Programmation Objet Avancée

Kevin Heraud

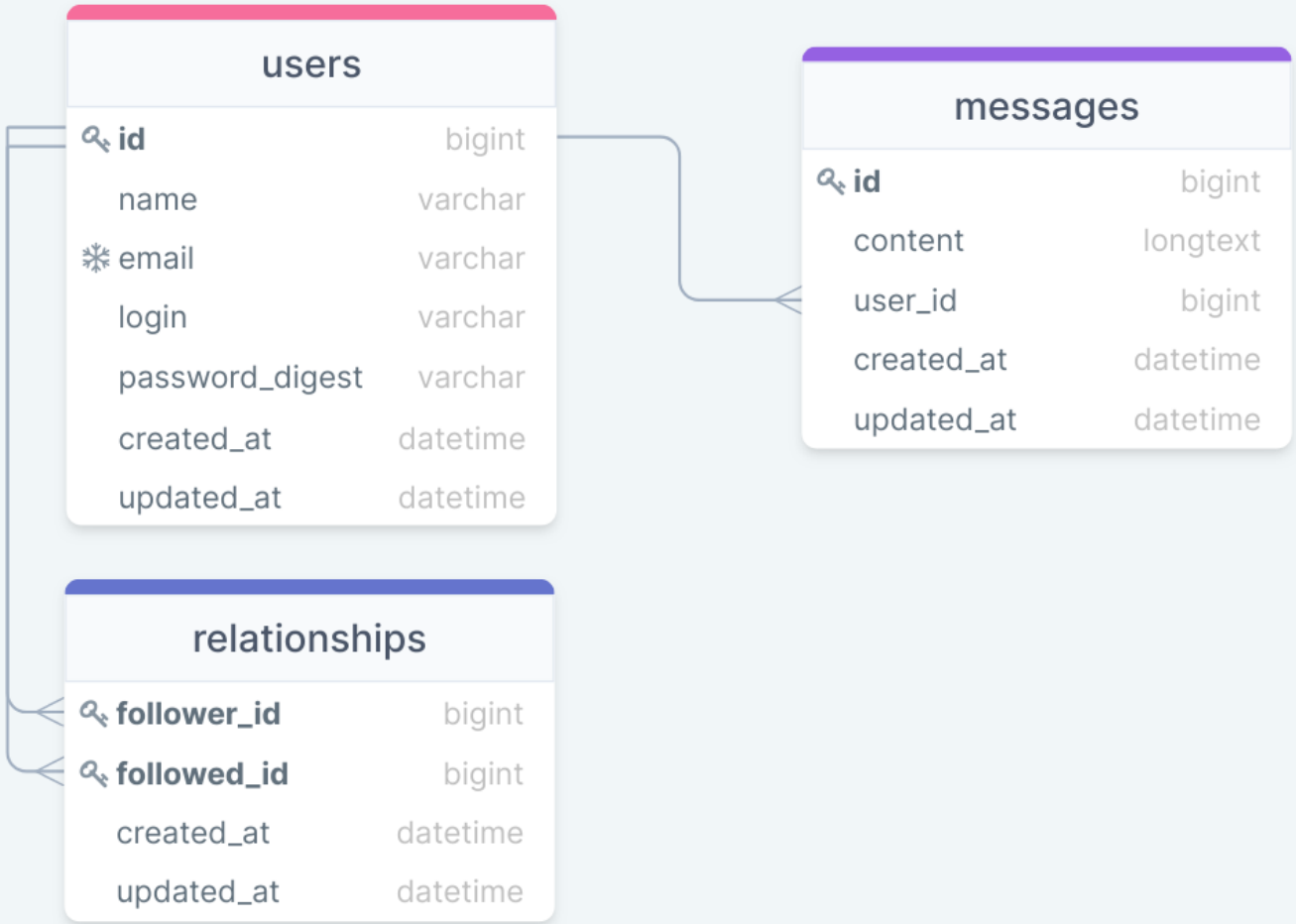
Cours 3 : base de données, collections et types énumérés

Département Informatique
Université de Tours

Accès à une base de données en Java

- Les différents type de **SGBD** (système de gestion de base de données) :
  - hiérarchiques
  - réseau
  - objet
  - NoSQL (clé/valeur, graphe, document et colonne)
  - **relationnelles** : dans ce type de bases, les données sont organisées en tables. Les bases de données de type relationnelle permettent d'assurer la cohérence des données ("Primary Key"/"Foreign Key")

