Podstawy Android Studio

➢ ➢ Zapoznanie się ze składnikami interfejsu użytkownika Android Studio

➢Nauczenie się, jak sterować i konfigurować główne obiekty

➢Przeglądanie hierarchii różnych układów i widżetów

Poznaliśmy proces instalacji i konfiguracji Android Studio, przejrzeliśmy skrypty Gradle Build i pracowaliśmy z projektami. Tu omówimy podstawy interfejsu użytkownika (UI) Android Studio i wejdziemy w interakcję z jego głównymi komponentami i widżetami. Android Studio można opisać jako standardowe środowisko okienkowe. Aby jak najlepiej wykorzystać ograniczony ekran i uniknąć pomyłki, Android Studio wyświetla tylko niewielką część wszystkich dostępnych informacji w danym momencie. Niektóre z tych elementów obszaru zawartości są zależne od kontekstu i pojawiają się tylko wtedy, gdy kontekst jest odpowiedni, podczas gdy inne pozostają ukryte, dopóki nie ustawisz ich pokazywania, lub odwrotnie, pozostają widoczne, dopóki nie ustawisz ich ukrywania. Dlatego, aby móc w pełni korzystać z Android Studio, konieczne jest zrozumienie podstawowych funkcji tych interaktywnych aplikacji, a także tego, jak i kiedy je wyświetlać. Pokażemy, jak zarządzać folderami, układami, ciągami i widokami w Android Studio.

UI STUDIA ANDROIDA

Standardowy interfejs użytkownika Android Studio składa się z paska akcji i obszaru zawartości aplikacji. Pasek akcji, zwykle nazywany głównym paskiem akcji, reguluje kontrolę obszaru widoku i zawartości. Zrozumienie elementów ekranu zaczyna się od podstawowej jednostki aplikacji na Androida, jaką jest aktywność. Interfejs użytkownika jest zdefiniowany w pliku XML, a podczas kompilacji każdy element w pliku XML jest kompilowany do równoważnej klasy graficznego interfejsu użytkownika systemu Android (GUI) z atrybutami reprezentowanymi przez metody. Każde działanie składa się z widoków. Widok tutaj oznacza widżet, który pojawia się na ekranie. Jeden lub więcej widoków można zgrupować w jeden GroupView. Typowym przykładem ViewGroup są układy. Jeśli chodzi o układy, musisz wiedzieć, z jakim typem masz do czynienia. Istnieje wiele rodzajów układów. Niektóre z nich to: układ liniowy, bezwzględny, tabelowy, ramkowy i względny. O układach omówimy szczegółowo w następnej części. Oprócz wyżej wymienionych atrybutów istnieją inne atrybuty, które są wspólne we wszystkich widokach i ViewGroups. Mogą być wymienione w następującej kolejności:

1. layout_width: Określa szerokość widoku lub grupy widoków.

2. layout_height: Określa wysokość widoku lub grupy widoków.

3. layout_marginTop: Określa dodatkowe miejsce w górnej części widoku lub grupy widoków.

4. layout_marginBottom: Określa dodatkowe miejsce w dolnej części widoku lub grupy widoków.

5. layout_marginLeft: Określa dodatkowe miejsce po lewej stronie widoku lub grupy widoków.

6. layout_marginRight: Określa dodatkowe miejsce po prawej stronie widoku lub grupy widoków.

7. layout_gravity: Określa, w jaki sposób widoki podrzędne są pozycjonowane.

8. layout_weight: Określa, jaka część dodatkowej przestrzeni w układzie powinna zostać przydzielona do widoku.

Rozważając jednostki miary i próbując określić rozmiar dowolnego elementu w interfejsie użytkownika Androida, należy pamiętać o następujących jednostkach oszacowania wymiarów:

1. DP: piksel niezależny od gęstości. 1 dp w Android Studio odpowiada jednemu pikselowi na ekranie o rozdzielczości 160 dpi.

2. sp: piksel niezależny od skali. Jest to podobne do dp i jest zalecane do określania rozmiarów czcionek w Android Studio.

3. pt: Punkt. Punkt jest zdefiniowany jako 1/72 cala na podstawie fizycznego rozmiaru ekranu.

4. px: piksel. Odpowiada rzeczywistym pikselom na ekranie.

Jeśli chodzi o gęstość ekranu, mimo że Android nie używa bezpośredniego mapowania pikseli, używa garści skwantowanych wartości Density Independent Pixel, a następnie skaluje się do rzeczywistego rozmiaru ekranu:

1. Niska gęstość (ldpi): 120 dpi

2. Średnia gęstość (mdpi): 160 dpi

3. Wysoka gęstość (hdpi): 240 dpi

4. Bardzo wysoka gęstość (xhdpi): 320 dpi

POJĘCIA I FOLDERY

Wracając do paska akcji, nadal należy go postrzegać jako jeden z najważniejszych elementów projektu w działaniach Twojej aplikacji, ponieważ zapewnia on strukturę wizualną i elementy interaktywne wymagane przez użytkowników. Korzystanie z paska aplikacji sprawia, że Twoja aplikacja jest spójna z innymi aplikacjami na Androida, pozwalając użytkownikom szybko zrozumieć, jak obchodzić się z nią i zapewnić płynne działanie. Kluczowe funkcje paska akcji obejmują:

• Odpowiednia przestrzeń do nadania Twojej aplikacji tożsamości i wskazania lokalizacji użytkownika w aplikacji

• Zapewnienie dostępu do ważnych działań w odpowiedni sposób, takich jak nawigacja

• Obsługa przełączania widoków za pomocą kart lub list rozwijanych

Pasek akcji to element znajdujący się w górnej części ekranu aktywności. Jest to wyróżniająca się cecha aplikacji mobilnej, która jest konsekwentnie obecna nad wszystkimi jej działaniami. Zapewnia wizualną architekturę aplikacji i zawiera niektóre z często używanych elementów dla użytkowników. Android Action Bar został wprowadzony przez Google w 2013 roku wraz z wydaniem Androida 3.0 (API 11). Wcześniej ten najwyższy element wizualny nazywał się AppBar. W tamtym czasie przechowywana była tylko nazwa aplikacji lub bieżąca aktywność. Nie był zbyt przydatny dla użytkowników, a programiści szukali opcji, aby go dostosować. Wraz z paskiem akcji Google zawiera bibliotekę wsparcia, która jest częścią AppCompat, której celem jest zapewnienie wstecznej kompatybilności dla starszych wersji Androida i obsługi interfejsów z kartami. Wszystkie aplikacje korzystające z domyślnego motywu dostarczanego przez system Android (nazwa projektu to — Theme.AppCompat.Light.DarkActionBar) zawierają domyślnie pasek akcji. Jednak programiści mogą go modyfikować na kilka sposobów w zależności od swoich wymagań. Komponenty zawarte na pasku akcji to:

• Ikona aplikacji: Wyświetla logo/ikoną marki aplikacji.

• Kontrolki widoku: Sekcja, która wyświetla nazwę aplikacji lub aktualnej aktywności. Możesz także włączyć nawigację w postaci pokrętła lub kart, aby przełączać się między widokami tutaj.

• Przycisk akcji: Zawiera kilka ważnych czynności/elementów aplikacji, które mogą być często wymagane.

• Przepełnienie akcji: Obejmuje inne akcje, które będą wyświetlane jako menu.

Dla każdego elementu menu można skonfigurować następujące atrybuty w celu zaawansowanego zastosowania:

• android:title: jego wartość obejmuje tytuł elementu menu, który zostanie wyświetlony, gdy użytkownik kliknie i przytrzyma ten element w aplikacji.

• android:id: Unikalny identyfikator elementu menu, który zostanie zastosowany w celu uzyskania do niego dostępu z dowolnego miejsca w całych plikach aplikacji.

• android:orderInCategory: Wartość tego atrybutu określa pozycję elementu w ActionBar. Istnieją dwa sposoby określenia pozycji różnych elementów menu. Pierwszym z nich jest podanie tej samej wartości tego atrybutu dla wszystkich pozycji, a pozycja zostanie zdefiniowana w tej samej kolejności, w jakiej są zapisane w kodzie. Drugi sposób to podanie różnej wartości liczbowej dla wszystkich produktów, a następnie przedmioty ułożą się w porządku rosnącym według wartości tego atrybutu.

• app:showAsAction: Ten atrybut określa, w jaki sposób element będzie obecny na pasku akcji.

• android:icon: ikona elementu jest przywoływana w katalogach do rysowania za pośrednictwem tego atrybutu.

Oprócz wyżej wymienionych atrybutów istnieją cztery możliwe akcje flagi, które można zastosować do komponentów paska akcji:

1. always: Wybranie wyświetlania elementu na pasku akcji przez cały czas.

