantageusement les tableaux []
    - Taille dynamique
    - Implémentation la plus simple de l'interface List
  - A utiliser dans la plupart des cas.
  - A privilégier à LinkedList si l'on ne maîtrise pas cette dernière.

Collections
Interface « List »

- **Classe ArrayList**

```
// initialisation de la liste
List<String> maListe = new ArrayList<>() {
    {
        add("Test0");
        add("Test1");
        add("Test2");
    }
};

maListe.add("Test3"); // ajout d'un élément
System.out.println(maListe.get(1)); // Test1
```

Collections
Interface « List »

- **Classe ArrayList**

```
// ajout de plusieurs éléments
String[] tableau = new String[] { "Test4", "Test5", "Test6" };
maListe.addAll(Arrays.asList(tableau)); // méthode asList() pour la conversion

System.out.println(maListe.contains("Test2")); // true
maListe.remove("Test2"); // suppression d'un élément
System.out.println(maListe.contains("Test2")); // false
```

Collections
Interface « List »

- **Classe ArrayList**

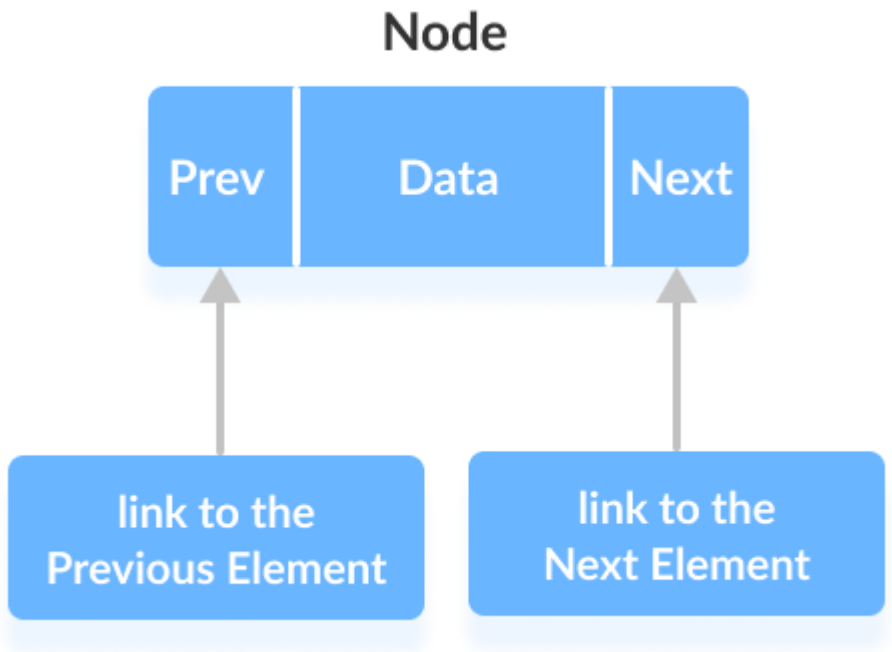
```
// parcours de la liste
for (String s : maListe) {
    System.out.println(s);
}

System.out.println(maListe.isEmpty()); // false
maListe.clear(); // suppression de tous les éléments
System.out.println(maListe.isEmpty()); // true
```

Collections
Interface « List »

- **Classe LinkedList**
  - Spécificités :
    - Permet d'ajouter des éléments en tête ou fin de liste
      - Comportement FIFO ou LIFO
    - Liste optimisée :
      - Pour l'ajout d'éléments en tête ou fin de liste
      - Pour l'ajout ou la suppression d'éléments à l'intérieur de la liste
      - Pour le parcours de liste
    - A ne pas utiliser :
      - Pour les accès indexés (méthode get() coûteuse)

