Métho Révision

Au final d’apres chat on a 4 cours a resumer

▼ Cours 1 :

Résumé du cours sur les Exceptions et Assertions en Java

Concepts Clés à Retenir :

1. **Exceptions** :
   - **Définition** : Représentent des événements anormaux qui ne doivent pas se produire dans le fonctionnement normal du programme.
   - **Gestion des exceptions** :
     - **Bloc** `try` : Code qui peut potentiellement lever une exception.
     - **Bloc** `catch` : Gère l'exception spécifique levée dans le bloc `try`.
     - **Bloc** `finally` : S'exécute toujours après les blocs `try` et `catch`, utilisé pour fermer les ressources.

Exemple : Gérer une erreur de format de numéro.

```
javaCopy code
try {
    int somme = 0;
    for(String arg : args)
        somme += Integer.parseInt(arg);
    System.out.println(somme);
} catch (NumberFormatException nfe) {
    System.err.println(arg + " n’est pas un nombre!");
```

```
}
```

2. **Assertions** :
   - **Utilisation** : Vérifier des conditions supposées vraies (invariants, préconditions, postconditions) durant le développement.
   - **Syntaxe** :

     ```
     javaCopy code
     assert condition : "Message en cas d'échec";
     ```

   - **Activation** : Doit être activée explicitement lors de la compilation et de l'exécution.

   Exemple : Vérifier qu'une somme n'est pas négative avant de créditer un compte.

   ```
   javaCopy code
   void créditer(int somme) {
       assert(somme >= 0) : "somme négative illégale : " +
   somme;
       solde += somme;
   }
   ```

3. **Types d'exceptions** :
   - **Exceptions contrôlées** (`IOException`) : Doivent être attrapées ou déclarées dans la signature de la méthode.
   - **Exceptions non-contrôlées** (`RuntimeException` comme `NullPointerException`) : Ne nécessitent pas d'être explicitement gérées.
4. **Gestion des ressources (ARM)** :
   - **Avant JDK 7** : Fermeture manuelle des ressources dans le bloc `finally`.

- **Depuis JDK 7** : Utilisation de la syntaxe `try` with-resources pour une fermeture automatique.

```
javaCopy code
try (FileReader fr = new FileReader("fic.txt");
     BufferedReader br = new BufferedReader(fr)) {
    // Utilisation des ressources
}
```

À retenir :

- Comprendre la différence entre exceptions contrôlées et non-contrôlées.
- Savoir implémenter et utiliser des assertions pour le débogage.
- Maîtriser les blocs `try`, `catch`, et `finally` pour une gestion robuste des erreurs.
- Utiliser `try` with-resources pour une gestion efficace des ressources.

▼ Cours 2:

Résumé du cours sur les Compléments de Java : Généricité, Classes Anonymes, et Expressions Lambda

1. Généricité :

- **Définition** : Permet d'écrire des classes, interfaces, et méthodes qui peuvent opérer sur différents types de données, rendant le code plus réutilisable et robuste.
- **Classes Génériques** : Utilisent des types de paramètres pour définir des classes. Exemple : `ArrayList<T>` peut stocker différents types de données.
- **Méthodes Génériques** : Similaires aux classes génériques mais appliquées aux méthodes. Exemple : méthode qui calcule la taille d'une collection.

Exemple :

```
javaCopy code
public class Pile<T> {
    private ArrayList<T> elements = new ArrayList<>();
    public void pousse(T element) { elements.add(element);
}
    public T tire() { return elements.remove(elements.size
() - 1); }
}
```

2. Classes Anonymes :

- **Définition** : Sont définies sans nom et souvent utilisées là où les classes sont utilisées une seule fois.
- **Utilisation** : Pratique pour créer des instances d'interfaces avec des modifications mineures sans avoir besoin de nommer ou de définir explicitement une classe complète.

Exemple :

```
javaCopy code
Button b = new Button("Click");
b.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
});
```

3. Expressions Lambda :

- **Introduction** : Permettent de traiter des fonctions comme des variables, simplifiant la syntaxe des classes anonymes et améliorant la lisibilité et la concision du code.
- **Syntaxe** : `(paramètres) -> expression` ou `(paramètres) -> { instructions; }`

- **Interfaces Fonctionnelles** : Utilisées avec les expressions lambda, une interface fonctionnelle est une interface avec une seule méthode abstraite.

