5 - Accès à Internet et coroutines

Introduction

- Pour avoir des donnés toujours à jour, il est préférable de passer par un serveur qui met à dispositions les dernières données
- Mais comment gérer le téléchargement de ces données depuis Internet qui peuvent prendre du temps ?

Comment faire ?

- Comment faire pour que le téléchargement des données ne rende pas l'app lente voir sans réponse ?

Tâches asynchrones

- Télécharger des informations
- Se synchroniser avec un serveur
- Écrire dans un fichier
- Grosse charge de travail
- Écrire, lire dans une base de données

Quand utiliser les tâches asynchrones

- Pas assez de temps pour faire des tâches complexes tout en gardant l'app réactive
- Équilibrer avec le besoin d'exécuter de longues tâches
- Contrôler comment et quand les tâches sont exécutées

Programmation asynchrone sur Android

- Threads
- Callbacks
- Et plus encore
- Mais quel est la meilleure option ?

Coroutines

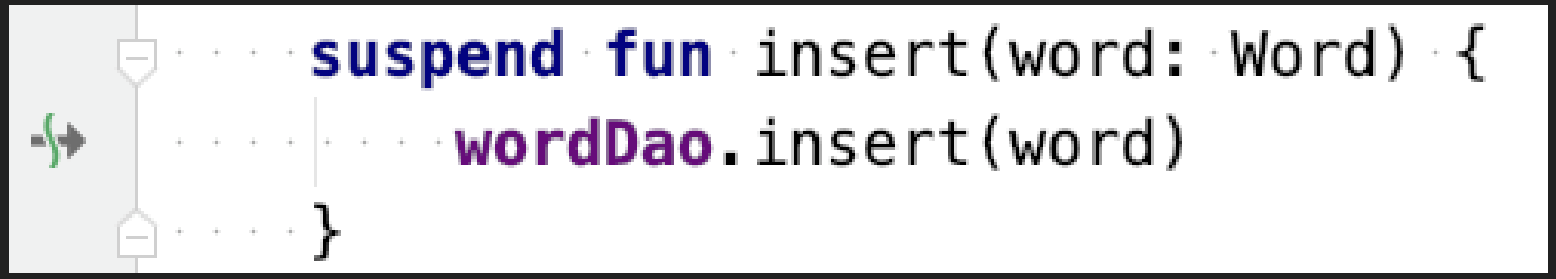
- Garder l'app réactif pendant l'exécution de tâche longue
- Simplifier le code asynchrone
- Écrire le code de façon séquentiel
- Gérer les exceptions avec un block try/catch

Bénéfices des coroutines

- Léger
- Moins de fuites de mémoire
- Gestion de l'annulation de tâches intégré
- Intégration avec Jetpack

Fonction suspend

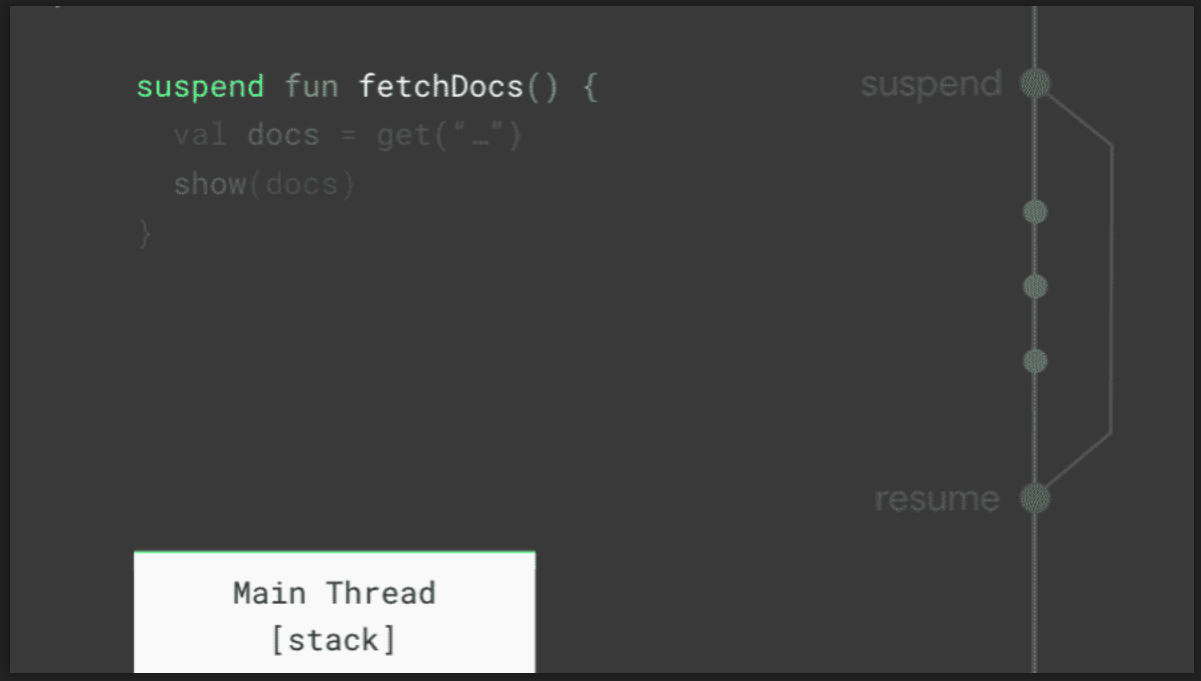
- Ajouter le mot clé **suspend**
- Les fonctions doivent être seulement dans une autre fonction suspend ou une coroutine



