

Le TP consiste à créer une application Android qui affiche une liste de données. Le TP6 lui ajoutera des fonctionnalités d'édition.

Cette application Android va afficher une liste de données. On a choisi les personnages de la série X-Men, parce que les données sont facilement disponibles¹, variées et pas trop nombreuses.

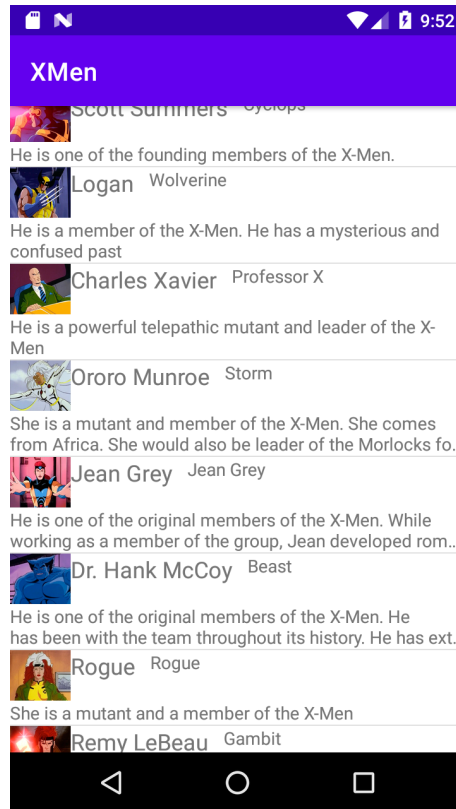


Figure 1: Projet XMen

Ce TP est extrêmement important car il met en œuvre de nombreux concepts essentiels dans Android, dont les *adaptateurs*, et aussi des aspects très particuliers de Kotlin.

👉 Créez un nouveau projet appelé **XMen**, de type « Empty Views Activity ». Attention, le package doit être nommé **fr.iutlan.xmen**. Le non-respect de cette consigne entraîne automatiquement la perte d'un point sur ce TP (1/3 de la note).

1. Classes Kotlin

On commence par quelques explications sur la programmation de classes en Kotlin.

👉 Pour essayer les extraits qui suivent, ouvrez l'URL <https://play.kotlinlang.org>. C'est une fenêtre partagée en deux, le source Kotlin et les résultats en dessous. C'est un peu moins confortable que les fichiers « scratch » : menu **File**, item **New**, sous-item **Scratch File**.

¹C'était une API Rest accessible par <https://api-xmen-animated-series.netlify.app/index.html>, chercher en bas **Characters**, mais elle ne fonctionne plus et n'a pas été reprise. Voir aussi sa [page GitHub](#).

1.1. Définition d'une classe

Voici une classe toute simple :



```
class Animal(val nom: String, var mois: Int) {  
  
    fun saluer(mot: String = "Bonjour", fin: String = ".") {  
        println("$mot $nom$fin")  
    }  
  
    override fun toString(): String {  
        return "$nom : $mois mois"  
    }  
}
```

Kotlin est extrêmement compact comparé à Java.

1. La première ligne déclare à la fois une classe, son constructeur, ainsi que deux variables membres. En Kotlin, si on ajoute des sortes de paramètres préfixés par **val** ou **var** après le nom de la classe, ils deviennent des constantes ou variables membres.
2. Il y a une méthode **saluer** qui prend un paramètre optionnel. Cette valeur est utilisée si le paramètre **mot** n'est pas fourni à l'appel.
3. Il y a une surcharge de la méthode **toString()**.

1.2. Instancier une classe

Pour créer une instance, on ne met pas **new** :



```
fun main() {  
    val monchien = Animal("filou", 36)  
  
    monchien.saluer()  
    println("Je possède ${monchien}.")  
}
```

L'appel à la méthode **saluer** ne fournit pas d'arguments, et donc ce sont les valeurs par défaut qui sont utilisées.

La dernière instruction fait afficher : **Je possède filou : 36 mois.** car il y a un appel implicite à la méthode **toString()**. Dans un modèle de chaînes (*template*), on peut entourer une variable par des **{}** pour la séparer du reste, mais ce n'est pas obligatoire.

Les arguments des fonctions et constructeurs peuvent être nommés et dans ce cas, on peut les fournir dans un autre ordre (les **...** signifient de garder ce qu'il y avait déjà) :



```
fun main() {  
    ...  
  
    val monchien2 = Animal(mois=64, nom="foufou")  
  
    monchien2.saluer(fin=" !", mot="Coucou")  
}
```

```
println("Je possède ${monchien2}.")  
}
```

1.3. Initialisation

Dans certains cas, on a besoin d'initialiser l'instance en cours de création :



```
class Animal(val nom: String, var mois: Int) {  
    ...  
  
    init {  
        println("ouaf miaou cuicui")  
    }  
}
```

C'est un peu comme un complément au constructeur. Toutes les instances créées afficheront ce message. On en verra l'utilité dans le TP.

1.4. Types simples

Les types de base sont un peu différents de ceux du Java : Boolean, Char, String, Int, Float...

1.5. Variables membres

Quand on définit une variable membre dans une classe, Kotlin crée automatiquement un *setter* et un *getter* (les lignes suivantes sont à mettre en dehors de la classe) :



```
fun main() {  
    ...  
  
    monchien.mois = 38  
    println("Je possède ${monchien.nom} qui a ${monchien.mois} mois")  
}
```

Les *getters* et *setters* sont écrits comme des accès à des variables membres publiques en Java, mais en interne, ce sont bien des méthodes, générées automatiquement, qui sont appelées. C'est assez compliqué et voici davantage d'explications.


En Java, toute variable membre est stockée en mémoire.

```
class Voiture {  
    private String marque;  
    private int annee;  
  
    public Voiture(String marque, int annee) {  
        this.marque = marque;  
        this.annee = annee;  
    }  
}
```

```
Voiture macaïsse = new Voiture("citrengéot", 2010)
```

L'instance est stockée en mémoire avec des octets réservés pour chacune des variables membres.

Kotlin a une vision un peu différente. Une classe contient des *propriétés* (*properties*) qui peuvent ne pas être stockées en mémoire. Une propriété Kotlin est constituée d'un *setter* et/ou *getter*.