Exemple de base de données relationnelle



drawSQL

Le langage SQL

- il est possible d'interroger une base de données avec le langage **SQL**
  - **S**tructured **Q**uery **L**anguage

```
SELECT id, name, email
FROM users
WHERE email = 'kevin.heraud@worldline.com'
AND password_digest = SHA2('foo', 256)
```

- trois types de requêtes en SQL
  1. **sélection** d'enregistrements ou parties d'enregistrements
  2. **mise à jour** d'enregistrements : ajout, suppression, modification de certains champs
  3. **gestion** : création ou suppression de tables, création d'index

SGBDR et JDBC

- un **Système de Gestion de Base de Données Relationnelles** (SGBDR) est un logiciel permettant de créer et d'utiliser des base de données relationnelles
  - Oracle, SQL Server, PostgreSQL, MySQL, ...
- un SGBDR qui fournit un pilote **Java Database Connectivity** (JDBC) permet à tout programme d'y accéder
  - pilote = ensemble de classes Java qui permettent l'utilisation du SGBDR par le biais de requêtes SQL

Utilisation de JDBC en 5 étapes

1. **chargement** du pilote JDBC
   - la documentation du driver fournit le nom de la classe à utiliser.
2. **connexion**
   - on créé une connection en précisant l'URL de la BD
   - la syntaxe URL peut varier en fonction du SGBDR
3. **exécution** de la requête via un `Statement` qui encapsule une requête SQL
4. **traitement** des résultats
5. **fermeture** de la connexion

Choix du pilote

- dans le cas du SGBDR MySQL, il existe des **connectors** pour tous les principaux languages

Developed by MySQL	
ADO.NET Driver for MySQL (Connector/NET)	Download
ODBC Driver for MySQL (Connector/ODBC)	Download
JDBC Driver for MySQL (Connector/J)	Download
Node.js Driver for MySQL (Connector/Node.js)	Download
Python Driver for MySQL (Connector/Python)	Download
C++ Driver for MySQL (Connector/C++)	Download
C Driver for MySQL (Connector/C)	Download
C API for MySQL (mysqlclient)	Download

- dans le cas de Java, il faut télécharger `Connector/J`

Configuration du pilote

- ajouter le fichier archive `jar` au projet et configurer le `path`
- configurer un objet *gestionnaire de pilotes*, instance de `DriverManager` qui gère les pilotes de BD
- il suffit de fournir la référence du pilote concerné pour rechercher et enregistrer le pilote

```
1 // enregistrement du pilote
2 Class.forName("com.mysql.cj.jdbc.Driver");
```

Configuration du pilote

- cet appel provoque
  - le chargement en mémoire de la classe correspondant au pilote
  - l'instanciation d'un objet de cette classe et appel du constructeur qui enregistre la classe auprès du gestionnaire de pilotes
- la chaîne `"com.mysql.cj.jdbc.Driver"` est fournie par le concepteur du pilote

Connexion à la base de données

- pour établir la connexion, il faut préciser :
  - l'**adresse** de la machine qui exécute le serveur de BD : `localhost`
  - le **port** de communication : par défaut `3306` pour MySQL
  - le **nom** de la base de données
  - le **login** et **mot de passe**
  - éventuellement un ensemble de **propriétés** pour que la connexion fonctionne en fonction du serveur, du pilote et du client

Connexion à la base de données

```
1  // connection a la base de donn es
2  Class.forName("com.mysql.cj.jdbc.Driver");
3
4  String uri = "jdbc:mysql://localhost:3306/tweetme?" +
5               "allowPublicKeyRetrieval=true&" +
6               "useSSL=false";
7
8  try(Connection con =
9      DriverManager.getConnection(uri, "root", "root")) {
10 }
```

Interrogation de la BD

- la connexion fournit soit un objet `Connection`
- soit `null` si la connexion n'a pas été établie
- une fois la connexion établie, il est possible d'exécuter des requêtes SQL
- plusieurs classes peuvent être utilisées pour obtenir des informations sur la BD
  - `DatabaseMetaData` : informations à propos de la base de données : nom des tables, index, version
  - `ResultSet` : résultat d'une requête. L'accès se fait enregistrement par enregistrement.
  - `ResultSetMetaData` : informations sur les colonnes (nom et type) d'un ResultSet

