Chapitre 3
Programmation orientée objet et fonctionnelle

1. Introduction

La programmation orientée objet (POO) est un paradigme essentiel en Kotlin, permettant de structurer le code sous forme d'objets représentant des entités du monde réel. En combinant les principes de la POO avec les fonctionnalités avancées de Kotlin, les développeurs peuvent écrire du code plus réutilisable, modulaire et facile à maintenir.

Dans ce chapitre, nous explorons les concepts fondamentaux de la POO en Kotlin, notamment la création et la manipulation des classes et objets, ainsi que l'ajout de comportements à travers les propriétés et méthodes.

2. Dominer la programmation orientée objet en Kotlin

Kotlin prend en charge la programmation orientée objet et y ajoute des fonctionnalités modernes. La POO repose sur quatre piliers fondamentaux :

– l'encapsulation, qui protège les données et limite leur accès aux seules parties du code qui en ont besoin ;
– l'héritage, qui permet à une classe d'hériter des fonctionnalités d'une autre pour favoriser la réutilisation du code ;
– le polymorphisme, qui permet de manipuler des objets de différentes manières selon leur type réel ;
– l'abstraction, qui aide à structurer le code en cachant les détails d'implémentation et en ne révélant que ce qui est nécessaire.

Comparé à des langages plus verbeux comme Java, Kotlin permet une déclaration plus concise des classes et supprime une partie des lourdeurs syntaxiques présentes dans d'autres langages orientés objet comme Java. Grâce à ces améliorations, le code obtenu est généralement plus court et plus lisible.

3. Créer vos propres types avec classes et objets

En Kotlin, tout repose sur les objets et les classes. Une classe est une structure qui définit les caractéristiques et le comportement d'un objet. Elle sert de modèle à partir duquel des instances peuvent être créées. Un objet, quant à lui, est une instanciation concrète d'une classe.

Créer une classe permet de structurer les données en définissant des propriétés pour stocker des valeurs et des méthodes pour encapsuler des comportements. Une classe peut être aussi bien une simple structure regroupant des données qu'un élément complexe disposant de fonctionnalités avancées.

Kotlin introduit également des concepts facilitant la gestion des classes et objets, comme les classes de données (*data classes*), qui sont optimisées pour représenter des données sans nécessiter de code supplémentaire pour les opérations courantes comme l'égalité et la copie.

Les objets uniques peuvent être créés sans avoir besoin d'une classe dédiée grâce au mot-clé `object`. Deux usages principaux existent : la déclaration `object MonSingleton`, qui définit un singleton accessible globalement, et l'expression `object : Type { ... }`, qui crée un objet anonyme à usage unique, souvent pour implémenter une interface à la volée.

Découper le code en classes bien définies facilite la maintenance et la réutilisation.

4. Ajouter du comportement avec propriétés et méthodes

Une classe en Kotlin ne se limite pas à stocker des données; elle peut également contenir des fonctionnalités sous forme de méthodes, qui définissent le comportement des objets. Les méthodes permettent d'interagir avec les données stockées dans les propriétés d'un objet et d'exécuter des opérations spécifiques.

Les propriétés d'une classe sont des variables associées à un objet. Elles peuvent être mutables ou immuables, et leur accès peut être contrôlé à l'aide de modificateurs de visibilité pour protéger l'intégrité des données. Kotlin offre également la possibilité d'utiliser des getters et setters personnalisés pour affiner le contrôle sur la modification et la lecture des propriétés.

Les méthodes définissent les actions qu'un objet peut exécuter. Comme en Java, Kotlin autorise la surcharge : plusieurs méthodes peuvent partager le même nom à condition d'avoir une signature différente (nombre ou types des paramètres). Kotlin permet en plus de définir des valeurs par défaut sur les paramètres, ce qui réduit souvent le besoin de surcharger.

Kotlin introduit également des fonctionnalités avancées telles que les extensions, qui permettent d'ajouter des méthodes à des classes existantes sans les modifier directement, et les fonctions d'ordre supérieur, qui rendent le code plus expressif et modulaire.

En combinant les propriétés et les méthodes, une classe devient une unité autonome capable de représenter un concept de manière complète, tout en encapsulant les données et les comportements qui lui sont propres.

5. Contrôler la création d'objets avec constructeurs

Un constructeur est une fonction spéciale qui permet d'initialiser un objet au moment de sa création. En Kotlin, chaque classe possède un constructeur primaire, qui est défini directement dans l'en-tête de la classe, et peut également contenir un ou plusieurs constructeurs secondaires pour gérer différentes formes d'initialisation.

Le constructeur primaire permet de définir des paramètres qui seront directement assignés aux propriétés de l'objet lors de son instanciation. Cette approche réduit la nécessité d'écrire du code répétitif et assure une initialisation propre et efficace.

