POLITECHNIKA LUBELSKA

PODRĘCZNIKI

Jakub Smołka

Programowanie aplikacji dla systemu Android

Lublin 2014

Programowanie aplikacji dla systemu Android

Podręczniki – Politechnika Lubelska

KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

Publikacja współfinansowana ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Jakub Smołka

Programowanie aplikacji dla systemu Android

Politechnika Lubelska
Lublin 2014

Recenzenci:
dr inż. Beata Pańczyk

Skład: Jakub Smołka

Publikacja dystrybuowana bezpłatnie.

Publikacja opracowana i finansowana z projektu „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW"

Projekt „Politechnika przyszłości – dostosowanie oferty do potrzeb rynku pracy i GOW" współfinansowany przez Unię Europejską w ramach Europejskiego Funduszu Społecznego Programu Operacyjnego Kapitał Ludzki.

Publikacja wydana za zgodą Rektora Politechniki Lubelskiej

ISBN: 978-83-7947-069-3

Wydawca: Politechnika Lubelska
ul. Nadbystrzycka 38D, 20-618 Lublin
Realizacja: Biblioteka Politechniki Lubelskiej
Ośrodek ds. Wydawnictw i Biblioteki Cyfrowej
ul. Nadbystrzycka 36A, 20-618 Lublin
tel. (81) 538-46-59, email: wydawca@pollub.pl
www.biblioteka.pollub.pl
Druk: TOP Agencja Reklamowa Agnieszka Łuczak
www.agencjatop.pl

Elektroniczna wersja książki dostępna w Bibliotece Cyfrowej PL www.bc.pollub.pl
Nakład: 100 egz.

Wstęp 6
1. Pierwsza aplikacja dla Androida 7
1.1. Niezbędne narzędzia 7
1.2. Tworzenie projektu 7
2. Tworzenie graficznego interfejsu użytkownika 17
2.1. Struktura pliku układu interfejsu użytkownika 17
2.2. Podstawowe komponenty 17
2.2. Typowe właściwości 18
2.3. Tworzenie układu 19
2.4 Podstawowa składowa aplikacji – Aktywność (Activity) 24
2.5. Cykl życia aktywności 24
2.6. Odwoływanie się do elementów układu w kodzie aplikacji 27
2.7. Obsługa zdarzeń 27
2.8. Zaawansowane rozmieszczanie elementów – Relative Layout 36
2.9. Aplikacje zawierające więcej niż jedną aktywność 40
2.10. Adaptery 44
2.11 Zachowywanie stanu aktywności 55
2.12. Przekazywanie wyników z aktywności 58
2.14 Tworzenie menu – pasek akcji 60
2.15. Fragmenty 67
3. Trwałe przechowywanie danych 99
3.1 Szkielet przykładowego projektu 99
3.2. Korzystanie z bazy danych SQLite 104
3.3. Korzystanie z ustawień współdzielonych 133
3.4 Zapisywanie/odczytywanie danych z pliku 141
4. Podstawy wielozadaniowości 153
4.1. Wykonywanie krótkich zadań w tle – klasa AsyncTask 154
4.2. Proste usługi – dziedziczące po IntentService 159
5. Wykorzystanie usług frameworku Android 176
5.1. Wykorzystanie map 176
5.2. Rozpoznawanie mowy 186
Bibliografia 189

Wstęp

Android to system operacyjny rozwijany przez Open Headset Alliance, któremu przewodniczy Google. Zbudowany został na bazie systemu Linux. Kod źródłowy systemu jest dostępny na licencji Apache. Do tej pory ukazały się cztery główne wersje systemu, jednak API dostępne dla aplikacji zmieniło się 16 razy (w sumie 17 wersji). Znacząca większość urządzeń z Androidem zawiera dodatkowo aplikacje Google, które są dodatkiem do systemu i nie są oprogramowaniem open-source (np. Mapy Google). Od 2010 roku Android to najpopularniejsza platforma mobilna na świecie.

Niniejszy podręcznik przedstawia najważniejsze zagadnienia związane z programowaniem aplikacji dla Androida.

1. Pierwsza aplikacja dla Androida

W niniejszym rozdziale zobaczymy, krok po kroku, jak tworzyć projekty z wykorzystaniem odpowiednio skonfigurowanego środowiska Eclipse i pakietu android SDK.

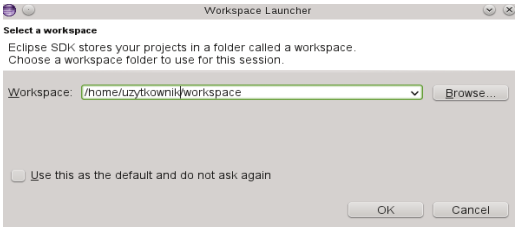
1.1. Niezbędne narzędzia

Zanim przystąpimy do tworzenia aplikacji dla systemu Android będziemy potrzebować odpowiednich narzędzi programistycznych. Najważniejszym z nich jest pakiet *Android Software Development Kit*. Jest on dostępny na stronie: http://developer.android.com/sdk/index.html. Po instalacji pakietu *SDK* należy za pomocą narzędzia *SDK Manager* pobrać i zainstalować tzw. platformy i pakiety [1]. Pakiety i platformy to różne wersje systemu Android oraz dodatkowe biblioteki, przykłady, sterowniki. Używanie samego SDK nie jest zbyt wygodnym rozwiązaniem. Warto więc zainstalować również wspierane przez Google środowisko *Eclipse* (www.eclipse.org) wraz z wtyczką *Android Developer Tools* [10]. Wszystkie te narzędzia dostępne są również w jednym pakiecie jako *ADT Bundle* [12]. Ponieważ wszystkie wspomniane narzędzia, jak również sam system Android są zbudowane wokół języka Java, to niezbędny będzie również *Java Development Kit*. Można korzystać z JDK firmy Oracle (http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html) jak też *OpenJDK* (http://openjdk.java.net/).

Oprócz zestawu złożonego ze środowiska *Eclipse* wraz z wtyczką *Android Developer Tools* dostępne jest również *Android Studio*. Jest to narzędzie, które docelowo ma się stać domyślnym narzędziem do tworzenia aplikacji przeznaczonych dla systemu Android. Jednak w momencie powstawania niniejszego podręcznika jest dostępna jedynie niestabilna wersja rozwojowa.

