Joseph Annuzzi Jr.
Lauren Darcey
Shane Conder

Android™

Wprowadzenie do programowania aplikacji

Wydanie V

Helion

Tytuł oryginału: Introduction to Android Application Development: Android Essentials, Fifth Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-2612-5

Authorized translation from the English language edition, entitled: Introduction TO ANDROID APPLICATION DEVELOPMENT: ANDROID ESSENTIALS, Fifth Edition; ISBN 013438945X; by Joseph Annuzzi; and by Lauren Darcey; and by Shane Conder; published by Pearson Education, Inc, publishing as Addison-Wesley Professional.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 32 231 22 19, 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *http://helion.pl* (księgarnia internetowa, katalog książek)

Spis treści

9

Dzielenie interfejsu aplikacji z użyciem fragmentów

Tradycyjnie każdy ekran w aplikacji przeznaczonej na Androida był skojarzony z jedną, konkretną aktywnością. Niemniej jednak w systemie Android 3.0 (Honeycomb) wprowadzono pojęcie fragmentu (reprezentowanego przez klasę `Fragment`). Następnie obsługę fragmentów dodano do biblioteki wsparcia systemu Android, tak by można ich było używać w Androidzie, zaczynając od wersji 1.6 (API poziomu 4). Fragmenty zapewniają separację komponentów interfejsu użytkownika lub zachowań (zarówno tych dysponujących interfejsem użytkownika, jak i tych, które są go pozbawione) od cyklu życia konkretnej aktywności. Z kolei klasy aktywności mogą dowolnie łączyć i stosować komponenty interfejsu użytkownika bądź zachowania, tworząc z ich pomocą znacznie bardziej elastyczny interfejs użytkownika. W tym rozdziale zostanie wyjaśnione, czym są fragmenty oraz jak można je tworzyć i stosować. Przedstawione zostanie także zagadnienie fragmentów zagnieżdżonych.

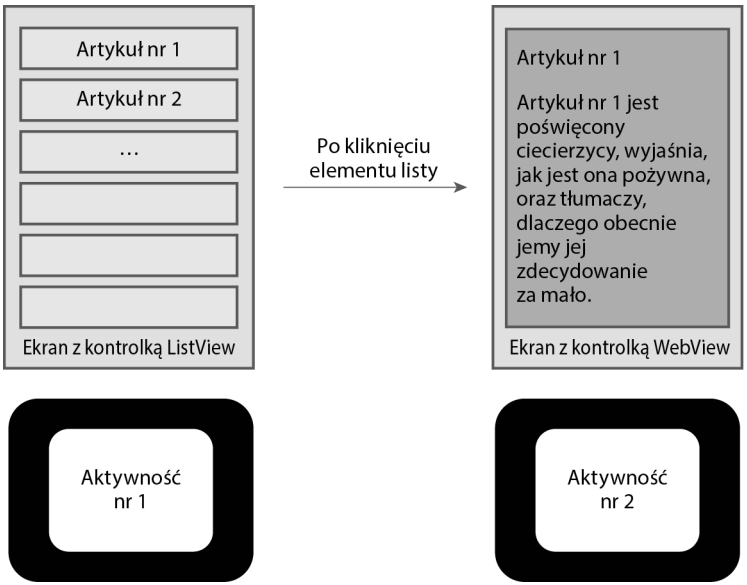
Wyjaśnienie pojęcia fragmentu

Fragmenty zostały dodane do Android SDK w kluczowym momencie, kiedy na rynku ogromnie wzrosły popularność i liczba dostępnych urządzeń z systemem Android. Obecnie stosowane są nie tylko smartfony, lecz także urządzenia z większymi ekranami, takie jak tablety i telewizory, także działające pod kontrolą Androida. Urządzenia te dysponują znacznie większą powierzchnią ekranu, którą mogą wykorzystać programiści. Na przykład typowy, dopracowany i elegancki interfejs użytkownika aplikacji, przygotowany z myślą o smartfonach, będzie na tabletach zazwyczaj wyglądał na zbytnio uproszczony. Jednak wprowadzając do projektu interfejsu użytkownika komponenty fragmentów, można napisać jedną aplikację, która będzie dostosowana do wszystkich tych cech charakterystycznych i orientacji ekranu, a nie do wielu wersji aplikacji przystosowanych do konkretnych typów urządzeń. Takie rozwiązanie znacznie zwiększa możliwości wielokrotnego stosowania kodu, upraszcza proces testowania aplikacji oraz ułatwia publikowanie pakietu aplikacji oraz zarządzanie nim.

Zgodnie z tym, co podano we wstępie do tego rozdziału, jedną z podstawowych reguł, których trzymali się programiści aplikacji przeznaczonych na Androida, było tworzenie jednej aktywności obsługującej jeden, konkretny ekran aplikacji. Rozwiązanie to w bardzo ścisły i bezpośredni sposób

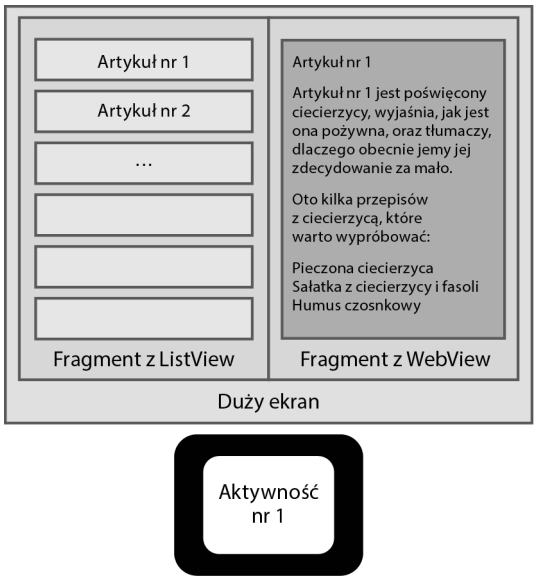
kojarzy zadania realizowane przez klasę aktywności z interfejsem użytkownika. Jednak okazało się, że w momencie, gdy na rynku pojawiły się urządzenia z większymi ekranami, takie rozwiązanie zaczęło przysparzać problemów. Kiedy na jednym ekranie pojawiło się więcej miejsca, umożliwiającego wykonywanie większej liczby czynności, konieczne stało się implementowanie odrębnych klas aktywności, przeznaczonych do obsługi tych rozbudowanych ekranów. Niemniej jednak funkcje owych prostszych oraz rozbudowanych aktywności w przeważającej mierze były identyczne. Problem ten można rozwiązać dzięki zastosowaniu fragmentów, gdyż pozwalają one hermetyzować funkcje udostępniane w obszarach ekranu w formie komponentów, których można wielokrotnie używać i, w ramach tej samej aktywności, łączyć z innymi fragmentami.

Poniżej przeanalizowano teoretyczny przykład aplikacji. Jest to tradycyjna aplikacja na smartfony składająca się z dwóch ekranów, przeznaczona na przykład do prezentowania najnowszych informacji z witryny internetowej. Jej pierwszy ekran tworzy aktywność typu `ListActivity` zawierająca kontrolkę `ListView`. Każdy element tej listy `ListView` reprezentuje artykuł z witryny, który użytkownik może przeczytać. Ponieważ jest to aplikacja internetowa, zatem kliknięcie konkretnego artykułu powoduje przejście na drugi ekran, na którym treść wybranego artykułu jest prezentowana w kontrolce `WebView`. Ta tradycyjna architektura aplikacji internetowej została przedstawiona na rysunku 9.1.