Suspend et resume

- **suspend**
  - Suspend l'exécution de la couroutine courante et sauvegarde l'état de ses variables
- **resume**
  - Recharge automatiquement les variables sauvegardées et poursuit l'exécution depuis la ligne qui a été suspendu

Exemple


suspend fun fetchDocs() {
val docs = get("…")
show(docs)
}
suspend
resume
Main Thread
[stack]

Contrôler où s'exécutent les coroutines

Dispatcher	Description of work	Examples of work
`Dispatchers.Main`	UI et courtes tâches non blocante	Mettre à jour un composable, appeler une fonction suspend
`Dispatchers.IO`	Réseau et gestion du disque	Base de données, écriture d'un fichier sur disque
`Dispatchers.Default`	Grosses ressources CPU nécessaire	Parsing un JSON

withContext

```
suspend fun get(url: String) {

// Start on Dispatchers.Main

    withContext(Dispatchers.IO) {
        // Switches to Dispatchers.IO
        // Perform blocking network IO here
    }

    // Returns to Dispatchers.Main
}
```

CoroutineScope

- Les coroutines doivent être lancées dans un CoroutineScope
  - Permet de suivre toutes les coroutine démarrées dedans
  - Fournit un moyen d'annuler toutes les coroutines du scope
  - Apporte un pont entre les fonctions classiques et les coroutines
- Exemples de scope de Jetpack
  - LifecycleScope
  - ViewModelScope

Lancer une nouvelle coroutine

- **launch** – pas de valeur de retour
- **async** – peut retourner une valeur

ViewModelScope

```
class MyViewModel: ViewModel() {

    init {
        viewModelScope.launch {
            // Coroutine that will be canceled
            // when the ViewModel is cleared
        }
    }
    ...
}
```

Exemple ViewModelScope

```
class AppViewModel(val api: RemoteApi)
    : ViewModel() {

 fun getData() {
     viewModelScope.launch {
         val data = api.getData()
        // update state with new data
     }
 }


 ...
}
```

Atelier 1 – Présentation des coroutines

https://developer.android.com/codelabs/basic-android-kotlin-compose-coroutines-kotlin-playground?hl=fr

Atelier 2 – Coroutines dans Android Studio

https://developer.android.com/codelabs/basic-android-kotlin-compose-coroutines-android-studio?hl=fr

Connexion internet quasi indispensable

- Pour avoir des données toujours à jour
- Afficher les données de l'utilisateur stockée dans le cloud
- Ne pas avoir à mettre à jour l'app à chaque fois qu'on veut changer les données

