CAHIER DE PROGRAMMATION

Apprendre Kotlin

Cours et exercices corrigés

Du premier message dans la console à un carnet de lecture que tu peux programmer, tester et faire évoluer.

PREMIER PROGRAMME

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

Console

```
Bonjour, Kotlin !
```

14 chapitres **2 bilans** **1 projet final**

Niveau débutant / Exemples pour Kotlin sur la JVM

Ouvrir le sommaire

Sommaire

Clique sur un titre pour ouvrir la leçon ou retrouver un exercice.

Prise en main

Où écrire ton code 3

Les bases

01 Le premier programme 4
02 Variables et types 6
03 Calculs et comparaisons 8
04 Conditions 10
05 Boucles 12
Bilan · Jours de lecture 14

Structurer un programme

06 Fonctions 15
07 Saisie et nullabilité 17
08 Collections 19
09 Lambdas 21
10 Classes et objets 23
Bilan · Relevé de lecture 25

Approfondir le langage

11 Interfaces et types scellés 26
12 Génériques et extensions 28
13 Erreurs et tests 30
14 Coroutines 32

Projet · Carnet de lecture

Le cahier des charges 34
Les données 35
Les commandes 36
Les vérifications 37
La saisie au clavier 38

Annexes

Installer les coroutines 39
Types de fonctions et let 40
Héritage et classes abstraites 41
Lire un écran Android 42
Autoévaluation et références 43

Choisir où écrire ton code

Un programme est un texte que tu écris pour faire exécuter des instructions à l'ordinateur. Pour les premières leçons, il te suffit d'un navigateur. Pour le projet final, une console locale te permettra de dialoguer au clavier avec ton programme.

Pour commencer aujourd'hui

Ouvre Kotlin Playground. Sélectionne la cible JVM si un choix de cible s'affiche. Efface le code de démonstration, puis recopie le premier programme du chapitre 1. Le triangle d'exécution lance le programme. Le texte qui apparaît dans la zone de sortie est ce que ton programme a affiché.

Garde un seul exemple à la fois dans l'éditeur. Chaque bloc portant la mention « Kotlin » est un petit programme entier. Les blocs intitulés « Console » montrent ce que tu peux lire à l'écran. Les fragments du projet final sont signalés séparément et s'assemblent dans un seul fichier.

Quand tu voudras saisir du texte au clavier

Dans IntelliJ IDEA, choisis `New Project`, puis `Kotlin`, et donne au projet un nom comme `MonAtelier`. Choisis le système de construction `IntelliJ` et un JDK, par exemple 21. Si nécessaire, `Download JDK` permet d'en installer un. Le JDK fournit les outils pour exécuter Kotlin sur la machine virtuelle Java, la JVM.

Active `Add sample code`, crée le projet et ouvre `Main.kt`. Remplace son contenu par l'exemple du cours. Clique sur le triangle à côté de `main`, puis dans la fenêtre `Run` pour taper tes réponses. Le chapitre sur les coroutines a sa propre préparation en annexe. Guide de création d'un projet console

Comment utiliser les corrigés

Avant de lancer chaque exemple, écris la sortie que tu attends. Si ta prédiction diffère, change une seule valeur et relance. Dans un atelier, cache la partie « Corrigé » jusqu'à ce que tu aies obtenu un premier résultat. L'archive qui accompagne le cours contient les programmes à ouvrir ou à recopier.

Faire parler la console

Un programme est une suite d'instructions. Ton premier programme donnera une seule instruction à l'ordinateur. Il lui demandera d'écrire une phrase. Ouvre Kotlin Playground dans ton navigateur, choisis la cible JVM si un choix de plateforme apparaît, puis remplace tout le code proposé par ces trois lignes.

KOTLIN
```kotlin
fun main() {
    println("Bonjour, Kotlin !")
}
```

Clique sur le triangle de lecture, aussi appelé Run. Le compilateur vérifie que le texte respecte les règles du langage et le traduit pour qu'il puisse être exécuté. Tu verras le résultat dans la console, la zone où le programme écrit ses messages.

CONSOLE
```
Bonjour, Kotlin !
```

Lire les trois lignes

Le mot `fun` annonce une fonction, un groupe d'instructions auquel on donne un nom. Ici, `main` est le nom du point de départ. Les parenthèses vides signifient qu'on ne lui donne aucune information. Les accolades `{` et `}` délimitent les instructions qu'elle contient.

À la deuxième ligne, `println` demande d'afficher ce qui se trouve entre ses parenthèses, puis de passer à la ligne. Les guillemets droits délimitent le texte. Compare ce que tu as tapé avec ce qui apparaît dans la console. Quels signes ont disparu ?

Remplace uniquement le mot `Kotlin` par ton prénom. Relance le programme. Tu viens de modifier une valeur tout en gardant la même instruction.

Imprimer un marque-page

Tu veux glisser un marque-page dans un roman. Écris un programme qui affiche le titre du livre sur une ligne, puis le numéro de la page sur la suivante. Pour cet exercice, utilise exactement ces deux lignes de sortie. Essaie d'abord avec le seul programme de la page précédente comme modèle.

CONSOLE
```
Le livre de la jungle
Page 12
```

Avant de cliquer sur Run, indique du doigt la ligne de ton code qui produira chaque ligne de sortie. Deux instructions peuvent se suivre entre les mêmes accolades. Les espaces placés devant les instructions permettent de voir qu'elles appartiennent à `main`.

Une correction possible

KOTLIN
```
fun main() {
    println("Le livre de la jungle")
    println("Page 12")
}
```

Kotlin exécute ici les instructions de haut en bas. Si tu échanges les deux lignes qui commencent par `println`, le numéro s'affiche en premier. Vérifie ta prédiction, puis remets le titre en tête.

Provoquer une erreur et la réparer

Enlève le guillemet droit qui suit le nombre 12 dans le code, puis clique sur Run. Le compilateur cherche la fin du texte. Son premier message peut pointer la fin de la ligne ou le symbole suivant. Relis la ligne signalée depuis son début et rétablis le guillemet. En relançant, tu dois retrouver les deux lignes attendues.

Donner un nom aux valeurs

Un lecteur a lu 12 pages de son roman, puis 3 pages de plus. Pour conserver le total pendant l'exécution, on utilise une variable. Lis le programme et prédis le nombre qui s'affichera.

KOTLIN
```kotlin
fun main() {
    val titre: String = "L'île au trésor"
    var pages: Int = 12
    pages = pages + 3
    println("$titre : $pages pages")
}
```

`val titre` donne un nom au texte du titre. Les deux points introduisent son type, `String`, qui représente du texte. Le signe `=` donne sa valeur au nom situé à gauche. Avec `val`, tu peux relire cette valeur aussi souvent que nécessaire. Avec `var`, tu peux aussi lui en attribuer une nouvelle.

Pour comprendre `pages = pages + 3`, commence par la droite. Tu lis l'ancienne valeur, 12, et tu lui ajoutes 3. Tu ranges ensuite 15 dans `pages`. Dans une chaîne, `$pages` insère la valeur au moment de l'affichage. La console écrit « L'île au trésor : 15 pages ».

Choisir un type

- `Int` représente un entier, comme `12`. `Double` représente un nombre à virgule flottante, comme `7.5`, écrit avec un point.
- `Boolean` admet deux valeurs, `true` et `false`.
- `Char` représente ici un caractère, comme `'A'`. `String` peut contenir un texte entier ou même être vide, comme `""`.

Kotlin sait souvent déduire le type. En écrivant `var pages = 12`, tu obtiens aussi une variable de type `Int`.

Tenir un carnet de lecture

Tu commences à la page 8 d'un livre et avances de 5 pages. Déclare le titre « Ossip », la lettre de son rayon, B, et ta page actuelle. Choisis entre `val` et `var` pour chaque information. Fais afficher « Ossip (B) : page 8 », puis « Reprise à la page 13 ». Commence avec un `main` vide. Où placeras-tu le calcul pour que la première ligne indique encore 8 ?

Vérifier ta solution

KOTLIN
```
fun main() {
    val titre = "Ossip"
    val rayon = 'B'
    var page = 8
    println("$titre ($rayon) : page $page")
    page = page + 5
    println("Reprise à la page $page")
}
```

Le titre et le rayon gardent leur valeur pendant le programme. La page change entre les deux affichages. Une ligne déjà écrite dans la console conserve son contenu même si la valeur de `page` change ensuite.

Deux expériences

Écris `val page = 8` et relance. Le compilateur signale la tentative d'attribuer une seconde valeur à `page`. Rétablis `var`. Déplace ensuite le premier affichage juste après le calcul. Cette fois, le programme peut s'exécuter. Les deux messages utilisent 13. La compilation vérifie les règles du langage. C'est à toi de vérifier que le résultat raconte bien ce que tu voulais.