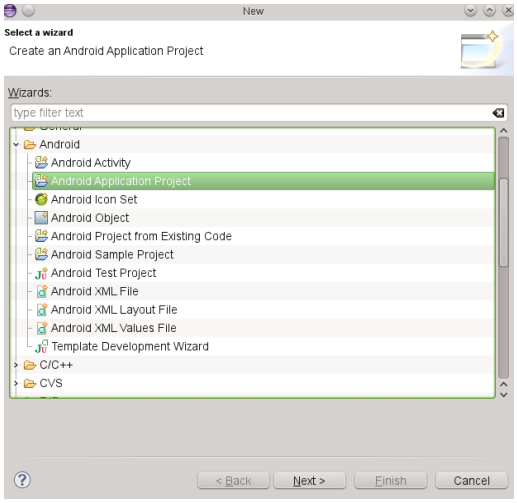
1.2. Tworzenie projektu

Tworzenie projektu rozpoczynamy od uruchomienia środowiska Eclipse. Wykorzystuje ono *przestrzenie robocze* (*ang. workspace*), z których każda znajduje się w odrębnym katalogu. Katalog wybiera się podczas uruchamiania środowiska (rys. 1.1). W każdej przestrzeni może znajdować się wiele projektów. Można również przełączać się między przestrzeniami w trakcie pracy środowiska.



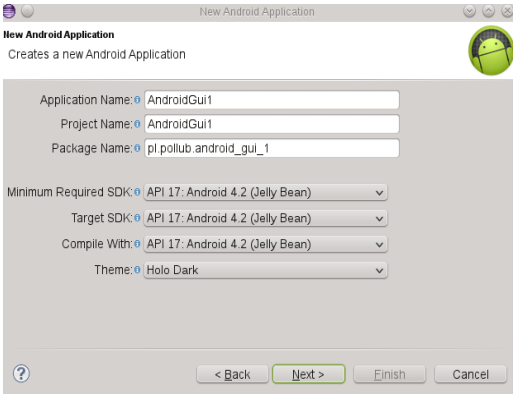
Rys. 1.1. Wybór *przestrzeni roboczej* – każda przestrzeń znajduje się w odrębnym katalogu

Po uruchomieniu Eclipse można przystąpić do tworzenia nowego projektu. Należy wybrać *File* | *New* | *Other* (oznacza to, że w menu *File* należy wybrać opcję *New*, a w pojawiającym się po wybraniu w/w opcji kolejnym menu wybrać opcję *Other* – w dalszej części podręcznika tak będą oznaczane sekwencje opcji, które należy wybrać).



Rys. 1.2. Okno wyboru typu tworzonego elementu

Potem pojawi się okno wyboru typu tworzonego elementu (rys. 1.2). Należy w nim oczywiście wybrać *Android Application Project*.

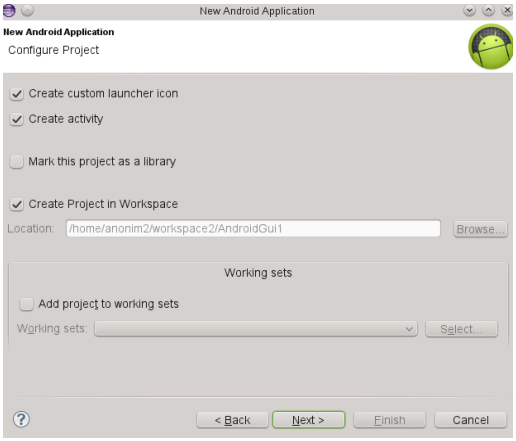


Rys. 1.3. Wybór nazwy, pakietu, poziomu API i tematu

Uruchomiony zostanie kreator projektu (rys. 1.3), w którym na początku ustawiamy:

- nazwę aplikacji (zazwyczaj rozpoczynającą się od wielkiej litery),
- nazwę projektu (widoczną w przestrzeni roboczej),
- nazwę pakietu (która rozpoczyna się zwyczajowo od odwróconej nazwy domenowej instytucji/organizacji odpowiedzialnej za aplikację),
- stosowanych wersji SDK,
- tzw. tematu czyli wyglądu aplikacji

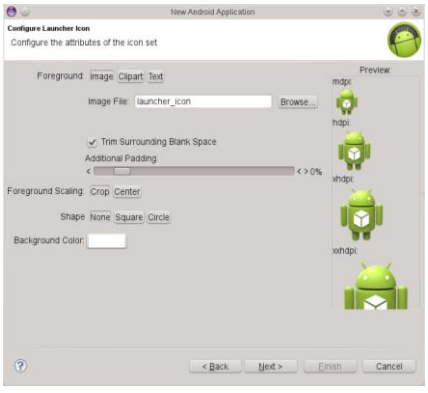
Jak widać na rys. 1.3 aplikację nazwiemy *AndroidGUI1*, pakietem będzie `pl.pollub.android_gui_1`, wszystkie SDK ustawimy na *API 17*: *Android 4.2 (Jelly Bean)*, a jako temat ustawimy *Holo Dark*. W tym momencie warto poświęcić więcej uwagi wspomnianym wersjom SDK. Kolejne wersje systemu android oprócz powszechnie znanych numerów wersji takich jak 4.0; 4.2 czy ich nazw kodowych takich jak *Ice Cream Sandwic*h czy *Jelly Bean* posiadają również odpowiadającą im wersję API Level. I tak wersji 4.0 odpowiada API Level 14 a 4.2 API Level 17. Różne wersje API Level oznaczają, że odpowiadające im systemy różnią się pod względem dostępnych możliwości z punktu widzenia programisty. Wersje API i ich numery są istotne z tego względu, że właśnie tych numerów używa się w trakcie tworzenia aplikacji przeznaczonej dla konkretnych wersji systemu (najczęściej dla określenia najniższej wymaganej wersji systemu).



Rys. 1.4. Drugi ekran kreatora

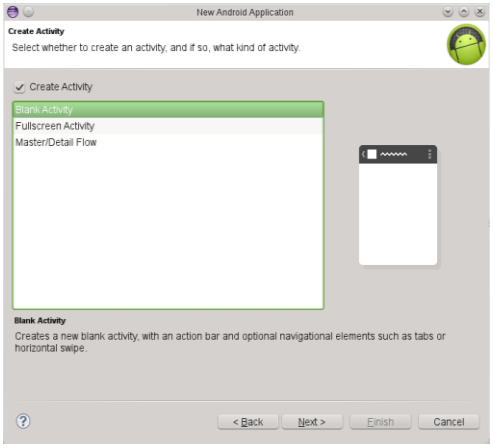
W kolejnym etapie kreatora (rys. 1.4) nie dokonujemy żadnych zmian. Zaznaczone opcje oznaczają, odpowiednio, że:

- chcemy utworzyć własną ikonę dla tzw. *lauchera*, czyli tą, która pojawia się na liście aplikacji zainstalowanych w urządzeniu,
- chcemy utworzyć pustą aktywność,
- utworzyć projekt w bieżącej przestrzeni roboczej.



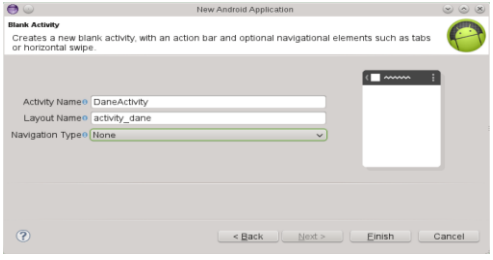
Rys. 1.5. Tworzenie / dostosowanie ikony launchera

Dalej (rys. 1.5) mamy możliwość dostosowania lub zmiany ikony lauchera, jej położenia czy kształtu.



