Jour1

Kotlin

Introduction à la plate-forme Android

Présentation du système d’exploitation et de ses possibilités

Android est un système d’exploitation mobile basé sur le noyau Linux.

Il a été développé par Google et est utilisé par de nombreux fabricants de smartphones et de tablettes.

Historique d’Android

Android a été créé en 2003 par une société appelée Android Inc.

Cette entreprise a été rachetée par Google en 2005.

La première version d’Android, appelée “Cupcake”, a été lancée en 2009. Depuis lors, de nombreuses versions ont été publiées, chacune avec de nouvelles fonctionnalités et améliorations.

Android est devenu l’un des systèmes d’exploitation mobiles les plus populaires, avec une part de marché de plus de 85% dans le monde entier.

Versions d’Android et versions d’API

Évolution des versions

Historique des versions d’Android

Android, le système d’exploitation mobile de Google, a été lancé en 2008. Depuis lors, de nombreuses versions ont été publiées, chacune avec un nom de code unique (inspiré de desserts) et un numéro de version.

Chaque version d’Android est associée à une version d’API, qui définit les fonctionnalités et les outils de développement disponibles pour cette version spécifique d’Android.

Nom de code et numéros de version

Voici un bref aperçu des noms de code et des numéros de version d’Android, dans l’ordre chronologique :

- Android 1.0 …
- Android 5.0 Lollipop ajout de Material Design
- Android 6.0 Marshmallow ajout de l’API de permission
- Android 7.0 Nougat
- Android 8.0 Oreo
- Android 9.0 Pie
- Android 10.0 Q
- Android 11.0 R

Chaque numéro de version est associé à une API, qui définit les fonctionnalités et les outils de développement disponibles pour cette version spécifique d’Android.

Les développeurs peuvent généralement cibler plusieurs versions d’API pour garantir la compatibilité avec différents appareils et versions d’Android.

Fréquence de sortie des mises à jour

Les mises à jour d’Android sont publiées régulièrement, avec une fréquence variant de quelques mois à plus d’un an entre chaque version.

Cependant, les fabricants d’appareils peuvent prendre du temps pour pousser ces mises à jour vers leurs appareils, en fonction de divers facteurs tels que les contrats de

support et les personnalisations de l’interface utilisateur.

Compatibilité ascendante

Le principe de compatibilité ascendante stipule que les applications développées pour une version d’Android précédente doivent être compatibles avec les nouvelles versions d’Android.

Il est donc important de développer des applications de manière à assurer cette compatibilité.

Gestion des dépendances et des bibliothèques

Il est important de prendre en compte les versions des dépendances et des bibliothèques utilisées dans le développement de votre application.

Il est recommandé d’utiliser les versions les plus récentes, mais également de prendre en compte les versions de compatibilité avec les différentes versions d’Android et d’API.

Pratiques recommandées pour assurer la compatibilité

Pour assurer la compatibilité ascendante de votre application, voici quelques pratiques recommandées :

- Utiliser les bibliothèques de compatibilité lorsque cela est possible
- Tester votre application sur différentes versions d’Android
- Utiliser les APIs les plus récentes lorsque cela est possible, tout en assurant la compatibilité avec les versions précédentes
- Prendre en compte les différences de comportement selon les versions d’Android et d’API

Améliorations de performance

Kotlin est un langage très performant.

Le compilateur Kotlin est capable de générer du bytecode Java qui est aussi performant que celui généré manuellement.

Les bibliothèques standard Kotlin sont également très performantes.

Nouveaux composants et API

Kotlin est compatible avec toutes les bibliothèques et API Android.

En outre, Kotlin ajoute de nouvelles API et bibliothèques qui facilitent le développement pour Android.

La bibliothèque Anko, par exemple, fournit de nombreuses extensions pour simplifier la création d'interfaces utilisateur.

Changements de design et d'interface utilisateur

Les développeurs peuvent utiliser Kotlin pour créer des applications Android avec des designs et des interfaces utilisateur modernes et intuitives.

Les bibliothèques Android, comme Material Design, ont été conçues pour être facilement utilisées avec Kotlin.

Rôle de Google dans l'écosystème Android

Google joue un rôle important dans l'écosystème Android en fournissant des services et des applications qui sont souvent considérés comme essentiels pour une expérience utilisateur satisfaisante.

Services et applications propriétaires de Google

Google fournit un certain nombre de services et d'applications qui sont préinstallés sur la plupart des appareils Android, tels que :

- Google Play Store : la boutique en ligne pour télécharger des applications
- Google Maps : service de cartographie et de navigation
- Google Drive : service de stockage en nuage
- Google Assistant : assistant personnel intelligent
- Gmail : service de messagerie électronique

Mises à jour de sécurité et de fonctionnalités

Google est responsable de la mise à jour de sécurité et de fonctionnalités pour son écosystème Android.

Les mises à jour de sécurité sont publiées régulièrement pour protéger les appareils Android contre les vulnérabilités connues.

Les mises à jour de fonctionnalités ajoutent de nouvelles fonctionnalités et améliorent l'expérience utilisateur.

Google Play Services et Google Play Store

Google Play Services est un ensemble de services basés sur le cloud fournis par Google pour les appareils Android.

Ces services permettent aux applications d'accéder à des fonctionnalités, telles que la géolocalisation, les notifications push, les services Google Play, etc.

Les développeurs peuvent utiliser ces services pour intégrer des fonctionnalités Google à leurs applications.

Fonctionnement de Google Play Services

Google Play Services est une application qui s'exécute en arrière-plan sur les appareils Android.

Elle fournit des services aux applications installées sur les appareils.

L'application Google Play Services est régulièrement mise à jour pour corriger les bugs, améliorer les performances et ajouter de nouvelles fonctionnalités.

Exemple basique

Un exemple simple d'utilisation de Google Play Services est l'API de géolocalisation.

Un développeur peut utiliser l'API de géolocalisation pour accéder à la position de l'utilisateur à partir de son appareil Android.

Grâce à Google Play Services, l'API de géolocalisation peut fournir une précision de localisation élevée, même en intérieur.

Langages de développement et NDK/SDK

Android SDK (Software Development Kit)

Présentation du SDK Android

Le SDK Android contient tous les outils dont vous avez besoin pour créer des applications Android.

Il comprend la plateforme Android, les outils de développement, les émulateurs, la documentation et les exemples de code.

Mise à jour et gestion des versions du SDK

La mise à jour du SDK Android est cruciale pour pouvoir bénéficier des dernières fonctionnalités et corrections de bugs.

Le SDK Manager permet de gérer les packages installés, de vérifier si de nouvelles mises à jour sont disponibles, et de les télécharger.

Il est recommandé de faire des mises à jour régulières pour maintenir votre environnement de développement à jour.

Android NDK (Native Development Kit)

Présentation du NDK Android

Le NDK Android, ou Native Development Kit, est un ensemble d’outils permettant de développer des applications Android en utilisant des langages de programmation natifs tels que C ou C++.

Cela permet de gagner en performances par rapport à l’utilisation de langages interprétés comme Java ou Kotlin.

Utilisation du NDK pour des performances optimales

Le NDK permet notamment de manipuler directement la mémoire du système, ce qui peut être utile pour des tâches nécessitant de hautes performances, comme le traitement d'images ou la simulation physique.

Intégration du NDK dans un projet Android

Pour intégrer le NDK dans un projet Android, il suffit de définir les fichiers sources en langage natif dans le répertoire `src/main/cpp` du projet, puis de définir les règles de compilation dans le fichier `CMakeLists.txt`.

La documentation officielle de Google fournit plus d'informations sur ces étapes.

Choisir le bon langage et kit de développement pour son projet

Le choix du langage de développement et du kit de développement est essentiel pour la réussite d'un projet.