Calculer et comparer

Tu répartis 17 livres entre 5 étagères. Combien peux-tu poser sur chacune si toutes doivent en recevoir autant ? Combien en restera-t-il sur le chariot ? Pour répondre, Kotlin propose deux opérations complémentaires.

KOTLIN
```
fun main() {
    val livres = 17
    val etageres = 5
    println(livres / etageres)
    println(livres % etageres)
    println(livres.toDouble() / etageres)
}
```

La console affiche successivement `3`, `2` et `3.4`. Avec deux `Int`, la division tronque le quotient vers zéro. Ainsi `17 / 5` donne `3`, et `-17 / 5` donne `-3`. L'opérateur `%` donne le reste. Tu peux contrôler le résultat sur papier, car 5 fois 3, plus 2, redonne 17. Pour calculer une moyenne décimale, `toDouble()` convertit un des nombres avant la division. Dans `(livres / etageres).toDouble()`, la division entière aurait déjà donné 3.

Obtenir une réponse vraie ou fausse

La comparaison `livres >= 15` produit `true`. Essaie aussi `<`, `<=`, `>`, `==` pour « égal à » et `!=` pour « différent de ». Compare des valeurs avec `==` et attribue une valeur à une variable avec `=`.

Pour combiner deux conditions, `a && b` exige que les deux soient vraies. `a || b` exige qu'au moins une soit vraie. `!a` inverse la réponse. Avec nos valeurs, prédis la réponse de `(livres > 10) && (etageres == 4)`. La seconde comparaison donne `false`, ce qui suffit pour que le résultat entier soit `false`.

Préparer des cartons

Chaque carton plein contient 6 livres. Il y a 20 livres à envoyer et 3 cartons disponibles. Écris un programme qui affiche le nombre de cartons pleins, le nombre de livres restants et une réponse vraie ou fausse à la question « Peut-on tout expédier ? ». Fais d'abord le calcul de tête. Les livres qui restent doivent eux aussi trouver une place.

Une correction possible

KOTLIN
```
fun main() {
    val livres = 20
    val capacite = 6
    val cartons = 3
    val pleins = livres / capacite
    val restants = livres % capacite
    val toutRentre = livres <= cartons * capacite
    println("Cartons pleins : $pleins")
    println("Livres restants : $restants")
    println("Envoi possible : $toutRentre")
}
```

Tu dois lire `3`, `2` et `false` dans les trois messages. Si tu as comparé seulement `pleins` et `cartons`, tu as laissé les deux derniers livres sur le quai. Dans la dernière formule, Kotlin effectue la multiplication avant la comparaison. Il compare ainsi 20 livres à 18 places.

Essayer les valeurs voisines

Recommence avec 18 livres. Les cartons sont exactement pleins et la dernière réponse devient `true`. Remplace ensuite 18 par 19. Un seul livre supplémentaire fait basculer la réponse. Ces deux essais contrôlent précisément le choix de `<=`. Garde l'habitude de tester une valeur juste à la limite et une valeur voisine chaque fois qu'un programme prend une décision.

Choisir avec if et when

Une bibliothèque prête deux livres à la fois. Quand une personne en a déjà emprunté deux, l'écran doit lui demander d'en rendre un. Une condition permet au programme de choisir son message.

KOTLIN
```
fun main() {
    val empruntes = 1
    val message = if (empruntes < 2) {
        "Tu peux emprunter un livre."
    } else {
        "Rends d'abord un livre."
    }
    println(message)
}
```

Kotlin évalue d'abord la comparaison entre parenthèses. Si elle donne `true`, il exécute la première branche. Sinon, il exécute la branche `else`. Ici, la dernière expression de la branche choisie donne sa valeur à `message`. Essaie les valeurs 1 et 2. Pourquoi l'écran change-t-il pour la seconde ?

Choisir parmi plusieurs cas

KOTLIN
```
fun main() {
    val jour = "samedi"
    val ouverture = when (jour) {
        "samedi" -> "10 h"
        "dimanche" -> "fermé"
        else -> "9 h"
    }
    println(ouverture)
}
```

Dans `when (jour)`, Kotlin compare la valeur de `jour` aux cas indiqués de haut en bas. La première branche qui correspond donne le résultat. La flèche `->` sépare le cas de son résultat. Ici, `else` fournit un résultat à toutes les autres valeurs. On suppose que `jour` contient un nom de jour correctement écrit en minuscules. Essaie « samedi », puis « lundi ». Quel résultat obtiens-tu si tu tapes « Samedi » avec une majuscule ? La comparaison distingue les majuscules des minuscules, et cette nouvelle valeur tombe dans `else`.

Déterminer le tarif d’entrée

Un musée demande une règle précise. L’entrée est gratuite de 0 à 5 ans inclus, coûte 4 euros de 6 à 17 ans inclus, puis 8 euros dès 18 ans. Un âge négatif doit afficher « Âge invalide ». Avant de coder, écris sur papier le message attendu pour −1, 0, 5, 6, 17 et 18 ans. Essaie ensuite de traduire ces phrases en conditions.

Une correction possible

KOTLIN
```kotlin
fun main() {
    val age = 6
    val tarif = when {
        age < 0 -> "Âge invalide"
        age < 6 -> "Gratuit"
        age < 18 -> "4 euros"
        else -> "8 euros"
    }
    println(tarif)
}
```

Ce `when` s’écrit sans valeur entre parenthèses. Chaque branche pose une question qui produit un booléen. À 6 ans, les deux premières questions donnent `false` et la troisième donne `true`. La console affiche donc « 4 euros ». Pour finir les essais, remplace 6 par chacune des autres valeurs de ta liste. Pour saisir un âge négatif en Kotlin, écris par exemple `-1`.

L’ordre des branches porte une partie de la règle. Quand on atteint `age < 18`, on sait déjà que l’âge vaut au moins 6. Place maintenant `age < 18` avant `age < 6` et teste 5. Tu obtiens un tarif payant. Corrige l’ordre et retrouve « Gratuit » pour que ton programme respecte de nouveau la grille.

Répéter avec for et while

Pour numéroter quatre casiers, tu pourrais écrire quatre appels à `println`. Si demain il y en a trente, une boucle évitera de recopier la même instruction. Lis le code ci-dessous et prédis à la fois le premier et le dernier numéro.

KOTLIN
```
fun main() {
    for (casier in 1..4) {
        println("Casier $casier")
    }
}
```

La plage `1..4` contient 1, 2, 3 et 4. À chaque tour de `for`, `casier` prend la valeur suivante, puis le programme exécute les instructions entre accolades. Avec `1 until 4`, la borne de droite est exclue et l'affichage s'arrête à 3.

Continuer tant qu'une condition est vraie

KOTLIN
```
fun main() {
    var restants = 3
    while (restants > 0) {
        println("$restants à ranger")
        restants = restants - 1
    }
    println("Étagère prête")
}
```

`while` relit la condition avant chaque tour. Au premier passage, le programme affiche 3 puis enlève un livre. Les passages suivants affichent 2 et 1. Dès que `restants` atteint 0, le programme quitte la boucle et affiche « Étagère prête ». Avec une valeur initiale de 0, ce dernier message est le seul affiché. Essaie avant de continuer.

Dans cet exemple, enlever un livre rapproche le programme de sa condition d'arrêt. Si tu oublies cette instruction avec une valeur initiale de 3, `restants > 0` restera vraie à chaque tour.

Suivre un total tour par tour

Lundi, tu lis 2 pages. Mardi, tu en lis 4. Mercredi, tu en lis 6. Écris une boucle qui affiche le total cumulé au terme de chaque journée. Numérote les jours de 1 à 3 et remarque que le nombre de nouvelles pages vaut toujours deux fois le numéro du jour. Avant de programmer, trace sur papier trois colonnes pour le jour, les nouvelles pages et le total.

Une correction possible

KOTLIN
```
fun main() {
    var total = 0
    for (jour in 1..3) {
        val nouvelles = 2 * jour
        total = total + nouvelles
        println("Jour $jour : $total pages")
    }
}
```

CONSOLE
```
Jour 1 : 2 pages
Jour 2 : 6 pages
Jour 3 : 12 pages
```

Le premier tour démarre avec un total de 0. Le deuxième démarre avec 2, y ajoute 4 et termine avec 6. Trace toi-même le troisième tour. La variable `total` est créée avant la boucle, donc sa valeur se transmet d'un tour au suivant. Si tu la crées à l'intérieur, elle repart à zéro à chaque passage. Déplace sa déclaration pour observer l'effet, puis remets-la à sa place.