2. ifRoom: Przechowywanie przedmiotu, jeśli jest dostępne miejsce.

3. never: z tą flagą element nie będzie wyświetlany jako ikona na pasku akcji, ale zostanie umieszczony w rozszerzonym menu.

4.withText: Aby przedstawić element zarówno jako ikonę, jak i tytuł, możesz również zastąpić tę flagę flagą always lub ifRoom (always|withText lub ifRoom|withText).

Pasek akcji ma wiele zalet. Po pierwsze, działa jako świetnie dostosowany obszar do budowania tożsamości aplikacji. Śledzi również najczęściej używane czynności i zapewnia określenie lokalizacji użytkownika w aplikacji. Z drugiej strony nie wszystkie funkcje paska akcji są wprowadzane od razu, ale zajęło to dość dużo czasu wraz z wydaniem różnych poziomów API, takich jak API 15, 17 i 19. Jednocześnie pasek akcji ma tendencję do zachowywania się inaczej, gdy działa na różnych poziomach interfejsu API, a funkcje wprowadzone z konkretnym interfejsem API nie zapewniają kompatybilności wstecznej.

Dzielenie paska działań

Ponieważ elementy akcji współdzielą nieruchomość paska akcji z ikoną i tytułem aplikacji, może być konieczne podzielenie paska akcji tak, aby elementy akcji przemieściły się na dół ekranu. Dzięki temu uzyskają więcej miejsca, a tym samym więcej elementów będzie dostępnych dla użytkownika. Jeśli jednak jest wystarczająco dużo miejsca, na przykład na większych ekranach lub w trybie poziomym, paska akcji nie można podzielić. Aby podzielić pasek działań, po prostu dodaj android:uiOptions="splitActionBarWhenNarrow" do każdej aktywności w pliku manifestu, dla której potrzebujesz podzielonego paska działań. Pamiętaj jednak, że obsługuje to tylko poziom API 14 i

wyższy. Jeśli potrzebujesz dodać obsługę niższych poziomów, powinieneś użyć następującego elementu meta-data:

```
< activity

android:name="com.example.actionbar.MainActivity"

android:label="@string/app_name"

android:uiOptions="splitActionBarWhenNarrow" >

< meta-data android:name="android.support.UI_OPTIONS"

android:value="splitActionBarWhenNarrow" / >

< intent-filter >

< action android:name="android.intent.action.MAIN" / >

< category android:name="android.intent.category.LAUNCHER" / >

< /intent-filter >

< /activity >
```

Ukrywanie paska akcji

Zdarzają się sytuacje, w których nie chcesz, aby pasek akcji był widoczny przez cały czas. Typowym przykładem jest aplikacja Galeria, która ukrywa pasek akcji, gdy użytkownik patrzy na obraz, i pokazuje pasek akcji, gdy go dotknie. Aby zmienić widoczność paska działań przy dotyku, wstaw następującą funkcję do pliku aktywności:

```
@Override

public boolean onTouchEvent(MotionEvent event) {

if(event.getAction() == MotionEvent.ACTION_DOWN) {

toggleActionBar();

}

return true;

}

private void toggleActionBar() {

ActionBar actionBar = getActionBar();

if(actionBar != null) {

if(actionBar.isShowing()) {

actionBar.hide();

}

else {
```

```
actionBar.show();

}

}

}
```

Należy pamiętać, że możliwe jest pokazanie/ukrycie paska akcji, dotykając ekranu podczas uruchamiania aplikacji. Możesz również zauważyć, że zawartość na ekranie zmienia pozycję z każdym pokazaniem/ukrywaniem. Dzieje się tak, ponieważ gdy ukryjesz/pokażesz pasek akcji, aktywność zmienia się, wpływając na rozmiar i położenie treści. Aby temu zapobiec, zaleca się nałożenie paska akcji zgodnie z opisem poniżej.

Nakładanie paska akcji

Nałożenie paska akcji zapewnia lepsze ukrycie/pokazanie, ponieważ aktywność nie zmienia rozmiaru przy każdym ukryciu/pokażeniu, dzięki czemu zawartość pozostanie na swoim miejscu. Możesz włączyć nakładanie, ustawiając android:windowActionBarOverlay na true w pliku motywu, który powinien być motywem Theme.Holo lub jednym z jego potomków. W res/values/styles.xml wstaw następujące:

```
< resources >

< style name="AppBaseTheme" parent="android:Theme.Light" >

< /style >

< !-- Application theme. -- >

< style name="AppTheme" parent="AppBaseTheme" >

< item name="android:windowActionBarOverlay" >true< /item >

< /style >

< /resources >
```

Po aktywacji aplikacji powinieneś być w stanie zauważyć, że zawartość na ekranie nie zmienia pozycji, niezależnie od ukrycia lub odsłonięcia paska akcji. Podsumowując tę sekcję, pasek akcji jest ważnym elementem projektu i należy go opanować, aby znacznie poprawić wrażenia użytkownika aplikacji. W tym miejscu przyjrzeliśmy się kilku podstawowym konfiguracjom, których możesz użyć na pasku akcji interfejsu użytkownika aplikacji, ale pamiętaj, że jest więcej do nauczenia się, gdy będziesz na to gotowy.

UKŁADY

Android Layout służy do definiowania interfejsu użytkownika, w którym znajdują się elementy sterujące interfejsu użytkownika lub widżety wyświetlane na ekranie aplikacji lub ekranie aktywności. Z reguły każda aplikacja jest kombinacją View i ViewGroup. A jak już wspomnieliśmy, aplikacja na Androida zawiera dużą liczbę czynności, a każdą czynność możemy traktować jako jedną stronę aplikacji. Oznacza to, że każde działanie zawiera wiele składników interfejsu użytkownika, a te składniki są instancjami View i ViewGroup. Wszystkie elementy widoczne w układzie są zbudowane przy użyciu hierarchii obiektów View i ViewGroup. Widok można również zdefiniować jako interfejs użytkownika, który jest niezbędny do tworzenia interaktywnych składników interfejsu użytkownika, takich jak TextView, ImageView, EditText lub RadioButton. Ponadto odpowiada również za obsługę zdarzeń i rysowanie. ViewGroup byłaby wtedy klasą bazową dla układów i parametrów układów, które zawierają

inne widoki lub ViewGroups, a także definiują właściwości układu. Platforma Androida zazwyczaj pozwala użytkownikom używać elementów interfejsu użytkownika lub widżetów na dwa sposoby: podczas budowania pliku XML lub podczas dynamicznego tworzenia elementów w pliku Kotlin.

Pozycja układu

Geometria (kształt) widoku to prostokąt. Widok ma również lokalizację, przedstawianą jako para lewych i górnych współrzędnych oraz dwa wymiary, które są szerokością i wysokością. Jednostką lokalizacji i wymiarów jest piksel. W razie potrzeby możesz pobrać lokalizację widoku, wywołując metody getLeft() i getTop(). Pierwsza z nich zwraca lewą lub X współrzędną prostokąta reprezentującego widok. Ta ostatnia zwraca górną lub Y współrzędną prostokąta reprezentującego widok. Obie te metody zwracają położenie widoku względem jego rodzica. Aby to zilustrować, gdy getLeft() zwraca 20, oznacza to, że widok znajduje się 20 pikseli na prawo od lewej krawędzi swojego bezpośredniego rodzica. Ponadto istnieje kilka wygodnych metod, które zapobiegają niepotrzebnym obliczeniom, a mianowicie GetRight() i getBottom(). Te metody zwracają współrzędne prawej i dolnej krawędzi prostokąta reprezentującego widok. Na przykład wywołanie GetRight() byłoby podobne do następującego obliczenia: getLeft() + getWidth(). Wielkość widoku wyraża się szerokością i wysokością. Widok zazwyczaj ma dwie pary wartości szerokości i wysokości. Pierwsza para jest znana jako zmierzona szerokość i zmierzona wysokość. Te wymiary są używane do określenia, jak duży widok ma być w swoim rodzicu. Zmierzone wymiary można zebrać, wywołując getMeasuredWidth() i getMeasuredHeight(). Druga para jest po prostu znana jako szerokość i wysokość, a niektórzy mogą nazywać ją szerokością i wysokością rysunku. Te wymiary określają rzeczywisty rozmiar widoku na ekranie, w czasie rysowania i po układzie. Wartości te mogą, ale nie muszą, różnić się od zmierzonej szerokości i wysokości. Szerokość i wysokość można uzyskać, wywołując getWidth() i getHeight(). Aby zmierzyć jego wymiary, widok musi wziąć pod uwagę jego wypełnienie. Dopełnienie jest wyrażone w pikselach dla lewej, górnej, prawej i dolnej części widoku. Dopełnienie można zastosować, aby przesunąć zawartość widoku o określoną liczbę pikseli. Na przykład lewe dopełnienie 2 przesunie zawartość widoku o 2 piksele w prawo od lewej krawędzi. W ten sposób dopełnienie można ustawić za pomocą metody setPadding(int, int, int, int) i zapytać, wywołując getPaddingLeft(), getPaddingTop(), getPaddingRight() i getPaddingBottom(). Warto zauważyć, że nawet jeśli widok może definiować dopełnienie, nie zapewnia obsługi marginesów. Jednak takie wsparcie można by zapewnić za pomocą grup widoków.