Il existe plusieurs langages de développement couramment utilisés pour le développement d'applications Android.

Java a longtemps été le choix par défaut des développeurs, mais Kotlin est de plus en plus populaire.

Décision entre SDK et NDK en fonction des besoins du projet

Lorsqu'il s'agit de choisir entre SDK et NDK, cela dépend des besoins du projet.

Le SDK est généralement utilisé pour les tâches de développement d'interface utilisateur, telles que la création d'activités, de fragments et de vues personnalisées.

Le NDK est utilisé pour les tâches de traitement intensif des performances, telles que les opérations de traitement d'image ou audio en temps réel.

Prise en main de l'environnement de développement Android Studio

Dans ce chapitre, nous verrons comment installer et configurer Android Studio, l'environnement de développement intégré (IDE) officiel pour développer des applications Android en utilisant Kotlin.

Installation et configuration d'Android Studio

Gestion des mises à jour d'Android Studio

Il est important de garder votre installation d'Android Studio à jour, car les mises à jour contiennent souvent des corrections de bugs, des améliorations de performances et de nouvelles fonctionnalités.

Pour vérifier si une mise à jour est disponible, cliquez sur "Help", puis "Check for Updates".

Si une mise à jour est disponible, suivez les instructions pour la télécharger et l'installer.

Création d'un nouveau projet

1. Ouvrir l'application Android Studio.
2. Cliquer sur "Nouveau projet".
3. L'assistant de création de projet apparaîtra.

Configurer le projet avec les options suivantes : _ Nom : le nom du projet. _ Package : le package du projet (il doit être unique pour chaque application). * API cible : la version du système d'exploitation Android que l'application prend en charge.

Structure de fichiers et de dossiers du projet

Après la création d'un nouveau projet, Android Studio génère automatiquement une structure de fichiers et de dossiers.

- Le dossier "app" contient le code de l'application, les tests unitaires et les ressources utilisées dans l'application.

- Le fichier “AndroidManifest.xml” est un fichier maintenu par Android qui définit les caractéristiques de l’application.
- Le fichier “build.gradle” est un fichier qui définit les dépendances et la configuration pour la compilation et la construction de l’application.
- Le dossier “res” contient les fichiers de ressources utilisés dans l’application, tels que les fichiers xml de formes, les fichiers de mise en page et les images.

Exploration de l’interface d’Android Studio

Lorsque vous lancez Android Studio, vous remarquerez plusieurs composants visibles à l’écran.

Voici une brève description de ceux-ci :

- La barre d’outils : Cette barre contient des boutons utiles pour la création, la compilation et le déploiement des applications.
- Le navigateur de projets : Cette vue affiche tous les fichiers associés à votre projet.
- L’éditeur de code : C’est ici que vous allez éditer et afficher le code source de votre projet.

Navigateur de projets et éditeur de code

Le navigateur de projets est l’endroit où se trouve la liste de tous les fichiers associés à votre projet.

C’est ici que vous pouvez ajouter des fichiers ou modifier des fichiers existants.

L’éditeur de code est l’endroit où vous pouvez voir et éditer le code source de votre projet.

Configuration et utilisation d’émulateurs Android

Pour configurer un émulateur Android, il suffit d’aller dans AVD Manager (Android Virtual Device) dans la barre d’outils et de créer un nouvel émulateur.

Il est possible de choisir le modèle d’appareil souhaité ainsi que les caractéristiques techniques (taille d’écran, processeur, etc.).

Une fois l’émulateur créé, il peut être lancé et utilisé pour tester l’application développée.

Connexion et débogage sur des appareils réels

Pour connecter et déboguer une application sur un appareil Android réel, il est nécessaire d’activer le mode développeur sur le téléphone et d’activer le débogage USB.

Une fois le téléphone connecté à l’ordinateur via un câble USB, il sera possible de transférer et tester l’application directement sur l’appareil réel.

Comparaison des performances entre émulateurs et appareils réels

Il est important de noter que les performances peuvent varier entre les émulateurs Android et les appareils réels.

Il est donc recommandé de tester l’application sur les deux supports pour s’assurer du bon fonctionnement de celle-ci.

Langage Kotlin

Introduction à Kotlin

Historique de Kotlin

Kotlin est un langage de programmation développé par JetBrains, une entreprise qui propose des outils de développement pour les langages de programmation.

Kotlin a été introduit en 2011, mais sa première version stable a été publiée en 2016.

Origines et motivations

Les concepteurs de Kotlin ont été motivés par le désir de créer un langage de programmation plus simple, plus expressif et plus sûr que Java, tout en restant

compatible avec celui-ci.

En effet, Kotlin fonctionne sur la machine virtuelle Java (JVM) et peut interagir avec les bibliothèques et les outils Java existants.

Évolution et versions majeures

Kotlin est un langage en constante évolution.

Depuis sa première version stable, plusieurs versions majeures ont été publiées, apportant de nouvelles fonctionnalités et améliorations.

La version actuelle de Kotlin est la version 1.5, publiée en mai 2021.

Exemple basique

Voici un exemple de programme Kotlin très simple:

```
fun main() {
    println("Bonjour, le monde!")
}
```

Avantages de Kotlin par rapport à Java

- **Lisibilité et concision** : Kotlin réduit considérablement la quantité de code nécessaire pour effectuer les mêmes tâches en Java.
- **Sécurité et nullabilité** : Kotlin introduit un système de typage plus sûr qui offre une protection accrue contre les erreurs de programmation courantes, en particulier les erreurs liées à la nullabilité.
- **Fonctionnalités modernes** : Kotlin offre de nombreuses fonctionnalités modernes comme les lambdas, les fonctions d’extension, la réplication des fonctions, la prise en charge des flux de données, la prise en charge de la programmation asynchrone, etc.
- **Interopérabilité**: Kotlin peut interagir avec des bibliothèques Java existantes sans aucun problème.

Kotlin et la compatibilité avec Java

Kotlin est totalement compatible avec les applications Java existantes.

Cela signifie qu'il est possible d'utiliser des bibliothèques et des frameworks Java depuis Kotlin.

De plus, la plupart des éditeurs de code Java peuvent prendre en charge le code Kotlin, ce qui facilite l'intégration de Kotlin dans les projets Java existants.

Appel de code Kotlin depuis Java

Pour utiliser du code Kotlin depuis Java, il est nécessaire de compiler le code Kotlin en bytecode Java.

Cela créera des fichiers `.class` qui pourront être utilisés par Java.

Mise à jour d'Android Studio

Android Studio doit être régulièrement mis à jour pour bénéficier des dernières fonctionnalités.

Pour vérifier si une mise à jour est disponible :

- Ouvrez Android Studio.
- Accédez à l'onglet "Help".
- Sélectionnez "Check for Updates".

Syntaxe et concepts de base

Variables et types de données

Les variables en Kotlin peuvent être déclarées en utilisant les mots-clés `val` et `var`.

- `val` est utilisé pour déclarer des variables immuables (constantes) dont la valeur ne peut pas être modifiée une fois définie.
- `var` est utilisé pour déclarer des variables mutables dont la valeur peut être modifiée tout au long du programme.

Types de données primitifs

- `Boolean` : vrai ou faux
- `Byte` : un entier de 8 bits
- `Short` : un entier de 16 bits
- `Int` : un entier de 32 bits
- `Long` : un entier de 64 bits
- `Float` : un nombre à virgule flottante de 32 bits
- `Double` : un nombre à virgule flottante de 64 bits
- `Char` : un caractère Unicode

Types de données complexes

- `String` : une chaîne de caractères
- `Array` : un tableau unidimensionnel (ou multidimensionnel) contenant des éléments d'un même type
- `List` : une liste ordonnée de plusieurs éléments (répétables)
- `Map` : une collection de clés et de valeurs associées

Type inference et casting

Kotlin supporte la **type inference**, c'est-à-dire que les types des variables peuvent être déduits automatiquement à partir de leur initialisation.