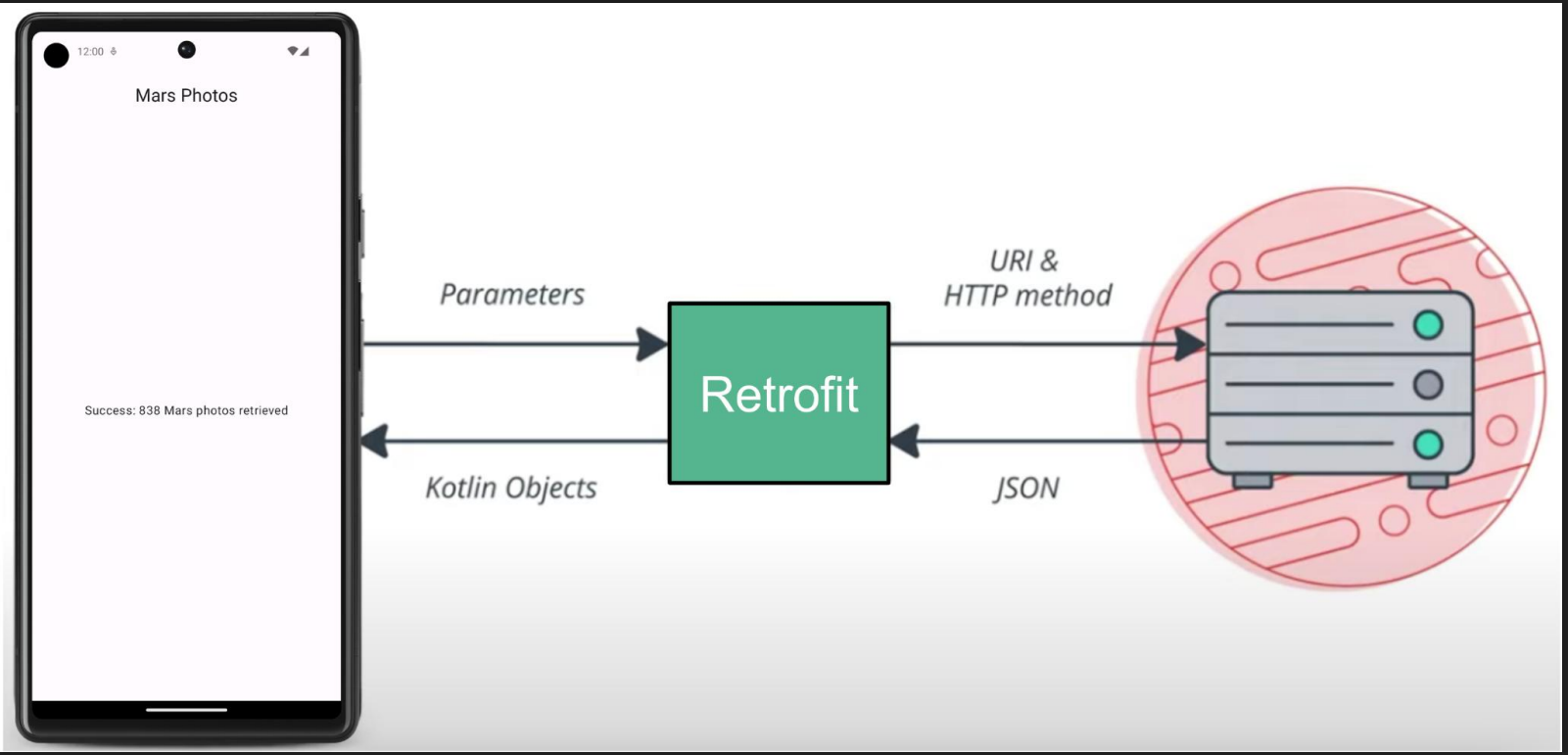
REST

- **RE**presentational **S**tate**T**ransfer
- Manière standardisée de communiquer avec des serveurs Web via des URI
- Chaque URI identifie une ressource grâce à son nom
- Passe par le protocole HTTP et ses opérations (GETT, POST, PUT, DELETE)
- Données récupérées peuvent être en HTML, JSON, XML…

Retrofit

- Bibliothèque open-source
- Largement utilisé par la communauté
- Très bien suivie (mise à jour, corrections de bugs)
- Permet de communiquer avec des services REST depuis l'app

Retrofit


12:00
Mars Photos
Success: 838 Mars photos retrieved
Parameters
Kotlin Objects
Retrofit
URI &
HTTP method
JSON

