TD : Dessiner et animer un élément graphique

Temps : **1h./4h.**

Dificulté : ∗

Héritage, constructeur

Il s'agit de créer une vue personnalisée puis de l'ajouter dans la vue principale.

1. Créez un nouveau projet, nommez-le *CustomView*.
2. Choisissez Empty Activity comme modèle d'activité.
3. Laissez le nom de MainActivity pour cette **Activity** principale.
4. Ajoutez une classe Kotlin [4], nommez-la CustomView.
5. Faites hériter CustomView de l'objet **View** situé dans le package android.view.View :

```
class CustomView : View
```

6. Implémentez le premier constructeur :

```
constructor(context: Context) : super(context)
```

7. Implémentez la fonction *onDraw()* afin d'y dessiner un cercle :

```
override fun onDraw(canvas: Canvas?) {
    super.onDraw(canvas)
    canvas?.drawCircle(50F, 50F, 40F, Paint())
}
```

   Remarquez la condition implicite, réalisée via le symbole ? [7]. Remarquez aussi la déclaration explicite de nombre de type Float via F [11].

8. Ajoutez cet élément graphique dans la vue du MainActivity, dans le fichier activity_main :

```
<com.chillcoding.customview.CustomView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Remarquez la déclaration de l'objet CustomView dans le fichier XML via le nom de package (ici « com.chillcoding.customview »). Dans quel package votre objet CustomView est-il situé ?

9. Exécutez, que se passe t-il ?
10. Dans la classe CustomView, ajoutez le constructeur manquant :

```
constructor(context: Context?, attrs: AttributeSet?) : super(context, attrs)
```

À présent, un cercle noir s'affiche à l'écran. Un premier changement est de modifier la couleur du cercle.

Variable d'instance, attribut de classe

1. Dans la classe CustomView, créez une variable d'instance, également appelé attribut de classe mutable, de type **Paint** [8] :

```
private var mPaint = Paint()
```

2. Initialisez la variable *mPaint* avec la couleur bleu, cela dans une fonction appelé *init()*

```
private fun init() {
    mPaint.color = Color.BLUE
}
```

Remarquez l'absence d'accesseur (*getter/setter*) sur la variable *mPaint* [6].

3. Appelez la fonction *init()* dans le deuxième constructeur et faites que le premier constructeur appelle le deuxième :

```
constructor(context: Context) : this(context, null)

constructor(context: Context?, attrs: AttributeSet?) : super(context, attrs) {
    init()
}
```

4. Remplacez le code dans la fonction *onDraw()* afin d'utiliser *mPaint* :

```
canvas?.drawCircle(50F, 50F, 40F, mPaint)
```

5. Exécutez votre application.

À présent, un cercle bleu s'affiche à l'écran. Le prochain développement est d'externaliser les coordonnées du cercle.

Classe de donnée (data class)

Maintenant, il s'agit de créer une classe de donnée, dans un nouveau fichier, pour représenter le cercle à afficher [2][9]. Le cercle possède un diamètre (*radius*) et des coordonnées pour son centre (*cx* et *cy*).

1. Créez une nouvelle classe Kotlin, appelez-là MagicCircle :

```
data class MagicCircle(val cx: Float = 50F, val cy: Float = 50F, val rad: Float = 40F)
```

   Ici, il est déclaré trois attributs de classe constants, dits immuables.

2. Créez une instance de MagicCircle (c'est-à-dire une variable d'instance de type MagicCircle) dans CustomView :

```
var mCircle = MagicCircle()
```

3. Remplacez le code dans la fonction *onDraw()* afin d'utiliser *mCircle* :

```
canvas?.drawCircle(mCircle.cx, mCircle.cy, mCircle.rad, mPaint)
```

4. Exécutez votre application.

Fonction, attribut immuable ou mutable

À présent, il s'agit de déplacer le cercle à l'écran, pour cela il est habituel d'utiliser une fonction [10].

1. Ajoutez une fonction *move()* dans MagicCircle :

