2ème Ingéniorat en Informatique

Programmation Orientée Objet II

Chapitre II
Concepts avancés de la POO

2023/2024

Dr. HANNOUSSE Abdelhakim – hannousse.abdelhakim@univ-guelma.dz

Contenu du chapitre

- Exceptions
  - Principe et définition
  - Gestion des exceptions en Java
  - Hiérarchie des exceptions
  - Les exceptions personnalisées
- Flux d'entrés/sorties
- Structures de données complexes
  - Listes
  - Maps et Ensembles
  - Graphes
- Réflexivité en Java
- Généricité en Java et en C++

Les exceptions

Qu’est-ce qu’une exception?

- Une exception est une condition ou une situation anormale qui survient lors de l'exécution d'un programme et qui peut perturber son déroulement normal.
- Les exceptions sont souvent causées par des erreurs ou des événements inattendus qui se produisent pendant l'exécution du programme:
  - Opérations de calculs invalides
  - Conditions de dépassement de mémoire
  - Accès a des ressources inexistantes, etc.
- La gestion des exceptions permet aux programmeurs de détecter et de réagir de manière appropriée à ces situations exceptionnelles, en leur permettant de prendre des mesures correctives ou de signaler les erreurs de manière appropriée.
- Les exceptions peuvent également être utilisées pour garantir que les ressources sont correctement nettoyées même en cas d'erreur.

La gestion des exceptions en Java (1/2)

- La gestion des exceptions, en Java, se divise généralement en deux parties :
  - **Capture d'exception :** Le code qui peut potentiellement générer une exception est entouré d'un bloc **try-catch**. Si une exception est levée durant l'exécution par la machine virtuelle, le bloc catch correspondant est exécuté pour gérer l'exception.

```
4  try {
5      val = val1 / val2;
6  } catch (ArithmeticException e) {
7      System.out.println("La valeur de val2 est nule se qui cause la division par zéro.");
8  }
```

  - **Levé d'exception** : Java fournit un mot-clé **throw** à l'aide duquel nous pouvons explicitement lever les exceptions dans le code.

```
4  float moyenne = etudiant.getMoyenne();
5  if (moyenne <0 || moyenne >20)
6        throw new ArithmeticException("Valeur incorrect du moyenne");
```

La gestion des exceptions en Java (2/2)

- Il est possible de capturer plusieurs exceptions différentes dans un même bloc **catch**. Cela permet de traiter les différentes exceptions qui peuvent être levées pendant l'exécution du même bloc d'instructions.

```
try {
    tableau[x] = tableau[x] / tableau[x + 1];
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Index hors limites.");
} catch(ArithmeticException e) {
    System.out.println("Division par zéro !!");
}
```

Bloc **try-catch-finally**

- Java permet l'utilisation d'un bloc **finally**. Les instructions à l'intérieur de ce bloc sont exécutées qu'une exception soit levée ou non. Cela est souvent utilisé pour libérer des ressources, même en cas d'erreur. Le bloc **finally** doit être spécifié à la fin du bloc **try-catch**.

```java
try {
    // Ouverture du fichier
    BufferedReader fichier = new BufferedReader(new FileReader("..."));
    try {
        // Lecture et traitement des données
        String ligne;
        while ((ligne = fichier.readLine()) != null) {
            // Un traitement quelconque
        }
    } catch (IOException e) {
        // Capturer une exception liée à la lecture ou au traitement des données
        System.err.println("Erreur de lecture/traitement : " + e.getMessage());
    } finally {
        // Fermeture du fichier dans le bloc finally pour s'assurer qu'il est toujours fermé
        try {
            fichier.close();
        } catch (IOException e) {
            // Capturer une exception liée à la fermeture du fichier
            System.err.println("Erreur lors de la fermeture du fichier : " + e.getMessage());
        }
    }
} catch (IOException e) {
    // Capturer une exception liée à l'ouverture du fichier
    System.err.println("Erreur d'ouverture du fichier : " + e.getMessage());
}
```

La clause **throws**

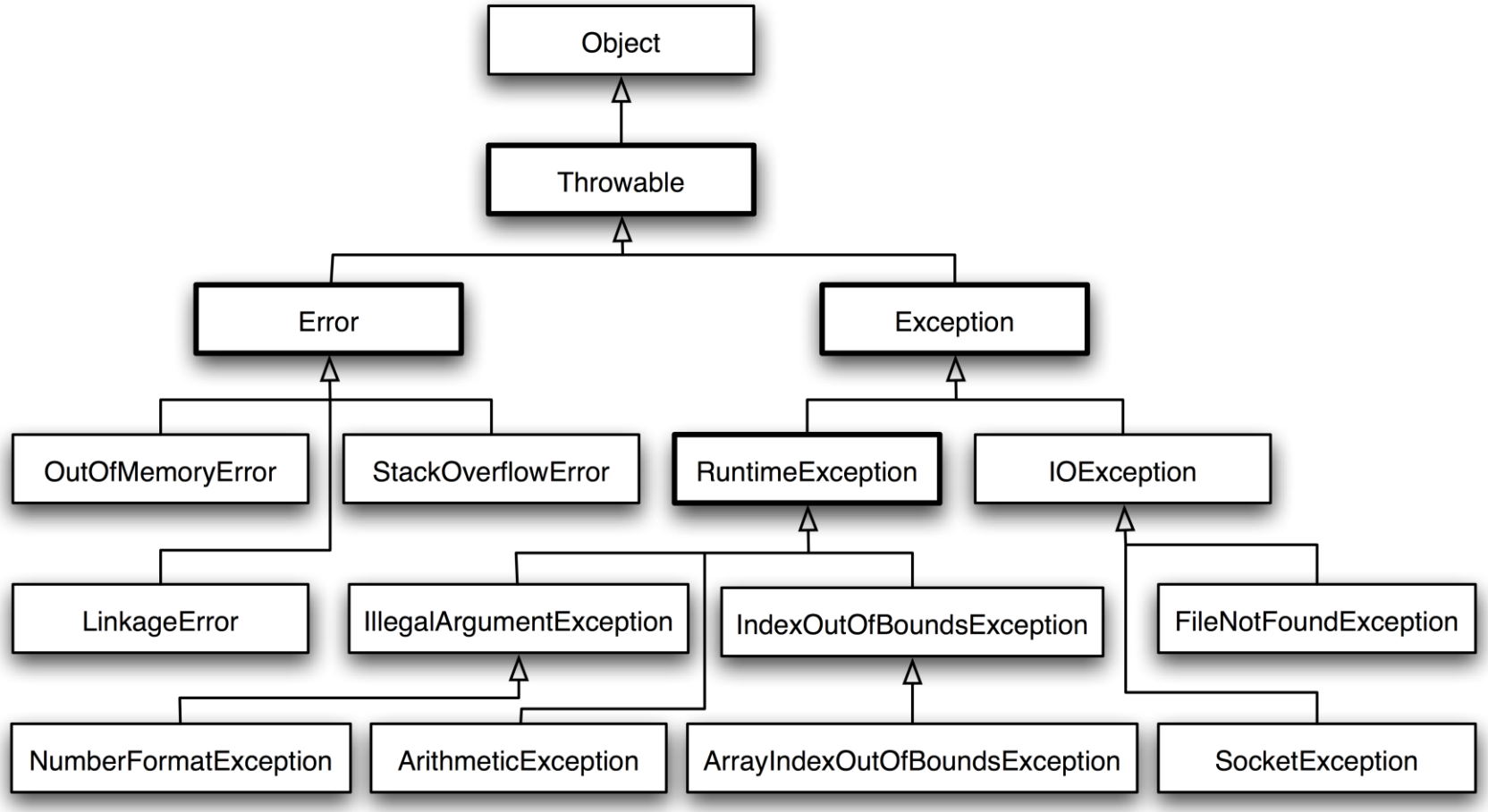
- L'utilisation de la clause **throws** offre une alternative à l'utilisation de blocs **try-catch** dans une méthode dans deux situations principales :
  - Gestion d'un grand nombre d'exceptions afin d'améliorer la lisibilité du code.
  - Absence de gestionnaire pour une exception déclarée, la clause **throws**, dans ce cas, déplace la responsabilité de la gestion de l'exception vers le code appelant.

```
3  public void lireFichier(String cheminFichier) throws IOException {
4      // Ouverture du fichier
5      BufferedReader lecteur = new BufferedReader(new FileReader(cheminFichier));
6      // Lecture et traitement des données
7      String ligne;
8      while ((ligne = lecteur.readLine()) != null) {
9          // Un traitement quelconque
10     }
11     lecteur.close();
12 }
```