Exemple :

```
val x = 5 // Kotlin déduit automatiquement que x est un entier de type Int
```

Type inference et casting

Néanmoins, il est possible de spécifier explicitement le type d’une variable en utilisant le signe deux-points suivi du type.

Exemple :

```
val x: Int = 5 // x est un entier de type Int
```

Casting

Kotlin ne supporte pas la conversion implicite de type.

Il est donc nécessaire de recourir au **casting** pour convertir un type de données en un autre.

Exemple :

```
val x = "5"
val y = x.toInt() // y est un entier de type Int
```

Exemple basique

```
fun main() {
    val helloWorld = "Hello, World!" // variable constante avec type inference

    var n = 5 // variable avec type inference et valeur modifiable

    println(helloWorld) // affiche la chaine de caractères "Hello, World!"

    n *= 2 // n est maintenant égal à 10

    println(n) // affiche le nombre entier 10
}
```

Opérateurs et expressions

Les opérateurs et expressions sont des outils essentiels pour manipuler des données en Kotlin.

Ils permettent des calculs mathématiques simples et complexes, la comparaison des valeurs, la logique booléenne, l'affectation de valeurs, et bien plus encore.

Opérateurs arithmétiques

Les opérateurs arithmétiques de Kotlin incluent les opérateurs suivants:

- Additon (+)
- Soustraction (-)
- Multiplication (*)
- Division (/)
- Modulo (%)

Opérateurs de comparaison

Les opérateurs de comparaison de Kotlin comprennent les opérateurs suivants:

- Égalité (==)
- Différence (!=)
- Inférieur (<)
- Inférieur ou égal (<=)
- Supérieur (>)
- Supérieur ou égal (>=)

Opérateurs logiques

Les opérateurs logiques de Kotlin incluent les opérateurs suivants:

- ET (&&)
- OU (||)
- NOT (!)

Opérateurs d'affectation

Les opérateurs d’affectation de Kotlin incluent les opérateurs suivants:

- Affectation simple (=)
- Affectation avec addition (+=)
- Affectation avec soustraction (-=)
- Affectation avec multiplication (*=)
- Affectation avec division (/=)
- Affectation avec modulo (%=)

Opérateurs spécifiques à Kotlin

Kotlin possède également des opérateurs spécifiques qui sont uniques au langage.

Par exemple, le “elvis operator” (?:) permet de définir une valeur de secours en cas de donnée nulle.

Le “safe call operator” (?.) permet de vérifier si un objet est nul avant d’appeler une méthode sur celui-ci.

Exemple basique:

val a = 10 val b = 2

// Addition val c = a + b // c = 12

Exemple basique:

// Comparaison val bool = a < b // bool = false

// Logique val bool2 = (a > b) && (a == 10) // bool2 = true

Exemple basique:

// Affectation var d = c d += 5 // d = 17

Exemple basique:

// Elvis operator val e = null val f = e ?: “valeur de secours” // f = “valeur de secours”

Exemple basique:

// Safe call operator val str: String? = null val length = str?.length // length = null (car str est nul)

Structures de contrôle

Conditionnelle if

L’instruction “if” permet de contrôler le flow d’un programme en vérifiant une condition.

```
if (condition) {
  // code à exécuter si la condition est vraie
} else {
  // code à exécuter si la condition est fausse
}
```

Expression when

L’expression “when” est similaire à un switch statement en Java.

Elle permet de vérifier une valeur sur plusieurs branches.

```
when (valeur) {
  valeur1 -> // code à exécuter si la valeur est égale à valeur1
  valeur2 -> // code à exécuter si la valeur est égale à valeur2
  else -> // code à exécuter si la valeur ne correspond à aucune des autres branches
}
```

Boucle for

La boucle “for” permet d’itérer sur une collection.

```
for (element in collection) {
  // code à exécuter pour chaque élément de la collection
}
```

Boucle while et do-while

Les boucles while et do-while sont similaires à celles en Java.

```
while (condition) {
  // code à exécuter tant que la condition est vraie
}

do {
  // code à exécuter au moins une fois, puis tant que la condition est vraie
} while (condition)
```

Fonctions et paramètres

Définition de fonctions

En Kotlin, les fonctions sont définies avec le mot-clé `fun`.

Voici un exemple simple de fonction qui renvoie la somme de deux nombres :

```
fun add(a: Int, b: Int): Int {
    return a + b
}
```

Paramètres nommés et valeurs par défaut

Kotlin permet d’utiliser des paramètres nommés.

Cela signifie que vous pouvez spécifier le nom d’un paramètre lors de l’appel de la fonction.

```
fun printPerson(firstName: String, lastName: String, age: Int = 0) {
    println("Nom complet : $firstName $lastName, âge : $age")
}

printPerson(lastName = "Doe", firstName = "John", age = 25)
```

Fonctions d’extension

Les fonctions d’extension permettent d’ajouter de nouvelles fonctionnalités à une classe existante sans avoir à modifier la classe elle-même.

```
fun String.addExclamation(): String {
    return "$this!"
}

val s = "Bonjour"
println(s.addExclamation()) // affiche "Bonjour!"
```

Fonctions de haut niveau et lambdas

Les fonctions de haut niveau sont des fonctions qui peuvent être passées comme arguments à d’autres fonctions.

```
fun main() {
    // Appel de la fonction printResult avec une lambda en paramètre
    printResult(5, 7, { a, b -> a + b })
}

// Fonction prenant trois paramètres : deux entiers (a et b) et une lambda (operation)
fun printResult(a: Int, b: Int, operation: (Int, Int) -> Int) {
    val result = operation(a, b) // Appel de la lambda avec les paramètres a et b
    println("Le résultat de l'opération est: $result")
}
```

Classes et objets en Kotlin

Les classes et les objets sont des éléments fondamentaux de la programmation orientée objet en Kotlin.

Définition de classes

Voici un exemple de définition de classe simple :

```
class Person(val name: String, var age: Int) {
    // Propriétés et méthodes de la classe Person
}
```

Constructeurs primaires et secondaires

En Kotlin, une classe peut avoir un ou plusieurs constructeurs.

Le premier constructeur défini dans une classe est appelé “constructeur primaire”.

Voici un exemple de classe qui possède un constructeur primaire :

```
class Person(val name: String, var age: Int) {
    // Propriétés et méthodes de la classe Person
}
```

Constructeurs primaires et secondaires

Une classe peut aussi avoir des “constructeurs secondaires”.

Voici un exemple de classe qui possède un constructeur secondaire :

```
class Person(val name: String, var age: Int) {
    // Propriétés et méthodes de la classe Person

    constructor(name: String) : this(name, 0) {
        // Constructeur secondaire qui accepte seulement le nom de la personne
    }
}
```

Initialisation et bloc init

Lors de la création d’une instance d’une classe, les propriétés de l’objet peuvent être initialisées dans le constructeur primaire ou dans un bloc “init”.

```
class Person(val name: String, var age: Int) {
    // Propriétés et méthodes de la classe Person

    val isAdult: Boolean

    init {
        isAdult = age >= 18
    }
}
```

Modificateurs de visibilité

En Kotlin, les propriétés et les méthodes d'une classe peuvent avoir des modificateurs de visibilité pour indiquer leur niveau de confidentialité.