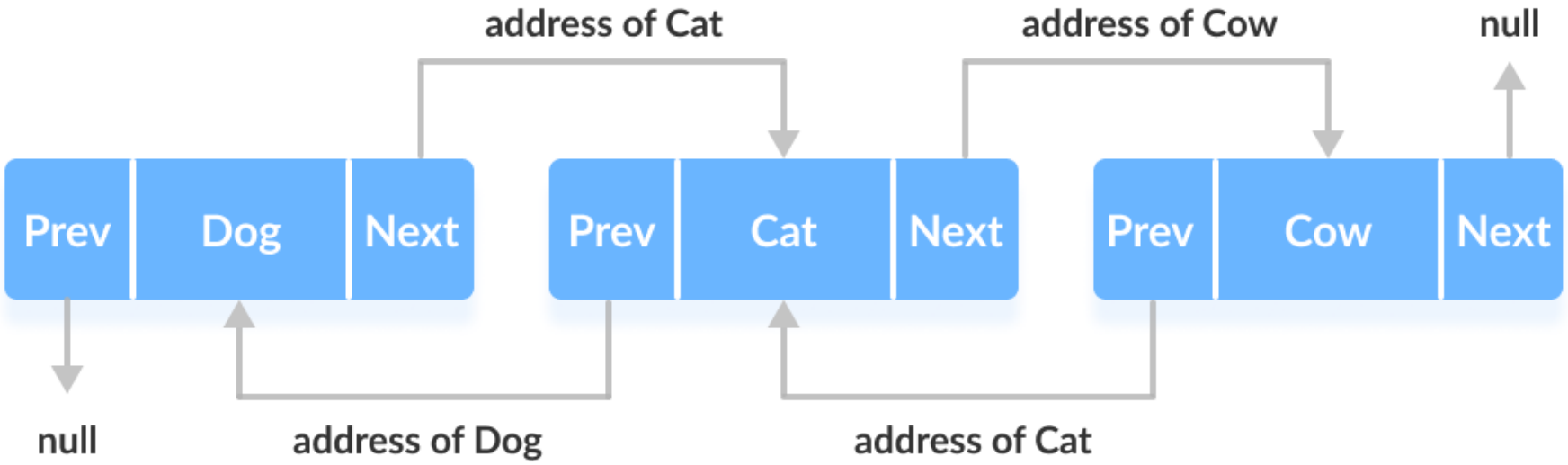
Collections
Interface « List »

- **Classe LinkedList**
  - Liste doublement chaînée



Collections
Interface « List »

- **Classe LinkedList**
  - Liste doublement chaînée



Collections
Interface « List »

- **Classe LinkedList**

```
// initialisation de la liste chaînée
List<String> maListeChainee = new LinkedList<>() {
    {
        add("Test1");
        add("Test2");
        add("Test3");
    }
};
```

Collections
Interface « List »

- **Classe LinkedList**

```
((LinkedList<String>) maListeChainee).addFirst("Test0"); // ajout en tête
((LinkedList<String>) maListeChainee).addLast("Test4"); // ajout en queue
System.out.println(((LinkedList<String>) maListeChainee).getFirst()); // Test 0
System.out.println(((LinkedList<String>) maListeChainee).getLast()); // Test 4
```

Collections
Interface « Set »

- **Interface « Set »**
- Spécificité :
  - Ne permet pas les doublons
- 4 implémentations principales :
  - HashSet, TreeSet, LinkedHashSet, EnumSet

Collections
Interface « Set »

- **HashSet** :
  - Collection à usage général
  - N’accepte pas les doublons.
  - L’ordre des éléments n’est pas conservé.

Collections
Interface « Set »

- **HashSet** :

```
// initialisation du set
Set<String> monSet = new HashSet<>() {
    {
        add("Test0");
        add("Test1");
        add("Test2");
    }
};
monSet.add("Test1"); // ajout d'un doublon

// Affichage :
// Test1
// Test0
// Test2
// => L'ordre n'est pas conservé
```

Collections
Interface « Set »

- **TreeSet** :
  - N’accepte pas les doublons.
  - Les éléments sont triés (ordre naturel ou personnalisé avec un Comparator).

```
Set<String> monTreeSet = new TreeSet<>() {
    {
        add("A");
        add("Z");
        add("E");
        add("R");
        add("T");
        add("Y");
    }
};
System.out.println(monTreeSet); // [A, E, R, T, Y, Z]
```