Hiérarchie des Exceptions (1/2)

- Les langages de programmation fournissent souvent une hiérarchie d'exceptions. Certaines exceptions sont des sous-types d'autres, permettant une gestion plus spécifique des erreurs.
- Les exceptions Java sont organisées dans une hiérarchie de classes, avec la classe de base étant **java.lang.Throwable**. Cette classe a deux sous-classes principales :
  - **Error :** Représente des erreurs graves qui ne devraient généralement pas être récupérées par le programme.
    - Exemple: **OutOfMemoryError**
  - **Exception :** Représente des erreurs qui peuvent être gérées et récupérées. La plupart des exceptions rencontrées dans la programmation quotidienne héritent de cette classe.
    - Exemple: **FileNotFoundException**

Hiérarchie des Exceptions (1/2)



Les exceptions personnalisées

- Il est possible de définir des exceptions personnalisées dans le programme pour:
  - Signaler des erreurs ou des conditions exceptionnelles qui ne sont pas couvertes par les exceptions standards fournies par le langage.
  - Fournir des informations plus spécifiques sur l'erreur ou la condition exceptionnelle qui s'est produite.
  - Organiser les différentes erreurs de manière logique en créant une hiérarchie d'exceptions personnalisées.

Les exceptions personnalisées – Exemple

```java
import java.util.HashMap;
import java.util.Map;

class EmpruntDejaEnCoursException extends Exception {
    public EmpruntDejaEnCoursException(String message, String titre) {
        super(message);
        Bibliothèque.empruntsEchouesParTitre.put(titre, empruntsEchouesParTitre.get() + 1);
    }
}
class Livre {
    private boolean estEmprunte = false;
    private String titre;

    public Livre(String titre) {
        this.titre = titre;
    }
    public void emprunter() throws EmpruntDejaEnCoursException {
        if (estEmprunte) {
            throw new EmpruntDejaEnCoursException("Le livre est déjà emprunté.", titre);
        } else {
            estEmprunte = true;
            System.out.println("Le livre '" + titre + "' a été emprunté avec succès.");
        }
    }
}
class Bibliothèque {
    private static Map<String, Integer> empruntsEchouesParTitre = new HashMap<>();
    ...
}
```

Les flux d’entrés/sorties

Flux d’écran, Flux de clavier et Flux de fichiers

Les notion de flux (1/2)

- En programmation, un flux (**stream** en anglais) fait référence à la manière dont les données sont transférées entre un programme et un périphérique d'entrée/sortie.
- Un flux est en quelque sorte un canal dans lequel de l'information transite. L'ordre dans lequel l'information y est transmise est respecté.
- Les flux facilitent la communication entre un programme et l'environnement externe.
- Il existe deux types de flux:
  - **Flux d'entrée :** utilisé pour récupérer des données depuis une source externe vers le programme.
    - Lecture d'un fichier, saisie au clavier ou réception de données d'un réseau.
  - **Flux de sortie :** utilisé pour envoyer des données depuis le programme vers une destination externe.
    - Écrire dans un fichier, afficher des résultats à l'écran ou envoyer des données sur un réseau.

Les notion de flux (2/2)

- Les flux peuvent aussi être caractérisés par leur nature, tels que les flux binaires (pour les données non textuelles) et les flux de texte (pour les données textuelles).
- En programmation, les langages fournissent souvent des classes ou des bibliothèques pour travailler avec des flux.
  - En Java, les classes **InputStream** et **OutputStream** sont utilisées pour la manipulation de flux.
  - En C++, les objets **ifstream** et **ofstream** sont utilisés pour les opérations d'entrée et de sortie de fichiers.
- Les opérations sur les flux incluent:
  - La lecture et l'écriture de données
  - Le déplacement dans le flux (positionnement)
  - La fermeture du flux après utilisation
  - La gestion des erreurs liées aux opérations précédentes.

