FICHE BASES DE JAVA

Types de données et variables

- En Java, les variables doivent être déclarées avec un type. Par exemple, `int`, `double`, `String` et `Date`.
- Les collections en Java sont **génériques**, il est donc important de spécifier le type d'objet qu'elles vont contenir.
- Les types primitifs (comme `int`, `double`) peuvent être automatiquement convertis en leurs équivalents objets (comme `Integer`, `Double`) par le compilateur.

Opérateurs * **Opérateurs d'affectation** : L'opérateur `=` est utilisé pour affecter une valeur à une variable. * **Opérateurs de comparaison**: `==` (égal à), `!=` (différent de), `<` (inférieur à), `>` (supérieur à), `<=` (inférieur ou égal à), `>=` (supérieur ou égal à) sont utilisés pour comparer les valeurs. * **Opérateurs arithmétiques**: `+` (addition), `-` (soustraction), `*` (multiplication), `/` (division), `%` (modulo) sont utilisés pour effectuer des opérations mathématiques. * **Opérateurs logiques** : `&&` (et logique), `||` (ou logique), `!` (non logique) sont utilisés pour combiner des expressions booléennes.

Structures de contrôle * `if, else`, et `else if` permettent d'exécuter des blocs de code conditionnellement. * `while` permet d'exécuter un bloc de code tant qu'une condition est vraie. * `for` permet d'itérer sur des collections ou d'exécuter un bloc de code un nombre de fois déterminé.

Collections

- Java offre plusieurs types de collections pour stocker et manipuler des groupes d'objets. Les principales sont:
  - `Set`: Ensemble d'objets non indexé, sans doublons. `HashSet` est une implémentation courante.
  - `List`: Ensemble d'objets non indexé, avec doublons possibles. `ArrayList` est une implémentation courante.
  - `Map`: Ensemble associatif d'objets (clé-valeur), non indexé. `HashMap` est une implémentation courante. Chaque clé est unique dans une `Map`.
  - `Queue`: Ensemble d'objets avec accès FIFO (First In First Out) ou LIFO (Last In First Out). `ArrayDeque` est une implémentation courante.
- **Parcours de Collections**:
  - `for`: Permet de parcourir les éléments d'une collection. Ne permet pas la modification de la collection pendant le parcours.
  - `Iterator`: Permet de parcourir et de modifier une collection pendant le parcours. `ListIterator` pour les listes permet d'aller dans les deux sens et de faire des ajouts.

Classes et Objets

- Java est un langage orienté objet.
- Une **classe** est un modèle pour créer des objets.
- Un **objet** est une instance d'une classe.
- Les classes peuvent avoir des **attributs** (variables) et des **méthodes** (fonctions).
- Les méthodes peuvent être appelées sur un objet en utilisant la notation pointée (par exemple, `objet.méthode()`).
- La classe `Cell` représente une cellule pour une liste chainée avec un champ `value` et un champ `next`.
- La classe `List` contient une cellule `head`, la taille de la liste et des méthodes pour manipuler la liste.

Listes chaînées

- Une liste chaînée est une structure de données qui consiste en une suite de nœuds (cellules), où chaque nœud contient une valeur et un pointeur vers le nœud suivant.
- Les opérations de base sur une liste chaînée incluent l'ajout (append), l'insertion (insert), la recherche (find), la récupération d'une valeur à un index donné (get), et le remplacement d'une valeur (replace).

Arbres * Un arbre est une structure de données hiérarchique composée de nœuds, où chaque nœud peut avoir des enfants. * Un parcours en **profondeur d'abord (DFS)** explore une branche de l'arbre autant que possible avant de passer à la branche suivante. * Un parcours en **largeur d'abord (BFS)** explore tous les nœuds d'un niveau avant de passer au niveau suivant. * Un **arbre binaire ordonné** est un type particulier d'arbre binaire où les valeurs des nœuds sont ordonnées de manière spécifique. * Les opérations courantes sur les arbres incluent l'insertion, la recherche, la détermination de la hauteur, et le comptage du nombre de feuilles.

Méthodes Importantes

- `add(E e)`: Ajoute un élément à une collection.
- `remove(Object o)/remove(int index)`: Supprime un élément d'une collection.
- `get(int index)`: Retourne l'élément à l'index spécifié dans une liste.
- `put(K key, V value)`: Ajoute ou modifie une association clé-valeur dans une map.
- `contains(Object o)`: Vérifie si une collection contient un élément spécifié.
- `indexOf(Object o)`: Retourne l'index de la première occurrence d'un élément dans une liste.