Rodzaje układu Androida

Jak już wspomniano, istnieją różne typy Android Layout. Lista wygląda tak:

• Układ liniowy systemu Android: LinearLayout jest podklasą ViewGroup, używaną do dostarczania potomnych elementów View jeden po drugim w określonym kierunku, poziomo lub pionowo w oparciu o właściwość orientacji.

• Układ względny systemu Android: RelativeLayout jest podklasą ViewGroup, używaną do określania pozycji podrzędnych elementów View względem siebie (A na prawo od B) lub względem rodzica (umocowane na górze elementu rodzica).

• Układ ograniczeń systemu Android: ConstraintLayout to podklasa ViewGroup, używana do ustawiania położenia ograniczeń układu dla każdego widoku podrzędnego względem innych obecnych widoków. ConstraintLayout jest prawie podobny do RelativeLayout, ale ma więcej funkcji.

• Układ ramki systemu Android: FrameLayout to podklasa ViewGroup, używana do określania położenia elementów widoku, które zawiera jeden nad drugim, aby wyświetlić tylko jeden widok wewnątrz ramki FrameLayout.

• Układ tabeli Android: TableLayout jest podklasą ViewGroup, używaną do wyświetlania podrzędnych elementów View w wierszach i kolumnach.

• Android Web View: WebView to przeglądarka używana do wyświetlania stron internetowych w naszym układzie aktywności.

• Android ListView: ListView to ViewGroup, używany do wyświetlania listy elementów z możliwością przewijania w jednej kolumnie.

• Widok siatki systemu Android: GridView to ViewGroup, który służy do wyświetlania przewijanej listy elementów w widoku siatki wierszy i kolumn.

Korzystanie z elementów interfejsu użytkownika w pliku XML

Za pomocą pliku XML można stworzyć układ podobny do stron internetowych. Plik układu XML zawiera co najmniej jeden element główny, w którym można uwzględnić dodatkowe elementy układu lub widżety w celu zbudowania hierarchii widoku. Spójrzmy na ten przykład:

```
< ?xml version="1.0" encoding="utf-8"? >

< LinearLayout

xmlns:android="http:// schemas.android.com/apk/res/android"

xmlns:tools="http:// schemas.android.com/tools"

android:orientation="vertical"

android:layout_width="match_parent"

android:layout_height="match_parent"

tools:context=".MainActivity" >

< !--EditText with id editText-- >

< EditText

android:id="@+id/editText"

android:layout_width=

"match_parent"

android:layout_height=

"wrap_content"

android:layout_margin="16dp"

android:hint="Input"

android:inputType="text"/ >

< !--Button with id showInput-- >
```

< Button

android:id="@+id/showInput"

android:layout_width=

"wrap_content"

android:layout_height=

"wrap_content"

android:layout_gravity=

"center_horizontal"

android:text="show"

android:backgroundTint="@color/colorPrimary"

android:textColor="@android:color/white"/ >

< /LinearLayout >

Po utworzeniu układu musisz teraz załadować zasób układu XML z metody wywołania zwrotnego Activity onCreate() i uzyskać dostęp do elementu UI z XML za pomocą findViewById:

override fun onCreate(savedInstanceState: Bundle?) {

super.onCreate(savedInstanceState)

setContentView(R.layout.activity_main)

// finding the button

val showButton = findViewById<Button>(R.id.showInput)

// finding the edit text

val editText = findViewById<EditText>(R.id.editText)

Powinieneś obserwować powyższy kod i zobaczyć, że wywołujemy nasz układ za pomocą metody setContentView w postaci R.layout.activity_main. Normalnie, podczas uruchamiania naszej aktywności, metoda wywołania zwrotnego onCreate() zostanie pobrana przez framework Androida, aby uzyskać wymagany układ dla aktywności. Układy to istotny segment aplikacji na Androida, który bezpośrednio wpływa na wygodę użytkownika. Jeśli projekt i wykonanie są źle zaprojektowane, Twój układ może prowadzić do aplikacji, która wymaga dużej ilości pamięci z wolnymi interfejsami użytkownika. Android SDK zawiera narzędzia, które pomagają w identyfikowaniu problemów z wydajnością układu, co po zaimplementowaniu spowoduje płynne przewijanie interfejsów przy minimalnym zużyciu pamięci. Powszechnym błędem jest myślenie, że przestrzeganie podstawowych struktur układu prowadzi do najbardziej wydajnych układów. Jednocześnie każdy widżet i układ zawarty w aplikacji wymaga inicjalizacji, układu i rysunku. Aby zilustrować to przykładem, użycie zagnieżdżonych wystąpień LinearLayout może prowadzić do zbyt głębokiej hierarchii widoków. Co więcej, zagnieżdżanie kilku wystąpień LinearLayout, które używają parametru layout_weight, może być szczególnie kosztowne, ponieważ każde dziecko musi być mierzone dwukrotnie. Jest to szczególnie ważne, gdy układ jest wielokrotnie napełniany, na przykład gdy jest używany w ListView lub GridView. Są jednak pewne rzeczy, które możesz zrobić, aby zoptymalizować wydajność układu

Sprawdź swój układ

Android SDK zawiera narzędzie o nazwie Hierarchy Viewer, które umożliwia analizowanie układu podczas działania aplikacji. Aktywacja tego narzędzia pomoże Ci odkryć potencjalne problemy z wydajnością układu. Sposób, w jaki działa Hierarchy Viewer, polega na umożliwieniu wyboru uruchomionych procesów na podłączonym urządzeniu lub emulatorze, a następnie przedstawieniu całego drzewa układu. Sygnalizacja świetlna na każdym bloku reprezentuje jego wydajność pomiaru, układu i rysowania, co pozwala zidentyfikować potencjalne problemy.

Popraw swój układ

Ponieważ wydajność układu może ulec spowolnieniu z powodu zagnieżdżonego LinearLayout, wydajność może się poprawić, spłaszczając układ — czyniąc go płytkim i szerokim, a nie wąskim i głębokim. Choć mogą się wydawać niewielkie, zalety są zwielokrotniane kilka razy, ponieważ ten układ można zastosować do prawie każdej aktywności. Większość różnic wynika z użycia parametru layout_weight w ustawieniu LinearLayout, co może spowolnić szybkość pomiaru. To tylko jeden przykład tego, jak każdy układ ma odpowiednie zastosowania i należy dokładnie sprawdzić, czy użycie wagi układu jest absolutnie konieczne. W niektórych złożonych układach system może zmarnować wysiłek, mierząc ten sam element interfejsu więcej niż jeden raz. Pojęcie to nazywa się podwójnym opodatkowaniem.

Użyj Linta

Zawsze dobrze jest uruchomić narzędzie lint w plikach układu, aby sprawdzić, czy istnieją możliwe optymalizacje hierarchii widoków. Lint zastąpił narzędzie Layoutopt i ma znacznie większą funkcjonalność w takich sprawach. Istnieją jednak pewne zasady dotyczące kłaczków, o których powinieneś wiedzieć przed rozpoczęciem:

• Użyj złożonych obiektów do rysowania: LinearLayout, który zawiera ImageView i TextView, może być efektywniej zarządzany jako złożony obiekt do rysowania.

• Scal ramkę główną: Jeśli FrameLayout jest główną ramką layoutu i nie zapewnia tła ani wypełnienia, można ją zastąpić tagiem scalania, który jest nieco bardziej wydajny.

• Bezużyteczny liść: układ, który nie ma dzieci ani tła, często można usunąć, aby uzyskać bardziej płaską i wydajną hierarchię układów.

• Bezużyteczny rodzic: układ z dziećmi, który nie ma rodzeństwa i tła, można usunąć i przenieść jego dzieci bezpośrednio do rodzica, aby uzyskać bardziej płaską i wydajniejszą hierarchię układu.

• Głębokie układy: Układy ze zbyt dużym zagnieżdżeniem mają zły wpływ na ogólną wydajność. Rozważ użycie bardziej płaskich układów, takich jak RelativeLayout lub GridLayout, aby poprawić wydajność. Domyślna maksymalna głębokość powinna wynosić dziesięć.