Rysunek 9.1. Tradycyjne powiązania pomiędzy ekranami w aplikacji, która nie korzysta z fragmentów

Taka architektura interfejsu użytkownika aplikacji sprawdza się na niewielkich smartfonach, jednak na dużych tabletach lub telewizorach będzie przykładem marnowania dostępnego miejsca. W przypadku tych większych urządzeń należałoby raczej umieścić listę artykułów oraz treść wybranego artykułu na tym samym ekranie. Gdyby kontrolki `ListView` oraz `WebView` umieścić w dwóch niezależnych fragmentach, bez trudu można by zdefiniować układ, który prezentowałby je na tym samym ekranie wówczas, gdyby było na nim dostatecznie dużo miejsca. Teoretyczną postać takiej aplikacji przedstawiono na rysunku 9.2.



Rysunek 9.2. Tradycyjne powiązania pomiędzy ekranami w aplikacji, która nie korzysta z fragmentów

Opis cyklu życia obiektów Fragment

Cykl życia aktywności (obiektów `Activity`) został opisany w rozdziale 4. „Prezentacja komponentów aplikacji". W tym punkcie rozdziału zostanie opisane, jak do schematu działania aktywności można dopasować fragmenty. Przede wszystkim fragment (obiekt klasy `Fragment`) musi być umieszczony w aktywności. Fragmenty mają swój własny cykl życia, niemniej jednak nie są one niezależnymi komponentami, które mogą istnieć poza kontekstem aktywności.

W przypadku gdy cały stan interfejsu użytkownika zostaje przeniesiony do poszczególnych fragmentów, odpowiedzialność klas `Activity` ulega znacznemu ograniczeniu i uproszczeniu. Klasy `Activity`, w których układach znajdują się same fragmenty, nie muszą już tracić czasu na zapisywanie oraz odtwarzanie stanu, gdyż obiekt aktywności zarządza wszelkimi dołączonymi do niego obiektami `Fragment`. Z kolei komponenty `Fragment` same zarządzają własnym stanem, wykorzystując do tego swój własny cykl życia. Oczywiście nic nie stoi na przeszkodzie, by w klasach aktywności dowolnie łączyć fragmenty z innymi kontrolkami `View`. W takich sytuacjach klasy `Activity` będą odpowiedzialne za zarządzanie używanymi kontrolkami `View`, jak działo się to do tej pory.

Aktywności muszą się także koncentrować na zarządzaniu używanymi w nich fragmentami. Koordynację działań między aktywnością a jej fragmentami upraszcza klasa `FragmentManager` (`android.app.FragmentManager`). Obiekt `FragmentManager` można pobrać przy użyciu metody `getFragmentManager()`, dostępnej w klasach `Activity` oraz `Fragment`. W przypadku korzystania z biblioteki wsparcia koordynację tę ułatwia klasa `FragmentManager` (`android.support.v4.app.FragmentManager`), której obiekt można pobrać przy użyciu metody `getSupportFragmentManager()`, dostępnej w API klasy `FragmentActivity`.

Definiowanie fragmentów

Implementacje klasy `Fragment`, które zostały zdefiniowane w aplikacji jako normalne klasy, można dodawać do plików układów przy użyciu znacznika XML `<fragment>`, a następnie wczytywać w aktywności, stosując standardową metodę `setContentView()`, która zazwyczaj jest wywoływana w metodzie `onCreate()` aktywności.