Collections
Interface « Set »

- **TreeSet** :
  - N’accepte pas les doublons.
  - Les éléments sont triés (ordre naturel ou personnalisé avec un Comparator).

```
Set<Integer> monTreeSet2 = new TreeSet<>() {
    {
        add(8);
        add(57);
        add(-6);
        add(0);
        add(999);
        add(888);
        add(999);
    }
};
System.out.println(monTreeSet2); // [-6, 0, 8, 57, 888, 999]
```

Collections
Interface « Set »

- **LinkedHashSet** :
  - N’accepte pas les doublons.
  - L’ordre des éléments est conservé.

```
Set<String> monLHS = new LinkedHashSet<>() {
    {
        add("Test3");
        add("Test2");
        add("Test1");
        add("Test0");
    }
};
System.out.println(monLHS); // [Test3, Test2, Test1, Test0]
```

Collections
Interface « Set »

- **LinkedHashSet** :
  - N’accepte pas les doublons.
  - L’ordre des éléments est conservé.

```
Set<Integer> monLHS2 = new LinkedHashSet<>() {
    {
        add(8);
        add(57);
        add(-6);
        add(0);
        add(999);
        add(888);
        add(999);
    }
};
System.out.println(monLHS2); // [8, 57, -6, 0, 999, 888]
```

Collections
Interface « Set »

- **EnumSet** :
  - N’accepte pas les doublons.
  - Set composé d’objets énumérés.
  - Les éléments sont triés.
  - Aucun accès direct à un élément de la collection.

Collections
Interface « Set »

- **EnumSet** :

```
enum Fruit {
    ABRICOT, BANANE, CERISE, FRAMBOISE, ORANGE
};

public static void main(String[] args) {

    // Creating a set
    EnumSet<Fruit> set1, set2, set3, set4;

    // Adding elements
    set1 = EnumSet.of(Fruit.FRAMBOISE, Fruit.CERISE, Fruit.BANANE, Fruit.ABRICOT);
    set2 = EnumSet.complementOf(set1);
    set3 = EnumSet.allOf(Fruit.class);
    set4 = EnumSet.range(Fruit.ABRICOT, Fruit.CERISE);
    System.out.println("Set 1: " + set1); // Set 1: [ABRICOT, BANANE, CERISE, FRAMBOISE]
    System.out.println("Set 2: " + set2); // Set 2: [ORANGE]
    System.out.println("Set 3: " + set3); // Set 3: [ABRICOT, BANANE, CERISE, FRAMBOISE, ORANGE]
    System.out.println("Set 4: " + set4); // Set 4: [ABRICOT, BANANE, CERISE]
```

Collections
Interface « Map »

- **Interface « Map »**
- Spécificités :
  - Associations de type clé/valeur (dictionnaire) :
    - Clés uniques.
    - Valeurs => sans importance.
  - Objectif : obtenir un objet à partir de sa clé.
  - La clé peut être un type primitif ou un objet.
- Méthodes :
  - keySet() => ensemble contenant toutes les clés (pas de doublons).
  - values() => collection regroupant toutes les valeurs (doublons possibles).

Collections
Interface « Map »

- **HashMap**
- Spécificités :
  - Association clé/valeur
  - Clés uniques
  - 1 seule clé « null » possible, illimité pour les valeurs
  - L'ordre des éléments n'est pas conservé.

Collections
Interface « Map »

- **HashMap**

```
Map<Integer, String> maHashMap = new HashMap<>() {
    {
        put(1, "Data1");
        put(23, "Data2");
        put(70, "Data3");
        put(4, "Data4");
        put(2, "Data5");
    }
};
System.out.println(maHashMap);
// {1=Data1, 2=Data5, 4=Data4, 70=Data3, 23=Data2}
```

Collections
Interface « Map »

- **TreeMap**
  - Spécificités :
    - Association clé/valeur
    - Clés uniques
    - 1 seule clé « null » possible, illimité pour les valeurs
    - Les éléments sont triés (ordre naturel ou personnalisé avec un Comparator).