Exemple :

```
javaCopy code
Arrays.asList(1, 2, 3, 4).forEach(e -> System.out.println
(e * e));
```

À retenir :

- La **généricité** permet une programmation plus flexible et sûre grâce à la vérification de type à la compilation.
- Les **classes anonymes** offrent une manière rapide et localisée d'implémenter des membres d'interfaces ou des sous-classes sans les nommer.
- Les **expressions lambda** fournissent une syntaxe claire et concise pour la programmation fonctionnelle, facilitant l'utilisation des interfaces fonctionnelles pour passer des comportements en tant qu'arguments.

▼ Cours 3: (kotlin)

3. Variables

Variables Mutables (`var`) et Immuables (`val`)

Mutable (`var`):

```
kotlinCopy code
var age = 25
age = 26  // La valeur de la variable peut être modifiée
```

Immutable (`val`):

```
kotlinCopy code
val name = "John"
// name = "Doe"  // Ceci générera une erreur: val cannot b
e reassigned
```

Explication: `val` est utilisé pour les variables dont la valeur ne changera pas une fois assignée, tandis que `var` est utilisé pour les variables dont la valeur peut changer au cours de l'exécution du programme.

6. Sécurité null

Utilisation de l'opérateur `?` pour permettre les valeurs null:

```
kotlinCopy code
var description: String? = "This is a nullable string"
description = null  // Pas d'erreur car description est dé
claré comme nullable
```

Opérateur Elvis `?:`:

```
kotlinCopy code
val length = description?.length ?: 0
println(length)  // Affiche 0 si description est null, sin
on la longueur de la chaîne
```

Explication: L'opérateur Elvis `?:` permet de fournir une valeur par défaut en cas de null. Si `description` est non-null, `description?.length` renvoie la longueur de la chaîne. Si `description` est null, l'expression après `?:` (ici 0) est retournée.

Opérateur `!!`:

```
kotlinCopy code
val nonNullableDescription = description!!
```

```
// Utiliser nonNullableDescription après ceci lancera une
NullPointerException si description était null
```

Explication: L'opérateur `!!` convertit une référence nullable en une référence non-nullable et lance une `NullPointerException` si la référence était null. Cela devrait être utilisé uniquement lorsque vous êtes absolument sûr que l'objet n'est pas null.

7. Templates de chaînes

Interpolation de chaînes simple:

```
kotlinCopy code
val userName = "Alice"
println("Hello, $userName!")  // Affiche: Hello, Alice!
```

Expression dans les templates de chaînes:

```
kotlinCopy code
val items = 3
val pricePerItem = 9.99
println("Total cost: $${items * pricePerItem}")  // Affich
e: Total cost: $29.97
```

Explication: L'interpolation de chaînes dans Kotlin permet d'intégrer des variables et des expressions directement dans des chaînes de texte en les précédant d'un signe dollar `$`. Si l'expression est complexe (par exemple, une multiplication), elle peut être placée entre accolades `{}`.

Résumé de la Leçon 3 : Classes et Objets

1. Classes et Instances d'Objets

- **Définition de Classe** : Les classes sont des plans pour créer des objets. Elles contiennent des définitions de propriétés et de méthodes.
- **Instance de Classe** : L'utilisation d'une classe pour créer un objet. Chaque objet créé à partir d'une classe est une instance de cette classe.

```
kotlinCopy code
class House {
    var color: String = "white"
    fun updateColor(newColor: String) {
        color = newColor
    }
}
val myHouse = House()  // Création d'une instance
```

2. Constructeurs

- **Constructeur Principal** : Déclaré dans l'en-tête de la classe. Peut avoir des paramètres avec des valeurs par défaut ou obligatoires.

```
kotlinCopy code
class Person(val name: String, val age: Int = 30)
```

3. Héritage

- **Superclasses et Sous-classes** : Kotlin supporte l'héritage simple-parent, où chaque classe a une seule classe parente dont elle peut hériter.
- **Interfaces** : Peuvent être utilisées pour implémenter des fonctionnalités que les classes peuvent partager.

```
kotlinCopy code
open class Vehicle {
    fun drive() { println("Driving") }
}
```

```
class Car : Vehicle() {
    fun accelerate() { println("Accelerating") }
}
```