Les modificateurs de visibilité disponibles en Kotlin sont :

- "public" (accessible depuis n'importe où)
- "private" (accessible seulement à l'intérieur de la classe)
- "protected" (accessible seulement à l'intérieur de la classe et de ses sous-classes)
- "internal" (accessible seulement à l'intérieur du module)

Modificateurs de visibilité

Voici un exemple de classe avec des propriétés et des méthodes avec différents modificateurs de visibilité :

```
class Person(val name: String, private var age: Int) {
    // Propriétés et méthodes de la classe Person

    internal fun getAge(): Int {
        return age
    }

    fun happyBirthday() {
        age++
    }
}
```

Exemple basique

```
fun main() {
    val person = Person("Alice", 25)
    println("Name: ${person.name}")
    println("Age: ${person.getAge()}")

    person.happyBirthday()
    println("New age: ${person.getAge()}")
}
```

Propriétés et méthodes

Une classe Kotlin peut avoir des propriétés et des méthodes.

Une propriété est une caractéristique de la classe tandis qu'une méthode est une action qui peut être effectuée sur la classe.

Propriétés avec getters et setters

En Kotlin, une propriété peut avoir un getter et un setter.

Le getter est une méthode qui permet de récupérer la valeur de la propriété tandis que le setter permet de la modifier.

Voici un exemple de déclaration d'une propriété avec getters et setters :

```
class Person {
  var name: String = ""
    get() = field
    set(value) {
      field = value
    }
}
```

Méthodes avec corps d'expression

En Kotlin, il est possible de définir une méthode avec un corps d'expression.

Cela permet de simplifier la syntaxe pour les instructions les plus simples.

```
class Calculatrice {
  fun somme(a: Int, b: Int) = a + b
}
```

Méthodes d’extension

En Kotlin, il est possible d’ajouter des méthodes à une classe existante sans devoir en hériter.

Ces méthodes sont appelées des méthodes d’extension et elles permettent d’étendre les fonctionnalités des classes de base.

```
fun String.direBonjour() = "Bonjour $this !"
```

Exemple basique

```
fun main() {
  val personne = Person()
  personne.name = "Alice"
  println(personne.name)

  val calculatrice = Calculatrice()
  println(calculatrice.somme(2, 3))

  val nom = "Jean"
  println(nom.direBonjour())
}
```

La sortie de ce programme sera :

```
Alice
5
Bonjour Jean !
```

Cet exemple utilise les classes et les méthodes que nous avons définies dans les sections précédentes.

Il crée une personne, une calculatrice et une chaîne de caractères, puis il utilise les méthodes pour effectuer différentes actions.

Héritage et polymorphisme

Le concept d'héritage est essentiel en programmation orientée objet.

Il permet de créer des classes dérivées à partir d'une classe de base, ce qui permet de réutiliser du code et de structurer son programme de manière logique.

En Kotlin, l'héritage est défini avec le mot-clé `open`, qui indique que la classe peut être héritée.

Par exemple :

```
open class Vehicule(val marque: String) {
    open fun afficherDetails() {
        println("Marque : $marque")
    }
}

class Voiture(marque: String, val modele: String) : Vehicule(marque) {
    override fun afficherDetails() {
        super.afficherDetails()
        println("Modèle : $modele")
    }
}
```

Dans cet exemple, la classe `Vehicule` est définie comme `open`, ce qui permet de la dériver avec la classe `Voiture`.

La classe `Voiture` hérite de la classe `Vehicule` grâce à la notation `: Vehicule(marque)`, qui permet de définir le constructeur de la classe de base.

La méthode `afficherDetails()` est définie comme `open` dans la classe de base, ce qui permet à la classe dérivée de la redéfinir avec le mot-clé `override`.

La méthode redéfinie utilise également `super.afficherDetails()` pour appeler la méthode de la classe de base avant d'afficher les détails propres à la classe dérivée.

Classe de base et classes dérivées

En Kotlin, chaque classe hérite implicitement de la classe `Any`, qui est la classe de base de toutes les classes.

Il est également possible de définir une classe de base personnalisée pour ses classes dérivées.

Par exemple :

```
abstract class Animal(val nom: String) {
    abstract fun dormir()
}

class Chat(nom: String) : Animal(nom) {
    override fun dormir() {
        println("Le chat dort enroulé en boule.")
    }

    fun chasser() {
        println("Le chat chasse les souris.")
    }
}
```

Dans cet exemple, la classe `Animal` est définie comme `abstract`, ce qui signifie qu'elle ne peut pas être instanciée directement.

La classe `Chat` hérite de la classe `Animal` et doit redéfinir la méthode `dormir()` avec le mot-clé `override`.

La classe `Chat` a également une méthode `chasser()` qui n'est pas définie dans la classe de base.

Cette méthode ne peut donc pas être appelée sur un objet de type `Animal`, mais uniquement sur un objet de type `Chat`.

Surcharge de méthodes

Il est possible de définir plusieurs méthodes avec le même nom dans une classe, à condition qu'elles aient des signatures différentes (c'est-à-dire qu'elles prennent des paramètres différents).

Par exemple :

```
open class Personne(val nom: String, val prenom: String) {
    open fun afficher() {
        println("$nom $prenom")
    }
}

class Employe(nom: String, prenom: String, val poste: String) : Personne(nom, prenom) {
    override fun afficher() {
        println("$nom $prenom, $poste")
    }

    fun afficher(nomFamille: String) {
        println("$nom $nomFamille, $poste")
    }
}
```

Dans cet exemple, la classe `Personne` a une méthode `afficher()` qui affiche les noms et prénoms de la personne.

La classe `Employe` hérite de `Personne` et redéfinit la méthode `afficher()` en y ajoutant le poste de l'employé.

La classe `Employe` a également une méthode `afficher(nomFamille: String)` qui prend un paramètre supplémentaire et qui permet d'afficher le nom de famille de la personne.

Polymorphisme et liaison dynamique

Le polymorphisme permet d'utiliser des objets dérivés comme s'ils étaient des objets de la classe de base.

Par exemple, si une méthode prend un objet de type `Animal` en paramètre, il est possible de lui passer un objet de type `Chat`, car `Chat` hérite de `Animal`.

La liaison dynamique permet à l'exécution de déterminer la méthode à appeler en fonction du type de l'objet, plutôt que du type de la variable qui le référence.

Par exemple :

```
fun afficherDetails(animal: Animal) {
    animal.dormir()
}

val chat = Chat("Minou")
afficherDetails(chat)
```

Dans cet exemple, `afficherDetails()` prend un argument de type `Animal`.

Lorsqu'on appelle cette fonction avec un objet de type `Chat`, la méthode `dormir()` de la classe `Chat` est appelée, même si la variable `animal` qui la référence est de type `Animal`.

Exemple

Voici un exemple basique qui utilise l'héritage et le polymorphisme en Kotlin :

```
open class Forme(val couleur: String) {
    open fun dessiner() {
        println("Je suis une forme de couleur $couleur.")
    }
}

class Carre(couleur: String) : Forme(couleur) {
    override fun dessiner() {
        println("Je suis un carré de couleur $couleur.")
    }
}

fun main() {
    val forme: Forme = Carre("rouge")
    forme.dessiner()
}
```

Dans cet exemple, la classe `Forme` est définie avec une méthode `dessiner()` qui affiche la couleur de la forme.

La classe `Carre` hérite de `Forme` et redéfinit la méthode `dessiner()` pour spécifier qu'il s'agit d'un carré.

Dans la fonction `main()`, on crée un objet `Carre` de couleur rouge qu'on stocke dans une variable de type `Forme`.

On appelle ensuite la méthode `dessiner()`, qui est redéfinie dans la classe `Carre`.

La méthode appelée dépend donc du type de l'objet, et non du type de la variable qui le référence.

