Jour2

Kotlin

Structure d’un projet Android

Gradle et build.gradle

Dans un projet Android, Gradle est un système de construction qui permet de gérer les dépendances et les tâches de compilation.

Le fichier `build.gradle` est le fichier de configuration de Gradle pour un module du projet.

Configuration du projet avec Gradle

Pour configurer un projet avec Gradle, vous devez comprendre les fichiers `build.gradle`.

Le fichier `build.gradle` pour le projet contient des informations telles que le nom du projet, le groupe, la version, les dépendances et les plugins.

Gestion des dépendances

La gestion des dépendances dans Android est effectuée par l’intermédiaire de Gradle.

Les dépendances sont des bibliothèques tierces utilisées dans votre application.

```
dependencies {
    implementation 'com.google.android.material:material:1.3.0'
}
```

Personnalisation du processus de build

Le processus de build peut être personnalisé en ajoutant des plugins et des tâches à votre fichier `build.gradle`.

Il est possible de compiler avec des options personnalisées pour les projets, les modules et les tâches.

Organisation des fichiers sources Java/Kotlin

Les fichiers sources Java/Kotlin se trouvent dans le dossier `src/main/java`, qui contient généralement un package correspondant au nom de l'application.

Dans ce dossier, on peut trouver plusieurs fichiers Java/Kotlin, comme `MainActivity`, qui définit la première activité qui sera lancée lorsque l'application est ouverte.

Gestion des ressources

Les ressources sont stockées dans le dossier `res` de l'application.

Les images, les sons ou les vidéos sont stockés dans le dossier `res/raw`, tandis que les images utilisées pour l'interface utilisateur sont stockées dans le dossier `res/drawable`.

Ressources XML

Les ressources XML sont utilisées pour la gestion de l'interface utilisateur, la définition de styles, de couleurs, ou la définition de l'organisation des éléments de l'interface utilisateur dans des fichiers de layout.

Ces fichiers XML de ressource sont stockés dans le dossier `res/layout`.

Rôle et structure de base

Le rôle principal du fichier AndroidManifest.xml est de déclarer l'ensemble des informations sur l'application que le système d'exploitation Android doit connaître avant de l'exécuter.

Déclaration des activités

La section `<application>` définit les composants de l'application, tels que les activités, les services et les récepteurs de diffusion.

Dans l'exemple ci-dessus, une activité nommée `MainActivity` est définie.

La balise `<activity>` définit une activité, qui est généralement l'interface utilisateur principale de l'application.

L'attribut `android:name` spécifie le nom complet de la classe qui implémente l'activité.

Gestion des permissions

La section `<uses-permission>` spécifie les autorisations requises par l'application pour accéder aux fonctionnalités du système, telles que l'accès au stockage de l'appareil.

Spécification des configurations matérielles et logicielles requises

La section `<uses-sdk>` spécifie les versions du SDK Android nécessaires pour que l'application fonctionne correctement.

Débogage avec Android Studio

Android Studio est un IDE qui dispose de nombreuses fonctionnalités de débogage intégrées.

En utilisant ces fonctionnalités, vous pouvez facilement déboguer votre application Android.

Utilisation du débogueur intégré

Le débogueur intégré d'Android Studio est une fonctionnalité qui permet de mettre en pause le code, de contrôler son exécution pas à pas, d'inspecter les variables et de parcourir la pile d'appels.

Points d'arrêt et exécution pas à pas

Les points d'arrêt sont utilisés pour mettre en pause votre code à un point spécifique.

En utilisant les points d'arrêt, vous pouvez inspecter les variables, parcourir la pile d'appels et exécuter votre code pas à pas.

Inspection des variables et de la pile d’appels

En inspectant les variables, vous pouvez voir les valeurs des variables à un point donné du code.

En parcourant la pile d’appels, vous pouvez suivre l’exécution de votre code et en comprendre la logique.

Bases du développement Android

Concepts et classes de base

Les concepts et les classes de base sont les éléments essentiels du développement Android.

Cela inclut les activités et les fragments.

Activités et fragments

Les activités et les fragments sont des composants clés de l’interface utilisateur d’une application Android.

Activités

Une activité est une classe qui représente une fenêtre de l’application.

Elle est responsable de la gestion des interactions utilisateur, ainsi que de l’affichage des informations sur l’écran.

Définition et rôle

L’activité est la composante responsable de l’interaction utilisateur.

Elle définit la fenêtre de l’application et gère les événements tels que les clics, les appuis longs, les appels, etc.

Cycle de vie d’une activité

Chaque activité suit un cycle de vie, qui comprend les états de création, de démarrage, de mise en pause, de reprise et de destruction.