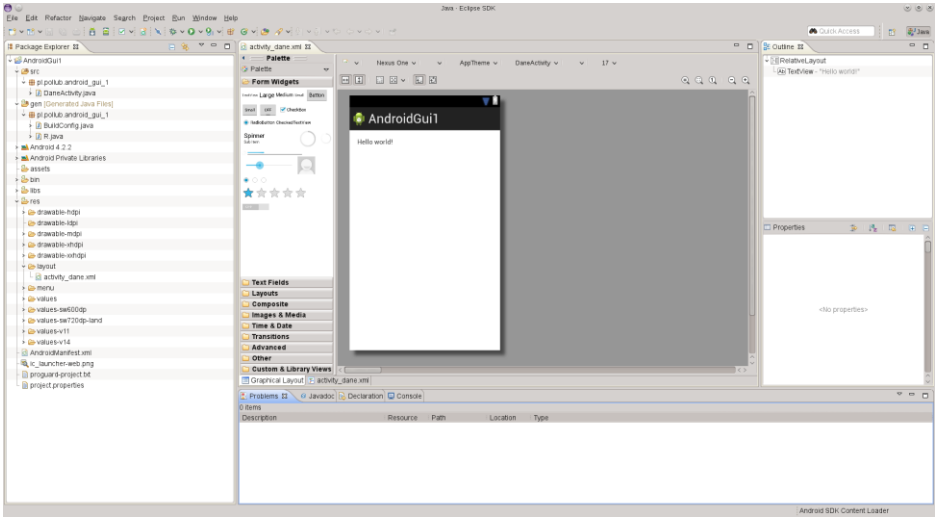
Rys. 1.6. Wybór rodzaju tworzonej aktywności

Po utworzeniu ikony kreator przechodzi do tworzenia tzw. aktywności – jednego z podstawowych elementów aplikacji dla systemu Android. Najpierw wybieramy szablon, według którego zostanie utworzona aktywność (w naszym wypadku wybierzemy najprostszą – pustą aktywność – rys. 1.6).



Rys. 1.7. Ustawienia pustej aktywności

Po wybraniu szablonu ustawiamy nazwę aktywności (nazwę klasy) oraz nazwę *układu* (ang. *layout*) opisującego wygląd aktywności. W pierwszej aplikacji aktywność nazwiemy `DaneActivity` a jej układ `activity_dane` (rys. 1.7).

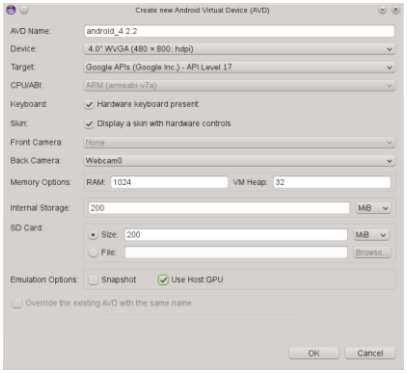


Rys. 1.8. Nowy projekt w środowisku Eclipse

Rys. 1.8 przedstawia środowisko Eclipse tuż po utworzeniu projektu. Jak widać, w eksploratorze pakietów (po lewej stronie okna) projekt składa się z wielu plików i katalogów. Najważniejsze z nich to:

- katalog `/src` – zawierający pliki źródłowe klas naszej aplikacji. Każda klasa (z wyjątkiem wewnętrznych i anonimowych) znajduje się w osobnym pliku. Klasy podzielone są na pakiety, chociaż w skład prostych aplikacji wchodzi tylko jeden pakiet,
- katalog `/gen` – zawiera pliki wygenerowane automatycznie przez narzędzia. Najważniejszym z nich jest plik `R.java` zawierający klasę `R` z identyfikatorami zasobów wykorzystywanych w aplikacji. Identyfikatory te są niezbędne w trakcie tworzenia aplikacji dla Androida,
- katalog `/res/layout` – zawiera pliki XML opisujące układ rozmaitych komponentów – m.in. aktywności,
- katalog `/res/values` – zawiera pliki XML z wartościami wykorzystywanymi w aplikacji np. napisami.

Dzięki takiej strukturze – oddzielającej wygląd, tekst i inne zasoby od kodu i logiki aplikacji można łatwo przygotować wersje aplikacji przeznaczone np. dla urządzeń o innych rozmiarach ekranu czy w innej wersji językowej.



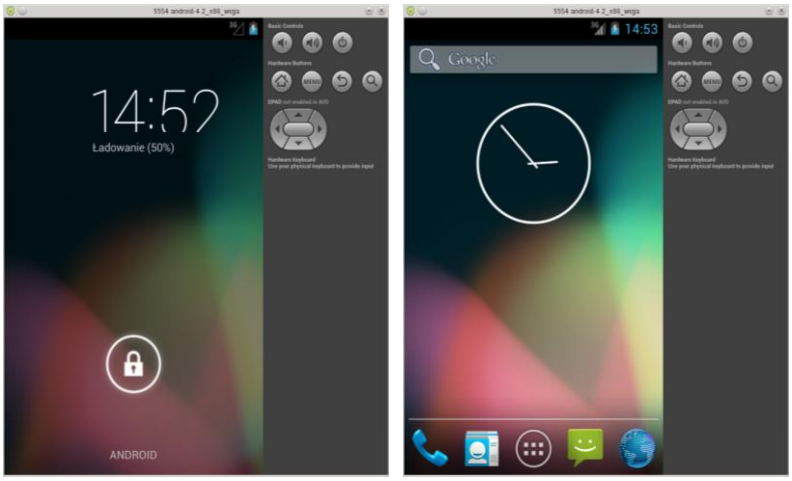
Rys. 1.9. Tworzenie nowej maszyny wirtualnej

Ponieważ dysponujemy w tym momencie działającą (aczkolwiek prostą) aplikacją warto wypróbować jej działanie. W tym celu niezbędna będzie maszyna wirtualna z systemem Android. Aby ją utworzyć wystarczy na pasku narzędzi środowiska Eclipse znaleźć ikonę *Android Virtual Device Manager* i wybrać opcję *New* (lub wybrać menu *Window* | *Android Virtual Device Manager* | *New*). Pojawi się okno przedstawione na rys. 1.9. Możemy w nim ustawić wiele właściwości nowej maszyny wirtualnej. Najważniejsze z nich to:

- rozmiar ekranu i rozdzielczość urządzenia,
- wersja API – nie może być mniejsza niż ta wybrana w trakcie tworzenia projektu,
- CPU/ABI – rodzaj procesora, dla którego tworzona jest aplikacja. Najbardziej typowym rodzajem procesora w urządzeniach mobilnych jest procesor ARM. Wybranie architektury x86 umożliwia stosowanie sprzętowej wirtualizacji na komputerach PC. Wybór ten jest bardzo istotny w przypadku testowania aplikacji zbudowanych z wykorzystaniem pakietu NDK (Native Development Kit), czyli zawierających kod dla konkretnego typu procesora a nie dla wirtualnej maszyny Java. W przypadku naszej aplikacji może mieć on wpływ jedynie na wydajność,
- dostępność sprzętowej klawiatury,
- wygląd emulatora – można wybrać skórkę: z przyciskami sprzętowymi lub bez nich,
- emulacja lub wykorzystanie kamery internetowej jako aparatu/przedniej kamery urządzenia
- rozmiar pamięci RAM urządzenia (warto zwiększyć tę wartość, jeżeli pozwala na to konfiguracja komputera),
- rozmiar pamięci masowej wbudowanej w emulowane urządzenie,

- rozmiar i położenie pliku, w którym znajduje się zawartość emulowanej karty SD,
- wykorzystanie procesora graficznego komputera do przyspieszania efektów graficznych generowanych przez emulator.