Pour finir, étends la plage à `1..4`. Prédis le dernier message avant l'exécution. Le quatrième jour ajoute 8 au total de 12. Si tu lis « Jour 4 : 20 pages », ton accumulateur a bien conservé les lectures précédentes.

Combien de jours de lecture ?

Tu lis dix pages par jour. Combien de jours de lecture faut-il pour finir un livre de 23 pages ? Le dernier jour, tu t'arrêtes à la dernière page. Écris une boucle qui affiche le nombre de pages restantes après chaque journée, puis le nombre total de jours.

Sur papier

Pars de 23 et retire dix. Recommence avec 13. Il reste alors trois pages. Après le troisième jour, le compteur doit indiquer zéro. Essaie aussi de partir de zéro page et de vingt pages. Ces deux cas permettent de repérer un tour de boucle en trop.

Corrigé

KOTLIN
```kotlin
fun main() {
    var restantes = 23
    var jours = 0
    while (restantes > 0) {
        val lues = if (restantes < 10) restantes else 10
        restantes -= lues
        jours++
        println("Jour $jours : $restantes restantes")
    }
    println("Jours de lecture : $jours")
}
```

CONSOLE
```
Jour 1 : 13 restantes
Jour 2 : 3 restantes
Jour 3 : 0 restantes
Jours de lecture : 3
```

`-=` soustrait la valeur de droite au compteur. `jours++` ajoute 1 à `jours`. Pour 23 pages, les trois valeurs de `lues` sont 10, 10 et 3. Pour zéro page, Kotlin teste la condition avant d'entrer dans la boucle et affiche directement le bilan. Pour vingt pages, la deuxième journée suffit.

Vérifie ta compréhension

Un ami propose `23 / 10` pour calculer la réponse. Quel résultat obtient-il ? La division de deux entiers donne 2. Retrouve sur ta trace les trois pages que ce calcul a laissées de côté.

Donner un nom à un calcul

Tu connais la durée d'une séance et le nombre de séances prévues. Pour afficher la durée totale, tu pourrais multiplier ces deux nombres partout où tu en as besoin. Si tu donnes un nom à ce calcul, tu peux le réutiliser et vérifier son résultat à part. C'est le rôle d'une fonction.

KOTLIN
```
fun dureeTotale(minutes: Int, seances: Int = 1): Int {
    val total = minutes * seances
    return total
}

fun main() {
    val lundi = dureeTotale(25)
    val mardi = dureeTotale(seances = 2, minutes = 15)
    println(lundi + mardi)
}
```

Suis la valeur jusqu'à la console

`minutes` et `seances` sont les paramètres. Les valeurs fournies à l'appel sont les arguments. Dans `minutes: Int`, les deux-points introduisent le type attendu pour ce paramètre. Après la parenthèse, `: Int` annonce le type du résultat. Au premier appel, Kotlin utilise la valeur par défaut de `seances`. Au second, les noms indiquent quel argument va où.

`return` renvoie le résultat au point d'appel. Ici, les deux appels produisent 25 et 30, puis `println` affiche leur somme, 55. Si tu effaces uniquement le `println`, les calculs ont toujours lieu et la console reste vide.

Le nom `total` existe à l'intérieur de la fonction. Dans `main`, tu accèdes au résultat avec `lundi` ou `mardi`. Kotlin déduit leur type `Int`. Pour une seule expression, tu peux même écrire `fun doubler(n: Int) = n * 2`. Le compilateur y déduit aussi le type du résultat.

Préparer les tables

Une salle dispose de tables de quatre places. Écris une fonction `tablesNecessaires` qui reçoit un nombre d'invités et renvoie le nombre de tables à ouvrir. Une table à moitié pleine compte comme une table entière. Fais de la capacité un second paramètre qui vaut quatre par défaut.

Essaie d'abord sur papier. Pour 8 invités, tu attends 2 tables. Pour 9, il en faut 3. Combien en faut-il pour 0 invité ? Ton calcul doit fonctionner pour un nombre d'invités positif ou nul et une capacité strictement positive. Tu peux utiliser `%` pour compter les personnes qui restent à placer.

Corrigé

KOTLIN
```
fun tablesNecessaires(invites: Int, places: Int = 4): Int {
    val completes = invites / places
    val reste = invites % places
    return if (reste == 0) completes else completes + 1
}

fun main() {
    check(tablesNecessaires(0) == 0)
    check(tablesNecessaires(8) == 2)
    check(tablesNecessaires(9) == 3)
    println(tablesNecessaires(invites = 9, places = 3))
}
```

La division entière compte les tables complètes. Le reste dit s'il faut ouvrir une dernière table. Ainsi, pour 9 invités et 4 places, `completes` vaut 2 et `reste` vaut 1. La dernière ligne utilise des tables de trois places et affiche 3.

`check(condition)` poursuit silencieusement si la condition est vraie et interrompt le programme si elle est fausse. Remplace temporairement `completes + 1` par `completes`. Parmi les trois vérifications, seule celle portant sur 9 invités détecte la table manquante. Rétablis le calcul pour terminer.

Lire une réponse incertaine

Quand tu demandes une durée au clavier, la personne peut saisir `25`, écrire `demain` ou laisser la ligne vide. La lecture peut aussi arriver à la fin de l'entrée. Comment représenter une valeur absente ? En Kotlin, `String?` accepte une chaîne ou la valeur `null`. Une chaîne vide `""` reste une chaîne dont la longueur vaut zéro.

KOTLIN
```
fun afficherDuree(saisie: String?) {
    val minutes = saisie?.trim()?.toIntOrNull()
    if (minutes == null) {
        println("Entre un nombre entier.")
        return
    }
    println("Dans dix minutes : ${minutes + 10}")
}

fun main() {
    afficherDuree(" 25 ")
    afficherDuree("demain")
    afficherDuree(null)
}
```

Déplie la première ligne

Chaque `?.` exécute l'opération qui suit si la valeur à gauche existe. Sinon, l'expression donne `null`. `trim()` enlève les espaces aux extrémités. `toIntOrNull()` tente de convertir le texte en `Int` et renvoie `null` si la conversion échoue. Le premier appel affiche donc 35 dans le message. Dans le texte affiché, `$minutes` insérerait la valeur seule. `${minutes + 10}` calcule d'abord ce qui est entre accolades, puis insère le résultat. Les deux autres appels invitent à entrer un entier.

Une fois le cas `null` traité par `return`, Kotlin sait que `minutes` est un `Int` dans la dernière instruction. Cette déduction s'appelle un « smart cast ». Quand une valeur de remplacement a du sens, écris par exemple `val nom = surnom ?: "Anonyme"`. `?:` choisit la valeur de droite uniquement si celle de gauche vaut `null`.

Avec `minutes!!`, tu demandes à Kotlin d'accepter la valeur comme présente. Une valeur `null` fait alors échouer le programme à l'exécution. Une branche explicite te permet ici de guider la personne qui a saisi la durée.

Valider des places

Un atelier accepte des réservations de 1 à 6 places. Écris une fonction `placesValides` qui reçoit une chaîne éventuellement absente et renvoie un `Int?`. Elle doit accepter `" 2 "` et refuser `"0"`, `"7"`, `"deux"` et `null`. Décide d'abord où vérifier le format et où vérifier les bornes. Dans la console locale, affiche ensuite la réponse correspondant à la ligne saisie.

Corrigé

KOTLIN
```kotlin
fun placesValides(texte: String?): Int? {
    val places = texte?.trim()?.toIntOrNull() ?: return null
    return if (places in 1..6) places else null
}

fun main() {
    println("Combien de places (1 à 6) ?")
    val ligne = readlnOrNull()
    val places = placesValides(ligne)
    if (places == null) println("Réservation impossible.")
    else println("Places réservées : $places")
}
```

Dans une console locale, `readlnOrNull()` lit une ligne après l'appui sur Entrée et renvoie `null` en fin d'entrée. Dans le Playground, remplace `val ligne = readlnOrNull()` par `val ligne = " 2 "` pour reproduire cette saisie. Consulte le résultat, puis remplace l'entrée par `"6"` et par `"7"`. Tu peux aussi écrire `val ligne: String? = null` pour simuler la fin de l'entrée.

Si la conversion échoue, `?: return null` quitte immédiatement la fonction. Sinon, `places` contient un entier que tu peux comparer aux deux bornes. Avec `"0"`, la conversion réussit puis la vérification des places échoue. Avec `"deux"`, la conversion échoue dès le départ. Ces deux chemins aboutissent au même message, pour des raisons différentes.