W przypadku odwoływania się do klas `Fragment` zdefiniowanych w pakiecie aplikacji w plikach XML układów należy korzystać ze znacznika `<fragment>`. Posiada on kilka ważnych atrybutów. Przede wszystkim w atrybucie `android:name` należy podać pełną nazwę klasy `Fragment`. Oprócz tego należy przypisać fragmentowi unikalny identyfikator, używając w tym celu atrybutu `android:id`, dzięki czemu, w razie konieczności, będzie można odwołać się do fragmentu w kodzie programu. Oprócz tego, podobnie jak we wszystkich innych kontrolkach umieszczanych w układach, trzeba także określić wymiary fragmentu za pomocą atrybutów `layout_width` oraz `layout_height`. Poniżej przedstawiona została prosta, przykładowa postać znacznika `<fragment>`, odwołującego się do klasy o nazwie `VeggieGardenListFragment`, zdefiniowanej jako klasa Javy należąca do pakietu:

```
<fragment
    android:name="com.introtoandroid.simplefragments.VeggieGardenListFragment"
    android:id="@+id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```

Zarządzanie modyfikacjami fragmentów

Jak można się przekonać, w sytuacjach gdy w ramach jednej aktywności jest umieszczonych na jednym ekranie kilka fragmentów, interakcje wykonywane przez użytkownika w jednym fragmencie (takim jak lista doniesień — `ListViewFragment` — przedstawionej wcześniej, przykładowej aplikacji) często będą powodować, że aktywność zmodyfikuje inny fragment (taki jak `WebViewFragment` przykładowej aplikacji). Aktualizacje lub modyfikacje fragmentów są wykonywane z wykorzystaniem obiektów `FragmentTransaction` (`android.app.FragmentTransaction` lub `android.support.v4.app.FragmentTransaction`). Korzystając z czynności udostępnianych przez klasę `FragmentTransaction`, można wykonywać na fragmentach kilka różnych operacji, takich jak:

- dołączanie fragmentu do nadrzędnej aktywności lub odłączanie od niej,
- ukrywanie lub wyświetlanie fragmentu w widoku.

W tym momencie można się zacząć zastanawiać, jak wygląda współpraca między przyciskiem *Wstecz* a takim właśnie interfejsem użytkownika korzystającym z fragmentów. Okazuje się, że obecnie klasa aktywności dysponuje własnym stosem cofnięć (ang. *back stack*). Programista może obecnie zdecydować, które operacje `FragmentTransaction` warto umieścić na tym stosie przy użyciu metody `addToBackStack()` klasy `FragmentTransaction`, a które można zignorować. Na przykład w przypadku przedstawionej wcześniej hipotetycznej aplikacji operacje wyświetlania artykułu we fragmencie `WebViewFragment` mogą być dodawane do stosu cofnięć, tak by klikanie przycisku *Wstecz* powodowało wyświetlanie poprzednio oglądanych artykułów, a dopiero potem wyście z aplikacji.

Dołączanie fragmentów do aktywności i odłączanie ich od niej

Kiedy już zostanie utworzony fragment, który ma być używany w klasie aktywności, pojawi się problem zarządzania jego cyklem życia. Poniżej przedstawiono metody zwrotne służące do tego

celu; należy przy tym pamiętać, że fragmenty są tworzone wtedy, kiedy staną się potrzebne, a usuwane wtedy, gdy nie są już używane. Wiele spośród przedstawionych w tym miejscu metod odpowiada analogicznym metodom cyklu życia aktywności:

- Metoda zwrotna `onAttach()` jest wywoływana podczas pierwszego dołączania fragmentu do aktywności.
- Metoda zwrotna `onCreate()` jest wywoływana w momencie pierwszego tworzenia fragmentu.
- Metoda zwrotna `onCreateView()` jest wywoływana w chwili, gdy powinien być utworzony układ bądź hierarchia kontrolek `View` zawierająca dany fragment.
- Metoda zwrotna `onActivityCreated()` jest wywoływana w celu „poinformowania" fragmentu o zakończeniu wykonywania metody `onCreate()` nadrzędnej aktywności, do której został on dołączony.
- Metoda zwrotna `onStart()` jest wywoływana, kiedy interfejs użytkownika fragmentu został już utworzony, lecz jeszcze nie jest aktywny.
- Metoda zwrotna `onResume()` jest wywoływana, kiedy interfejs użytkownika fragmentu stanie się aktywny i gotowy do prowadzenia interakcji z użytkownikiem po wcześniejszym wznowieniu aktywności lub aktualizacji fragmentu przy użyciu obiektu `FragmentTransaction`.
- Metoda zwrotna `onPause()` jest wywoływana po wstrzymaniu wykonywania nadrzędnej aktywności lub po aktualizacji fragmentu z wykorzystaniem obiektu `FragmentTransaction`. Jej wywołanie oznacza, że fragment nie jest już dłużej aktywny bądź że jest obecnie wyświetlany w tle.
- Metoda zwrotna `onStop()` jest wywoływana po zatrzymaniu wykonywania nadrzędnej aktywności lub po aktualizacji fragmentu za pomocą obiektu `FragmentTransaction`. Jej wywołanie oznacza, że fragment nie jest już widoczny.
- Metoda zwrotna `onDestroyView()` jest wywoływana w celu wyczyszczenia zasobów skojarzonych z danym fragmentem, używanych przez układ lub hierarchię kontrolek `View`.
- Metoda zwrotna `onDestroy()` jest wywoływana w celu wyczyszczenia wszelkich pozostałych zasobów używanych przez fragment.
- Metoda zwrotna `onDetach()` jest wywoływana bezpośrednio przed odłączeniem fragmentu od aktywności.

Stosowanie specjalnych typów fragmentów

W rozdziale 8., zatytułowanym „Umiejscawianie elementów z użyciem układów", przedstawiono kilka szczególnych wersji klasy `Activity`, przeznaczonych do zarządzania pewnymi często używanymi typami interfejsów użytkownika. Na przykład klasa `ListActivity` upraszcza tworzenie aktywności zarządzających kontrolką `ListView`. I analogicznie klasa `PreferenceActivity` ułatwia tworzenie aktywności zarządzających współdzielonymi preferencjami. A jak pokazał przedstawiony wcześniej przykład hipotetycznej aplikacji czytnika informacji, często będzie się pojawiać konieczność umieszczania kontrolek, takich jak `ListView` lub `WebView`, we fragmentach.

Ponieważ fragmenty zostały opracowane w celu odseparowania tej możliwości funkcjonalnej od klas aktywności, obecnie można znaleźć klasy pochodne klasy `Fragment`, odpowiadające wyspecjalizowanym klasom aktywności. Spośród nich warto dokładniej przyjrzeć się następującym klasom:

- **`ListFragment`** (**`android.app.ListFragment`**). Podobnie jak `ListActivity`, ta klasa `Fragment` ułatwia zarządzanie kontrolką `ListView`.
- **`PreferenceFragment`** (**`android.preference.PreferenceFragment`**). Podobnie jak `PreferenceActivity`, ta klasa `Fragment` ułatwia zarządzanie preferencjami użytkownika.
- **`WebViewFragment`** (**`android.webkit.WebViewFragment`**). Ten typ fragmentu zawiera kontrolkę `WebView`, dzięki czemu bez trudu można w nim wyświetlać treści z internetu. Aby aplikacja miała dostęp do internetu, wciąż będzie jej potrzebne uprawnienie `android.permission.INTERNET`.
- **`DialogFragment`** (**`android.app.DialogFragment`**). Odseparowanie możliwości funkcjonalnych związanych z interfejsem użytkownika od aktywności oznacza także, że obecnie nie jest pożądane także zarządzanie oknami dialogowymi z poziomu aktywności. Zamiast tego klasa ta pozwala na gromadzenie i zarządzanie kontrolkami w oknie dialogowym w sposób charakterystyczny dla fragmentów. Zagadnienia związane z oknami dialogowymi zostały opisane w rozdziale 10. „Określanie architektury z użyciem wzorców”.

> **Uwaga**
>
> Pełną listę wyspecjalizowanych rodzajów fragmentów można znaleźć w dokumentacji klasy `Fragment`, w sekcji prezentującej jej klasy pochodne, na stronie *http://d.android.com/reference/android/app/Fragment.html*.

Projektowanie aplikacji korzystających z fragmentów

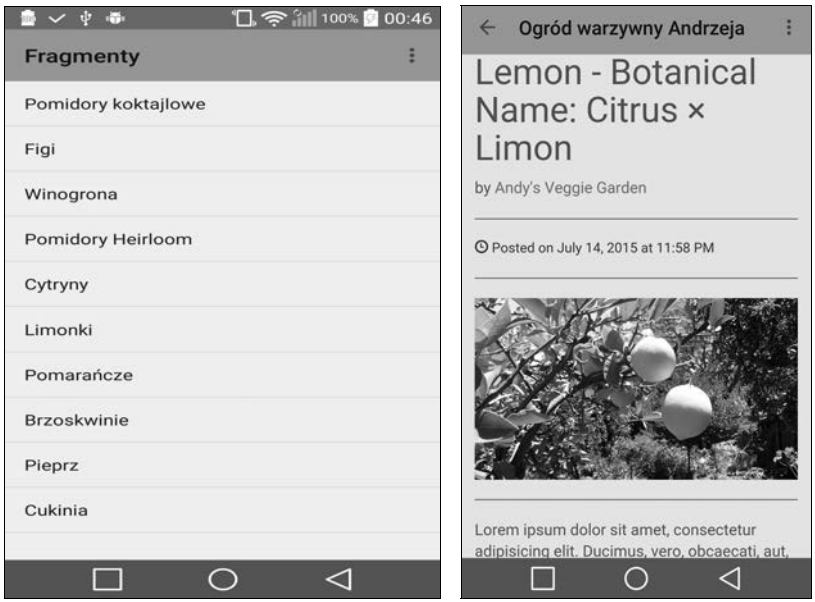
W ostatecznym rozrachunku tworzenia aplikacji korzystających z fragmentów najlepiej jest się uczyć na przykładzie. Właśnie z tego powodu w dalszej części rozdziału zostanie przedstawiony stosunkowo prosty przykład, którego zadaniem będzie wyjaśnienie oraz praktyczne przedstawienie wielu opisanych tu pojęć. W celu uproszczenia rozwiązania przykład ten będzie przeznaczony dla konkretnej wersji platformy Android, a ściśle: Android Marshmallow. Niemiej jednak niebawem okaże się, że dzięki zastosowaniu bibliotek wsparcia aplikacje korzystające z fragmentów można tworzyć na niemal wszystkie urządzenia działające pod kontrolą Androida.

> **Wskazówka**
>
> Wiele fragmentów kodu przedstawionych w tym rozdziale pochodzi z aplikacji `SimpleFragments`. Jej kod źródłowy jest dostępny w przykładach dołączonych do książki, które można pobrać z serwera FTP wydawnictwa Helion pod adresem *ftp://ftp.helion.pl/przyklady/anwpp5.zip*.

Robi (fikcyjny robot) jest wielkim miłośnikiem prac ogrodowych i w swoim ogródku hoduje wiele owoców i warzyw. Przedstawiona tu przykładowa aplikacja będzie zawierać kontrolkę `ListView` prezentującą ich nazwy. Kliknięcie elementu tej listy będzie powodować wczytanie kontrolki `WebView` i wyświetlenie w niej strony poświęconej wybranej roślinie. Aby uprościć rozwiązanie, nazwy owoców i warzyw oraz adresy URL odpowiadających im stron WWW będą przechowywane w zasobach aplikacji, w tablicach łańcuchów znaków. (Pełna implementacja tej aplikacji jest dostępna w kodach dołączonych do książki).

A jak będą działać fragmenty używane w tej aplikacji? Lista zostanie zaimplementowana za pomocą fragmentu klasy `ListFragment`, natomiast za wyświetlanie stron będzie odpowiadał fragment typu `WebViewFragment`. W układzie pionowym na ekranie będzie wyświetlany tylko jeden fragment,

przez co konieczne będzie użycie dwóch klas aktywności, jak pokazano na rysunku 9.3. W przypadku tej przykładowej aplikacji aktywności zostaną zaimplementowane z wykorzystaniem klasy `AppCompatActivity` (`android.support.v7.AppCompatActivity`).



Rysunek 9.3. Sytuacja, w której każdy fragment jest umieszczony w osobnej aktywności i wyświetlany pojedynczo na ekranie

W układzie poziomym oba fragmenty będą wyświetlane na tym samym ekranie w ramach jednej aktywności `AppCompatActivity`. Tę sytuację przedstawia zrzut zamieszczony na rysunku 9.4.

Rysunek 9.4. W układzie poziomym oba fragmenty są widoczne na ekranie jednocześnie i działają w ramach jednej aktywności

Implementacja fragmentu ListFragment

W pierwszej kolejności przedstawiona zostanie implementacja klasy pochodnej klasy `ListFragment`, która w prezentowanej aplikacji będzie nosić nazwę `VeggieGardenListFragment`. Fragment ten będzie odpowiadał za wyświetlanie listy nazw warzyw i owoców. Wspomniana klasa musi określić, czy należy wczytać drugi fragment — `VeggieGardenWebViewFragment` — czy też kliknięcie elementu listy ma powodować uruchomienie drugiej aktywności (`VeggieGardenViewActivity`):

