Fondamentaux d'Android

8. Conversion en Kotlin

Hatem Aziza - November 2024

Developers

Introduction

Dans cet atelier, vous apprendrez à convertir votre code de Java en Kotlin. Vous apprendrez également ce que sont les conventions du langage Kotlin et comment vous assurer que le code que vous écrivez les suit.

Cet atelier convient à tout développeur qui utilise Java et qui envisage de migrer son projet vers Kotlin. Nous commencerons par quelques classes Java que vous convertirez en Kotlin en utilisant l'IDE. Ensuite, nous allons jeter un coup d'œil au code converti et voir comment nous pouvons l'améliorer en le rendant plus idiomatique et en évitant les pièges courants.

Ce que vous apprendrez

- Gestion de la nullité
- Mise en œuvre de singletons
- Classes de données
- Manipulation des cordes
- Opérateur Elvis
- Déstructuration
- Propriétés et propriétés de soutien
- Arguments par défaut et paramètres nommés
- Travailler avec les collections
- Fonctions d'extension
- Fonctions et paramètres de haut niveau
- let, apply, with, and run keywords

Mise en place

Créer un nouveau projet

Créez un nouveau projet avec le modèle ***Pas d'activité***. Choisissez **Kotlin** comme langue du projet.

Le code

Nous allons créer un objet modèle ***User*** et une classe singleton ***Repository*** qui fonctionne avec les objets User et expose des listes d'utilisateurs et des noms d'utilisateurs formatés.

Créez un nouveau fichier appelé **User.java** sous *app/java/<yourpackagename>* et collez le code suivant :

```
public class User {

    @Nullable
    private String firstName;
    @Nullable
    private String lastName;

    public User(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getFirstName() {
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

Vous remarquerez que votre IDE vous indique que **@Nullable** n'est pas défini. Importez donc **androidx.annotation.Nullable** si vous utilisez Android Studio.

Créez un nouveau fichier appelé **Repository.java** et collez le code suivant :

```
import java.util.ArrayList;
import java.util.List;

public class Repository {

   private static Repository INSTANCE = null;

   private List<User> users = null;

   public static Repository getInstance() {
      if (INSTANCE == null) {
         synchronized (Repository.class) {
            if (INSTANCE == null) {
               INSTANCE = new Repository();
            }
         }
      }
      return INSTANCE;
   }

   // garder le constructeur privé pour imposer l'utilisation de getInstance
   private Repository() {

      User user1 = new User("Zeinab", "");
      User user2 = new User("Yassine", null);
      User user3 = new User("Youssef", "Tounsi");

      users = new ArrayList();
      users.add(user1);
      users.add(user2);
      users.add(user3);
   }

   public List<User> getUsers() {
      return users;
   }

   public List<String> getFormattedUserNames() {
      List<String> userNames = new ArrayList<>(users.size());
```

```
        for (User user : users) {
            String name;

            if (user.getLastName() != null) {
                if (user.getFirstName() != null) {
                    name = user.getFirstName() + " " + user.getLastName();
                } else {
                    name = user.getLastName();
                }
            } else if (user.getFirstName() != null) {
                name = user.getFirstName();
            } else {
                name = "Unknown";
            }
            userNames.add(name);
        }
        return userNames;
    }
}
```

Déclaration de nullabilité, de val, de var et des classes de données

Notre IDE peut faire un assez bon travail de conversion automatique du code Java en code Kotlin, mais parfois il a besoin d'un peu d'aide. Laissons notre IDE faire une première passe à la conversion. Ensuite, nous passerons en revue le code résultant pour comprendre comment et pourquoi il a été converti de cette façon.

Accédez au fichier **User.java** et convertissez-le en ***Kotlin*** : *Barre de menus -> Code -> Convertir le fichier Java en fichier Kotlin*.

Vous devriez voir le code Kotlin suivant :