Choisir une collection

Une billetterie conserve l'ordre des inscriptions, les villes représentées et les capacités par salle. Ces trois informations ne se consultent pas de la même manière. Une `List` garde l'ordre et accepte les doublons. Un `Set` conserve une seule occurrence de chaque valeur. Une `Map` associe une clé à une valeur. Ici, chaque nom de salle sert de clé pour retrouver sa capacité.

KOTLIN
```
fun main() {
    val inscrits: List<String> = listOf("Lina", "Noé", "Lina")
    val villes: Set<String> = setOf("Lyon", "Lyon", "Lille")
    val capacites: Map<String, Int> = mapOf("A" to 8, "B" to 6)
    println(inscrits[0])
    println(inscrits.getOrNull(3))
    println(villes.size)
    println(capacites["B"])
    for (nom in inscrits) println(nom)
}
```

Prédis quatre résultats

Les indices d'une liste commencent à 0. `inscrits[0]` vaut donc `"Lina"`. Les trois indices valides sont 0, 1 et 2. `getOrNull(3)` renvoie `null`, tandis que l'accès direct `[3]` déclencherait une erreur à l'exécution. Le nombre de villes distinctes vaut 2. La capacité de la salle B vaut 6. La boucle affiche enfin les trois inscriptions, y compris le second passage de Lina.

`capacites["C"]` renverrait `null` puisque cette clé est absente. L'expression `"A" to 8` forme la paire utilisée pour associer une salle et sa capacité.

Les types `List`, `Set` et `Map` offrent une interface de lecture. Pour créer une liste dont tu peux changer le contenu, écris `val attente = mutableListOf<String>()`, puis `attente.add("Mila")`. `val` garde la même référence à la liste tout en autorisant les opérations offertes par sa version mutable.

Tenir une présence

À l'entrée d'un atelier, Lina présente son badge, puis Noé, puis Lina une seconde fois. Construis une liste des arrivées où chaque personne apparaît une seule fois, dans l'ordre de son premier passage. Affiche ensuite le troisième nom, ou le texte « Aucune » si cette place est vide. Essaie avec une seule boucle. Prédis ce que tu liras pour une liste de passages vide.

Corrigé

KOTLIN
```kotlin
fun presences(passages: List<String>): List<String> {
    val arrives = mutableListOf<String>()
    val vus = mutableSetOf<String>()
    for (nom in passages) {
        if (vus.add(nom)) arrives.add(nom)
    }
    return arrives
}

fun main() {
    val noms = presences(listOf("Lina", "Noé", "Lina"))
    println(noms)
    println(noms.getOrNull(2) ?: "Aucune")
    check(presences(emptyList()).isEmpty())
}
```

`vus.add(nom)` renvoie `true` lorsque le nom vient d'être ajouté. Au second passage de Lina, il renvoie `false`, donc la liste `arrives` conserve deux éléments. La console affiche `[Lina, Noé]`, puis `Aucune`. `emptyList()` crée une liste vide et `isEmpty()` vérifie qu'une collection ne contient aucun élément.

Ajoute `"Mila"` en fin de liste avant l'appel. Le troisième nom devient `Mila`. Ici, l'exercice prend les noms comme identifiants. Dans une véritable liste de présence, deux personnes peuvent avoir le même prénom. Tu pourrais alors appliquer exactement le même raisonnement au numéro du badge.

Pour ce besoin courant, la bibliothèque Kotlin propose aussi `passages.distinct()`. Compare sa sortie à celle de ta boucle sur ces deux jeux de données.

Passer une action en argument

Tu disposes de quatre durées en minutes. Tu veux garder celles d'au moins vingt minutes, puis les exprimer en secondes. Tu pourrais écrire deux boucles. Kotlin propose déjà ces parcours et te laisse décrire ce qui change d'un problème à l'autre, ici le critère de sélection et le calcul à effectuer.

KOTLIN
```
fun main() {
    val minutes = listOf(10, 20, 35, 15)
    val longues = minutes.filter { duree -> duree >= 20 }
    val secondes = longues.map { it * 60 }
    println(longues)
    println(secondes)
    println(minutes)
}
```

Lis les accolades comme une petite fonction

Dans `{ duree -> duree >= 20 }`, la valeur avant `->` est le paramètre. La dernière expression fournit le résultat de cette petite fonction, appelée lambda. Pour 10, la réponse est `false`, donc `filter` écarte 10. Pour 20 et 35, elle est `true`. La première ligne affichée est `[20, 35]`.

`map` transforme chaque élément conservé. Quand une lambda attend un seul paramètre, `it` permet de le désigner sans le nommer. `{ it * 60 }` équivaut ici à `{ duree -> duree * 60 }`. La deuxième ligne affiche `[1200, 2100]`. Le dernier `println` retrouve les quatre durées d'origine.

Une fonction qui reçoit une fonction en argument ou qui en renvoie une porte le nom de fonction d'ordre supérieur. `filter` attend une fonction qui répond par un booléen. `map` attend une fonction qui produit la nouvelle valeur. Essaie le seuil 40. Les deux listes calculées seront vides.

Préparer des étiquettes

Un formulaire t'envoie les noms `" Noé "`, `""`, `"Mila"` et `"  "`. Tu veux préparer des étiquettes propres. Enlève les espaces autour des noms, retire toute ligne désormais vide, puis trie par longueur croissante. Avant d'écrire le code, dessine trois listes sur une feuille, une après chaque étape. À quel moment le dernier nom devient-il vide ?

Corrigé

KOTLIN
```
fun etiquettes(saisies: List<String>): List<String> {
    return saisies
        .map { nom -> nom.trim() }
        .filter { nom -> nom.isNotEmpty() }
        .sortedBy { nom -> nom.length }
}

fun main() {
    val saisies = listOf(" Noé ", "", "Mila", "   ")
    val noms = etiquettes(saisies)
    println(noms)
    check(noms == listOf("Noé", "Mila"))
    check(etiquettes(listOf(" ")).isEmpty())
}
```

`map` traite les quatre chaînes et fournit encore quatre éléments. Après le nettoyage, la deuxième et la quatrième chaîne sont vides. `filter` garde les deux autres. `sortedBy` utilise la longueur de chaque nom comme clé de tri. La sortie est `[Noé, Mila]`. Inverse l'ordre des deux prénoms dans l'entrée et observe que le tri retrouve la même sortie.

Intervertis maintenant `map` et `filter` en gardant les mêmes expressions. La chaîne faite de trois espaces passe le filtre, puis devient vide. Le premier `check` arrête le programme. Déplace le second avant lui pour isoler le cas de la chaîne composée d'espaces. Rétablis l'ordre initial. Comme chaque étape reçoit la liste produite par la précédente, tu peux comprendre le programme en lisant la chaîne d'appels de haut en bas.

Une classe, plusieurs objets

Tu veux que chaque séance possède un titre et son propre nombre de places réservées. Une classe décrit comment créer un objet et ce que tu peux faire avec lui. Un appel à son constructeur crée une instance. Les deux séances ci-dessous ont la même structure et chacune garde son propre compteur.

KOTLIN
```
class Seance(val titre: String) {
    var reservations = 0
        private set

    fun reserver() {
        reservations += 1
    }
}

fun main() {
    val matin = Seance("Kotlin")
    val soir = Seance("Dessin")
    matin.reserver()
    println("${matin.titre} : ${matin.reservations}")
    println("${soir.titre} : ${soir.reservations}")
}
```

Où vit le compteur ?

Dans `class Seance(val titre: String)`, les parenthèses décrivent le constructeur principal. `val` fait de `titre` une propriété accessible sur l'instance. Ainsi, `matin.titre` lit le titre de la première séance. `reserver()` est une fonction membre, aussi appelée méthode. Dans son corps, `reservations` désigne la propriété de l'objet sur lequel la méthode a été appelée.

La sortie contient `Kotlin : 1`, puis `Dessin : 0`. Ajoute `val autre = matin` et appelle `autre.reserver()` avant les affichages. Le compteur du matin monte à 2. La nouvelle variable désigne le même objet. Pour obtenir une séance indépendante, il faut appeler de nouveau `Seance(...)`.

Grâce à `private set`, du code extérieur peut consulter `matin.reservations`. Seule la classe peut affecter une nouvelle valeur à cette propriété. La classe garde ainsi la maîtrise des modifications, ce qu'on appelle l'encapsulation.

Prolonger une séance

Un planning doit conserver la durée initiale d'une séance et produire une seconde version, plus longue. Chaque fiche contient un titre et une durée en minutes. Crée une `data class Fiche` avec deux propriétés `val`. Écris une fonction qui ajoute quinze minutes à la durée et renvoie une nouvelle fiche. Vérifie que l'ancienne durée est encore accessible et que deux fiches construites avec les mêmes valeurs sont égales.

