Kotlin : les fondamentaux

1. La fonction main

La fonction main est le point d'entrée des programmes Kotlin, ce sont les instructions de cette fonction qui seront exécutées en premier.

Exemple avec un Hello world

fun main(args: Array<String>)   
{   
    println("Hello world !!!")   
}   
   
Résultat :   
   
Hello world !!!

L'exécution de ce programme permet d'afficher en console "Hello world !!!". Le mot-clé fun permet de définir une fonction. Le mot-clé main est le nom de la fonction. Les parenthèses après le nom de la fonction permettent de définir les paramètres de la fonction. Les accolades définissent le début et la fin du bloc d'instructions. La ligne println("Hello world !!!") permet d'afficher dans la console : "Hello world !!!".

* Les points-virgules sont facultatifs à la fin d'une ligne d'instruction.

1. Les variables
   1. Présentation

Une variable permet d'associer une valeur à un nom.

Le langage Kotlin est construit autour de deux paradigmes :

* Le paradigme de la programmation orientée objet qui ne voit aucun inconvénient à utiliser des variables mutables (c’est-à-dire des variables qui peuvent changer d’état).
* Le paradigme de la programmation fonctionnelle qui invite à utiliser des variables immutables au maximum dans le but d'éviter les effets de bord.

La première chose à faire lors de la déclaration d'une variable est de définir si elle est mutable ou immutable grâce aux mots-clés var ou val.

Ensuite, le nom de la variable doit être obligatoirement indiqué.

Le type de la variable peut être défini après deux-points (:), mais cela n'est pas obligatoire si on lui affecte une valeur au moment de sa déclaration. Effectivement, le type de la variable peut être défini explicitement après le nom de la variable, mais aussi implicitement puisque le langage Kotlin supporte l'inférence de type, ce qui permet au compilateur de déduire automatiquement le type d'une variable.

* La convention de nommage pour les noms de variables est lowerCamelCase. C’est-à-dire que le nom doit être en minuscule mais chaque première lettre de chaque nouveau mot doit être en majuscule (ex. : indexCompteurPlantes).
  1. Les variables immutables

Une variable immutable ne peut pas changer de valeur, elle ne peut être affectée qu'une seule fois. Le mot-clé pour définir une variable immutable est val.

* Dans la mesure du possible, il est conseillé d'utiliser au maximum les variables immutables afin d'avoir un code plus concis et se rapprochant du paradigme de programmation fonctionnelle.

Exemple avec la définition explicite du type de la variable

val index : Int = 0

Dans cet exemple, la variable index est égale à 0 et est de type Int.

Exemple utilisant l'inférence de type

val index = 5

Dans cet exemple, la variable index est égale à 5 et est de type Int.

* L'inférence de type ne peut pas fonctionner si aucune valeur n'est affectée à la variable.

Exemple utilisant une affectation unique, mais différée

val index : Int   
index = 1564

Dans cet exemple, la variable index est égale à 1564.

Exemple utilisant deux affectations conditionnées

val index : Int   
   
if(isOk()) {   
    index = 10   
}   
else {   
    index = 0}

Dans cet exemple, la variable index sera égale à 10 si la fonction isOk() retourne true. Dans le cas où la fonction isOk() retourne false la variable index sera égale à 0.

* 1. Les variables mutables

Une variable mutable peut être affectée plusieurs fois. Le mot-clé pour définir une variable mutable est var.

* Le type d'une variable est fixe.

Exemple avec la définition explicite du type de la variable

var index : Int = 0   
index++

Dans cet exemple, la variable nommée index est de type Int.

Exemple utilisant l'inférence de type

var index = 0   
index++

Dans cet exemple, la variable nommée index est de type Int.

* 1. Les constantes

Les variables dont la valeur est constante et connue au moment de la compilation peuvent être définies comme des constantes à l'aide du mot-clé const. Par convention, le nom de ces variables utilise la convention UPPER\_SNAKE\_CASE.

Exemple de constante

const val VALEUR\_CONSTANTE: String = "Valeur constante"

Dans cet exemple, la variable nommée VALEUR\_CONSTANTE est une constante de type String dont la valeur est "Valeur constante".

* 1. Les types de base

Les types de base sont des wrappers (classes représentant des données primitives) proposant une collection de méthodes permettant de les manipuler.

* Lors de la compilation, les wrappers sont considérés comme des types primitifs Java lorsque cela est possible.
  + 1. Les types entiers

Voici la liste des types entiers :

|  |  |  |  |
| --- | --- | --- | --- |
| Entiers signés | | | |
| Byte | -128 | 127 | 8 bits |
| Short | -32768 | 32767 | 16 bits |
| Int | -2147483648 | 2147483647 | 32 bits |
| Long | -9223372036854775808 | 9223372036854775807 | 64 bits |

Le maximum et le minimum de ces types peuvent être retrouvés grâce aux constantes MAX\_VALUE et MIN\_VALUE.

Exemple

val valeurMax = Int.MAX\_VALUE   
   
val valeurMin = Int.MIN\_VALUE

* Les variables de type entier n'ont pas de valeur par défaut. Pour être utilisées, elles doivent être obligatoirement initialisées.
  + 1. Les types décimaux

Voici la liste des types décimaux :

|  |  |  |  |
| --- | --- | --- | --- |
| Décimaux signés | | | |
| Float | 1.4E-45 | 3.4028235E38 | 32 bits |
| Double | 4.9E-324 | 1.7976931348623157E308 | 64 bits |

Le maximum et le minimum de ces types peuvent être retrouvés grâce aux constantes MAX\_VALUE et MIN\_VALUE.

Exemple de consultation du minimal et du maximal

val valeurMax = Long.MAX\_VALUE   
   
val valeurMin = Long.MIN\_VALUE

* Les variables de type décimal n'ont pas de valeur par défaut. Pour être utilisées, elles doivent être obligatoirement initialisées.

Étant donné la manière dont sont stockés les décimaux, de nombreux résultats peuvent s'avérer inexacts.

Exemple d'inexactitude

val unDouble: Double = 1.35 - 0.22   
   
println(unDouble)   
   
RESULTAT :   
   
1.1300000000000001

* Pour éviter les inexactitudes, la classe java.math.BigDecimal doit être utilisée.
  + 1. Le type booléen

Le type Boolean peut prendre les valeurs true et false.