Les méthodes principales du cycle de vie sont :

- `onCreate()` : Cette méthode est appelée lorsque l'activité est créée pour la première fois.
- `onStart()` : Cette méthode est appelée lorsque l'activité est visible à l'écran.
- `onPause()` : Cette méthode est appelée lorsque l'activité n'est plus visible à l'écran.
- `onResume()` : Cette méthode est appelée lorsque l'activité reprend après avoir été en pause.
- `onDestroy()` : Cette méthode est appelée lorsque l'activité est en cours de destruction.

Création d'une nouvelle activité

Pour créer une nouvelle activité, vous devez créer une nouvelle classe qui étend la classe `Activity`.

Ensuite, vous devez définir le contenu de l'activité en ajoutant des éléments d'interface utilisateur, tels que des boutons, des champs de texte, etc.

Navigation entre activités

Pour naviguer entre les activités, vous pouvez utiliser la méthode `startActivity()` pour lancer une nouvelle activité.

Vous pouvez également spécifier des données qui seront transmises à l'activité à l'aide d'un objet `Intent`.

Fragments

Les fragments sont une partie essentielle de l'interface utilisateur d'une application Android.

Ils permettent de créer une interface utilisateur flexible qui peut être adaptée à différentes tailles d'écran et orientations.

Les développeurs peuvent les utiliser pour fournir des éléments de navigation, des zones de contenu et des boîtes de dialogue.

Définition et rôle

Un fragment est un composant réutilisable qui peut être combiné avec d'autres fragments pour créer une interface utilisateur flexible et modulaire.

Les fragments sont utilisés pour fournir une interface utilisateur pour une application Android et sont souvent utilisés pour fournir des éléments de navigation, des zones de contenu et des boîtes de dialogue dans l'application.

Cycle de vie d'un fragment

Un fragment suit un cycle de vie similaire à celui d'une activité.

Le cycle de vie d'un fragment comprend les étapes suivantes :

```
1. Attachement : le fragment est attaché à une activité parent.
2. Création : le fragment est créé.
3. Vue : la vue du fragment est créée.
4. Activité créée : la méthode onCreate de l'activité parent est appelée.
5. Démarrage : le fragment est démarré.
6. Activité démarrée : la méthode onStart de l'activité parent est appelée.
7. Affichage : le fragment est affiché à l'utilisateur.
8. Activité affichée : la méthode onResume de l'activité parent est appelée.
9. Pause : le fragment est mis en pause.
10. Activité mise en pause : la méthode onPause de l'activité parent est appelée.
11. Arrêt : le fragment est arrêté.
12. Activité arrêtée : la méthode onStop de l'activité parent est appelée.
13. Destruction : le fragment est détruit.
14. Activité détruite : la méthode onDestroy de l'activité parent est appelée.
```

Création d'un nouveau fragment

Pour créer un nouveau fragment, vous devez suivre les étapes suivantes :

1. Créez une nouvelle classe Java ou Kotlin qui étend la classe Fragment.
2. Implémentez la méthode onCreateView pour créer la vue du fragment.
3. Utilisez la méthode getActivity pour obtenir l'activité parent du fragment.

Navigation entre fragments

Pour naviguer entre plusieurs fragments, vous devez utiliser la classe FragmentManager.

Le FragmentManager est responsable de la gestion de tous les fragments de l'application et vous pouvez l'utiliser pour ajouter, remplacer et supprimer des fragments.

Vous pouvez également utiliser le FragmentManager pour créer des transactions de fragment et les ajouter à la pile de fragments de l'application.

Intent

Les Intents sont un mécanisme important dans le développement Android.

Ils permettent de communiquer entre les différentes parties de l'application et avec d'autres applications.

Un Intent est une description abstraite d'une opération à effectuer, elle spécifie l'action à effectuer.

Définition et rôle

Un Intent est un objet qui encapsule une requête à une autre partie du système Android.

Ces requêtes peuvent être des demandes d'informations ou de services, ou encore des demandes d'exécution d'une activité.

Les Intents peuvent aussi servir à transférer des données entre différentes composantes de l'application.

Types d'intent

Il existe deux types d'Intents : explicites et implicites.

Les Intents explicites sont utilisés pour lancer une activité spécifique de l'application en utilisant son nom de classe spécifique.

Les Intents implicites sont utilisés pour lancer une activité basée sur une action et une catégorie.

Utilisation d’intents pour la navigation

Lorsqu’on utilise un Intent pour naviguer dans une application, il est important de spécifier l’activité de destination (explicitement ou implicitement) pour que l’utilisateur puisse se retrouver facilement.

Par conséquent, tous les Intents doivent spécifier l’activité cible.

Utilisation d’intents pour les actions

Les Intents impliquent souvent des déclencheurs d’action.

