Kotlin : la programmation orientée objet

Kotlin

* Introduction

Kotlin est un langage orienté objet et propose donc :

* la création de classes avec son lot de modificateurs,
* la création d'interfaces,
* la mise en place d'héritages,
* le principe de polymorphisme,
* des collections.

Tous ces concepts seront traités dans ce chapitre ainsi qu'un point sur les expressions lambda.

* Les classes
* Présentation

La classe est l'élément central de la programmation orientée objet. Elle permet de définir et de créer des objets.

Une classe peut contenir :

* des constructeurs,
* des propriétés,
* des fonctions,
* des classes imbriquées,
* des classes internes,
* des déclarations d'objets.

Nous allons examiner chacun de ces éléments.

* Syntaxe

Voici la syntaxe permettant de créer une classe :

class <NomDeLaClasse> [constructeur][([paramètres])]

* Déclaration d'une classe

Le mot-clé class permet de déclarer une classe.

Exemple de déclaration d'une classe vide

class Animal

Cet exemple montre la déclaration d'une classe vide. Les accolades permettant de définir le bloc d'instructions ne sont pas obligatoires.

Exemple de déclaration d'une classe avec un bloc d'instructions vide

class Animal   
{   
       
}

Le bloc d'instructions d'une classe se situe entre les accolades.

Les deux exemples ci-dessus produisent exactement la même chose.

* Création et utilisation d'instances d'une classe

L'appel du constructeur permet de créer un objet/instance d'une classe.

Exemples de création d'objets de type Personne

val personne1 = Personne()   
   
val personne2:Personne = Personne()   
   
var personne3:Personne = Personne()

Dans cet exemple, trois objets de type Personne ont été créés.

* Les modificateurs de classe

Un modificateur de classe permet d'attribuer des caractéristiques spécifiques à une classe. Par défaut, une classe a les caractéristiques des modificateurs public et final. Une classe peut avoir les modificateurs suivants : final, open, abstract, public, internal, protected et private. Certains modificateurs sont incompatibles de par leur nature.

* Modificateur public

Le modificateur public est un modificateur de visibilité. La spécificité du modificateur public est que la classe l'utilisant est disponible et utilisable depuis n'importe quel endroit. Par défaut, les classes sont public.

* Modificateur internal

Le modificateur internal est un modificateur de visibilité. La spécificité du modificateur internal est que la classe l'utilisant est uniquement disponible et utilisable depuis n'importe quel autre fichier de son module. Un module est un ensemble de fichiers compilés ensemble.

* Modificateur protected

Le modificateur protected est un modificateur de visibilité. La spécificité du modificateur protected est que la classe l'utilisant doit être interne à une autre classe devant être open ; l'autre spécificité est que la classe l'utilisant est disponible pour les autres classes filles de la classe parente.

* Modificateur private

Le modificateur private est un modificateur de visibilité. La spécificité de ce modificateur est que la classe l'utilisant n'est disponible que depuis le fichier dans lequel elle est déclarée.

* Modificateur final

La spécificité d'une classe final est qu'aucune autre classe ne peut en hériter. Par défaut, les classes sont final. Le modificateur final est incompatible avec le modificateur open. Une classe final ne peut avoir de membres dont le modificateur est open.

* Modificateur open

La spécificité d'une classe open est que les autres classes sont autorisées à en hériter. Le modificateur open est incompatible avec le modificateur final.

* Modificateur abstract

La spécificité d'une classe abstract est qu'elle n'est pas instanciable, sa raison d'être est de proposer un héritage aux autres classes.

* Pseudo modificateur nested (imbriqué)

Les classes déclarées à l'intérieur d'une autre classe sont par défaut considérées comme nested, c’est-à-dire qu'elle est interne, mais qu'elle ne peut pas accéder aux éléments de la classe englobante. Le mot-clé nested n'existe pas.

* Modificateur inner (interne)

Par défaut, une classe interne ne peut pas accéder aux éléments de sa classe englobante. Le modificateur de classe inner permet à une classe interne d'accéder aux éléments de sa classe englobante.

* Modificateur sealed (fermé)

Une classe sealed a les mêmes particularités qu'une classe abstraite à la différence qu'elle restreint les classes pouvant en hériter. Seules les classes se trouvant dans le même fichier peuvent en hériter.

* Les constructeurs
* Constructeur principal

Définition

Le mot-clé constructor permet de déclarer le constructeur. Il se positionne juste après le nom de la classe. Le mot-clé constructor est facultatif. Pour chaque paramètre du constructeur, une propriété peut être créée. Si le paramètre est de type var, alors la propriété sera en lecture et écriture ; si le paramètre est de type val, alors la propriété sera en lecture seule ; si le paramètre n'est pas spécifié avec un val ou un var, alors le paramètre pourra être utilisé dans un bloc d'initialisation ou lors de l'initialisation d'une propriété, mais aucune propriété ne sera créée pour celui-ci.

Exemple de définition d'un constructeur   
par défaut avec le mot-clé constructor

class Animal constructor(var nom:String, val espece:String)   
{   
   
}

La classe animal contient un constructeur avec deux paramètres : nom de type var et espece de type val. Cela implique que la classe Animal contient une propriété nom en lecture et écriture, ainsi qu'une propriété espece en lecture seule.

Exemple de définition d'un constructeur   
par défaut sans le mot-clé constructor

class Animal (var nom:String, val espece:String)   
{   
   
}

Le mot-clé constructor est facultatif.

Exemple de définition d'un constructeur   
sans création implicite de propriétés

class Animal (nom:String, espece:String)   
{   
   
}

La classe animal contient un constructeur avec deux paramètres : nom et espece. val et var n'ont pas été spécifiés pour ces deux paramètres. Cela implique que la classe Animal ne contient pas de propriété nom ni espece. Les deux paramètres pourront être utilisés dans un bloc d'initialisation et lors de l'initialisation d'une propriété.

Visibilité du constructeur principal

Par défaut, le constructeur principal a une visibilité public. Pour modifier sa visibilité il faut utiliser le mot-clé constructor et le préfixer avec le modificateur de visibilité voulu.

Exemple d'un constructeur avec une visibilité private

class Animal private constructor() {   
   
}

Dans cet exemple, le constructeur utilise le modificateur private. Le constructeur ne pourra donc pas être utilisé hors de sa classe.

Constructeur par défaut

Un constructeur par défaut n'attendant aucun paramètre d'entrée est créé lorsqu'aucune déclaration de constructeur principal n'est faite.

Exemple de classe avec un constructeur par défaut

class Animal   
{   
      
}

Cette classe détient un constructeur par défaut qui n'attend aucun paramètre et ne contient aucune instruction.

Paramètres par défaut

Le constructeur peut définir des valeurs par défaut pour ses paramètres. Il suffit de définir ces valeurs directement dans la définition du constructeur.

Exemple de valeurs par défaut pour un constructeur

class Animal constructor(var nom:String = "Sans nom", val   
espece:String = "Sans espèce") {   
   
}

Dans cet exemple, le constructeur attend deux paramètres. Si au moment de l'utilisation de ce constructeur aucune valeur n'est passée aux paramètres, alors ils auront les valeurs : "Sans nom" et "Sans espèce".

* Bloc d'initialisation

Présentation

Le mot-clé init permet de définir un bloc d'instructions exécuté lors de l'initialisation d'un objet de la classe. Init peut être nommé "bloc d'initialisation". Cela permet de pallier le fait que le constructeur principal ne peut contenir de code.

Exemple de code exécuté lors de l'initialisation d'un objet de type Animal

class Animal (var nom:String, val espece:String)   
{   
    init   
    {   
        print("Un objet Animal vient d'être créé")   
    }   
}

Lors de l'initialisation d'un objet Animal " Un objet Animal vient d'être créé " sera affiché.

Plusieurs blocs d'initialisation possibles

Il est possible de définir plusieurs blocs d'initialisation. Ils seront exécutés dans le même ordre que celui dans lequel ils ont été définis.

Exemple de définition de plusieurs blocs d'initialisation

class Animal (var nom:String, val espece:String) {   
    init {   
        println("Un objet Animal vient d'être créé")   
    }   
   
    init {   
        println("L'initialisation se fait en plusieurs fois")   
    }   
}   
   
Résultat :   
   
Un objet Animal vient d'être créé   
   
L'initialisation se fait en plusieurs fois

Dans cet exemple, la classe Animal contient deux blocs init. Ils sont exécutés dans l'ordre dans lequel ils ont été définis.