Kolejną zaletą Lint jest płynna integracja z Android Studio. Lint uruchamia się automatycznie za każdym razem, gdy kompilujesz swój program. Dzięki Android Studio możesz mieć pewność, że przeprowadzisz inspekcje lint dla określonego wariantu kompilacji lub dla wszystkich wariantów kompilacji. Chociaż system Android oferuje różne widżety, które zapewniają małe i wielokrotnego użytku elementy interaktywne, może być również konieczne ponowne użycie większych elementów, które wymagają specjalnego układu. Aby efektywnie ponownie wykorzystać całe układy, możesz użyć znaczników < include/ > i < merge/ > , aby dołączyć inny układ do bieżącego układu. Ponowne wykorzystanie układów jest szczególnie dobre, ponieważ umożliwia tworzenie złożonych układów wielokrotnego użytku. Na przykład panel przycisków tak/nie lub niestandardowy pasek postępu z tekstem opisu. Oznacza to

również, że wszelkie elementy aplikacji, które są wspólne w wielu układach, można wyodrębnić i administrować oddzielnie przed połączeniem ich w jeden układ. Tak więc, chociaż możesz tworzyć indywidualne komponenty interfejsu użytkownika, pisząc niestandardowy widok, możesz również łatwo to zrobić, ponownie używając pliku układu.

Tworzenie układu wielokrotnego użytku

Jeśli masz już układ, którego chcesz użyć ponownie, zacznij od utworzenia nowego pliku XML i zdefiniowania układu. Na przykład weźmy układ, który definiuje pasek tytułu, który ma być uwzględniony w każdym działaniu (titlebar.xml):

< FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:tools="http://schemas.android.com/tools"

android:layout_width="match_parent"

android:layout_height="wrap_content"

android:background="@color/titlebar_bg"

tools:showIn="@layout/activity_main" >

< ImageView android:layout_width=

"wrap_content"

android:layout_height=

"wrap_content"

android:src="@drawable/gafricalogo" / >

< /FrameLayout >

Teraz widok główny powinien być wyświetlany dokładnie tak, jak chcesz, w każdym segmencie aktywności, do którego dodasz ten układ. Należy również pamiętać, że atrybut tools:showIn w powyższym pliku XML jest specjalnym atrybutem, który jest usuwany podczas kompilacji i pobierany tylko w czasie projektowania w Android Studio - identyfikuje układ, który zawiera ten plik, dzięki czemu można przeglądać i modyfikować ten plik pojawia się, gdy jest osadzony w układzie nadrzędnym.

Użyj tagu <include>

Zaleca się również dodanie tagu < include/ > wewnątrz układu, do którego chcesz dodać komponent wielokrotnego użytku. Aby to zademonstrować, spójrz na układ, który zawiera pasek tytułu z góry:

< LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

android:orientation="vertical"

android:layout_width="match_parent"

android:layout_height="match_parent"

android:background="@color/app_bg"

android:gravity="center_horizontal" >

Teraz dołącz < include layout=”@layout/titlebar”/ >

< TextView android:layout_width=

"match_parent"

android:layout_height=

"wrap_content"

android:text="@string/hello"

android:padding="10dp" / >

< /LinearLayout >

Możliwe jest również zastąpienie wszystkich parametrów układu (dowolnych atrybutów android:layout_*) w widoku głównym dołączonego układu, wstawiając je w znaczniku <include/>. Na przykład:

< include android:id="@+id/news_title"

android:layout_width=

"match_parent"

android:layout_height=

"match_parent"

layout="@layout/title"/ >

Jednocześnie, jeśli chcesz zastąpić atrybuty układu za pomocą tagu <include>, musisz zastąpić zarówno android:layout_height, jak i android:layout_width, aby aktywować inne atrybuty układu.

Użyj tagu < merge >

< merge / > jest doskonałym rozwiązaniem, gdy trzeba wyeliminować nadmiarowe grupy widoków w hierarchii widoków podczas wstawiania jednego układu w innym. Mówiąc dokładniej, jeśli układ główny jest pionowym układem LinearLayout, w którym dwa kolejne widoki mogą być ponownie użyte w wielu układach, wówczas układ wielokrotnego użytku, w którym ustawiono dwa widoki, wymaga własnego widoku głównego. Niemniej jednak użycie innego LinearLayout jako głównego układu wielokrotnego użytku zamieniłoby się w pionowy LinearLayout wewnątrz pionowego LinearLayout. Zagnieżdżony LinearLayout nie służy żadnemu innemu celowi, niż spowolnieniu wydajności interfejsu użytkownika. Aby uniknąć dodawania takiej nadmiarowej grupy widoków, możesz użyć następującego elementu < merge > jako widoku głównego dla układu wielokrotnego użytku:

< merge xmlns:android="http://schemas.android.com/apk/res/android" >

< Button

android:layout_width="fill_parent"

android:layout_height=

"wrap_content"

android:text="@string/add"/ >

< Button

android:layout_width="fill_parent"

android:layout_height=

"wrap_content"

android:text="@string/delete"/ >

< /merge >

Teraz, jeśli uwzględnisz ten układ w innym układzie (używając znacznika < include/ >), system po prostu pominie element < merge > i umieści dwa przyciski bezpośrednio w układzie, zamiast znacznika < include/ >.

CIĄGI ZNAKÓW

Element ciągu udostępnia ciągi tekstowe dla aplikacji z dodatkowym stylem i formatowaniem tekstu. Istnieją trzy główne typy ciągów znaków, których można używać w Android Studio:

• Ciąg: Identyfikowany jako zasób XML zawierający pojedynczy ciąg.

• Tablica ciągów: oznacza zasób XML, który udostępnia tablicę ciągów.

• Ciągi ilościowe (liczba mnoga): zasób XML, który zawiera różne ciągi do liczby mnogiej.

Na wypadek, gdybyś się zastanawiał, wszystkie trzy style ciągów są w stanie zastosować pewne znaczniki stylizacji i argumenty formatowania. Przyjrzyjmy się teraz bliżej każdemu typowi ciągu.

Ciąg

Do pojedynczego ciągu można odwoływać się z aplikacji, a także z innych plików zasobów, takich jak układ XML. Jest uważany za prosty element, do którego odwołuje się wartość podana w atrybucie name. Możliwe jest również połączenie zasobów tekstowych z innymi prostymi zasobami w jednym pliku XML, w jednym elemencie < resources >. Typowa lokalizacja pliku to: res/values/filename.xml . Nazwa pliku jest dowolna, ponieważ nazwa elementu <string> będzie później używana jako identyfikator zasobu. Standardowe odniesienia do zasobów to:

W Javie: R.string.string_name

Do XML:@string/string_name

Składnia, którą należy zastosować dla ciągu znaków, to:

< ?xml version="1.0" encoding="utf-8"? >

< resources >

< string

name="string_name"

>text_string< /string >

< /resources >

Jeśli chodzi o elementy składni, zwróć uwagę, że:

< resources > – to wymagana funkcja, która musi być węzłem głównym, bez atrybutów.

< string > – może zawierać tagi stylizacji, ale nie może zawierać apostrofów ani cudzysłowów.

Tablica ciągów

Do tablicy ciągów można również odwoływać się z aplikacji i być postrzegana jako prosty zasób uruchamiany przy użyciu wartości podanej w atrybucie name. Ponadto można połączyć zasoby tablicy ciągów z innymi prostymi zasobami w jednym pliku XML, w jednym elemencie < resources >. Standardowa lokalizacja pliku to: res/values/filename.xml Nazwa pliku jest również dowolna i nazwa elementu <string-array> będzie później używana jako identyfikator zasobu. Odniesienie do zasobu w Javie to: R.array.string_array_name Ogólna składnia do zastosowania to:

< ?xml version="1.0" encoding="utf-8"? >

< resources >

< string-array

name="string_array_name" >

< item

>text_string< /item >

< /string-array >

< /resources >

W tym przypadku tablica ciągów może zawierać znaczniki stylów, a także odnosić się do innego zasobu ciągu w przypadku, gdy jest potomkiem elementu <string-array>. Jednak nadal powinieneś unikać używania apostrofów lub cudzysłowów.