Voici l'exemple de propriétés qui ne sont pas stockées en mémoire, mais qui sont utilisables exactement comme des variables membres : 

```
class Animal(val nom: String, var mois: Int) {
    ...

    var annees          // type Int optionnel, déduit du getter
        get() = mois / 12
        set(annees) {
            mois = annees * 12
        }


    val isJeune: Boolean
        get() {
            return mois < 36
        }
}

fun main() {
    ...

    monchien.annees = 13          // setter sur une propriété non stockée en mémoire
    println("${monchien.nom} a ${monchien.mois} mois, càd ${monchien.annees} ans.")
    println("${monchien.nom} est-il jeune ? ${monchien.isJeune}.")
}
```

Les propriétés `annees` et `isJeune` ne sont que calculées. Elles ne sont pas stockées en mémoire. Pour cela, il faut programmer à la fois le *setter* et le *getter*.


De plus, la propriété `isJeune` est déclarée `val`, car elle n'est pas modifiable (dans ce cas, pas de *setter*).

S'il y a une vraie variable membre occupant des octets en mémoire, comme `nom` et `mois`, alors cette variable est appelée *champ* (*field*). Ce champ est qualifié de *backing field* (doublement de la propriété par de la mémoire). Pour définir un champ : 

```
class Animal(val nom: String, var mois: Int) {
    ...

    var dateNaissance: String? = null
        // getter et setter générés automatiquement
}
```

```
fun main() {  
    ...  
  
    monchien.dateNaissance = "31/02/2019"  
    println("Il est né le ${monchien.dateNaissance}, je crois...")  
}
```

Dans un *setter*, pour affecter le champ en mémoire, étrangement, on ne peut pas utiliser `this`. Il faut employer le mot-clé `field`. Voici une propriété avec champ, *getter* et *setter* : 

```
class Animal(val nom: String, var mois: Int) {  
    ...  
  
    var race: String = "?"  
    get() {  
        return "${field} très joueur"  
    }  
    set(t) {  
        field = "sublime $t"  
    }  
}  
  
fun main() {  
    ...  
  
    monchien.race = "épagneul breton"  
    println("C'est un ${monchien.race}.")  
}
```

Le `println` affiche "C'est un sublime épagneul breton très joueur.". Notez que ce *setter* enregistre une valeur modifiée, et ce *getter* re-modifie la valeur qui avait été enregistrée – normalement, on ne fait pas ça.

Il y a plusieurs syntaxes pour les *getter*, soit `get() = expression`, soit `get() { return expression }`.

1.6. Héritage


Pour définir une sous-classe, il faut que la superclasse soit dérivable. Pour cela, il faut que le mot-clé `open` lui ait été ajouté lors de sa définition : 

```
open class Animal(val nom: String, var mois: Int) {  
    ...  
}  
  
class Chat(nom: String, val couleur: String, mois: Int) : Animal(nom, mois) {  
    override fun toString(): String {  
        return "chat $nom : $couleur, $mois mois"  
    }  
}
```

```
fun main() {  
    ...  
  
    val monchat = Chat("dikie", "beige et blanc", 96)  
    monchat.saluer()  
    println("Je possède ${monchat} et ${monchien}.")  
}
```

On ne doit pas mettre `var` ou `val` pour des propriétés qui sont dans la superclasse, uniquement pour celles qui sont dans la sous-classe.

1.7. Méthode et constantes de classe

En Java, les méthodes et variables qui s'appliquent à la classe sont définies avec le mot-clé `static`. On les utilise ensuite par `NomClasse.nomMéthode(...)` et `NomClasse.nomVariable`, c'est à dire sur la classe elle-même. En Kotlin, la définition est syntaxiquement assez bizarre. On définit un objet qui accompagne la classe. Mais ensuite, à l'usage, c'est comme en Java : 

```
open class Animal(val nom: String, var mois: Int) {  
    ...  
  
    companion object {  
        var nombreNaissances: Int = 0  
  
        fun naissance(nom: String): Animal {  
            nombreNaissances++  
            return Animal(nom, 0)    // constructeur  
        }  
    }  
}  
  
fun main() {  
    ...  
  
    val monchiot = Animal.naissance("tifilou")  
    println("Je possède maintenant ${monchiot}.")  
    println("J'ai vu ${Animal.nombreNaissances} naissances.")  
}
```

Ces concepts permettent de comprendre la syntaxe utilisée avec l'API Realm.

2. Base de données Realm

Dans le TP5 et le TP6, nous allons utiliser une API appelée *Realm*² pour stocker et interroger des données, voir le [site web](#). Realm n'est pas un SGBD classique, mais une [base de données objet](#) :

²Realm a été renommé en Atlas Device SDK, mais le nom Realm est plus court à employer.

- Une table est représentée par une classe,
- Un n-uplet dans une table est représenté par une instance de la classe,
- Créer un n-uplet dans une table se fait en créant une instance,
- Modifier un n-uplet se fait en utilisant ses *setters*,
- Interroger la base se fait à l'aide de méthodes sur un objet représentant la base,
- etc.

Tout se fait en termes de méthodes sur des objets et des classes, et non pas de requêtes SQL. Il est très simple de créer un modèle de données : il suffit de définir autant de classes qu'il y a de tables, avec les colonnes en tant que variables membres. Les relations entre tables sont représentées par des variables membre du type de l'autre classe.

En ce qui concerne les aspects stockage, les données d'une base peuvent être uniquement en local sur le smartphone, c'est ce qu'on va faire dans ce TP, ou elles peuvent être enregistrées *sur le cloud*, c'est à dire sur un serveur distant et accessible en permanence de partout. Dans ce dernier cas, le support des données est un [cloud MongoDB](#), voir le cours de Big Data.

Un des points forts de Realm, quand les données sont dans le cloud, c'est qu'elles sont à la fois :

- partageables entre plusieurs utilisateurs, c'est à dire que chacun voit les mêmes données que les autres, si c'est voulu par l'application, sinon les données peuvent être séparées. Comme dans un SGBD SQL, il y a une notion de compte utilisateur et de droits.
- synchronisées, c'est à dire que les données peuvent être manipulées par un ou plusieurs utilisateurs sur un ou plusieurs appareils en même temps (smartphone et ordinateur), et la magie de Realm, c'est qu'un changement fait sur l'un des appareils est aussitôt communiqué à tous les autres appareils, avec extrêmement peu de trafic réseau.