* Constructeurs secondaires

Constructor

Pour définir un constructeur secondaire il faut utiliser le mot-clé constructor à l'intérieur du bloc d'instructions de la classe. Contrairement au constructeur principal, un constructeur secondaire ne crée pas de propriété implicitement.

Exemple de définition d'un second constructeur

class Animal (var nom:String, val espece:String)   
{   
    init   
    {   
        println("Un objet Animal vient d'être créé")   
    }   
   
    constructor(nom:String,espece:String,poid:Float):this(nom,espece)   
    {   
        println("Instruction du second constructeur")   
    }   
}

Dans cet exemple, le bloc init sera exécuté avant le constructeur secondaire.

Il est possible de définir plusieurs "constructeurs secondaires".

Chaînage de constructeurs

Le chaînage des constructeurs est obligatoire. Il faut voir cela comme une façon d'imposer une bonne pratique et ainsi de réduire au maximum le code redondant. Les constructeurs secondaires doivent donc obligatoirement faire référence au constructeur principal. Cela se fait grâce au mot-clé :this() derrière la définition du constructeur secondaire.

Exemple de définition d'un constructeur secondaire

constructor(nom:String,espece:String,poid:Float):this(nom,espece)

Dans cet exemple, le constructeur secondaire fait appel au constructeur de la classe mère grâce au mot-clé this.

Constructeur secondaire et bloc init

Quel que soit le nombre de blocs init et leur position, ils s'exécutent toujours avant le constructeur secondaire.

Exemple avec plusieurs blocs init et un constructeur secondaire

class Animal (var nom:String, val espece:String) {   
    init {   
        println("Un objet Animal vient d'être créé")   
    }   
   
    constructor(nom:String,espece:String,poid:Float):this(nom,espece)   
    {   
        println("Instruction du second constructeur")   
    }   
   
    init {   
        println("L'initialisation se fait en plusieurs fois")   
    }   
}

Même s'il y a un bloc init situé en dessous du constructeur secondaire, il sera tout de même exécuté avant le constructeur secondaire.

Résultat de l'exemple ci-dessus

Un objet Animal vient d'être créé   
   
L'initialisation se fait en plusieurs fois   
   
Instruction du second constructeur

* Les champs et les propriétés
* Présentation

Pour définir les spécificités des données représentées par les classes, il est nécessaire de bien savoir définir les champs et les propriétés d'une classe. Avec Kotlin, la création d'un champ créée automatiquement une propriété pour le champ. Pour rappel, le champ (field) permet de stocker la valeur, la propriété permet de personnaliser la lecture et l'écriture du champ. Ainsi, un champ défini avec var aura une propriété en lecture et écriture et un champ défini avec val aura seulement une propriété en lecture étant donné que ce champ est immuable.

* Syntaxe

Voici la syntaxe permettant de créer une propriété :

[modificateur] var <nomDeLaPropriete>[:<Type>] [=<valeur>]   
   
[modificateur][<getter>]   
   
[modificateur][<setter>]

* Création de propriétés via le constructeur

Lorsqu'un paramètre est défini dans le constructeur principal d'une classe, cela implique qu'il y a une propriété du même nom et du même type que ce paramètre à l'intérieur de la classe. Si le paramètre est défini avec val, alors la propriété sera en lecture seule ; si le paramètre est défini avec var, alors la propriété sera en lecture et écriture. Pour utiliser une propriété, il suffit de faire référence à son nom.

Exemple avec une classe simple

class Animal (var nom:String, val espece:String) {   
   
}

Il y a deux paramètres au sein du constructeur. Cela implique que la classe Animal contient une propriété nom de type String et une propriété espece de type espece.

Exemple d'utilisation des propriétés implicitement créées

fun main(args: Array<String>) {   
    var lion: Animal = Animal("Lion", "Fauve")   
    lion.nom = "Lionne"   
    println(lion.espece)   
}

Un objet de type Lion est créé. Il est possible de modifier le champ nom, car la propriété est en lecture et écriture, et il est seulement possible de consulter la valeur de champ espèce, car la propriété est en lecture seule.

* Création de propriétés dans la classe

Usage

Pour définir une nouvelle propriété dans une classe, il suffit de déclarer une variable à l'intérieur de son bloc d'instructions. Si cette variable est de type var, alors la propriété sera accessible en lecture et écriture ; si cette variable est de type val, alors la propriété sera accessible en lecture seule.

Exemple d'une classe avec deux propriétés

class Animal ()   
{   
    var nom:String = ""   
    val espece:String = "xxx"   
}

Dans cet exemple, la classe Animal aura deux propriétés, une en lecture/écriture et une en lecture seule.

Exemple d'utilisation de la classe définie dans l'exemple ci-dessus

fun main(args: Array<String>) {   
    var lion: Animal = Animal()   
    lion.nom = "Lionne"   
    println(lion.espece)   
    println(lion.nom)   
}   
   
   
Résultat :   
   
xxx   
Lionne

Dans cet exemple, on ne peut que lire la valeur du champ espece, il n'est pas possible de la modifier.

Les propriétés doivent obligatoirement initialiser les champs au moment de leur déclaration.

* Création de propriétés personnalisées

Personnaliser l'accesseur de lecture d'une propriété (le getter)

Le getter est l'accesseur permettant de lire la valeur d'une propriété. Lorsque l'on accède à une propriété, le getter est appelé.

Exemple d'utilisation d'un getter

class Animal   
{   
    var nom:String = "Aucun nom"   
}   
   
fun main(args: Array<String>)    
{   
    var zebre: Animal = Animal() //Création d'un objet Animal   
   
    print(zebre.nom) // Appel du getter   
   
}

Dans cet exemple, il y a une classe Animal qui contient une propriété nom. Dans la fonction main, un objet nommé zebre de type Animal est créé. La dernière ligne d'instruction affiche la propriété nom de l'objet zebre. Il y a un accès à la propriété nom et donc un appel implicite au getter de nom qui retournera Aucun nom.

Il est possible de personnaliser ce getter en utilisant le mot-clé get() en dessous de la définition de la propriété et le mot-clé field (aussi appelé backing field) qui représente la valeur de la propriété.

Exemple de personnalisation d'un getter

class Animal   
{   
    var nom:String = "Aucun nom"   
        get()=field.toUpperCase()   
}

Le getter fournira la valeur de la propriété nom en majuscules.

Si plusieurs instructions sont nécessaires pour personnaliser le getter, alors il faut définir un bloc d'instructions incluants un return à la fin.

Exemple de personnalisation sur plusieurs lignes

class Animal   
{   
    var nom:String = "Aucun nom"   
        get(){   
            field = field.trim()   
            return field.toUpperCase()   
        }   
}

Le getter fournira la valeur de la propriété sans espace au début ni à la fin et en majuscules.

Personnaliser l'accesseur d'écriture (le setter)

Le setter est l'accesseur permettant de modifier la valeur d'une propriété de type var. Lorsque l'on modifie la valeur d'une propriété, le setter est appelé.

Exemple d'utilisation d'un setter

class Animal   
{   
    var nom:String = "Aucun nom"   
}   
   
fun main(args: Array<String>)   
{   
    var zebre: Animal = Animal() //Création d'un objet Animal   
     zebre.nom = "Hypolite" // Appel du setter   
}

Dans cet exemple, il y a une classe Animal qui contient une propriété nom. Dans la fonction main, un objet nommé zebre de type Animal est créé. La dernière ligne d'instruction modifie le champ nom de l'animal en lui affectant la valeur "Hypolite". Il y a une modification via la propriété nom et donc un appel implicite au setter de nom.

Il est possible de personnaliser un setter en utilisant le mot-clé set(value) en dessous de la définition de la propriété et le mot-clé field qui représente la valeur de la propriété. value représente la valeur avec laquelle on veut affecter la propriété.

Exemple de personnalisation d'un setter

class Animal   
{   
    var nom:String = "Aucun nom"   
        set(value)   
        {   
            field = value.toLowerCase()   
        }   
}

Le setter modifiera le champ afin qu'il soit en minuscules.

Personnaliser les deux accesseurs en même temps

Il est tout à fait possible de personnaliser les deux accesseurs d'une même propriété.