Interfaces et classes abstraites

Les interfaces et classes abstraites sont des éléments clés de la programmation orientée objet en Kotlin.

Elles permettent de définir des modèles communs pour le comportement des objets.

Définition d'interfaces

Une interface est une définition abstraite d'un ensemble de méthodes et propriétés qui doit être implémenté par une classe.

Pour définir une interface en Kotlin, on utilise le mot-clé `interface`.

Par exemple :

```
interface Drawable {
    fun draw()
}
```

Ici, l'interface `Drawable` définit une seule méthode `draw()` qui doit être implémentée par toute classe qui implémente l'interface.

Implémentation d'interfaces

Pour implémenter une interface en Kotlin, il suffit de mettre le mot-clé `implements` suivi du nom de l'interface dans la définition de la classe.

Par exemple :

```
class Circle : Drawable {
    override fun draw() {
        //code pour dessiner un cercle
    }
}
```

Ici, la classe `Circle` implémente l'interface `Drawable` et redéfinit la méthode `draw()` pour dessiner un cercle.

Classes abstraites et méthodes abstraites

Les classes abstraites sont des classes qui ne peuvent pas être instanciées directement, mais qui définissent des modèles pour des classes qui peuvent être instanciées.

Les méthodes abstraites sont des méthodes qui doivent être implémentées par une classe dérivée.

Pour définir une classe abstraite en Kotlin, on utilise le mot-clé `abstract`.

Par exemple :

```
abstract class Shape {
    abstract fun draw()
}
```

Ici, la classe `Shape` est une classe abstraite qui définit une méthode abstraite `draw()`.

Pour implémenter une classe abstraite en Kotlin, on utilise le mot-clé `extends` suivi du nom de la classe dans la définition de la classe.

Par exemple :

```
class Circle : Shape() {
    override fun draw() {
        //code pour dessiner un cercle
    }
}
```

Ici, la classe `Circle` étend la classe abstraite `Shape` et redéfinit la méthode `draw()` pour dessiner un cercle.

Comparaison entre interfaces et classes abstraites

Les interfaces et les classes abstraites ont des similitudes et des différences.

Les interfaces :

- Définissent un contrat de comportement pour une classe
- N’ont pas d’implémentation
- Peuvent être implémentées par plusieurs classes

- Peuvent être utilisées pour définir des types dans les variables et les paramètres

Les classes abstraites :

- Définissent un modèle commun de comportement pour des classes dérivées
- Peuvent avoir des méthodes implémentées et non implémentées
- Peuvent être étendues par des classes dérivées
- Peuvent être utilisées pour définir des types dans les variables et les paramètres

---

Exemple basique

Voici un exemple simplifié d'utilisation d'interfaces et de classes abstraites en Kotlin :

```
interface Drawable {
    fun draw()
}

abstract class Shape : Drawable {
    abstract var x: Int
    abstract var y: Int
    abstract fun area(): Double
}

class Circle(override var x: Int, override var y: Int, val radius: Double) : Shape() {
    override fun draw() {
        //code pour dessiner un cercle
    }

    override fun area(): Double {
        return Math.PI * radius * radius
    }
}

fun main() {
    val circle = Circle(0, 0, 5.0)
    circle.draw()
    println("Area: ${circle.area()}")
}
```

Ici, nous avons une interface `Drawable` qui définit une méthode `draw()`.

Nous avons aussi une classe abstraite `Shape` qui étend l'interface `Drawable` et définit des propriétés abstraites `x` et `y`, ainsi qu'une méthode abstraite `area()`.

Enfin, nous avons une classe concrète `Circle` qui étend la classe abstraite `Shape` et redéfinit les propriétés abstraites et la méthode abstraite pour représenter un cercle.

Dans la fonction `main()`, nous créons un cercle et appelons sa méthode `draw()` et sa méthode `area()`.

Lambdas et fonctions d'ordre supérieur

Les lambdas et les fonctions d'ordre supérieur sont deux concepts clés de la programmation fonctionnelle.

Kotlin, en tant que langage orienté objet avec des fonctionnalités de programmation fonctionnelle, permet l'utilisation de ces deux concepts.

Définition et syntaxe des lambdas

Les lambdas sont des fonctions anonymes qui peuvent être utilisées comme des expressions.

Ils sont souvent passés comme arguments à d'autres fonctions, ou retournés en tant que résultat d'une autre fonction.

La syntaxe pour une lambda ressemble à ceci :

```
val lambda: (Int, Int) -> Int = { a, b -> a + b }
```

Ici, `lambda` est une variable avec une fonction lambda assignée.

Le type de la variable est défini comme `(Int, Int) -> Int`, ce qui indique une fonction qui prend deux paramètres de type `Int` et renvoie un résultat de type `Int`.

Dans ce cas, la fonction lambda ajoute les deux paramètres.

Utilisation des lambdas dans les fonctions

Les fonctions peuvent prendre des lambdas comme paramètres en utilisant la syntaxe suivante :

```
fun fonctionAvecLambda(a: Int, b: Int, lambda: (Int, Int) -> Int): Int {
    return lambda(a, b)
```

```
}
```

Cette fonction prend deux arguments de type `Int` et une fonction lambda qui prend deux paramètres `Int` et renvoie un résultat `Int`.

La fonction lambda est appelée à l'intérieur de la fonction `fonctionAvecLambda` en utilisant `lambda(a, b)`.

Fonctions d'ordre supérieur

Les fonctions d'ordre supérieur sont des fonctions qui prennent des fonctions comme paramètres, ou qui retournent des fonctions en tant que résultat.

Les lambdas sont souvent utilisées dans les fonctions d'ordre supérieur.

Voici un exemple de fonction d'ordre supérieur :

```
fun operationSurListe(liste: List<Int>, operation: (Int) -> Int): List<Int> {
    return liste.map(operation)
}
```

Cette fonction prend une liste d'entiers et une fonction qui prend un paramètre de type `Int` et renvoie un résultat de type `Int`.

La fonction `operationSurListe` applique la fonction passée en paramètre à chaque élément de la liste en utilisant `map`, et retourne une nouvelle liste avec les résultats.

Exemples d'utilisation de fonctions d'ordre supérieur

Voici un exemple simple d'utilisation de la fonction `operationSurListe` :

```
val liste = listOf(1, 2, 3, 4, 5)
val doubleListe = operationSurListe(liste) { a -> a * 2 }
```

Ici, nous créons une liste d'entiers de `1` à `5`.

Nous passons cette liste et une lambda qui multiplie chaque élément par `2` à la fonction `operationSurListe`.

La fonction renvoie une nouvelle liste avec les éléments multipliés.

La programmation fonctionnelle et l’utilisation de lambdas et de fonctions d’ordre supérieur sont des concepts importants pour les programmeurs Kotlin.

Avec ces outils, nous pouvons écrire des fonctions plus réutilisables et expressives.

Collections et opérations fonctionnelles

Collections en Kotlin (List, Set, Map)

En Kotlin, les collections sont des objets qui peuvent contenir plusieurs éléments du même type ou de types différents.

Il existe trois types de collections : les listes, les ensembles et les maps.

- La liste est une collection ordonnée d’éléments où chaque élément est accessible par son index.

Par exemple, si vous avez une liste de noms, vous pouvez accéder au troisième élément en utilisant le code :

```
val listeNoms = listOf("Luc", "Anne", "Pierre", "Nicolas")
println(listeNoms[2])
```

Ce qui affichera “Pierre” dans la console.

- L’ensemble est une collection d’éléments où chaque élément est unique.

Par exemple, si vous avez un ensemble de lettres, vous ne pouvez pas avoir deux fois la lettre “a”.

- La map est une collection clé-valeur où chaque clé est unique et correspond à une valeur.