Trois opérations sont possibles sur les variables booléennes :

|  |  |  |  |
| --- | --- | --- | --- |
| Opération | Opérateur | Exemple | Résultat |
| Ou logique | || | true || false | true |
| Et logique | && | true && false | false |
| Négation logique | ! | !true | false |

Exemple d'utilisation

val unBool:Boolean = true

* Les variables de type Boolean n'ont pas de valeur par défaut. Pour être utilisées, elles doivent être obligatoirement initialisées.
  + 1. Le type caractère

Le type Char permet d'enregistrer un caractère. La valeur à affecter à la variable doit être entourée de simples quotes. Les variables de type Char ne peuvent être traitées comme des nombres.

Exemple d'affectation d'une variable de type Char

val unChar:Char = 'L'

* Les variables de type Char n'ont pas de valeur par défaut. Pour être utilisées, elles doivent être obligatoirement initialisées.

Des séquences d'échappement sont disponibles pour utiliser des caractères spécifiques. Voici la liste :

|  |  |
| --- | --- |
| Séquence | Description |
| \t | Tabulation |
| \b | Retour arrière |
| \n | Saut de ligne |
| \r | Retour chariot |
| \' | Simple quote |
| \" | Double quote |
| \\ | Antislash |
| \$ | Dollar |

Pour encoder d'autres caractères spéciaux, il est nécessaire d'utiliser les séquences d'échappement Unicode au format \Unnnn.

Exemple

val unChar:Char = '\U004C' // représente la lettre L

* 1. Les valeurs numériques littérales
     1. Long et Float

Par défaut, les valeurs entières sont considérées comme des valeurs de type Int. Pour forcer le compilateur à considérer une valeur entière en type Long, il faut utiliser le caractère L.

Exemple

val unLong = 207178207178L;

Dans cet exemple, la variable unLong est de type long et est égale à 207178207178.

Par défaut, les valeurs réelles sont considérées comme des valeurs de type Double. Pour forcer le compilateur à considérer une valeur réelle en type Float, il faut utiliser le caractère F.

Exemple

val unFloat = 2.07178207178F;

Dans cet exemple, la variable unFloat est de type Float et est égale à 2.07178207178.

* + 1. Écritures binaire, octale et hexadécimale

Il est possible de représenter des valeurs au format binaire en les préfixant avec 0b.

Exemple d'une valeur binaire

val uneValeur: Int = 0b01010111

Dans cet exemple, la variable unValeur est égale à 87.

Il est possible de représenter des valeurs au format hexadécimal en les préfixant avec 0x.

Exemple d'une valeur hexadécimale

val uneAutreValeur: Int = 0xABCDEF

Dans cet exemple, la variable uneAutreValeur est égale à 11 259 375.

* Le format octal n'est pas pris en compte.
  + 1. Utilisation des underscores

Pour rendre les grandes valeurs numériques plus lisibles, il est possible d'utiliser des underscores :

Exemple

val x: Int = 207178   
val y: Int = 207\_178   
   
println(x == y) //true

Ce mécanisme est possible avec tous les types de base :

Exemple

val unByte: Byte = 1\_1\_1   
val unLong = 207\_178\_207\_178L   
val unFloat = 2.07\_178\_207\_178F

Ce mécanisme est possible avec les représentations binaires et hexadécimales :

Exemple

val uneValeur: Int = 0xAB\_CD\_EF   
val uneAutreValeur: Int = 0b0101\_0111

* 1. Les chaînes de caractères

Le type String permet d'enregistrer des chaînes de caractères.

Exemple

val uneValeur: String = "La nuit porte conseil"

* + 1. Immuable

Les variables de type String sont immutables et cela, même si une variable de type String est définie avec le mot-clé var. En effet, si une variable de type String est modifiée, alors son espace mémoire est détruit, puis un nouvel espace mémoire est recréé afin de stocker la nouvelle valeur.

Exemple

var uneValeur: String = "La nuit porte conseil" //un espace mémoire   
est créé   
   
uneValeur = "La nuit permet de dormir" //l'espace mémoire   
anciennement créé est détruit puis un nouvel espace est créé afin   
de stocker la nouvelle valeur

* À cause de ce mécanisme de gestion de la mémoire, les traitements dans lesquels il est nécessaire de modifier un très grand nombre de fois des chaînes de caractères ne doivent pas utiliser de variables de type String. Il est fortement recommandé d'utiliser des variables de type StringBuilder ou StringBuffer.
  + 1. Itérable

Il est possible d'itérer sur une variable de type String afin de consulter tous les caractères les uns après les autres.

Exemple d'itération sur une chaîne de caractères

var text = "Bonjour"   
   
for(caractere in text)   
{   
    println(caractere)   
}   
   
Résultat :   
   
B   
   
o   
   
n   
   
j   
   
o   
   
u   
   
r

La variable nommée caractere est de type Char.

* + 1. Texte préformaté

Il est possible d'enregistrer du texte préformaté dans une variable de type String grâce à l'utilisation de triples doubles quotes ("""), ainsi le texte est présenté tel qu'il est écrit lors de l'affectation.

Exemple

var uneString = """   
   
       Hello   
   Bonjour   
       Hola   
   
    """   
   
println(uneString)   
   
Résultat :   
   
           Hello   
   
       Bonjour   
   
           Hola

* + 1. String Templates

Des variables peuvent être évaluées directement à l'intérieur d'une chaîne de caractères sans passer par un mécanisme de concaténation qui peut parfois se révéler assez lourd. Le caractère $ doit préfixer le nom de la variable se trouvant dans la chaîne de caractères. On appelle cette combinaison "template expression".

Exemple d'utilisation d'une template expression simple

val nom = "Nicolas"   
val phrase = "Bonjour $nom"   
   
println(phrase)   
   
Résultat :   
Bonjour Nicolas

En utilisant des accolades, il est possible d'évaluer des opérations dans une chaîne de caractères.

Exemple d'utilisation d'une template expression   
combinée avec une opération

val phrase = "3 + 3 = ${3 + 3}"   
   
println(phrase)   
   
Résultat :   
   
Bonjour 3 + 3 = 6

* 1. Les types de base Nullable

Les types nullable représentent des variables de type de base. Ils peuvent être affectés avec les mêmes valeurs que les types de base plus la valeur null. C'est le point d'interrogation derrière le nom du type qui indique que le type est Nullable. Lors de la comparaison entre deux types nullable, il est obligatoire de protéger votre code avec un test de non-nullité.