Permissions

- Android repose sur un système de **permissions** pour restreindre l'accès par des apps tierces aux ressources du téléphone
- L'app doit spécifier les ressources auxquelles elle doit avoir accès pour fonctionner
- Ces permissions doivent être déclarées dans le fichier **AndroidManifest.xml**
- Suivant la sévérité, chaque autorisation est
  - soit acceptée à l'installation (Internet, accéder à un espace mémoire spécifique à l'app…)
  - soit demandé à la volée à l'utilisateur lors de l'utilisation de l'app (localisation, connexion BLE…)

Permission pour Internet

- `<uses-permission android:name="android.permission.INTERNET" />`

Parsing JSON

- Conversion des données JSON envoyé par le seveur en objet Kotlin
- Utilisation de la bibliothèque serialisations pour effectuer cette conversion facilement
- Définir une data class @Serializable
- Donner le même nom aux propriétés que les clés JSON
- OU
- Utiliser l'annotation **@SerialName(value = "json_key")** pour spécifier un nom différent à la propriété

Parsing JSON

```
@Serializable
data class MarsPhoto(
    val id: String,
    @SerialName(value = "img_src")
    val imgSrc: String,
)
```

Retrofit x Serializable

- Définir les points d'entrée de l'api grâce aux URI
- Spécifier les données en sortie qui sont les données parsées

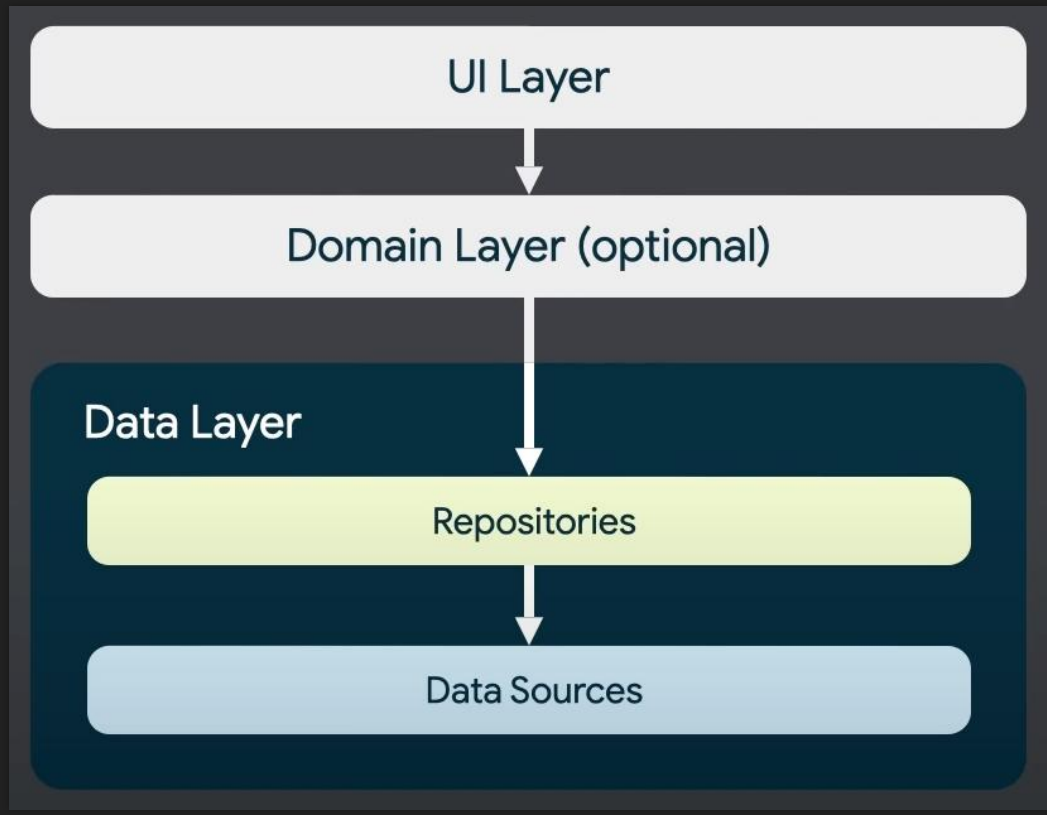
```
interface MarsApiService {
    @GET("photos")
    suspend fun getPhotos(): List<MarsPhoto>
}
```