Flux d’écran

- Les flux d'écran sont utilisés pour afficher des données à la console.
- Les classes **System.out** et **System.err** sont utilisées pour la sortie standard et la sortie d'erreur, respectivement.
- Les deux sorties sont affichées sur la même console. La plupart des IDE différencient la sortie d'erreur par la couleur rouge.
- Nous pouvons reconfigurer les flux pour que, par exemple, **System.out** imprime toujours sur la console mais **System.err** écrit dans un fichier.

```java
import java.io.*;

public class RedirectionFlux {
    public static void main(String[] args) {
        // Sauvegarde des flux de sortie standard
        PrintStream originalOut = System.out;
        PrintStream originalErr = System.err;

        try {
            // Redirection de System.out vers un fichier
            PrintStream fileOut = new PrintStream(new FileOutputStream("sortie.txt"));
            System.setOut(fileOut);

            // Redirection de System.err vers un autre fichier
            PrintStream fileErr = new PrintStream(new FileOutputStream("erreur.txt"));
            System.setErr(fileErr);

            // Messages envoyés aux flux redirigés
            System.out.println("Ceci est un message pour System.out (dans sortie.txt).");
            System.err.println("Ceci est un message pour System.err (dans erreur.txt).");

            // Restauration des flux d'origine
            System.setOut(originalOut);
            System.setErr(originalErr);

            // Messages envoyés aux flux d'origine
            System.out.println("Ceci est à nouveau System.out (affiché à la console).");
            System.err.println("Ceci est à nouveau System.err (affiché à la console).");
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

Flux de clavier (1/3)

- Les flux de clavier sont utilisés pour lire des données depuis le clavier.
- Une approche courante consiste à utiliser la classe **BufferedReader** en association avec **InputStreamReader** pour lire depuis **System.in.**

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class LectureClavierBufferedReader {
    public static void main(String[] args) {
        // Création d'un objet BufferedReader pour lire depuis le clavier
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));

        try {
            // Demande à l'utilisateur d'entrer son nom
            System.out.print("Veuillez entrer votre nom : ");
            String nomUtilisateur = reader.readLine();

            // Demande à l'utilisateur d'entrer son âge
            System.out.print("Veuillez entrer votre âge : ");
            int ageUtilisateur = Integer.parseInt(reader.readLine());

            // Affichage des informations saisies
            System.out.println("Nom : " + nomUtilisateur);
            System.out.println("Âge : " + ageUtilisateur);
        } catch (IOException | NumberFormatException e) {
            System.err.println("Erreur de lecture : " + e.getMessage());
        }
}
```

Flux de clavier (2/3)

- La classe **Scanner** est souvent utilisée pour lire les entrées du clavier.

```java
import java.util.Scanner;

class Main {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    System.out.println("Enter name, age and salary:");

    // String input
    String name = input.nextLine();

    // Numerical input
    int age = input.nextInt();
    double salary = input.nextDouble();

    // Output input by user
    System.out.println("Name: " + name);
    System.out.println("Age: " + age);
    System.out.println("Salary: " + salary);
  }
}
```

Flux de clavier (3/3) – Les méthodes de la classe Scanner

Méthode	Description
byte nextByte()	Renvoie un byte lu au clavier
double nextDouble()	Renvoie un double lu au clavier
float nextFloat()	Renvoie un float lu au clavier
int nextInt()	Renvoie un int lu au clavier
long nextLong()	Renvoie un long lu au clavier
short nextShort()	Renvoie un short lu au clavier
boolean nextBoolean()	Renvoie un boolean lu au clavier
String next()	Renvoie le prochain jeton (mot) de la séquence d'entrée
String nextLine()	Renvoie une chaîne de caractère lu au clavier
boolean hasNextInt()	Vérifie si la prochaine donnée dans l'entrée peut être interprétée comme un int
boolean hasNextLine()	Vérifie s'il y a une autre ligne dans l'entrée de ce scanner ou non

Gestion de Fichiers (1/2)

- La gestion de fichiers en Java se fait par l'intermédiaire de la classe **File.**
- Cette classe possède des méthodes qui permettent d'interroger ou d'agir sur le système de gestion de fichiers du système d'exploitation.
- Un objet de la classe **File** peut représenter un fichier ou un répertoire.

```
1  import java.io.File;
2
3  public class Listeur {
4      public static void main(String[] args) {
5          litrep(new File("."));
6      }
7      public static void litrep(File rep) {
8          if (rep.isDirectory()) {
9              //liste les fichier du répertoire
10             String t[] = rep.list();
11             for (int i = 0; i < t.length; i++) System.out.println(t[i]);
12         }
13     }
14 }
```

Gestion de fichiers (2/2)

Constructeur/Méthode	**Description**
File(String chemin)	Crée un objet File en utilisant le chemin d'accès spécifié sous forme de chaîne de caractères.
File(String chemin, String nomFichier)	Crée un objet File représentant un fichier dont le chemin est spécifié par le répertoire parent (chemin) et le nom de fichier.
File(File parent, String nomFichier)	Similaire au précédent, mais au lieu de spécifier le chemin d'accès du répertoire parent sous forme de chaîne de caractères, il utilise un objet File représentant le répertoire parent.
boolean isFile( )	Teste si l'objet représente un fichier normal.
boolean isDirectory( )	Teste si l'objet représente un répertoire.
boolean mkdir( )	Créer le répertoire portant le nom de ce chemin d'accès abstrait.
boolean exists( )	Tester si le fichier ou le répertoire existe.
boolean delete( )	Supprimer le fichier ou le répertoire.
boolean canWrite( )/boolean canRead( )	Tester si l'application peut modifier/lire le fichier ou le répertoire.
File getParentFile( )	Renvoie l'objet représentant le parant de l'objet courant.
long lastModified( )	Renvoie l'heure à laquelle le fichier ou le répertoire a été modifié.

Flux de fichiers (1/2)