Exemple dans lequel le programme ne peut pas compiler,   
car il n'y a pas de test de non-nullité

fun main(args: Array<String>) {   
    val uneString:String? = max(50,45)   
    println(uneString)   
}   
fun max(unEntier:Int?,unAutreEntier:Int?):String?   
{   
    val resultat:String   
    if(unEntier > unAutreEntier)   
    {   
        resultat = "Un entier est supérieur"   
    }else{   
        resultat = "Un entier est inférieur"   
    }   
    return resultat   
}

Dans cet exemple, le programme ne peut pas compiler, car on ne teste pas la nullité des variables de type Nullable.

Exemple où le programme peut compiler, car il y a un test de non-nullité

fun main(args: Array<String>) {   
   
    val uneString:String? = max(50,45)   
    println(uneString)   
   
}   
   
fun max(unEntier:Int?,unAutreEntier:Int?):String?   
{   
    if(unEntier == null || unAutreEntier == null)   
    {   
        return null;   
    }   
   
    val resultat:String   
    if(unEntier > unAutreEntier)   
    {   
        resultat = "Un entier est supérieur"   
    }else{   
        resultat = "Un entier est inférieur"   
    }   
    return resultat   
}

Dans cet exemple, le programme peut compiler, car on teste la nullité des variables de type Nullable.

Lors de la compilation, les types nullable Kotlin sont remplacés par les wrappers Java correspondants.

* 1. Les types Any et Any?

La classe Any est la base de la hiérarchie des classes non nullable. Cela signifie que toutes les classes non nullable, y compris les classes de type comme Int, héritent de Any. La classe Any? est la base de la hiérarchie des classes nullable. Les classes Any et Any? contiennent les fonctions equals(), hashCode() et toString(), cela signifie que toutes les classes Kotlin héritent de ces fonctions.

Contenu de la classe Any

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

La fonction equals() permet d'indiquer si un autre objet est égal à l'objet courant.

La fonction hashCode() renvoie une valeur de hachage pour l'objet courant.

La fonction toString() renvoie une chaîne de caractères décrivant l'objet courant.

* 1. Le type Unit

Unit est le type spécifié par les fonctions qui ne retournent aucune valeur.

L'utilisation de Unit pour les fonctions qui ne retournent aucune valeur est facultative.

Exemple d'utilisation du type Unit

fun bonjour(prenom:String):Unit   
{   
    println("Bonjour $prenom")   
}

Dans cet exemple, la fonction ne retourne rien, c'est la raison pour laquelle Unit est définit comme type de retour.

Exemple équivalent au précédent sans le Unit

fun bonjour(prenom:String)   
{   
    println("Bonjour $prenom")   
}

Dans cet exemple, la fonction ne retourne rien, c'est la raison pour laquelle Unit est défini comme type de retour étant donné que Unit est le type de retour par défaut.

* Le type Unit est compilé en void dans le bytecode Java.
  1. Le type Nothing

Nothing est un type de retour dont le but est d'indiquer que la fonction qui l'utilise ne se terminera jamais. Il peut y avoir plusieurs raisons pour lesquelles une fonction ne se termine jamais : si elle lance toujours une exception, si elle contient une boucle infinie, etc.

Exemple d'utilisation

fun lancerException():Nothing   
{   
    throw Exception("Erreur")   
}

Dans cet exemple, la fonction ne pourra jamais se terminer, car elle lance une erreur. C'est la raison pour laquelle le type de retour est Nothing.

* 1. Les tableaux

Un tableau est une variable permettant de faire référence à plusieurs autres variables d'un même type. Pour accéder aux variables référencées dans un tableau, on utilise un index dont la première valeur est égale à 0. Le type générique Array permet de créer des tableaux.

* + 1. Création d'un tableau

Il existe différentes manières de créer un tableau :

La fonction arrayOf

La fonction arrayOf permet de créer un tableau contenant les valeurs passées en paramètres.

Exemple d'utilisation de la fonction arrayOf

//Avec inférence de type   
   
val unTableau = arrayOf("Bleu","Blanc","Rouge")   
   
//Avec spécification du type   
   
val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge")

Dans cet exemple, un tableau de String est créé avec les valeurs "Bleu", "Blanc" et "Rouge".

La fonction arrayOfNulls

Cette fonction permet de créer un tableau contenant des objets nullable. La fonction arrayOfNulls attend en paramètre la taille du tableau.

Exemple d'utilisation de la fonction arrayOfNulls

val unTableau = arrayOfNulls<String>(10)

Dans cet exemple, un tableau de String, pouvant stocker dix variables de type String, est créé. Toutes les valeurs sont égales à null.

Le constructeur Array

Le constructeur du type Array attend en paramètres la taille du tableau, ainsi qu'un initialiseur sous la forme d'une expression lambda (cf. chapitre Kotlin : la programmation orientée objet - Les expressions lambda).

Exemple d'utilisation du constructeur Array

//En spécifiant le type   
val unTab = Array<String>(10){i->"Vide"}   
//Avec inférence de type   
val unTab2 = Array(10){i->"Vide"}

Dans cet exemple, un tableau de String, pouvant stocker dix variables de type String, est créé. Toutes les valeurs sont égales à "Vide".

Les tableaux de types primitifs

Si vous avez besoin d'un tableau avec des types primitifs (pour utiliser une fonction Java par exemple), il existe autant de méthodes qu'il y a de types primitifs en Java : IntArray(), LongArray(), CharArray()…

Exemple de création de tableaux contenant des types primitifs

//Tableau de int de taille 5   
val tableau = IntArray(5)   
//Tableau de int de taille 3, initialisé   
val tableau2 = IntArrayOf(4,5,6)

Dans cet exemple, un premier tableau d'entiers vide d'une taille de cinq est créé. Un second tableau d'entiers d'une taille de trois est créé, ce tableau contient les valeurs 4, 5 et 6.

* + 1. Utilisation courante

Taille d'un tableau

La propriété size permet de connaître le nombre d'éléments que peut contenir un tableau.

Exemple d'utilisation de la propriété size

val unTab = Array<String>(10){i->"Vide"}   
println("taille du tableau : ${unTab.size}")   
   
Résultat :   
   
taille du tableau : 10

Remplir un tableau

La fonction fill permet de remplir un tableau. Elle attend en premier paramètre la valeur avec laquelle remplir le tableau. Le second paramètre est optionnel, il permet de définir à partir de quel élément on souhaite remplir le tableau. Le troisième paramètre est optionnel, il permet de définir jusqu'à quel élément on souhaite remplir le tableau.