Corrigé

KOTLIN
```
data class Fiche(val titre: String, val minutes: Int)

fun prolonger(fiche: Fiche): Fiche {
    return fiche.copy(minutes = fiche.minutes + 15)
}

fun main() {
    val initiale = Fiche("Kotlin", 20)
    val prolongee = prolonger(initiale)
    println("${initiale.minutes} puis ${prolongee.minutes}")
    check(prolongee == Fiche("Kotlin", 35))
    check(initiale != prolongee)
}
```

La console affiche `20 puis 35`. Une `data class` fournit une méthode `copy` et une comparaison fondée sur les propriétés du constructeur principal. Dans `copy(minutes = ...)`, le titre est repris et la durée reçoit la nouvelle valeur. L'opérateur `==` compare les données des deux fiches. `!=` vérifie qu'elles sont différentes.

Essaie `check(initiale.copy() == initiale)`. Une copie sans modification contient le même titre et la même durée. Ici les propriétés ont les types `String` et `Int`. Si une propriété contenait une liste mutable, les deux fiches partageraient la référence à cette liste après `copy()`. Cette méthode réalise une copie superficielle.

Tu disposes maintenant de deux façons de modéliser une séance. Pour suivre des réservations successives, la méthode du chapitre fait évoluer un compteur. Pour garder l'historique d'un planning, chaque copie représente un état que tu peux conserver dans une liste.

Transformer un relevé de lecture

Une lectrice a noté son avancée sur trois lignes de texte. Chaque ligne doit contenir un nombre de pages strictement positif. Écris une fonction qui retourne le total des lignes valides. La liste `["12", "pause", " 8 "]` doit donner 20. Une liste vide doit donner zéro.

À toi

Écris d'abord une boucle. Pour chaque ligne, enlève les espaces avec `trim()`, tente la conversion, puis décide si la valeur doit augmenter le total. Avant d'exécuter ton code, teste-le sur papier avec `["-2", "0", "4"]`. Quelle condition élimine les deux premières lignes ?

Corrigé

KOTLIN
```
fun totalPages(lignes: List<String>): Int {
    var total = 0
    for (ligne in lignes) {
        val pages = ligne.trim().toIntOrNull()
        if (pages != null && pages > 0) total += pages
    }
    return total
}

fun main() {
    println(totalPages(listOf("12", "pause", " 8 ")))
    println(totalPages(listOf("-2", "0", "4")))
    println(totalPages(emptyList()))
}
```

La console affiche `20`, puis `4`, puis `0`. Avec `&&`, Kotlin évalue les deux conditions de gauche à droite. Il ne compare `pages` à zéro que si la première condition a établi qu'une valeur existe. Le compilateur sait alors que `pages` contient un `Int`.

Une deuxième écriture

Après `lignes`, enchaîne `.mapNotNull { it.trim().toIntOrNull() }`, puis `.filter { it > 0 }`, puis `.sum()`. `mapNotNull` applique la conversion et conserve les résultats présents. Pour le premier relevé, les valeurs intermédiaires sont `[12, 8]`, puis `[12, 8]`, puis `20`. Vérifie que tes deux fonctions donnent le même résultat.

Décrire une famille de valeurs

Une bibliothèque range ses livres par rayon. Si tu écris le rayon comme une chaîne, le compilateur accepte aussi bien `"ROMAN"` que `"ROAMN"`. Une `enum class` donne un nom à chaque choix autorisé. En déclarant une propriété de type `Rayon`, tu demandes à Kotlin de vérifier ce choix à chaque création de livre.

KOTLIN
```
enum class Rayon { ROMAN, BD }

interface Catalogable {
    fun etiquette(): String
}

data class Livre(val titre: String, val rayon: Rayon)
    : Catalogable {
    override fun etiquette() = "$titre (${rayon.name})"
}

fun afficher(article: Catalogable) {
    println(article.etiquette())
}

fun main() {
    afficher(Livre("Terre des hommes", Rayon.ROMAN))
}
```

Suivre un appel

Une interface décrit ici ce qu'un objet sait faire. Dans `Livre`, le signe `:` indique que la classe s'engage à fournir la fonction demandée par `Catalogable`. Le mot `override` désigne la réalisation de cette fonction. La propriété `name` donne le nom exact du choix dans l'énumération.

Avant de lancer le programme, suis l'objet depuis `main` jusqu'à `println`. Tu dois obtenir `Terre des hommes (ROMAN)`. Remplace ensuite `Rayon.ROMAN` par `Rayon.BD` et observe quelle partie change.

Imagine qu'un jeu de société rejoigne le catalogue. Une classe `Jeu` pourra fournir sa propre étiquette. La fonction `afficher` acceptera ce jeu dès que sa classe implémentera `Catalogable` puisque le seul service qu'elle lui demande est `etiquette()`.

Afficher l'état d'un emprunt

Un livre disponible n'a rien de particulier à afficher. Pour un livre prêté, tu veux aussi connaître le prénom de la personne qui l'a emprunté. Une énumération convient aux rayons fixes. Ici, chaque nouvel emprunt peut porter un prénom différent. Une `sealed interface` permet de décrire des variantes connues du compilateur, chacune avec ses propres données.

À toi

Écris les deux variantes puis une fonction `message` qui produit `En rayon` ou `Prêté à Nina`. Prévois aussi ce que devrait afficher un second emprunt au nom de Malik. Essaie de construire indépendamment les deux emprunts avant de lire le corrigé.

Corrigé

KOTLIN
```
sealed interface Etat
object Disponible : Etat
data class Prete(val prenom: String) : Etat

fun message(etat: Etat): String = when (etat) {
    Disponible -> "En rayon"
    is Prete -> "Prêté à ${etat.prenom}"
}

fun main() {
    println(message(Disponible))
    println(message(Prete("Nina")))
    println(message(Prete("Malik")))
}
```

Les trois lignes affichent successivement `En rayon`, `Prêté à Nina` et `Prêté à Malik`. Le mot `object` crée une seule valeur partagée. Chaque appel à `Prete(...)` construit une nouvelle valeur. Après `is Prete`, Kotlin sait dans cette branche que `etat` possède une propriété `prenom`.

Ajoute maintenant `object Perdu : Etat`. Essaie de relancer. Le compilateur signale que le `when` doit aussi couvrir cette variante pour produire une chaîne dans tous les cas. Complète-le avec `Perdu -> "Introuvable"`, puis vérifie ton ajout en appelant `message(Perdu)`.

Garder le type d'un résultat

Dans un atelier, les participants prennent la parole à tour de rôle. Dans un parcours de lecture, les chapitres se suivent. Pour trouver ce qui vient après un élément, la recherche est la même dans les deux listes. Comment écrire une seule fonction qui rende un prénom dans le premier cas et un numéro dans le second ?

KOTLIN
```
fun <T> suivant(ordre: List<T>, actuel: T): T? {
    val position = ordre.indexOf(actuel)
    if (position < 0) return null
    return ordre.getOrNull(position + 1)
}

fun main() {
    val noms = listOf("Inès", "Ali", "Lou")
    val nom: String? = suivant(noms, "Inès")
    val chapitre: Int? = suivant(listOf(2, 4, 7), 4)
    println(nom)
    println(chapitre)
    println(suivant(noms, "Lou"))
}
```

Lire la signature

`<T>` déclare un paramètre de type. À chaque appel, le compilateur déduit un type compatible avec les arguments. Le même `T` relie les éléments de la liste, l'élément recherché et le résultat. Le point d'interrogation prévoit le cas où personne ne vient ensuite. Essaie de prédire les trois lignes avant d'exécuter le code.

Tu obtiens `Ali`, `7`, puis `null`. Pour Lou, la position existe mais l'indice suivant dépasse la fin de la liste. Que se passe-t-il avec un prénom absent ? `indexOf` renvoie alors −1. Le retour anticipé évite de lui ajouter 1, ce qui désignerait par erreur la première personne ! Si le même prénom apparaît deux fois, la recherche retient sa première occurrence.

Organiser un tour de table

Après la dernière personne, la parole doit revenir à la première. Garde la règle du prénom absent et prévois aussi un groupe vide. On voudrait écrire `groupe.tourSuivant("Lou")`. Une fonction d'extension permet cette écriture. Dans sa déclaration, place le type du groupe juste avant le point. À l'intérieur, `this` représente le groupe sur lequel on a appelé la fonction.