- Les classes **FileInputStream** et **FileOutputStream** sont utilisées pour les opérations sur des fichiers binaires.

```java
import java.io.FileInputStream;
import java.io.IOException;

public class LectureEcritureFichierBinaire {
    public static void main(String[] args) {
        // Lecture depuis un fichier binaire
        try {
            FileInputStream fis = new FileInputStream("fichier.bin");
            int octet;
            while ((octet = fis.read()) != -1) {
                System.out.print((char) octet);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Écriture dans un fichier binaire
        try {
            FileOutputStream fos = new FileOutputStream("fichier.bin");
            // Données à écrire dans le fichier binaire
            // Correspond à "ABCDE" en ASCII
            byte[] donnees = { 65, 66, 67, 68, 69 };
            fos.write(donnees);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Flux de fichiers (2/2)

- Les classes **FileReader** et **FileWriter** sont utilisées pour les opérations sur des fichiers texte.

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class LectureEcritureFichierTexte {
    public static void main(String[] args) {
        // Lecture depuis un fichier texte
        try {
            BufferedReader reader = new BufferedReader(new FileReader("fichier.txt"));
            String ligne;
            while ((ligne = reader.readLine()) != null) {
                System.out.println(ligne);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        // Écriture dans un fichier texte
        try {
            BufferedWriter writer = new BufferedWriter(new FileWriter("fichier.txt"));
            String donnees = "Ceci est une ligne de texte.";
            writer.write(donnees);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Structures de données complexes

Listes, Ensembles, Maps et Graphes

Types des structures de données complexes

- Java fournit différentes structures de données complexes :
  - Les listes:
    - ArrayList
    - LinkedList
  - Les Maps
    - HashMap
    - TreeMap
  - Les ensembles
    - HashSet
    - TreeSet
  - Les graphes

Tableaux dynamiques (1/2)

- **ArrayList (java.util):**
  - Utilise un tableau dynamique pour stocker les éléments.
  - Lorsqu'une ArrayList doit augmenter sa capacité, elle le fait par un facteur pour réduire les opérations coûteuses de redimensionnement. Par défaut, l'ArrayList augmente sa capacité de 50%.
  - L'accès aux éléments par leur index est rapide, mais l'insertion ou la suppression d'éléments au milieu peut être coûteuse car cela nécessite le déplacement des éléments suivants.
  - Performant pour l'accès aléatoire et la lecture séquentielle. Les opérations d'ajout ou de suppression fréquentes au milieu peuvent être moins efficaces.

Tableaux dynamiques (2/2)

- **Constructeurs et méthodes:**
  - **new** ArrayList<>(), **new** ArrayList<>(int), **new** ArrayList<>(Collection): Créer un tableau vide de taille 10, un tableau avec une taille initiale, un tableau avec des éléments prédéfinies.
  - *add*(e), *add*(*i*, e): ajouter l'élément e a la fin ou a la position i.
  - *remove*(e), *remove*(*i*): supprimer l'élément e ou l'élément a la position i.
  - *get*(*i*): renvoie l'élément de la position i.
  - *contains*(e): renvoie **true** si l'élément e existe.
  - *clear*(): vider le tableau.
  - *size*(): renvoie le nombre des éléments dans le tableau.

Les listes chainées (1/2)

- **LinkedList (java.util):**
  - Une liste doublement chaînées où chaque élément est lié à son élément précédent et suivant.
  - Pas de liste simplement chainées (par défaut) en Java.
  - L'insertion et la suppression d'éléments sont plus efficaces que dans les ArrayList.
  - sont performantes pour les opérations d'insertion et de suppression, notamment lorsqu'il s'agit d'opérations en début ou en fin de liste.
  - Cependant, l'accès direct aux éléments (par indice) est moins efficace que dans les ArrayList car il nécessite de parcourir la liste depuis le début ou la fin jusqu'à l'élément souhaité.

Les listes chainées (2/2)

- **Constructeurs et méthodes:**
  - **new** LinkedList(), **new** LinkedList<>(Collection): Créer une liste vide, une liste avec des éléments prédéfinies.
  - *addFirst*(e), *add*(e), *add*(*index*, e): ajouter un élément e au début, a la fin, ou a la position index (si index > taille, une exception IndexOutOfBound est générée).
  - *remove*(e), *remove*(*i*), *removeFirst*(), *removeLast*(), *clear*(): supprimer l'élément e, l'élément a la position i, le premier ou dernier élément, tous les éléments
  - *getFirst*(), *getLast*(), *get*(*i*), *indexOf*(e): renvoie le premier élément, le dernier élément, l'élément de la position i, l'index de l' élément e.
  - *contains*(e): renvoie **true** si l'élément e existe.
  - *size*(): renvoie le nombre des éléments dans le liste.
  - *toArray*(): renvoie un tableau contenant tous les éléments de la liste.

Les Maps (1/3)

- **HashMap (java.util):**
  - Stockage des éléments dan une table de hachage.
  - La capacité de la table de hachage est déterminée lors de la création du HashMap et peut être ajustée dynamiquement.
  - Un facteur de charge est utilisé pour déterminer quand la table de hachage doit être redimensionnée (rehashée) pour maintenir un équilibre entre les performances et l'utilisation de la mémoire.
  - Les éléments dans un HashMap ne sont pas triés selon l'ordre naturel des clés.
  - Les clés dans un HashMap doivent être uniques. L'ajout d'une paire clé-valeur avec une clé existante implique le remplacement de la valeur associée à cette clé.

Les Maps (2/3)

- HashMap n'est pas synchronisée, ce qui signifie qu'elle n'est pas thread-safe. Si plusieurs threads accèdent et modifient une HashMap en même temps, des problèmes de concurrence peuvent se produire. Si la synchronisation est requise, il faut utiliser ***Collections.synchronizedMap(new HashMap(…))*** pour créer une version synchronisée.
- Permet un accès rapide aux éléments en fonction de leur clé. La complexité moyenne pour les opérations de recherche, d'insertion et de suppression est de O(1).

Les Maps (3/3)

- **Constructeurs et méthodes:**
  - ***new*** *HashMap<key, valeur>()*: Créer une map vide (taille 16, facteur 0.75) en spécifiant le type de clés et valeurs.
  - ***new*** *HashMap<key, valeur>(int, float)*: Créer une map avec une capacité et facteur de chargement spécifiques.
  - *put(key, value)* : insérer une entrée.
  - *putAll(map)*: insérer la Map spécifiée dans une Map existant.
  - *get(key)*: renvoie la valeur de la clé spécifiée.
  - *containsKey(key)*: rechercher la clé spécifiée dans une Map.
  - *keySet()*: renvoie un ensemble contenant toutes les clés.
  - *entrySet()*: renvoie un ensemble contenant toutes les clés et valeurs (***Entry***).