Il peut s’agir de boutons d’application, de liens Web ou même d’autres applications.

Lorsqu’on utilise un Intent pour une action, il est important d’utiliser la méthode startActivity() de la classe Context pour lancer l’activité appropriée.

Services

Les services sont des composants permettant d’exécuter des tâches de manière asynchrone, en arrière-plan et indépendamment de l’interface utilisateur.

Leur rôle est d’assurer la continuité de certaines fonctionnalités, même lorsque l’application n’est pas en cours d’utilisation.

Définition et rôle

Un service est une classe qui étend la classe Service d’Android.

Il peut être démarré de manière programmée ou déclarative par une autre composante (activité, service, récepteur).

Le rôle du service est de fournir des fonctionnalités indépendantes de l’interface utilisateur.

Concrètement, il peut s’agir de la gestion du téléchargement de données en arrière-plan, de la lecture de musique, etc.

Création d’un service

Pour créer un service en Kotlin, il faut :

- Créer une nouvelle classe en étendant la classe Service
- Implémenter les méthodes onStartCommand(), onBind() et onCreate() selon les besoins de l’application
- Ajouter le service à la liste des composants de l’application dans le fichier AndroidManifest.xml

---

Communication entre services et activités

Pour communiquer entre un service et une activité en Kotlin, il est possible d’utiliser :

- L’intent
- Le binder

L’intent permet de transmettre des informations entre le service et l’activité, tandis que le binder est un objet permettant d’implémenter une interface personnalisée permettant la communication entre les deux composants.

Types de services

Il existe deux types de services en Android :

- Les services liés
- Les services de démarrage

Les services liés sont utilisés pour communiquer avec un autre composant de l’application, tandis que les services de démarrage sont utilisés pour exécuter des tâches de manière récurrente en arrière-plan.

BroadcastReceiver

Les `BroadcastReceiver` sont des composants clés du système Android qui permettent à l’application d’écouter et de répondre à des messages envoyés par le système ou d’autres applications.

Ils sont utilisés pour déclencher certaines actions en réponse à des événements tels que la réception d'un appel ou l'état du réseau.

Définition et rôle

Un `BroadcastReceiver` est une classe qui hérite de la classe `android.content.BroadcastReceiver`.

Il doit être enregistré dans le fichier AndroidManifest.xml pour qu'il soit reconnu par le système Android.

Le rôle principal d'un BroadcastReceiver est de répondre à un événement en déclenchant une action.

Création d'un BroadcastReceiver

Pour créer un `BroadcastReceiver`, il faut tout d'abord créer une classe qui hérite de la classe `BroadcastReceiver`.

Ensuite, il faut implémenter la méthode `onReceive()` qui sera appelée lorsque le Broadcast sera envoyé.

Cette méthode prend deux paramètres : le contexte et l'intent du Broadcast.

Enregistrement d'un BroadcastReceiver

Pour que le système Android soit en mesure d'envoyer des Broadcasts à notre `BroadcastReceiver`, nous devons l'enregistrer dans le fichier AndroidManifest.xml.

Pour cela, nous devons utiliser la balise `<receiver>` et spécifier le nom complet de la classe, comme ceci :

```
<receiver android:name=".MyBroadcastReceiver" />
```

Réception et traitement des messages

Lorsqu'un Broadcast est envoyé, le système Android vérifie tous les `BroadcastReceiver` enregistrés pour savoir s'ils doivent y répondre.

Si le `BroadcastReceiver` est enregistré avec le filtre d'intention approprié, la méthode `onReceive()` est appelée.

Les éléments graphiques de base

Vues (Views)

Les vues sont les composants de base de l'interface utilisateur.

Elles sont utilisées pour afficher des informations ou pour interagir avec l'utilisateur.

Types de vues

Quelques exemples de vues :

- TextView : affiche du texte
- ImageView : affiche une image
- Button : permet à l'utilisateur de déclencher une action
- EditText : permet à l'utilisateur de saisir du texte
- RadioButton : permet à l'utilisateur de sélectionner une option parmi plusieurs

Positionnement des éléments sur un écran

Coordonnées X et Y

Les éléments graphiques sont positionnés sur l'écran en utilisant des coordonnées X et Y.

Les coordonnées X définissent la position horizontale de l'élément, tandis que les coordonnées Y définissent sa position verticale.

Par défaut, les éléments graphiques sont positionnés dans le coin supérieur gauche de l'écran.

Marges et espacements

Les marges et les espacements sont utilisés pour définir l'espace autour d'un élément graphique.

Les marges sont l'espace entre l'élément et les bords de la vue parente.

Les espacements, ou padding en anglais, sont l'espace entre les bords de l'élément et son contenu.