À toi

Prends une feuille et inscris les indices de trois prénoms. Quel indice suit le dernier ? Essaie ensuite de traiter un groupe d'une seule personne. Tu peux utiliser une condition ou le reste de la division par la taille du groupe.

Corrigé

KOTLIN
```
fun <T> List<T>.tourSuivant(actuel: T): T? {
    val i = indexOf(actuel)
    if (i < 0) return null
    return this[(i + 1) % size]
}

fun main() {
    val groupe = listOf("Ali", "Lou")
    println(groupe.tourSuivant("Lou"))
    println(listOf(7).tourSuivant(7))
    println(emptyList<String>().tourSuivant("Ali"))
}
```

Tu lis `Ali`, `7`, puis `null`. Avec deux personnes, l'indice de Lou est 1. Le reste de `2 % 2` vaut 0. Dans la liste vide, la recherche échoue et la fonction sort avant toute division. Essaie également un prénom absent d'une liste remplie.

L'extension utilise les opérations publiques de la liste. Elle laisse la classe d'origine intacte et n'obtient aucun accès spécial à ses propriétés privées. Tu peux appliquer le même principe à un type précis, par exemple en écrivant `fun String.initiale() = this.firstOrNull()`.

Comprendre une erreur à l'exécution

Un atelier accueille cinq personnes. Le texte saisi peut être illisible, et même un entier comme 6 peut contredire la capacité de la salle. Avant de lancer ce code, prédis ce qu'il affichera pour chacune des trois saisies. Repère la ligne où chacune cesse d'être traitée.

KOTLIN
```
fun placesLibres(capacite: Int, inscrits: Int): Int {
    require(capacite >= 0) { "Capacité négative" }
    require(inscrits in 0..capacite) { "Effectif invalide" }
    return capacite - inscrits
}

fun main() {
    for (saisie in listOf("deux", "3", "6")) {
        val n = saisie.toIntOrNull()
        if (n == null) {
            println("Entier attendu")
        } else {
            try {
                println(placesLibres(5, n))
            } catch (e: IllegalArgumentException) {
                println(e.message)
            }
        }
    }
}
```

`deux` donne `Entier attendu`. L'échec d'une conversion est une issue prévue, représentée ici par `null`. Pour `3`, le calcul s'effectue et affiche `2`. Pour `6`, le second `require` échoue et déclenche une exception. Le calcul s'interrompt et le bloc `catch` reçoit cette exception. Sa propriété `message` contient `Effectif invalide`.

Remonter à la cause

Dans une application, `require` exprime ce que les appelants d'une fonction doivent respecter. Si l'interface attend régulièrement une saisie hors limites, tu peux aussi la vérifier avant l'appel et demander de la corriger. Ici, le `catch` ciblé permet d'observer la rupture du contrat.

Pour voir une trace d'erreur, appelle temporairement `placesLibres(5, 6)` au début de `main`. Lis d'abord le type et le message de l'exception. Repère ensuite, parmi les lignes de la trace, ta fonction et l'appel qui lui a transmis 6. Retire cet appel après l'essai.

Tester les deux côtés d'une limite

Un collègue affirme que le calcul des places disponibles est terminé parce que son essai avec trois inscrits a réussi. Il te reste à vérifier une salle vide, une salle pleine et une inscription en trop. Écris les résultats attendus sur papier pour une salle de cinq places. À quelle catégorie appartient le cas d'une salle de zéro place avec zéro inscrit ?

Corrigé

KOTLIN
```kotlin
fun placesLibres(capacite: Int, inscrits: Int): Int {
    require(capacite >= 0) { "Capacité négative" }
    require(inscrits in 0..capacite) { "Effectif invalide" }
    return capacite - inscrits
}

fun main() {
    check(placesLibres(5, 0) == 5)
    check(placesLibres(5, 3) == 2)
    check(placesLibres(5, 5) == 0)
    check(placesLibres(0, 0) == 0)
    val refuse = try {
        placesLibres(5, 6)
        false
    } catch (e: IllegalArgumentException) {
        true
    }
    check(refuse) { "Six inscrits auraient dû être refusés" }
    println("5 vérifications réussies")
}
```

`check` évalue la condition à chaque exécution. Si elle est fausse, il déclenche une `IllegalStateException`. La dernière vérification utilise `try` comme expression. Si l'appel renvoie normalement une valeur, on atteint `false`. Si l'exception attendue survient, le bloc `catch` produit `true`. Une autre exception reste visible.

Mettre le test à l'épreuve

Remplace provisoirement `capacite - inscrits` par `capacite + inscrits`. La vérification avec zéro inscrit passe encore. Celle avec trois inscrits échoue, car le résultat vaut désormais 8. Remets la soustraction et vérifie que la dernière ligne s'affiche. Pour prolonger, reproduis le test du refus pour un effectif de −1, juste de l'autre côté de la borne inférieure.

Attendre deux réponses

Deux bibliothèques répondent chacune avec leur stock. Tu veux lancer les deux demandes avant d'attendre les réponses, puis afficher leur somme. Dans cet exemple, `delay` simule une attente de 100 millisecondes. Pour exécuter les deux pages de ce chapitre en local, ajoute d'abord la bibliothèque `kotlinx.coroutines` comme indiqué en annexe.

KOTLIN
```
import kotlinx.coroutines.*

suspend fun lireStock(nombre: Int): Int {
    delay(100)
    return nombre
}

suspend fun main() {
    val total = coroutineScope {
        val nord = async { lireStock(4) }
        val sud = async { lireStock(3) }
        nord.await() + sud.await()
    }
    println("$total livres")
}
```

La ligne `import kotlinx.coroutines.*` rend accessibles les noms du package, dont `async` et `delay`. Le mot `suspend` autorise la suspension d'une fonction appelée depuis une coroutine ou une autre fonction suspendable. Pendant `delay`, la coroutine libère le fil d'exécution, qu'on appelle aussi un thread. À la reprise, elle poursuit juste après l'attente. Avec deux appels directs à `lireStock`, le second commencerait après le retour du premier.

Suivre les deux demandes

`async` démarre ici une coroutine enfant et rend immédiatement un objet représentant son résultat à venir. Chaque `await()` récupère l'entier dès qu'il est prêt. Les deux demandes ont donc été lancées quand on atteint le premier `await()`. Le programme affiche `7 livres`, quelle que soit la première réponse reçue.

Le bloc `coroutineScope` attend la fin de ses enfants avant de rendre son résultat. L'annulation de sa portée se transmet à ses enfants. Les attentes peuvent se chevaucher, même sur un seul thread. L'exécution simultanée sur plusieurs threads dépend de l'environnement et de la façon dont le travail y est réparti.

Fermer un écran pendant l'attente

Tu simules un écran qui attend une réponse. Si l'utilisateur le ferme, la lecture doit s'arrêter. Si l'écran reste ouvert, le programme doit afficher `Disponible`. Écris sur papier les lignes attendues dans les deux cas. Comment vérifier la fermeture sans parier sur la vitesse de l'ordinateur ?

Corrigé

KOTLIN
```kotlin
import kotlinx.coroutines.*

suspend fun essai(fermer: Boolean) = coroutineScope {
    val reponse = CompletableDeferred<String>()
    val lecture = launch { println(reponse.await()) }
    if (fermer) lecture.cancelAndJoin()
    reponse.complete("Disponible")
    lecture.join()
    println("Annulation : ${lecture.isCancelled}")
}

suspend fun main() {
    essai(true)
    essai(false)
}
```

CONSOLE
```
Annulation : true
Disponible
Annulation : false
```

`CompletableDeferred<String>()` crée un résultat en attente que tu peux fournir toi-même. `launch` démarre une coroutine et renvoie un objet qui sert à suivre sa tâche. Ici, la tâche attend le texte avec `await()`. Le programme choisit le moment exact où il transmet ce texte avec `complete`.

Pour le premier essai, `cancelAndJoin()` demande l'annulation et attend la fin de la tâche avant que le texte soit disponible. Pour le second, `join()` attend que l'affichage soit terminé. Ainsi, même une réponse instantanée respecte l'ordre annoncé.

Une coroutine en attente dans `await` ou `delay` réagit à l'annulation. Un long calcul sans point de suspension doit coopérer lui aussi, par exemple en appelant régulièrement `ensureActive()` dans le corps d'un `launch`. Rejoue enfin l'exercice avec une chaîne vide. L'écran ouvert imprime une ligne vide, qui reste une réponse reçue.

Construire un carnet de lecture

Tu prêtes ton ordinateur à une amie. Elle veut saisir les titres qu'elle compte lire, retrouver la liste numérotée et cocher les livres terminés. Elle préfère taper quelques commandes dans la console. À toi de construire ce carnet à partir d'un fichier Kotlin vide.

