Android

Hierarchie widoków i ich wyświetlanie

Piotr Fulmański

Instytut Nauk Ekonomicznych i Informatyki,
Państwowa Wyższa Szkoła Zawodowa w Płocku, Polska

March 14, 2016

Table of contents

Co w tym wykładzie

- xxx

Hierarchie widoków i ich wyświetlanie

Widoki w Androidzie są zwykle definiowane deklaratywnie w formacie XML. Wewnętrznie **interfejs użytkownika każdej aplikacji na Android jest reprezentowany jako drzewo obiektów klasy View**. Jest to tak zwana hierarchia widoków lub drzewo widoków. Korzeń każdego drzewa widoków (i interfejsu użytkownika każdej aplikacji) to jeden obiekt klasy `DecorView`. Jest to wewnętrzna klasa frameworku, której nie można bezpośrednio używać. Reprezentuje ona bieżące widoczne okno urządzenia.

Układ a menedżer układu

Kiedy mówimy o układzie, mamy na myśli zestaw wszystkich widoków jednej aktywności uporządkowanych za pomocą pliku XML układu. Plik ten znajduje się w katalogu `res/layout`. Nie należy mylić takich układów z konkretnymi klasami układu, tak zwanymi menedżerami układu. Układ aktywności (w zależności od stopnia skomplikowania) może obejmować wiele menedżerów układu. Menedżer układu jest także obiektem klasy `View` (a dokładniej – klasy `ViewGroup`). Pełni funkcję kontenera i porządkuje widoki w określony sposób.

Atrybuty i parametry układu

Każdy układ w widoku może mieć atrybuty dwóch rodzajów – właściwe dla klasy widoku i klas bazowych oraz właściwe dla menedżera układu, w którym dany widok się znajduje.
Atrybuty dostępne dla widoku można znaleźć w jego dokumentacji.
Parametry układu wyglądają inaczej. Można odróżnić je od zwykłych z uwagi na przedrostek `layout_`. Parametry określają, w jaki sposób widok należy wyświetlić w układzie.

Marginesy zewnętrzne i wewnętrzne

Parametry określające szerokość i wysokość muszą występować w każdym widoku. Przyjmują nie tylko wartości liczbowe (w pikselach), ale też dwie zarezerwowane wartości:

- `fill_parent` . Określa, że widok zajmuje maksymalny dostępny obszar w widoku nadrzędnym. Próbuje wypełnić obszar widoku nadrzędnego (z pominięciem marginesów wewnętrznych i zewnętrznych) niezależnie od miejsca zajmowanego przez własne elementy podrzędne.
- `wrap_content`. Określa, że widok zajmuje w widoku nadrzędnym tyle miejsca, ile potrzeba na wyświetlenie całej zawartości.

Jest to najprostszy z wszystkich menedżerów układu. `FrameLayout` nie wykonuje żadnych operacji na układzie i pełni tylko funkcję kontenera (ramki; ang. *frame*). Menedżer `FrameLayout` wyświetla jeden element podrzędny naraz. Obsługuje wiele elementów podrzędnych, umieszczanych jeden na drugim. Elementy podrzędne są wyrównywane do lewego górnego rogu i wyświetlane jeden na drugim zgodnie z kolejnością ich deklaracji.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/a
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView
      android:layout_width="150px"
      android:layout_height="150px"
      android:background="@android:color/darker_gray"
   />
   <TextView
      android:layout_width="75px"
      android:layout_height="75px"
      android:background="@android:color/white"
   />
</FrameLayout>
```

`LinearLayout` to najczęściej stosowany menedżer układu. Jest prosty, łatwy w użyciu i pełni wiele funkcji. W menedżerze `LinearLayout` wszystkie widoki są wyświetlane w wierszach lub kolumnach (zależy to od wartości atrybutu `android:orientation`). Jeśli nie określono bezpośrednio orientacji, elementy domyślnie rozmieszczane są w poziomie. Menedżer `LinearLayout` ma dwa dodatkowe parametry układu, które można stosować do jego elementów podrzędnych.

- `android:layout_weight` Informuje menedżer układu o tym, ile miejsca powinien zajmować widok względem siostrzanych widoków. Wielkość widoku jest określana na podstawie stosunków wag widoków. Jeśli na przykład wszystkie widoki mają tę samą wagę, dostępny obszar jest dzielony między nie po równo. To, którego wymiaru mają dotyczyć wagi (szerokości czy wysokości), można kontrolować przez ustawienie odpowiedniej wartości dla danego wymiaru na `0px`. Warto zauważyć, że wagi nie muszą sumować się do 1, choć standardowo są rozdzielane między wszystkie elementy podrzędne jako ułamki liczby 1 (semantyka oparta na procentach). Istotne są stosunki wartości przypisanych poszczególnym widokom.
- `android:layout_gravity` Informuje menedżer układu, w którym kierunku widoki powinny być rozmieszczane w kontenerze. Atrybut ten ma znaczenie tylko wtedy, kiedy wielkość widoku w danym wymiarze jest albo ściśle określona, albo ma wartość `wrap_content`.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView
      android:layout_width="0px"
      android:layout_height="100px"
      android:layout_weight="0.5"
      android:background="@android:color/darker_gray"
   />
   <TextView
      android:layout_width="0px"
      android:layout_height="100px"
      android:layout_weight="0.5"
      android:background="@android:color/white"
   />
</LinearLayout>
```