```
class User(var firstName: String?, var lastName: String?)
```

Notez que **User.java** a été renommé en **User.kt**. Les fichiers Kotlin ont l'extension *.kt*.

Conseil: Si vous collez du code Java dans un fichier Kotlin, l'IDE convertira automatiquement le code collé en Kotlin.

Dans notre classe Java **User**, nous avions deux propriétés : *firstName* et *lastName*. Chacune avait une méthode *getter* et *setter*, ce qui rendait sa valeur modifiable. Le mot-clé de Kotlin pour les variables mutables est ***var***, donc le convertisseur utilise *var* pour chacune de ces

propriétés. Si nos propriétés Java n'avaient que des getters, elles seraient en lecture seule et auraient été déclarées comme variables ***val***. val est similaire au mot-clé final en Java.

L'une des principales différences entre Kotlin et Java est que Kotlin spécifie explicitement si une variable peut accepter une valeur nulle. Il le fait en ajoutant un **?** à la déclaration de type.

Étant donné que nous avons marqué *firstName* et *lastName* comme nullables, le convertisseur automatique a automatiquement marqué les propriétés comme nullables avec **String?**. Si vous annotez vos membres Java comme non nuls (en utilisant org.jetbrains.annotations.NotNull ou androidx.annotation.NonNull), le convertisseur le reconnaîtra et rendra également les champs non nuls dans Kotlin.

Conseil: en Kotlin, nous recommandons d'utiliser autant que possible des objets en lecture seule et immuables (c'est-à-dire d'utiliser val au lieu de var) et d'éviter les types nullables.

La conversion de base est déjà effectuée. Mais nous pouvons écrire cela d'une manière plus idiomatique. Voyons comment.

Classe de données

Notre classe d'utilisateurs contient uniquement des données. Kotlin a un mot-clé pour les classes avec ce rôle: **data**. En marquant cette classe en tant que classe de données, le compilateur créera automatiquement des Getters et des Setters pour nous. Il dérivera également les fonctions **equals ()**, **hashcode ()** et **toString ()**.

Ajoutons le mot-clé de données à notre classe d'utilisateurs:

```
data class User(var firstName: String?, var lastName: String?)
```

Kotlin, comme Java, peut avoir un constructeur primaire et un ou plusieurs constructeurs secondaires. Celui de l'exemple ci-dessus est le constructeur principal de la classe User. Si vous convertissez une classe Java qui a plusieurs constructeurs, le convertisseur créera automatiquement plusieurs constructeurs dans Kotlin également. Ils sont définis à l'aide du mot-clé constructor.

Si nous voulons créer une instance de cette classe, nous pouvons le faire comme ceci :

```
val user1 = User("Youssef", "Tounsi")
```

Égalité

Kotlin a deux types d'égalité :

- L'égalité structurelle utilise l'opérateur == et les appels **equals ()** pour déterminer si deux instances sont égales.
- L'égalité référentielle utilise l'opérateur === et vérifie si deux références pointent vers le même objet.

Les propriétés définies dans le constructeur principal de la classe de données seront utilisées pour les contrôles d'égalité structurelle.

```
val user1 = User("Youssef", "Tounsi")
val user2 = User("Youssef", "Tounsi")
val structurallyEqual = user1 == user2 // true
val referentiallyEqual = user1 === user2 // false
```

Arguments par défaut, arguments nommés

Dans Kotlin, nous pouvons attribuer des valeurs par défaut aux arguments dans les appels de fonction. La valeur par défaut est utilisée lorsque l'argument est omis. Dans Kotlin, les constructeurs sont également des fonctions, nous pouvons donc utiliser des arguments par défaut pour spécifier que la valeur par défaut de LastName est nul. Pour ce faire, nous attribuons simplement Null à LastName.

```
data class User(var firstName: String?, var lastName: String? = null)

// usage
val user1 = User("Zeinab") // same as User("Zeinab", null)
val user2 = User("Youssef", “Tounsi")
```

Kotlin vous permet d'étiqueter vos arguments lorsque vos fonctions sont appelées :

```
val user1 = User(firstName = "Youssef", lastName = "Tounsi")
```

En tant que cas d'utilisation différent, disons que le *FirstName* a **NULL** comme sa valeur par défaut. Dans ce cas, comme le paramètre par défaut précéderait un paramètre sans valeur par défaut, vous devez appeler la fonction avec des arguments nommés:

```
data class User(var firstName: String? = null, var lastName: String?)