Requêtes d'interrogation

- les requêtes d'interrogation SQL sont exécutées avec les méthodes d'un objet de la classe `Statement`
- l'objet `Statement` est obtenu à partir d'un objet de la classe `Connection`
- le résultat d'une requête d'interrogation est renvoyé dans un objet de la classe `ResultSet` par la méthode `executeQuery()`

```
1 try (Statement stmt = con.createStatement();
2 ResultSet rs = stmt.executeQuery("SELECT * FROM users")) {
3 } catch (SQLException e) {}
```

Requêtes de mise à jour

- utilisation de la méthode `executeUpdate()` qui retourne le nombre d'enregistrement mis à jour
- `executeUpdate()` est également utilisée pour exécuter les traitements de type Data Definition Langage (DDL) ou définition de données
  - dans le cas de la création d'une table, la méthode retourne 0

```
1 String query = "INSERT INTO users (`name`, `email`, " +
2     "`password_digest`, `created_at`) VALUES " +
3     "('Heraud Kevin', 'kevin.heraud@worldline.com', " +
4     "SHA2('foo', 256), NOW())";
5 try (Statement stmt = con.createStatement()) {
6     int row = stmt.executeUpdate(query);
7 } catch (SQLException e) {}
```

La classe `ResultSet`

- classe représentant une **abstraction d'une table** qui se compose de plusieurs enregistrements
- les enregistrements sont constitués de colonnes ou champs qui contiennent les données
- principales méthodes
  - `next()` : se déplace sur le prochain enregistrement : retourne `false` si la fin est atteinte
  - `getMetaData()` : retourne un objet `ResultSetMetaData` associé au `ResultSet`
  - `getString(int)` : retourne le contenu de la colonne dont le numéro est passé en paramètre sous forme d'entier.
  - `getString(String)` : retourne le contenu de la colonne dont le nom est passé en paramètre sous forme de chaîne de caractères.
  - `close()` : ferme le `ResultSet`

Quelques "Best Practices"

- try-with-resources
- Utiliser des `prepareStatement`
- Pensez objet ! (DAO - Data Access Object)

La méthode `getMetaData()`

- `getMetaData()` retourne un objet de la classe `ResultSetMetaData`
- permet d'obtenir des informations sur le résultat de la requête
- par exemple le nombre de colonnes peut être obtenu grâce à la méthode `getColumnCount()` et on peut afficher toutes les informations des colonnes ainsi

La méthode `getMetaData()`

```
1  ResultSet rs = stmt.executeQuery("SELECT a, b, c
2    FROM TABLE_T");
3  ResultSetMetaData rsmd = rs.getMetaData();
4  for (int i = 1; i <= rsmd.getColumnCount(); i++){
5   System.out.println(rsmd.getColumnName(i)); // nom colonne
6   System.out.println(rsmd.getColumnType(i)); // type SQL
7   System.out.println(rsmd.isAutoIncrement(i)); // auto-incr. ?
8  }
```

La méthode `next()`

- la méthode `next()` déplace le curseur sur le prochain enregistrement
- le curseur pointe initialement **juste avant** le premier enregistrement
- il est nécessaire de faire un premier appel à la méthode `next()` pour se placer sur le premier enregistrement
- des appels successifs à `next()` permettent de parcourir l'ensemble des enregistrements
- `next()` retourne false lorsqu'il n'y a plus d'enregistrement

Exemple de parcours avec `next()`

```
1  try {
2   ResultSetMetaData rsmd = results.getMetaData();
3   int nbCols = rsmd.getColumnCount();
4   while (results.next()) {
5    // prochain enregistrement / ligne
6    // parcours des colonnes
7    for (int i = 1; i <= nbCols; i++)
8     System.out.print(results.getString(i) + "\t");
9    System.out.println();
10  }
11 } catch (SQLException e) { // traitement de l'exception }
12 finally {if (results != null) results.close();}
```

Utilisation des accesseurs par type

- il existe des méthodes `getString()`, `getDouble()`, `getInteger()` permettent d'extraire les données selon leur type
- il existe deux formes de ces méthodes
  - indiquer le numéro de la colonne en paramètre, **en commençant par 1**
  - indiquer le nom de la colonne en paramètre
- la première méthode peut générer des "erreurs" si la structure de la table évolue