Dernier avantage de Realm, c'est que la même API est disponible sur de nombreuses plateformes : Android, iOS, React Native, Node.js, .NET, Flutter... et les données manipulées sur l'une des plateformes sont directement utilisables sur les autres.

Tous ces avantages sont évidemment payants. Pour nos TP, on choisira une solution gratuite qui consiste à ne pas stocker les données dans le cloud.

Voyons maintenant comment on utilise Realm dans une application Android.

2.1. Configuration du projet

Il faut modifier les deux fichiers `build.gradle` (`build.gradle.kts` sur une version récente d'Android Studio), celui du projet global et celui du dossier `app`. Le problème sur les machines de l'IUT est que la version des outils est relativement ancienne (9 mois). Voici ce qui fonctionne en février 2024.

👉 Vous pouvez faire ces modifications sur le projet XMen dès maintenant.

2.1.1. projet/build.gradle

Attention à ne pas vous tromper de fichier. C'est le premier dans la liste `Gradle Scripts` affichée par Android Studio.

👉 Complétez de cette manière :



```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "io.realm:realm-gradle-plugin:10.15.1"
    }
}
plugins {
    id 'com.android.application' version '8.0.2' apply false
    id 'com.android.library' version '8.0.2' apply false
    id 'org.jetbrains.kotlin.android' version '1.8.20' apply false
    id 'org.jetbrains.kotlin.kapt' version '1.8.20' apply false
}
```

☛ Cliquez sur le lien **Sync now**. Android Studio télécharge des dépendances et il ne doit pas y avoir d'erreur à la fin.

2.1.2. projet/app/build.gradle

☛ Complétez de cette manière :



```
plugins {
    id 'com.android.application'
    id 'org.jetbrains.kotlin.android'
    id 'org.jetbrains.kotlin.kapt'
}

apply plugin: 'realm-android'

android {
    ... ne rien changer ...

    buildTypes {
        ... ne rien changer ...
    }
    buildFeatures {
        viewBinding = true // génération des view bindings
    }
    compileOptions {
        sourceCompatibility = JavaVersion.VERSION_17
        targetCompatibility = JavaVersion.VERSION_17
    }
    kotlinOptions {
        jvmTarget = '17'
    }
}
```



```
dependencies {  
    implementation 'androidx.recyclerview:recyclerview:1.3.2'  
  
    ... autres dépendances à ne pas modifier ...  
}
```

À l'IUT, ne changez pas les numéros de version, même s'ils sont en orange.

Sur la version actuelle d'Android Studio, on met `apply(plugin = "realm-android")` et la déclaration des plugins est légèrement différente, avec des parenthèses, comme des appels de fonctions.

☛ Cliquez sur le lien `Sync now`. Il ne doit pas y avoir d'erreur à la fin.

2.2. Initialisation d'une base

Avant toute chose, il faut initialiser l'API Realm au démarrage de l'application. Pour cela, il faut impérativement créer une sous-classe d'`Application` déclarée dans le manifeste.

☛ Créez une classe Kotlin `XMenApplication` comme ceci :



```
package fr.iutlan.xmen  
  
import android.app.Application  
import io.realm.Realm  
import io.realm.RealmConfiguration  
  
class XMenApplication : Application() {  
  
    override fun onCreate() {  
        super.onCreate()  
        Realm.init(this)  
        val config = RealmConfiguration.Builder()  
            .name("my-realm")  
            .deleteRealmIfMigrationNeeded()    // voir remarque  
            .compactOnLaunch()  
            .build()  
  
        // set this config as the default realm  
        Realm.setDefaultConfiguration(config)  
  
        // si nécessaire pour d'autres initialisations, récupérer la base  
        // val realm = Realm.getDefaultInstance()  
    }  
}
```

Remarque : comme dans toute base de données, on doit prévoir la *migration des données* en cas de changement de modèle. Par exemple, si on ajoute une colonne, on doit recopier les données et mettre une valeur par défaut à la nouvelle colonne. Mais ici, `deleteRealmIfMigrationNeeded` demande de tout supprimer.

Ces instructions vont créer une base Realm locale au smartphone. Elle ne sera pas sur le cloud.

👉 Déclarez cette classe `XMenApplication` dans `AndroidManifest.xml` :



```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools">

    <application android:name=".XMenApplication"
        ...
```

2.3. Utilisation de la base dans une activité

Pour utiliser Realm dans une activité, on doit récupérer une référence sur l'objet `realm` qui représente la base et la libérer à la destruction de l'activité.

👉 Modifiez `MainActivity` ainsi :



```
package fr.iutlan.xmen

import androidx.appcompat.app.AppCompatActivity
import android.os.Bundle
import io.realm.Realm

class MainActivity : AppCompatActivity() {

    // représente la base de données Realm
    lateinit var realm: Realm

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        realm = Realm.getDefaultInstance()
    }

    override fun onDestroy() {
        realm.close()
        super.onDestroy()
    }
}
```

Toutes les actions sur la base s'effectuent via cet objet `realm`.

3. API Realm

Avant de continuer, voici quelques informations théoriques sur Realm. Certaines serviront dans ce TP, les autres dans le TP6.

Rq: ne saisissez pas les exemples fournis. Dans le TP5, on gère des XMen, pas des animaux.

3.1. Définition d'une table

On doit seulement créer une classe dérivable (*open*) qui hérite de `RealmObject`. Les champs de cette classe sont les colonnes de la table.

Voici un exemple :



```
import io.realm.RealmObject
import org.bson.types.ObjectId

open class Animal(
    var id: ObjectId? = null,
    var nom: String? = null,
    var mois: Int? = null
) : RealmObject()
```

Ça définit une table appelée `Animal` ayant trois colonnes, `id`, `nom` et `mois`, tous `var`. L'identifiant est d'un type particulier, `ObjectId`, un type défini par MongoDB, qui garantit l'unicité, sans occuper trop de place en mémoire.

Quand un champ est un identifiant, on le préfixe par l'annotation `@PrimaryKey` :



```
import io.realm.annotations.PrimaryKey

open class Animal(
    @PrimaryKey var id: ObjectId? = null,
    var nom: String? = null,
    var mois: Int? = null
) : RealmObject()
```

Un champ marqué par `@PrimaryKey` est automatiquement indexé ; les recherches sont très rapides.