Ilość ciągów (liczba mnoga)

Zwykle różne języki mają różne zasady dotyczące układów gramatycznych z ilością. Na przykład w języku angielskim ilość 1 jest przypadkiem szczególnym. Możesz napisać „1 jabłko”, ale dla każdej innej ilości musiałbyś napisać „n jabłek”. Ta różnica między liczbą pojedynczą a mnogą jest bardzo powszechna, ale istnieje wiele innych języków, które mają swoje własne różnice i osobliwości. Zwykły zestaw obsługiwany przez Androida to zero, jeden, dwa, kilka, wiele i inne. Zasady decydowania o tym, który przypadek użyć dla danego języka i ilości, mogą początkowo być bardzo mylące. Dlatego Android udostępnia metody, takie jak getQuantityString(), aby wybrać odpowiedni zasób. Nawet jeśli nazywa się to „ciągami ilościowymi”, technicznie rzecz biorąc, ciągi ilościowe powinny być używane tylko w liczbie mnogiej. Nie zadziała, jeśli zastosujesz ciągi ilościowe do zaimplementowania czegoś takiego jak „Skrzynka odbiorcza” lub „Skrzynka odbiorcza (12)” Gmaila, gdy na przykład są nieprzeczytane wiadomości. Może wydawać się odpowiednie użycie ciągów ilościowych zamiast instrukcji if, ale ważne jest, aby pamiętać, że niektóre języki (takie jak chiński) w ogóle nie mają tych różnic gramatycznych, więc zawsze otrzymasz drugi ciąg. z których ciąg ma być wykonany głównie w oparciu o konieczność gramatyczną. Aby zilustrować, w języku angielskim ciąg oznaczający zero jest pomijany, nawet jeśli ilość wynosi 0, ponieważ 0 nie jest gramatycznie różne od 2 ani żadnej innej liczby z wyjątkiem 1 („zero jabłek”, „jedno jabłko”, „dwa jabłka”, Itp.). Z drugiej strony, na przykład w języku koreańskim, używany jest tylko drugi ciąg. Ważne jest, aby nie dać się zwieść faktowi, że dwa podobne dźwięki mogą odnosić się tylko do liczby 2: Język może wymagać, aby 2, 12, 101 były traktowane równo względem siebie, ale różniły się od innych wielkości. Tutaj najlepsze, co możesz zrobić, to polegać na tłumaczu, aby dowiedzieć się, na jakie różnice tak naprawdę kładą się języki obce. Czasami można w ogóle uniknąć łańcuchów ilościowych, używając formuł neutralnych pod względem ilości, takich jak „Jabłka: 1.” Może

to potencjalnie ułatwić pracę Twojej aplikacji i tłumaczom, jeśli jest to akceptowalny styl dla Twojego produktu końcowego. Podsumowując, kolekcja liczby mnogiej jest prostym zasobem, do którego odwołuje się wartość podana w atrybucie name. Dodatkowo możliwe jest również łączenie wielu zasobów z innymi prostymi zasobami w jednym pliku XML, pod jednym elementem < resources >. Domyślna lokalizacja pliku to: res/values/filename.xml. Nazwa pliku jest tutaj dowolna, ponieważ nazwa elementu będzie później używana jako identyfikator zasobu. Odniesienie do zasobów w Javie to: R.plurals.plural_name Wygenerowana składnia do zastosowania to:

< ?xml version="1.0" encoding="utf-8"? >

< resource >

< plurals

name="plural_name" >

< item

quantity=["zero" | "one" | "two" | "few" | "many" | "other"]

>text_string< /item >

< /plurals >

< /resources >

Jednocześnie, używając tego typu ciągu, pamiętaj o dodaniu wskaźników pokazujących jego wartość. Prawidłowe wartości, z niewyczerpującymi przykładami w nawiasach, obejmują:

1. Zero: ma zastosowanie, gdy język wymaga specjalnego traktowania liczby 0 (jak w języku arabskim).

2. Jeden: ma zastosowanie, gdy język wymaga specjalnego traktowania liczb, takich jak jeden (jak w przypadku liczby 1 w języku angielskim lub w języku rosyjskim z dowolną liczbą kończącą się na 1, ale nie kończącą się na 11).

3. Dwa: Ma zastosowanie, gdy język wymaga specjalnego traktowania liczb, takich jak dwa (jak w przypadku 2 w języku walijskim lub 102 w języku słoweńskim).

4. Kilka: Ma zastosowanie, gdy język wymaga specjalnego traktowania „małych” cyfr (jak w przypadku 2, 3 i 4 w języku czeskim lub cyfr kończących się na 2, 3 lub 4, ale nie 12, 13 lub 14 w języku polskim).

5. Wiele: Ma zastosowanie, gdy język wymaga specjalnego traktowania „dużych” liczb (jak w przypadku liczb kończących się 11-99 w języku maltańskim).

6. Inne: Ma zastosowanie, gdy język nie wymaga specjalnego traktowania danej ilości (jak wszystkie liczby w języku chińskim lub 42 w języku angielskim).

Jest też kilka ważnych rzeczy, o których należy się dowiedzieć, jak prawidłowo formatować i stylizować zasoby tekstowe

Obsługa znaków specjalnych

Gdy ciąg zawiera znaki, które mają specjalne zastosowanie w XML, należy je pominąć zgodnie ze standardowymi regułami kodu XML/HTML. W takich przypadkach, gdy musisz uciec przed znakiem, który ma specjalne znaczenie w Androidzie, możesz wstawić poprzedzający ukośnik odwrotny. Domyślnie Android łączy sekwencje białych znaków w jedną spację. Możesz temu zapobiec,

umieszczając odpowiednią część ciągu w podwójnym cudzysłowie. W takim przypadku wszystkie znaki odstępu (w tym znaki nowej linii) pozostaną nieedytowane w obrębie cytowanego regionu. Podwójne cudzysłowy pozwolą ci również używać zwykłych pojedynczych cudzysłowów bez ucieczki. Standardowe formularze ze znakami specjalnymi obejmują:

Znak : Znak ucieczki

@ : \@

? : \?

Nowa linia: \n

Tabulator : \t

U + XXXX znak Unicode: \uXXXX

Pojedynczy cudzysłów (') : \'

Cudzysłów (") : \"

Czasami możesz mieć do czynienia z przypadkami zwijania białych znaków i ucieczki Androida po przetworzeniu pliku zasobów jako XML. Oznacza to po prostu, że < string > &# 30; &# 830; &# 895; < /string > (spacja, spacja interpunkcyjna, spacja Unicode Em) wszystkie zwijają się do pojedynczej spacji („ "), ponieważ wszystkie są spacją Unicode po przetworzeniu pliku jako XML. Aby zachować te spacje takimi, jakimi są, możesz je zacytować (< string >„ &# 30; &# 830; &# 895;"< /string >) lub użyć ucieczki Androida (< string > \ u0030 \ u830 \ u895< /string >). Jednak z punktu widzenia parsera XML nie ma żadnej różnicy między < string >„Testuj to"< /string > a < string >„Testuj to"< /string > w ogóle. Oba formularze nie wyświetlają żadnych cudzysłowów, ale aktywują cytowanie z zachowaniem białych znaków w systemie Android, które w tym przypadku nie będzie miało praktycznego wpływu.

Formatowanie ciągów

Jeśli musisz sformatować swoje ciągi, możesz to zrobić, umieszczając argumenty formatu w zasobie ciągu, jak pokazano w poniższym przykładzie:

< string name="welcome_messages">Witaj, %1$s! Masz %2$d nowych wiadomości.< /string >

W tym przykładzie ciąg formatu ma dwa argumenty: %1$s to ciąg, a %2$d to liczba dziesiętna. Tutaj musisz sformatować ciąg, wywołując getString(int, Object…). Na przykład:

var text = getString(R.string.welcome_messages, username, mailCount)

Możliwe jest również dodanie stylu do swoich ciągów za pomocą znaczników HTML. Na przykład:

< ?xml version="1.0" encoding="utf-8"? >

< resources >

< string name="welcome" >Welcome to Android!< /string >

< /resources >

Możesz skorzystać z następujących obsługiwanych elementów HTML:

1. Pogrubienie: < b > , < em >

2. Kursywa: < i >, < cite >, < dfn >

3. 25% większy tekst: < big >

4. 20% mniejszy tekst: < small >

5. Ustawianie właściwości czcionki: < font face="font_family" color="hex_color" >. Przykłady możliwych rodzin czcionek to monospace, serif i sans_serif.