```
1  import java.util.*;
2  public class Main {
3      public static void main(String args[]){
4        Map<Integer, String> map = new HashMap<Integer, String>();
5        map.put(10, "Java");
6        map.put(11, "PHP");
7        map.put(12, "Python");
8        map.put(13, "C++");
9
10       //Parcourir les éléments du Map
11       for(Map.Entry mp:map.entrySet()){
12         System.out.println(mp.getKey()+" "+mp.getValue());
13       }
14     }
15 }
```

HashMap vs HashTable

HashMap	HashTable
HashMap autorise une clé null et plusieurs valeurs null.	HashTable ne permet pas les clés ou les valeurs null. Si vous essayez d'ajouter une clé ou une valeur null à une HashTable, une NullPointerException sera levée.
HashMap n'est pas synchronisée par défaut.	HashTable est synchronisée.
HashMap permet d'avoir de meilleures performances dans un environnement à un seul thread.	HashTable, étant synchronisée, est plus lente dans un environnement à un seul thread.
HashMap est une implémentation plus récente et plus flexible introduite dans les versions ultérieures de Java. Elle est largement préférée à HashTable en raison de sa performance et de sa flexibilité accrues.	HashTable est une classe héritée de la version originale de Java (legacy class) et est obsolète depuis l'introduction de HashMap. Elle est maintenue pour des raisons de compatibilité avec les anciennes versions de Java.

Les ensembles – HashSet (1/2)

- **HashSet (java.util):**
  - Les éléments sont stockes à l'aide d'un mécanisme de hachage.
  - Elle utilise un HashMap pour stocker les éléments → Utilise le patron adaptateur pour adapter une HashMap afin de gérer uniquement des clés sans valeurs.
  - Ne contient que des éléments uniques.
  - Autorise une seule valeur nulle.
  - La classe HashSet n'est pas synchronisée par defaut. Il faut utiliser ***Collections.synchronizedSet(new HashSet(...))*** pour assurer sa synchronisation.
  - Les opérations d'ajout, de suppression et de recherche ont une complexité en temps proche de O(1) en moyenne.
  - Utilisé lorsque l'ordre des éléments n'est pas important, et la priorité est la rapidité des opérations.

Les ensembles - HashSet (2/2)

- **Constructeurs et méthodes:**
  - ***new*** *HashSet<Type>()*: Créer un ensemble vide en spécifiant le type de ses valeurs.
  - *add(e)* : ajouter l'élément e.
  - *remove(e)*: supprimer l'élément e.
  - *clear()*: supprimer tous les éléments.
  - *contains(e)*: vérifier si l'élément e existe.
  - *isEmpty()*: vérifier si l'ensemble est vide.
  - *size()*: renvoie le nombre des éléments dans l'ensemble.

```java
import java.util.*;
public class Main {
    public static void main(String args[]){
      HashSet<String> hs = new HashSet<String>();

        hs.add("Python");
        hs.add("PHP");
        hs.add("Java");
        hs.add("C++");

        System.out.println("Contenu du HashSet : " + hs);
   }
}
```

```
java -cp /tmp/nVrnkKbGwv Main
Contenu du HashSet : [Java, C++, PHP, Python]
```

Les ensembles – TreeSet (1/2)

- **TreeSet (java.util):**
  - Stocke les éléments dans un arbre binaire de recherche équilibré (AVL).
  - Les éléments sont triés dans l'ordre naturel des éléments, ou selon un ***Comparateur*** fourni lors de la création de l'ensemble. Cela signifie que les éléments sont stockés dans un ordre déterminé et peuvent être itérés dans cet ordre.
  - TreeSet n'est pas synchronisé par défaut, ce qui signifie qu'il n'est pas thread-safe. Il faut donc utiliser ***Collections.synchronizedSortedSet(new TreeSet(…))*** dans un environnement multithreadé.
  - TreeSet ne permet pas d'éléments en double.
  - TreeSet n'accepte pas de valeurs null. Tenter d'ajouter null à un TreeSet entraînera une NullPointerException.
  - Les opérations ont une complexité en temps logarithmique (O(log n)), car les éléments sont triés.

Les ensembles - TreeSet (2/2)

- **Constructeurs et méthodes:**
  - *TreeSet()* : crée un ensemble vide trié selon l'ordre naturel des éléments.
  - *TreeSet(Comparator)* : crée un ensemble vide trié selon l'ordre défini par le comparateur spécifié.
  - *add(e)* : ajoute l'élément spécifié à l'ensemble.
  - *remove(e)* : supprime l'élément spécifié de l'ensemble.
  - *size()* : renvoie le nombre d'éléments.
  - *clear()* : supprime tous les éléments de l'ensemble.
  - *isEmpty()* : vérifier si l'ensemble es
  - *contains(e)* : vérifier si e existe dans l'ensemble.
  - *first()* : renvoie le premier élément de l'ensemble.
  - *last()* : renvoie le dernier élément de l'ensemble.
  - *headSet(e)* : renvoie la partie de l'ensemble dont les éléments sont strictement inférieurs à e.
  - *tailSet(e)* : renvoie la partie de l'ensemble dont les éléments sont supérieurs ou égaux à e.
  - *subSet(e1, e2)* : renvoie la partie de l'ensemble dont les éléments sont compris entre *e1* et *e2*.
  - *ceiling(e)* : renvoie le plus petit élément dans l'ensemble qui est supérieur ou égal e.
  - *floor(e)* : renvoie le plus grand élément dans l'ensemble qui est inférieur ou égal e.
  - *higher(e)* : renvoie le plus petit élément dans l'ensemble qui est strictement supérieur e.
  - *lower(e)* : renvoie le plus grand élément dans l'ensemble qui est strictement inférieur à e.

Main.java

Run

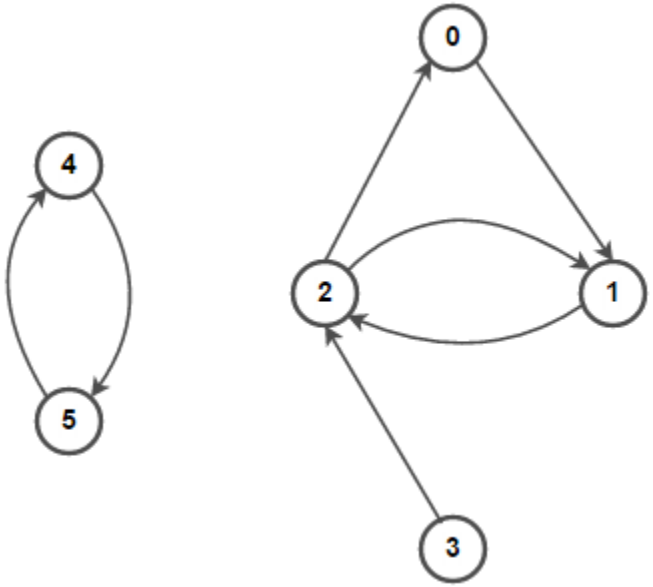
Output

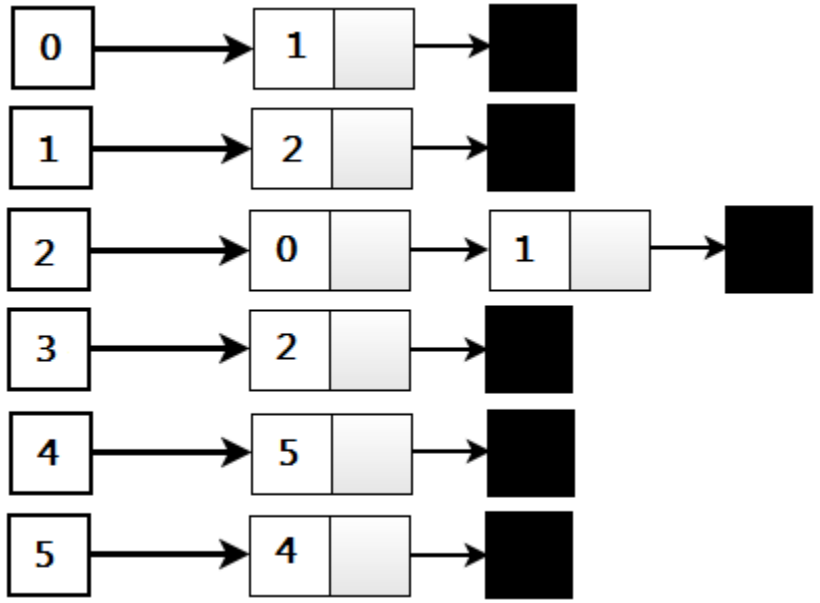
```java
import java.util.Comparator;
import java.util.TreeSet;