Atelier 3 – Récupérer des données d'Internet

https://developer.android.com/codelabs/basic-android-kotlin-compose-getting-data-internet?hl=fr

La couche de données

- Elle contient les **données de l'app** et la **logique de l'app** (business logic)
- La couche d'interface utilisateur (et la couche optionnelle du domaine) en dépend(ent)
- Composée de plusieurs **repositories** (dépôts) qui eux-mêmes dépendent de 0, 1 ou plusieurs **data sources** (sources de données)



Data sources

- Fournissent des données peut importe la source
  - Internet
  - Une base de données locale
  - Un fichier
  - Données en mémoire
- S'occupe d'une seule source de donnée
  - Films, utilisateurs, articles…

Repositories

- Permettent aux autres couches d'interagir avec la couche de données
  - Expose les données
  - Centralisent les changements
  - Résolvent les conflits
  - Détiennent la logique de l'app
- Un repository pour chaque type de donnée
- Un repository peut gérer différentes sources de données
  - Si c'est le cas, doit également résoudre les conflits
  - Exemple données différentes entre celles du serveur et stockées en local

Le repository expose les données

- Point d'entrée vers les données
- Créer des méthodes qui font une seule tâche
  - Créer, modifier, obtenir, supprimer
- Peuvent être implémentées avec les fonctions suspend (car cela peut être long)
- Possibilité d'utiliser des flux de données comme les Flows pour exposer des données

Source de vérité pour le repository

- Il faut définir un data source qui pourra toujours nous fournir les données quand nécessaire
- Les sources de conflits sont gérées de sorte que la source de vérité soit le plus à jour

Exemple source de vérité

```
class LocalNewsDataSource(news: NewsDao) {
    suspend fun fetchNews(): List<Article>{ ... }
    suspend fun updateNews(news: List<Article>) { ... }
}
class RemoteNewsDataSource(apiClient: ApiClient) {
    suspend fun fetchNews(): List<Article>{ ... }
}
```

Exemple source de vérité

```
class NewsRepository (
    val localNewsDataSource: LocalNewsDataSource,
    val remoteNewsDataSource: RemoteNewsDataSource
) {
 suspend fun fetchNews() : List<Article>{
     try {
         val news = remoteNewsDataSource.fetchNews()
         localNewsDataSource.updateNews(news)
     } catch (exception: RemoteDataSourceNotAvailableException) {
         Log.d("NewsRepository", "Connection failed, using local data
 source")
     }
     return localNewsDataSource.fetchNews()
 }
```

Classes immuables

- Pour s'assurer que les données ne soient pas modifiées, les données exposées par la couche de données doivent être immuables
- Sinon risque de données inconsistantes entre les classes qui consomment les données
- Gestion simplifiée pour de multiples threads
- Utiliser les **data class** avec des propriétés immuables **val**

Modèle

- Ne pas hésiter à créer différents modèles suivant les besoins
- Les données renvoyées par l'API distant ou la base de données contient souvent des données qui ne seront pas utiles à afficher dans l'app
- Créer des modèles spécifiques pour appliquer la séparation des préoccupations

Exemple modèle

```
data class ArticleApiModel(
    val id: Long,
    val title: String,
    val content: String,
    val publicationDate: Date,
    val modifications: Array<ArticleApiModel>,
    val comments: Array<CommentApiModel>,
    val lastModificationDate: Date,
    val authorId: Long,
    val authorName: String,
    val authorDateOfBirth: Date,
    val readTimeMin: Int
)
```

```
data class Article(
    val id: Long,
    val title: String,
    val content: String,
    val publicationDate: Date,
    val authorName: String,
    val readTimeMin: Int
)
```