Par exemple, si vous avez une map de noms avec leurs âges, vous pouvez trouver l’âge d’une personne en utilisant son nom comme clé.

Méthodes fonctionnelles courantes (map, filter, reduce)

Les collections en Kotlin ont des méthodes fonctionnelles courantes telles que map, filter et reduce, qui permettent de transformer, filtrer et réduire une collection.

- La méthode map permet de transformer chaque élément de la collection en un autre élément.

Par exemple, si vous avez une liste de temparatures en Celsius, vous pouvez utiliser la méthode map pour la transformer en Fahrenheit :

```
val tempsCelsius = listOf(20, 25, 30, 35)
val tempsFahrenheit = tempsCelsius.map { celsius -> celsius * 9 / 5 + 32 }
println(tempsFahrenheit) // affiche [68, 77, 86, 95]
```

- La méthode filter permet de filtrer les éléments de la collection qui répondent à un certain critère.

Par exemple, si vous avez une liste de noms, vous pouvez utiliser la méthode filter pour trouver tous les noms commençant par la lettre “A” :

```
val listeNoms = listOf("Luc", "Anne", "Pierre", "Nicolas")
val listeNomsA = listeNoms.filter { nom -> nom.startsWith("A") }
println(listeNomsA) // affiche [Anne]
```

- La méthode reduce permet de réduire une collection à une seule valeur en appliquant une opération itérative à chaque élément de la collection.

Par exemple, si vous avez une liste de nombres, vous pouvez utiliser la méthode reduce pour trouver leur somme :

```
val listeNombres = listOf(1, 2, 3, 4, 5)
val sommeNombres = listeNombres.reduce { somme, nombre -> somme + nombre }
println(sommeNombres) // affiche 15
```

Opérations de partition et de groupement

Les collections en Kotlin ont également des méthodes permettant de partitionner ou de grouper les éléments selon certains critères.

- La méthode partition permet de diviser la collection en deux parties : ceux qui répondent à un certain critère et ceux qui ne le répondent pas.

Par exemple, si vous avez une liste de nombres, vous pouvez utiliser la méthode partition pour trouver tous les nombres pairs et impairs :

```
val listeNombres = listOf(1, 2, 3, 4, 5)
val (nombresPairs, nombresImpairs) = listeNombres.partition { nombre -> nombre % 2 == 0 }
println(nombresPairs) // affiche [2, 4]
println(nombresImpairs) // affiche [1, 3, 5]
```

- La méthode groupBy permet de regrouper les éléments en fonction de certains critères.

Par exemple, si vous avez une liste de noms, vous pouvez utiliser la méthode groupBy pour regrouper les noms par leur première lettre :

```
val listeNoms = listOf("Luc", "Anne", "Pierre", "Nicolas")
val nomsParPremiereLettre = listeNoms.groupBy { nom -> nom.first() }
println(nomsParPremiereLettre) // affiche {L=[Luc], A=[Anne], P=[Pierre], N=[Nicolas]}
```

Opérations sur les séquences

Les séquences en Kotlin permettent d’effectuer des transformations sur une collection sans créer de nouvelles collections intermédiaires.

Cela permet des gains de performance dans certaines situations.

Par exemple, si vous avez une liste de nombres, vous pouvez utiliser une séquence pour trouver la somme de tous les nombres pairs :

```
val listeNombres = listOf(1, 2, 3, 4, 5)
val sommeNombresPairs = listeNombres.asSequence().filter { nombre -> nombre % 2 == 0 }.sum
()
println(sommeNombresPairs) // affiche 6
```

Exemple basique

Supposons que vous ayez un tableau de nombres et que vous vouliez trouver la somme de tous les nombres pairs.

Voici un exemple de code en Kotlin qui effectue cette tâche :

```
fun sommeNombresPairs(tableauNombres: List<Int>): Int {
    return tableauNombres.filter { nombre -> nombre % 2 == 0 }.sum()
}

fun main() {
    val tableauNombres = listOf(1, 2, 3, 4, 5)
    val sommeNombresPairs = sommeNombresPairs(tableauNombres)
    println(sommeNombresPairs) // affiche 6
}
```

Dans cet exemple, nous avons créé une fonction `sommeNombresPairs` qui prend un tableau de nombres et renvoie la somme de tous les nombres pairs.

Nous avons également créé une fonction `main` qui appelle cette fonction avec un tableau de nombres, puis affiche le résultat dans la console.

Notez que nous avons utilisé les méthodes `filter` et `sum` pour effectuer les opérations sur le tableau de nombres.

Ces méthodes font partie des méthodes fonctionnelles courantes disponibles dans les collections en Kotlin.

Extension de fonctions et propriétés

Les fonctions et propriétés sont des éléments fondamentaux de tout programme Kotlin.

Mais parfois, il peut être utile d’étendre une fonction ou une propriété déjà existante pour ajouter une nouvelle fonctionnalité ou en modifier le comportement.

Définition et syntaxe des fonctions d’extension

Une fonction d’extension est une fonction qui ajoute une fonctionnalité à une classe, même si cette classe n’a pas été conçue pour cette fonctionnalité.

La syntaxe pour définir une fonction d’extension est la suivante :

```
fun NomméDeLaFonctionDeExtension(receveur: LeTypeCible).

NomDeLaFonction(params) :Retour { ... }
```

- `fun` : mot-clé pour définir une fonction.
- `NomDeLaFonctionDeExtension` : nom de la fonction d'extension.
- `receveur` : le type cible auquel cette extension est applicable.
- `NomDeLaMéthode` : nom de la méthode d'extension.
- `params` : paramètres de la méthode d'extension.
- `Retour` : type de retour de la méthode d'extension (facultatif).

Par exemple, pour ajouter une fonction `isEven()` à la classe `Int` qui vérifie si un nombre est pair, voici la représentation en Kotlin :

```
fun Int.isEven() : Boolean = this % 2 == 0
```

Extension de propriétés

Il est également possible d'étendre une propriété déjà existante dans une classe.

La syntaxe pour définir une propriété d'extension est la suivante :

```
var LeTypeCible.nomDeLaPropriété : TypeDeLaPropriété
    get() = ...
    set(value) { ... }
```

- `var` : mot-clé pour définir une propriété.
- `LeTypeCible` : le type cible auquel cette extension est applicable.
- `NomDeLaPropriété` : nom de la propriété d'extension.
- `TypeDeLaPropriété` : type de la propriété d'extension.
- `get()` : méthode getter pour la propriété d'extension.
- `set(value)` : méthode setter pour la propriété d'extension.

Par exemple, pour ajouter une propriété `isPositive` à la classe `Int` qui vérifie si un nombre est positif ou non, voici la représentation en Kotlin :

```
val Int.isPositive: Boolean
    get() = this > 0
```

Utilisation des fonctions et propriétés d'extension dans la pratique

L'utilisation des fonctions et propriétés d'extension permet de simplifier le code et de le rendre plus lisible.

Par exemple, pour utiliser la fonction `isEven()` et la propriété `isPositive` que nous avons définies ci-dessus, voici un exemple de code :

```
fun main() {
    val a = 10
    val b = -5

    println("$a est pair : ${a.isEven()}")
    println("$b est positif : ${b.isPositive}")
}
```

Cet exemple affichera :

```
10 est pair : true
-5 est positif : false
```

Conclusion

Les fonctions et propriétés d'extension sont des outils essentiels en programmation Kotlin.

Ils permettent d'étendre facilement les fonctions et propriétés des classes existantes, tout en respectant les principes de la programmation fonctionnelle.

L’utilisation des fonctions et propriétés d’extension peut grandement améliorer la clarté et la lisibilité du code, et faciliter la maintenance et l’évolution du développement.