```
fun move() {
    cx++
    cy++
}
```

2. Quelle erreur est indiquée par **Android Studio** ?
3. Modifiez les attributs de classe concernés, immuables en mutables, et sortez les du constructeur :

```
var cx: Float = 50F
var cy: Float = 50F
```

4. Placez l'appel de la fonction *move()* et *invalidate()* (cette dernière indique à la vue de se redessiner) à l'endroit où le cercle est dessiné :

```
mCircle.move()
canvas?.drawCircle(mCircle.cx, mCircle.cy, mCircle.rad, mPaint)
invalidate()
```

5. Exécutez votre application.

Le prochain développement consiste à gérer la vitesse de déplacement du cercle.

Constante de classe

L'objet compagnon d'une classe Kotlin sert à stocker toutes le constantes de classe, il est équivalant à *public static final int DELTA = 8* en Java. Dans cette partie, il s'agit de créer un constante représentant une quantité de déplacement, le delta.

1. Ajoutez une constante de classe dans CustomView, afin de gérer la quantité de déplacement dans cette dernière. Cela, au même niveau que la déclaration des attributs de classe :

```
companion object {
    val DELTA = 8
}
```

2. Modifiez la fonction *move()* afin d'utiliser cette constance :

```
    cx += CustomView.DELTA
    cy += CustomView.DELTA
```

3. Que se passe t-il à l'exécution de l'application ?

Condition when

Dans cette partie, il s'agit de modifier la classe de donnée MagicCircle afin que le cercle reste dans l'écran.

1. Modifiez MagicCircle de façon à ce que son constructeur prenne deux attributs de classe immuable :

```
data class MagicCircle(val maxX: Int, val maxY: Int) {
```

2. Déplacez l'attribut de classe *rad* dans le corps de la classe.
3. Ajoutez les deux attributs de classe mutables, suivants dans MagicCircle :

```
var dx = CustomView.DELTA
var dy = CustomView.DELTA
```

   dx et *dy* intègrent le sens de déplacement du cercle en plus de la vitesse

4. Modifiez la fonction *move()* afin que les coordonnées du cercle restent dans le carcan :

```
fun move() {
    when {
            cx !in 0..maxX -> dx = -dx
            cy !in 0..maxY -> dy = -dy
        }
    cx += dx
    cy += dy
}
```

   Remarquez également l'utilisation de la condition *when* comme celle d'un *switch/case* évolué [5].

5. Modifiez l'initialisation de *mCircle* :

```
var mCircle = MagicCircle(100, 100)
```

6. Que se passe t-il à l'exécution de l'application ?

Délégué lateinit

À présent il s'agit de modifier l'initialisation de *mCircle* dans CustomView afin qu'elle prennent en compte les dimensions de l'écran. En effet, le constructeur de MagicCircle nécessite la largeur et la hauteur de la vue CustomView. Hors, la largeur et la hauteur de la vue sont connues depuis la fonction mère, **View**, *onLayout()*.
mCircle ne peut pas être initialisé au départ parce qu'on ne connait pas les valeurs *maxX* et *maxY* donc *mCircle* est initialisé dans l'implementation de cette fonction *onLayout()*(c'est-à-dire plus tard)[3].

1. Modifiez la déclaration de *mCircle* :

```
lateinit var mCircle: MagicCircle
```

Remarquez l'utilisation du mot clé *lateinit*, aussi dans ce cas la déclaration du type est obligatoire.

2. Initialisez *mCircle* depuis *onLayout()* :