4. Fonctions d'Extension

- **Ajout de Fonctionnalités** : Permettent d'ajouter des méthodes à des classes existantes sans en modifier le code source.

```
kotlinCopy code
fun String.addExclamation() = this + "!"
println("Hello".addExclamation())  // Affiche "Hello!"
```

5. Classes Spéciales

- **Classes de Données** (`data class`) : Simplifient la création de classes qui sont principalement utilisées pour stocker des données. Kotlin génère automatiquement des méthodes telles que `equals()`, `hashCode()`, et `toString()`.
- **Objet Singleton** (`object`) : Utilisé pour créer une classe avec une seule instance dans l'application.

```
kotlinCopy code
data class User(val name: String, val age: Int)
```

6. Organisation du Code

- **Paquets** (`packages`) : Utilisés pour organiser le code en groupes logiques, facilitant la gestion des espaces de noms et l'utilisation des éléments à travers différentes parties de l'application.
- **Visibilité** : Contrôle qui peut accéder à quoi dans votre classe grâce aux modificateurs de visibilité (`private`, `protected`, `internal`, `public`).

À Retenir

Ces concepts sont cruciaux pour construire des applications robustes et maintenables en Kotlin. La compréhension de la façon de structurer le code avec des classes, de gérer l'héritage, et d'étendre les fonctionnalités des classes existantes sans les modifier directement est essentielle pour tout développeur Kotlin.

▼ Cours 4:

Résumé de la Partie 3 : Programmation Android avec Kotlin et Jetpack Compose

1. Jetpack Compose

- **Définition** : Jetpack Compose est un framework moderne pour construire des interfaces utilisateur en Kotlin de manière déclarative. Il simplifie et accélère le développement UI en permettant aux développeurs de décrire ce que leur interface doit faire, plutôt que de gérer les détails de mise en œuvre.
- **Approche Déclarative** : Contrairement à l'approche traditionnelle basée sur XML, Jetpack Compose utilise des fonctions Kotlin pour définir l'interface, ce qui permet une intégration plus fluide avec le code Kotlin et réduit la nécessité de code boilerplate.

2. Intégration avec Material Design

- **Composants Prêts à l'Emploi** : Jetpack Compose intègre étroitement Material Design, offrant une large gamme de composants d'interface utilisateur prêts à l'emploi qui suivent les principes de design de Material.
- **Personnalisation** : Bien que de nombreux composants soient prêts à l'emploi, ils peuvent être personnalisés pour répondre aux besoins spécifiques de votre application, garantissant que vous pouvez maintenir une identité visuelle unique tout en respectant les directives de Material Design.

3. Principes de Base de la Création d'Interface Utilisateur

- **Composables** : Les éléments de l'interface utilisateur dans Jetpack Compose sont définis par des fonctions composable, qui peuvent maintenir un état et réagir aux changements de manière efficace.
- **Gestion de l'État** : Jetpack Compose gère l'état de l'interface utilisateur de manière plus intuitive, facilitant la construction d'interfaces réactives sans la complexité du modèle traditionnel de gestion d'état.

4. Utilisation de Kotlin dans le Développement Android

- **Avantages** : Kotlin offre une syntaxe plus concise et expressive, une meilleure gestion des erreurs et une intégration transparente avec les API Java existantes.
- **Interopérabilité avec Java** : Kotlin fonctionne parfaitement avec Java, permettant aux développeurs d'utiliser les bibliothèques Java existantes tout en bénéficiant des améliorations de Kotlin.

5. Ressources et Tutoriels

- Le cours inclut des liens vers des guides de développement Android, des tutoriels pour apprendre Kotlin et Jetpack Compose, et des ressources pour utiliser Material Design efficacement.

À Retenir

Jetpack Compose et Material Design sont des outils puissants pour le développement d'applications Android, permettant une intégration plus profonde et plus naturelle du design et de la fonctionnalité. L'utilisation de Kotlin accélère le développement et améliore la qualité du code, ce qui est crucial pour créer des applications modernes, efficaces et maintenables.

Themes a réviser :

1. **Compléments de Java (exceptions, classes anonymes, expressions lambda) : 2 cours**

2. **Éléments de Kotlin : 3 mini - cours**
3. **Éléments de Jetpack Compose : 1 grand cours qui peux aussi avoir d’autre element du cours**