Gestion des erreurs et exceptions

Types nullables et opérateur Elvis

Les types nullables permettent de gérer les valeurs qui peuvent être nulles.

En Kotlin, un type peut être déclaré comme nullable en ajoutant un point d’interrogation à la fin du type.

Par exemple, `String?` est un type pouvant contenir soit une chaîne de caractères soit la valeur nulle.

L’opérateur Elvis `?:` peut être utilisé pour fournir une valeur par défaut si une valeur nullable est nulle.

Par exemple, si on a un objet `text: String?` qui peut être nul, on peut utiliser l’opérateur Elvis pour afficher une chaîne de caractères par défaut si `text` est nul :

```
val length = text?.length ?: -1
```

Si `text` est nul, la valeur -1 est utilisée comme valeur par défaut.

Dans ce cas, `length` vaut -1.

Syntaxe et utilisation des types nullables

La syntaxe pour déclarer un type nullable est d’ajouter un point d’interrogation à la fin du type.

Par exemple, `String?` est un type pouvant contenir soit une chaîne de caractères soit la valeur nulle.

Pour utiliser un type nullable, on peut utiliser des expressions conditionnelles pour vérifier si la valeur est nulle ou non.

Par exemple :

```
val text: String? = null
if (text != null) {
    print(text.length)
}
```

Dans cet exemple, on vérifie si `text` est nul ou non avant d'afficher sa longueur.

Si `text` est nul, l'affichage ne sera pas effectué.

Opérateur Elvis et expressions conditionnelles

L'opérateur Elvis `?:` peut être utilisé pour fournir une valeur par défaut si une valeur nullable est nulle.

Par exemple :

```
val length = text?.length ?: -1
```

Si `text` est nul, la valeur -1 est utilisée comme valeur par défaut.

Les opérations sûres permettent de simplifier les expressions conditionnelles pour les types nullables.

Par exemple :

```
val text: String? = null
val length = text?.length
```

Dans cet exemple, `length` vaut `null` si `text` est nul.

Autrement, `length` vaut la longueur de `text`.

Cette opération est sûre car elle ne produit pas d'erreur si `text` est nul.

Exemple basique

Imaginons que nous avons une fonction qui lit un numéro de téléphone à partir d'une entrée utilisateur.

Cette entrée peut être nulle ou ne pas respecter le format d'un numéro de téléphone.

Dans ce cas, on peut utiliser des types nullables et l'opérateur Elvis pour fournir une valeur par défaut si la saisie utilisateur n'est pas valide :

```
fun readPhoneNumber(input: String?): String {
    return input?.replace("[^\\d]".toRegex(), "") ?: "Numéro de téléphone invalide"
}
```

Cette fonction prend en paramètre une chaîne de caractères `input` pouvant être nulle.

Elle retourne un numéro de téléphone sous la forme d'une chaîne de caractères où tous les caractères autres que les chiffres ont été supprimés.

Si `input` est nul, la chaîne `"Numéro de téléphone invalide"` est retournée comme valeur par défaut.

Gestion des exceptions

En informatique, les exceptions sont des erreurs qui se produisent lors de l'exécution d'un programme.

Kotlin permet de gérer ces exceptions à travers les blocs `try`, `catch` et `finally`.

Syntaxe et utilisation de try, catch et finally

La syntaxe générale de la gestion d'exceptions en Kotlin est la suivante:

```
try {
   // code susceptible de générer une exception
}
catch (e: Exception) {
   // traitement de l'exception
}
finally {
   // code exécuté quelle que soit l'issue du bloc try
}
```

Le bloc `try` contient le code susceptible de générer une exception.

Si une exception est générée, la première clause `catch` correspondante est exécutée.

La clause `finally` est exécutée quelle que soit l'issue du bloc `try`.

Gestion des erreurs multiples

Il est possible de gérer plusieurs types d'exceptions dans le même bloc `try`, en les spécifiant dans des clauses `catch` différentes:

```
try {
   // code susceptible de générer des exceptions
}
catch (e: IOException) {
   // traitement de l'exception de type IOException
}
catch (e: SQLException) {
   // traitement de l'exception de type SQLException
}
finally {
   // code exécuté quelle que soit l'issue du bloc try
}
```

Exemples d'utilisation de la gestion d'exceptions

Voici un exemple basique de gestion d'exception en Kotlin, avec la génération d'une exception de type `java.lang.ArithmeticException`:

```
fun diviser(a: Int, b: Int): Int {
   return a / b
}

fun main() {
   try {
      val resultat = diviser(4, 0)
      println("Le résultat est: $resultat")
   }
   catch (e: ArithmeticException) {
      println("Erreur: Division par zéro!")
   }
   finally {
      println("Fin du programme")
   }
}
```

Dans cet exemple, la fonction `diviser` génère une exception si le deuxième argument est égal à 0. Dans le bloc `try` de la fonction `main`, nous appelons cette fonction avec des arguments qui génèrent une telle exception.

Le bloc `catch` correspondant est exécuté, et un message d'erreur est affiché.

Le bloc `finally` est exécuté pour terminer proprement le programme.

Conclusion

En utilisant les blocs `try`, `catch` et `finally`, il est possible de gérer efficacement les exceptions en Kotlin.

En cas d'exception, le programme peut les traiter intelligemment et éviter des plantages ou comportements étranges.

La gestion d'exceptions est donc une compétence essentielle pour tout développeur Kotlin.

Lancer et propager des exceptions personnalisées

Dans Kotlin, nous pouvons créer des exceptions personnalisées en définissant une classe qui étend la classe `Exception`.

Voici un exemple d'une telle classe :

```
class MonException(message: String) : Exception(message)
```

Dans cet exemple, nous avons créé une classe `MonException` qui hérite de la classe `Exception`.

Le constructeur de cette classe prend en paramètre un message d'erreur qui sera affiché lorsque l'exception sera lancée.

Création d'exceptions personnalisées

Pour lancer une exception personnalisée, nous pouvons utiliser l'opérateur `throw`.

```
fun calculerDivison(a: Int, b: Int): Int {
    if (b == 0) {
        throw MonException("Division par zéro impossible")
    }
    return a / b
}
```

Dans cet exemple, nous avons défini une fonction qui effectue une division.

Si le dénominateur est égal à zéro, nous lançons une exception personnalisée de type `MonException`.

Propagation et gestion d'exceptions personnalisées

Une exception peut être propagée à travers les appels de méthode en utilisant le mot-clé `throws`.

```
fun fonction1() {
    fonction2()
}

fun fonction2() {
    fonction3()
}

fun fonction3() throws MonException {
    throw MonException("Une exception s'est produite")
}
```

Dans cet exemple, la fonction `fonction3` lance une exception personnalisée.

Cette exception est propagée à travers les appels de méthode jusqu'à la fonction `fonction1`.

La déclaration `throws MonException` dans la fonction `fonction3` indique que cette fonction peut lancer une exception de type `MonException`.

Enfin, voici un exemple basique qui illustre l'utilisation de nos connaissances sur la gestion des erreurs et exceptions en Kotlin.

```
fun main() {
    try {
        val resultat = calculerDivison(10, 0)
    } catch (e: MonException) {
        println(e.message)
    }
}
```

Dans cet exemple, nous avons appelé la fonction `calculerDivison` en lui passant les valeurs 10 et 0 en tant que paramètres.

Comme le dénominateur est égal à zéro, l'exception de type `MonException` est levée et capturée dans le bloc `catch`.

Nous imprimons simplement le message d'erreur dans la console.

Kotlin et Android

Intégration de Kotlin dans un projet Android

L'intégration de Kotlin dans un projet Android est assez simple.

Pour cela, vous devez configurer votre environnement de développement.