Le contrat du programme

- `ajouter Le Horla` ajoute un livre à lire. Un titre peut contenir des espaces. Un titre vide doit être refusé.
- `liste` affiche les titres dans l'ordre d'ajout, avec un numéro qui commence à 1 et la mention « à lire » ou « lu ». Sur une liste vide, affiche un message.
- `lu 2` marque le deuxième livre comme lu. Si le numéro manque, contient du texte ou dépasse la liste, explique l'erreur. Si le livre est déjà lu, dis-le.
- `quitter` termine la session. Le carnet vit en mémoire et repart vide au lancement suivant. Les commandes s'écrivent en minuscules.

Essaie avant d'ouvrir le corrigé

Commence par faire fonctionner `liste` avec un titre ajouté directement dans le code. Ajoute ensuite une deuxième fiche afin de distinguer le numéro affiché et l'indice utilisé par Kotlin. Branche la lecture du clavier en dernier. Tu pourras ainsi régler un problème à la fois.

Voici une séance possible. Les lignes qui commencent par `>` représentent ce que la lectrice tape. Le signe lui-même ne fait pas partie de la commande.

CONSOLE
```
> ajouter Le Horla
Ajouté : Le Horla
> lu 1
Marqué comme lu.
> liste
1. Le Horla (lu)
```

Le corrigé tient dans le fichier `Projet/Carnet.kt` de l'archive. Les quatre pages suivantes en donnent les morceaux, à placer dans cet ordre dans le même fichier.

Représenter les livres

Un livre a deux propriétés. Son titre est fixé à l'ajout. Son état vaut `false` au départ, puis peut passer à `true`. Une `data class` permet de remplacer une fiche par sa copie mise à jour. La liste reste la même pendant toute la session.

KOTLIN / FRAGMENT DE CARNET.KT

```
data class Livre(val titre: String, val lu: Boolean = false)

fun ajouter(livres: MutableList<Livre>, titre: String): String {
    val propre = titre.trim()
    if (propre.isEmpty()) return "Titre manquant."
    livres.add(Livre(propre))
    return "Ajouté : $propre"
}

fun marquerLu(livres: MutableList<Livre>, numero: Int?): String {
    if (numero == null || numero !in 1..livres.size) {
        return "Numéro inconnu."
    }
    val indice = numero - 1
    val livre = livres[indice]
    if (livre.lu) return "Déjà lu."
    livres[indice] = livre.copy(lu = true)
    return "Marqué comme lu."
}
```

Suis un changement à la main

Dessine deux cases, l'une pour « Le Horla », l'autre pour « Matilda ». Écris les indices 0 et 1 sous les cases. L'amie choisit le numéro 2. La condition autorise `2` car il appartient à `1..2`. La soustraction donne l'indice 1. Seule la deuxième case reçoit une nouvelle fiche.

Si elle entre `0`, la fonction renvoie « Numéro inconnu. » avant l'accès par indice. Si elle envoie `lu 2` une seconde fois, la ligne `if (livre.lu)` interrompt la fonction avant la copie. Cette organisation permet de relire toutes les raisons de sortir avant la modification.

Interpréter une commande

La personne qui tape `ajouter Le Horla` pense en mots. La fonction `ajouter` a besoin du seul titre. Une petite fonction fait le lien entre les deux. Avant de lire la solution, indique où tu couperais la commande pour conserver un titre de plusieurs mots.

KOTLIN / FRAGMENT DE CARNET.KT

```
fun afficher(livres: List<Livre>): String {
    if (livres.isEmpty()) return "Carnet vide."
    return livres.mapIndexed { indice, livre ->
        val etat = if (livre.lu) "lu" else "à lire"
        "${indice + 1}. ${livre.titre} ($etat)"
    }.joinToString("\n")
}

fun repondre(ligne: String, livres: MutableList<Livre>): String {
    val propre = ligne.trim()
    val action = propre.substringBefore(' ')
    val suite = propre.substringAfter(' ', "").trim()
    return when (action) {
        "ajouter" -> ajouter(livres, suite)
        "lu" -> marquerLu(livres, suite.toIntOrNull())
        "liste" -> if (suite.isEmpty()) afficher(livres)
                   else "Tape liste sans argument."
        else -> "Commande inconnue."
    }
}
```

`mapIndexed` passe deux arguments à la lambda, l'indice et le livre. La dernière expression fabrique une ligne d'affichage. `joinToString("\n")` assemble les lignes en insérant un saut de ligne entre elles.

Pour `ajouter Le Horla`, `substringBefore(' ')` donne `ajouter`. `substringAfter(' ', "")` garde `Le Horla` en entier. Le deuxième argument `""` fournit la suite quand la ligne ne contient aucun espace. Ainsi, `ajouter` seul envoie une chaîne vide à la validation déjà écrite.

Essaie cette commande sur papier

`lu douze` donne l'action `lu` et la suite `douze`. La conversion renvoie `null`. Retrouve dans la page précédente le message que verra la lectrice.

Vérifier le carnet sans clavier

Avant de demander à une personne d'essayer le carnet, reproduis une petite séance directement dans le code. La fonction suivante traite les mêmes chaînes que la future console. Écris tes résultats attendus sur papier, puis compare-les aux valeurs situées à droite des `==`.

KOTLIN / FRAGMENT DE CARNET.KT

```
fun verifierCarnet() {
    val livres = mutableListOf<Livre>()
    check(repondre("liste", livres) == "Carnet vide.")
    check(repondre("ajouter   ", livres) == "Titre manquant.")
    check(repondre("ajouter Le Horla", livres) ==
          "Ajouté : Le Horla")
    check(repondre("lu 0", livres) == "Numéro inconnu.")
    check(repondre("lu deux", livres) == "Numéro inconnu.")
    check(repondre("lu 2", livres) == "Numéro inconnu.")
    check(repondre("lu 1", livres) == "Marqué comme lu.")
    check(repondre("lu 1", livres) == "Déjà lu.")
    check(repondre("liste", livres) == "1. Le Horla (lu)")
}
```

Pourquoi ces neuf vérifications ?

La première fixe le comportement d'un carnet neuf. La deuxième garantit qu'un titre vide est refusé. La troisième ajoute le seul livre attendu. Les trois suivantes envoient un indice trop petit, du texte et un indice trop grand. Les deux appels avec `1` doivent suivre des chemins différents puisque le premier change l'état du livre. La dernière vérification confirme que cet état se retrouve dans la liste.

Pour détecter une régression, remplace temporairement `livre.copy(lu = true)` par `livre.copy(lu = false)`. Une fois `main` ajouté à la page suivante, le premier `lu 1` annoncera un succès, mais le contrôle du deuxième `lu 1` échouera. Rétablis ensuite `true`.

Dialoguer dans la console

Il manque le point d'entrée. Dans un projet console local, colle ces lignes à la suite du fichier. Exécute `main`. Le programme lance ses vérifications avec un carnet réservé au test, puis crée un carnet vide pour ta séance.

KOTLIN / FRAGMENT DE CARNET.KT

```kotlin
fun main() {
    verifierCarnet()
    val livres = mutableListOf<Livre>()
    println("Commandes : ajouter TITRE, lu NUMÉRO, liste, quitter")
    while (true) {
        val ligne = readlnOrNull() ?: break
        if (ligne.trim() == "quitter") break
        println(repondre(ligne, livres))
    }
}
```

`while (true)` recommence après chaque réponse. `break` quitte la boucle, soit parce que la saisie est terminée, soit parce que la lectrice a demandé à quitter. L'appel à `repondre` reçoit toujours la liste créée avant la boucle. Déplace mentalement cette création à l'intérieur. Que se passerait-il entre `ajouter` et `liste` ?

Essaie deux livres

Tape, dans cet ordre, `ajouter Le Horla`, `ajouter Matilda`, `lu 2`, puis `liste`. Tu dois lire les deux lignes suivantes. Lance ensuite `lu 1` et vérifie que les deux livres sont lus.

CONSOLE

```
1. Le Horla (à lire)
2. Matilda (lu)
```

Défi de fin de cours

Ajoute une commande `reste` qui affiche le nombre de livres à lire. Elle doit donner `0` sur une liste vide, `2` après les deux ajouts et `1` après `lu 2`. Commence par écrire ces trois vérifications. Dans le `when` de `repondre`, une correction possible est `"reste" -> livres.count { !it.lu }.toString()`. `count` compte les éléments qui satisfont la condition. Si tu peux expliquer où écrire la commande et pourquoi son résultat change, tu sais déjà faire évoluer ce programme toi-même.