Alignement et gravité

L'alignement et la gravité sont utilisés pour positionner les éléments graphiques par rapport aux autres éléments de la vue parente.

L'alignement définit comment les éléments s'alignent les uns par rapport aux autres, tandis que la gravité définit la façon dont les éléments sont alignés par rapport à la vue parente.

ViewGroup et view, les Layout

Les ViewGroups sont des conteneurs qui sont utilisés pour organiser les View.

Les Views sont les éléments graphiques individuels.

Les Layouts sont des types de ViewGroup qui déterminent la disposition des Views.

La façon dont les Views sont disposées affecte la présentation de l'application pour l'utilisateur.

Il existe différents types de Layouts en Kotlin, que nous allons examiner en détails.

LinearLayout

LinearLayout est un Layout qui permet d'organiser les Views en une seule file, horizontalement ou verticalement.

```
<LinearLayout
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
```

```
        android:text="Mon texte"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

RelativeLayout

RelativeLayout est un Layout qui permet de positionner les Views les unes par rapport aux autres.

On peut par exemple ancrer une vue à un bord de l’écran, à une autre vue, etc.

RelativeLayout

```
<RelativeLayout
    android:layout_width="match_parent"
    android:layout_height="wrap_content">

    <TextView
        android:text="Mon texte"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_toLeftOf="@id/mon_bouton"/>

</RelativeLayout>
```

FrameLayout

FrameLayout est un Layout qui permet d’empiler les Views les unes sur les autres.

Cela peut être utilisé pour superposer plusieurs images ou pour afficher un fond d’écran.

FrameLayout

```
<FrameLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">
```

```
    <ImageView
        android:src="@drawable/image1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</FrameLayout>
```

ConstraintLayout

ConstraintLayout est un Layout qui permet de créer des relations entre les Views, en utilisant des contraintes.

Il est très flexible et permet de créer des interfaces utilisateur complexes et personnalisées.

Contraintes

```
<ConstraintLayout
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/texte1"
        android:text="Texte1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintLeft_toLeftOf="parent"/>

</ConstraintLayout>
```

Composants graphiques de base

Les composants graphiques sont essentiels pour la création d’interfaces utilisateur (UI) efficaces et agréables à utiliser.

Les boutons font partie des éléments graphiques les plus couramment utilisés dans les applications Android.

Kotlin dispose de plusieurs types de boutons pour répondre aux exigences spécifiques de chaque projet :

Bouton simple (Button)

Le bouton simple est le type de bouton le plus couramment utilisé.

Il dispose d'un texte défini à l'intérieur, et peut réagir aux événements utilisateur tels que le toucher ou le clic.

Exemple de bouton simple

```
<Button   android:id="@+id/myButton"   android:layout_width="wrap_content"   android:layou
t_height="wrap_content"   android:text="Cliquez sur moi" />
```

Bouton avec image (ImageButton)

Le bouton avec image est similaire au bouton simple, mais il dispose d'une image définie à l'intérieur.

Pour créer un bouton avec image, vous devez utiliser la balise `ImageButton` dans votre layout XML :

```
<ImageButton   android:id="@+id/myImageButton"   android:layout_width="wrap_content"   and
roid:layout_height="wrap_content"   android:src="@drawable/myImage" />
```

Exemple basique :

```
val myImageButton = findViewById<ImageButton>(R.id.myImageButton)myImageButton.setOnClickL
istener {   // Insérez ici le code à exécuter lors du clic sur le bouton}
```

Bouton personnalisé (Custom Button)

Le bouton personnalisé est le type le plus avancé de bouton disponible en Kotlin.

Il permet de créer des boutons avec des formes, des tailles et des fonctionnalités spécifiques à votre application.

Pour créer un bouton personnalisé, vous devez étendre la classe `Button` et ajouter des fonctionnalités supplémentaires :

```
class CustomButton(context: Context) : Button(context) {   init {      // Ajoutez ici les
 modifications que vous souhaitez apporter au bouton   }}
```

Bouton personnalisé (Custom Button)

Vous pouvez ensuite utiliser votre bouton personnalisé dans votre layout XML :

```
<com.example.myapp.CustomButton   android:id="@+id/myCustomButton"   android:layout_width
="wrap_content"   android:layout_height="wrap_content"   android:text="Cliquez sur moi" />
```

Exemple de bouton personnalisé

En Kotlin, vous pouvez facilement référencer et manipuler ce bouton personnalisé à l’aide de son identifiant.

Voici un exemple basique :