class Voiture {
    private String marque;
    private int anneeFabrication;

    public Voiture(String marque, int anneeFabrication) {
        this.marque = marque; this.anneeFabrication = anneeFabrication;
    }
    public String getMarque() { return marque;}
    public int getAnneeFabrication() { return anneeFabrication;}
}

class ComparateurParAnneeFabrication implements Comparator<Voiture> {
    @Override
    public int compare(Voiture v1, Voiture v2) {
        return Integer.compare(v1.getAnneeFabrication(), v2.getAnneeFabrication());
    }
}

public class Main {
    public static void main(String[] args) {
        TreeSet<Voiture> ensemble = new TreeSet<>(new ComparateurParAnneeFabrication());
        Voiture v1 = new Voiture("Toyota", 2015);
        Voiture v2 = new Voiture("BMW", 2010);
        Voiture v3 = new Voiture("Tesla", 2020);
        ensemble.add(v1);  ensemble.add(v2);  ensemble.add(v3);

        for (Voiture voiture : ensemble.headSet(v3)) {
            System.out.println(voiture.getMarque() + ":" + voiture.getAnneeFabrication());
        }
    }
}
```

```
java -cp /tmp/yMAZKdAsTY Main
BMW:2010
Toyota:2015
```

Les graphes (1/2)

- Il est courant d'utiliser des bibliothèques spécialisées comme ***JGraphT*** (https://jgrapht.org/) pour manipuler des graphes en Java.
- Les graphes peuvent aussi être représentés à l'aide de structures de données personnalisées, telles que des listes d'adjacence ou des matrices d'adjacence.





Les graphes (2/2)

```java
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Edge {
    int src, dest;
    Edge(int src, int dest) { this.src = src; this.dest = dest; }
}

class Graph {
    List<List<Integer>> adjList = new ArrayList<>();
    public Graph(List<Edge> edges) {
        // trouve le sommet numéroté maximum
        int n = 0;
        for (Edge e: edges) n = Integer.max(n, Integer.max(e.src, e.dest));
        // alloue de la mémoire pour la liste de contiguïté
        for (int i = 0; i <= n; i++) adjList.add(i, new ArrayList<>());
        // ajoute des arêtes au graphe orienté
        for (Edge current: edges) adjList.get(current.src).add(current.dest);
    }
    // Fonction pour imprimer la représentation de la liste d'adjacence d'un graphe
    public static void printGraph(Graph graph) {
        int src = 0;
        int n = graph.adjList.size();
        while (src < n) {
            for (int dest: graph.adjList.get(src))
                System.out.print("(" + src + " --> " + dest + ")\t");
            System.out.println();
            src++;
        }
    }
}
```

```java
public class Exemple {
    public static void main (String[] args) {
        // Entrée : liste des arêtes dans un digraphe (selon le schéma ci-dessus)
        List<Edge> edges = Arrays.asList(new Edge(0, 1), new Edge(1, 2),
                new Edge(2, 0), new Edge(2, 1), new Edge(3, 2),
                new Edge(4, 5), new Edge(5, 4));

        // construit un graphe à partir de la liste d'arêtes donnée
        Graph graph = new Graph(edges);

        // affiche la représentation de la liste d'adjacence du graphe
        Graph.printGraph(graph);
    }
}
```

Généricité

Généricité en Java, Généricité en C++

Principe & Définition

- La généricité est un mécanisme permettant de créer des entités génériques (classes, interfaces, méthodes) qui peuvent être utilisés avec différents types de données.
- Elle permet d'éviter la duplication de code en écrivant des entités génériques pour traiter différents types ➤ Clarté et lisibilité du code.

Généricité	Polymorphisme
Création de composants réutilisables avec différents types de données sans connaître à l'avance leurs types (pas de duplication).	Permet aux objets de prendre différentes formes ou de se comporter de différentes manières (duplication selon le(s) type(s) souhaité(s)).
Utilise des entités paramétrées en Java ou des modèles en C++ pour créer des composants génériques.	Réalisé par héritage, interfaces, et classes abstraites.

Généricité en Java – Classes génériques (1/4)

- La généricité en Java utilise des paramètres de type génériques pour créer des classes, interfaces ou méthodes génériques.
- Une classe générique est définie en utilisant des paramètres de type délimités par (<>).

```
1  class MaClasse<T> {}
```

- Les *wildcards* tels que ? peuvent aussi être utilisés pour représenter des types inconnus.

```
2  class MaClasse<?> {}
```

- **Attention:** les *wildcards* ne sont pas utilisés pour définir directement les types de retour ou de paramètres de méthode.
- Il est possible de limiter les types autorisés en utilisant des paramètres de type bornés.

```
3  class MaClasse<T extends Number> {}
```

- Les classes génériques peuvent être héritées en conservant les paramètres de type génériques.