Il suffit d'avoir cette classe quelque part dans l'application pour que la table soit créée. En général, on regroupe ces classes dans un sous-package appelé `model`.

3.2. Création de n-uplets

Voici une méthode qui crée un n-uplet :




```
fun createAnimal(realm: Realm, nom: String, mois: Int,
    id: ObjectId = ObjectId()): ObjectId {
    val animal = realm.createObject(Animal::class.java, id)
    animal.nom = nom
    animal.mois = mois
    return id
}
```

- Cette méthode demande trois paramètres obligatoires, `realm`, `nom` et `mois`. Le quatrième paramètre, `id` peut être omis, c'est pour ça qu'il est le dernier, et dans ce cas, il prend une valeur générée par `ObjectId()` qui est le constructeur d'un identifiant unique. La fonction retourne cet identifiant.

- La méthode `realm.createObject(classe, identifiant)` est une fabrique. Elle demande la classe de l'objet à créer ainsi que son identifiant, celui qui est marqué par `@PrimaryKey`.

3.3. Création de n-uplets dans le modèle

Ce qui est utile, c'est d'en faire une méthode de classe, et aussi d'ajouter les noms des champs ; ils sont utiles pour les requêtes : 

```
open class Animal(  
    @PrimaryKey var id: ObjectId? = null,  
    var nom: String? = null,  
    var mois: Int? = null  
) : RealmObject() {  
  
    companion object {  
        // noms des champs  
        val ID    = "id"  
        val NOM   = "nom"  
        val MOIS  = "mois"  
  
        // création d'une instance  
        fun create(realm: Realm,  
            nom: String, mois: Int,  
            id: ObjectId = ObjectId())  
        ): ObjectId {  
            val animal = realm.createObject(Animal::class.java, id)  
            animal.nom = nom  
            animal.mois = mois  
            return id  
        }  
    }  
}
```

On peut alors faire `val idAnimal = Animal.create(realm, "youki", 7)`. Attention, cette méthode ne retourne pas l'objet créé, mais son identifiant. Il faut faire une requête pour récupérer l'objet créé, mais vous verrez que c'est plus utile comme ça.

C'est une bonne solution, parce que si on veut modifier le modèle, tout est sous les yeux. Ça ne se compilera pas s'il y a un défaut (noms ou types des champs qui ne correspondent pas).

3.4. Transactions asynchrones

La méthode précédente ne peut pas être employée n'importe comment, parce que toute modification des données doit se faire dans une transaction. C'est simplement une *lambda* qui est fournie à une méthode de lancement.

Voici comment faire : 

```
realm.executeTransactionAsync {  
    Animal.create(it, "filou", 32)
```

```
}
```

Il y a un petit point de syntaxe Kotlin. Dans un cas comme ça, où il n'y a que le corps de la *lambda* sans paramètre, alors le paramètre s'appelle **it** (à ne pas confondre avec **id**).

La méthode `executeTransactionAsync` permet de ne pas bloquer l'application quand l'insertion de données prend du temps. La *lambda* est exécutée dans un *thread* d'arrière-plan. L'insertion n'est pas faite immédiatement, mais au bout de quelques instants. Donc, attention, si vous faites une requête dessus juste après, votre objet n'existera pas (encore).

C'est pour ça qu'on n'a pas le droit d'utiliser la variable **realm** à l'intérieur de la *lambda*. Il faut utiliser **it**. La raison, c'est que les objets Realm ne sont pas transmis entre *threads*. La mémoire allouée pour un objet Realm dans un *thread* ne peut pas être utilisée par un autre *thread*.

Voici une petite amélioration, dans le cas où il y a beaucoup de colonnes à fournir, c'est de nommer les arguments pour ne pas se tromper :



```
realm.executeTransactionAsync {  
    Animal.create(  
        realm = it,  
        nom = "filou",  
        mois = 32)  
}
```

Le nommage des paramètres permet de réduire les erreurs, de fournir les arguments dans un autre ordre et de faire utiliser les valeurs par défaut quand tous les arguments ne sont pas fournis (nombreux avantages, pas d'inconvénients, donc solution à préférer).

3.5. Sélection de n-uplets

Avant de voir comment modifier ou supprimer des données, voici comment on interroge la base pour récupérer une liste de n-uplets ou seulement l'un d'entre eux.

Voici la requête la plus simple qui récupère la liste de tous les n-uplets d'une table :



```
val results = realm.where(Animal::class.java).findAllAsync()
```

La méthode **where** aurait pu s'appeler **from** pour ressembler au SQL. On lui indique le nom de la classe, avec cette syntaxe bizarre **Classe::class.java** pour la compatibilité avec Java.

La méthode `findAllAsync` retourne une liste d'objets qui est du type `RealmResults<Animal>`. C'est un peu comme un `ArrayList<Animal>`, mais ce qui est absolument magique, c'est que cette liste est automatiquement mise à jour en cas de changement des données, par exemple dans le cloud. Un `RealmResults` n'est donc pas comme un simple `ArrayList`. C'est vivant. Ça se remplit, se vide, se modifie en temps réel, grâce à des écouteurs gérés automatiquement.

La requête est *asynchrone*, c'est à dire que la requête démarre quand on exécute cette instruction, mais la liste retournée est vide au début. Elle se remplira ultérieurement quand les données seront disponibles.

3.5.1. Tri des n-uplets

Comme avec SQL, on peut trier la sélection selon une colonne :



```
val sortedResults = realm.where(Animal::class.java).findAllAsync()  
    .sort(Animal.MOIS, Sort.DESENDING)
```

On voit l'intérêt d'utiliser des symboles pour les noms des champs. On est sûr de ne pas faire d'erreur.

3.5.2. Conditions simples

Comme avec SQL, on peut rajouter des conditions :



```
val results = realm.where(Animal::class.java)  
    .equalTo(Animal.NOM, "filou").greaterThan(Animal.MOIS, 3)  
    .findAllAsync()
```

La méthode `equalTo(nom_champ, valeur)` définit une condition sur le champ fourni par son nom et la valeur. D'autres comparaisons existent, voir la [doc de RealmQuery](#) :

- pour tous les types de champs : `equalTo`, `notEqualTo`, `in`
- pour les nombres : `between`, `greaterThan`, `lessThan`, `greaterThanOrEqualTo`, `lessThanOrEqualTo`...
- pour les chaînes : `contains`, `beginsWith`, `endsWith`, `like`...