Ustawienia maszyny wirtualnej zatwierdzamy przyciskiem *OK*. W tym momencie maszyna jest tworzona i następuje powrót do list maszyn. Możemy utworzyć wiele maszyn wirtualnych o różnych konfiguracjach. Aby przetestować naszą maszynę wybieramy ją z listy i klikamy przycisk *Start*.

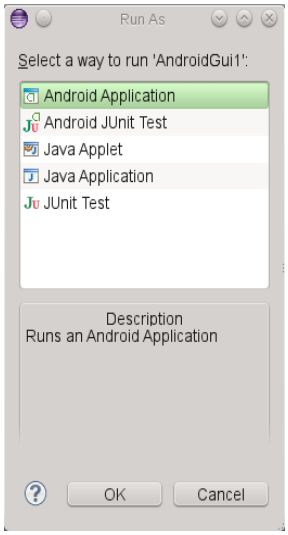


Rys. 1.10. Pracujący emulator

Pierwsze uruchomienie zazwyczaj trwa dłużej niż kolejne. W skrajnych przypadkach jest to nawet kilka minut, więc należy cierpliwie poczekać na zakończenie tego procesu. Niestety kolejne uruchomienia też nie należą do najszybszych. Negatywnie na jego wydajność wpływa fakt, iż uruchamiany na emulatorze system jest skompilowany dla procesora o architekturze ARM, a programowo musi być emulowany przez procesor x86 pracujący w komputerze PC. To wszystko powoduje, że już uruchomiony emulator warto pozostawić pracujący w tle. Rozwiązaniem (przynajmniej częściowym) tego problemu jest wykorzystanie obrazów przygotowanych dla procesorów x86 (wybieranych w trakcie tworzenia maszyny) i wirtualizatora KVM (w systemach linux) lub HAXM (w systemach Windows i OS X). Istnieją też alternatywne rozwiązania emulacji systemu Android, których nie wspiera jednak firma Google. Możliwe jest również wykorzystanie rzeczywistego urządzenia, ale sposób konfiguracji takiego rozwiązania zależy od samego urządzenia, jego producenta oraz wykorzystywanego systemu operacyjnego na komputerze programisty. Wykorzystanie

wirtualizacji, niewspieranych rozwiązań i konfiguracja urządzeń są bardziej zaawansowanymi tematami i wykraczają poza zakres niniejszego podręcznika. Rysunek 1.10 przedstawia wygląd pracującego emulatora. Standardowo uruchamia się on do ekranu blokady – aby rozpocząć korzystanie z niego, należy przesunąć ikonę kłódki w prawo.

Gdy uruchomimy emulator możemy przejść do uruchomienia naszej aplikacji. W eksploratorze pakietów (po lewej stronie środowiska Eclipse) należy wybrać plik źródłowy np. `DaneActivity.java` i kliknąć przycisk *uruchom* znajdujący się na pasku narzędziowym lub skorzystać ze skrótu *Ctrl-F11*.



Rys. 1.11. Wybór sposobu uruchamiania

Przy pierwszym uruchomieniu pojawia się okno wyboru sposobu uruchamiania (rys. 1.11), w którym wybieramy: *Android Application*. Aplikacja zostanie skompilowana (jeżeli to koniecznie – ponieważ Eclipse buduje projekty Android automatycznie), skopiowana na emulator/podłączone urządzenie i uruchomiona. Jeżeli pracuje równocześnie kilka emulatorów (lub jest podłączonych kilka urządzeń) pojawi się okno wyboru urządzenia.

Rys. 1.12. Aplikacja pracująca na emulatorze

Rysunek 1.12 Przedstawia aplikację *Hello World*, którą utworzyliśmy z wykorzystaniem szablonów wbudowanych w *Android Developer Tools*. W kolejnym rozdziale będziemy modyfikować i rozbudowywać aplikację.

2. Tworzenie graficznego interfejsu użytkownika

W rozdziale tym zajmiemy się bardzo ważnym zagadnieniem, dotyczącym chyba każdej aplikacji mobilnej, jakim jest tworzenie graficznego interfejsu użytkownika. Za przykład posłuży nam aplikacja pozwalająca wpisywać oceny studenta i obliczać ich średnią.

Jak już wspomniano, definiowanie wyglądu interfejsu użytkownika zostało oddzielone od kodu aplikacji w języku Java. GUI jest opisywane za pomocą plików XML, które dotyczą układu elementów całych aktywności lub nowych komponentów (np. złożonych z kilku komponentów podstawowych). W typowym układzie znajduje się odwołanie do przynajmniej jednego *zarządcy układu* (*ang. layout manager*) i wielu komponentów.

2.1. Struktura pliku układu interfejsu użytkownika

Przykładowy plik XML opisujący układ elementów można znaleźć na listingu 2.2. W każdym układzie pierwsza linia ma postać: `<?xml version="1.0" encoding="utf-8"?>` natomiast główny element układu (niezależnie od tego czy jest to zarządca układu czy komponent) musi zawierać odwołanie do przestrzeni nazw:

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

Właściwości poszczególnych komponentów i zarządców definiuje się, określając wartości atrybutów postaci `android:właściwość="wartość"`. Szczególną uwagę należy zwrócić na atrybuty

```
android:id="@+id/unikalnyIdentyfikator".
```

W zapisie tym **„@”** oznacza, że pozostała część wartości powinna być interpretowana jako identyfikator, natomiast znak **„+”** oznacza, że wymieniony obok identyfikator jest nowym identyfikatorem (tzn. nie jest odwołaniem do istniejącego już identyfikatora). Identyfikatory pozwalają na odwoływanie się do poszczególnych elementów w kodzie programu. Główny element nie posiada identyfikatora.

2.2. Podstawowe komponenty

Typowe komponenty dostępne na platformie Android dzielą się na dwie podstawowe grupy: dziedziczące po klasie `View` oraz dziedziczące (również) po klasie `ViewGroup` (klasa `ViewGroup` dziedziczy po `View`). Do pierwszej kategorii należą pojedyncze komponenty, takie jak przyciski czy pola tekstowe. Do

drugiej kategorii należą zarządcy układu i komponenty, które są pojemnikami na „zwykłe” komponenty (np. listy).