Exemple de personnalisation des deux accesseurs

class Animal   
{   
    var nom:String = "Aucun nom"   
        get() = field.toUpperCase()   
        set(value)   
        {   
            field = value.toLowerCase()   
        }   
}

La valeur de la propriété nom sera toujours enregistrée en minuscules et sera toujours restituée en majuscules.

Visibilité sur un accesseur

Il est possible de modifier la visibilité des accesseurs. En effet, on peut très bien avoir un champ mutable (var) qui ne peut être modifié qu'à l'intérieur de sa propre classe par exemple.

Pour ce faire, il faut mettre le nom du modificateur devant le mot-clé get ou set.

Exemple d'un champ mutable en lecture seule

class Animal   
{   
    var nom:String = "Aucun nom"   
        get() = field.toUpperCase()   
        private set(value)   
        {   
            field = value.toLowerCase()   
        }   
}

Il est possible de définir un modificateur de visibilité à une propriété sans pour autant la redéfinir.

Exemple d'un champ mutable en lecture seule

class Animal   
{   
    var nom:String = "Aucun nom"   
        get() = field.toUpperCase()   
        private set   
}

* Les modificateurs de propriétés
* Modificateur public

Le modificateur public est un modificateur de visibilité. La spécificité du modificateur public est que la propriété l'utilisant est disponible et utilisable depuis n'importe quel endroit. Par défaut, les propriétés sont public.

* Modificateur private

Le modificateur private est un modificateur de visibilité. La spécialité de ce modificateur est que la propriété l'utilisant n'est disponible qu'à l'intérieur de la classe dans laquelle elle est déclarée.

* Modificateur protected

Le modificateur protected est un modificateur de visibilité. La spécialité de ce modificateur est que la propriété l'utilisant n'est disponible qu'à l'intérieur de la classe dans laquelle elle est déclarée, ainsi que dans les classes filles de cette classe.

* Modificateur internal

Le modificateur internal est un modificateur de visibilité. La spécificité du modificateur internal est que la propriété l'utilisant est uniquement disponible et utilisable depuis n'importe quel autre fichier de son module. Un module est un ensemble de fichiers compilés ensemble.

* Modificateur open

Par défaut, une propriété ne peut pas être redéfinie. La spécificité du modificateur open est de donner la possibilité de redéfinir une propriété dans le but de remplacer son code. Le modificateur open ne peut être utilisé dans une classe final.

* Modificateur override

Le modificateur override est utilisé par les fonctions redéfinissant le code d'une autre fonction.

* Modificateur final

La spécificité d'une propriété final est qu'elle ne peut être redéfinie. Le modificateur final est incompatible avec le modificateur open.

* Les fonctions de classes

Les fonctions de classes sont des fonctions dont l'action est étroitement liée au type. Par exemple, pour un type nommé Mathematiques, on retrouvera des fonctions permettant de faire des multiplications, des divisions, etc.

* Syntaxe

Voici la syntaxe permettant de créer une fonction :

[modificateur] fun   
<nomDeLaFonction>([paramètres])[:<TypeDeRetour>]   
   
{   
   
}

* Usage

Il est courant d'avoir plusieurs fonctions dans une classe. Il est possible d'ajouter un modificateur aux fonctions de classes afin de leur attribuer des spécificités.

Exemple d'une fonction de classe

open class Animal   
{   
    var nom:String = "Aucun nom"   
       
    open fun afficherNom()   
    {   
        println(nom)   
    }   
}

Cette classe contient une méthode associée au modificateur open.

* Les modificateurs de fonctions

Les propriétés sont des fonctions implicites. Parmi les modificateurs de fonctions, il y a donc tous les modificateurs de propriétés. Il y a un modificateur supplémentaire pour les fonctions explicites qui est le modificateur infix. Ce modificateur permet d'utiliser une fonction avec la notation infixée (cf. la section Notation infixée dans ce chapitre).

* Les fonctions d'extensions

Présentation

L'objectif des fonctions d'extensions est de pouvoir ajouter des fonctions à des types non nullable et nullable déjà existants depuis n'importe quel fichier Kotlin et sans contrainte, c’est-à-dire qu'il n'est pas nécessaire de créer un nouveau type spécifique, ni de recompiler ou modifier le type que l'on veut étendre. Les fonctions d'extensions sont utilisées comme des fonctions membres.

Syntaxe

Voici la syntaxe des fonctions d'extensions :

fun <TypeReceveur>.<nomFonctionDExtension>([paramètres])   
   
{   
   
}

Usage

Les fonctions d'extensions sont utilisées comme des fonctions membres. Il est possible d'utiliser le mot-clé this dans une fonction d'extension. Le mot-clé this représente l'objet avec lequel la fonction d'extension est appelée, il n'est pas possible de modifier les valeurs de cet objet. La fonction d'extension appelée dépend uniquement du type de l'objet avec lequel la fonction d'extension est appelée. Il n'est pas possible de redéfinir une fonction membre avec une fonction d'extension, cependant la surcharge est possible.

Les fonctions d'extensions sur des types nullable peuvent être appelées même si l'objet avec lequel la fonction d'extension est appelée est égal à null. C'est la raison pour laquelle il est possible d'appeler la fonction toString() sur certains objets (du type Any? par exemple) même s’ils sont null.

Il est possible d'ajouter des propriétés d'extensions à une classe, mais le getter ne pourra retourner qu'une valeur calculée et le setter ne peut être défini. Les propriétés d'extensions sont aussi des fonctions statiques, c'est la raison pour laquelle elles n'ont pas de backing field et qu'il n'est pas possible de définir de setter.

Exemple simple d'une fonction d'extension

fun String.bonjour(prefixe:String):String   
{   
     return prefixe + this;   
}   
   
fun main(args: Array<String>)   
{   
    val unePhrase = "mi casa es tu casa";   
   
    val resultat = unePhrase.bonjour("==>")   
   
    println(resultat)   
}   
   
Résultat :   
   
==>mi casa es tu casa

Dans cet exemple, la classe String contient à présent une fonction d’extension nommée bonjour.

Exemple simple d'une fonction d'extension sur un type nullable

class Personne(var prenom:String)   
   
fun Personne?.afficherPrenom():String   
{   
    if(this == null)   
    {   
        return "L'objet est null"   
    }else{   
        return this.prenom   
    }   
}   
   
   
fun main(args: Array<String>)   
{   
    var unePersonne:Personne? = Personne("Anthony")   
   
    println(unePersonne.afficherPrenom())   
   
    unePersonne = null;   
   
    println(unePersonne.afficherPrenom())   
   
}   
   
Résultat :   
   
Anthony   
L'objet est null

Dans cet exemple, on voit que les fonctions d'extensions sur des types nullable peuvent être appelées même si l'objet avec lequel la fonction d'extension est appelée est égal à null.

Exemple simple d'une propriété d'extension

val String.taille: Int   
    get() = this.length   
   
   
fun main(args: Array<String>)   
{   
    val mot = "coucou"   
   
    println(mot.taille)   
   
}   
   
Résultat :   
   
6

Dans cet exemple, on voit comment ajouter une propriété d'extension à une classe.

Portée des fonctions d'extensions

Une fonction d'extension peut être utilisée dans l'ensemble des fichiers de votre application à condition que ces fichiers aient importé le fichier dans lequel la fonction d'extension a été définie.

* Notation infixée

Présentation

Le mot-clé infix permet d'indiquer qu'une fonction peut être utilisée en notation infixée.

Exemple de notation infixée

5 x 4

Dans cet exemple, nous avons un premier opérande (5), puis une opération (multiplication) et pour finir un nouvel opérande (4). Cette écriture constitue une notation infixée classique.

Syntaxe

Voici la syntaxe pour utiliser une fonction avec la notation infixée :

infix fun <nomDeLaFonction>(<paramètreUnique>)[:<TypeDeRetour>]   
   
{   
   
}

Usage

Une fonction infixée doit obligatoirement :

* être une fonction membre ou une fonction d'extension,
* avoir un seul paramètre ne stockant qu'une seule valeur et n'ayant pas de valeur par défaut.

Lors de l'utilisation, le premier opérande doit être du type de la classe de la fonction infixée. Une fonction infixée peut être utilisée sans la notation infixée.