3.5.3. Agrégation des n-uplets

Pour compter les n-uplets d'une requête :



```
val nombre = results.count()
```

On peut calculer différentes agrégations d'un champ sur les n-uplets d'une requête :



```
val ageMoyen = results.average(Animal.MOIS)  
val ageMin   = results.min(Animal.MOIS)  
val ageMax   = results.max(Animal.MOIS)  
val ageTotal = results.sum(Animal.MOIS)
```

3.5.4. Sélection d'un seul n-uplet

Pour ne récupérer qu'un seul n-uplet, par son indice dans la liste :



```
val monchien = realm.where(Animal::class.java).findAll()[indice]
```

Pour ne récupérer que le premier n-uplet, par exemple sélectionné par son identifiant :



```
val monchien = realm.where(Animal::class.java).equalTo(Animal.ID, idmonchien)  
    .findFirst()
```

Attention, `findFirst()` ne doit être utilisé que dans le *thread* d'arrière-plan, c'est à dire dans une transaction `executeTransactionAsync`.

3.6. Modification d'un n-uplet

Il suffit de modifier les champs d'un objet avec les *setters*, mais il faut le faire dans une transaction. D'autre part, on ne peut travailler qu'à partir de l'identifiant de l'objet, car on ne peut pas

transférer des objets entre *threads*. Donc il faut faire une requête dans la transaction pour obtenir l'objet à modifier.

Voici un exemple :



```
fun renommerAnimal(idAnimal: ObjectId, nouveauNom: String) {
    realm.executeTransactionAsync {
        val animal = it.where(Animal::class.java).equalTo(Animal.ID, idAnimal)
            .findFirst()
        animal?.nom = nouveauNom
    }
}
```

Remarquez la notation `animal?.nom` parce que la requête peut retourner `null` si elle ne trouve pas l'objet.

3.7. Suppression de n-uplets

Pour supprimer un seul n-uplet, il suffit d'appeler la méthode `deleteFromRealm()` dessus, mais il faut être dans une transaction. Attention au fait que l'objet peut être `null`.

Voici un exemple :



```
fun supprimerAnimal(idAnimal: ObjectId) {
    realm.executeTransactionAsync {
        val animal = it.where(Animal::class.java).equalTo(Animal.ID, idAnimal)
            .findFirst()
        animal?.deleteFromRealm()
    }
}
```

Si vous voulez supprimer plusieurs n-uplets désignés par une requête :



```
fun supprimerTousLesVieux() {
    realm.executeTransactionAsync {
        it.where(Animal::class.java).greaterThan(Animal.MOIS, 3).findAll()
            .deleteAllFromRealm()
    }
}
```

Si vous voulez supprimer tous les n-uplets de toutes les tables :



```
fun supprimerTOUT() {
    realm.executeTransactionAsync {
        it.deleteAll()
    }
}
```

4. Construction du projet

On revient au projet `XMen` pour appliquer ce qu'on vient de voir.

☛ Vous avez déjà modifié les deux fichiers `build.gradle` sans erreur. Vous avez rajouté la classe `XMenApplication` et vous l'avez déclarée dans le manifeste. Vous avez aussi modifié la classe `MainActivity` avec `onCreate` et `onDestroy` pour récupérer l'objet `realm` dans une variable membre.

4.1. Données initiales

On commence par mettre en place les données initiales.

☛ Téléchargez le fichier : [XMenRes.zip](#). Il contient un dossier `res` qui est à **fusionner** avec celui de votre projet, `app/src/main/res`. À vous de voir comment faire : les dossiers qui ont le même nom doivent être fusionnés. Enlevez ensuite le fichier `XMenRes.zip`.

☛ Une fois que c'est fait correctement, vous devriez trouver un fichier `res/values/arrays.xml` dans le projet. Regardez ce qu'il contient. Il y a 5 tableaux décrivant les personnages : les noms, les alias, les descriptions, les pouvoirs et les identifiants d'images. Ces identifiants d'images correspondent à des images placées dans les dossiers `res/drawable` (allez voir).

4.2. Modèle des données

☛ Créez un sous-package appelé `model` dans `fr.iutlan.xmen` et dedans, créez une classe appelée `XMen` :

```
package fr.iutlan.xmen.model

import ...

open class XMen(
    @PrimaryKey var id: ObjectId? = null,
    var nom: String? = null,

    @DrawableRes var idImage: Int? = null
) : RealmObject()
```

☛ Ajoutez les champs manquants : `alias`, `description` et `pouvoirs`.

☛ Ajoutez la méthode de classe `create` et les noms des champs dans un `companion object`. Relire la page 12. Attention, il faut aussi appliquer l'annotation `@DrawableRes` sur le paramètre `idImage` de la méthode `create`.


4.3. Initialisation des données

Les données sont dans le fichier `res/values/arrays.xml`. Ce fichier possède cette structure :

```
<TYPE-array name="IDENTIFIANT">
  <item>valeur 0</item>
  <item>valeur 1</item>
  <item>valeur 2</item>
  ...
</TYPE-array>
```


On veut en faire des n-uplets dans Realm. Le principe est d'accéder aux ressources par programme. L'API Android propose une classe gestionnaire pour ça, **Resources**. On en demande une instance au système et ensuite, ses méthodes permettent de récupérer les valeurs des ressources, sous forme de tableaux. Les méthodes sont de la forme `getTYPEArray(R.array.IDENTIFIANT)`, sauf pour le tableau des images, voir ci-dessous.

👉 Affichez le code source de `XMenApplication`.

👉 Voici la méthode pour initialiser la liste à partir des ressources. C'est à mettre après `onCreate` et à compléter : 

```
fun initXMens(realm: Realm) {
    realm.executeTransactionAsync {
        // accès aux ressources
        val res = getResources()
        val noms = res.getStringArray(R.array.noms)

        val images = res.obtainTypedArray(R.array.idimages)

        // recopier les données dans la base Realm
        for (i in 0..noms.size - 1) {
            // constructeur avec tous les paramètres
            XMens.create(
                realm = it,
                nom = noms[i],

                idImage = images.getResourceId(i, R.drawable.undef)
            )
        }
        // libérer les images (obligation Android)
        images.recycle()
    }
}
```

Remarquez la particularité pour les images et le fait que son tableau dans `arrays.xml` est différent des autres.