```
val myCustomButton = findViewById<CustomButton>(R.id.myCustomButton)myCustomButton.setOnCl
ickListener {   // Insérez ici le code à exécuter lors du clic sur le bouton}
```

Textes

Les textes sont l’un des composants graphiques les plus couramment utilisés dans une application.

Il existe plusieurs types de textes, notamment TextView, EditText et AutoCompleteTextView.

TextView

Le TextView est un composant qui permet d’afficher du texte de manière statique dans une application.

Il est souvent utilisé pour afficher des titres, des sous-titres, des labels ou des messages d'erreur.

Pour personnaliser les propriétés de texte du TextView, vous pouvez utiliser les méthodes setText(), setTextSize(), setTextColor(), setFontFamily(), etc.

EditText

L'EditText est un composant qui permet à l'utilisateur de saisir du texte.

Il est souvent utilisé dans les formulaires ou les champs de recherche.

Pour personnaliser les propriétés de l'EditText, vous pouvez utiliser les méthodes setHint(), setInputType(), setTextSize(), setTextColor(), setBackground(), etc.

AutoCompleteTextView

L'AutoCompleteTextView est un composant qui permet à l'utilisateur de saisir du texte, mais qui fournit également des suggestions automatiques pour compléter le texte.

Il est souvent utilisé lorsque l'application nécessite des entrées spécifiques, telles que des adresses e-mail ou des noms de ville.

Pour personnaliser les propriétés de l'AutoCompleteTextView, vous pouvez utiliser les méthodes setAdapter(), setTextSize(), setTextColor(), setBackground(), etc.

Personnalisation du texte

Il est possible de personnaliser le texte en utilisant les ressources de style prédéfinies dans l'application ou en créant vos propres styles.

Les propriétés de texte personnalisables incluent la taille, la couleur, la police, l'alignement, le soulignement et le gras, entre autres.

Listes

Les listes dans Kotlin peuvent être gérées à l'aide de plusieurs composants tels que ListView ou RecyclerView, en fonction de vos besoins.

ListView

ListView est un composant qui affiche les données sous forme de liste verticale basée sur les objets dans la mémoire.

Pour afficher les éléments dans une ListView, vous devez fournir un adaptateur contenant les données à afficher.

RecyclerView

RecyclerView est un composant qui affiche également les éléments sous forme de liste verticale, mais il est plus flexible que ListView.

Il peut gérer des listes avec de grands ensembles de données et peut également prendre en charge des dispositions personnalisées pour les éléments de liste.

Adapteurs

Les adaptateurs aident à lier les données aux éléments de la liste pour une ListView ou un RecyclerView.

Il peut être personnalisé à l'aide des méthodes de onBindViewHolder, onCreateViewHolder, ou simplement à l'aide d'un objet créé personnalisé qui étend l'Adapter.

Gestion des événements des éléments de liste

Pour gérer les événements tels que les clics sur les éléments de la liste, vous devez configurer un écouteur d'événements sur les éléments de la liste en utilisant la méthode setOnItemClickListener pour une ListView et setOnClickListener pour un RecyclerView.

Dans cet exemple, nous créons une liste de noms de personnages, que nous affichons via une ListView à l'aide d'un adaptateur personnalisé.

Nous définissons également un écouteur d'événements sur la ListView pour gérer les clics sur les éléments de la liste.

Lorsqu'un élément est cliqué, nous affichons une toast avec le nom du personnage sélectionné.

Images

Les images ajoutent une dimension visuelle à une application.

Kotlin permet d’afficher des images avec sa classe ImageView.

Voici comment afficher une image dans une ImageView :

```
val imageView = findViewById<ImageView>(R.id.imageView)
imageView.setImageResource(R.drawable.image)
```

Image personnalisée

Vous pouvez également créer votre propre objet Drawable, qui est un objet qui représente une image.

Pour créer une image personnalisée, placez votre image dans le dossier res/drawable, puis utilisez Drawable.createFromPath() pour créer votre objet Drawable :

```
val drawable = Drawable.createFromPath(filePath)
val imageView = findViewById<ImageView>(R.id.imageView)
imageView.setImageDrawable(drawable)
```

Gestion de la taille et du redimensionnement

La taille de l’image peut être gérée à l’aide des attributs android:layout_width et android:layout_height.

Par exemple, pour définir une largeur de 200dp et une hauteur de 100dp :

```
<ImageView    android:id="@+id/imageView"    android:layout_width="200dp"    android:layou
t_height="100dp" />
```

Cases à cocher (CheckBox)

Une case à cocher est un composant graphique qui permet à l’utilisateur de sélectionner une ou plusieurs options à la fois.

La sélection est indiquée par une coche dans la case.

Pour utiliser une case à cocher dans Kotlin, vous devez d’abord la créer en tant qu’objet CheckBox.

Ensuite, vous pouvez ajouter un écouteur pour détecter les changements d’état de la case à cocher.

Exemple