Do podstawowych elementów interfejsu użytkownika należą: `TextView` – etykieta, `EditText` – pole tekstowe, `Button` – przycisk, `CheckBox` – pole wyboru, `RadioButton` – przycisk radiowy, `RadioGroup` – grupa przycisków radiowych. Z dostępnymi komponentami najłatwiej można się zapoznać, przeglądając paletę dostępną z lewej strony graficznego edytora układu (rys. 1.8).

2.2. Typowe właściwości

Zanim przystąpimy do pracy nad układem, warto zapoznać się z podstawowymi atrybutami elementów interfejsu użytkownika, które będziemy wykorzystywać wraz z ich typowymi wartościami.

Atrybut `android:orientation="vertical"` lub `"horizontal"` pozwala (m.in.) na określenie sposobu rozmieszczenia elementów przez liniowego zarządcę układu (w pionie lub w poziomie).

Atrybuty `android:layout_width="match_parent"` lub `"wrap_content"` i `android:layout_height="match_parent"` lub `"wrap_content"` pozwalają na ustalenie szerokości oraz wysokości zarządcy/komponentu. Wartość `match_parent` oznacza, że zarządca/komponent wypełni cały obszar zajmowany przez swojego rodzica. Wartość `wrap_content` oznacza, że rozmiar komponentu zostanie dopasowany do zawartości (np. rozmiar etykiety zostanie dostosowany do długości tekstu).

Atrybut `android:layout_weight="1"` pozwala na określenie, w jakim stopniu komponenty są rozciągane przez liniowego zarządcę układu (`LinearLayout`). Wartość 0 oznacza, że komponent nie jest rozciągany. Dodatkowe piksele dostępne w układzie będą rozdysponowane proporcjonalnie do wartości atrybutu np. gdy komponenty mają `layout_weight` równe `2` i `3`, to pierwszy otrzyma 2/5 a drugi 3/5 wolnej przestrzeni.

Atrybut `android:text="@string/unikalnyIdentyfikator"` służy do określenia tekstu etykiety lub początkowej wartości pola tekstowego. Zalecane jest, aby zawierał odwołanie do napisu zdefiniowanego w pliku z zasobami.

Atrybut `android:id="@+id/unikalnyIdentyfikator"` definiuje unikalny identyfikator elementu interfejsu użytkownika. Główny element układu nie posiada identyfikatora, ponieważ jest identyfikowany przez nazwę pliku XML.

Atrybut `android:digits="1234567890"` pozwala na ograniczenie znaków, które można wpisywać w polu tekstowym.

Atrybuty:

- `android:paddingBottom = "@dimen/activity_vertical_margin"`,
- `android:paddingLeft = "@dimen/activity_horizontal_margin"`,
- `android:paddingRight = "@dimen/activity_horizontal_margin"`,

- `android:paddingTop = "@dimen/activity_vertical_margin"`

określają odpowiednio dolny, lewy, prawy i górny margines wewnętrzny. Są podobne do atrybutu `margin`, który określa margines zewnętrzny.

Atrybut `android:ems="10"` określa rozmiar pola w jednostkach *em* stosowanych w typografii (dla czcionki o rozmiarze 16 punktów 1 em jest równy 16 punktów)

Atrybut android: `inputType="textPersonName"` pozwala określić rodzaj danych, jakie będą wpisane do pola tekstowego. Inną możliwą wartością jest np. `number` – pozwala ona ograniczyć zakres wprowadzanych znaków i spowodować, że po przejściu do tego pola pojawi się klawiatura numeryczna

Atrybut `android:visibility="gone"` lub `"invisible"` lub `"visible"` pozwala na określenie widoczności elementów. Wartość visible oznacza, że element jest widoczny, natomiast `gone` i `invisible` oznaczają, że element jest niewidoczny. Różnica polega na tym, że komponent z wartością `invisible` zajmuje miejsce w układzie elementów (widoczne jest puste miejsca), natomiast komponent z wartością `gone` traktowany jest tak, jakby nie został umieszczony w układzie komponentów.

Znacznik `<requestFocus />` powoduje, że pole domyślnie, po uruchomieniu aktywności otrzyma fokus.

2.3. Tworzenie układu

Przejdziemy teraz do utworzenia *układu* (*ang. layout*) pierwszej aktywności naszej przykładowej aplikacji. Ponieważ kreator aplikacji utworzył już dla nas jeden plik układu, to wykorzystamy go i zmodyfikujemy, tak aby odpowiadał wymaganiom naszej aplikacji.

Na początek skorzystamy z najprostszego *zarządcy układu* (*ang. layout manager*) czyli *układu liniowego* (*ang. linear layout*). Ponieważ kreator domyślnie wybiera *układ względny* (*ang. relative layout*), musimy zmienić główny element naszego układu. W tym celu w edytorze GUI, w drzewie przedstawiającym *hierarchię układu* (*outline*), znajdującym się po prawej stronie graficznego edytora (rys. 1.8), klikamy prawym przyciskiem myszy na wspomnianym elemencie i wybieramy *ChangeLayout* z *relative* na *LinearLayout (Vertical)*. Następnie usuwamy etykietę z napisem *Hello world*, która została dodana przez kreator i dodajemy potrzebne nam elementy przeciągając je z palety komponentów (widoków):

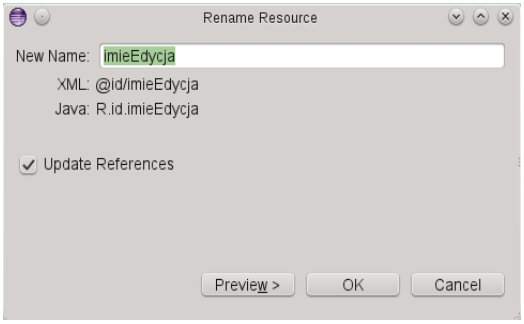
- przeciągamy *LinearLayout (Horizontal)* do nowego układu (dzięki temu mamy układ poziomy zagnieżdżony w układzie pionowym),
- przeciągamy *TextView* (z *Form Widgets*) i *Person Name* (z *Text Fields*) do poziomego układu liniowego.