👉 Faites appeler cette méthode `initXMens` par la méthode `onCreate` de `XMenApplication`.

👉 Il manque quelque chose à cette méthode. Réfléchissez à ce qui va se passer si on lance plusieurs fois l'application. Cette méthode est appelée à chaque lancement... Relisez les explications sur Realm.

5. Affichage de la liste


5.1. Activité MainActivity

La première activité s'appelle `MainActivity`. Son but est d'afficher la liste des personnages.

👉 Redéfinissez le layout de cette activité, `activity_main.xml` en ceci : 

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/recycler"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    app:layoutManager="androidx.recyclerview.widget.LinearLayoutManager"
    tools:context=".MainActivity" />
```

Ce layout ne contient qu'une vue qui prend tout l'écran, un `RecyclerView` identifié par `recycler`. C'est une vue spécialisée dans l'affichage en liste défilante de nombreux éléments. Vous allez comprendre peu à peu comment ça fonctionne.

👉 Il faut maintenant mettre en place ce layout dans l'activité à l'aide d'un *View Binding* comme dans le TP4. Modifiez le chargement de l'interface par l'activité avant l'initialisation de `realm` : 

```
class MainActivity : AppCompatActivity() {

    // interface utilisateur
    private lateinit var ui: ActivityMainBinding

    // données
    private lateinit var realm: Realm

    override fun onCreate(savedInstanceState: Bundle?) {
        // initialisation interne de l'activité
        super.onCreate(savedInstanceState)
        // TODO mise en place du layout par un view binding, voir TP4

        // TODO obtention de realm
    }
}
```

👉 Enlevez le mot `TODO` des commentaires une fois que c'est fait.

On complètera cette méthode ultérieurement.

6. RecyclerView pour afficher une liste

Il faut maintenant configurer le `RecyclerView`. Android met en œuvre le patron MVC. La figure 2 montre la situation générale.

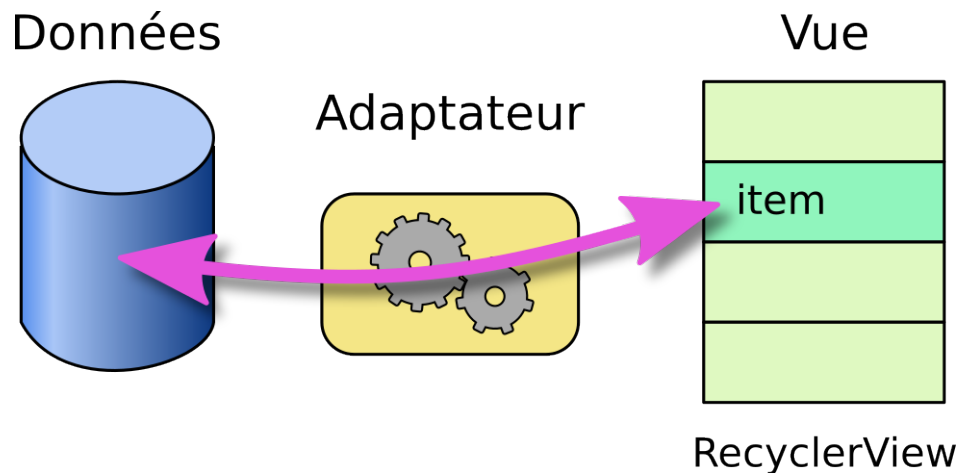


Figure 2: Vue, adaptateur et données

Dans ce schéma, il y a :

- des données (le modèle) et ici elles sont dans la base Realm. On va les extraire sous la forme d'un `RealmResults<XMen>`.
- une vue qui affiche un certain nombre d'éléments. Les éléments sont tous affichés de la même manière. Par exemple dans ce TP, ils ont tous une image et un nom.

Tous les éléments de la liste ne peuvent pas être affichés simultanément, seulement une partie qui dépend de la hauteur d'affichage des éléments et de la hauteur disponible. Quand on fait défiler la vue, certains éléments deviennent visibles et en même temps, d'autres deviennent invisibles. La vue est appelée *Recycler View* parce qu'elle recycle les éléments devenus invisibles pour y mettre ceux qui deviennent visibles. C'est exactement comme un escalator dont les marches sont recyclées par dessous après un passage en tant qu'escalier.

Le *recycler* réagit donc aux touches-glissées sur l'écran, en décalant les éléments dans le sens voulu, et les éléments qui disparaissent sont renvoyés de l'autre côté en changeant ce qui est affiché pour correspondre au nouvel élément devenu visible.

Comme les données peuvent être de toutes natures, listes, bases de données, requêtes REST sur un serveur... Il est nécessaire d'avoir une sorte d'interface entre le *recycler* et les données : un **adaptateur**. Cet objet a pour but de construire un *layout d'item* pour afficher l'un des éléments de la liste. Il y a des adaptateurs pour des listes, des adaptateurs pour des bases de données, des adaptateurs pour des API Rest, etc. Et on retrouve ce concept d'adaptateur pour d'autres types de vues, comme les onglets.

En simplifiant les relations entre ces trois objets, on peut dire que le *recycler view* demande un *layout d'item* à l'adaptateur pour afficher l'élément n°N. L'adaptateur demande à la liste de lui retourner l'élément en question et il en fait une vue qu'il retourne au *recycler*. La figure 3 montre le processus.

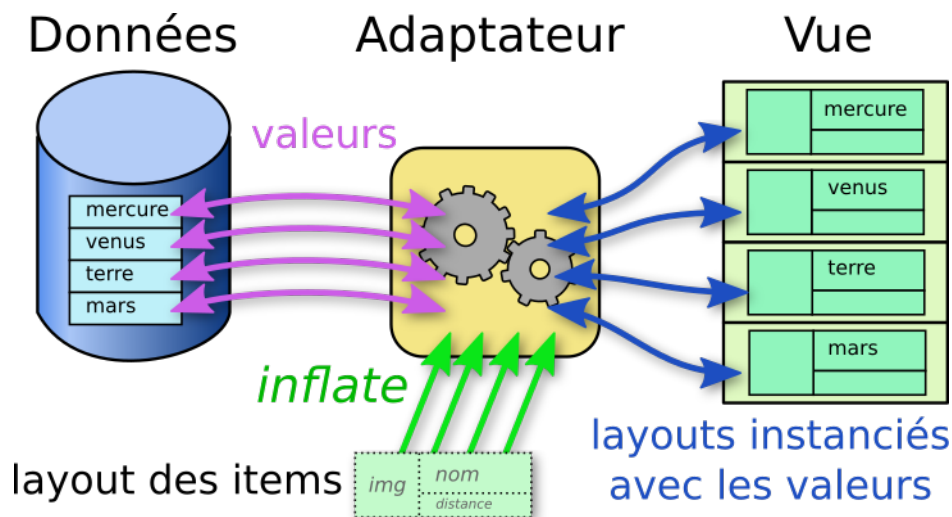


Figure 3: Adaptateur entre les données et la vue

On prend l'exemple d'une liste contenant des planètes. Le *recycler* a besoin d'afficher les 4 premières. Elle s'adresse à l'adaptateur. L'adaptateur récupère chaque planète auprès de la liste. Chacune est ensuite affichée à l'aide d'un *layout d'item* qui est retourné au *recycler*. Ce *layout* contient des vues (TextView, ImageView...) qui permettent d'afficher un élément de la liste (un par *layout*). Il est créé et rempli par l'adaptateur.

Ces *layouts d'items* sont nommés *View Holders*. Attention à ne pas confondre, même si c'est très proche, des *view holders* et des *view bindings*. Le premier est pour stocker et mettre à jour les vues liées à un élément précis de la liste, tandis que le second est seulement pour stocker les vues de toute l'interface. Les *view holders* sont des *view bindings* avec des fonctionnalités supplémentaires.

Nous allons maintenant définir ces *view holders*, puis un adaptateur et enfin configurer le *recycler*.

7. Création d'un *View Holder* pour les éléments

Chaque XMen doit être affichable sur l'écran. Il faut un *layout* associé à une classe. Voici les grandes étapes.

7.1. *Layout* d'élément

👉 Créez un layout appelé `x_men.xml` et contenant les vues nécessaires pour l'affichage : des TextView pour les textes et un ImageView pour l'image.

RQ: si vous nommez ce layout `xmen.xml`, alors le View Binding sera appelé `XmenBinding` au lieu de `XMenBinding`.

Important : pour une première version, contentez-vous de mettre seulement le nom et l'image dans un `LinearLayout` horizontal. Quand tout marchera, vous améliorerez ce layout en rajoutant les vues manquantes.

Un gros piège dans lequel il ne faut pas tomber, c'est de mettre une hauteur de `match_parent` au layout entier. Il remplirait tout l'écran et vous ne verriez qu'un seul élément à la fois.

7.2. Classe *ViewHolder*

☛ Créez une classe appelée `XMenViewHolder` avec ceci :