Les constructeurs secondaires, quant à eux, offrent plus de flexibilité en permettant des logiques d'initialisation spécifiques lorsqu'un objet est créé avec différentes configurations. Ils peuvent être utiles lorsque plusieurs façons de construire un objet sont nécessaires.

Kotlin introduit également des blocs d'initialisation (`init`) qui permettent d'exécuter du code dès la création de l'objet, sans avoir besoin de passer par un constructeur secondaire. Ces blocs sont exécutés immédiatement après l'appel au constructeur primaire.

Grâce à ces différentes options, la création d'objets en Kotlin est flexible, efficace et adaptée à divers cas d'usage, tout en garantissant un code clair et concis.

6. Réutiliser avec héritage et interfaces

L'héritage est un mécanisme central de la programmation orientée objet qui permet à une classe de réutiliser le comportement et les propriétés d'une autre classe. En Kotlin, une classe peut hériter d'une autre en utilisant le mot-clé `open`, car par défaut, toutes les classes sont finales et ne peuvent pas être dérivées.

Lorsqu'une classe hérite d'une autre, elle peut redéfinir certaines méthodes ou ajouter de nouvelles fonctionnalités tout en conservant celles de la classe parente. Cela favorise la réutilisation du code et permet d'organiser les fonctionnalités de manière plus modulaire.

Les interfaces offrent une autre manière de réutiliser du code. En Kotlin, comme en Java, une classe ne peut hériter que d'une seule classe parente, mais elle peut implémenter plusieurs interfaces. Une interface définit un ensemble de méthodes et de propriétés qu'une classe doit fournir, sans imposer d'implémentation par défaut.

En combinant héritage et interfaces, il est possible de structurer le code de manière efficace, en favorisant la modularité et en permettant des extensions futures sans compromettre la lisibilité et la maintenabilité du programme.

7. Adopter la programmation fonctionnelle pour un code efficace

La programmation fonctionnelle est un paradigme qui repose sur l'utilisation de fonctions pures, l'immuabilité et l'évitement des effets de bord (par exemple : modifier une variable globale, écrire dans un fichier ou afficher quelque chose à l'écran depuis l'intérieur d'une fonction). Kotlin prend en charge ce style de programmation et fournit des outils pour écrire du code plus concis et plus facile à tester.

Une fonction est dite pure lorsqu'elle renvoie toujours le même résultat pour les mêmes entrées et n'affecte pas l'état global du programme. Cela améliore la lisibilité et facilite la détection des erreurs.

L'immuabilité est un autre concept clé de la programmation fonctionnelle. En Kotlin, il est recommandé d'utiliser des variables immuables (`val`) et des structures de données immuables pour éviter les modifications accidentelles qui peuvent introduire des bugs difficiles à diagnostiquer.

Enfin, Kotlin permet d'utiliser des fonctions d'ordre supérieur, des expressions lambda et des fonctions anonymes, qui rendent le code plus fluide et expressif en permettant d'écrire des opérations complexes en quelques lignes seulement.

Combiné à la POO, ce style fonctionnel permet d'écrire un code plus concis et plus facile à maintenir.

8. Simplifier vos fonctions avec lambdas et expressions anonymes

Les expressions lambda et les fonctions anonymes sont des éléments clés de la programmation fonctionnelle en Kotlin. Une lambda est une fonction sans nom qui peut être stockée dans une variable ou passée en argument à une autre fonction. Elle permet d'écrire du code plus concis en supprimant la nécessité de déclarer explicitement une fonction classique.

Les fonctions anonymes sont similaires aux lambdas, mais elles permettent d'avoir plus de contrôle, notamment lorsqu'il s'agit de retourner des valeurs dans des expressions complexes.

L'utilisation des lambdas est particulièrement utile dans les opérations sur les collections, comme les filtres et les transformations de données, où elles remplacent avantageusement les boucles traditionnelles en rendant le code plus lisible et expressif.

Grâce à ces outils, il est possible d'écrire des programmes plus compacts et efficaces, en réduisant la complexité syntaxique et en améliorant la compréhension du code.

9. Manipuler des fonctions avec fonctions d'ordre supérieur

Une fonction d'ordre supérieur est une fonction qui peut accepter une ou plusieurs fonctions en paramètre, ou en retourner une en sortie. Ce concept, issu de la programmation fonctionnelle, permet d'écrire du code plus modulaire et réutilisable en encapsulant des comportements sous forme de fonctions.

Kotlin facilite l'utilisation des fonctions d'ordre supérieur en permettant de passer des lambdas ou des références de fonctions en argument à d'autres fonctions. Cela est particulièrement utile pour abstraire des opérations répétitives et rendre le code plus générique.