Następnie zmieniamy właściwości właśnie dodanych elementów:

- zmieniamy właściwość *Layout Width* nowego pola tekstowego na *match* (można skorzystać z listy właściwości znajdującej się po prawej stronie

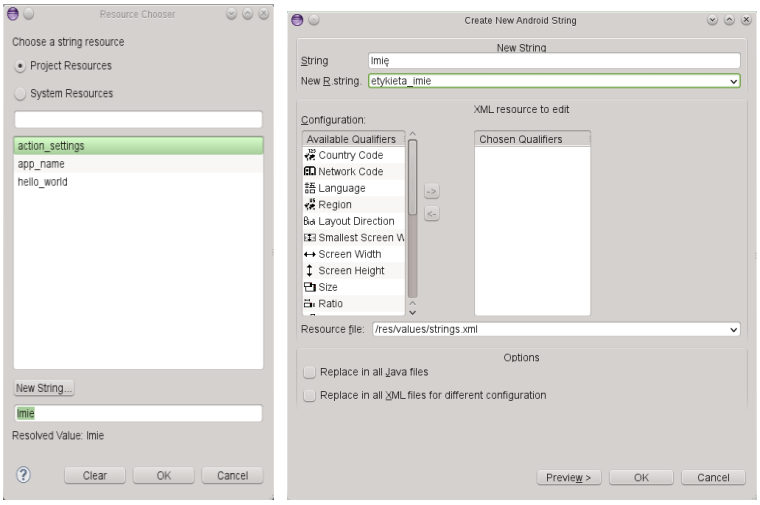
edytora GUI lub kliknąć prawym przyciskiem pole tekstowe i wybrać w menu kontekstowym wspomnianą właściwość),

- w podobny sposób ustawiamy właściwość *id* (identyfikator) etykiety i pola tekstowego. Nadajemy im odpowiednio identyfikatory: *imieEtykieta* oraz *imieEdycja*, w trakcie zmiany identyfikatora pojawia się okno przedstawione na rys. 2.1. Warto w nim zaznaczyć opcję *Update References*, która spowoduje aktualizację nazwy we wszystkich miejscach, w których występuje odwołanie do zmienianego identyfikatora,



Rys. 2.1. Okno zmiany nazwy zasobu

- Ustawiamy tekst etykiety: *imieEtykieta*. Możemy skorzystać w listy właściwość po prawej stronie edytora GUI lub za pomocą menu kontekstowego elementu i opcji *Edit Text*. Pierwsza metoda ustawia wpisany tekst bezpośrednio w pliku XML, opisującym układ. Lepszym rozwiązaniem jest druga metoda, która uruchamia kreator dodawania łańcuchów tekstowych przedstawiony na rys. 2.2. Na pierwszym ekranie wybieramy opcję New String..., na drugim wpisujemy wartość łańcucha (w tym przypadku *„Imię”*), oraz jego identyfikator (*„etykieta_imie”*). Zatwierdzamy wprowadzone wartości i z listy łańcuchów wybieramy utworzony właśnie łańcuch.



Rys. 2.2. Kreator dodawania łańcuchów

Dodawanie łańcucha za pomocą opisanego kreatora spowoduje dodanie go do pliku `res/values/strings.xml`. Na listingu 2.1 przedstawiono zawartość pliku `strings.xml` naszej aplikacji. Plik ten można oczywiście modyfikować bezpośrednio (bez użycia kreatora), co wydaje się prostszym sposobem. Zaletą umieszczania łańcuchów w osobnym pliku jest to, że inną wersję językową aplikacji można stworzyć, tłumacząc zawartość tylko tego pliku i umieszczając go w odpowiednim katalogu w drzewie projektu (system Android automatycznie wybierze odpowiedni plik na podstawie ustawień językowych urządzenia). My jednak poprzestaniemy na wersji polskiej.

Listing 2.1. Plik `res/values/strings.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="app_name">AndroidGui1</string>
    <string name="action_settings">Settings</string>
    <string name="hello_world">Hello world!</string>
    <string name="etykieta_imie">Imię</string>
</resources>
```

Listing 2.2 przedstawia plik układu, który powinniśmy otrzymać. Warto wspomnieć, że czasami dużo prostszym i szybszym sposobem tworzenia układu jest bezpośrednia praca z plikiem XML (bez edytora graficznego). Środowisko

zapewnia automatyczne formatowanie (skrót *Shift+Ctrl+F*), a także podpowiedzi nazw atrybutów (wystarczy wpisać np. android:p i nacisnąć *Ctrl+Spacja*) czy dopuszczalnych wartości atrybutu (wystarczy wpisać =”” i również użyć skrótu *Ctrl+Spacja*). Oczywiście bardzo szybko można również powielać elementy.

Listing 2.2. plik `res/layout/activity_dane.xml`

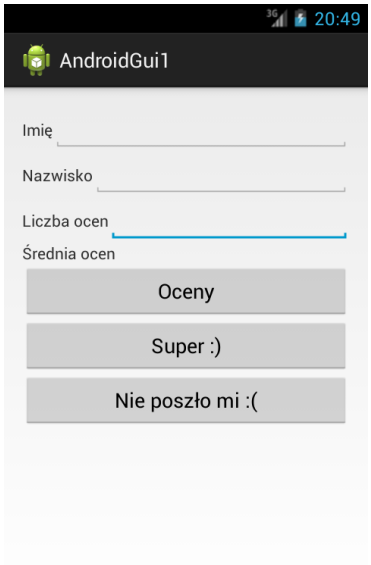
```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".DaneActivity" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >

        <TextView
            android:id="@+id/imieEtykieta"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/etykieta_imie" />

        <EditText
            android:id="@+id/imieEdycja"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:ems="10"
            android:inputType="textPersonName" >
            <requestFocus />
        </EditText>