Exemple d'utilisation de la fonction fill

val unTab = Array<String>(10){i->"Vide"}   
unTab.fill("coucou",2,8)

Dans cet exemple, un tableau de String d'une taille de 10 est créé. Toutes les valeurs sont égales à "Vide". Ensuite, on modifie le contenu du tableau avec la fonction fill, ainsi les éléments d’indices 2 à 8 sont égaux à "coucou".

Accéder à un élément

La fonction get permet d'accéder à un élément d'un tableau. Utiliser la fonction get revient à utiliser les crochets [].

Exemple d'accès à un élément de tableau.

val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge","Vert")   
// accède au troisième élément du tableau   
println(unAutreTableau.get(2));   
// accède au troisième élément du tableau   
println(unAutreTableau[2]);   
   
Résultat :   
   
Rouge   
Rouge

Modifier un élément

La fonction set permet de modifier un élément d'un tableau. Utiliser la fonction set revient à utiliser les crochets [].

Exemple de modification d'un tableau

val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge","Vert")   
unAutreTableau.set(2,"Violet")   
unAutreTableau[3] = "Marron"   
println(unAutreTableau[2])   
println(unAutreTableau[3])   
   
   
Résultat :   
   
Violet   
Marron

Lister les éléments d'un tableau

La boucle foreach permet de parcourir simplement un tableau.

Exemple d'utilisation du foreach

val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge","Vert")   
   
for(info in unAutreTableau)   
{   
    println("Info : " + info)   
}   
   
Résultat :   
   
Info : Bleu   
Info : Blanc   
Info : Rouge   
Info : Vert

Exemple d'utilisation du foreach en se basant sur l'index

val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge","Vert")   
   
for (i in unAutreTableau.indices) {   
    println("La valeur à l'index $i est : ${unAutreTableau[i]}")   
}   
   
Résultat :   
   
La valeur à l'index 0 est : Bleu   
La valeur à l'index 1 est : Blanc   
La valeur à l'index 2 est : Rouge   
La valeur à l'index 3 est : Vert

Exemple d'utilisation du foreach avec une expression lambda

val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge")   
   
unAutreTableau.forEach { i -> println(i) }   
   
Résultat :   
   
Bleu   
Blanc   
Rouge

Afficher le contenu d'un tableau

La fonction joinToString permet de récupérer le contenu d'un tableau sous forme de String.

Exemple d'affichage du contenu d'un tableau

val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge")   
   
println(unAutreTableau.joinToString())   
   
Résultat :   
   
Bleu, Blanc, Rouge

La classe Arrays offre de multiples fonctionnalités pour manipuler les tableaux telles que la récupération du contenu d'un tableau sous forme de String.

Exemple d'affichage du contenu d'un tableau

val unAutreTableau = arrayOf<String>("Bleu","Blanc","Rouge")   
   
println(Arrays.toString(unAutreTableau))   
   
Résultat :   
   
[Bleu, Blanc, Rouge]

* 1. Conversions des types de base

Tous les types de base proposent des fonctions de conversion très simples à utiliser. Il est même obligatoire d'utiliser ces fonctions pour passer d'un type à un autre étant donné que les conversions implicites sont interdites. Parmi les fonctions de conversion, on retrouve les fonctions suivantes : toByte(), toShort(), toInt(), toLong(), toFloat(), toDouble(), toChar() et toString().

* Il y a toujours un risque de perte de données lorsqu'on passe d'un type à un autre type plus petit. Par exemple, si l'on passe d'un type long sur 64 bits à un type int sur 32 bits.

Exemples de conversions

val unLong:long = 46465   
val unEntier:Int = unLong.toInt()   
   
val unChar:Char = 'A'   
val unShort:Short = unChar.toShort();   
   
var unString:String = "5"   
val unByte:Byte =  unString.toByte()   
   
val unFloat:Float = 15.6F;   
unString = unFloat.toString()

* Attention, il est possible de comparer deux variables avec la fonction equals, uniquement si elles sont du même type.
  1. Portée des variables

La portée d'une variable est la partie de code dans laquelle la variable est disponible. Sa portée dépend de la zone dans laquelle la variable est déclarée. Cela peut être dans le bloc d'instructions d'une classe, dans le bloc d'instructions d'une fonction ou dans le bloc d'instructions à l’intérieur d'une fonction. En général, sa portée est limitée au bloc d'instructions dans lequel elle a été déclarée. La définition de la portée est similaire à celle du langage Java à une exception près : avec Kotlin, il est possible de définir des variables globales. Pour définir une variable globale, il suffit de la définir hors de tout bloc d'instructions, ainsi elle sera disponible depuis n'importe quel fichier, de n'importe quelle fonction ou classe. Cela est fortement déconseillé afin d'éviter des effets de bord et un non respect de l'architecture applicative qui pourrait amener à rendre votre application difficilement maintenable.

1. Les opérateurs
   1. Les opérateurs arithmétiques

Voici la liste des opérateurs arithmétiques :

|  |  |  |  |
| --- | --- | --- | --- |
| Opérateurs | Description | Exemple | Résultat |
| \* | Multiplier | 2\*2 | 4 |
| / | Diviser | 4/2 | 2 |
| % | Modulo | 5 %2 | 1 |
| + | Additionner | 1+1 | 2 |
| - | Soustraire | 1-1 | 0 |

* 1. Les opérateurs de bits

Voici la liste des opérateurs de bits :

|  |  |  |  |
| --- | --- | --- | --- |
| Opérateurs | Description | Exemple | Résultat |
| shl | Décalage à gauche | 0b10 shl 1 | 4 |
| shr | Décalage à droite | 0b10 shr 1 | 1 |
| ushr | Décalage à droite et met les bits de gauche à zéro | 0b10 shr 1 | 1 |
| and | ET binaire | 0b10 and 0b1 | 0 |
| or | OU binaire | 0b10 or 0b1 | 3 |
| xor | Ou exclusif | 0b10 xor 0b1 | 3 |
| inv | Inversement des bits | 0b1.inv() | -2 |