6. Ustawianie rodziny czcionek o stałej szerokości: < tt >

7. Przekreślenie: < s >, < strike >, < del >

8. Podkreślenie: < u >

9. Indeks górny: < sup >

10. Indeks dolny: < sub >

11. Punktory: < ul >, < li >

12. Łamanie wierszy: < br >

13. Dzielenie: < div >

14. Styl CSS: < span style="color|background_color|text-decoration" >

15.Paragrafy: < p dir="rtl | ltr" style=„… >

Jeśli jednak nie musisz stosować formatowania, możesz ustawić tekst TextView bezpośrednio, wywołując setText(java.lang.CharSequence). Jeśli od czasu do czasu może zajść potrzeba utworzenia zasobu stylizowanego tekstu, który jest również używany jako ciąg formatu, należy to zrobić, pisząc znaczniki HTML z jednostkami ze znakami ucieczki, które są następnie odzyskiwane za pomocą zromHtml(String) po zakończeniu formatowania. Na przykład :

Zacznij od zapisania stylizowanego zasobu tekstowego jako ciągu znaków ze znakami ucieczki HTML:

< resources >

< string name="welcome_messages" >Hello, %1$s! You have <b >%2$d new messages<

/b >.< /string >

< /resources >

Następnie sformatuj ciąg jak zwykle, ale wywołaj także fromHtml (String), aby przekonwertować tekst HTML na tekst stylizowany:

val text: String = getString(R.string.welcome_messages, username, mailCount)

val styledText: Spanned = Html.fromHtml(text, FROM_HTML_MODE_LEGACY)

Ponieważ metoda fromHtml(String) formatuje wszystkie jednostki HTML, upewnij się, że wszystkie możliwe znaki HTML w ciągach używanych z sformatowanym tekstem zostały zmienione, używając htmlEncode(String). Na przykład, jeśli formatujesz ciąg, który zawiera znaki takie jak „<" lub „&", przed formatowaniem należy je zmienić, tak aby po przejściu sformatowanego ciągu znaków fromHtml(String) znaki wychodziły w taki sposób, w jaki zostały pierwotnie napisane. Aby zilustrować przykładem:

val escapedUsername: String = TextUtils.htmlEncode(username)

val text: String = getString(R.string.welcome_messages, escapedUsername, mailCount)

val styledText: Spanned = Html.fromHtml(text, FROM_HTML_MODE_LEGACY)

VIEW

Android oferuje wyrafinowany i silnie skomponowany model do budowania interfejsu użytkownika, oparty na podstawowych klasach układu: View i ViewGroup. Przejrzeliśmy już różne wstępnie zbudowane podklasy View i ViewGroup - zwane odpowiednio widżetami i układami - które można wykorzystać do skonstruowania swojego interfejsu użytkownika. Częściowa lista dostępnych widżetów, o których wspomnieliśmy, obejmowała Button, TextView, EditText, ListView, CheckBox, RadioButton, Gallery, Spinner i bardziej specjalne AutoCompleteTextView, ImageSwitcher i TextSwitcher. Wśród dostępnych układów są LinearLayout, FrameLayout, RelativeLayout i inne. Jeśli znajdziesz się w sytuacji, w której żaden z gotowych widżetów lub układów nie spełnia Twoich potrzeb, zawsze możesz utworzyć własną podklasę View. Jeśli potrzebujesz tylko dokonać niewielkich zmian w istniejącym widżecie lub układzie, możesz po prostu podklasy widżetu lub układu i nadpisać jego metody. Tworzenie własnych podklas View zapewnia precyzyjną kontrolę nad wyglądem i funkcją wszystkich elementów ekranu. Aby dać ci mgliste pojęcie o autorytecie, jaki uzyskujesz dzięki niestandardowym widokom, oto kilka przykładów tego, co możesz z nimi zrobić:

• Możesz stworzyć całkowicie niestandardowy typ widoku lub całkowitą kontrolę głośności za pomocą grafiki 2D, która przypomina analogową kontrolkę elektroniczną.

• Możesz połączyć grupę elementów Widoku w nowy pojedynczy komponent, aby stworzyć coś takiego jak ComboBox (kombinacja listy podręcznej i pola tekstowego swobodnego wprowadzania) lub dwupanelowa kontrolka selektora (lewy i prawy panel z w każdym miejscu, w którym możesz zmienić przypisanie, który element znajduje się na której liście).

• Możesz zmodyfikować sposób, w jaki składnik EditText pojawia się na ekranie (Samouczek Notatnika wykorzystuje to z dobrym skutkiem do tworzenia strony notatnika w linie).

• Powinieneś także być w stanie uchwycić inne zdarzenia, takie jak naciśnięcia klawiszy, i obsłużyć je w niestandardowy sposób (np. w przypadku gry).

Zobaczmy teraz kilka praktycznych podejść do tworzenia niestandardowych widoków i używania ich w twojej aplikacji.

Podstawowe podejście

Oto krótki przegląd tego, co musisz wiedzieć, aby rozpocząć tworzenie własnych komponentów widoku:

Najpierw rozszerz istniejącą klasę lub podklasę View o własną klasę. Upewnij się, że zastąpiłeś niektóre metody z nadklasy. Metody nadklasy do przesłonięcia zaczynają się od „on”, na przykład onDraw(), onMeasure() i onKeyDown(). Są to te same, których używasz dla zdarzeń w działaniu lub ListActivity, które zastępujesz dla cyklu życia i innych ustawień funkcji. W takim razie powinieneś używać swojej nowej klasy rozszerzeń. Po zakończeniu nowa klasa rozszerzenia może zostać zastosowana w miejscu widoku, na którym została oparta. Ponadto te same klasy rozszerzające można zdefiniować jako klasy wewnętrzne w działaniach, które ich używają. Jest to korzystne, ponieważ kontroluje dostęp do nich, ale jednocześnie nie jest absolutnie konieczne (może być w przypadku, gdy chcesz utworzyć nowy widok publiczny do szerszego wykorzystania w swojej aplikacji).

W pełni dostosowane komponenty

W pełni dostosowane komponenty mogą być używane do tworzenia komponentów graficznych, które pojawiają się w dowolnym momencie iw dowolnej formie i formie. Możesz potrzebować graficznego miernika głośności, który wygląda jak stary miernik analogowy, lub widoku tekstu do śpiewania, dzięki czemu możesz śpiewać razem z maszyną do karaoke. Tak czy inaczej, standardowe komponenty wbudowane nie mogą tego zapewnić, bez względu na to, jak je połączysz. Możesz jednak łatwo tworzyć komponenty, które wyglądają i działają w dowolny sposób, ograniczone być może tylko Twoją wyobraźnią, rozmiarem ekranu i dostępną mocą operacyjną (pamiętaj, że Twoja aplikacja może wymagać uruchomienia na czymś znacznie mniej energii niż twoja stacja robocza). Aby utworzyć w pełni dostosowany komponent: zwykle musisz zacząć od rozszerzenia widoku, aby utworzyć nowy superkomponent. Możesz dostarczyć konstruktor, który może pobierać atrybuty i parametry z XML, a także możesz ustawić własne parametry (na przykład kolor, zakres lub szerokość). Wtedy prawdopodobnie chciałbyś stworzyć własne detektory zdarzeń, akcesory i modyfikatory właściwości, a być może również bardziej wyrafinowane działanie klasy komponentów. W takim przypadku prawie na pewno będziesz musiał przesłonić onMeasure() i prawdopodobnie będziesz także musiał przesłonić onDraw(), jeśli chcesz, aby komponent coś pokazywał. Chociaż oba mają domyślne zachowanie, domyślna onDraw() nic nie zrobi, a domyślna onMeasure() zawsze ustawi rozmiar 100 × 100 — co nie jest najbardziej pożądaną opcją.

Rozszerz onDraw() i onMeasure()

Metoda onDraw() działa jak płótno, na którym można zaimplementować wszystko, czego potrzebujesz: grafikę 2D, inne standardowe lub niestandardowe komponenty, stylizowany tekst lub cokolwiek innego, czego możesz potrzebować do projektu. Na marginesie, pamiętaj, że nie dotyczy to grafiki 3D. Jeśli potrzebujesz użyć grafiki 3D, musisz rozszerzyć SurfaceView zamiast View i rysować z oddzielnego gwintu. onMeasure() powinien być postrzegany jako kluczowy element umowy renderowania między twoim komponentem a jego kontenerem. onMeasure() należy zmodyfikować, aby sprawnie i odpowiednio raportować pomiary zawartych w nim części. Nieco bardziej komplikują to wymagania dotyczące limitów od rodzica (które są przekazywane do metody onMeasure()) oraz wymóg wywołania metody setMeasuredDimension() ze zmierzoną szerokością i wysokością po ich obliczeniu. Jeśli nie wywołasz tej metody z nadpisanej metody onMeasure(), wynikiem będzie wyjątek w czasie pomiaru. Na wyższym poziomie implementacja metody onMeasure() powinna przebiegać w następujący sposób: Zastąpiona metoda onMeasure() jest wywoływana ze szczegółami miary szerokości i wysokości (parametry widthMeasureSpec i heightMeasureSpec, oba są kodami całkowitymi reprezentującymi wymiary), które należy traktować jako wymagania dla ograniczenia dotyczące pomiarów szerokości i wysokości, które należy zestawić. Metoda onMeasure() Twojego komponentu powinna oszacować szerokość i wysokość pomiaru, które będą wymagane do renderowania komponentu. Powinien starać się pozostać w ramach przekazanych specyfikacji, chociaż potencjalnie może je przekroczyć (w tym przypadku rodzic może zdecydować, co zrobić, w tym przycinać, przewijać, rzucać wyjątek lub prosić onMeasure(), aby spróbował ponownie, używając różne specyfikacje pomiarowe). Po ustawieniu szerokości i wysokości należy wywołać metodę setMeasured Dimension(int width, int height) z obliczonymi pomiarami. Jeśli ten krok zostanie pominięty, może zostać zgłoszony wyjątek. Podsumowanie niektórych standardowych metod, które framework wywołuje w widokach, znajduje się w poniższej tabeli

Metody standardowe i ich definicje

Kategoria : Metody : Opis

Tworzenie : Konstruktory : forma konstruktora jest wywoływana, gdy widok jest tworzony z kodu, a formularz jest wywoływany, gdy widok jest powiększany z pliku układu. Drugi formularz powinien przeanalizować i zastosować wszystkie atrybuty zdefiniowane w pliku układu.