Les fonctions d'ordre supérieur sont couramment utilisées dans les bibliothèques standard de Kotlin pour manipuler les collections, comme avec les méthodes `map`, `filter` ou `reduce`, qui permettent d'appliquer des transformations sans avoir recours aux boucles classiques.

En maîtrisant ces concepts, il devient possible d'écrire un code plus flexible, évolutif et facile à maintenir, en exploitant pleinement la puissance des fonctions en Kotlin.

10. Optimiser avec fonctions inline

L'optimisation du code est un enjeu majeur en développement, notamment lorsqu'il s'agit d'améliorer les performances d'une application. En Kotlin, l'utilisation des fonctions `inline` permet d'optimiser l'exécution du code en réduisant les coûts liés aux appels de fonctions d'ordre supérieur.

Normalement, lorsqu'une fonction est appelée, la JVM crée une nouvelle entrée sur la pile d'exécution et gère un saut d'exécution pour retourner le résultat. Pour une fonction d'ordre supérieur, la JVM crée en plus un objet pour représenter la lambda passée en paramètre, ce qui peut peser sur les performances quand l'appel est répété très souvent.

En déclarant une fonction comme `inline`, Kotlin remplace l'appel de cette fonction par son code directement au moment de la compilation. Cela réduit la surcharge liée aux appels de fonction et améliore les performances, en particulier dans des cas où des lambdas sont utilisées intensivement.

L'inlining est particulièrement utile lorsque des fonctions sont utilisées comme arguments dans d'autres fonctions, notamment dans des opérations sur les collections. Toutefois, il est important d'utiliser cette optimisation avec précaution, car elle peut entraîner une augmentation de la taille du bytecode si elle est appliquée de manière excessive.

En résumé, `inline` permet d'éviter le coût d'allocation des lambdas tout en conservant la souplesse des fonctions d'ordre supérieur.

11. Immutabilité et fonctions pures : fiabilité et prévisibilité

L'immutabilité est un concept clé en programmation fonctionnelle qui consiste à éviter la modification des données après leur création. Une variable immuable ne peut être modifiée une fois initialisée, ce qui réduit le risque d'erreurs imprévisibles et améliore la robustesse du programme.

En Kotlin, il est recommandé d'utiliser val plutôt que var autant que possible afin de garantir que les valeurs ne puissent pas être altérées accidentellement. De plus, les collections immuables sont privilégiées pour empêcher toute modification involontaire des données stockées.

Une fonction pure est une fonction qui, pour les mêmes entrées, renvoie toujours le même résultat et ne modifie pas l'état global du programme. Elle n'a aucun effet de bord, ce qui signifie qu'elle ne change pas de variable externe et ne dépend pas d'un état extérieur.

L'utilisation de fonctions pures présente plusieurs avantages :

– Prévisibilité : le code devient plus facile à comprendre et à tester.
– Réduction des bugs : l'absence d'effets de bord élimine de nombreuses erreurs liées à la modification de variables globales.
– Facilité de test : les fonctions pures sont plus simples à tester puisqu'elles ne nécessitent pas d'environnement spécifique pour fonctionner.

Combiner immutabilité et fonctions pures aide à écrire un code dont le comportement est plus facile à prévoir, ce qui est utile pour les applications où la stabilité compte.

12. Manipuler des collections avec map, filter, etc.

La manipulation des collections est un élément clé en Kotlin, permettant d'appliquer des transformations et des filtres sur les données de manière concise et lisible. Grâce aux fonctions de traitement de collections, il est possible d'éviter les boucles classiques en favorisant un style plus expressif et fonctionnel.

L'une des opérations les plus courantes est la transformation d'une collection avec la fonction `map`. Elle permet de modifier chaque élément d'une liste et de générer une nouvelle liste avec les valeurs transformées. Par exemple, lorsqu'il s'agit d'extraire ou de convertir des données d'un format à un autre, `map` s'avère particulièrement utile.

Le filtrage des données est également une tâche essentielle et se fait grâce à la fonction `filter`. Elle permet de ne conserver que les éléments répondant à une condition précise. Cette opération est souvent utilisée pour extraire des valeurs spécifiques dans un ensemble de données, comme récupérer uniquement les nombres pairs d'une liste ou isoler des objets respectant certains critères.

Lorsque l'on travaille avec des collections imbriquées, `flatMap` offre une solution efficace. Contrairement à `map`, qui applique une transformation et conserve la structure de la liste, `flatMap` aplatit le résultat en une seule liste. Cela est particulièrement utile lorsqu'une liste contient des sous-listes et que l'on souhaite extraire tous les éléments en une seule séquence continue.

Le regroupement des données est une autre opération fréquemment utilisée. Grâce à `groupBy`, une collection peut être organisée selon une clé donnée. Cela permet, par exemple, de classer des éléments par catégorie ou de structurer des données de manière plus efficace sans avoir à manipuler manuellement des structures de regroupement.