* 1. Les opérateurs unaires

Voici la liste des opérateurs unaires :

|  |  |  |  |
| --- | --- | --- | --- |
| Opérateurs | Description | Exemple | Résultat |
| + | Addition | 2+2 | 4 |
| - | Soustraction | 2-2 | 0 |
| ! | Permet d'inverser la valeur d'un booléen | !true | False |
| ++ | Permet d'incrémenter une valeur de 1 | var i:Int = 0  i++ | i vaut 1 |
| -- | Permet de décrémenter une valeur de 1 | var i:Int = 0  i-- | i vaut -1 |

* 1. Les opérateurs d'affectation

Voici la liste des opérateurs d'affectation :

|  |  |  |  |
| --- | --- | --- | --- |
| Opérateurs | Description | Exemple | Résultat |
| = | Affectation standard | Var x = 2 | X est égal à 2 |
| += | Addition | Var x = 2  X += 2 | X est égal à 4 |
| -= | Soustraction | Var x = 2  X -= 2 | X est égal à 4 |
| \*= | Multiplication | Var x = 2  X \*= 2 | X est égal à 4 |
| /= | Division | Var x = 2  X /= 2 | X est égal à 1 |
| %= | Modulo | Var x = 2  X %= 2 | X est égal à 0 |

* 1. Les opérateurs de comparaison

Voici la liste des opérateurs de comparaison :

|  |  |  |  |
| --- | --- | --- | --- |
| Opérateurs | Description | Exemple | Résultat |
| == | Égalité | 1 == 1 | True |
| != | Inégalité | 1 != 1 | False |
| < | Inférieur à | 1 < 1 | False |
| > | Supérieur à | 2>1 | True |
| <= | Inférieur ou égal | 1<=1 | True |
| >= | Supérieur ou égal | 2>=2 | True |
| in | Égalité à un élément d'une multitude de valeurs | 5 in 1..10 | True |
| !in | Inégalité à aucun élément d'une multitude de valeurs | 5 !in 1..10 | False |
| is | Teste si une variable est d'un certain type | 5 is Int | True |
| !is | Teste si une variable n'est pas d'un certain type | 5 !is Int | False |

* 1. Les opérateurs logiques

Voici la liste des opérateurs logiques :

|  |  |  |  |
| --- | --- | --- | --- |
| Opérateurs | Description | Exemple | Résultat |
| && | Et logique | True && false | False |
| || | Ou logique | True || false | True |
| ! | Négation | !true | false |

1. Les structures de contrôle
   1. Les conditionnelles

Les conditionnelles permettent d'orchestrer les programmes selon certaines conditions. Il existe deux expressions conditionnelles : if et when. On parle d'"expression", car if et when peuvent retourner une valeur.

* La différence entre une déclaration et une expression est que la déclaration ne peut pas retourner de valeur contrairement à l'expression.
  + 1. L'expression if

Le if permet d'exécuter un bloc d'instructions spécifique selon le résultat de la condition qu'il a à exécuter. Le if permet aussi d'affecter une valeur spécifique à une variable selon le résultat de la condition qu'il a à exécuter.

Exemple d'utilisation simple du if

val age:Int = 80   
val info:String   
   
if(age < 18)   
{   
    info = "Vous ne pouvez pas conduire"   
}else if(age >= 18 && age < 77)   
{   
    info = "Vous pouvez conduire"   
}else{   
    info = "Vous ne devriez pas conduire"   
}   
   
println(info)   
   
Résultat :   
   
Vous ne devriez pas conduire

Exemple d'utilisation du if comme Expression

val age:Int = 15   
val info:String   
   
info = if(age < 18)   
{   
    println("""Il est possible d'ajouter des instructions   
              |du moment que la dernière ligne contient   
              |une valeur""".trimMargin())   
    "Vous ne pouvez pas conduire"   
}else if(age > 18 && age < 77)   
{   
    "Vous pouvez conduire"   
}else{   
    "Vous ne devriez pas conduire"   
}   
   
println(info)   
   
Résultat :   
   
Vous ne pouvez pas conduire

Dans cet exemple, l'affectation de la variable info se fait sur le if.

* L'opérateur ternaire n'existe pas.
  + 1. L'expression when

L'expression when est utilisée lorsque de nombreuses conditions doivent être traitées. L'expression when peut être utilisée de plusieurs façons différentes :

* Tester l'égalité d'une variable par rapport à une liste de valeurs.
* Réunir et tester plusieurs conditions indépendantes.
* Affecter une valeur spécifique à une variable selon des conditions spécifiques.

Exemple avec condition selon une valeur

val age = 15   
   
when (age) {   
    in 1..17 -> print("Vous êtes mineur")   
    18 -> print("Vous devriez passer votre permis")   
    else -> {   
        print("Faites ce qu'il vous plaît")   
    }   
}   
   
Résultat :   
   
Vous êtes mineur

Exemple avec des conditions indépendantes les unes des autres

when {   
    total == 0 -> print("Vous n'avez rien commandé")   
    total > 0 && total <= 200 -> print("Vos frais de port   
ne sont pas gratuits")   
    total > 200  -> print("Frais de port gratuits")   
    else -> print("Etrange")   
}

Exemple du when utilisé comme expression

val position = 1    
   
val medaille =    
when {   
    position == 1 -> "Or"   
    position == 2 -> "Argent"   
    position == 3  -> "Bronze"   
    else -> "L'important c'est de participer"   
}   
// medaille est egal à "Or"

Dans cet exemple, l'affectation de la variable info se fait sur le when.

* 1. Les répétitives
     1. La boucle for

Permet de répéter un bloc d'instructions un nombre de fois défini.