Définition d'une transaction

- une **transaction** est un mécanisme permettant de passer d'un état A à un état B en réalisant une suite d'opérations et en s'assurant que cette suite soit
  - **atomique** : la suite d'opérations est indivisible, en cas d'échec d'une opération, la suite entière est annulée : `rollback`
  - **cohérente** : le contenu de la base de données à la fin de la transaction doit être cohérent
  - **isolée** : les modifications effectuées par une transaction A ne sont ni visibles, ni modifiables par une transaction B, avant la terminaison et validation de A par un `commit`

Utilisation des transactions

- une transaction est gérée à partir de l'objet `Connection`
- par défaut, une connexion est en mode `auto-commit`
  - dans ce mode, chaque opération forme sa propre transaction et est validée par un `commit` après son exécution
- pour pouvoir rassembler plusieurs traitements dans une seule transaction, il faut désactiver le mode `auto-commit`

```
1 connection.setAutoCommit(false);
```

`commit` et `rollback`

- une fois le mode `auto-commit` désactivé,
- l'appel à la méthode `commit()` de la classe `Connection` permet de
  - valider la transaction courante
  - créer implicitement une nouvelle transaction
- si une anomalie survient durant la transaction, il est possible de faire un retour en arrière pour revenir à la situation de la base de données au début de la transaction en appelant la méthode `rollback()` de la classe `Connection`

Illustration de `commit` et `rollback`

```
1  Savepoint sav = con.setSavepoint(); // point de restauration
2  // debut transaction - selection cafe
3  String query = "SELECT COF_NAME, PRICE FROM COFFEES " +
4          "WHERE COF_NAME = '" + coffeeName + "'";
5  Statement q = con.createStatement();
6  ResultSet res = q.executeQuery(query);
7  if (res){
8   res.first();
9   float oldPrice = res.getFloat("PRICE");
10  float newPrice = oldPrice * 2;
11  String update = "UPDATE COFFEES SET PRICE = " + newPrice +
12                  " WHERE COF_NAME = '" +
13                  coffeeName;
14  Statement u = con.createStatement(); // update statement
15  u.executeUpdate(update);
16  if (newPrice > maxPrice){
17   // erreur ! retour en arriere
18   con.rollback(sav);
19  }
```

