MOBG56 - Développement mobile

Base de données

SQLite dans Android

- Ou se trouve le fichier SQLite ?
  - dans le *file system* du téléphone : `/data/data/(applicationName)/databases`
    - Les fichiers sont créés automatiquement quand l'application utilise SQLite
- Comment incorporer SQLite dans le code ?
  - **ROOM** : framework développé par Google, sorti en 2018. Permet d'écrire dans une base de données SQLite avec moins de code, de manière plus sécurisée.

ROOM

- Room est un Object-relational mapping (ORM). Il converti des objets Kotlin en entités enregistrables dans une base de données.

Classe Kotlin

```
class Customer(customerFirstName : String,
               customerLastName : String,
               customerAge : Int,
               customerId : Int) {

    val firstName : String = customerFirstName
    val lastName : String = customerLastName
    val age : Int = customerAge
    val id : Int = customerId
}
```

Table SQL

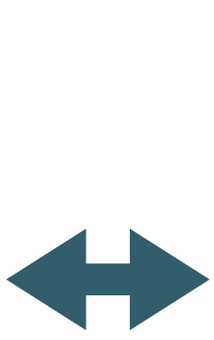

Table: Customer			
Id	Prénom	Nom	Age
1	Bob	Johnes	33
2	Alice	Jenkins	21
3	Alan	Smithee	27
4	Matt	Duffer	45
5	Millie	Brown	7
6	Tom	Holland	41

ROOM

- Comment indiquer à Room la correspondance entre les champs de la classe et les colonnes de la base de données ?
  - Grâce à des **annotations Kotlin** spécifiques à Room

```
@Entity(tableName = "CustomerTable")
class Customer(customerFirstName : String,customerLastName : String,
               customerAge : Int,customerId : Int) {

    @ColumnInfo(name = "Prénom")
    val firstName : String = customerFirstName
    @ColumnInfo(name = "Nom")
    val lastName : String = customerLastName
    @ColumnInfo(name = "age")
    val age : Int = customerAge
    @PrimaryKey
    val id : Int = customerId
}
```

Annotations Kotlin/Java

- Meta-data ajoutées à certaines parties de code
- Peut être utilisé pour ajouter des informations
  - Déclaration de classe / interface
  - Déclaration de méthode
    - Déclaration de champ de méthode
- Les annotations commencent toujours par le signe “@“
- Les annotations peuvent être lues
  - par le compilateur
  - dans le code Kotlin, à l’exécution (utilise la Réflexion Kotlin)
- Ne modifie pas le code (contrairement aux macro en C/C++) !

```
@Entity(tableName = "CustomerTable")
class Customer(customerFirstName : String
               customerLastName : String,
               customerAge : Int,
               customerId : Int) {

    @ColumnInfo(name = "Prénom")
    val firstName : String = customerFirs
    @ColumnInfo(name = "Nom")
    val lastName : String = customerLastN
    @ColumnInfo(name = "age")
    val age : Int = customerAge
    @PrimaryKey
    val id : Int = customerId
}
```

Annotations Kotlin/Java

- Exemple : en Java, le mot clé *override* n'existe pas. On utilise donc une annotation **@Override** pour signifier que la méthode est une redéfinition.

(Java)

```java
@Override
protected void onStart() {
    super.onStart();

}
```

- **@Override** est lu par le compilateur Java.
  - Vérifie si la méthode existe bien dans la classe de base
- Peut contenir des propriétés
  - Syntaxe : **@Annotation (property = "value")**
  - Syntaxe raccourcie s'il n'y a qu'une seule propriété : **@Annotation ("value")**

Room ETAPE 1 : @Entity

- **Room étape 1** : définir une classe entité
  - La classe doit commencer avec l'annotation **@Entity**, suivi du nom de la table SQL
  - Chaque champ peut être annoté avec le nom équivalent de la colonne dans la DB
  - Un champ doit être désigné comme clé primaire et être annoté @PrimaryKey

```
@Entity(tableName = "CustomerTable")
class Customer(customerFirstName : String,customerLastName : String,
                    customerAge : Int,customerId : Int) {

    @ColumnInfo(name = "Prénom")
    val firstName : String = customerFirstName
    @ColumnInfo(name = "Nom")
    val lastName : String = customerLastName
    @ColumnInfo(name = "age")
    val age : Int = customerAge
    @PrimaryKey
    val id : Int = customerId
}
```

ROOM ETAPE 2 : @Dao

- **Room étape 2** : définir une interface *Data Access Objects* (DAO)
- L'interface DAO permet de définir les méthodes qui exécuteront des opérations SQL sur la base de données (SELECT, INSERT, DELETE … etc)
- L'interface DAO doit commencer par l'annotation **@Dao**

```
@Dao
interface CustomerDao {

    @Query("SELECT * FROM CustomerTable WHERE age < 20")
    fun getYoungCustomers() : List<Customer>

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    fun addNewCustomer(customer : Customer)

    @Update
    fun updateCustomer(customer : Customer)

    @Delete
    fun deleteCustomer(customerToDelete : Customer)
}
```

ROOM ETAPE 3 : @Database

- **Room étape 3** : définir une classe abstraite *Database*
- La classe doit être annotée @Database
- La classe doit être dérivée de RoomDatabase
- La classe doit déclarer une méthode abstraite qui retourne le DAO définit précédemment

```
@Database(entities = [Customer::class], version = 1, exportSchema = false)
abstract class  CustomerDatabase() : RoomDatabase() {
    abstract fun customerDao(): CustomerDao
}
```

- Pour créer la base de données :

```
val dbBuilder = Room.databaseBuilder(context,
        CustomerDatabase::class.java,DATABASE_NAME)

val customerDatabase : CustomerDatabase= dbBuilder.build()
```

ROOM : utilisation

- On peut maintenant utiliser le DAO pour lire/écrire la base de données

- Exemples :

```
val customerDao : CustomerDao = database.customerDao()
...
// get a list of young customers
val yougCustomers : List<Customer> = customerDao.getYoungCustomers()
...
// insert a new customer
val newCustomer = Customer("John","Smith",35,124547)
customerDao.addNewCustomer(newCustomer)
```