```
val checkBox = CheckBox("Option 1")
checkBox.selectedProperty().addListener {
observable, oldValue, newValue ->
if (newValue == true) {
  println("La case à cocher est cochée")
}
else
{
  println("La case à cocher est décochée")
}}
```

Boutons radio (RadioButton)

Les boutons radio sont similaires aux cases à cocher, mais ils sont conçus pour permettre à l’utilisateur de sélectionner une seule option parmi plusieurs.

Ils sont souvent utilisés dans les groupes pour représenter des choix mutuellement exclusifs.

Pour créer et utiliser des boutons radio dans Kotlin, vous devez d’abord créer un groupe de boutons radio à l’aide d’un objet RadioGroup.

Ensuite, vous pouvez ajouter des boutons radio individuels à ce groupe.

Gestion des états et des événements

Pour les cases à cocher et les boutons radio, vous pouvez ajouter des écouteurs pour détecter les changements d’état.

Les écouteurs sont des fonctions qui seront déclenchées chaque fois qu’il y aura un changement d’état.

Vous pouvez utiliser ces fonctions pour effectuer des tâches spécifiques liées à l'interface utilisateur.

Dans les exemples ci-dessus, nous avons ajouté des écouteurs pour les cases à cocher et les boutons radio qui impriment simplement une chaîne à la console lorsqu'un changement d'état est détecté.

Les états des composants graphiques peuvent être mis à jour en utilisant leurs propriétés correspondantes.

Par exemple, pour cocher ou décocher une case à cocher, vous pouvez définir sa propriété selected sur true ou false.

Exemple

```
val checkBox = CheckBox("Option 1")checkBox.isSelected = true // cocher la case à cocherch
eckBox.isSelected = false // décocher la case à cocher
```

Gestion des évènements

En Kotlin, la gestion des évènements se fait principalement à travers la mise en place de listeners.

Les listeners permettent de détecter différents types d'évènements et de déclencher des traitements en conséquence.

Interaction avec les composants

Dans la gestion de l'interface graphique, il est important de comprendre comment interagir avec les différents composants.

En Kotlin, les composants se manipulent principalement à travers des méthodes.

Par exemple, pour modifier le texte d'un bouton, on appelle la méthode `setText()` sur cet objet.

Les interactions entre l'utilisateur et l'interface graphique peuvent prendre différentes formes, telles que le touch (appui), le drag (glisser) ou encore le zoom (pincer).

Les composants doivent être configurés pour réagir aux interactions de l’utilisateur de manière appropriée.

Exemple basique

Voici un exemple basique d’interaction avec un bouton en Kotlin:

```
val button = findViewById<Button>(R.id.my_button)button.setOnClickListener {    // Traitement à effectuer lorsque le bouton est cliqué}
```

Gestion des clics et des gestes

Les clics et les gestes sont parmi les interactions les plus courantes de l’utilisateur avec l’interface graphique.

OnClickListener

L’interface OnClickListener permet de détecter le clic sur un élément de l’interface graphique, tel qu’un bouton ou une image.

Pour utiliser OnClickListener, vous pouvez ajouter le code suivant à votre activité :

```
bouton.setOnClickListener {    // Le code à exécuter lors du clic sur le bouton}
```

OnLongClickListener

L’interface OnLongClickListener permet de détecter l’appui prolongé sur un élément de l’interface graphique.

Pour utiliser OnLongClickListener, vous pouvez ajouter le code suivant à votre activité :

```
bouton.setOnLongClickListener {    // Le code à exécuter lors de l'appui prolongé sur le bouton    true}
```

GestureDetector

Le GestureDetector permet de détecter les gestes tels que le balayage, le pincement ou le zoom.

Pour utiliser GestureDetector, vous pouvez ajouter le code suivant à votre activité :

```
val gestureDetector = GestureDetector(context, object : GestureDetector.SimpleOnGestureLis
tener() {    override fun onFling(e1: MotionEvent?, e2: MotionEvent?, velocityX: Float, ve
locityY: Float): Boolean {        // Le code à exécuter lors du balayage        return tru
e    }})votre_view.setOnTouchListener { v, event -> gestureDetector.onTouchEvent(event) }
```

Création de vues en XML

Introduction au langage XML

XML (eXtensible Markup Language) est un langage de balisage qui permet de structurer des données de manière hiérarchique.

Une balise XML est composée d’un nom et d’attributs facultatifs qui permettent de définir les propriétés de la balise.

Exemple de balise XML:

```
<balise attribut1="valeur1" attribut2="valeur2"> contenu </balise>
```

Les espaces de noms sont utilisés pour distinguer les balises qui ont le même nom mais qui ont un contexte différent.

Les espaces de noms sont déclarés dans la balise racine du document XML.

Exemple de déclaration d’espace de noms:

```
<root xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

Structure hiérarchique

Les balises peuvent être imbriquées les unes dans les autres pour former une structure hiérarchique.