- `offer(E e)`: Ajoute un élément à la fin d'une file.
- `poll()`: Supprime et retourne l'élément au début d'une file.
- `push(E e)`: Ajoute un élément au début d'une pile (LIFO).
- `pop()`: Supprime et retourne l'élément au début d'une pile (LIFO).

Concepts Avancés

- **Récursivité**: Technique où une fonction s'appelle elle-même.
- **Généricité**: Utilisation de types paramétrés pour rendre le code plus flexible et sûr.
- **Interfaces**: Définitions de méthodes que les classes peuvent implémenter.

Cette fiche récapitule les concepts fondamentaux de Java abordés dans les sources. Il est important de noter que Java est un langage riche avec de nombreuses autres fonctionnalités, mais ces points sont essentiels pour commencer.

Voici une fiche plus détaillée sur les structures de contrôle, la généricité, les lambdas, les erreurs, les classes (abstraites, normales, héritées, interfaces) en Java, en utilisant les informations des sources et en ajoutant des exemples pour illustrer chaque concept.

Structures de Contrôle

- **Instructions conditionnelles (`if`, `else`, `else if`)**
  - Permettent d'exécuter un bloc de code uniquement si une condition est vraie.
  - `if (condition) { /* code si condition vraie */ }`
  - `if (condition) { /* code si condition vraie */ } else { /* code si condition fausse */ }`
  - `if (condition1) { /* code si condition1 vraie */ } else if (condition2) { /* code si condition2 vraie */ } else { /* code si aucune condition vraie */ }`
  - Exemple: `java int x = 10; if (x > 0) { System.out.println("x est positif"); } else if (x == 0) { System.out.println("x est nul"); } else { System.out.println("x est negatif"); }`
- **Boucles (`while`, `for`)**
  - `while`: Exécute un bloc de code tant qu'une condition est vraie.
    - `while (condition) { /* code à répéter */ }`
  - `for`: Utilisée pour itérer sur des collections ou exécuter un bloc de code un nombre de fois déterminé.
    - `for (initialisation; condition; incrémentation) { /* code à répéter */ }`
    - `for (Type element : collection) { /* code pour chaque élément */ }` (parcours de collection)
  - Exemples:

```java
// while
int i = 0;
while (i < 5) {
    System.out.println(i);
    i++;
}

// for (boucle classique)
for (int j = 0; j < 5; j++) {
    System.out.println(j);
}

// for (parcours de collection)
List<String> names = new ArrayList<>();
names.add("Alice");
names.add("Bob");
for (String name : names) {
     System.out.println(name);
}
```

Généricité

- La **généricité** permet de paramétrer les classes et les interfaces par des types. Cela permet d'écrire du code qui peut fonctionner avec différents types sans avoir besoin de le réécrire à chaque fois.

- Les collections en Java sont **génériques**, il est donc important de spécifier le type d'objet qu'elles vont contenir.
- On utilise des `<>` pour indiquer le type générique.
- Exemples:

```java
// Listes
List<Integer> numbers = new ArrayList<>();
List<String> words = new ArrayList<>();

// Sets
Set<Double> prices = new HashSet<>();

// Maps
Map<String, Integer> ages = new HashMap<>();
```

Lambdas * Les expressions **lambda** (ou lambdas) sont des fonctions anonymes qui peuvent être utilisées pour simplifier le code [Aucune source ne mentionne explicitement les lambdas, information hors source]. * Elles sont souvent utilisées avec les interfaces fonctionnelles, qui sont des interfaces avec une seule méthode abstraite. * Elles permettent de traiter des blocs de code comme des objets. * Exemple:

```java
interface Operation {
    int apply(int a, int b);
}

public class Main {
    public static void main(String[] args) {
      // Utilisation d'une lambda pour une addition
      Operation add = (a, b) -> a + b;
      System.out.println(add.apply(5, 3));

      // Utilisation d'une lambda pour une soustraction
      Operation subtract = (a, b) -> a - b;
      System.out.println(subtract.apply(5, 3));
  }
}
```