Cela signifie que vous devez avoir la dernière version d'Android Studio et la version adaptée à Kotlin.

Pour créer un nouveau projet Android avec Kotlin, vous devez suivre les étapes suivantes:

- Ouvrez Android Studio
- Cliquez sur "File" puis "New" et choisissez "New Project"
- Saisissez le nom de votre projet, le nom de votre entreprise et le chemin d'accès à votre projet.

Cliquez ensuite sur "Next".

- Choisissez les options de votre projet: la langue utilisée, la version minimale de l'OS Android, le modèle de prise en charge de l'activité.

Cliquez ensuite sur "Next".

- Choisissez le modèle de base de votre projet.

Cliquez sur "Finish" lorsque vous avez terminé.

Vous pouvez maintenant convertir vos fichiers Java en Kotlin.

Pour ce faire, vous devez:

- Cliquez avec le bouton droit de la souris sur le fichier Java que vous souhaitez convertir.
- Choisissez l'option "Convert Java File to Kotlin File".
- Android Studio ouvrira une nouvelle fenêtre contenant le code Java converti en Kotlin.

Interopérabilité avec Java

Kotlin est compatible avec Java.

Cela signifie que vous pouvez écrire du code Java dans un projet Kotlin ou du code Kotlin dans un projet Java.

Vous pouvez également utiliser des bibliothèques Java dans un projet Kotlin ou des bibliothèques Kotlin dans un projet Java.

Exemple basique:

Voici un exemple simple montrant l'utilisation de Kotlin dans un projet Android:

- Créez une nouvelle application Android en utilisant Kotlin.
- Ajoutez un bouton à l'interface utilisateur.
- Écrivez un code Kotlin pour prendre en charge le clic sur le bouton.
- Exécutez l'application sur votre téléphone ou dans un émulateur.

Ce sont là les bases de l'utilisation de Kotlin dans un projet Android.

Vous pouvez maintenant utiliser cette langue pour écrire du code robuste et facile à maintenir pour vos applications Android.

Utilisation de Kotlin avec les composants Android (Activity, Fragment, …)

Kotlin est un langage de programmation moderne, concis et sécurisé qui peut être utilisé avec Android pour améliorer la qualité et la productivité du développement.

Les composants Android tels que les activités et les fragments peuvent être implémentés en utilisant Kotlin, ce qui permet une réduction significative des lignes de code et une amélioration de la lisibilité.

Syntaxe et utilisation des coroutines dans Android

Les coroutines sont un moyen léger et efficace pour exécuter du code de manière asynchrone dans Android.

Avec Kotlin, il est facile d'utiliser les coroutines pour gérer les événements et les cycles de vie des applications Android.

Gestion des événements et des cycles de vie

Les événements tels que les clics de boutons peuvent être gérés en utilisant les coroutines, ce qui permet d'effectuer des tâches en arrière-plan tout en maintenant une expérience utilisateur fluide et réactive.

La gestion du cycle de vie est également plus facile avec Kotlin, car il offre une syntaxe concise pour lier les composants Android ensemble tout en garantissant que les ressources sont libérées correctement.

Exemples d'utilisation de Kotlin dans les composants Android

Voici un exemple de code de l'utilisation de Kotlin pour implémenter une activité et un fragment dans Android :

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        // Utilisation de coroutines pour un traitement asynchrone
        GlobalScope.launch {
            // Effectuer une tâche en arrière-plan
            // ...
        }
    }
}

class ExampleFragment : Fragment() {

    override fun onCreateView(
            inflater: LayoutInflater,
            container: ViewGroup?,
            savedInstanceState: Bundle?
```

```
    ): View? {
        // Utilisation de Kotlin pour une mise en page plus lisible
        return inflater.inflate(R.layout.example_fragment, container, false)
    }
}
```

Ces exemples montrent comment Kotlin peut être utilisé pour créer des composants Android de manière concise et efficace.

Kotlin et les librairies Android (Anko, Ktor, …)

Les librairies Android sont des outils très utiles pour le développement d'applications Android en Kotlin.

Parmi les librairies populaires, on retrouve Anko et Ktor.

Présentation des librairies populaires

Utilisation d'Anko pour le développement d'interfaces utilisateur

Anko est une librairie qui permet de créer des interfaces utilisateur rapidement et facilement en Kotlin.

Elle permet de définir les vues en utilisant une syntaxe plus concise que XML, et fournit également des fonctions pour gérer les événements de vues.

Voici un exemple d'utilisation d'Anko pour créer une vue :

```
val myView = verticalLayout {
    textView("Hello, World!")
    button("Click me!") {
        onClick { /* do something */ }
    }
}
```

Utilisation de Ktor pour les requêtes réseau

Ktor est une librairie qui permet de gérer les requêtes réseau en Kotlin.

Elle fournit une interface fluide pour créer des serveurs HTTP et gérer les clients HTTP.

Voici un exemple d’utilisation de Ktor pour envoyer une requête GET :

```
val response = HttpClient().use { client ->
    client.get<String>(url)
}
```

À la fin de chaque chapitre, voici un exemple basique pour illustrer les propos :

```
// Exemple d'utilisation d'Anko pour créer une vue contenant un bouton
val myView = verticalLayout {
    button("Click me!") {
        onClick { toast("Button clicked!") }
    }
}
```

Bonnes pratiques et conventions de codage

Lors de l’utilisation de Kotlin pour le développement Android, il est important de respecter certaines bonnes pratiques et conventions de codage pour garantir la qualité et la cohérence tout au long du projet.

Les bonnes pratiques comprennent :

Style de codage recommandé

Il est important d’utiliser un style de codage recommandé pour Kotlin qui consiste à :

- Utiliser des noms de variable et de fonction significatifs
- Éviter les changements d’état implicites
- Éviter les valeurs par défaut inutiles
- Éviter les nombres magiques
- Utiliser les fonctions déclaratives plutôt que les boucles
- Éviter les règles de complexité cyclomatique

Utilisation des commentaires et de la documentation

Les commentaires et la documentation peuvent aider à garantir que le code est clair et facile à comprendre pour les autres membres de l'équipe de développement.

Cela peut également aider à documenter les processus et les fonctionnalités.

Il est recommandé d'utiliser des commentaires javadoc pour documenter les classes, les fonctions et les variables.

Tests unitaires et d'intégration en Kotlin

Les tests sont une partie essentielle de tout projet de développement, et Kotlin facilite l'écriture de tests.

Les tests doivent être effectués à deux niveaux différents : les tests unitaires et les tests d'intégration.

Les tests unitaires sont utilisés pour tester une fonction ou une méthode individuelle, tandis que les tests d'intégration sont utilisés pour tester une fonctionnalité complète.

Il est recommandé d'utiliser des tests unitaires à chaque fois que cela est possible pour garantir la qualité du code.

Exemple basique

Voici un exemple basique de code Kotlin qui utilise certaines des bonnes pratiques et conventions de codage recommandées pour Kotlin :

```
// Un exemple basique de fonction Kotlin
fun add(a: Int, b: Int): Int {
    return a + b
}

// Un exemple de fonction Kotlin nullable
fun checkIsNull(value: String?) {
  checkNotNull(value)
  print("La valeur n'est pas nulle")
}
```

Dans l'exemple ci-dessus, nous avons une fonction "add" qui ajoute deux nombres entiers.

Cette fonction utilise des noms de variable significatifs et est déclarative plutôt qu’utilisant une boucle.

La deuxième fonction “checkIsNull” utilise une variable nullable avec l’opérateur “?” et utilise la fonction “checkNotNull” pour vérifier si la variable n’est pas nulle.

Cela permet de garantir que la valeur utilisée n’est pas nulle et évite les erreurs de type “NullPointerException”.