```
override fun onLayout(changed: Boolean, left: Int, top: Int, right: Int, bot: Int) {
    super.onLayout(changed, left, top, right, bot)
    mCircle = MagicCircle(width, height)
}
```

3. Exécutez votre projet.

TP : Gestion des interactions

A. Gestion de tableau

1. Créez un tableau d'objet MagicCircle dans la classe CustomView[?][1].
2. Affichez tous les cercles à l'écran

B. Gestion du touché

1. Interceptez le touché de l'utilisateur dans la classe CustomView.
2. Ajoutez un cercle dans le tableau à chaque touché.
3. Déplacez le cercle à l'endroit touché.

C. Utilisation de *Random*

1. Initialisez aléatoirement la couleur du cercle.
2. Initialisez aléatoirement la position du cercle.

D. Pour aller plus loin

Faites les exercices en ligne de la partie *Koans* [12].

Références du cours Kotlin pour Android

Le thème Kotlin pour Android a pour objectif de mettre en perspective le langage Kotlin avec le langage de programmation orient objet (POO) Java ainsi que de présenter les principaux concepts apportés par ce langage fonctionnel.

Dans un premier temps, la différence de syntaxe du langage Kotlin par rapport au Java est mise en évidence. Cela sur les notions de POO suivantes :

Classe : déclaration, propriété, statique, fonction, héritage

Variable : déclaration, nullité, type, opérateur, condition

Dans un second temps, le langage Kotlin est approfondi par rapport aux nouveaux concepts apportés par la programmation fonctionnel. Notamment, les fonctions lambdas ainsi que les fonctions d'extension sont détaillées.

Enfin, la bibliothèque Anko est mise sous les projecteurs. Cette dernière, écrite en Kotlin, est une surcouche du SDK Android, elle permet de faciliter les développements les plus courants dans une application Android (affichage de message à l'utilisateur, etc.).

A l'issue de ce thème, vous serez capable de programmer des applications Android avec le langage Kotlin.

Lien vers la présentation : https://www.slideshare.net/secret/lOuMXlCGZa3VB8

A. Classe

- Déclaration
- Propriété
- Objet compagnon
- Fonction
- Qualificatifs
- Héritage

B. Variable

- Déclaration
- Nullité sécurisée
- Différents types
- Opérateurs
- Conditions
- Boucle

C. Programmation Fonctionnelle

— Fonction d'extension
— Lambdas
— Inline
— Initialisation tardive
— Initialisation avec une classe déléguée

D. Kotlin pour l'Interface Utilisateur

— Configuration de Vue
— Gestion du clic
— Élément Graphique Natif avec la bibliothèque Anko
— Navigation entre Écrans

Webographie

[1] Macha Da Costa. Initialiser un tableau en kotlin. https://www.chillcoding.com/blog/2019/09/26/kotlin-array/, 2019.

[2] JournalDev. Kotlin array. https://www.journaldev.com/17339/kotlin-array, 2018.

[3] Antonio Leiva. Classe de données kotlin. https://antonioleiva.com/data-classes-kotlin/, 2018.

[4] Antonio Leiva. Classe déléguée. https://antonioleiva.com/property-delegation-kotlin/, 2018.

[5] Antonio Leiva. Classe kotlin. https://antonioleiva.com/classes-kotlin/, 2018.

[6] Antonio Leiva. Condition when. https://antonioleiva.com/when-expression-kotlin/, 2018.

[7] Antonio Leiva. Kotlin integration android sdk. https://antonioleiva.com/kotlin-integrations-android-sdk/, 2018.

[8] Antonio Leiva. Nullité kotlin. https://antonioleiva.com/nullity-kotlin/, 2018.

[9] Antonio Leiva. Variable kotlin. https://antonioleiva.com/variables-kotlin/, 2018.

[10] Kotlin Offical. Classe de données kotlin. https://antonioleiva.com/data-classes-kotlin/, 2018.

[11] Kotlin Offical. Fonctions. kotlinlang.org/docs/reference/functions.html, 2018.

[12] Kotlin Offical. Type de base kotlin. kotlinlang.org/docs/reference/basic-types.html, 2018.

[13] Kotlin Official. Try kotlin. https://try.kotlinlang.org/#/KotlinKoans/, 2018.