```
class XMenViewHolder(val ui: XMenBinding) : RecyclerView.ViewHolder(ui.root) {  
  
    var xmen: XMen?  
        get() = null  
        set(xmen) {  
            if (xmen == null) return  
            ui.image.setImageResource(xmen.idImage!!)  
            ui.nom.text = xmen.nom  
        }  
}
```

Que fait-elle ?

- D'abord son constructeur reçoit un `XMenBinding` dont la racine (`ui.root`) est envoyée à sa superclasse. Ce `ui` contient les vues, `ImageView` et `TextView`, que vous avez définies dans `x_men.xml` et qui affichent le `XMen` dans la future liste.
- Ensuite, la propriété `xmen` est un setter qui permet d'afficher les informations d'un `XMen` dans ces vues : ici, le nom et l'image.

Cette classe `XMenViewHolder` permet donc d'afficher l'un des `XMen`. Quelqu'un doit juste instancier cette classe et utiliser son *setter*, et ce quelqu'un, c'est l'adaptateur.

8. Création d'un adaptateur

Le `RecyclerView` pose la question : « que dois-je afficher à telle position ? ». L'adaptateur y répond avec un `XMenViewHolder` rempli avec les informations de l'élément concerné.

☛ Créez une classe `XMenAdapter` comme ceci :



```
class XMenAdapter(val xmens: RealmResults<XMen>) :  
    RecyclerView.Adapter<XMenViewHolder>()
```

Cette classe hérite de `RecyclerView.Adapter`, et elle doit travailler sur des `XMenViewHolder`. Quand on créera une instance de `XMenAdapter`, on devra fournir la liste des `XMen`. L'adaptateur mémorise cette liste dans le champ (variable membre) `xmens`.

☛ Ajoutez les 3 imports manquants (ALT-entrée puis `import`)

Il reste une erreur car la superclasse `RecyclerView.Adapter` est abstraite. Elle demande qu'on implémente 3 méthodes.

☛ Avec l'assistant d'Android Studio, faites créer les trois méthodes manquantes : mettez le curseur sur `XMenAdapter` et tapez ALT-Entrée, puis choisissez `Implement members`, cliquez OK dans le dialogue qui affiche les méthodes en question.


Il faut maintenant programmer les trois méthodes.

8.1. Méthode `getItemCount`

Elle doit retourner le nombre d'éléments de la liste. Ça permet au *recycler* de savoir combien d'éléments il doit afficher.

👉 La liste des XMens est dans la propriété `xmens` passée au constructeur de la classe. C'est un résultat de requête. On peut donc compter les n-uplets. Relisez les explications sur Realm pour savoir comment faire.

8.2. Méthode `onCreateViewHolder`

Cette méthode est appelée quand le *recycler* commence son travail. La liste vient tout juste de lui être fournie et il commence à afficher les premiers éléments visibles. La méthode doit instancier un nouveau `XMenViewHolder` ainsi : 

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int) : XMenViewHolder {  
    val ui = XMenBinding.inflate(LayoutInflater.from(parent.context), parent, false)  
    return XMenViewHolder(ui)  
}
```

8.3. Méthode `onBindViewHolder`

Lorsque l'utilisateur fait défiler la liste et rend des éléments visibles en même temps que d'autres invisibles, les derniers sont réutilisés pour contenir un autre item de la liste. C'est le rôle de cette méthode, de modifier ce qu'on voit dans un *view holder*. Le paramètre `position` est le numéro du `XMen` qui doit être affiché dans le *view holder*.

👉 Voici son code : 

```
override fun onBindViewHolder(holder: XMenViewHolder, position: Int) {  
    holder.xmen = xmens[position]  
}
```

Voyez comment on accède au *n*-ième élément d'un `RealmResult`, avec une notation de tableau.