Exemple d'utilisation de fonctions infixées

class Artiste(nomArtiste:String)   
{   
    var nbPeinture:Int = 0   
   
    val nom:String = nomArtiste   
   
    infix fun newPaint(nombre:Int)   
    {   
        nbPeinture += nombre;   
    }   
   
    infix fun bonjour(prenom:String):String   
    {   
        return "$nom dit bonjour à $prenom"   
    }   
}   
   
fun main(args: Array<String>)   
{   
    var unArtiste = Artiste("Hassoun")   
   
    unArtiste newPaint 5   
    unArtiste.newPaint(2)   
   
    val message = unArtiste.bonjour("Deluxe")   
    val message2 = unArtiste bonjour "Deluxe"   
   
    println(unArtiste.nbPeinture)   
    println(message)   
    println(message2)   
}   
   
   
   
Résultat :   
   
7   
Hassoun dit bonjour à Deluxe   
   
Hassoun dit bonjour à Deluxe

Dans cet exemple, les éléments en gras présentent une utilisation de fonctions infixées.

Lors des appels, les opérateurs arithmétiques sont prioritaires sur les fonctions infixées.

* Les classes objet
* Présentation

Tout projet informatique contient des classes permettant de stocker des données, ce sont des classes que l'on appelle bo, model, pojo, entité, etc. Elles sont là pour stocker et utiliser les informations contenues dans votre source de données.

La structure de ces classes est souvent la même. Kotlin appelle ces classes des data class et propose un mécanisme pour faciliter la création de celles-ci, grâce au mot-clé data.

* Syntaxe

Voici la syntaxe permettant de créer une classe objet :

data class <NomDeLaClasse>(propriété,[propriétés])

* Usage

Grâce au mot-clé data devant la déclaration d'une classe, le compilateur va créer automatiquement les propriétés déclarées dans le constructeur et créer et définir les fonctions equals(), hashCode() et copy().

La déclaration d'une data class doit respecter les points suivants :

* Le constructeur doit contenir au moins un paramètre.
* Tous les paramètres du constructeur doivent être marqués du type val ou var.
* Une data class ne peut pas avoir de modificateur abstract, open, sealed ou inner.

Il est possible de redéfinir les fonctions equals(), hashCode() et toString().

Exemple de déclaration d'une data class

data class Personne(var nom:String,val prenom:String, var   
nombreEnfant:Int = 0)

Dans cet exemple, une classe objet Personne est créée.

Description des fonctions générées

|  |  |
| --- | --- |
| Fonctions | Description |
| equals() | Cette fonction permet de tester l'égalité entre deux objets. La fonction equals() autogénérée calcule l'égalité uniquement sur les valeurs des propriétés définies à l'intérieur du constructeur principal. |
| hashcode() | Cette fonction génère une valeur de hachage grâce aux valeurs des propriétés définies dans le constructeur principal de l'instance qui l'utilise. |
| toString() | Cette fonction retourne une chaîne de caractères contenant les valeurs des propriétés définies dans le constructeur principal. |
| copy() | Cette fonction permet de faire une copie d'un objet. Il est aussi possible de faire une copie et de modifier des propriétés spécifiques en même temps. |

Exemple d'utilisation de la fonction equals()

data class Personne(var nom:String,val prenom:String,   
var nombreEnfant:Int = 0)   
{   
    var pays:String = "France"   
}   
   
fun main(args: Array<String>)   
{   
    val personne1 = Personne("Jean","Jacques")   
   
    val personne2:Personne = Personne("Jean","Jacques")   
   
    println(personne1.equals(personne2))//true   
   
    personne1.pays = "Belgique"   
   
    println(personne1.equals(personne2))//true   
   
    personne1.nom = "Roussel"   
   
    println(personne1.equals(personne2))//false   
}

Cet exemple présente des tests d'égalité entre deux objets.

Exemple d'utilisation de la fonction toString()

data class Personne(var nom:String,val prenom:String,   
var nombreEnfant:Int = 0)   
{   
    var pays:String = "France"   
}   
   
fun main(args: Array<String>)   
{   
    val personne1 = Personne("Jean","Jacques")   
   
    println(personne1)   
    println(personne1.toString())   
}   
   
Résultat :   
   
Personne(nom=Jean, prenom=Jacques, nombreEnfant=0)   
Personne(nom=Jean, prenom=Jacques, nombreEnfant=0)

Cet exemple montre comment la fonction toString() retourne une chaîne de caractères contenant les valeurs des propriétés.

Exemple d'utilisation de la fonction copy()

data class Personne(var nom:String,val prenom:String,   
var nombreEnfant:Int = 0)   
{   
    var pays:String = "France"   
}   
   
fun main(args: Array<String>)   
{   
    val personne1 = Personne("Jean","Jacques")   
   
    val personne2 = personne1.copy()   
   
    println(personne1)   
    println(personne2)   
    println(personne1.equals(personne2))   
}   
   
Résultat :   
   
Personne(nom=Jean, prenom=Jacques, nombreEnfant=0)   
Personne(nom=Jean, prenom=Jacques, nombreEnfant=0)   
true

Cet exemple montre comment faire une copie d'un objet.

Exemple d'utilisation de la fonction copy() avec modification

data class Personne(var nom:String,val prenom:String,   
var nombreEnfant:Int = 0)   
{   
    var pays:String = "France"   
}   
   
fun main(args: Array<String>)   
{   
    val personne1 = Personne("Jean","Jacques")   
   
    val personne2 = personne1.copy(nom="Martin")   
   
    println(personne1)   
    println(personne2)   
    println(personne1.equals(personne2))   
}   
   
Résultat :   
   
Personne(nom=Jean, prenom=Jacques, nombreEnfant=0)   
Personne(nom=Martin, prenom=Jacques, nombreEnfant=0)   
false

Cet exemple montre comment faire une copie d'un objet et modifier un attribut spécifique de la copie.

* L'héritage
* Présentation

Une classe peut hériter d'une et seulement d'une seule autre classe. Lors de l'héritage, on parlera de classes filles qui héritent d'une classe mère. Lorsqu'une classe hérite d'une autre classe, la classe fille peut hériter des propriétés et des fonctions de la classe mère. Par défaut, toutes les classes héritent de la classe Any. La classe Any contient trois fonctions : equals(), hashCode(), toString(), cela signifie donc que tous les objets de toutes les classes disposent des fonctions equals(), hashCode() et toString().

Contenu de la classe Any

package kotlin   
   
public open class Any {   
   
    public open operator fun equals(other: Any?): Boolean   
   
    public open fun hashCode(): Int   
   
    public open fun toString(): String   
}

La classe Any contient deux modificateurs : le modificateur public pour être disponible depuis n'importe quel autre fichier ; le modificateur open pour indiquer qu'il est possible d'en hériter. Les trois fonctions dans la classe Any ont le modificateur open, cela signifie que toutes les classes peuvent redéfinir les fonctions equals(), hashCode() et toString().

* Les bases de l’héritage
* Présentation

L'héritage permet de mettre en place une hiérarchie entre les classes : l'intérêt principal est de factoriser au mieux les structures des classes. Il y a donc des classes mères qui contiennent plutôt les structures génériques et des classes filles qui contiennent du code plus spécialisé.

* Syntaxe

Voici la syntaxe pour mettre en place un héritage :

[modificateurs] class <ClasseFille>[([parametres])]:<ClasseMere>  
([parametres])

* Héritage simple

Pour définir un héritage explicitement, il est nécessaire d’écrire deux points accompagnés du nom de la classe mère derrière le nom de la classe fille.

Exemple d'héritage

open class Mere   
   
class Fille:Mere()

Dans cet exemple, une première classe nommée Mere est déclarée. La classe Mere contient le modificateur open afin que l'on puisse en hériter. Une seconde classe nommée Fille est déclarée, elle hérite de la classe nommée Mere.

* Héritage et constructeur principal

Une classe fille doit toujours faire référence au constructeur principal de sa classe mère.

Exemple d'héritage avec une classe mère   
contenant un constructeur principal

open class Mere(valeur:Int)   
   
class Fille(uneValeur:Int):Mere(uneValeur)

Dans cet exemple, une première classe nommée Mere est déclarée. La classe Mere contient un constructeur principal nécessitant une valeur entière. Cela signifie que pour créer un objet de cette classe, il faut obligatoirement lui fournir une valeur entière. Une seconde classe nommée Fille est déclarée. La classe nommée Fille hérite de la classe nommée Mere, cela signifie que pour créer un objet de cette classe, il faut faire référence au constructeur de la classe mère et lui fournir une valeur entière. C'est pourquoi, dans le constructeur de la classe nommée Fille, une valeur entière est demandée, cette valeur est ensuite fournie au constructeur de la classe nommée Mere. Le constructeur de la classe nommée Mere utilise donc le paramètre du constructeur principal de la classe fille.