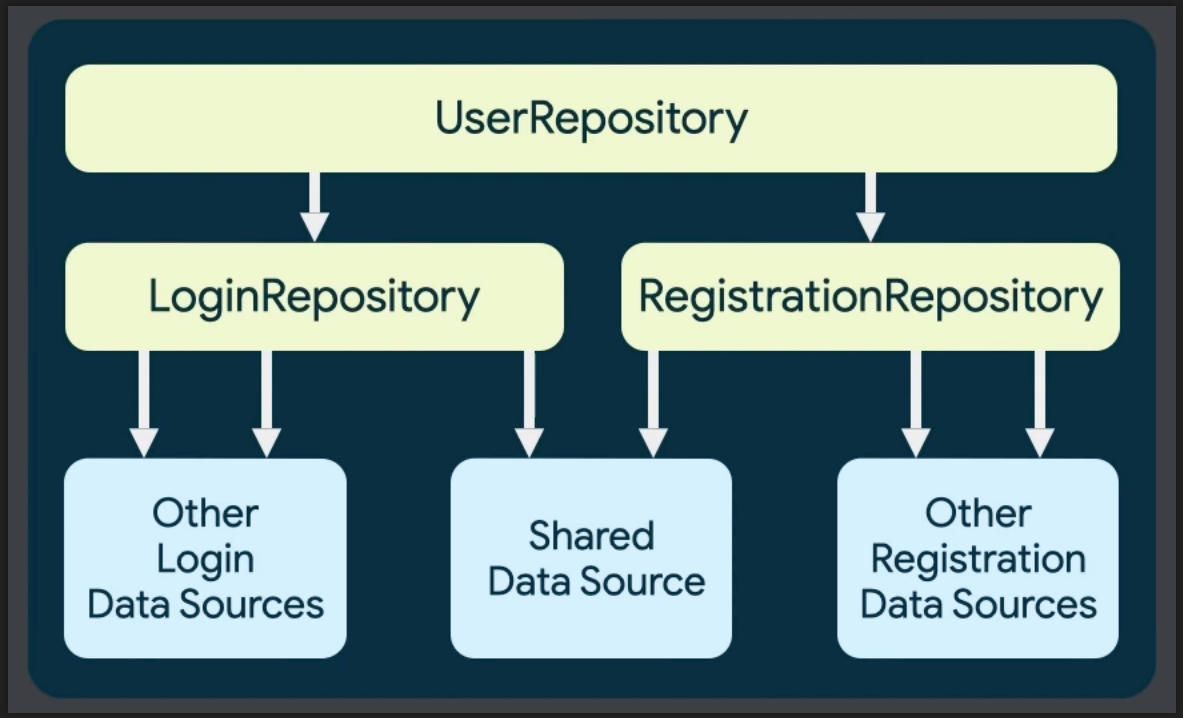
Main safe

- Appeler les méthodes des repositories ou des data sources doit être main-safe
- C'est-à-dire sûres quand on les appelle depuis le main thread
- Les repositories et data sources doivent s'assurer que l'exécution des tâches longues soit déporté et qu'elle n'ait pas lieu depuis le thread principal

Erreurs

- Important de gérer les erreurs car les opérations sur les données peuvent parfois échouées
- Il faut alors remonter l'information de la couche de données vers la couche d'interface
- 2 solutions :
  - Laisser les exceptions remontées au niveau supérieur et les gérer avec block **try/catch**, ou opérateur **catch** avec les Flow
  - Exposer une donnée qui peut être soit de type Succès ou Error

Possibilité d'avoir plusieurs niveaux de repositories



Tester les repositories

- Utiliser des tests unitaires
- Remplacer les data sources par des data sources simulées
- Vérifiés que le repository gère correctement ces données

Tester les data sources

- Plus difficile car dépend souvent de bibliothèques externes
- Utiliser des tests unitaires couplées aux classes de tests ou mécanismes de la bibliothèque en question
- Exemple Jetpack Room (base de données) permet de stocker les données de test dans la mémoire vive

Comment gérer les dépendances ?