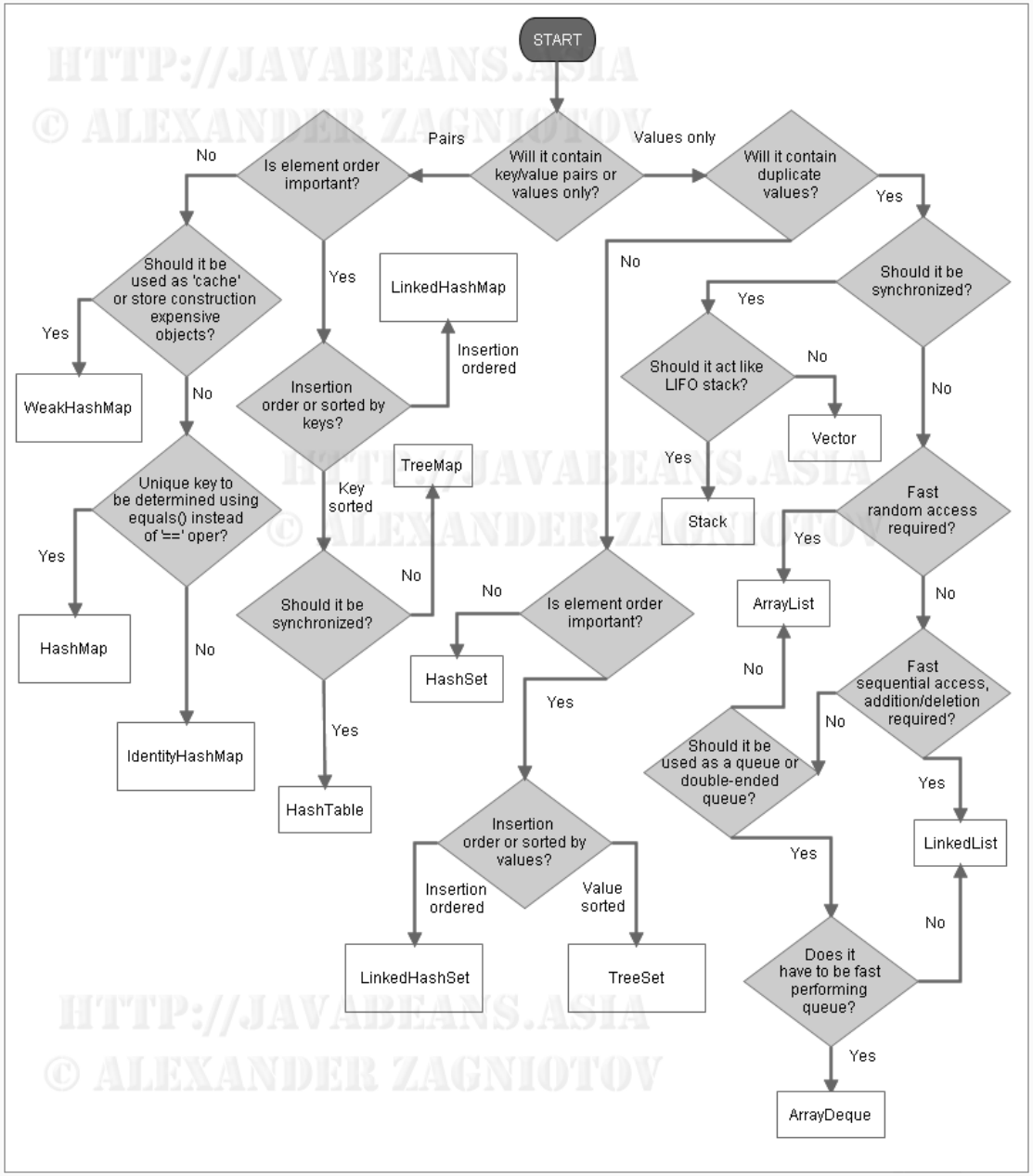
Les collections

- les **collections** en Java permettent de manipuler des structures de données génériques telles que
  - les **vecteurs dynamiques** `ArrayList`
  - les listes chaînées `LinkedList`
  - les ensembles `hashSet` et `TreeSet`
  - les queues avec priorité `PriorityQueue`
  - les queues avec double entrée `ArrayDeque`
- il existe d'autres types pour les structures de données
  - les tableaux `int[]...`
  - les **tables associatives** `HashMap` issue de l'interface `Map`

Critères de choix d'une collection

- critères de choix
  - éléments ordonnées ou non
  - l'accès direct a un élément est possible ou pas
  - l'accès à la valeur se fait en connaissant une clé correspondante
  - les doublons sont-ils possibles ?
- dans ce cours, focus sur les `ArrayList` et les `HashMap`

Critères de choix d'une collection



Principales propriétés des collections

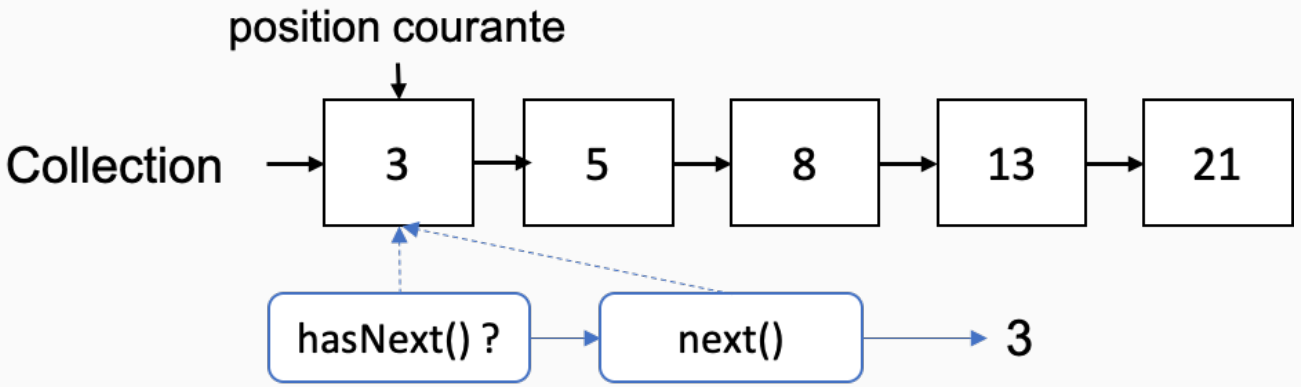
- contenues dans le package `java.util`
- depuis le JDK5, les collections sont **génériques** : `Collection<E>`
- certaines collections sont **ordonnées** : `ArrayList`, `LinkedList` ...
  - possibilité d'accéder au 1er, 2nd, i-ème élément
  - possibilité de les trier selon les valeurs
  - les éléments `E` de la collection implémentent l'interface `Comparable<E>` et recourent à la méthode `compareTo()`