// usage
val user1 = User(lastName = "Tounsi") // same as User(null, "Tounsi")
val user2 = User("Youssef", “Tounsi")
```

Initialisation d'objet, objet compagnon et singletons

Avant de continuer cet atelier, assurez-vous que votre classe utilisateur est une classe de données. Maintenant, convertissons la classe de référentiel en Kotlin. Le résultat de la conversion automatique devrait ressembler à ceci:

```
import java.util.*

class Repository private constructor() {
    private var users: MutableList<User?>? = null
    fun getUsers(): List<User?>? {
        return users
    }

    val formattedUserNames: List<String?>
        get() {
            val userNames: MutableList<String?> =
                ArrayList(users!!.size)
            for (user in users) {
                var name: String
                name = if (user!!.lastName != null) {
                    if (user!!.firstName != null) {
                        user!!.firstName + " " + user!!.lastName
                    } else {
                        user!!.lastName
                    }
                } else if (user!!.firstName != null) {
                    user!!.firstName
                } else {
                    "Unknown"
                }
                userNames.add(name)
            }
            return userNames
        }

    companion object {
        private var INSTANCE: Repository? = null
        val instance: Repository?
            get() {
                if (INSTANCE == null) {
                    synchronized(Repository::class.java) {
                        if (INSTANCE == null) {
                            INSTANCE =
```

```
                    Repository()
                }
            }
        }
        return INSTANCE
    }
}

// keeping the constructor private to enforce the usage of getInstance
init {
    val user1 = User("Zeinab", "")
    val user2 = User("Yassine", null)
    val user3 = User("Youssef", "Tounsi")
    users = ArrayList<Any?>()
    users.add(user1)
    users.add(user2)
    users.add(user3)
}
}
```

Voyons ce que le convertisseur automatique a fait :

- La liste des utilisateurs est nullable car l'objet n'a pas été instancié au moment de la déclaration
- Les fonctions dans Kotlin comme **getUsers ()** sont déclarées avec le modificateur **fun**
- La méthode **getFormattedUsernames()** est désormais une propriété appelée **FormattedUsnames**
- L'itération sur la liste des utilisateurs (qui faisait initialement partie de **getFormattedUsernames ()**) a une syntaxe différente de celle de Java
- Le champ **static** fait désormais partie d'un bloc d'objet compagnon
- Un bloc **init** a été ajouté

Avant d'aller plus loin, nettoyons un peu le code. Si nous regardons dans le constructeur, nous remarquons que le convertisseur a fait de la liste des utilisateurs une liste mutable qui contient des objets nullables. Bien que la liste puisse en effet être nul, supposons qu'elle ne peut pas détenir des utilisateurs nuls. Alors faisons ce qui suit:

Supprimer le **?** Dans **User?** dans la déclaration de Type Users
Supprimer le **?** dans **User?** pour le type de retour de **getUsers()** pour qu'il renvoie **List<User>?**

Bloc d’init

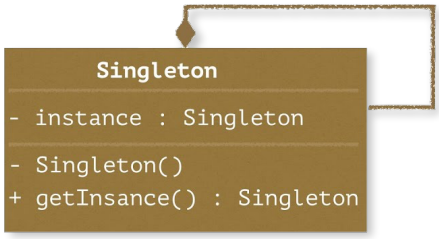
Dans Kotlin, le constructeur principal ne peut contenir aucun code, de sorte que le code d'initialisation est placé dans des blocs init. La fonctionnalité est la même.

```
class Repository private constructor() {
   ...
   init {
      val user1 = User("Zeinab", "")
      val user2 = User("Yassine", null)
      val user3 = User("Youssef", "Tounsi")
      users = ArrayList<Any?>()
      users.add(user1)
      users.add(user2)
      users.add(user3)
   }
}
```

Une grande partie du code **init** gère les propriétés d'initialisation. Cela peut également être fait dans la déclaration de la propriété. Par exemple, dans la version Kotlin de notre classe de référentiel, nous voyons que la propriété des utilisateurs a été initialisée dans la déclaration.

```
private var users: MutableList<User>? = null
```

Manipulation de singletons



Parce que nous n'avons besoin qu'une seule instance de la classe de référentiel, nous avons utilisé le modèle Singleton en Java. Avec Kotlin, vous pouvez appliquer ce modèle au niveau du compilateur en remplaçant le mot-clé de classe par objet.

Supprimez le constructeur privé et remplacez la définition de classe par **object Repository**. Supprimez également l'objet compagnon.