Exemple d'une boucle for définie avec un intervalle d'entiers

for(i in 1..10)   
{   
    println(i)   
}   
   
Résultat :   
   
1   
   
2   
   
3   
   
4   
   
5   
   
6   
   
7   
   
8   
   
9   
   
10

Exemple d'une boucle for définie avec un intervalle d'entiers   
et une incrémentation de l'index spécifique

for(i in 1..10 step 2)   
{   
    println(i)   
}   
   
Résultat :   
   
1   
   
3   
   
5   
   
7   
   
9

Exemple d'une boucle for définie avec un intervalle de caractères

for(lettre in 'A'..'G')   
{   
    println(lettre)   
}   
   
Résultat :   
   
A   
   
B   
   
C   
   
D   
   
E   
   
F   
   
G

Exemple de boucle foreach

val unTableau = arrayOf("Bleu","Blanc","Rouge")   
   
for(item in unTableau)   
{   
    println(item)   
}   
   
Résultat :   
   
Bleu   
   
Blanc   
   
Rouge

Exemple de boucle foreach en utilisant les indices d'un tableau

val unTableau = arrayOf("Manue","Manuche","Manunue")   
   
for (i in unTableau.indices) {   
    println(unTableau[i])   
}   
   
Résultat :   
   
Manue   
   
Manuche   
   
Manunue

Exemple de boucle foreach récupérant des paires index/valeur

val unTableau = arrayOf("Bleu","Blanc","Rouge")   
   
for ((index, couleur) in unTableau.withIndex()) {   
    println("La valeur à l'index $index est $couleur")   
}   
   
Résultat :   
   
La valeur à l'index 0 est Bleu   
   
La valeur à l'index 1 est Blanc   
   
La valeur à l'index 2 est Rouge

* + 1. La boucle while

Permet de répéter une instruction aussi longtemps que la condition de la boucle est égale à true.

Exemple d'utilisation d'une boucle while

var compteur = 1;   
   
while(compteur <=3)   
{   
    println(compteur)   
    compteur++   
}   
println("Soleil")   
   
Résultat :   
   
1   
   
2   
   
3   
   
Soleil

La boucle do while

Permet d'exécuter une instruction au moins une fois puis l'instruction est répétée aussi longtemps que la condition de la boucle est égale à true.

Exemple d'utilisation d'une boucle do while

var compteur = 1;   
       
do{   
    println(compteur)   
    compteur++   
}while(compteur <=3)   
   
println("Soleil")   
   
Résultat :   
   
1   
   
2   
   
3   
   
Soleil

L'interruption d'une boucle

Return

L'utilisation du mot-clé return dans une boucle permet de quitter la fonction dans laquelle la boucle se trouve.

Break

L'utilisation du mot-clé break dans une boucle permet de la quitter tout en restant dans la fonction contenant la boucle.

Continue

L'utilisation du mot-clé continue dans une boucle permet de passer à l'itération suivante.

Les fonctions

Syntaxe

Voici la syntaxe permettant de créer une fonction :

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

Le mot-clé fun permet de déclarer une fonction.

|  |  |
| --- | --- |
| Paramètres | |
| nomDeLaFonction | Par convention, il est préférable que le nom de la fonction respecte la convention de nommage lowerCamelCase. Généralement, son nom contient au moins un verbe informant sur l'action réalisée par la fonction. |
| paramètres | Liste des paramètres de la fonction. Une fonction peut contenir de zéro à plusieurs paramètres. Les paramètres sont séparés par des virgules. Tous les paramètres sont automatiquement de type val. Il n'est pas possible de définir un paramètre de type var. |
| typeDeRetour | Type de la valeur qui sera retournée par la fonction. Unit si la fonction ne retourne rien. Par défaut, le type est Unit. |

Usage

Les fonctions permettent de factoriser, structurer et ainsi de rendre plus maintenable une application.

Exemple simple de fonction

fun direBonjour(prenom:String):String   
{   
    return "Bonjour $prenom"   
}

La fonction se nomme direBonjour, elle attend en paramètre une variable de type String et retourne une variable de type String. Dans cet exemple, la fonction retournera "Bonjour " suivi du prénom passé en paramètre.

Si la fonction ne retourne rien, il n'est pas nécessaire de définir son type de retour.

Exemple d'une fonction ne retournant aucune valeur

fun afficherBonjour(prenom:String)   
{   
    println("Bonjour $prenom")   
}

La fonction se nomme afficherBonjour et attend en paramètre une variable de type String et ne retourne rien (c'est la raison pour laquelle le type de retour n'est pas défini). Dans cet exemple, la fonction affichera "Bonjour " suivi du prénom passé en paramètre.

L'utilisation des fonctions est similaire aux autres langages.

Exemple d'utilisation d'une fonction

fun creerBonjour(prenom:String):String   
{   
    return "Bonjour $prenom"   
}   
   
fun main(args: Array<String>)   
{   
    val phrase = creerBonjour("Leïla")   
    println(phrase)   
}   
   
Résultat :   
   
Bonjour Leïla

Le premier bloc d'instructions est la définition d'une fonction nommée creerBonjour et dans le second bloc, il y a un appel à cette fonction.

Les paramètres

Syntaxe

Voici la syntaxe permettant de définir un paramètre :

<nomDuParametre>:<type>[=valeurParDefaut]

|  |  |
| --- | --- |
| Paramètres | |
| nomDuParametre | Par convention, il est préférable que le nom d’un paramètre respecte la convention de nommage lowerCamelCase et que son nom soit explicite. |
| type | Type souhaité du paramètre. Cette information est obligatoire. |
| valeurParDefaut | Valeur par défaut du paramètre. La valeur par défaut est facultative. |

Une fonction peut avoir plusieurs paramètres. Pour ce faire, les différents paramètres doivent être séparés par une virgule.

Immutable

Kotlin encourage les développeurs à utiliser le paradigme de programmation fonctionnelle et donc à aller vers l'utilisation de variables immutables, c'est pourquoi les paramètres d'une fonction sont obligatoirement immutables. Les paramètres sont implicitement de type val.

Exemple de tentative de modification d'un paramètre

fun afficherPrenom(prenom:String)   
{   
    prenom = prenom.toUpperCase() //ERREUR DE COMPILATION   
}

Dans cet exemple, le compilateur lève une erreur indiquant : "Val cannot be reassigned".

Il n'est pas possible de forcer un paramètre à devenir mutable.

Exemple de tentative de rendre un paramètre mutable

fun afficherPrenom(var prenom:String) //ERREUR   
{   
   ...   
}

Dans cet exemple, le compilateur lève une erreur indiquant : "'var' on function parameter is not allowed".

Paramètre par défaut

Les paramètres d'une fonction peuvent avoir une valeur par défaut. La valeur est à définir après la déclaration du paramètre avec le signe =. La valeur par défaut est prise en compte lorsque le paramètre n'est pas fourni à la fonction.

Exemple d'un paramètre avec une valeur par défaut

fun afficherPrenom(prenom:String = "Nicolas")   
{   
    println(prenom)   
}   
   
fun main(args: Array<String>)   
{   
    afficherPrenom();   
    afficherPrenom("Anthony");   
       
}   
   
Résultat :   
   
Nicolas   
Anthony

Le premier appel à la fonction afficherPrenom() ne contient aucun paramètre et pourtant la fonction affiche "Nicolas".