Gestion des Erreurs * En Java, les erreurs sont gérées en utilisant des exceptions. * Il existe différents types d'exceptions, par exemple, `IndexOutOfBoundsException` lors d'une tentative d'accéder à un index invalide d'une liste. * On utilise des blocs `try`, `catch`, et `finally` pour gérer les exceptions. [Aucune source ne mentionne explicitement la gestion des exceptions, information hors source]. * `try` : contient le code qui pourrait lever une exception. * `catch` : contient le code à exécuter en cas d'exception. * `finally` : contient le code à exécuter que l'exception soit levée ou non (optionnel) * Exemple: `java List<String> list = new ArrayList<>(); list.add("A"); try { System.out.println(list.get(5)); // ceci va lever une exception } catch (IndexOutOfBoundsException e) { System.out.println("Erreur : index hors limites"); } finally { System.out.println("Ceci sera exécuté quoi qu'il arrive"); }`

Classes

- Une **classe** est un modèle pour créer des objets. Elle définit les attributs (variables) et les méthodes (fonctions) que les objets de cette classe auront.
  - `class MaClasse { /* attributs et méthodes */ }`
- **Classes normales**: Des classes concrètes que l'on peut instancier directement [Aucune source ne mentionne explicitement ce terme, information hors source].
  - Exemple :

```java
public class Personne {
    String nom;
    int age;

public Personne(String nom, int age) {
    this.nom = nom;
     this.age = age;
```

```
    }
}
// Utilisation
Personne p = new Personne("Alice", 30);
```

- **Classes abstraites**: Des classes qui ne peuvent pas être instanciées directement, elles servent de base pour d'autres classes [Aucune source ne mentionne explicitement ce terme, information hors source].
  - Elles peuvent contenir des méthodes abstraites (sans implémentation) que les classes filles doivent implémenter.
  - Définition avec le mot clé `abstract`.
  - Exemple :

```
abstract class Forme {
  abstract public double calculerSurface();
  }

class Cercle extends Forme {
    double rayon;
    public Cercle(double rayon) {
        this.rayon = rayon;
    }
    @Override
     public double calculerSurface() {
        return Math.PI * rayon * rayon;
    }
  }
```

- **Classes héritées**: Des classes qui héritent des attributs et méthodes d'une autre classe (la classe parent).
  - L'héritage permet la réutilisation du code.
  - On utilise le mot clé `extends` pour l'héritage.
  - Exemple :

```
class Etudiant extends Personne {
    String matricule;
    public Etudiant(String nom, int age, String matricule) {
        super(nom, age); // appelle le constructeur de la classe Personne
         this.matricule = matricule;
    }
}
```

- **Interfaces**: Définissent un ensemble de méthodes que les classes peuvent implémenter.
  - Les interfaces n'ont pas d'attributs ni de méthodes concrètes (avant Java 8).
  - Une classe peut implémenter plusieurs interfaces.
  - On utilise le mot clé `implements` pour implémenter une interface.
  - Exemple :

```
  interface Affichable {
        void afficher();
  }

 class Livre implements Affichable {
    String titre;
     public Livre(String titre) {
        this.titre = titre;
     }
     @Override
     public void afficher() {
         System.out.println("Titre du livre: " + titre);
     }
  }
```

Exemple récapitulatif

Voici un exemple qui intègre plusieurs concepts :

```
import java.util.ArrayList;
import java.util.List;
interface Calculable {
    int calculer(int a, int b);
```

```
}
// Classe abstraite
abstract class Forme {
    abstract public double calculerSurface();
}
// Classe heritee
class Rectangle extends Forme {
   int longueur, largeur;
    public Rectangle(int longueur, int largeur) {
        this.longueur = longueur;
        this.largeur = largeur;
   }
   @Override
    public double calculerSurface() {
        return longueur * largeur;
   }
}
public class Main {
  public static void main(String[] args) {
      // Utilisation de lambda
      Calculable addition = (a, b) -> a + b;
      System.out.println("Addition: " + addition.calculer(5, 3));

      // Création d'une liste générique
      List<Integer> nombres = new ArrayList<>();
       nombres.add(10);
      nombres.add(20);
    try {
          System.out.println(nombres.get(5));
     } catch (IndexOutOfBoundsException e) {
          System.out.println("Erreur : index hors limites.");
     }
      // Boucle for
      for (Integer nombre : nombres) {
           System.out.println(nombre);
      }

    // Instanciation de classes concrètes et héritée
      Rectangle rectangle = new Rectangle(5, 10);
     System.out.println("Surface du rectangle : "+ rectangle.calculerSurface());
    }
}
```