```
object Repository {

   private var users: MutableList<User>? = null
   fun getUsers(): List<User>? {
     return users
```

```
    }

    val formattedUserNames: List<String>
        get() {
            val userNames: MutableList<String> =
                ArrayList(users!!.size)
        for (user in users) {
            var name: String
            name = if (user!!.lastName != null) {
                if (user!!.firstName != null) {
                    user!!.firstName + " " + user!!.lastName
                } else {
                    user!!.lastName
                }
            } else if (user!!.firstName != null) {
                user!!.firstName
            } else {
                "Unknown"
            }
            userNames.add(name)
        }
        return userNames
    }

    init {
        val user1 = User("Zeinab", "")
        val user2 = User("Yassine", null)
        val user3 = User("Youssef", "Tounsi")
        users = ArrayList<Any?>()
        users.add(user1)
        users.add(user2)
        users.add(user3)
    }
}
```

Lorsque nous utilisons la classe **object**, nous appelons simplement des fonctions et des propriétés directement sur l'objet, comme ceci :

```
val formattedUserNames = Repository.formattedUserNames
```

Notez que si une propriété n'a pas de modificateur de visibilité dessus, elle est publique par défaut, comme dans le cas de la propriété FormattedUserNames dans l'objet de référentiel.

Gestion de la Nullabilité dans une Classe Repository

Lors de la conversion de la classe Repository en Kotlin, le convertisseur automatique a rendu la liste users nullable, car elle n'était pas initialisée lors de sa déclaration. Cela signifie qu'il faut utiliser l'opérateur **!!** pour chaque utilisation de users (par exemple, **users!!**). Cependant, l'utilisation excessive de **!!** peut entraîner des exceptions à l'exécution si la variable est effectivement nulle.

Alternatives pour Gérer la Nullabilité

Il est préférable d'éviter les risques de nullabilité en utilisant l'une des méthodes suivantes:

- Effectuer une vérification de nullité : **if (users != null) { ... }**
- Utiliser l'opérateur **elvis ?:** : cet opérateur renvoie une valeur alternative si la variable est nulle.
- etc.

Dans notre cas, nous savons que users n'a pas besoin d'être nullable, car elle est initialisée dès la création de l'objet dans le bloc init. Nous allons donc l'initialiser directement lors de la déclaration. En utilisant la fonction mutableListOf(), nous pouvons créer une liste mutable d'objets User :

```
private val users = mutableListOf<User>()
```

Remplacement de var par val

Nous utilisons maintenant **val** au lieu de **var**, ce qui rend la référence à users en lecture seule. Cela signifie que la liste elle-même est modifiable (on peut ajouter ou supprimer des éléments), mais la référence ne peut pas être changée pour pointer vers une autre liste.

Initialisation dans le Bloc init

Comme **users** est maintenant initialisé directement lors de sa déclaration, nous n'avons plus besoin d'initialiser la liste dans le bloc init. Voici à quoi ressemble le bloc init mis à jour :

```
init {
    val user1 = User("Zeinab", "")
    val user2 = User("Yassine", null)
    val user3 = User("Youssef", "Tounsi")

    users.add(user1)
    users.add(user2)
    users.add(user3)
}
```

Formatage des Noms des Utilisateurs

Nous allons créer une liste de noms d'utilisateurs, où chaque nom est construit en combinant le prénom et le nom de famille (s'ils sont disponibles) ou en affichant "Unknown" si les deux sont nuls.

```
val userNames = ArrayList<String>(users.size)

for (user in users) {
    val name = if (user.lastName != null) {
        if (user.firstName != null) {
            "${user.firstName} ${user.lastName}"
        } else {
            user.lastName
        }
    } else user.firstName ?: "Unknown"

    userNames.add(name)
}
```

Utilisation de la Déstructuration

Kotlin permet la déstructuration, ce qui nous permet d'accéder directement aux attributs d'un objet dans une boucle. Nous pouvons donc simplifier le code de la boucle :

```
for ((firstName, lastName) in users) {
    val name = if (lastName != null) {
        if (firstName != null) {
            "$firstName $lastName"
        } else {
            lastName
        }
    } else firstName ?: "Unknown"

    userNames.add(name)
}
```

Utilisation de l'Opérateur Elvis ?:

Nous pouvons simplifier davantage le code en utilisant l'opérateur elvis ?:. Cet opérateur retourne la valeur à gauche s'il n'est pas nul, sinon il retourne la valeur à droite.