Argument nommé

Améliore la lisibilité

Pour améliorer la lisibilité du code, il est possible de nommer les valeurs passées en paramètres d'une fonction. Avec des paramètres nommés, il est possible de fournir les valeurs nécessaires dans n'importe quel ordre.

Exemple d'utilisation des paramètres nommés

fun afficherPrenom(prenom:String = "Nicolas",nom:String="Dupont")   
{   
    println("$prenom $nom")   
}   
   
fun main(args: Array<String>)   
{   
    afficherPrenom(nom = "Anthony",prenom="Anthony");   
}

Optimise l'utilisation des paramètres par défaut

Grâce aux paramètres nommés, il est possible de choisir dans une fonction les paramètres que l'on veut fournir parmi une multitude de paramètres par défaut.

Exemple d'utilisation d'une fonction avec plusieurs paramètres par défaut

fun afficherPrenom(prenom:String =   
"Nicolas",nom:String="Dupont",ville:String="Caen", xp:Int=0)   
{   
    println("$prenom $nom $ville : $xp")   
}   
   
fun main(args: Array<String>)   
{   
    afficherPrenom(prenom = "Anthony",ville="Rennes");   
}

Il est possible d'utiliser des paramètres non nommés et nommés dans l'appel d'une même méthode. Dans ce cas, il est nécessaire de mettre les paramètres nommés à droite des paramètres non nommés qui, eux, doivent suivre l'ordre établi.

Exemple mixte paramètres nommés et non nommés

fun afficherPrenom(prenom:String =   
"Nicolas",nom:String="Dupont",ville:String="Caen", xp:Int=0)   
{   
    println("$prenom $nom $ville : $xp")   
}   
   
fun main(args: Array<String>)   
{   
    afficherPrenom("Anthony",xp=50,ville="Rennes");   
}

Paramètre variadique

Un paramètre variadique est un paramètre qui peut prendre un nombre variable de valeurs.

Syntaxe d'un paramètre variadique

Voici la syntaxe d'un paramètre variadique :

vararg <nomDuParametre > : <type> [=valeurParDefaut]

vararg est le mot-clé pour indiquer qu'une variable est variadique.

|  |  |
| --- | --- |
| Paramètres | |
| nomDuParametre | Par convention, il est préférable que le nom d’un paramètre respecte la convention de nommage lowerCamelCase et que son nom soit explicite. |
| type | Type souhaité du paramètre. Cette information est obligatoire. |
| valeurParDefaut | Valeur par défaut du paramètre. La valeur par défaut est facultative. Elle doit obligatoirement être de type Array. |

Usage

Les valeurs d'un paramètre variadique sont récupérées sous forme de tableau (Array), cela offre aux fonctions une certaine flexibilité. Une fonction peut n'avoir qu'un seul paramètre variadique. Si une fonction contient plusieurs paramètres dont un paramètre variadique, il est judicieux de mettre le paramètre variadique en dernier, auquel cas il sera nécessaire d'utiliser des arguments nommés.

Exemple d'une fonction avec un paramètre variadique

fun afficherPrenoms(vararg prenoms:String=arrayOf("Sylvie","Dominique"))   
{   
    for(prenom in prenoms)   
    {   
        println(prenom)   
    }   
}   
   
fun main(args: Array<String>)   
{   
    afficherPrenoms()   
    println("--------")   
    afficherPrenoms("Vincent","Romain","Guénolé","Julie")   
}   
   
Résultat :   
   
Sylvie   
Dominique   
--------   
Vincent   
Romain   
Guénolé   
Julie

Les fonctions contenant une expression simple

Lorsque les instructions d'une fonction ne se résument qu'à une seule ligne de code, il est possible d'utiliser un = à la place des accolades.

Exemple d'une fonction sans accolades

fun afficherBonjour(prenom:String):String = "Bonjour $prenom"

Les instructions de la fonction se trouvent derrière le signe égal. La fonction retourne "Bonjour " suivi du prénom passé en paramètre.

Les fonctions avec une expression simple permettent d'utiliser l'inférence de type pour le type de retour.

Exemple d'une fonction utilisant l'inférence de type

fun afficherBonjour(prenom:String) = "Bonjour $prenom"

Grâce à l'inférence de type, il n'est pas nécessaire de définir le type de retour. Dans cet exemple, la fonction retournera "Bonjour " suivi du prénom passé en paramètre.

Le type de retour

Explicite

Une fonction avec un bloc d'instructions défini avec des accolades doit obligatoirement indiquer son type de retour, sauf si elle ne retourne rien. Effectivement, le type de retour par défaut d'une fonction est Unit.

* La raison principale pour laquelle l'inférence de type ne se fait pas sur le type de retour est que cela risquerait, pour le développeur, de complexifier la lecture du code.

Implicite

Une fonction sans bloc d'instructions défini par des accolades ne doit pas obligatoirement indiquer son type de retour. Effectivement, l'inférence de type déterminera automatiquement le type de retour de la fonction. De plus, les fonctions ne retournant pas de valeur peuvent ne pas indiquer le type de retour étant donné que le type de retour par défaut est Unit.

Exemple avec une fonction contenant une expression simple

fun afficherBonjour(prenom:String) = "Bonjour $prenom"

L'inférence de type détecte automatiquement que le type de retour sera String.

Exemple avec une fonction contenant un bloc d'instructions

fun direBonjour()   
{   
    println("Bonjour")   
}

Le type de retour est par défaut Unit.

Fonction récursive

Le modificateur de fonction tailrec permet d'optimiser les fonctions récursives. La condition requise est que la dernière opération de la fonction soit un appel à elle-même.

Exemple avec la suite de Conway

tailrec fun conway(debutSuite: String, profondeur: Int) {   
    println(debutSuite)   
    if(profondeur != 0)   
    {   
        var nombre = debutSuite   
        val resultat = StringBuilder()   
        var analyse = nombre[0]   
        nombre = nombre.substring(1) + " "   
        var compteur = 1   
   
        for (item in nombre.toCharArray()) {   
            if (item != analyse) {   
                resultat.append(compteur.toString() + "" + analyse)   
                compteur = 1   
                analyse = item   
            } else {   
                compteur++   
            }   
        }   
        conway(resultat.toString(), profondeur - 1)   
    }   
}

Portée des fonctions

Il est possible de définir des fonctions :

* hors d'une classe,
* dans une classe,
* dans une fonction.