👉 Que fait l'instruction `holder.xmen = ...` ? Relisez la définition du *setter* `xmen` de la classe `XMenViewHolder`. Comprenez que les textes et l'image vont devenir ceux du `XMen` n°`position`.

8.4. Réactivité de l'adaptateur

On va être très déçu(e) si on s'arrête là. Il faut se souvenir que les requêtes Realm sont asynchrones. Les données ne sont pas immédiatement disponibles. Par exemple, `getItemCount()` retournera zéro au tout début. Les données vont arriver après quelques instants, mais ça sera après l'initialisation de l'adaptateur.

Ce qu'il faut, c'est mettre à jour l'adaptateur à chaque fois qu'il y a un changement sur la liste `xmens`. Voici comment faire.

👉 Ajoutez ceci dans l'adaptateur : 

```
init {  
    xmens.addChangeListener { _, changeSet ->
```

```
        for (change in changeSet.deletionRanges) {
            notifyItemRangeRemoved(change.startIndex, change.length)
        }
        for (change in changeSet.insertionRanges) {
            notifyItemRangeInserted(change.startIndex, change.length)
        }
        for (change in changeSet.changeRanges) {
            notifyItemRangeChanged(change.startIndex, change.length)
        }
    }
}
```

C'est un bloc d'initialisation, et ces instructions lui permettent de s'enregistrer en tant qu'écouteur pour tous les changements à venir sur la liste.

Ces écouteurs existaient dans une classe appelée `RealmRecyclerViewAdapter`, mais elle a disparu des versions récentes de *Realm*. C'est très dommage.

9. Mise en place de la liste dans MainActivity

Il reste à configurer le `RecyclerView` pour l'affichage. On retourne dans `MainActivity`.

9.1. Initialisation d'un adaptateur

👉 Complétez la méthode `onCreate()` (c'est à mettre à la fin de la méthode) :



```
...

// obtenir la liste des xmens
val xmens = ... liste de tous les XMens, asynchrone ...

// créer l'adaptateur
val adapter = XMenAdapter(xmens)

// fournir l'adaptateur au recycler
ui.recycler.adapter = adapter
```

👉 Est-ce que vos *View Holders* ont tous la même hauteur ? Il faut le signaler au *recycler* afin qu'il optimise les calculs de hauteur :



```
...

// dimensions constantes
ui.recycler.setHasFixedSize(true)
```

9.2. Autres *Layout Managers*

Un *RecyclerView* est capable d'afficher une liste de différentes manières : en liste, en tableau ou en blocs empilés. La mise en page des *View Holders* est réalisée par un gestionnaire de

disposition (*Layout Manager*) lié au *recycler*. Actuellement, il est défini dans le layout de l'activité `activity_main.xml`, en tant qu'attribut du *recycler*. On peut le changer par programme : 

```
...  
  
// layout manager  
val lm: RecyclerView.LayoutManager = LinearLayoutManager(this)  
ui.recycler.layoutManager = lm
```

Consulter les transparents du cours [CM4](#), transparents 35 et suivants (TODO à mettre à jour).

9.3. Ligne de séparation entre items

👉 Ajoutez des lignes de séparation entre les items : 

```
// séparateur  
val dividerItemDecoration = DividerItemDecoration(this,  
    DividerItemDecoration.VERTICAL  
)  
ui.recycler.addItemDecoration(dividerItemDecoration)
```

10. Test de l'application

👉 Testez votre application dans un AVD :

- Exception ? Cherchez dans le **LogCat** ce qui a fait planter : null pointer (vue non trouvée dans un layout), class cast (oubli de l'attribut **name** de l'application dans le manifeste)...
- Écran vide ? Est-ce que le **RecyclerView** est affiché ? qu'y-a-t-il dans **setContentView** de **MainActivity** ? est-ce qu'il a un *LayoutManager* sur le **RecyclerView** ? Est-ce que la liste **xmens** est correctement remplie (mettre un message de log par exemple dans **getItemCount** de l'adaptateur) ? Est-ce qu'il y a bien des écouteurs pour les changements dans l'adaptateur ?
- Un seul élément visible ? Est-ce que la hauteur du layout d'item est minimale ou bien remplit-elle tout l'écran ?

10.1. Compléter et améliorer le layout des items

👉 Vous pouvez maintenant ajouter toutes les vues dans le layout `x_men.xml`. Vous pouvez jouer sur le style et la taille des polices. Attention à la longueur des textes, les descriptions sont très longues alors on peut demander à Android de tronquer le texte à l'affichage.

Exemple : 

```
<TextView  
    android:id="@+id/nom"  
    android:textStyle="bold"  
    android:textSize="16dp"  
    ... />  
  
<TextView
```



```
android:id="@+id/description"  
android:textStyle="normal"  
android:maxLines="2"  
android:ellipsize="end"  
... />
```

👉 Enlevez ou augmentez l'attribut `maxLines` si vous l'avez mis, afin de retrouver des éléments ayant des tailles toutes différentes, puis essayez le `StaggeredGridLayoutManager`, voir le [cours 4](#), transparent 37. Ce dernier pourrait être assez sympa en mode paysage.

11. Travail à rendre

Très important : Ce TP va être repris pour le prochain TP, donc faites attention à ne pas le perdre.

Important : votre projet doit se compiler et se lancer sur un AVD. La note du TP sera zéro si ce n'est pas le cas. Mettez donc en commentaire ce qui ne compile pas. C'est impératif que l'application puisse être lancée même si elle ne fait pas tout ce qui est demandé.

Avec le navigateur de fichiers, descendez dans le dossier `app/src` du projet *XMen*. Cliquez droit sur le dossier `main`, compressez-le au format zip. Déposez l'archive `main.zip` sur Moodle au bon endroit, remise du TP5 sur la page [Moodle R4.A11 Développement Mobile](#).

Rajoutez un fichier appelé exactement `IMPORTANT.txt` dans l'archive, si vous avez rencontré des problèmes techniques durant le TP : plantages, erreurs inexplicables, perte du travail, etc. mais pas les problèmes dus à un manque de travail ou de compréhension. Décrivez exactement ce qui s'est passé. Le correcteur pourra lire ce fichier au moment de la notation et compenser votre note.