```
val name = if (lastName != null) {
    "$firstName $lastName"
} else firstName ?: "Unknown"
```

En utilisant ces techniques, nous avons rendu notre code Kotlin plus idiomatique, plus lisible et sécurisé contre les exceptions de nullité.

Utilisation des Templates de Chaînes de Caractères

Kotlin simplifie la manipulation des chaînes de caractères grâce aux templates de chaînes. Ces templates permettent d'insérer des variables directement dans les chaînes en utilisant le symbole $ devant le nom de la variable. Par exemple, pour afficher firstName dans une chaîne, on peut écrire : "$firstName". Pour inclure une expression, il suffit de l'encadrer entre { } après le symbole $, comme dans ${user.firstName}.

Remplacement de la Concatenation de Chaînes

Dans le code actuel, les noms complets des utilisateurs sont formés en utilisant la concaténation :

```
if (firstName != null) {
   firstName + " " + lastName
}
```

En utilisant les templates de chaînes, cette concaténation peut être simplifiée :

```
if (firstName != null) {
   "$firstName $lastName"
}
```

Kotlin identifiera souvent ces possibilités d'optimisation et vous proposera de les appliquer via des avertissements dans l'IDE.

Simplification de la Création de la Liste des Noms d'Utilisateurs

Pour simplifier davantage, supprimons la déclaration explicite du type **String** pour **name**, car Kotlin peut inférer le type. De plus, puisque notre interface utilisateur affiche "Unknown" lorsqu'un nom est absent, nous allons nous assurer que la liste de noms est de type **List<String>** (sans nullable). Voici le code final :

```
val formattedUserNames: List<String>
   get() {
      val userNames = ArrayList<String>(users.size)
      for ((firstName, lastName) in users) {
         val name = if (lastName != null) {
            if (firstName != null) {
```

```
                "$firstName $lastName"
            } else {
                lastName
            }
        } else {
            firstName ?: "Unknown"
        }
        userNames.add(name)
    }
    return userNames
}
```

Utilisation de map pour des Opérations

Kotlin offre des fonctionnalités avancées pour manipuler les collections de manière plus simple et idiomatique. Parmi celles-ci, la fonction map permet de transformer chaque élément d'une collection en un nouvel élément, en appliquant une fonction de transformation à chaque élément de la liste d'origine. En utilisant map, nous pouvons rendre notre code plus lisible et concis.
Dans l'exemple ci-dessous, le getter formattedUserNames utilise map pour appliquer la logique de formatage des noms d'utilisateur directement à chaque élément de la liste users.

Code d'Origine
Actuellement, le code crée une nouvelle liste userNames, puis parcourt chaque utilisateur dans users, construit un nom formaté, et l'ajoute à la liste. Enfin, il retourne cette liste.

```
val formattedUserNames: List<String>
    get() {
        val userNames = ArrayList<String>(users.size)
        for ((firstName, lastName) in users) {
            val name = if (lastName != null) {
                if (firstName != null) {
                    "$firstName $lastName"
                } else {
                    lastName
                }
            } else {
                firstName ?: "Unknown"
            }
            userNames.add(name)
        }
        return userNames
```

```
}
```

Optimisation avec map

En remplaçant cette logique par map, nous éliminons la nécessité de créer et de gérer manuellement une liste temporaire. La transformation des noms d'utilisateur se fait directement dans le corps de map. Voici le code refactorisé :

```
val formattedUserNames: List<String>
  get() {
    return users.map { user ->
      if (user.lastName != null) {
        if (user.firstName != null) {
          "${user.firstName} ${user.lastName}"
        } else {
          user.lastName ?: "Unknown"
        }
      } else {
        user.firstName ?: "Unknown"
      }
    }
  }
```

Utilisation des Propriétés et des Propriétés de Support

Kotlin simplifie la gestion des propriétés de classe en utilisant des propriétés immuables et des propriétés de support (backing properties). Cela permet un code plus clair et un meilleur contrôle de l'accès aux propriétés. Dans l'exemple ci-dessous, la classe Repository utilise des propriétés de support pour exposer une liste d'utilisateurs en lecture seule tout en permettant des modifications internes.

Code Complet

Voici le code complet de la classe Repository :