Fonction hors d'une classe

Il est possible de définir une fonction dans un fichier sans que celle-ci soit incluse à l'intérieur d'une classe. Elle sera disponible depuis n'importe quel fichier, de n'importe quelle fonction ou classe. Cela est fortement déconseillé afin d'éviter des effets de bord et un non respect de l'architecture applicative qui pourrait amener à rendre votre application difficilement maintenable.

Fonction dans une classe

Une fonction définie à l'intérieur d'une classe est nommée "fonction membre".

Exemple d'une fonction membre

class Chien   
{   
    fun aboyer()   
    {   
        println("waf waf")   
    }   
}

Fonction dans une fonction

Il est possible de définir une fonction dans une fonction. La fonction "fille" peut accéder aux variables de la fonction "mère". On utilise ce mécanisme lorsque le lien entre les deux fonctions est étroit.

Exemple de fonction dans une fonction

fun direBonjour(langue:String):Boolean   
{   
    var resultat = false;   
   
    fun bonjourArabe()   
    {   
        println("Salam")   
        resultat = true   
    }   
    fun bonjourAnglais()   
    {   
        println("Hello")   
        resultat = true   
    }   
    when(langue)   
    {   
        "arabe" -> bonjourArabe()   
        "anglais" -> bonjourAnglais()   
    }   
    return resultat   
}

Les packages

Présentation

Un programme peut contenir des milliers de fichiers tout comme une bibliothèque peut contenir des milliers de livres. Afin de s'y retrouver, on doit classer les fichiers exactement comme on classe des livres. Pour classer des fichiers, il faut utiliser des packages qui contiendront nos différents fichiers. Par exemple, tous les fichiers permettant d'accéder à une base de données se retrouveront ensemble. Pour créer un package, il faut faire un clic droit sur le nom du projet puis sélectionner "New" puis sélectionner "Package". Il faut ensuite donner un nom au package. Le nom du package doit suivre une convention bien spécifique. Tout d'abord, il faut mettre une extension de nom de domaine comme fr, com, bzh… puis un nom. En général on utilisera le nom de l'entreprise ou du projet puis un autre nom pour détailler encore plus le contenu du package. Le but étant d'avoir un nom unique qui détaille ce qui se trouve à l'intérieur.

Quelques exemples de noms de packages

* fr.eni.contacts.bdd
* fr.eni.exercices
* com.chezbatoo.devinet.ihms
* Il est possible d'avoir plusieurs fichiers avec le même nom s'ils se trouvent dans des packages différents.

Conventions et bonnes pratiques

Respecter les conventions d'organisation et de nommage permet de réduire au maximum sa dette technique.

Voici quelques règles à respecter :

* Un fichier doit toujours se trouver dans un package.
* Les noms de packages doivent respecter la convention lower\_snake\_case.
* Les noms de packages ne doivent pas contenir de mots-clés Kotlin.

Déclaration d'un package

Lorsqu'un fichier se trouve dans un package, la première ligne de ce fichier doit contenir le mot-clé package suivi du nom du package dans lequel il se trouve.

Exemple d'un fichier se trouvant dans le package fr.eni.exemples

package fr.eni.exemples   
   
fun main(args: Array<String>)   
{   
    println("Parlons package")   
}

* En général, la déclaration du package est générée automatiquement par l'IDE.

Importation d'un package

Le nom complet d'une fonction, d'une propriété ou d'une classe contient le nom du package dans lequel elle se trouve puis le nom de l'élément.

Exemple

package fr.eni.exemples   
   
class Animal   
   
fun direAurevoir()   
{   
    println("Au revoir")   
}

Le nom complet de la fonction direAuRevoir est fr.eni.exemples.direAuRevoir.

Lorsque les fonctionnalités d'un fichier utilisent des fonctions ou des classes d'un autre fichier, vous devez les importer, sauf si elles se trouvent dans le même package.

Il est possible :

* d'importer spécifiquement la fonction ou la classe qui sera utilisée,
* d'importer la totalité du package,
* d'importer des propriétés et des énumérateurs.

Le mot-clé import permet de faire les importations. Il est possible d'avoir plusieurs importations dans un même fichier. La définition des importations se fait en haut du fichier.

Démonstration.kt

package fr.eni.exemples   
   
class Animal   
   
fun direAurevoir()   
{   
    println("Au revoir")   
}

Cet exemple est le contenu du fichier nommé "Demonstration.kt". Ce fichier se trouve dans le package fr.eni.exemples. Dans ce fichier se trouvent une classe nommée Animal et une fonction nommée direAuRevoir.

Exemple d'importation

package fr.eni.utilisations   
   
import fr.eni.exemples.direAurevoir   
   
fun main(args: Array<String>)   
{   
    direAurevoir();   
}

Cet exemple est le contenu du fichier nommé "Test.kt". Ce fichier se trouve dans le package fr.eni.utilisations. La fonction direAuRevoir est importée, il est ainsi possible de l'utiliser.

Exemple d'importation

package fr.eni.utilisations   
   
import fr.eni.exemples.\*   
   
fun main(args: Array<String>)   
{   
    direAurevoir();   
}

Cet exemple est le contenu du fichier nommé "Test.kt". Ce fichier se trouve dans le package fr.eni.utilisations. Le contenu du package fr.eni.exemples est importé, il est ainsi possible de l'utiliser.

Démonstration

La démonstration suivante est un programme permettant d'afficher les nombres premiers compris entre 0 et 100.

Pour rappel, un nombre est premier s'il admet uniquement deux diviseurs différents entiers et positifs. 2, 3, 5,7 sont des nombres premiers par exemple.

/\*\*   
 \* Point d'entrée du programme.   
 \*/   
fun main(args: Array<String>)   
{   
    for(nombreATester in 1..100) {   
        if(isPremier(nombreATester))   
        {   
            println("$nombreATester est premier")   
        }   
    }   
}   
   
/\*\*   
 \* Fonction permettant de déterminer si un nombre est premier.   
 \*/   
fun isPremier(nombre:Int):Boolean   
{   
    var nombreDeDiviseur = 0;   
   
    for(i in 1..nombre)   
    {   
        if(nombre%i == 0)   
        {   
            nombreDeDiviseur++   
        }   
    }   
    return (nombreDeDiviseur == 2)   
}