Układ :

onFinishInflate() : Wywoływana po tym, jak widok i wszystkie jego elementy potomne zostały napełnione z XML.

onMeasure(int, int) : Wywoływany w celu określenia wymagań dotyczących rozmiaru tego widoku i wszystkich jego elementów podrzędnych.

onLayout(boolean, int, int, int, int) : Wywoływany, gdy ten widok powinien przypisać rozmiar i pozycję wszystkim swoim dzieciom.

onSizeChanged(int, int, int, int) : Wywoływane w przypadku zmiany rozmiaru tego widoku.

Rysunek :

onDraw(Canvas) : Wywoływane, gdy widok powinien renderować swoją zawartość.

Przetwarzanie zdarzeń :

onKeyDown(int, KeyEvent) : Wywoływane, gdy wystąpi nowe zdarzenie związane z kluczem.

onKeyUp(int, KeyEvent) : Wywoływane, gdy wystąpi zdarzenie związane z kluczem.

onTrackballEvent(MotionEvent) : wywoływane, gdy wystąpi zdarzenie ruchu trackballa.

onTouchEvent(MotionEvent) : Wywoływane, gdy wystąpi zdarzenie ruchu na ekranie dotykowym.

Focus :

onFocusChanged(boolean, int, Rect) : Wywoływane, gdy widok staje się lub traci ostrość.

onWindowFocusChanged(boolean) : Wywoływane, gdy okno zawierające widok staje się lub traci ostrość.

Dołączanie :

onAttachedToWindow() : Wywoływana, gdy widok jest dołączony do okna.

onDetachedFromWindow() : Wywoływana, gdy widok jest odłączony od swojego okna.

onWindowVisibilityChanged(int) : Wywoływane, gdy zmieniła się widoczność okna zawierającego widok.

Jeśli nie chcesz tworzyć całkowicie dostosowanego komponentu, ale zamiast tego chcesz połączyć komponent wielokrotnego użytku, który zawiera grupę istniejących kontrolek, utworzenie komponentu złożonego (lub kontroli złożonej) może być idealną opcją. Mówiąc prościej, łączy to szereg bardziej niezależnych kontrolek (lub widoków) w logiczną grupę elementów, które mogą być wykonywane jako jedna rzecz. Aby to zilustrować, Combo Box może być traktowany jako kombinacja jednowierszowego pola EditText z dołączoną PopupList. Jeśli naciśniesz przycisk i wybierzesz coś z listy, wypełni on pole EditText, ale także pozwoli użytkownikowi wpisać coś bezpośrednio w EditText. W systemie Android są w rzeczywistości dostępne dwa inne widoki: Spinner i AutoCompleteTextView, ale

niezależnie od tego koncepcja pola kombi wydaje się być najprostszym przykładem w tym przypadku. Aby utworzyć komponent złożony: Zwykle punktem wyjścia jest Layout, więc powinieneś stworzyć klasę, która rozszerza Layout. Być może w przypadku pola kombi możesz użyć LinearLayout z orientacją poziomą. Jednocześnie, ponieważ inne układy mogą być zagnieżdżone w środku, komponent złożony może być dowolnie złożony i nadmiernie ustrukturyzowany. Jednak podobnie jak w przypadku działania, możesz użyć metody deklaratywnej (opartej na XML) tworzenia zawartych komponentów lub możesz zagnieździć je programowo z kodu. W konstruktorze nowej klasy możesz wziąć dowolne parametry wymagane przez nadklasę i najpierw przekazać je do konstruktora nadklasy. Następnie powinieneś skonfigurować inne widoki do użycia w nowym komponencie, w którym utworzysz pole EditText i PopupList. Tutaj możesz również wprowadzić własne atrybuty i parametry do XML, które mogą być później wywoływane i używane przez Twój konstruktor. Ponadto dozwolone jest również tworzenie detektorów zdarzeń, które zawarte w nim widoki mogą generować, na przykład metoda detektora dla elementu listy Kliknij Listener, aby zaktualizować zawartość EditText, jeśli zostanie dokonany wybór listy. Podsumowując, wykorzystanie Układu jako podstawy Kontroli niestandardowej ma szereg zalet, w tym:

• Możesz określić układ za pomocą deklaratywnych plików XML podobnych do ekranu czynności lub możesz tworzyć widoki programowo i zagnieżdżać je w układzie z kodu.

• Metody onDraw() i onMeasure() najprawdopodobniej będą miały odpowiednią funkcjonalność, więc nie będziesz musiał ich modyfikować.

• Nareszcie można bardzo szybko zaprojektować dowolnie złożone widoki złożone i zastosować je później, tak jakby były pojedynczym komponentem.

Dobrze zaprojektowany widok niestandardowy należy traktować jak każdą inną dobrze zaprojektowaną klasę. Łączy w sobie określony zestaw funkcji z przyjaznym interfejsem użytkownika i dużym rozmiarem pamięci. Oprócz tego, że jest dobrze zaprojektowaną klasą, oczekuje się, że widok niestandardowy będzie: był zgodny ze standardami systemu Android, zapewniał niestandardowe atrybuty współpracujące z układami XML systemu Android, wysyłał zdarzenia ułatwień dostępu i był zgodny z wieloma platformami systemu Android. Platforma Android udostępnia zestaw klas bazowych i tagów XML, które zachęcają do tworzenia widoku, który spełnia wszystkie te wymagania. Porozmawiajmy teraz, jak wykorzystać platformę Android do tworzenia podstawowych funkcji widoku podklas.

Podklasa View

Wszystkie klasy widoku zawarte w strukturze systemu Android rozszerzają widok. Twój widok niestandardowy może również bezpośrednio rozszerzyć widok lub możesz zaoszczędzić zasoby, rozszerzając jedną z istniejących podklas widoku, na przykład Przycisk. Aby Android Studio odpowiadało Twojemu widokowi, powinieneś przynajmniej dostarczyć konstruktor, który jako parametry przyjmuje obiekt Context i obiekt AttributeSet. Ten onstructor umożliwia edytorowi układu tworzenie i edycję wystąpienia Twojego widoku, na przykład class PieChart(context: Context, attrs: AttributeSet): View(context, attrs).

Zdefiniuj atrybuty niestandardowe

Aby dodać wbudowany widok do interfejsu użytkownika, musisz określić go w elemencie XML i regulować jego wygląd i funkcjonalność za pomocą atrybutów elementu. Dobrze napisane widoki niestandardowe można również dodawać i stylizować za pomocą XML. Aby włączyć tę funkcję w widoku niestandardowym, należy:

• Zdefiniuj niestandardowe atrybuty dla swojego widoku w elemencie zasobów < declare-styleable >

• Zidentyfikuj wartości atrybutów w układzie XML

• Pobieranie wartości atrybutów w czasie wykonywania

• Zastosuj pobrane wartości atrybutów do swojego widoku

Aby zdefiniować atrybuty niestandardowe, wstaw zasoby <declare-styleable> do projektu. Zazwyczaj umieszcza się te zasoby w pliku res/values/attrs.xml. Spójrz na ten przykład pliku attrs.xml:

< resources >

< declare-styleable name="PieChart" >

< attr name="showText" format="boolean" / >

< attr name="labelPosition" format="enum" >

<enum name="left" value="0"/ >

< enum name="right" value="1"/ >

< /attr >

< /declare-styleable >

< /resources >

Ten kod definiuje dwa główne atrybuty, showText i labelPosition, które należą do stylizowanej jednostki o nazwie PieChart. Nazwa encji z możliwością stylizacji jest taka sama, jak tytuł klasy, która definiuje widok niestandardowy i chociaż nie jest konieczne przestrzeganie tego układu, wiele popularnych edytorów kodu nadal używa tej konwencji nazewnictwa, aby umożliwić uzupełnianie instrukcji. Po zdefiniowaniu atrybutów niestandardowych można ich używać w plikach XML układu, tak jak atrybutów wbudowanych. Jedyną różnicą jest to, że Twoje niestandardowe atrybuty będą znajdować się w innej przestrzeni nazw. Zamiast pozostać w przestrzeni nazw http://schemas.android.com/apk/res/android, byliby pod adresem http://schemas.android.com/apk/res/[nazwa pakietu]. Aby to zademonstrować, spójrzmy na te atrybuty wykresu kołowego:

< ?xml version="1.0" encoding="utf-8"? >

< LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"

xmlns:custom="http://schemas.android.com/apk/res/com.example.customviews" >

< com.example.customviews.charting.PieChart

custom:showText="true"

custom:labelPosition="left" / >

< /LinearLayout >

Jeśli chcesz uniknąć konieczności powtarzania długiego identyfikatora URI przestrzeni nazw, możesz skorzystać z dyrektywy xmlns. Ta dyrektywa przypisuje niestandardowy alias do przestrzeni nazw http://schemas.android.com/apk/res/com.example.customviews i umożliwia wybranie dowolnego aliasu, którego potrzebujesz dla swojej przestrzeni nazw. Zobacz także nazwę tagu XML, który zawiera

niestandardowy widok układu. Jest to w pełni kwalifikowana nazwa niestandardowej klasy widoku. A jeśli twoja klasa widoku jest klasą wewnętrzną, powinieneś ją dalej zakwalifikować nazwą klasy zewnętrznej widoku. Na przykład powyższa klasa PieChart ma klasę wewnętrzną o nazwie PieView. Aby użyć atrybutów niestandardowych z tej klasy, należy użyć tagu com.example.custom views.charting.PieChart$PieView.

Zastosuj niestandardowe atrybuty

Gdy widok jest tworzony z układu XML, wszystkie atrybuty w znaczniku XML są pobierane z pakietu zasobów i kierowane do konstruktora widoku jako AttributeSet. Mimo że możliwe jest odczytywanie wartości bezpośrednio z AttributeSet, ma to pewne wady, takie jak odwołania do zasobów w wartościach atrybutów, które nie zostaną w pełni rozwiązane, a style nie są zwykle do niego stosowane. Zamiast tego możesz przekazać AttributeSet do uzyskaniaStyledAttributes(). Ta metoda dostarczy tablicę TypedArray wartości, które zostały już wyłuskane i oznaczone stylem. Kompilator zasobów systemu Android w rzeczywistości wykonuje dużo pracy, aby łatwiej uzyskać dostęp do metody gainStyledAttributes(). Tak więc dla każdego zasobu <deklaruj-stylizowanego> w katalogu res wygenerowany R.java określa zarówno tablicę identyfikatorów atrybutów, jak i zestaw stałych, które definiują indeks dla każdego atrybutu w tablicy. Możesz użyć predefiniowanych stałych, aby odczytać atrybuty z TypedArray. Oto jak wspomniana wyżej klasa PieChart odczytuje swoje atrybuty:

```
init {

context.theme.obtainStyledAttributes(

attrs,

R.styleable.PieChart,

0, 0).apply {

try {

mShowText = getBoolean

(R.styleable.PieChart_showText, false)

textPos = getInteger

(R.styleable.PieChart_labelPosition, 0)

} finally {

recycle()

}

}

}
```

Ponadto należy pamiętać, że ponieważ obiekty TypedArray są zasobem udostępnionym, po użyciu należy je poddać recyklingowi.

Dodaj właściwości i wydarzenia

Atrybuty to świetny sposób na regulowanie zachowania i wyglądu widoków, ale można je odczytać tylko wtedy, gdy widok jest inicjowany. Aby ustalić dynamiczną tendencję, możesz spróbować

udostępnić parę pobierającą i ustawiającą właściwości dla każdego atrybutu niestandardowego. Poniższy fragment ilustruje, w jaki sposób PieChart udostępnia właściwość o nazwie showText:

```
fun isShowText(): Boolean {

return mShowText

}

fun setShowText(showText: Boolean) {

mShowText = showText

invalidate()

requestLayout()

}
```

Tutaj pamiętaj, że setShowText wywołuje zarówno funkcje invalidate(), jak i requestLayout(). Wezwania te są ważne, aby zapewnić stabilność widoku. Może zaistnieć potrzeba unieważnienia widoku po wprowadzeniu jakichkolwiek zmian w jego właściwościach, które mogą zmienić jego wygląd, aby system otrzymał sygnał, że należy go przerysować. Podobnie musisz poprosić o nowy układ, jeśli właściwość wprowadza zmiany, które mogą wpłynąć na rozmiar lub kształt widoku. Zapomnienie tych wywołań metod może spowodować trudne do znalezienia błędy w systemie. Ponadto widoki niestandardowe powinny również obsługiwać detektory zdarzeń w celu przesyłania ważnych zdarzeń. Na przykład PieChart uwidacznia zdarzenie niestandardowe o nazwie OnCurrentItemChanged, aby powiadomić odbiorniki, że użytkownik zmodyfikował wykres kołowy, aby skupić się na nowym wycinku kołowym. Łatwo zapomnieć o ujawnieniu właściwości i zdarzeń, zwłaszcza jeśli jesteś jedynym użytkownikiem widoku niestandardowego. Jednak poświęcenie trochę czasu na dokładne określenie interfejsu widoku znacznie zmniejszy wszelkie przyszłe koszty utrzymania. Dobrą radą, którą należy wziąć pod uwagę, jest zawsze eksponowanie każdej właściwości, która wpływa na widoczny wygląd lub zachowanie niestandardowego widoku.

Projekt dla dostępności

Ważne jest również, aby niestandardowy widok obsługiwał jak najszerszy zakres użytkowników. Dotyczy to również użytkowników z niepełnosprawnościami, które uniemożliwiają im zobaczenie lub korzystanie z ekranu dotykowego. W celu wsparcia użytkowników niepełnosprawnych zaleca się:

- Oznacz pola wejściowe za pomocą atrybutu android:contentDescription
- Wysyłaj zdarzenia ułatwień dostępu, wywołując sendAccessibilityEvent() w razie potrzeby
- Obsługa alternatywnych kontrolerów, takich jak D-pad i trackball

Przy tak dobrze zaprojektowanym widoku, który reaguje na gesty i przejścia między obiektami, ważne jest również, aby widok przebiegał szybko i płynnie. Aby uniknąć interfejsu użytkownika, który wydaje się powolny i powolny podczas odtwarzania, upewnij się, że animacje działają stale z co najmniej 60 klatkami na sekundę. Aby przyspieszyć przeglądanie, może być konieczne wyeliminowanie niepotrzebnego kodu z często używanych podprogramów. Zacznij od działania na onDraw(), co przyniesie Ci największy zwrot. W szczególności powinieneś usunąć alokacje w onDraw(), ponieważ alokacje mogą skutkować niepotrzebnym wyrzucaniem śmieci, które powodowałyby przerwy.

Pomocne byłoby również przydzielanie obiektów podczas inicjalizacji lub między animacjami. Ponadto uważa się, że złą praktyką jest dokonywanie alokacji, gdy w tym samym czasie działa animacja. Inną bardzo kosztowną pod względem zasobów eksploatacją jest przechodzenie przez układy. Za każdym razem, gdy widok wywołuje requestLayout(), system interfejsu użytkownika systemu Android musi przejrzeć całą hierarchię widoków, aby dowiedzieć się, jak duży musi być każdy widok. Jeśli znajdzie sprzeczne pomiary, wielokrotnie przejdzie przez hierarchię. Dlatego projektanci interfejsu użytkownika mają tendencję do tworzenia głębokich hierarchii zagnieżdżonych obiektów ViewGroup, aby interfejs użytkownika zachowywał się prawidłowo. Pamiętaj jednak, aby nie tworzyć zbyt głębokich hierarchii widoków, ponieważ może to spowodować problemy z wydajnością. Podsumowując, w tym rozdziale omówiliśmy podstawy interfejsu użytkownika Android Studio i dowiedzieliśmy się, jak korzystać z jego głównych komponentów i widżetów. W szczególności ustaliliśmy, w jaki sposób możesz zarządzać folderami, układami, ciągami i widokami na platformie. W następnym rozdziale skupimy się na wykorzystaniu kluczowych narzędzi Android Studio, takich jak SDK Manager, AVD Manager i Navigation Editor.