```
object Repository {

  // Propriété de support, modifiable seulement en interne
  private val _users = mutableListOf<User>()

  // Propriété publique en lecture seule
  val users: List<User>
    get() = _users
```

```
    // Propriété calculée pour le format des noms d'utilisateur
    val formattedUserNames: List<String>
        get() {
            return _users.map { user ->
                if (user.lastName != null) {
                    if (user.firstName != null) {
                        "${user.firstName} ${user.lastName}"
                    } else {
                        user.lastName ?: "Unknown"
                    }
                } else {
                    user.firstName ?: "Unknown"
                }
            }
        }

    // Bloc d'initialisation pour ajouter des utilisateurs exemples
    init {
        val user1 = User("Zeinab", "")
        val user2 = User("Yassine", null)
        val user3 = User("Youssef", "Tounsi")

        _users.add(user1)
        _users.add(user2)
        _users.add(user3)
    }
}
```

Explication de l'Implémentation

1. Propriété de Support _users :

- La liste _users est une liste mutable de type mutableListOf<User>(), accessible uniquement au sein de la classe Repository.
- Elle est définie comme private pour limiter son accès aux éléments de la classe et garantir que personne d'autre ne peut modifier directement cette liste.

2. Propriété Publique en Lecture Seule users :

- La propriété users est définie comme une propriété publique de type List<User>, permettant un accès en lecture seule aux éléments de la liste depuis l'extérieur de la classe.
- Le getter get() = _users permet d'accéder aux éléments de _users sans exposer la possibilité de modification.

- En Kotlin, cette propriété est non modifiable pour les appels en Kotlin, mais elle reste modifiable lorsqu'elle est utilisée en Java, car elle est traduite en java.util.List.

3. Propriété formattedUserNames :
- formattedUserNames est une propriété calculée qui renvoie une liste formatée des noms des utilisateurs, en utilisant map pour transformer chaque utilisateur en son nom complet ou une chaîne "Unknown" si les informations sont manquantes.

4. Initialisation de la Liste _users :
- Dans le bloc init, trois utilisateurs sont ajoutés à la liste _users à titre d'exemple, illustrant comment la liste peut être modifiée en interne sans exposer cette capacité à l'extérieur de la classe.

Utilisation des Fonctions et Propriétés d'Extension

Kotlin offre des moyens puissants pour étendre les fonctionnalités des classes, notamment à travers les fonctions d'extension et les propriétés d'extension. Ces outils permettent de réutiliser le code et d'éviter la duplication, tout en améliorant la lisibilité.

Problème Actuel

Actuellement, la classe Repository sait comment calculer le nom formaté d'un objet User. Cependant, pour pouvoir réutiliser cette logique dans d'autres classes sans duplication, nous pouvons utiliser des fonctions ou propriétés d'extension.

Déclaration de Fonctions et Propriétés d'Extension

Kotlin permet de déclarer des fonctions et des propriétés en dehors de toute classe, objet ou interface. Pour notre cas d'utilisation, nous allons créer une propriété d'extension pour la classe User afin de calculer le nom formaté.

Fonction d'Extension

Voici comment une fonction d'extension pourrait être définie :

```
fun User.getFormattedName(): String {
    return if (lastName != null) {
        if (firstName != null) {
            "$firstName $lastName"
        } else {
            lastName ?: "Unknown"
        }
    } else {
        firstName ?: "Unknown"
```

```
    }
}
```

Propriété d'Extension

Cependant, dans notre cas, nous allons choisir d'utiliser une propriété d'extension, car le nom formaté est davantage une caractéristique de l'utilisateur qu'une fonctionnalité d'un autre objet. Voici comment déclarer cette propriété d'extension :

```
val User.formattedName: String
    get() {
        return if (lastName != null) {
            if (firstName != null) {
                "$firstName $lastName"
            } else {
                lastName ?: "Unknown"
            }
        } else {
            firstName ?: "Unknown"
        }
    }
```

Mise à Jour de la Classe Repository

Avec la propriété d'extension en place, nous mettons à jour la classe Repository pour utiliser cette fonctionnalité :