La balise racine contient toutes les autres balises.

Exemple de structure hiérarchique:

```
<root>
   <baliseEnfant attribut="valeur"> contenu </baliseEnfant>
</root>
```

Utilisation des fichiers XML pour définir des vues

Dans Kotlin, nous avons la possibilité de définir des vues en utilisant des fichiers XML.

Les fichiers XML permettent une plus grande flexibilité dans la création d'interfaces utilisateur.

Les vues XML peuvent être créées à partir de plusieurs éléments de base tels que des layouts, des widgets, des ressources et des valeurs.

Création de layouts

Les layouts sont des éléments de base dans la création de vues XML.

Ils sont utilisés pour organiser les vues de manière cohérente et structurée.

Les layouts peuvent être imbriqués les uns dans les autres pour créer des interfaces utilisateur plus complexes.

Dans Kotlin, nous avons des layouts prédéfinis tels que LinearLayout, RelativeLayout, FrameLayout, etc.

Personnalisation des widgets

Les widgets sont les éléments de base qui composent les vues XML.

Les widgets tels que les boutons, les champs de saisie de texte, les images, etc., peuvent être personnalisés en utilisant des attributs tels que la couleur de fond, la taille, la police, la marge, etc.

Styles et thèmes

En Kotlin, nous pouvons créer des styles et des thèmes pour personnaliser l'apparence de nos vues.

Les styles sont utilisés pour regrouper les attributs communs des widgets et les thèmes sont utilisés pour définir l'apparence globale de l'application.

Les styles et les thèmes sont définis dans des fichiers XML distincts et sont appliqués à l'aide de l'attribut 'style'.

Création de vues par code

Pour créer une vue en Kotlin, la première étape consiste à l'instancier en utilisant le constructeur correspondant.

Par exemple, pour une `TextView`, on peut utiliser le constructeur `TextView(context)` :

```
val textView = TextView(context)
```

On peut ensuite configurer les propriétés de la vue à l'aide des méthodes disponibles.

Par exemple, pour définir le texte de la `TextView`, on peut utiliser la méthode `setText(text: CharSequence)` :

```
textView.setText("Hello world!")
```

Une fois qu'on a créé une vue, on peut l'ajouter à un layout parent en utilisant la méthode `addView(view: View)` :

```
val layout = findViewById<LinearLayout>(R.id.my_layout)layout.addView(textView)
```

Création dynamique de vues

Pour créer des vues dynamiquement, on peut utiliser une boucle pour instancier plusieurs vues avec les mêmes propriétés, puis les ajouter au layout parent.

Par exemple, pour créer une série de `TextView` numérotées de 1 à 5, on peut utiliser le code suivant :

```
val layout = findViewById<LinearLayout>(R.id.my_layout)for (i in 1..5) {    val textView = TextView(context)    textView.setText("TextView #$i")    layout.addView(textView)}
```

Modification des propriétés des vues à l’exécution

Il est possible de modifier les propriétés des vues à l’exécution en utilisant les méthodes associées.

Par exemple, pour modifier la taille d’une vue, vous pouvez utiliser la méthode `setWidth()` ou `setHeight()`.

Pour modifier la couleur de fond d’une vue, vous pouvez utiliser la méthode `setBackgroundColor()`.

Exemple basique :

```
val myButton = Button(this)myButton.text = "Bonjour"myButton.setBackgroundColor(Color.GREEN)myLayout.addView(myButton)
```

Animer les vues

Kotlin dispose également de nombreuses fonctions pour animer les vues.

Par exemple, la méthode `animate()` permet de spécifier une animation.

Il est possible de combiner plusieurs animations pour créer des effets plus complexes.

Exemple basique :

```
val myButton = Button(this)
myButton.text = "Bonjour"
myButton.setBackgroundColor(Color.GREEN)
myButton.animate().translationYBy(100f).setDuration(1000)
myLayout.addView(myButton)
```

Gestion des vues dans le code Java ou Kotlin

La gestion des vues dans le code peut être effectuée en utilisant la méthode findViewById().

Cette méthode permet d’obtenir une instance d’une vue par son identifiant.

---

findViewById()

Voici un exemple de code montrant comment utiliser la méthode findViewById() pour obtenir une instance d’une vue :

```
val textView = findViewById<TextView>(R.id.textView)
```

Dans cet exemple, nous obtenons une instance d’un TextView avec l’identifiant “textView”.