Collections
Interface « Map »

- **TreeMap**

```
Map<Integer, String> maTreeMap = new TreeMap<>() {
    {
        put(1, "Data1");
        put(23, "Data2");
        put(70, "Data3");
        put(4, "Data4");
        put(2, "Data5");
    }
};
System.out.println(maTreeMap);
// {1=Data1, 2=Data5, 4=Data4, 23=Data2, 70=Data3}
```

Collections
Interface « Map »

- **LinkedHashMap**
  - Spécificités :
    - Association clé/valeur
    - Clés uniques
    - 1 seule clé « null » possible, illimité pour les valeurs
    - L'ordre des éléments est conservé.

Collections
Interface « Map »

- **LinkedHashMap**

```
Map<Integer, String> maLHM = new LinkedHashMap<>() {
    {
        put(1, "Data1");
        put(23, "Data2");
        put(70, "Data3");
        put(4, "Data4");
        put(2, "Data5");
    }
};
System.out.println(maLHM);
// {1=Data1, 23=Data2, 70=Data3, 4=Data4, 2=Data5}
```

Collections
Interface « Map »

- **EnumMap**
  - Spécificités :
    - Association clé/valeur
    - Clés uniques issues d'une même énumération
    - Clé « null » impossible, illimité pour les valeurs

Collections
Interface « Map »

- **EnumMap**

```
// création de l'énumération
public enum Jour {
    LUNDI, MARDI, MERCREDI, JEUDI
};
```

```
// initialisation de la map
EnumMap<Jour, String> map = new EnumMap<Jour, String>(Jour.class) {
    {
        put(Jour.LUNDI, "1");
        put(Jour.MARDI, "2");
        put(Jour.MERCREDI, "3");
        put(Jour.JEUDI, "4");
    }
};
System.out.println(map);
// {LUNDI=1, MARDI=2, MERCREDI=3, JEUDI=4}
```

Collections
Parcours d'une « Collection »

- **Parcours d'une « Collection »**
  - Opérations d'agrégations (streams)
    - Avantages :
      - Moins de lignes de code.
      - Optimisation des traitements.
      - Les collections bénéficient des fonctions d'agrégation de la programmation fonctionnelle (description du résultat souhaité, pas de la manière de l'obtenir).
      - Traitements parallélisés possibles.

Collections
Parcours d'une « Collection »

- Streams

```
List<String> maListeChainee2 = new LinkedList<>() {
    {
        add("Test1");
        add("Test2");
        add("Test3");
        add("Test22");
    }
};
((LinkedList<String>) maListeChainee2).stream().forEach(
    e -> System.out.println(e)
); // Test1, Test2, Test3, Test22
```

Collections
Parcours d'une « Collection »

- Structure for-each

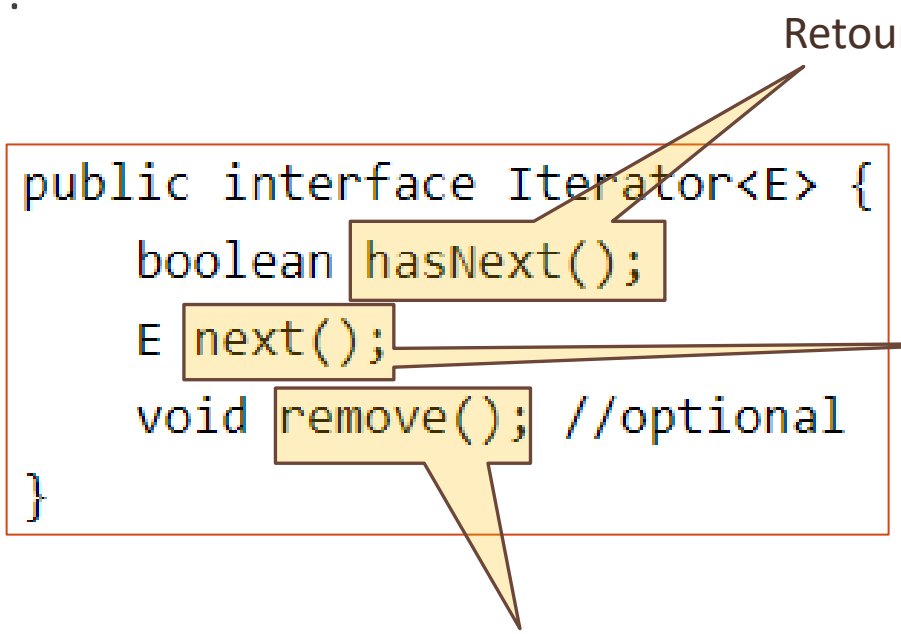
```
for (String s : maListeChainee2) {
     System.out.println(s);
}
// Test1, Test2, Test3, Test22
```