```
object Repository {

    private val _users = mutableListOf<User>()
    val users: List<User>
        get() = _users

    val formattedUserNames: List<String>
        get() {
            return _users.map { user -> user.formattedName }
        }

    init {
        val user1 = User("Zeinab", "")
        val user2 = User("Yassine", null)
        val user3 = User("Youssef", "Tounsi")

        _users.add(user1)
```

```
        _users.add(user2)
        _users.add(user3)
    }
}
```

Utilisation des Extensions

Avec cette configuration, nous pouvons utiliser la propriété **formattedName** sur n'importe quel objet User comme s'il s'agissait d'une méthode de la classe :

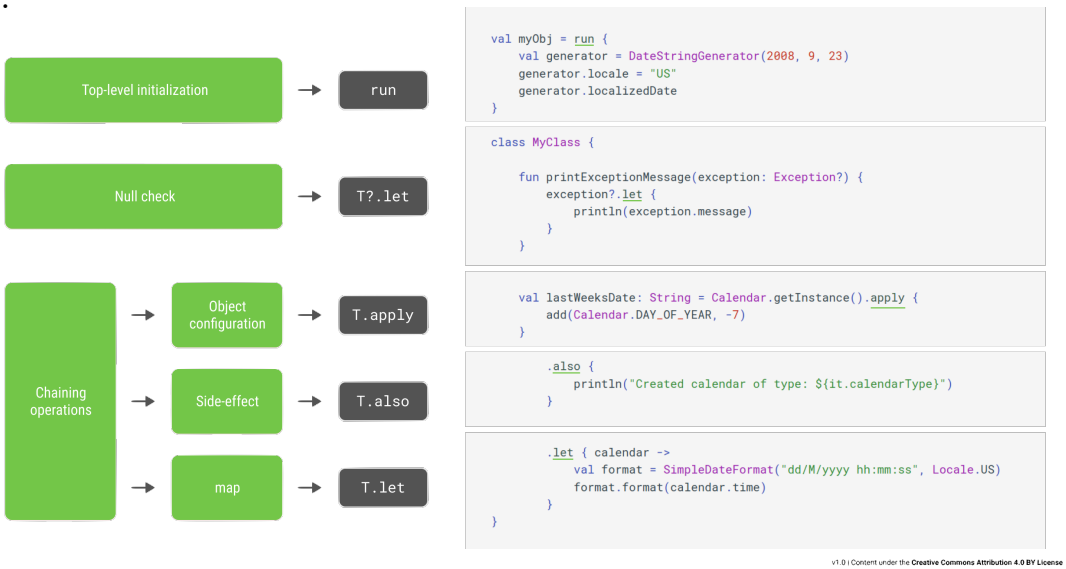
```
val user = User("Youssef", "Tounsi")
val name = user.formattedName  // Utilisation de la propriété d'extension
```

En utilisant des fonctions et des propriétés d'extension, nous avons amélioré la structure du code en rendant la logique de formatage des noms facilement réutilisable dans différentes parties de l'application, sans avoir à modifier la classe User. Les extensions permettent de garder le code organisé et de réduire le couplage entre les classes, ce qui est une bonne pratique en programmation.

Fonctions de portée : let, apply, with, run, also

Dans notre code de classe **Repository**, nous ajoutons plusieurs objets **User** à la liste **_users**. Ces appels peuvent être rendus plus idiomatiques à l'aide des fonctions de portée Kotlin.
Pour exécuter du code uniquement dans le contexte d'un objet spécifique, sans avoir besoin d'accéder à l'objet en fonction de son nom, Kotlin propose 5 fonctions de portée : let, apply, with, run, also. Ces fonctions rendent votre code plus facile à lire et plus concis. Toutes les fonctions de portée ont un récepteur (this), peuvent avoir un argument (it) et peuvent renvoyer une valeur.

Voici une aide-mémoire pratique pour vous aider à vous rappeler quand utiliser chaque fonction :



Puisque nous configurons notre objet **_users** dans notre Repository, nous pouvons rendre le code plus idiomatique en utilisant la fonction apply:

```
init {
    val user1 = User("Zeinab", "")
    val user2 = User("Yassine", null)
    val user3 = User("Youssef", "Tounsi")

    _users.apply {
      // this == _users
      add(user1)
      add(user2)
      add(user3)
    }
}
```