Parcours des éléments d'une collection

- un **itérateur** permet de parcourir un par un les éléments d'une collection
- deux types d'itérateurs
  1. les itérateurs **monodirectionnels** parcourent la collection du début à la fin
  2. les itérateurs **bidirectionnels** parcourent la collection dans les deux sens

Itérateurs monodirectionnels

- accessibles par la méthode `iterator()` qui implémente l'interface `Iterator<E>`
- un itérateur indique une position courante qui désigne un élément de la collection
- la méthode `hasNext()` permet de savoir s'il y a un élément accessible dans la collection
- pour se placer sur et récupérer l'élément courant, il faut appeler la méthode `next()`



Utiliser un itérateur monodirectionnel

- exemple de parcours

```
1 Iterator<E> iter = c.iterator();
2 while (iter.hasNext()){
3  E elem = iter.next();
4  // utilisation de elem ...
5 }
```

Utiliser un itérateur monodirectionnel

- exemple de suppression du dernier élément retourné

```
1 Iterator<E> iter = c.iterator();
2 while (iter.hasNext()){
3  E elem = iter.next(); // avance d'une position et retourne
4                        // l'element courant
5  if (condition) elem.remove();
6 }
```

Supprimer le 1er élément d'une collection

1. récupérer l'itérateur de la collection
2. récupérer l'objet courant et se positionner sur la position suivante
3. supprimer l'objet courant

```
1 Iterator<E> iter = c.iterator();
2 E elem = iter.next(); // recuperation du 1er objet
3 iter.remove(); // suppression du 1er objet
```

- `remove` ne supprime pas l'élément à la position courante mais l'objet courant qui a été accédé précédemment
- il est nécessaire d'avoir un objet courant pour appeler la méthode `remove`

Parcours unidirectionnel avec for ... each

- il est aussi possible de parcourir une collection avec une boucle `for ... each`

```
1 for (E elem : c){
2 }
```

Les vecteurs dynamiques `ArrayList`

- structure ordonnée, à accès direct par indice, qui autorise les doublons
- comparable à un tableau :
  - offre un accès rapide aux éléments
  - les éléments sont contigus en mémoire
- mais le nombre d'éléments peut varier au cours de l'exécution
- accepte n'importe quels types d'objets, y compris la valeur `null`
- dérive de l'interface `List<E>`

Opérations usuelles : ajout

- **construction** d'un vecteur vide ou bien à partir d'une autre collection existante `c`

```
1 List<E> v1 = new ArrayList<E>();
2 List<E> v2 = new ArrayList<E>(c);
```

- **ajout** d'un ou de plusieurs éléments, à une certaine position

```
1 v1.add(e1); // ajout de e1 a la fin
2 v1.add(3, e2); // ajout de e2 en 4eme position
3 v2.addAll(c); // ajout de tous les elements de c a la fin
4 v2.addAll(2, c);; // ajout des elements de c a l'indice 2
```

Opérations usuelles : suppression

- **suppression** d'un ou plusieurs éléments avec décalage de l'indice de tous les éléments suivants

```
1 E elem = v1.remove(3); // ici elem = e2
2 v2.removeRange(1, 4);
```

Opérations usuelles : parcours et accesseurs

- **taille** du vecteur accessible par la méthode `size()`
- **accès** aux valeurs du vecteur avec la méthode

  `get(int index)`

Autres méthodes de la classe `ArrayList`

- `isEmpty()` : renvoie vrai si la liste est vide
- `indexOf(objet)` : retourne l'indice i de l'objet en paramètre ou -1 s'i l'objet n'est pas dans la liste
- `contains(objet)` : retourne vrai si l'objet est dans la liste
- `set(i, objet)` : remplace l'élément situé en i par l'objet en paramètre
- `clear()` : supprime tous les objets de la liste
- `isEmpty()` : renvoie vrai si la liste est vide
- `contains(objet)` retourne vrai si l'objet est dans la liste

Exemple d'utilisation

```
1 List<Point> points = new ArrayList<Point>();
2 points.add(p1);
3 points.add(p2);
4 points.add(p3);
5
6 // ou
7 List<Point> points = Arrays.asList(p1, p2, p3);
8
9 for (int i=0;i<points.size();i++) points.get(i).affiche();
```