* Héritage et constructeur secondaire

Si la classe fille a plusieurs constructeurs secondaires et aucun constructeur principal, alors les constructeurs secondaires devront faire référence au constructeur principal de la classe mère grâce au mot-clé super.

Exemple d'utilisation du mot-clé super

open class Mere(valeur:Int)   
   
class Fille:Mere   
{   
    constructor(uneValeur:Int):super(uneValeur)   
   
    constructor(uneValeur:Int,uneAutreValeur:String):super(uneValeur)   
}

Dans cet exemple, chaque constructeur secondaire de la classe nommée Fille fait appel au constructeur de la classe mère nommée Mere grâce au mot-clé super.

* Super

Ce mot-clé peut être utilisé de trois manières différentes :

* Pour accéder aux propriétés et aux fonctions de la classe mère.
* Pour faire appel au constructeur de la classe mère.
* Dans une classe interne, pour faire appel à la classe englobante.

Accès aux propriétés et aux fonctions de la classe mère

Le mot-clé super permet d'accéder aux propriétés et fonctions de la classe mère.

Exemple d'utilisation d'un membre d'une classe mère

open class Vehicule(var vitesse:Float)   
   
class Voiture(vitesse:Float,puissanceMoteur:Int):Vehicule(vitesse)   
{   
    init {   
        this.vitesse = this.vitesse \* 0.9F   
    }   
}

Le bloc d'initialisation de la classe mère est toujours exécuté avant le bloc d'initialisation de la classe fille.

Accès au constructeur de la classe mère

Le mot-clé super, avec des parenthèses, permet de faire référence au constructeur de la classe mère. Cette utilisation du mot-clé super se combine avec les constructeurs secondaires.

Exemple d'utilisation du constructeur de la classe mère

open class Vehicule(var vitesse:Float)   
   
class Voiture:Vehicule   
{   
    constructor(vitesse:Float) : super(vitesse)   
   
    constructor(vitesse:Float,puissanceMoteur:Int):super(vitesse)   
   
    init {   
        this.vitesse = this.vitesse \* 0.9F   
    }   
}

Accès aux propriétés et aux fonctions d'une classe englobante

Le mot-clé super, avec une arobase et le nom de la classe englobante, permet de faire référence aux éléments d'une classe englobante.

Exemple d'utilisation d'un membre d'une classe englobante

open class Vehicule(var vitesse:Float)   
{   
    open val nombreRoues: Int get() = 4   
}   
   
class Voiture(var speed:Float):Vehicule(speed)   
{   
    public open fun f() { println("Foo.f()") }   
   
    open var info:String = "information"   
        get() = field.toUpperCase()   
   
    inner class ClasseInterne   
    {   
        fun uneFonction()   
        {   
            println(info)   
            println(super@Voiture.nombreRoues)   
        }   
    }   
}

Règle d'accès lors de multiple implémentation

Si une classe hérite d'une autre classe et implémente plusieurs interfaces (cf. la section Les interfaces de ce chapitre), alors le mot-clé super fait référence à plusieurs implémentations. Si plusieurs de ces implémentions déclarent une fonction avec le même nom, alors il peut il y avoir ambiguïté. Pour lever ces ambiguïtés, il est possible d'utiliser des chevrons afin de spécifier l'implémentation à référencer.

Exemple

open class Vehicule {   
    open fun rouler() { print("Vrrr") }   
    fun freiner() { print("Freinage en cours") }   
}   
   
interface Citroen {   
    fun rouler() { print("Ouinnn") }   
}   
   
class Voiture() : Vehicule(), Citroen   
{   
    override fun rouler() {   
        super<Citroen>.rouler()   
        super<Vehicule>.rouler()   
    }   
}

* Redéfinition de propriétés et de fonctions
* Différence entre surcharge et redéfinition

Une redéfinition (override) de fonction signifie qu’une classe fille modifie le comportement d'une fonction de sa classe mère.

Une surcharge de fonction signifie qu’une fonction est créée avec le même nom qu'une autre fonction déjà existante, mais avec des paramètres différents.

* Redéfinition de propriétés

Il est possible de redéfinir une propriété. Par défaut, cela n’est pas autorisé. Le modificateur open doit être utilisé pour autoriser une propriété à être redéfinie. La propriété remplaçant le code de la propriété mère doit posséder le modificateur override.

Il est possible de redéfinir une propriété de type val en type var.

Exemple de redéfinition de propriété

open class Vehicule(var vitesse:Float)   
{   
    open val info:String = "10L/100"   
        get() = field.toUpperCase()   
   
}   
   
class Voiture(var speed:Float):Vehicule(speed)   
{   
    override var info: String = "5L/100"   
}

L'utilisation du modificateur override équivaut à l'utilisation du modificateur open et la propriété peut donc être redéfinie. Le modificateur final permet d'interdire la redéfinition d'une redéfinition.

* Redéfinition de fonctions

Il est possible de redéfinir une fonction. Par défaut, cela n’est pas autorisé. Le modificateur open doit être utilisé pour autoriser une propriété à être redéfinie. La fonction remplaçant le code de la fonction de la classe mère doit posséder le modificateur override.

Exemple de redéfinition de fonction

open class Vehicule(var vitesse:Float)   
{   
    open fun getSon() = "Non défini"   
}   
   
class Voiture(var speed:Float):Vehicule(speed)   
{   
    override open fun getSon() = "Vroum"   
}

L'utilisation du modificateur override équivaut à l'utilisation du modificateur open et la fonction peut donc être redéfinie. Le modificateur final permet d'interdire la redéfinition d'une redéfinition.

* Les classes abstraites
* Présentation

Une classe abstraite permet de factoriser des comportements et sert de base à des classes qui en héritent. Une classe abstraite n'est pas instanciable.

* Syntaxe

Voici la syntaxe permettant de définir une classe abstraite :

abstract class <NomDeLaClasse>([paramètres])   
{   
...   
}

* Usage

Exemple de classe abstraite

abstract class Vehicule(var vitesse:Float)   
{   
    open var info:String = "information"   
        get() = field.toUpperCase()   
   
}   
   
class Voiture(var speed:Float):Vehicule(speed)   
{   
    override var info: String = "5L/100"   
}

* Les classes scellées
* Présentation

Une classe scellée (sealed class) se comporte comme une classe abstraite avec une particularité en plus. Une classe scellée permet de restreindre les classes pouvant être une sous-classe de celle-ci. Seules les classes définies dans le même fichier que la classe scellée peuvent en hériter.

* Syntaxe

Voici la syntaxe permettant de définir une classe scellée :

sealed class <NomDeLaClasse>

* Usage

Exemple d'utilisation d'une classe scellée

sealed class Vehicule   
data class Bus(var nombrePassager:Int) : Vehicule()   
object PorteAvion : Vehicule()   
class Voiture :  Vehicule()

* Les interfaces
* Présentation

Une interface permet de décrire un comportement. Une classe implémentant une interface doit donc implémenter le comportement décrit par l'interface. Les interfaces ne peuvent pas stocker d'état et peuvent contenir :

* des déclarations de fonctions abstraites,
* des fonctions,
* des propriétés abstraites,
* des propriétés avec redéfinition des accesseurs.
* Syntaxe

Voici la syntaxe permettant de définir une interface :

interface <NomDeLInterface> [extends AutresInterfaces]    
{   
    ...   
}

Voici la syntaxe permettant à une classe d'implémenter une interface :

class <NomClasse> : <NomInterface> {   
    ...   
}

* Usage

Exemple d’une interface

interface MonInterface {   
   fun uneFonction()   
   fun uneAutreFonction() {   
      //blocs d’instructions possibles   
   }   
}

Exemple d’implémentation

class UneClasse:MonInterface {   
   override fun uneFonction(){   
      //blocs d’instructions   
   }   
}

* Les classes internes
* Présentation

Une classe interne est une classe définie à l'intérieur d'une autre classe. Une classe interne peut accéder aux éléments de sa classe englobante. Ce mécanisme est mis en place lorsqu'il y a un lien très étroit entre la classe interne et sa classe englobante.