```
public class VeggieGardenListFragment extends ListFragment implements
FragmentManager.OnBackStackChangedListener {

    private static final String DEBUG_TAG = "VeggieGardenListFragment";
    int mCurPosition = 1;
    boolean mShowTwoFragments;

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        getListView().setChoiceMode(ListView.CHOICE_MODE_SINGLE);
        String[] veggies = getResources().getStringArray(R.array.veggies_array);
        setListAdapter(new ArrayAdapter<>(getActivity(),
            android.R.layout.simple_list_item_activated_1, veggies));

        View detailsFrame = getActivity().findViewById(R.id.veggieentry);
        mShowTwoFragments = detailsFrame != null
            && detailsFrame.getVisibility() == View.VISIBLE;

        if (savedInstanceState != null) {
            mCurPosition = savedInstanceState.getInt("curChoice", 0);
        }

        if (mShowTwoFragments == true || mCurPosition != 1) {
            viewVeggieInfo(mCurPosition);
        }

        getFragmentManager().addOnBackStackChangedListener(this);
    }

    @Override
    public void onBackStackChanged() {
        VeggieGardenWebViewFragment details =
            (VeggieGardenWebViewFragment) getFragmentManager()
                .findFragmentById(R.id.veggieentry);
        if (details != null) {
            mCurPosition = details.getShownIndex();
            getListView().setItemChecked(mCurPosition, true);

            if (!mShowTwoFragments) {
                viewVeggieInfo(mCurPosition);
            }
        }
    }

    @Override
    public void onSaveInstanceState(Bundle outState) {
        super.onSaveInstanceState(outState);
        outState.putInt("curChoice", mCurPosition);
    }
```

```
    @Override
    public void onListItemClick(ListView l, View v, int position, long id) {
        viewVeggieInfo(position);
    }

    void viewVeggieInfo(int index) {
        mCurPosition = index;

        if (mShowTwoFragments == true) {
            // Sprawdzamy, który fragment jest aktualnie wyświetlany, i w razie potrzeby go zastępujemy.
            VeggieGardenWebViewFragment details =
                (VeggieGardenWebViewFragment) getFragmentManager()
                    .findFragmentById(R.id.veggieentry);
            if (details == null || details.getShownIndex() != index) {

                VeggieGardenWebViewFragment newDetails =
                    VeggieGardenWebViewFragment.newInstance(index);

                FragmentManager fm = getFragmentManager();
                FragmentTransaction ft = fm.beginTransaction();
                ft.replace(R.id.veggieentry, newDetails);
                if (index != 1) {
                    String[] veggies =
                        getResources().getStringArray(R.array.veggies_array);
                    String strBackStackTagName = veggies[index];
                    ft.addToBackStack(strBackStackTagName);
                }

                ft.setTransition(FragmentTransaction.TRANSIT_FRAGMENT_FADE);
                ft.commit();
            }

        } else {
            Intent intent = new Intent();
            intent.setClass(getActivity(), VeggieGardenViewActivity.class);
            intent.putExtra("index", index);
            startActivity(intent);
        }
    }
}
```

Większość operacji związanych z inicjowaniem fragmentu jest wykonywana w metodzie zwrotnej `onActivityCreated()` — dzięki jej zastosowaniu zainicjowanie kontrolki `ListView` zostanie wykonane tylko jeden raz. Następnie fragment określa, w którym trybie wyświetlania ma działać — w tym celu sprawdza, czy w układzie jest dostępny drugi fragment. Pozostałe szczegóły związane z wyświetleniem fragmentu zostają przekazane do metody pomocniczej o nazwie `viewVeggieInfo()`, która będzie wywoływana także po każdym kliknięciu elementu listy `ListView`.

Logika działania metody `viewVeggieInfo()` uwzględnia oba tryby wyświetlania aplikacji. Jeśli urządzenie działa w układzie pionowym, to przy użyciu obiektu `Intent` uruchamiana jest aktywność `VeggieGardenViewActivity`. Z kolei w układzie poziomym konieczne jest wykonanie pewnych operacji na fragmentach.

Konkretnie rzecz biorąc, za pomocą obiektu klasy `FragmentManager` i na podstawie unikalnego identyfikatora (a konkretnie `R.id.veggieentry` zdefiniowanego w pliku układu) odnajdywany jest fragment `VeggieGardenWebViewFragment`. Następnie tworzona jest nowa instancja fragmentu `VeggieGardenWebViewFragment`, w której zostanie wyświetlona strona WWW nowego, wybranego

z listy owocu lub warzywa. Kolejną operacją jest utworzenie transakcji (obiektu `FragmentTransaction`), w ramach której poprzedni fragment `VeggieGardenWebViewFragment` zostaje zastąpiony nowym. Poprzedni fragment zostaje także umieszczony na stosie cofnięć, tak by naciśnięcie przycisku *Wstecz* działało prawidłowo. Oprócz tego zostanie określona animacja przejścia pomiędzy fragmentami, po czym transakcja zostanie zatwierdzona, co spowoduje asynchroniczną aktualizację ekranu.

Dodatkowo stos cofnięć jest monitorowany przez wywołanie metody `addOnBackStackChangedListener()`. Metoda zwrotna `onBackStackChanged()` aktualizuje listę, dodając do niej obecnie wybrany element. Rozwiązanie to stanowi solidny sposób synchronizacji elementów wybieranych z listy `ListView` z zawartością wyświetlaną we fragmencie, i to zarówno w przypadku dodawania nowego fragmentu do stosu cofnięć, jak i podczas usuwania fragmentów z tego stosu, na przykład gdy użytkownik naciśnie przycisk *Wstecz*.

Implementacja fragmentu WebViewFragment

Kolejnym etapem tworzenia aplikacji jest zaimplementowanie klasy pochodnej `WebViewFragment` o nazwie `VeggieGardenWebViewFragment`, reprezentującej fragment, w którym będzie wyświetlana strona poświęcona wybranemu owocowi lub warzywu. Działanie tej klasy ogranicza się niemal jedynie do określenia adresu URL strony i wyświetlenia jej w kontrolce `WebView`:

```
public class VeggieGardenWebViewFragment extends WebViewFragment {

    private static final String DEBUG_TAG = "VGWebViewFragment";

    public static VeggieGardenWebViewFragment newInstance(int index) {
        Log.v(DEBUG_TAG, "Tworzenie nowej instancji: " + index);
        VeggieGardenWebViewFragment fragment = new VeggieGardenWebViewFragment();
        Bundle args = new Bundle();
        args.putInt("index", index);
        fragment.setArguments(args);
        return fragment;
    }

    public int getShownIndex() {
        int index = 1;
        Bundle args = getArguments();
        if (args != null) {
            index = args.getInt("index", -1);
        }
        if (index == -1) {
            Log.e(DEBUG_TAG, "Brak takiego indeksu w tablicy.");
        }

        return index;
    }

    @Override
    public void onActivityCreated(Bundle savedInstanceState) {
        super.onActivityCreated(savedInstanceState);

        String[] veggieUrls = getResources().getStringArray(R.array.veggieurls_array);
        int veggieUrlIndex = getShownIndex();

        WebView webview = getWebView();
        webview.setPadding(0, 0, 0, 0);
```

```
        webview.getSettings().setLoadWithOverviewMode(true);
        webview.getSettings().setUseWideViewPort(true);

        if (veggieUrlIndex != 1) {
            String veggieUrl = veggieUrls[veggieUrlIndex];
            webview.loadUrl(veggieUrl);
        } else {
            String veggieUrl = "http://andys-veggie-garden." +
                "appspot.com/cherrytomatoes";
            webview.loadUrl(veggieUrl);
        }
    }
}
```

Także w tym fragmencie przeważająca większość czynności związanych z jego inicjowaniem jest wykonywana w metodzie zwrotnej `onActivityCreated()`, dzięki czemu będzie je trzeba wykonać tylko jeden raz. Domyślna konfiguracja kontrolki `WebView` nie zapewnia dostatecznie dobrych efektów, dlatego metoda wprowadza w niej pewne zmiany; konkretnie rzecz biorąc, usuwa wypełnienia wokół kontrolki i zapewnia, że obszar przeglądarki będzie idealnie pasować do obszaru dostępnego na ekranie. Jeśli zostało przekazane żądanie wyświetlenia strony konkretnego owocu lub warzywa, to odnajdywany jest jej adres URL, po czym strona zostaje wczytana; w przeciwnym razie wyświetlana jest „domyślna” strona WWW aplikacji, poświęcona pomidorom koktajlowym.

Definiowanie plików układów

Po zaimplementowaniu klas obu fragmentów można dodać te fragmenty do odpowiednich plików zasobów z definicjami układów. Konieczne będzie utworzenie dwóch takich plików. Jeśli aplikacja będzie działać w układzie poziomym, stosowany będzie jeden plik układu, *activity_simple_fragments.xml*, w którym zostaną umieszczone oba fragmenty. Z kolei w przypadku gdy aplikacja będzie działać w układzie pionowym, zastosowany zostanie podobny plik układu zawierający wyłącznie fragment `ListFragment`. Interfejs użytkownika zawierający fragment `WebViewFragment` będzie generowany programowo w trakcie działania aplikacji.

W pierwszej kolejności przedstawiony zostanie plik układu używanego w układzie poziomym — *res/layout-land/activity_simple_fragments.xml*. Warto zwrócić uwagę, że plik ten jest umieszczony w specjalnym katalogu przeznaczonym dla układów używanych wyłącznie w układzie poziomym. Więcej informacji dotyczących przechowywania alternatywnych zasobów można znaleźć w rozdziale 13. „Projektowanie zgodnych aplikacji”. Jak na razie wystarczy powiedzieć, że układ ten będzie wczytywany automatycznie, zawsze gdy urządzenie będzie działać w układzie poziomym.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <include
        android:id="@+id/toolbar"
        layout="@layout/tool_bar" />
    <LinearLayout
        android:orientation="horizontal"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:baselineAligned="false">
```

```
        <fragment
            android:name="com.introtoandroid.simplefragments.VeggieGardenListFragment"
            android:id="@+id/list"
            android:layout_weight="1"
            android:layout_width="200dp"
            android:layout_height="match_parent" />
        <FrameLayout
            android:id="@+id/veggieentry"
            android:layout_weight="4"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </LinearLayout>
</LinearLayout>
```

W tej sytuacji mamy do czynienia ze stosunkowo prostym układem typu `LinearLayout`, wewnątrz którego jest umieszczony drugi układ `LinearLayout` zawierający dwie kontrolki podrzędne. Pierwszą z nich jest statyczny komponent `Fragment`, odwołujący się do zaimplementowanego wcześniej fragmentu `ListFragment`. Jeśli natomiast chodzi o drugą kontrolkę, reprezentującą obszar, w którym będą przedstawiane fragmenty `WebViewFragment`, to zostanie ona zdefiniowana jako układ `FrameLayout`. W trakcie działania aplikacji układ ten będzie programowo zastępowany obiektami `VeggieGardenWebViewFragment`.

Zasoby umieszczone w zwyczajnym katalogu układów będą używane w sytuacjach, gdy urządzenie nie będzie pracować w układzie pionowym (czyli, innymi słowy, gdy będzie trzymane w układzie pionowym). W takim przypadku konieczne jest zdefiniowanie dwóch układów. W pierwszej kolejności zostanie przedstawiony statyczny układ z fragmentem `ListFragment`, zapisany w pliku *res/layout/activity_simple_fragments.xml*. Jest on bardzo podobny do poprzedniego układu, lecz nie występuje w nim kontrolka `FrameLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".SimpleFragmentActivity">
    <include
        android:id="@+id/toolbar"
        layout="@layout/tool_bar" />
    <fragment
        android:name="com.introtoandroid.simplefragments.VeggieGardenListFragment"
        android:id="@+id/list"
        android:layout_weight="1"
        android:layout_width="0dp"
        android:layout_height="match_parent"
        tools:layout="@layout/activity_simple_fragments" />
</LinearLayout>
```

Definiowanie klas aktywności

To już prawie wszystko. Kolejnym krokiem jest napisanie klas aktywności, w których zostaną umieszczone przygotowane wcześniej fragmenty. Potrzebne będą dwie takie klasy: główna oraz dodatkowa, która będzie używana wyłącznie w celu wyświetlenia fragmentu `VeggieGardenWebViewFragment`, w sytuacji gdy urządzenie będzie działać w układzie pionowym. Główna klasa aktywności będzie nosić nazwę `SimpleFragmentsActivity`, a druga — `VeggieGardenActivity`.

Zgodnie z zamieszczonymi wcześniej informacjami przeniesienie całej logiki obsługi interfejsu użytkownika do klas fragmentów ogromnie upraszcza implementację klas aktywności. Poniżej przedstawiony został kompletny kod klasy `SimpleFragmentsActivity`:

```
public class SimpleFragmentsActivity extends AppCompatActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_simple_fragments);
        Toolbar toolbar;
        Toolbar = (Toolbar) findViewById(R.id.toolbar);
        getSupportActionBar(toolbar);
    }
}
```

No właśnie! To cały niezbędny kod! Klasa `VeggieGardenViewActivity` jest tylko nieco bardziej interesująca:

```
public class VeggieGardenViewActivity extends AppCompatActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        if (getResources().getConfiguration().orientation ==
            Configuration.ORIENTATION_LANDSCAPE) {
                finish();
                return;
        }
        if (savedInstanceState == null) {
            setContentView(R.layout.activity_simple_fragments);
            Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
            getSupportActionBar(toolbar);
            getSupportActionBar().setDisplayHomeAsUpEnabled(true);
            VeggieGardenWebViewFragment details = new VeggieGardenWebViewFragment();
            details.setArguments(getIntent().getExtras());

            FragmentManager fm = getFragmentManager();
            FragmentTransaction ft = fm.beginTransaction();
            ft.replace(R.id.list, details);
            ft.commit();
        }
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        if (item.getItemId() == android.R.id.home) {
            onBackPressed();
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

Powyższa aktywność w pierwszej kolejności sprawdza, czy urządzenie znajduje się w odpowiednim ułożeniu, i na tej podstawie określa, czy w ogóle można wykonywać dalsze operacje. Potem tworzona jest instancja fragmentu `VeggieGardenWebViewFragment`, która następnie zostaje użyta do zastąpienia widoku `R.id.list`, będącego głównym widokiem domyślnego układu każdej aktywności typu `ListActivity`. To wszystkie czynności, które trzeba wykonać w celu zapewnienia prawidłowego

działania prezentowanej tu prostej aplikacji korzystającej z fragmentów. Dodatkowo warto zwrócić uwagę, że używa ona paska narzędzi, komponentu `Toolbar`, jako paska akcji.

Stosowanie pakietu biblioteki wsparcia

Fragmenty są tak ważne dla przyszłości platformy Android, że twórcy systemu zdecydowali się na przygotowanie specjalnej biblioteki, dzięki której programiści mogą, jeśli tylko zechcą, zaktualizować swoje stare aplikacje, przeznaczone nawet dla systemu Android 1.6, i wykorzystać w nich fragmenty. Biblioteka ta była początkowo określana jako *Compatibility Package* (pakiet zgodności), jednak obecnie określa się ją jako pakiet *Android Support Library* (pakiet biblioteki wsparcia Androida).

Dodawanie wsparcia dla fragmentów do starych aplikacji

Decyzja o tym, czy warto aktualizować stare aplikacje, czy nie, należy jedynie do ich twórców. Aplikacje, w których fragmenty nie są używane, powinny bez żadnych błędów działać przez całą przewidywalną przyszłość, a to głównie ze względu na prowadzoną przez twórców Androida politykę maksymalnie długiego wspierania starszych aplikacji w kolejnych wydawanych wersjach systemu. Poniżej przedstawiono kilka uwag, które twórcy aplikacji powinni przemyśleć, rozważając, czy warto aktualizować ich istniejący kod:

- Stare aplikacje można pozostawić w niezmienionej postaci, gdyż konsekwencje, z jakimi będzie się to wiązać, nie będą katastrofalne. Takie aplikacje nie będą co prawda wykorzystywać najnowszych i najlepszych możliwości oferowanych przez system Android (co użytkownicy na pewno zauważą), niemniej jednak będą cały czas działać tak jak wcześniej, bez żadnego dodatkowego nakładu pracy ze strony ich twórców. Jeśli twórcy nie mają zamiaru aktualizować ani rozwijać aplikacji, to takie rozwiązanie będzie całkowicie uzasadnione. Pewnym problemem może być mało optymalne wykorzystanie dostępnego obszaru ekranu, choć nie powinno to spowodować nowych błędów.
- Jeśli aplikacja jest bardzo popularna na rynku, a jej twórcy aktualizowali ją wraz z rozwojem platformy Android, to będą też zapewne bardziej skłonni do rozważenia zastosowania pakietu Support Library. Mogą tego także wymagać użytkownicy aplikacji. Oczywiście można kontynuować wsparcie dla starej aplikacji i jednocześnie napisać jej nową, poprawioną wersję korzystającą z możliwości nowszych wersji platformy. Jednak oznacza to konieczność zorganizowania dwóch wersji kodu i dwóch pakietów aplikacji oraz zarządzania nimi, a także komplikuje publikowanie aplikacji i zgłaszanie błędów, nie wspominając nawet o dodatkowych problemach związanych z utrzymaniem aplikacji oraz z ich reklamowaniem. Znacznie lepszym rozwiązaniem będzie unowocześnienie już istniejącej aplikacji i wykorzystanie w niej pakietu Support Library oraz dołożenie wszelkich starań w celu zachowania możliwości utrzymania jednej bazy kodu aplikacji. Ważnymi czynnikami przy podejmowaniu takiej decyzji będą na pewno wielkość i zasoby organizacji.
- Samo rozpoczęcie korzystania z pakietu Support Library w aplikacji nie oznacza wcale, że od razu trzeba w niej wykorzystywać wszystkie nowe możliwości, takie jak fragmenty, mechanizmy wczytywania (ang. *loaders*) czy paski narzędzi. Można wybrać tylko te spośród

możliwości, które mają sens w przypadku danej aplikacji, a pozostałe dodawać stopniowo wraz z jej rozwojem, o ile zespół twórców będzie miał niezbędne zasoby i chęci.

- Rezygnacja z aktualizowania aplikacji i wprowadzania w niej nowych kontrolek może sprawić, że w porównaniu z innymi aplikacjami będzie ona wyglądać staro. Jeśli tworzona aplikacja jest w bardzo dużym stopniu dostosowana do konkretnych potrzeb i wymagań i nie korzysta z żadnych domyślnych kontrolek — co często dzieje się w grach lub innych aplikacjach o wyszukanej formie graficznej — może się okazać, że wcale nie trzeba jej aktualizować. Jeśli jednak aplikacja używa standardowych kontrolek i standardowego sposobu prezentacji i obsługi, to nadanie jej nowoczesnego, świeżego wyglądu może mieć bardzo duże znaczenie.

Stosowanie fragmentów w nowych aplikacjach przeznaczonych dla starszych wersji platformy

W przypadku rozpoczynania pracy nad nową aplikacją, która jednak ma działać także w starszych wersjach platformy Android, podjęcie decyzji o zastosowaniu fragmentów może być znacznie prostsze. W takiej sytuacji po prostu trudno znaleźć argumenty przemawiające za niewykorzystywaniem nowych możliwości, natomiast bez kłopotu można znaleźć powody, by je wykorzystywać:

- Niezależnie od tego, z myślą o których konkretnie platformach i urządzeniach jest tworzona aplikacja, w przyszłości i tak pojawią się kolejne, które obecnie trudno jeszcze przewidzieć. Fragmenty zapewniają elastyczność pozwalającą na łatwe dostosowywanie interfejsu użytkownika do zmieniających się sposobów pracy, bez konieczności ponownego pisania i testowania całego kodu aplikacji.
- Wczesne zaadaptowanie pakietu Support Library w aplikacji oznacza, że jeśli w przyszłości zostaną dodane nowe możliwości, wystarczy wówczas zaktualizować biblioteki i będzie można z tych możliwości skorzystać.
- Dzięki wykorzystaniu pakietu Support Library aplikacja będzie znacznie wolniej wykazywać objawy starzenia się, gdyż będzie korzystać z nowszych możliwości platformy i udostępniać je także użytkownikom używającym starszych wersji systemu Android.

Dołączanie pakietu Support Library do aplikacji

Pakiet Android Support Library jest po prostu zestawem statycznych bibliotek wsparcia (udostępnianym w formie pliku *.jar*), który wystarczy dołączyć do aplikacji, by móc go używać. Pakiet ten można pobrać za pomocą programu Android SDK Manager, a następnie dodawać do wybranych projektów. Jest to pakiet opcjonalny i domyślnie nie jest dołączony do żadnych projektów. Pakiety Android Support Library są udostępniane w wersjach, podobnie jak wszystkie inne elementy Android SDK, i od czasu do czasu są wzbogacane o nowe możliwości oraz, co ważniejsze, uzupełniane o poprawki wykrytych błędów.