Collections
Parcours d'une « Collection »

- Iterator
  - Définition :
    - Un iterator est un objet qui permet de traverser une collection et de supprimer des éléments de cette même collection de manière sélective.
    - Un iterator est obtenu en appelant la méthode iterator() de la collection.
  - A utiliser pour :
    - Supprimer l'élément courant (for-each ne fonctionne pas)
    - Itérer sur des collections en parallèle

Collections
Parcours d'une « Collection »

- Iterator
  - Interface :



- Supprime le dernier élément retourné par next().
- Ne peut être appelée qu'une seule fois à chaque next().
- Seule manière sécurisée de modifier une collection pendant une itération.

Collections
Parcours d'une « Collection »

- Iterator

Initialisation de l'itérateur depuis la méthode iterator() de la collection

```
// Ex : suppression des chiffres pairs d'une liste d'entiers
List<Integer> c = new ArrayList<>(
    Arrays.asList(new Integer[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 })
);

// parcours de la liste grâce à l'itérateur
for (Iterator<?> it = c.iterator(); it.hasNext();) { // initialisation de la boucle
    if ((Integer) it.next() % 2 == 0) { // récupération du prochain élément
        it.remove(); // suppression de l'élément
    }
}
System.out.println(c); // [1, 3, 5, 7, 9]
```

Tant qu'il y a des éléments dans l'itérateur

Suppression de l'élément courant

Récupération du prochain élément

Collections

- **Parcours d'une « Map »**
  - Utilisation d'une boucle for()

```
Map<Integer, String> map = new LinkedHashMap<>() {
    {
        put(1, "Data1");
        put(23, "Data2");
        put(70, "Data3");
        put(4, "Data4");
        put(2, "Data5");
    }
};
for (var entry : map.entrySet()) {
    System.out.println(entry.getKey() + "/" + entry.getValue());
}
// 1/Data1, 23/Data2, 70/Data3, 4/Data4, 2/Data5
```

Collections
Tri (sorting)

- En Java, il existe 2 interfaces pour spécifier un ordre de tri :
  - Comparable (ordre naturel, ex. ordre alphanumérique, ordre alphabétique, ordre chronologique…)
  - Comparator (ordre de tri quelconque)

Collections
Tri (sorting)

- Comparable :
  - Interface permettant de comparer 2 objets d'un même type (tri naturel).
  - A utiliser pour définir un ordre de tri « simple » par défaut.
  - Ne définit qu'une seule méthode : int compareTo(Object)
  - Renvoie :
    - Valeur entière négative si l'objet courant est inférieur à l'objet fourni.
    - Valeur entière positive si l'objet courant est supérieur à l'objet fourni.
    - 0 (zéro) s'ils sont égaux.

Collections
Tri (sorting)

- Comparable :
  - Les classes String et Date implémentent cette interface. En appelant la méthode Collections.sort(maListe); la liste sera triée (tri naturel) automatiquement en fonction du type des objets la composant.
  - Il est possible également de créer ses propres types de tri en implémentant l'interface Comparable.