* Syntaxe

Voici la syntaxe permettant de définir une classe interne :

class <NomClasseExterne>   
{   
    inner class <NomClasseInterne>   
    {   
        ...   
    }   
}

* Usage

Exemple d'utilisation d'une classe interne

class ClasseEnglobante   
{   
    private val info: String = "Ola"   
    inner class ClasseInterne {   
        fun test() = info   
    }   
}

* Les classes imbriquées
* Présentation

Une classe imbriquée est une classe définie à l'intérieur d'une autre classe. Une classe imbriquée ne peut pas accéder aux éléments de sa classe englobante. Ce mécanisme est mis en place lorsqu'il y a un lien très étroit entre la classe imbriquée et sa classe englobante.

* Syntaxe

Voici la syntaxe permettant de définir une classe imbriquée :

class <NomClasseExterne>    
{   
    class <NomClasseImbriquée>    
    {   
        ...   
    }   
}

* Usage

Exemple d'utilisation d'une classe imbriquée

class ClasseEnglobante   
{   
    private val info: String = "Ola"   
    class ClasseImbriquee {   
        fun test() = "Hello"   
    }   
}

* Les classes anonymes
* Présentation

Une classe anonyme est une classe interne qui dérive d'une classe mère ou qui implémente une interface. Une classe anonyme ne porte pas de nom.

Il y a deux façons d'utiliser une classe anonyme :

* avec les object expressions
* avec les object declarations
* Syntaxe avec les object expressions

Voici la syntaxe permettant de créer une classe anonyme :

object : Implementation() {   
   // ...   
}

* Usage avec les object expressions

Lorsqu'il est nécessaire d'avoir un objet d'une classe qui ne sera utilisée qu'une seule fois, alors l'utilisation d'une classe anonyme est la bienvenue.

Les classes anonymes sont souvent utilisées en tant que paramètre d'une méthode.

Exemple de classe anonyme

   window.addMouseListener(object : MouseAdapter() {   
    override fun mouseClicked(e: MouseEvent) {   
        // ...   
    }   
   
    override fun mouseEntered(e: MouseEvent) {   
        // ...   
    }   
})

* Singleton
* Présentation

Le Singleton est un célèbre design pattern fréquemment utilisé permettant à une classe de n'avoir qu'une seule instance.

* Syntaxe

Voici la syntaxe permettant de définir un singleton :

object Bus    
{   
    ...   
}

* Usage

Exemple de singleton

object Bus   
{   
    var nombrePassager:Int = 0   
   
    fun entrer(nom:String)   
    {   
        println("$nom entre")   
        nombrePassager++   
    }   
   
    fun sortir(nom:String)   
    {   
        println("$nom sort")   
        nombrePassager--   
    }   
}

* Fonctions statiques
* Présentation

Une fonction statique peut être exécutée grâce au nom de la classe et ne nécessite pas d'instance de la classe dont elle fait partie. Les fonctions statiques sont utiles pour certains patterns et pour factoriser des fonctionnalités qui n'ont pas besoin d'utiliser de valeur dans un objet. Elles se contentent simplement des valeurs qui leur sont fournies en paramètres et des constantes de la classe dans laquelle elles se trouvent.

* Syntaxe

Voici la syntaxe permettant de créer une fonction statique :

class <NomDeLaClasse>   
{   
    companion object {   
   
        fun <nomDeLaFonctionStatique()   
        {   
            ...   
   
        }   
    }   
}

* Usage

Pour définir des fonctions de classes (appelées aussi fonctions statiques), c’est-à-dire des fonctions que l'on appelle via le nom de la classe, un bloc d'instructions préfixé de companion objet doit être utilisé.

Exemple de déclaration d'une fonction de classe

class Test   
{   
    companion object {   
   
        fun bonjour()   
        {   
            println("bonjour")   
        }   
    }   
}   
   
fun main(args: Array<String>)   
{   
   Test.bonjour()   
}

* Enum classe
* Présentation

Une enum classe permet de définir des objets qui pourront avoir une valeur parmi un panel de valeurs prédéfinies. Ce panel de valeurs est défini lors de la déclaration de l'enum classe.

* Déclarations

Lors de la déclaration d'une classe enum, vous pouvez définir un ensemble de valeurs simples.

Exemple d'utilisation d'un Enum avec des valeurs simples

enum class jours {   
    LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI    
}

Vous pouvez définir un ensemble de valeurs associées à des propriétés initialisables.

Exemple d'utilisation d'un Enum avec des propriétés initialisables

enum class Color(val rgb: Int) {   
    RED(0xFF0000),   
    GREEN(0x00FF00),   
    BLUE(0x0000FF)   
}

Vous pouvez définir un ensemble de valeurs associées à des fonctions.

Exemple d'utilisation d'un Enum avec des fonctions

enum class Pays {   
   
    FRANCE {   
        override fun getCapitale() = "Paris"   
    },   
   
    BELGIQUE {   
        override fun getCapitale() = "Bruxelle"   
    };   
   
    abstract fun getCapitale(): String   
}

* Généricité
* Présentation

Le principe de généricité est de développer des classes, des interfaces ou des fonctions de manière à ce qu'elles puissent être utilisées par différents types d'objets. C'est pourquoi les classes génériques ont des paramètres de type.

* Syntaxe

Voici la syntaxe permettant de définir une classe avec un paramètre de type :

class <NomDeLaClasseGenerique><<ParametreDeType>><([parametres]) {   
       
}

Voici la syntaxe permettant de définir une interface avec un paramètre de type :

interface <NomDeLInterfaceGenerique><<ParametreDeType>> {   
   
}

Voici la syntaxe permettant de définir une fonction avec un paramètre de type :

fun <ParametreDeType> <NomDeLaFonctionGenerique>([parametres]):   
[TypeDeRetour] {   
    // ...   
}

* Usage

Exemple d'une classe générique

/\*\*   
 \* Classe permettant de concaténer des objets d'un même type.   
 \* Les différents éléments concaténés sont séparés par un '|'.   
 \*/   
class Concateneur<T>(vararg items:T)   
{   
    var nombreDeConcatenation = 0   
        get()= field   
        private set   
   
    var info:String = ""   
        get()=field.trim()   
        private set   
   
    fun concatenation(item:T)   
    {   
        nombreDeConcatenation++;   
        info += item.toString() + "|";   
    }   
}

* La gestion des erreurs d'exécution
* Présentation

Lorsqu'il y a une erreur d'exécution dans un programme, un objet de type Exception est créé. Cet objet hérite toujours de la classe Throwable et contient plusieurs informations :

* un message de description de l'erreur,
* une stack trace afin d'indiquer où se trouve l'erreur,
* le motif, cette information est optionnelle.

Pour traiter une erreur, afin que le programme se termine de manière convenable ou continue son exécution normalement, il existe un mécanisme de traitement des erreurs. Ce mécanisme est articulé autour des mots-clés try catch et finally : le mot-clé try permet d'englober les instructions susceptibles de produire une erreur, try peut être utilisé comme une expression. Le mot-clé catch permet de définir les instructions à exécuter en cas d'erreur, il est possible d'utiliser plusieurs fois le mot-clé catch pour définir des instructions spécifiques selon le type d'erreur. Le mot-clé finally permet de définir les instructions à exécuter, qu'il y ait ou non une erreur ; ce mot-clé est facultatif.

Il est possible de générer une exception en utilisant le mot-clé throw combiné avec un objet Exception dérivant de la classe Throwable.