Klasa `TableLayout` jest klasą typu `LinearLayout` (dziedziczy po niej) wzbogaconą o mechanizmy przydatne do wyświetlania tabel lub siatek. Zastosowano tu specjalną klasę widoku, `TableRow`, pełniącą funkcję kontenera na komórki tabeli. Każda komórka może obejmować tylko jeden widok. Widok ten może być oparty na menedżerze układu lub dowolnym innym obiekcie klasy `ViewGroup`.

```
<?xml version="1.0" encoding="utf-8"?>
   <TableLayout xmlns:android="http://schemas.android.com/apk/re
      android:layout_width="fill_parent"
      android:layout_height="fill_parent">
   <TableRow>
      <TextView
         android:layout_width="150px"
         android:layout_height="100px"
         android:background="@android:color/darker_gray"
      />
      <TextView
         android:layout_width="150px"
         android:layout_height="100px"
         android:background="@android:color/white"
      />
   </TableRow>
</TableLayout>
```

`RelativeLayout` to najbardziej zaawansowany spośród czterech menedżerów układu. Umożliwia niemal dowolne rozmieszczanie widoków przez uporządkowanie ich względem siebie. Menedżer `RelativeLayout` udostępnia parametry pozwalające elementom podrzędnym wskazywać siebie nawzajem za pośrednictwem identyfikatorów (notacja z wykorzystaniem `@id`).

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/re
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView android:id="@+id/text_view_1"
      android:layout_width="150px"
      android:layout_height="100px"
      android:background="@android:color/darker_gray"
   />
   <TextView android:id="@+id/text_view_2"
      android:layout_width="150px"
      android:layout_height="100px"
      android:layout_toRightOf="@id/text_view_1"
      android:layout_centerVertical="true"
      android:background="@android:color/white"
   />
</RelativeLayout>
```

Scalanie i dołączanie układów

Zamiast tak

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView android:text="Aktywność z paskiem przycisków"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
   />
   <!-- Pasek przycisków -->
   <LinearLayout android:orientation="horizontal"
      android:layout_width="fill_parent"
      android:layout_height="wrap_content">
      <Button android:text="@string/ok"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
      />
      <Button android:text="@string/cancel"
         android:layout_width="wrap_content"
```

Scalanie i dołączanie układów

Można tak: stworzyć najpierw przeznaczony do wielokrotnego użytku komponent paska przycisków zdefiniowany w odrębnym pliku układu (button_bar)

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android:layout_width="fill_parent"
   android:layout_height="wrap_content">
   <Button android:text="@string/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
   />
   <Button android:text="@string/cancel"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
   />
</LinearLayout>
```

a następnie dołączyć ten fragment lub komponent układu do innego pliku używając elementu <include>.

```
<include layout="@layout/button_bar" />
```

Scalanie i dołączanie układów

Najlepiej tak

```
<?xml version="1.0" encoding="utf-8"?>
<merge xmlns:android="http://schemas.android.com/apk/res/android
   <Button android:text="@android:string/ok"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
   />
   <Button android:text="@android:string/cancel"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
   />
</merge>
```

Pozwala to dołączyć zestaw komponentów do dowolnego układu w zależności od potrzeb. Nie mamy tutaj, jak poprzednio, żadnego układu (poprzednio `LinearLayout`), ale za to jedynie ogólny element `merge` w celu zapewnienie zgodności z XML (bez tego nie otrzymamy poprawnego dokumentu XML).

Zamiast tak

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/andr
   android:text="Witaj, widoku tekstowy!"
   android:textSize="14sp"
   android:textStyle="bold"
   android:textColor="#CCC"
   android:background="@android:color/transparent"
   android:padding="5dip"
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
/>
```

co jest dobre gdy mamy tylko jeden przycisk.

Można zdefiniować ogólny styl

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <style name="MyCustomTextView" parent="@android:style/Widget.
        <item name="android:textSize">14sp</item>
        <item name="android:textStyle">bold</item>
        <item name="android:textColor">#CCC</item>
        <item name="android:background">@android:color/transparent
        <item name="android:padding">5dip</item>
    </style>
</resources>
```

a następnie go użyć

```
<?xml version="1.0" encoding="utf-8"?>
<TextView xmlns:android="http://schemas.android.com/apk/res/andr
    android:text="Witaj, widoku tekstowy!"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    style="@style/MyCustomTextView"
/>
```

Motywy

Łączenie atrybutów widoków w style jest przydatne, ale to tylko połowa rozwiązania. Nadal trzeba zastosować style do wszystkich docelowych widoków. Powinno się to odbywać automatycznie.
Motywy to style. Jedyna różnica między motywem a stylem polega na tym, że motywy dotyczą aktywności lub całej aplikacji (czyli wszystkich aktywności z programu), a nie pojedynczych widoków. Różnica związana jest więc z zakresem, a nie ze sposobem działania lub nawet strukturą rozwiązania.

Motywy

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <style name="MyAppTheme"
      parent="@android:style/Theme.Black">
      <item name="android:listViewStyle">@style/MyListViewStyle<
      <item name="android:windowBackground">@drawable/film_bg</i
   </style>
   <style name=" MyListViewStyle"
      parent="@android:style/Widget.ListView">
      <item name="android:background">#A000</item>
      <item name="android:fastScrollEnabled">true</item>
      <item name="android:footerDividersEnabled">false</item>
   </style>
</resources>

<application android:theme="@style/MyAppTheme" ...>
...
</application>
```