> **Wskazówka**
>
> Więcej informacji na temat najnowszej wersji tego pakietu można znaleźć w witrynie *Android Developer*, a konkretnie na stronie *http://d.android.com/tools/support-library/index.html*.

Dostępnych jest siedem wersji pakietu Support Library: v4, v7, v8, v13, v17, Annotation oraz Design. Wersja v4 pakietu obejmuje nowe klasy wprowadzone w systemie Honeycomb oraz następnych i udostępnia je w wersjach platformy Android zgodnych z API poziomu 4 (Android 1.6). To właśnie tego pakietu należy używać, aby zapewnić możliwość stosowania fragmentów w starych aplikacjach. Wersja v7 pakietu obejmuje dodatkowe klasy nieobecne w wersji v4 i jest przeznaczona do udostępniania nowych możliwości w wersjach platformy zgodnych z API poziomu 7 (Android 2.1). Jest ona podzielona na następujące grupy: `appcompat`, `cardview`, `gridlayout`, `mediarouter`, `palette` oraz `recyclerview`. Wersja v8 udostępnia pakiet `renderscript` wspomagający obliczenia `RenderScript` w wersjach platformy zgodnych z API poziomu 8 (Android 2.2). W wersji v13 znalazły się bardziej wydajne implementacje niektórych klas, takich jak `FragmentCompat`, przeznaczone dla wersji platformy zgodnych z API poziomu 13 i nowszych. Wersja v17 zawiera widżety przeznaczone do tworzenia interfejsów użytkownika aplikacji służących do uruchamiania na telewizorach, takie jak `BrowseFragment`, `DetailsFragment`, `PlaybackOverlayFragment` czy też `SearchFragment`. Pakiet `Annotation` zapewnia możliwość dodawania do kodu adnotacji z metadanymi, natomiast pakiet `Design` — stosowanie wzorców projektowych oraz elementów użytkownika typowych dla Material Design.

Aby dodać do aplikacji pakiet Android Support Library, należy wykonać następujące czynności:

1. W przypadku korzystania z Android Studio trzeba pobrać Android Support Repository, używając programu Android SDK Manager. Element Android Support Library jest przeznaczony do użycia w Eclipse.
2. Następnie należy znaleźć plik modułu *build.gradle* (nie plik projektu *build.gradle*) projektu wyświetlony w panelu *Project* Android Studio i otworzyć go.
3. W sekcji zależności (`dependencies`) pliku trzeba dodać wybrane elementy pakietu Support Library, które będą używane w projekcie, podając ich prawidłowe identyfikatory oraz numery wersji. W przypadku prezentowanej tu aplikacji `SimpleFragments` dodane zostały następujące pakiety: `support-v4`, `appcompat-v7` oraz `design`, przy czym każdy z nich, jak pokazano na poniższym przykładzie, w wersji `23.0.0`:

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    compile "com.android.support:support-v4:23.0.0"
    compile "com.android.support:appcompat-v7:23.0.0"
    compile 'com.android.support:design:23.0.0'
}
```

4. Należy rozpocząć korzystanie z nowych API dodanych do projektu. Na przykład w celu napisania klasy dziedziczącej po `FragmentActivity` należy zaimportować klasę `android.support.v4.app.FragmentActivity`.

> **Uwaga**
>
> Istnieje kilka różnic między API udostępnianymi przez pakiet Android Support Library a tymi, które można znaleźć w nowszych wersjach Android SDK. Niemniej jednak w pakiecie tym jest kilka klas, których nazwy zmieniono w celu uniknięcia konfliktów, a poza tym nie wszystkie klasy oraz ich możliwości zostały zaimplementowane prawidłowo.

Dodatkowe sposoby stosowania fragmentów

Fragmenty idealnie nadają się do tworzenia komponentów interfejsu użytkownika przeznaczonych do wielokrotnego stosowania; istnieją jednak także inne możliwości wykorzystania fragmentów w aplikacjach. Na przykład można tworzyć komponenty funkcjonalne pozbawione interfejsu użytkownika bądź też zagnieżdżać jedne fragmenty wewnątrz innych.

Fragmenty funkcjonalne pozbawione interfejsu użytkownika

Fragmenty nie służą wyłącznie do zapewniania separacji interfejsu użytkownika od kodu aktywności. Można ich także używać do implementacji określonych możliwości funkcjonalnych czy zachowań aplikacji, w formie odrębnych komponentów nadających się do wielokrotnego użycia. W takiej sytuacji, dodając lub zastępując fragment, należy zamiast identyfikatora zasobu podać unikalny łańcuch znaków pełniący rolę znacznika. Ponieważ nie wiąże się to z dodawaniem kolejnego widoku do układu, nigdy nie jest wywoływana metoda `onCreateView()`. Trzeba także pamiętać, by takie fragmenty pobierać z aktywności, stosując metodę `findFragmentByTag()`.

Poznawanie zagnieżdżonych fragmentów

W systemie Android 4.2 (API poziomu 17) wprowadzona została możliwość zagnieżdżania fragmentów w innych fragmentach. Analogiczna możliwość została także dodana do pakietu Android Support Library, dzięki czemu z zagnieżdżonych fragmentów można korzystać nawet w systemie Android 1.6 (API poziomu 4). Aby umieścić jeden fragment wewnątrz innego, należy wywołać metodę `getChildFragmentManager()` klasy `Fragment`, która zwraca obiekt typu `FragmentManager`. Po pobraniu tego obiektu można rozpocząć transakcję fragmentu (`FragmentTransaction`), wywołując w tym celu metodę `beginTransaction()`, po czym wywołać metodę `add()`, przekazując do niej fragment oraz jego układ, a na koniec zatwierdzić transakcję z wykorzystaniem metody `commit()`. W zagnieżdżonym fragmencie można nawet wywołać metodę `getParentFragment()`, by pobrać fragment zewnętrzny i wykonać na nim jakieś operacje.

Wprowadzenie fragmentów zagnieżdżonych otwiera nowe możliwości tworzenia dynamicznych komponentów, doskonale nadających się do wielokrotnego stosowania. Jako potencjalne przykłady wykorzystania można tu podać fragmenty zawierające karty umieszczane w innych fragmentach z kartami, przechodzenie z jednego fragmentu ze szczegółami do innego z użyciem kontrolki `ViewPager`, stosowanie kontrolki `ViewPager` do przeglądania fragmentów w ramach fragmentu z kartami, zagnieżdżanie fragmentów funkcjonalnych wewnątrz fragmentów posiadających własny interfejs użytkownika i tak dalej.

Podsumowanie

Fragmenty zostały wprowadzone w Android SDK, aby ułatwić obsługę różnych wielkości ekranów, które muszą być obsługiwane teraz i będą musiały być obsługiwane w przyszłości. Fragment, obiekt typu `Fragment`, jest po prostu samodzielnym, niezależnym elementem interfejsu użytkownika, bądź też implementacją pewnego zachowania, który to element posiada własny cykl życia i może być niezależny od jakiejś jednej, konkretnej aktywności. Fragmenty muszą być umieszczane

w klasach aktywności, przy czym zapewniają programistom znaczną elastyczność pod względem możliwości podziału ekranów aplikacji na komponenty, które można łączyć ze sobą na różne sposoby w zależności od dostępnej wielkości ekranu. Fragmenty zostały wprowadzone w systemie Android 3.0, jednak dzięki pakietowi Android Support Library można ich używać także w starszych aplikacjach, nawet tych przeznaczonych na platformę zgodną z API poziomu 4 (Android 1.6). Jeszcze większą elastyczność podziału interfejsu użytkownika na komponenty zapewniają API pozwalające na tworzenie fragmentów zagnieżdżonych.

Pytania kwizowe

1. Która klasa ułatwia koordynację działań pomiędzy aktywnością a umieszczonymi w niej fragmentami?
2. Którą metodę należy wywołać w celu pobrania obiektu klasy ułatwiającej koordynację działań pomiędzy aktywnością a umieszczonymi w niej fragmentami?
3. Co powinno być wartością atrybutu `android:name` znacznika XML `<fragment>`?
4. Prawda czy fałsz? Metoda zwrotna `onActivityAttach()` jest wywoływana, gdy fragment zostanie po raz pierwszy dołączony do aktywności.
5. Podaj nazwy kilku klas pochodnych klasy `Fragment` (`android.app.Fragment`).
6. Jaką kontrolkę zawiera w sobie fragment typu `ListFragment` (`android.app.ListFragment`)?
7. Fragmenty zostały wprowadzone w API poziomu 11 (Android 3.0). W jaki sposób należałoby dodać możliwość stosowania fragmentów w aplikacji, która ma działać na urządzeniach z systemem Android zgodnym z API poziomu 11?

Ćwiczenia

1. Korzystając z dokumentacji Androida, dowiedz się, jak można dodawać fragmenty do stosu cofnięć. Napisz prostą aplikację korzystającą z układu składającego się z jednego fragmentu służącego do podawania liczby (fragment wyświetlany po uruchomieniu aplikacji powinien wyświetlać liczbę 1) oraz przycisku umieszczonego poniżej. Kliknięcie przycisku powinno spowodować zastąpienie pierwszego fragmentu drugim, w którym będzie wyświetlona liczba 2. Aplikacja ma pozwalać na takie zastępowanie fragmentów aż do wyświetlenia liczby 10, przy czym każdy z fragmentów ma być dodawany do stosu cofnięć, by zapewnić użytkownikowi możliwość powrotu do wcześniej wyświetlanej wartości.
2. Utwórz nowy projekt typu *Phone and Tablet*, używając Android Studio i wykorzystując kreator nowego projektu — na stronie *Add an activity to Mobile* zaznacz opcję *Master/Detail Flow*, następnie wykonaj kolejne kroki kreatora, a na końcu kliknij przycisk *Finish*. Uruchom wygenerowaną aplikację zarówno na telefonie, jak i urządzeniu z ekranem o wielkości typowej dla tabletów, by sprawdzić, jak będzie wyglądać w obu przypadkach; następnie przeanalizuj jej kod, by przekonać się, jak są w niej używane fragmenty.

3. Utwórz układ składający się z dwóch fragmentów, z których oba są generowane i wstawiane do niego programowo w trakcie działania aplikacji. Każdy z fragmentów powinien zajmować 50% dostępnego obszaru ekranu i mieć inny kolor.

Odwołania i inne źródła informacji

Szkolenie: *Building a Dynamic UI with Fragments*[1]:
http://d.android.com/training/basics/fragments/index.html

Przewodnik dla programistów: *Fragments*[2]:
http://d.android.com/guide/components/fragments.html

Dokumentacja Android SDK dotycząca klasy `Fragment`:
http://d.android.com/reference/android/app/Fragment.html

Dokumentacja Android SDK dotycząca klasy `ListFragment`:
http://d.android.com/reference/android/app/ListFragment.html

Dokumentacja Android SDK dotycząca klasy `PreferenceFragment`:
http://d.android.com/reference/android/preference/PreferenceFragment.html

Dokumentacja Android SDK dotycząca klasy `WebViewFragment`:
http://d.android.com/reference/android/webkit/WebViewFragment.html

Dokumentacja Android SDK dotycząca klasy `DialogFragment`:
http://d.android.com/reference/android/app/DialogFragment.html

Narzędzia platformy Android: *Support Library*:
http://d.android.com/tools/support-library/index.html

Blog twórców systemu Android: *The Android 3.0 Fragments API*:
http://android-developers.blogspot.com/2011/02/android-30-fragments-api.html

[1] Tworzenie dynamicznych interfejsów użytkownika z użyciem fragmentów — *przyp. tłum.*

[2] Fragmenty — *przyp. tłum.*

Skorowidz

A

B

C

D

E

F

G

I

J

K

L

Ł

M

N

O

P

R

S

S

T

U

W

Z

Ż

Android — ogranicza go tylko Twoja wyobraźnia!

Android jest otwartą, kompletną i darmową platformą mobilną. Pod jego kontrolą pracują setki milionów urządzeń, przy czym nie są to jedynie tablety i smartfony. Mogą to być właściwie wszystkie urządzenia, do których da się włożyć procesor. Rynek aplikacji mobilnych dla Androida to raj dla programistów, jednak sukces zależy od pomysłu, talentu i wysokiej jakości tworzonych aplikacji.

Niniejsza książka to bezcenne kompendium dla każdego programisty piszącego aplikacje na Androida. Będzie niezastąpiona dla początkujących, którzy dzięki niej zyskają solidne podstawy, ale docenią ją również poważni projektanci profesjonalnych aplikacji. Kompleksowo prezentuje środowisko Android Studio oraz Android SDK, opisuje nowy system uprawnień w Android 6.0, przedstawia sposób wykorzystania bazy SQLite, pokazuje sporo sztuczek i porad związanych z programowaniem na Androida — a to wszystko, by możliwie najpełniej i najstaranniej wyeksponować aktualne i najbardziej interesujące możliwości tej platformy.

Najważniejsze zagadnienia ujęte w książce:

- podstawy platformy Android, jej architektura i zasady funkcjonowania
- architektura aplikacji mobilnej i jej cechy szczególne
- planowanie procesu wytwarzania aplikacji mobilnych i prowadzenie kontroli jakości
- Material Design i jego znaczenie
- pisanie aplikacji, jej testowanie i debugowanie oraz publikacja
- strategie tworzenia oprogramowania dla Androida

Joseph Annuzzi Jr. jest programistą, grafikiem, przedsiębiorcą i autorem książek. Biegle posługuje się różnymi językami programowania, interesuje się kryptografią, algorytmami biometrycznymi i tworzeniem grafik 3D. Mieszka w Dolinie Krzemowej.**Lauren Darcey** jest profesjonalną programistką i uznanym autorytetem w dziedzinie architektury aplikacji oraz aplikacji mobilnych. **Shane Conder** od ponad dziesięciu lat pisze aplikacje dla środowisk mobilnych i osadzonych. Zaprojektował i napisał wiele komercyjnych aplikacji na Androida, iPhone'a, BREW, BlackBerry, Palm oraz dla środowisk J2ME i Windows Mobile.

Helion

księgarnia internetowa
http://helion.pl
zamówienia telefoniczne
0 801 339900
0 601 339900

Helion SA
ul. Kościuszki 1c, 44-100 Gliwice
tel.: 32 230 98 63
e-mail: helion@helion.pl
http://helion.pl

Sprawdź najnowsze promocje:
http://helion.pl/promocje
Książki najchętniej czytane:
http://helion.pl/bestsellery
Zamów informacje o nowościach:
http://helion.pl/nowosci

ISBN 978-83-283-2612-5

Informatyka w najlepszym wydaniu
cena: 99,00 zł

Addison Wesley

ALWAYS LEARNING
PEARSON