Préparer le chapitre sur les coroutines

`delay` et `async` font partie de la bibliothèque `kotlinx.coroutines`. Le compilateur connaît le mot `suspend`, mais il a besoin de cette bibliothèque pour trouver ces fonctions. Voici comment préparer un projet local distinct pour les deux programmes du chapitre 14.

Créer un projet Kotlin/JVM avec Gradle

Dans IntelliJ IDEA, crée un nouveau projet `Kotlin`. Cette fois, choisis `Gradle` comme système de construction et `Kotlin` pour le langage des scripts Gradle. Choisis un JDK, par exemple 21, active le code d'exemple et ouvre le projet. Gradle est l'outil qui récupère les bibliothèques et organise la compilation.

Ouvre le fichier `build.gradle.kts` du module qui contient `src/main/kotlin`. Dans le bloc `repositories`, vérifie la présence de `mavenCentral()`. Dans le bloc `dependencies`, ajoute la ligne `implementation` ci-dessous. Si ces blocs existent déjà, complète-les. Conserve les lignes `plugins` et le reste de la configuration créée par l'IDE.

GRADLE / BUILD.GRADLE.KTS

```
repositories {
    mavenCentral()
}

dependencies {
    implementation("org.jetbrains.kotlinx:kotlinx-coroutines-core:1.8.0")
}
```

Vérifier l'installation

Recharge les modifications Gradle dans l'IDE. Le premier téléchargement demande une connexion. Dans `Main.kt`, colle le premier programme du chapitre 14, puis exécute sa fonction `main`. Le résultat attendu est `7 livres`.

Si `delay` apparaît en rouge, vérifie d'abord que la synchronisation Gradle s'est terminée et que l'import du chapitre figure tout en haut de `Main.kt`. La version `1.8.0` est celle utilisée pour exécuter les exemples de ce cours. Documentation officielle des coroutines

Lire une lambda et son type

Dans un projet Kotlin, tu peux rencontrer `(String) -> Unit`. La partie à gauche de la flèche donne le type de l'argument, celle de droite le type du résultat. `Unit` indique que l'appel se termine sans produire d'information à réutiliser. Une fonction qui affiche un message peut convenir à ce type.

KOTLIN
```kotlin
fun presenter(nom: String, action: (String) -> Unit) {
    action(nom.trim())
}

fun main() {
    presenter("  Lina ") { nom ->
        println("Bonjour, $nom")
    }
    val entree: String? = " 42 "
    val double = entree?.let { it.trim().toIntOrNull() }
        ?.let { it * 2 }
    println(double ?: 0)
}
```

À quoi se rapporte it ?

Le premier appel transmet une fonction entre accolades après les parenthèses. Cette écriture est possible quand le dernier argument est une lambda. La fonction `presenter` retire les espaces avant d'appeler `action`. Tu lis donc « Bonjour, Lina ».

`let` passe la valeur qui le précède à sa lambda et renvoie le résultat de cette lambda. Avec `?.let`, le bloc s'exécute uniquement si cette valeur est présente. Dans le premier bloc, `it` désigne la chaîne ; dans le deuxième, il désigne l'entier 42. La dernière ligne affiche 84.

Remplace `" 42 "` par `"aucun"`. La conversion produit `null`, le second bloc est sauté, puis `?:` choisit zéro. Tu peux renommer chaque `it`, par exemple `{ texte -> texte.trim().toIntOrNull() }`, si les types sont difficiles à suivre. Fonctions de portée

Étendre une classe

Tu as déjà utilisé une interface pour décrire ce qu'un objet sait faire. Une classe de base peut aussi fournir des propriétés et du code à ses classes dérivées. Imagine que tous les documents d'un catalogue portent un titre et puissent fabriquer une étiquette. Seul le type de document change.

KOTLIN
```
abstract class Document(val titre: String) {
    abstract fun categorie(): String
    fun etiquette() = "${categorie()} : $titre"
}

class Roman(titre: String) : Document(titre) {
    override fun categorie() = "Roman"
}

fun main() {
    val doc: Document = Roman("Matilda")
    println(doc.etiquette())
}
```

`abstract` indique ici que `Document` doit être complétée par une sous-classe. `Document(titre)` initialise la partie commune de l'objet. `Roman` fournit sa réalisation de `categorie()` avec `override`. Quand `etiquette()` demande la catégorie, Kotlin utilise celle de l'objet réellement créé. La console affiche « Roman : Matilda ».

À toi

Ajoute `class BD(titre: String) : Document(titre)` et donne-lui un corps qui contient `override fun categorie() = "BD"`. Prédis puis vérifie la sortie de `println(BD("Astérix").etiquette())`. Elle doit être « BD : Astérix ».

Par défaut, une classe ordinaire et ses méthodes sont fermées à l'héritage et à la redéfinition. `open` autorise l'extension d'une classe ou la redéfinition d'une méthode déjà définie. Une méthode `abstract`, comme `categorie`, demande à la sous-classe concrète d'en fournir le corps. Héritage en Kotlin

Retrouver Kotlin dans ton projet Android

Dans ton projet `jeia`, le fichier `app/src/main/java/com/jeia/MainActivity.kt` contient cette fonction. Elle peut déjà te servir d'exercice de lecture. La page se lit avec le fichier ouvert à côté du cours.

ANDROID / EXTRAIT DE MAINACTIVITY.KT

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
        text = "Hello $name!",
        modifier = modifier,
    )
}
```

Reconnais ce que tu as appris

- `fun Greeting` déclare une fonction. Le premier paramètre reçoit un texte. Le deuxième reçoit un `Modifier` et possède une valeur par défaut.
- Sur la ligne `modifier = modifier`, le nom de gauche désigne le paramètre nommé de `Text`. Le nom de droite désigne la valeur reçue par `Greeting`.
- Dans la chaîne `"Hello $name!"`, Kotlin insère le contenu de `name`, comme dans ton premier carnet. L'appel `Greeting("Android")` fournit donc le texte « Hello Android! » à `Text`.

Ce qui vient d'Android

`@Composable` est une annotation utilisée par le compilateur Compose. `Text` décrit un texte à afficher dans l'interface de l'application. En haut du fichier, `package com.jeia` indique à quel espace de noms appartient ton code. Les lignes `import` permettent d'employer des noms tels que `Text` et `Modifier` dans le fichier.

Pour t'exercer dans la console, écris `fun salutation(nom: String) = "Bonjour, $nom !"` puis appelle cette fonction dans un `println` de ton `main`. Essaie « Louise » puis une chaîne vide. La partie qui fabrique le message peut ainsi se tester indépendamment de l'écran qui l'affiche.
Comprendre le fonctionnement de Compose

Que sais-tu refaire sans le corrigé ?

Reprends une feuille ou un fichier vide. Pour chaque question, donne une réponse et un exemple. Si tu hésites, essaie sur deux valeurs avant de relire le chapitre indiqué.

- **Chapitres 1 à 5.** Prédis `5 / 2`, puis `5 / 2.0`. Écris une boucle qui affiche les entiers de 1 à 5 inclus. Comment l'arrêterais-tu à 4 ?
- **Chapitres 6 et 7.** Écris `fun minutes(texte: String?): Int?` qui accepte les entiers positifs ou nuls et refuse le reste. Essaie `null`, `"0"`, `"-1"` et `" 12 "`.
- **Chapitres 8 à 10.** Avec trois livres, dont deux lus, affiche uniquement les titres des livres lus. Que doit produire une liste vide ?
- **Chapitres 11 à 14.** Explique comment un `when` sur un type `sealed` t'aide lors de l'ajout d'une variante. Dans le programme des deux stocks, indique à quel moment les deux demandes ont été lancées.

Repères pour te corriger

La division entière donne 2 et la division décimale 2.5. La plage `1..5` inclut 5 et `1 until 5` s'arrête à 4. Pour `minutes`, retire les espaces, convertis avec `toIntOrNull()` et vérifie que l'entier est supérieur ou égal à zéro. Les quatre résultats doivent être `null`, `0`, `null` et `12`.

Avec la classe `Livre` du projet final, `livres.filter { it.lu }.map { it.titre }` produit les titres attendus et renvoie une liste vide quand aucun livre ne correspond. Un `when` exhaustif doit couvrir la nouvelle variante. Les deux demandes de stock sont lancées avant le premier `await`.

Retrouver la règle dans la documentation

Les ressources ci-dessous reprennent les sujets du cours, avec d'autres exemples à exécuter.

- Parcours Kotlin officiel · Syntaxe de base · Fonctions
- Valeurs nulles · Collections · Classes
- Types scellés · Génériques · Extensions · Exceptions