```
4  class MaSousClasse<T> extends MaClasse<T> {}
```

Généricité en Java – Classes génériques (2/4)

- N'importe quelle lettre peut être utiliser pour désigner le type paramétré (la généricité). Le choix de T est conventionnel et largement utilisé, mais ce n'est pas une obligation.
- Le choix de la lettre est purement conventionnel. Cependant, il est recommandé de choisir une lettre qui soit descriptive et qui reflète le rôle du type paramétré, ce qui peut améliorer la lisibilité et la compréhension du code.
  - T : Il est couramment utilisé pour représenter n'importe quel type.
  - E : Souvent utilisé pour représenter un élément (comme dans une collection).
  - K : Souvent utilisé pour représenter une clé (comme dans une map).
  - V : Souvent utilisé pour représenter une valeur (comme dans une map).
  - N : Souvent utilisé pour représenter un nombre.
  - R : Souvent utilisé pour représenter un résultat.
  - Même les lettres minuscules et des mots peuvent être utilisées comme types paramétrés.

Généricité en Java – Classes génériques (3/4)

- Lors de la création d'une instance de la classe générique, il est recommandé de spécifier le type que la classe (**T**) pour bénéficier des avantages de la vérification de type les warnings.

```
5  MaClasse<Integer> instance = new MaClasse<Integer> ();
```

- La généricité peut être utilisée pour définir des méthodes avec un nombre variable d'arguments. Cela est souvent appelé **varargs** (*variable-length argument*) et permet à une méthode de recevoir un nombre variable d'arguments.

```
1  public class GenericTestExemple {
2      // Méthode générique avec varargs
3      public static <T> void afficherElements(T... elements) {
4          System.out.print("Éléments : ");
5          for (T element : elements) System.out.print(element + " ");
6          System.out.println();
7      }
8
9      public static void main(String[] args) {
10         // Appel de la méthode avec différents types et nombres d'arguments
11         afficherElements(1, 2);
12         afficherElements("Java", "est", "puissant");
13         afficherElements(3.14, 2.71, 0.99, 1.56);
14     }
15 }
```

Généricité en Java – Classes génériques (4/4)

- L'utilisation de la généricité en Java permet d'éviter la nécessité de casting.

```java
public class Paire  {
    Object premier ;
    Object second;
    public Paire (Object a, Object b){
        premier = a; second = b;
    }
    public Object getPremier() { return premier; }
    public Object getSecond() { return second; }

    public static void main(String[] args) {
        Paire  p = new Paire  ("abc", "xyz");
        String x =  (String)p.getPremier();
        String y =  (String)p.getSecond();
        System.out.println(x);
        System.out.println(y);
    }
}
```

Généricité en profitant de la relation d'héritage

Casting obligatoire

```java
public class Paire<T>  {
    T premier;
    T second;
    public Paire (T a, T b){
        premier = a; second = b;
    }
    public T getPremier(){ return premier; }
    public T getSecond(){ return second; }

    public static void main(String[] args) {
        Paire<String>  p = new Paire<String> ("abc", "xyz");
        String x =  p.getPremier();
        String y =  p.getSecond();
        System.out.println(x);
        System.out.println(y);
    }
}
```

Utilisation d'un type générique

Pas de casting

Généricité en Java – Méthodes génériques

```java
import java.util.Arrays;
public class GenericTestExemple {
    // Méthode générique pour échanger deux éléments dans un tableau
    public static <T> void echangerElements(T[] tableau, int indice1, int indice2) {
        if (indice1 < 0 || indice1 >= tableau.length ||
            indice2 < 0 || indice2 >= tableau.length) {
            System.out.println("Indices invalides.");
        }
        T temp = tableau[indice1];
        tableau[indice1] = tableau[indice2];
        tableau[indice2] = temp;
    }

    public static void main(String[] args) {
        // Exemple d'utilisation avec un tableau d'entiers
        Integer[] tableauEntiers = {1, 2, 3, 4, 5};
        System.out.println("Avant l'échange : " + Arrays.toString(tableauEntiers));
        echangerElements(tableauEntiers, 1, 3);
        System.out.println("Après l'échange : " + Arrays.toString(tableauEntiers));

        // Exemple d'utilisation avec un tableau de chaînes de caractères
        String[] tableauChaines = {"Java", "C", "Python", "JavaScript"};
        System.out.println("Avant l'échange : " + Arrays.toString(tableauChaines));
        echangerElements(tableauChaines, 0, 2);
        System.out.println("Après l'échange : " + Arrays.toString(tableauChaines));
    }
}
```

Généricité en C++ (1/5)

- En C++, le mécanisme de généricité repose principalement sur les modèles (**templates**).
- Les modèles permettent de définir des classes, des fonctions et des objets dont les types peuvent être spécifiés plus tard.
- La déclaration d'un modèle commence avec le mot-clé **template** suivi de la définition du modèle.

```
1  template <typename T>
2  class MaClasse {
3      private:
4          T valeur;
5      public:
6          T obtenirValeur() { return valeur; }
7          void définirValeur(T nouvelleValeur) { valeur = nouvelleValeur; }
8  };
```

Généricité en C++ (2/5)

- Lors de l' utilisation d' une classe ou une fonction modèle, on doit spécifier le type générique entre les chevrons (<>).

```
1  MaClasse<int> instance;
2  instance.définirValeur(42);
3  int valeur = instance.obtenirValeur();
```

- Les modèles peuvent également avoir des paramètres non typés, comme des valeurs entières.

```
1  template <int N>
2  class TableauFixe {
3      public:
4          int tableau[N];
5  };
```

Généricité en C++ (3/5)

- Les fonctions peuvent également être génériques en utilisant des modèles.

```
1  #include <iostream>
2  template <typename T>
3  T somme(T a, T b) { return a + b; }
4  int main() {
5      int sommeEntiers = somme(5, 7);
6      double sommeDoubles = somme(3.14, 2.71);
7      std::cout << "Somme : " << sommeEntiers << std::endl;
8      std::cout << "Somme : " << sommeDoubles << std::endl;
9  }
```

Généricité en C++ (4/5)

- Les classes modèles peuvent être héritées, et le polymorphisme est possible avec des modèles. Cela permet une grande flexibilité dans la création de structures génériques.

```
1  template <typename T>
2  class ClasseDeBase {
3      public:
4          virtual void afficher() { /* ... */ }
5  };
6
7  template <typename T>
8  class ClasseDérivée : public ClasseDeBase<T> {
9      public:
10         void afficher() override { /* ... */ }
11 };
```

Généricité en C++ (5/5)

- Les modèles de **variadic** permettent de travailler avec un nombre variable d'arguments de types différents.