Autre notation utilisant un `for ... each`

```
1 for (Point point : points) point.affiche();
```

Vecteurs dynamiques et héritage

- on suppose avoir les classes `Cercle`, `Triangle` et `Parallelogramme` qui héritent de la classe abstraite `Forme`
- la classe `Forme` définit la méthode abstraite `affiche`

```
1  Cercle c1 = new Cercle(p1,12.6); // centre p1, rayon 12.6
2  Triangle t1 = new Triangle(p2,v1,v2); // 1 point et 2 vecteurs
3  Parallelogramme pa1 = new Parallelogramme(p3,v1,v3);
4
5  List<Forme> formes = new ArrayList<Forme>();
6  formes.add(pa1);  // un parallelogramme
7  formes.add(c1); // un cercle
8  formes.add(t1);  // un triangle
9
10 // ou
11 List<String> formes = Arrays.asList(c1, t1, pa1);
12
13 for (Forme forme: formes) forme.affiche();
```

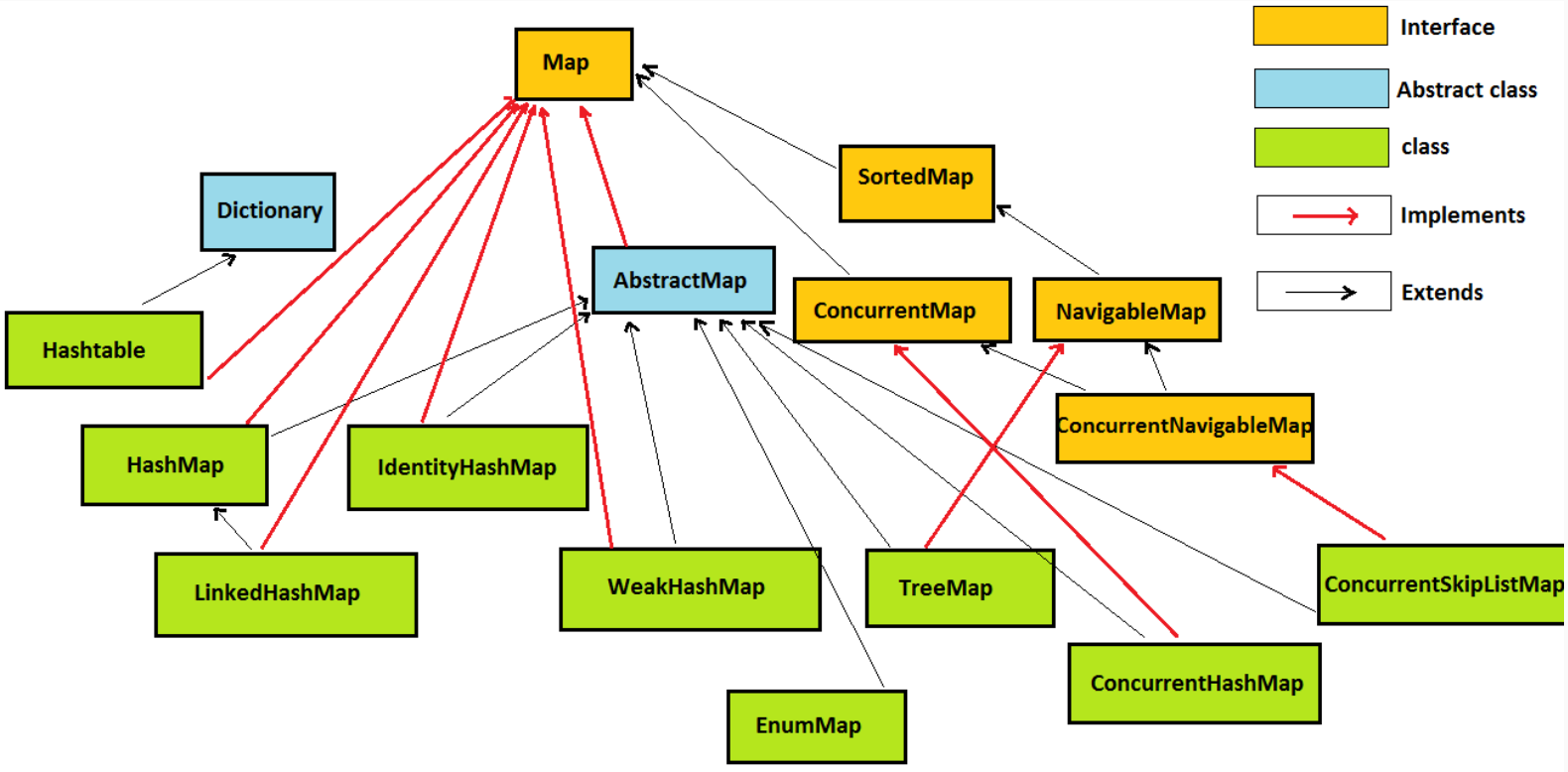
ArrayList personnalisée

```
1  public class EnvelopeArrayList<Page> extends ArrayList<Page> {
2      private final int defaultMaxPages;
3
4      public EnvelopeArrayList(int defaultMaxPages) {
5          this.defaultMaxPages = defaultMaxPages;
6      }
7
8      @Override
9      public boolean add(Page page) {
10         return super.size() < defaultMaxPages ?
11             super.add(page) : false;
12     }
13 }
14
15 List<Page> pages = new EnvelopeArrayList(5)<>;
```