* Syntaxe

Voici la syntaxe permettant de mettre en place un mécanisme de gestion d'erreur :

try {   
    // instructions susceptibles de produire une erreur   
}   
catch (e: <ClasseException>) {   
    // instructions à exécuter en cas d'erreur   
}   
[finally {   
    // instructions à exécuter dans tous les cas   
}]

Voici la syntaxe permettant de produire une erreur :

throw Exception([message])

* Usage

Exemple de gestion d'une erreur

fun main(args: Array<String>)   
{   
    var resultat:Int = 0   
    val diviseur = 0   
    val uneValeur = 50   
   
    resultat = try   
    {   
        uneValeur / diviseur   
    }   
    catch(e:ArithmeticException)   
    {   
        println("Erreur mathématique")   
        -1   
    }   
    catch(ex:Exception)   
    {   
        println("Erreur inconnu")   
        -2   
    }   
    finally   
    {   
        println("Fin des calculs $resultat")   
    }   
   
    println("Résultat : $resultat")   
}   
   
Résultat :   
   
Erreur mathématique   
Fin des calculs 0   
Résultat : -1

Exemple de production d'une erreur

fun main(args: Array<String>)   
{   
    throw Exception("Une erreur personnelle")   
}   
   
Résultat :   
   
Exception in thread "main" java.lang.Exception: Une erreur personnelle   
at Demonstration.main(Demonstration.kt:4)

* Les collections
* Présentation

Les collections permettent d'enregistrer des listes d'objets de même type. Il en existe plusieurs types pour stocker vos objets :

|  |  |
| --- | --- |
| Type | Description |
| List | Permet d'enregistrer des éléments d'un même type de manière indexée et ordonnée. Un objet de type List ne peut être modifié. |
| MutableList | Permet d'enregistrer des éléments d'un même type de manière indexée et ordonnée. Un objet de type MutableList peut être modifié. |
| Set | Permet d'enregistrer des éléments d'un même type de manière non ordonnée, ce type de liste n'accepte pas de valeur en double. Un objet de type Set ne peut pas être modifié. |
| MutableSet | Permet d'enregistrer des éléments d'un même type de manière non ordonnée, ce type de liste n'accepte pas de valeur en double. Un objet de type MutableSet peut être modifié. |
| Map | Permet d'enregistrer des données sous forme de clé/valeur. Un objet de type Map ne peut pas être modifié. |
| MutableMap | Permet d'enregistrer des données sous forme de clé/valeur. Un objet de type MutableMap peut être modifié. |

Les collections créées à partir des classes List, Set et Map sont immuables.

* Syntaxe
* Collection de type List

Voici la syntaxe permettant de définir un objet de type List :

<val|var> <nomVariable>[:List<Int><ParametreDeType>>] =   
mutableListOf([valeurs])

* Collection de type MutableList

Voici la syntaxe permettant de définir un objet de type MutableList :

<val|var> <nomVariable>[:MutableList<] = listOf([valeurs])

* Collection de type Set

Voici la syntaxe permettant de définir un objet de type Set :

<val|var> <nomVariable>[:Set<Int>] = setOf([valeurs])

* Collection de type MutableSet

Voici la syntaxe permettant de définir un objet de type MutableSet :

<val|var> <nomVariable>[:MutableSet<Int>] =   
mutableSetOf([valeurs])

* Collection de type Map

Voici la syntaxe permettant de définir un objet de type Map :

<val|var> <nomVariable>[:Map<String, Int>] = mapOf([valeurs])

Les valeurs se définissent comme ceci :

<valeurClé> to <valeur>

* Collection de type MutableMap

Voici la syntaxe permettant de définir un objet de type MutableMap :

<val|var> <nomVariable>[:MutableMap<String, Int>] =   
mutableMapOf([valeurs])

Les valeurs se définissent comme ceci :

<valeurClé> to <valeur>

* Usage
* Collection de type List

Exemple d'utilisation d'une collection de type List

fun main(args: Array<String>)   
{   
    val couleurs: List<String> = listOf("Rose", "Jaune", "Bleu")   
    //couleurs.add("Violet")  //Erreur de compilation   
    println(couleurs.get(2))   
}

* Collection de type MutableList

Exemple d'utilisation d'une collection de type MutableList

fun main(args: Array<String>)   
{   
    val couleurs: MutableList<String> = mutableListOf("Rose",   
"Jaune", "Bleu")   
    couleurs.add("Violet")   
    println(couleurs.get(2))   
}

* Collection de type Set

Exemple d'utilisation d'une collection de type Set

fun main(args: Array<String>)   
{   
    val couleurs: Set<String> = setOf("Rose", "Jaune", "Bleu")   
    //couleurs.add("Violet") //Erreur de compilation   
    couleurs.forEach { s: String -> println(" val = $s") }   
}   
//val = Rose   
//val = Jaune   
//val = Bleu

* Collection de type MutableSet

Exemple d'utilisation d'une collection de type MutableSet

fun main(args: Array<String>)   
{   
    val couleurs: MutableSet<String> = mutableSetOf("Rose",   
"Jaune", "Bleu")   
    couleurs.add("Violet")   
    couleurs.add("Rose")   
    couleurs.forEach { s: String -> println(" val = $s") }   
}   
//val = Rose   
//val = Jaune   
//val = Bleu   
//val = Violet

* Collection de type Map

Exemple d'utilisation d'une collection de type Map

fun main(args: Array<String>)   
{   
    val cartes: Map<String, Int> = mapOf("AS" to 11, "Dix" to   
10,"Roi" to 5)   
    //cartes.put("Reine",4)   
   
    for ((carte, valeur) in cartes)   
    {   
        println("$carte = $valeur")   
    }   
}   
//AS = 11   
//Dix = 10   
//Roi = 5

* Collection de type MutableMap

Exemple d'utilisation d'une collection de type MutableMap

fun main(args: Array<String>)   
{   
    val cartes: MutableMap<String, Int> = mutableMapOf("AS" to 11,   
"Dix" to 10,"Roi" to 5)   
    cartes.put("Reine",4)   
   
    for ((carte, valeur) in cartes)   
    {   
        println("$carte = $valeur")   
    }   
}   
//AS = 11   
//Dix = 10   
//Roi = 5   
//Reine = 4

* Fonctions d'agrégation
* Présentation

Un predicate est une expression lambda qui prend un ou plusieurs arguments et retourne un booléen.

Voici la liste des principales fonctions d'agrégation.

|  |  |
| --- | --- |
| Fonction | Description |
| Any | Renvoie true si la collection a au moins un élément qui correspond au predicate passé à la fonction. |
| All | Renvoie true si tous les éléments de la collection correspondent au predicate passé à la fonction. |
| Count | Compte le nombre d'éléments dans la collection. |
| Foreach | Exécute une action autant de fois qu'il y a d'éléments. Un élément courant est disponible. |
| forEachIndexed | Exécute une action autant de fois qu'il y a d'éléments et fournit l'index de l'élément. Un élément courant et son index sont disponibles. |
| Max | Retourne le plus grand élément d'une collection. |
| maxBy | Donne la possibilité d'effectuer un traitement sur chaque élément d'une collection et retourne l'élément le plus grand d'après le résultat du traitement. |
| Min | Retourne le plus petit élément d'une collection. |
| minBy | Donne la possibilité d'effectuer un traitement sur chaque élément d'une collection et retourne l'élément le plus petit d'après le résultat du traitement. |
| none | Retourne true si aucun élément ne correspond au predicate passé en paramètre. |
| sumBy | Retourne la somme de tous les éléments de la collection. Il est possible de passer un traitement en paramètre afin de transformer la valeur de chaque élément. |

* Usage

any

Voici un exemple d'utilisation de la fonction any() :

val list = listOf(5,10,15,20)   
println(list.any{it>5})   
   
Résultat :   
   
true

all

Voici un exemple d'utilisation de la fonction all() :

val list = listOf(5,10,15,20)   
println(list.all{it%5==0})   
   
Résultat :   
   
true

count

Voici un exemple d'utilisation de la fonction count() :

val list = listOf(5,10,15,20)   
println(list.count())   
   
Résultat :   
   
4

forEach

Voici un exemple d'utilisation de la fonction forEach() :

val list = listOf(5,10,15,20)   
println(list.forEach{println(it)})   
   
Résultat :   
   
5   
10   
15   
20

forEachIndexed

Voici un exemple d'utilisation de la fonction forEachIndexed() :

val list = listOf(5,10,15,20)   
 println(list.forEachIndexed{index,valeur->println("$index :   
valeur")})   
   
Résultat :   
   
 0 : 5   
 1 : 10   
 2 : 15   
 3 : 20

max

Voici un exemple d'utilisation de la fonction max() :

val list = listOf(5,10,15,20)   
println(list.max())   
   
Résultat :   
   
20

maxBy

Voici un exemple d'utilisation de la fonction maxBy() :

val list = listOf(5,10,15,20)   
println(list.maxBy{it\*-1})   
   
Résultat :   
   
5

min

Voici un exemple d'utilisation de la fonction min() :

val list = listOf(5,10,15,20)   
println(list.min())   
   
Résultat :   
   
5

minBy

Voici un exemple d'utilisation de la fonction minBy() :

val list = listOf(5,10,15,20)   
println(list.minBy{it\*-1})   
   