```cpp
#include <iostream>
template <typename... Args>
void afficherArguments(Args... args) {
    (std::cout << ... << args) << std::endl;
}

int main() {
    afficherArguments(1, "Deux", 3.0);
}
```

Généricité en C++ - Exemple complet

```cpp
#include <iostream>
#include <vector>
template <typename T>
class Pile {
    private:
        std::vector<T> éléments;
    public:
        void empiler(const T &élément) { éléments.push_back(élément); }
        void dépiler() { if (!estVide()) éléments.pop_back(); }
        const T &sommet() const {
            if (!estVide()) return éléments.back();
            else throw std::out_of_range("La pile est vide.");
        }
        bool estVide() const { return éléments.empty(); }
};

int main() {
    // Utilisation de la pile avec des entiers
    Pile<int> pileEntiers;
    pileEntiers.empiler(10); pileEntiers.empiler(20); pileEntiers.empiler(30);
    std::cout << "Sommet de la pile d'entiers : " << pileEntiers.sommet() << std::endl;

    // Utilisation de la pile avec des chaînes de caractères
    Pile<std::string> pileChaines;
    pileChaines.empiler("Première chaîne");
    pileChaines.empiler("Deuxième chaîne");
    std::cout << "Sommet de la pile de chaînes : " << pileChaines.sommet() << std::endl;
    pileEntiers.dépiler();
    std::cout << "Nouveau sommet de la pile d'entiers : " << pileEntiers.sommet() << std::endl;
    return 0;
}
```

Réflexivité

Principe & Définition

- La réflexivité se réfère à la capacité d'un programme à examiner/manipuler son propre état, comportement et les informations relatives aux objets en exécution.
- La réflexivité est utile dans les cas suivants:
  - **Sérialisation et désérialisation :** Lorsque vous avez besoin de convertir des objets en formats interchangeables tels que JSON, XML, ou d'autres, la réflexivité peut être utilisée pour inspecter les champs d'objets et les convertir en données sérialisées.
  - **Développement d'outils de débogage et de profilage :** Certains outils de débogage et de profilage utilisent la réflexivité pour examiner l'état interne des objets, les performances des méthodes, etc.
  - **Construction de frameworks génériques :** Les frameworks qui nécessitent une certaine flexibilité et extensibilité (ex. plugins), peuvent utiliser la réflexivité pour permettre la découverte et l'ajout dynamique de fonctionnalités.

Principe & Définition

- **Tests unitaires et mocking :** En test unitaire, la réflexivité peut être utilisée pour créer des instances de classe, accéder à des champs privés pour les tester, ou appeler des méthodes privées à des fins de test.
- **Génération dynamique de code (rétro-ingénierie** ou **reverse-engineering) :** La rétro-ingénierie implique la génération de code à la volée. La réflexivité permet d'explorer le code pendant l'exécution pour avoir des informations nécessaires à la génération du code source qui correspond.

- Bien que la réflexivité est utile et offre une grande flexibilité mais doit être utilisée avec prudence, car elle peut contourner certaines vérifications de type à la compilation.
- La réflexivité peut également affecter les performances.

La réflexivité en Java (1/3)

- En Java, tout objet est associé à une instance de la classe ***Class (java.long.Class)*** qui fournit des méthodes pour inspecter les informations d'une classe à l'exécution.
- On peut obtenir l'instance de ***Class*** associé à une classe ou un objet de trois façons :
  - En invoquant la méthode *obj*.***getClass()*** sur n'importe quel objet *obj*;
  - En appelant cet objet directement sur un nom d'une classe *MaClass*.***class***
  - En utilisant la méthode statique ***Class.forName(****MaClass****)***
- À l'aide de la réflexivité, il est possible de créer une instance d'une classe quelconque en se référant à son objet de classe ***Class: Class.forName(****"NomDeLaClasse"****).newInstance();***.
- Les champs peuvent être lus ou modifiés dynamiquement en utilisant les méthodes: ***getField(****"fieldName"****)*** et ***setField(****"fieldName", value****)***
  - ***getFileds()*** renvoie un tableau contenant les attributs publics et ***getDeclaredFields()*** renvoie un tableau de tous les attributs de la classe.
  - **setAccessible(true)** permet de changer la visibilité d'un attribut d'une classe.

La réflexivité en Java (2/3)

- Les méthodes peuvent être invoquées dynamiquement en utilisant ***getMethod(****"methodName", paramTypes****)*** et ***invoke(****obj, args****)***
  - **getMethods**() retourne uniquement les méthodes publics, ***getDeclaredMethods()*** retourne toutes les méthodes de la classe.
- ***getConstructors()*** et ***getConstructor(Class<?>... types)*** retournent le tableau des constructeurs publics de cette classe, ou le constructeur qui prend en paramètre la liste des classes (donc des types) indiquée.
  - ***getDeclaredConstructors()*** et ***getDeclaredConstructor(Class<?>... types)*** retournent le tableau des constructeurs de cette classes, publics ou non.

La réflexivité en Java (3/3)

- Autres méthodes utiles:
  - ***getName() :*** retourne le nom du type de cette classe (class, interface, type primitif ou tableau).
  - ***getSimpleName()*** : retourne le nom de cette classe, sans son nom de package.
  - ***getPackage()*** : retourne une instance de la classe Package, qui modélise le package dans lequel se trouve cette classe.
  - ***getSuperClass()*** : retourne la classe parant de cette classe, sous forme d'une instance de Class.
  - ***getInterfaces()*** : retourne un tableau des interfaces implémentées par cette classe, sous forme d'instances de Class.
  - ***isInterface() :*** retourne *true* si cette classe est une interface.

Réflexivité en Java - Exemple

```java
import java.lang.reflect.Field;
class Personne {
    private String nom;
    private int age;
    public Personne(String nom, int age) { this.nom = nom; this.age = age; }
}

public class Exemple {
    public static void main(String[] args) throws IllegalAccessException {
        // Créer une instance de la classe Person
        Personne personne = new Personne("Ahmed Salem", 30);
        // Utiliser la réflexivité pour obtenir les valeurs des champs
        Class<?> classPersonne = personne.getClass();
        Field[] attributs = classPersonne.getDeclaredFields();
        StringBuilder result = new StringBuilder();
        for (Field attr : attributs) {
            attr.setAccessible(true); // Autoriser l'accès aux champs privés
            String nomAttr = attr.getName();
            Object valAttr = attr.get(nomAttr);
            result.append(nomAttr).append(": ").append(valAttr).append(", ");
        }
        // Supprimer la virgule et l'espace extra à la fin
        if (result.length() > 0) result.setLength(result.length() - 2);
        System.out.println("Valeurs des attributs de l'objet Personne : " + result.toString());
    }
}
```