Utilisation de l’API ViewBinding

Une autre méthode pour gérer les vues dans le code est d’utiliser l’API ViewBinding.

Cette API permet de lier les vues avec le code en générant des classes de liaison.

Exemple :

```
private lateinit var binding: ActivityMainBindingoverride

fun onCreate(savedInstanceState: Bundle?) {
  super.onCreate(savedInstanceState)
  binding = ActivityMainBinding.inflate(layoutInflater)
  setContentView(binding.root)
}
```

Extension Kotlin synthétique

Enfin, Kotlin offre une troisième méthode pour gérer les vues dans le code : les extensions Kotlin synthétiques.

Ces extensions permettent d’accéder directement aux éléments du layout XML en utilisant l’identifiant de la vue.

Voici un exemple de code montrant comment utiliser les extensions Kotlin synthétiques :

```
import kotlinx.android.synthetic.main.activity_main.*

class MainActivity : AppCompatActivity() {
  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)
    textView.text = "Hello world!"
  }
}
```

Création de menus d’ActionBar

Pour créer des menus d’ActionBar, nous utilisons `MenuInflater`, qui permet d’inflater un menu spécifique depuis un fichier XML.

Le menu est ensuite attacher à l’ActionBar.

```
//Inflate le menu depuis un fichier XML
menuInflater.inflate(R.menu.mon_menu, menu)

//Attache le menu à l'ActionBar
return true
```

Items de menu

Dans un menu d’ActionBar, chaque élément est un item de menu.

Un item de menu peut être un objet simple comme un bouton ou un objet hiérarchique.

Dans le fichier XML du menu, chaque item est représenté par une balise `<item>`.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <item
    android:id="@+id/mon_item"
    android:title="@string/mon_titre"
```

```
    android:icon="@drawable/mon_icone"
    android:showAsAction="ifRoom"/>
</menu>
```

Groupes et ordre des items

Nous pouvons ajouter des groupes d'items pour séparer des actions liées ou créer des sections dans le menu.

Chaque groupe est défini par une balise `<group>`.

De plus, nous pouvons spécifier l'ordre des items en utilisant l'attribut `android:orderInCategory`.

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
  <group android:id="@+id/group1" android:checkableBehavior="single">
    <item
      android:id="@+id/mon_item_1"
      android:title="@string/mon_titre_1" />
    <item
      android:id="@+id/mon_item_2"
      android:title="@string/mon_titre_2" />
  </group>
  <group android:id="@+id/group2" android:checkableBehavior="none">
    <item
      android:id="@+id/mon_item_3"
      android:title="@string/mon_titre_3" />
  </group>
</menu>
```

Gestion des événements de l'ActionBar

Pour gérer les événements de l'ActionBar, on utilise la méthode onOptionsItemSelected().

Cette méthode prend en paramètre un MenuItem qui représente l'item de menu sélectionné par l'utilisateur.

Il est recommandé d'utiliser un when statament pour gérer tous les Items de menu.

On peut également utiliser la méthode onPrepareOptionsMenu() pour préparer le menu c'est-à-dire donner une apparence à l'ActionBar.

Navigation avec les onglets et les menus déroulants

Dans cette partie du cours, nous aborderons la navigation dans une application Android en utilisant Kotlin.

Nous nous concentrerons sur deux éléments clés de l'interface utilisateur : l'ActionBar et les menus.

L'ActionBar est l'une des composantes les plus importantes de l'interface Android, car elle fournit des options de navigation et d'accès rapide aux fonctionnalités les plus couramment utilisées.

De plus, les menus permettent de grouper des options en fonction de leur utilisation.

Navigation par onglets (TabLayout)

Le TabLayout est un élément d'interface utilisateur Android qui permet d'afficher des onglets horizontaux et de les connecter à des fragments.

Les utilisateurs peuvent changer de fragment en cliquant sur l'onglet correspondant.

Pour ajouter un TabLayout à votre application, vous devez utiliser la bibliothèque AndroidX.

Navigation par menus déroulants (Spinner)

Le Spinner est une composante qui permet à l'utilisateur de choisir une option dans une liste déroulante.

Il est souvent utilisé pour afficher des options de navigation ou pour sélectionner des données.

Gestion des événements de navigation

Il est important de pouvoir gérer les événements de navigation dans votre application Android.

Les événements de navigation peuvent inclure la sélection d'un onglet, le choix d'une option dans un menu déroulant ou la pression d'un bouton de retour.

Pour gérer les événements de navigation dans votre application, vous pouvez utiliser des méthodes telles que `onTabSelected()` ou `onItemSelected()`.

Ces méthodes peuvent être utilisées pour ajuster le comportement de l’application en fonction des actions de l’utilisateur.