- Une classe à souvent besoin d'autres classes pour fonctionner (ex les repositories ont besoin des data sources pour récupérer les données)
- Ce sont des **dépendances**

2 façons d'obtenir les dépendances

- La classe instancie elle-même les objets

```
interface Engine {
    fun start()
}

class GasEngine : Engine {
    override fun start() {
        println("GasEngine started!")
    }
}

class Car {

    private val engine = GasEngine()

    fun start() {
        engine.start()
    }
}

fun main() {
    val car = Car()
    car.start()
}
```

- Transmettre les objets en arguments

```
interface Engine {
    fun start()
}

class GasEngine : Engine {
    override fun start() {
        println("GasEngine started!")
    }
}

class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

fun main() {
    val engine = GasEngine()
    val car = Car(engine)
    car.start()
}
```

Problèmes que la classe instancie les objets

- Rend la classe inflexible, complexe à tester car la classe et les dépendances sont fortement couplées
- La classe doit gérer la construction de l'objet, alors que c'est un détail d'implémentation
- Plusieurs instances sont créées alors qu'une seule aurait pu suffire
- Pour être plus flexible et adaptable, une classe ne doit pas instancier ses dépendances mais être construites à l'extérieur et reçues en paramètre
- Ainsi, plus besoin de modifier chaque classe qui dépendent d'un objet pour modifier son implémentation

Injection de dépendances

- C'est l'injection de dépendances
- Aide à la réutilisation du code
- Facilite la factorisation
- Aide à la réalisation de tests

```
interface Engine {
    fun start()
}

class ElectricEngine : Engine {
    override fun start() {
        println("ElectricEngine started!")
    }
}

class Car(private val engine: Engine) {
    fun start() {
        engine.start()
    }
}

fun main() {
    val engine = ElectricEngine()
    val car = Car(engine)
    car.start()
}
```

Comment récupérer les dépendances ?

- Il existe des bibliothèques qui permettent de simplifier l'injection de dépendances en définissant des modules qui fournissent pour chaque interface, l'implémentation à utiliser et simplifient son utilisation dans les classes qui en dépendent (Koin, Hilt, Dagger…)
- Nous verrons une version simplifiée dans ce cours qui ne dépend pas d'une autre bibliothèque
- Définition d'un **conteneur d'application**, objet qui contient et définit les **dépendances** de l'application
- Utilisation de la classe **Application** qui a une instance unique qui représente notre application Android et qui détiendra notre conteneur d'application

Conteneur d'application

```kotlin
class DefaultAppContainer : AppContainer {

    private val baseUrl =
        "https://android-kotlin-fun-mars-server.appspot.com"

    /**
     * Use the Retrofit builder to build a retrofit object using a kotlinx.serialization
     */
    private val retrofit = Retrofit.Builder()
        .addConverterFactory(Json.asConverterFactory("application/json".toMediaType()))
        .baseUrl(baseUrl)
        .build()

    private val retrofitService: MarsApiService by lazy {
        retrofit.create(MarsApiService::class.java)
    }

    override val marsPhotosRepository: MarsPhotosRepository by lazy {
        NetworkMarsPhotosRepository(retrofitService)
    }
}
```

```kotlin
package com.example.marsphotos

import android.app.Application
import com.example.marsphotos.data.AppContainer
import com.example.marsphotos.data.DefaultAppContainer

class MarsPhotosApplication : Application() {
    lateinit var container: AppContainer
    override fun onCreate() {
        super.onCreate()
        container = DefaultAppContainer()
    }
}
```

Atelier 4 - Repository et injection de dépendances dans Mars Photos

https://developer.android.com/codelabs/basic-android-kotlin-compose-add-repository?hl=fr

La bibiliothèque Coil

- Bibliothèque open-source
- Basée sur les coroutines
- Légère et rapide
- Compatible avec Compose
- Permet de charger des images d'internet dans son application

Utilisation de Coil

- Ajout de la dépendance dans le fichier gradle de l'app
  - `implementation("io.coil-kt:coil-compose:2.4.0")`
- Utilisation de la fonction Composable **AsyncImage**

```
AsyncImage(
    model = "https://android.com/sample_image.jpg",
    contentDescription = null
)
```

Utilisation de Coil

- **model** est l'argument qui prend en paramètre un **ImageRequest** qui défini comment est chargé l'image à l'aide d'un builder
  - **data** permet de définir comment charger l'image (URL, drawable, bitmap, fichier…)
  - **placeholder** permet de définir une image en attendant que l'image cible soit chargé
  - **crossfade** permet d'ajout une transition fondu au chargement de l'image
  - **error** permet de spécifier une image à afficher quand l'image cible ne peut pas être chargé

Atelier 5 - Afficher des images d'Internet

https://developer.android.com/codelabs/basic-android-kotlin-compose-load-images?hl=fr