Map

- `Map` est une interface représentant une table associative (<K,V>, "Key"/"Value")

Map : la hiérarchie



HashMap

- `HashMap<K,V>` est une classe implémentant l'interface `Map<K,V>`
- les clés ne peuvent pas être dupliquées
- une clé peut être `null`
- les valeurs `null` sont acceptées
- l'ordre des clés n'est pas maintenu
- fonctionne sur la base de la technique de hachage (bucketIndex = key.hashCode() % bucketSize)

HashMap : quelques méthodes

- `put(K, V)` : Insère l'entrée (K, V). Si la clé est déjà présente, la nouvelle valeur remplace l'ancienne
- `putIfAbsent(K, V)` : Insère l'entrée (K, V) si la clé K n'est pas déjà associée à la valeur V
- `get(K)` : Retourne la valeur associée à la clé K spécifiée. Si la clé n'est pas trouvée, elle retourne null
- `containsKey(K)` : Vérifie si la clé spécifiée K est présente dans le Map ou non
- `containsValue(V)` : Vérifie si la valeur spécifiée V est présente dans le Map ou non
- `remove(K)` : Supprime l'entrée du Map représentée par la clé K
- `keySet()` : Retourne l'ensemble de toutes les clés présentes dans une Map
- `values()` : Retourne un ensemble de toutes les valeurs

HashMap

```
1  Map<String, String> phonebook = new HashMap<>();
2
3  phonebook.put("01000005", "Tom");
4  phonebook.put("01000006", "Jerry");
5  phonebook.put("01000003", "Tom");
6  phonebook.put("01000004", "Donald");
7
8  // ou
9  Map<String, String> phonebook = Map.of("01000004", "Donald");
10
11 for (Map.Entry entry : phonebook.entrySet()) {
12 System.out.println(entry.getKey() + "|" + entry.getValue());
13 }
```

Les types énumérés

- le mot-clé `enum` permet de déclarer un type énuméré
- type de données comportant en ensemble fini de constantes
- les constantes seront séparées par une virgule et en MAJUSCULES (+ snake case)
- utilisées lorsque nous connaissons toutes les valeurs possibles au moment de la compilation
- Très utile dans les `switch`

Les types énumérés : les méthodes associées

- une énumération peux contenir un constructeur privé
- `values()` : renvoie toutes les valeurs présentes dans l'énumération
- `valueOf()` : renvoie la constante d'énumération de la valeur de chaîne spécifiée, si elle existe

Les types énumérés : exemple

```
1  public enum CurrencyEnum {
2  EUR("euro", "E", 978),
3  USD("dollar des Etats-Unis", "S", 840);
4
5  private final String displayName;
6  private final String symbol;
7  private final int code;
8
9  CurrencyEnum(String displayName, String symbol, int code) {
10     this.displayName = displayName;
11     this.symbol = symbol;
12     this.code = code;
13 }
14
15 public String getDisplayName() { return displayName; }
16 public String getSymbol() { return symbol; }
17 public int getNumericCode() { return numericCode; }
18 }
```

Les types énumérés : le pattern Singleton

```
1 public enum EasySingleton{
2    INSTANCE;
3 }
```