Résultat :   
   
20

none

Voici un exemple d'utilisation de la fonction none() :

val list = listOf(5,10,15,20)   
println(list.none{it>50})   
   
Résultat :   
   
true

sumBy

Voici un exemple d'utilisation de la fonction sumBy() :

val list = listOf(5,10,15,20)   
println(list.sumBy{it+5})   
   
Résultat :   
   
70

* Fonctions de sélection
* Présentation

|  |  |
| --- | --- |
| Fonction | Description |
| contains(element:T) | Retourne true si l'élément passé en paramètre se trouve dans la collection. |
| drop(n:Int) | Renvoie une liste contenant tous les éléments sauf les n premiers éléments. |
| elementAt(index:Int) | Retourne l'élément à l'index passé en paramètre. |
| first() | Retourne le premier élément de la collection. |
| firstOrNull() | Retourne le premier élément de la collection, si la collection est vide alors la fonction retourne Null. |
| indexOfFirst(predicate:(T)  -> Boolean) | Renvoie le premier index de l'élément passé en paramètre, ou -1 si la collection ne contient pas cet élément. |
| indexOfLast(predicate:(T)  -> Boolean) | Retourne l'index du dernier élément correspondant au prédicat passé en paramètre, ou -1 si la collection ne contient pas cet élément. |
| dropWhile(predicate:(T)  -> Boolean) | Renvoie une liste contenant tous les éléments sauf les premiers éléments qui correspondent au prédicat passé en paramètre. |
| filter(predicate:(T)  -> Boolean) | Renvoie une liste contenant uniquement les éléments correspondant au prédicat passé en paramètre. |
| filterNot(predicate:(T)  -> Boolean) | Renvoie une liste contenant tous les éléments ne correspondant pas au prédicat passé en paramètre. |
| filterNotNull() | Renvoie une liste contenant tous les éléments qui ne sont pas NULL. |
| last() | Renvoie le dernier élément. |
| take(n:Int) | Renvoie une liste contenant les n premiers éléments. |
| takeLast(n:Int) | Renvoie une liste contenant les n derniers éléments. |
| takeWhile(predicate:(T)  -> Boolean) | Renvoie une liste contenant les premiers éléments satisfaisant le prédicat passé en paramètre. |

* Usage

contains(element:T)

Voici un exemple d'utilisation de la fonction contains() :

val list = listOf(5,10,15,20)   
   
val result = list.contains(15)   
   
println(result)   
   
Résultat :   
   
true

drop(n:Int)

Voici un exemple d'utilisation de la fonction drop() :

val list = listOf(5,10,15,20)   
   
val result = list.drop(2);   
   
println(result)   
   
Résultat :   
   
[15, 20]

elementAt(index:Int)

Voici un exemple d'utilisation de la fonction elementAt() :

val list = listOf(5,10,15,20)   
   
val result = list.elementAt(2)   
   
println(result)   
   
Résultat :   
   
15

first()

Voici un exemple d'utilisation de la fonction first() :

val list = listOf(5,10,15,20)   
   
val result = list.first()   
   
println(result)   
   
Résultat :   
   
5

indexOfFirst(predicate:(T) -> Boolean)

Voici un exemple d'utilisation de la fonction indexOfFirst() :

val list = listOf(5,10,15,20)   
   
val result = list.indexOfFirst({it%2==0})   
   
println(result)   
   
Résultat :   
   
1

Retourne l'index du premier nombre pair.

indexOfLast(predicate:(T) -> Boolean)

Voici un exemple d'utilisation de la fonction indexOfLast() :

val list = listOf(5,10,15,20)   
   
val result = list.indexOfLast({it%2==0})   
   
println(result)   
   
Résultat :   
   
3

Retourne le résultat du dernier nombre pair.

dropWhile(predicate:(T) -> Boolean)

Voici un exemple d'utilisation de la fonction dropWhile() :

val list = listOf(5,10,15,20)   
   
val result = list.dropWhile({i->i<12})   
   
println(result)   
   
Résultat :   
   
[15, 20]

filter(predicate:(T) -> Boolean)

Voici un exemple d'utilisation de la fonction filter() :

val list = listOf(5,10,15,20)   
   
val result = list.filter({i->i<12})   
   
println(result)   
   
Résultat :   
   
[5, 10]

filterNot(predicate:(T) -> Boolean)

Voici un exemple d'utilisation de la fonction filterNot() :

val list = listOf(5,10,15,20)   
   
val result = list.filterNot({i->i<12})   
   
println(result)   
   
Résultat :   
   
[15, 20]

filterNotNull()

Voici un exemple d'utilisation de la fonction filterNotNull() :

val list = listOf(5,10,null,15,20, null)   
   
val result = list.filterNotNull()   
   
println(result)   
   
Résultat :   
   
[5, 10, 15, 20]

last()

Voici un exemple d'utilisation de la fonction last() :

val list = listOf(5,10,null,15,20, null,33)   
   
 val result = list.last()   
   
 println(result)   
   
Résultat :   
   
33

take(n:Int)

Voici un exemple d'utilisation de la fonction take() :

val list = listOf(5,10,null,15,20, null,33)   
   
 val result = list.take(3)   
   
 println(result)   
   
Résultat :   
   
[5, 10, null]

takeLast(n:Int)

Voici un exemple d'utilisation de la fonction takeLast() :

val list = listOf(5,10,null,15,20, null,33)   
   
 val result = list.takeLast(3)   
   
 println(result)   
   
Résultat :   
   
 [20, null, 33]

takeWhile(predicate:(T) -> Boolean)

Voici un exemple d'utilisation de la fonction takeWhile() :

val list = listOf(5,10,null,15,20, null,33)   
   
 val result = list.takeWhile({it!=null})   
   
 println(result)   
   
Résultat :   
   
 [5, 10]

* Les expressions lambda
* Présentation

Les expressions lambda permettent de passer des traitements en paramètres d'une méthode ; elles permettent de réduire de manière significative la quantité de code.

Une expression lambda est une fonction anonyme, il n'y a pas de déclaration de type de retour, pas de modificateur de fonction, pas de nom. C'est une façon concise et rapide de définir une fonction à l'endroit où elle est utilisée. Les expressions lambda sont juste composées d'une liste de paramètres et d'un bloc d'instructions.

Les lambdas sont une des composantes principales de la programmation fonctionnelle.

* Syntaxe

Voici la syntaxe d'une expression lambda :

[paramètres] [->] <[{]blocDInstructions[}]>

L'opérateur -> sépare les paramètres du bloc d'instructions qui va les utiliser.

Quelques règles concernant les expressions lambda :

* La partie "paramètres" n'est pas obligatoire si aucun paramètre n'est utilisé.
* Il peut y avoir de zéro à n paramètres.
* Il doit y avoir des parenthèses autour de la partie "paramètres" sauf s'il n'y a qu'un seul paramètre et que son type est inféré, alors les parenthèses sont facultatives.
* Il doit y avoir une virgule entre chaque paramètre.
* Le bloc d'instructions doit être entouré par des accolades sauf s'il n'y a qu'une seule instruction, alors les accolades sont facultatives et le type de retour est inféré.
* Le type d'un paramètre peut être déclaré ou inféré.
* Une expression lambda retourne la valeur de la dernière instruction ou void.
* Les variables passées en paramètres peuvent être utilisées dans la partie instruction.
* Des variables peuvent être définies dans la partie instruction.
* Usage

Voici plusieurs exemples permettant de saisir le fonctionnement des expressions lambda.

Exemple d'expression lambda sans paramètre

() -> "bonjour"

Exemple d'expression lambda avec un paramètre

param -> param + param

Exemple d'expression lambda avec un paramètre et des parenthèses

(param) -> param + param

Exemple d'expression lambda avec deux paramètres

(a, b) -> a + b

Exemple d'expression lambda avec une condition

(nombre)-> nombre%2!=0

Exemple d'expression lambda avec plusieurs instructions

() ->    
   
{    
    println("bonjour je retourne 55")    
   
    55   
   
}

* Stocker le traitement dans une variable

Il est possible d'enregistrer une expression lambda dans une variable et d'utiliser cette variable comme une fonction.

Exemple d'une expression lambda stockée dans une variable

fun main(args: Array<String>)   
{   
    val paire = { unEntier : Int -> unEntier%2 == 0 }   
   
    println(paire(5))   
}   
   
Résultat :   
   
false