    </LinearLayout>
</LinearLayout>
```

Ćwiczenie 2.1 (do samodzielnego wykonania)

Dokończ tworzenie układu pierwszej aktywności przykładowej aplikacji. Spróbuj (tam gdzie to będzie możliwe i uzasadnione) bezpośrednio modyfikować plik XML. Dodaj następujące elementy:

- kolejny poziom `LinearLayout` z etykietą i polem tekstowym. W pole użytkownik będzie wpisywał nazwisko studenta. Identyfikatory dodawanych elementów to `nazwiskoEtykieta` i `nazwiskoEdycja`,
- kolejny poziom `LinearLayout` z etykietą i polem tekstowym. W tym polu użytkownik będzie wpisywał liczbę ocen studenta. W przypadku tego pola tekstowego wykorzystaj właściwość `inputType` i ustaw ją na: `number` (dzięki temu będzie się wyświetlać odpowiednia klawiatura i nie będzie można w to pole wpisać liter). Identyfikatory dodawanych elementów to: `liczbaOcenEtykieta` i `liczbaOcenEdycja`,
- etykietę oraz trzy przciski do układu głównego (pionowego) o identyfikatorach odpowiednio: `sredniaEtykieta`, `ocenyPrzycisk`, `superPrzycisk`, `niePoszloPrzycisk`. Odpowiednie napisy zdefiniuj w pliku `strings.xml`

Sprawdź działanie aplikacji. Poza wyświetlaniem interfejsu aplikacja nic nie będzie robić. Efekt powinien być podobny do tego na rysunku 2.3.



Rys. 2.3. Efekt wykonania ćwiczenia 1

2.4 Podstawowa składowa aplikacji – Aktywność (Activity)

Jednym z podstawowych składników aplikacji dla systemu Android są *aktywności*. Aktywności mogą być wywoływane bezpośrednio przez użytkownika (za pomocą wspomnianego wcześniej lauchera), ale również jedna aktywność może uruchomić drugą. Możliwe jest nawet wywoływanie aktywności należących do innych aplikacji (np. aplikacji do wysyłania e-maili czy obsługi aparatu fotograficznego). Ze względu na to, że ani programista aplikacji, ani jej użytkownik nie mają stuprocentowej kontroli nad *cyklem życia aktywności*, ważne jest poprawne zapewnienie obsługi wszystkich możliwych sytuacji.

2.5. Cykl życia aktywności

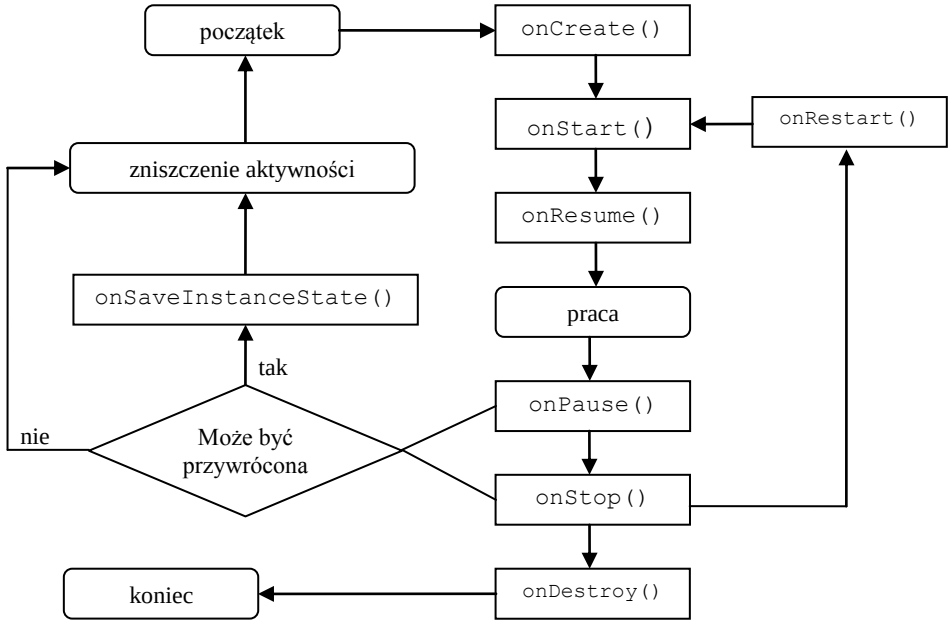
Cykl życia aktywności jest bardzo istotny. Zapewnienie poprawnej obsługi zdarzeń z nim związanych ma wpływ nie tylko na ogólne wrażenia użytkownika, lecz co ważniejsze, na poprawność zachowania danych przetwarzanych w aplikacji a także na czas działania urządzenia mobilnego bez podłączania ładowarki. W szkielecie aktywności przedstawionym na listingu 2.3 znajdują się metody odpowiedzialne za cykl życia aktywności. Para metod `onCreate()` i `onDestroy()` obejmuje cały cykl życia aktywności. Metody `onStart()` i `onStop()` obejmują okres, w którym aktywność jest widoczna dla użytkownika (niekoniecznie na pierwszym planie). Metody `onResume()` i `onPause()` wywoływane są odpowiednio na początku i końcu okresu, w którym aktywność jest na pierwszym planie i posiada focus (pojawienie się okna dialogowego lub wejście urządzenia w stan uśpienia powoduje, że aktywność schodzi z pierwszego planu).

Listing 2.3. Przykładowa aktywność

```
public class PrzykładowaAktywnosc extends Activity
{
    //Aktywność jest tworzona
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
      //Tworzenie komponentów, ustawienie obsługi zdarzeń,
      //odczytanie zapisanego stanu aktywności z obiektu
      //savedInstanceState
    }
```

```
    //Aktywność za chwilę stanie się widoczna
    @Override
    protected void onStart()
    {
        super.onStart();
       //Tworzenie elementów niezbędnych do uaktualniania
        //interfejsu użytkownika
    }

      //Aktywność jest na pierwszym planie
      @Override
      protected void onResume()
      {
           super.onResume();
      }

      //Aktywność traci "focus". Zostanie zapauzowana.
      @Override
      protected void onPause()
      {
           super.onPause();
           //tutaj należy zwolnić zasoby i ew. zapisać istotne
           //elementy stanu ponieważ aktywność może zostać
           //"zabita" przez system
      }

    //Aktywność nie jest widoczna. Została wstrzymana.
    @Override
    protected void onStop()
    {
         super.onStop();
         //tutaj należy zwolnić zasoby i ew. zapisać istotne
         //elementy stanu ponieważ aktywność może zostać
         //"zabita" przez system
    }

  //Za chwilę aktywność zostanie zniszczona
  @Override
  protected void onDestroy()
  {
      super.onDestroy();
      //tutaj należy zwolnić zasoby i ew. zapisać istotne
      //elementy stanu ponieważ aktywność może zostać "zabita"
      //przez system
  }
}
```

Na schemacie z rysunku 2.4 przestawiono kompletny cykl życia aktywności – od momentu utworzenia do zniszczenia. Jak już wspomniano, ani użytkownik, ani programista nie ma pełnej kontroli nad cyklem życia aktywności. Przykładowo w sytuacji, w której użytkownik uruchomi zbyt dużo aplikacji powodując brak pamięci, system operacyjny może „zabić" niektóre aplikacje, powodując w ten sposób niespodziewane zakończenie aktywności. Wynika stąd potrzeba implementacji metod `onPause()`, `onStop()` i `onDestroy()`, które powinny w takiej sytuacji zapisać wszystkie istotne dane (przed zniszczeniem aktywności wykonanie przynajmniej jednej z tych metod jest gwarantowane). Innym uzasadnieniem dla konieczności poprawnej obsługi cyklu życia aktywności jest oszczędzanie energii (szczególnie istotne w urządzeniach mobilnych) – np. wyświetlana animacja powinna być zatrzymana w momencie, gdy aktywność staje się niewidoczna (np. gdy użytkownik odebrał rozmowę telefoniczną lub przełączył się na inną aplikację).



Rys. 2.4. Cykl życia aktywności

2.6. Odwoływanie się do elementów układu w kodzie aplikacji

W kodzie aplikacji często zachodzi potrzeba odwołania się do elementów zdefiniowanych w plikach XML. Odwołania są możliwe dzięki identyfikatorom (wartościom typu int), które są umieszczane w automatycznie generowanej klasie `R` (od *ang. resources*), o której wspomniano wcześniej. Identyfikatory w klasie `R` są podzielone na kategorie.

Odwołania do definicji układu mają postać `R.layout.nazwa`, gdzie ostatni element odpowiada nazwie pliku XML, w którym zdefiniowano układ interfejsu użytkownika. Przykładowo ustawienie w aktywności układu zdefiniowanego w pliku `activity_dane.xml` odbywa się w sposób następujący:

```
setContentView(R.layout.activity_dane);
```

Aby uzyskać referencję do komponentu (np. przycisku czy pola tekstowego), należy posłużyć się funkcją `findViewById()`. Odwołania do komponentów mają postać `R.id.identyfikator`, gdzie ostatni element to identyfikator określony w pliku XML następująco: „`@+id/identyfikator`”. Przykład uzyskania identyfikatora przycisku:

```
mOcenyPrzycisk = (Button)findViewById(R.id.ocenyPrzycisk);
```

Warto zwrócić uwagę na to, że metoda `findViewById()` zwraca referencję do obiektu typu `View`, więc należy go rzutować na rzeczywisty typ obiektu, który chcemy uzyskać. Odwołania do łańcuchów są analogiczne do poprzednich przypadków i mają postać `R.string.identyfikator`. Aby uzyskać łańcuch zdefiniowany w pliku `strings.xml`, należy posłużyć się funkcją `getString()` np.:

```
getString(R.string.app_name);
```

2.7. Obsługa zdarzeń

Ponieważ nasza przykładowa aplikacja „nie potrafi” jeszcze reagować na akcje użytkownika musimy zająć się *obsługą zdarzeń*. W aplikacjach dla systemu Android obsługa zdarzeń odbywa się w sposób typowy (nie licząc jednego wyjątku) dla programów tworzonych w języku Java – poprzez *implementację interfejsu słuchacza* (*ang. listener*) reagującego na odpowiedni typ zdarzeń.

2.7.1. Obsługa zdarzenia – całość kodu w klasie anonimowej

Poniżej, na listingu 2.4, zamieszczono kod obsługi kliknięcia przycisku *Oceny*, który dodajemy do naszej aplikacji. Obsługa jest zapewniana przez klasę anonimową implementującą interfejs `View.OnClickListener`. Warto zwrócić uwagę na to, że niektóre interfejsy są zdefiniowane wewnątrz innych klas np. `OnClickListener` jest zdefiniowany wewnątrz klasy `View`.

Listing 2.4. Obsługa zdarzenia – całość kodu w klasie anonimowej

```
public class DaneActivity extends Activity {

       private Button mOcenyPrzycisk;

       @Override
       protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_dane);

         mOcenyPrzycisk =
          (Button)findViewById(R.id.ocenyPrzycisk);
         mOcenyPrzycisk.setOnClickListener(
          new View.OnClickListener() {
           @Override
           public void onClick(View v) {
             Toast.makeText(DaneActivity.this,
                            getString(R.string.nie_obslugiwana),
                            Toast.LENGTH_LONG).show();
           }
         });
       }
       //... - część kodu usunięto
}
```

Jak widać na listingu zapewnienie obsługi zdarzenia polega na:

- odszukaniu widoku (w tym wypadku przycisku) na podstawie jego identyfikatora z klasy `R` generowanej automatycznie na podstawie plików XML i uzyskaniu referencji do tego widoku,
- wywołanie na rzecz tego widoku metody `set...Listener()` lub `add...Listener()` ustawiającej klasę implementującą odpowiedni interfejs. Może to być klasa anonimowa utworzona w miejscu dodawania słuchacza. Typowym miejscem, w którym rejestrowana jest obsługa zdarzeń jest metoda `onCreate()` aktywności.

Zanim zajmiemy się następnym sposobem obsługi zdarzeń, istotne jest też to, co się dzieje po kliknięciu przycisku *Oceny*. W obsłudze zdarzenia wykorzystamy klasę `Toast` służącą do konstruowania komunikatów wyświetlanych przez krótki czas w ramkach o kształcie prostokąta lub prostokąta z zaokrąglonymi rogami (stąd nazwa klasy – grzanka/tost) na tle aktywności. Warto również zwrócić uwagę na pierwszy parametr metody `makeText()`. Wymaga ona referencji *tzw. kontekstu* czyli do klasy dziedziczącej po klasie `Context`. W naszym wypadku możemy przekazać referencję do aktywności, ponieważ m.in. ona dziedziczy właśnie po klasie `Context`. Tu pojawia się problem polegający na tym, że wewnątrz klasy anonimowej słowo kluczowe `this` oznacza referencję do obiektu klasy anonimowej a nie do aktywności. Problem ten można rozwiązać poprzedzając słowo kluczowe `this` nazwą aktywności.

2.7.2 Obsługa zdarzenia – z metodą pomocniczą

Obsługa zdarzenia przycisku *Super* przedstawiona na listingu 2.5 jest dość podobna do tej z poprzedniego punktu. Tutaj jednak w klasie aktywności zdefiniowano prywatną metodę `superPrzycisk()` zawierającą cały kod obsługi zdarzenia. Klasa anonimowa obsługująca zdarzenie jedynie wywołuje wspomnianą metodę (klasy anonimowe mogą się odwoływać do prywatnych składowych klasy, w której się znajdują).

Listing 2.5. Obsługa zdarzenia – z metodą pomocniczą

```
public class DaneActivity extends Activity {

	//... - część kodu usunięto

	private Button mPrzyciskSuper;

	@Override
	protected void onCreate(Bundle savedInstanceState) {
		super.onCreate(savedInstanceState);
		setContentView(R.layout.activity_dane);
		mPrzyciskSuper =
			(Button)findViewById(R.id.superPrzycisk);
		mPrzyciskSuper.setOnClickListener(
			new View.OnClickListener() {
				@Override
				public void onClick(View v) {
					superPrzycisk(v);
				}
		});
	}
```

```
        //... - część kodu usunięto

        private void superPrzycisk(View v) {
                Toast.makeText(this,
                                getString(R.string.zaliczenie),
                                Toast.LENGTH_LONG).show();
                finish();
        }

        //... - część kodu usunięto
}
```

Warto zwrócić uwagę na to, że w tym wypadku nie występuje problem z referencją do kontekstu w wywołaniu metody `makeText()`. Metoda `finish()` wywoływana w obsłudze zdarzenia powoduje zakończenie aktywności.

2.7.3 Obsługa kliknięć na skróty

Ostatni sposób obsługi zdarzeń jest ograniczony do kliknięcia widoku. Polega on na:

- utworzeniu w kontekście (czyli np. aktywności), w którym będzie wykorzystywany układ, publicznej metody obsługującej zdarzenie. Metoda musi przyjmować parametr typu `View`,
- ustawieniu w pliku XML atrybutu `onClick`, tak aby wskazywał na zdefiniowaną metodę.

Na listingach 2.6 i 2.7 przedstawiono obsługę kliknięcia przycisku `niePoszlo` zrealizowaną w ten sposób. W aktywności definiujemy metodę publiczną `niePoszloPrzycisk()` obsługującą zdarzenie.

Listing 2.6. Obsługa kliknięć na skróty – kod w aktywności

```
public class DaneActivity extends Activity {
        //... - część kodu usunięto

        //MUSI BYĆ PUBLICZNA
        public void niePoszloPrzycisk(View v) {
                Toast.makeText(this, getString(R.string.warunek),
                                Toast.LENGTH_LONG).show();
                finish();
        }
}
```

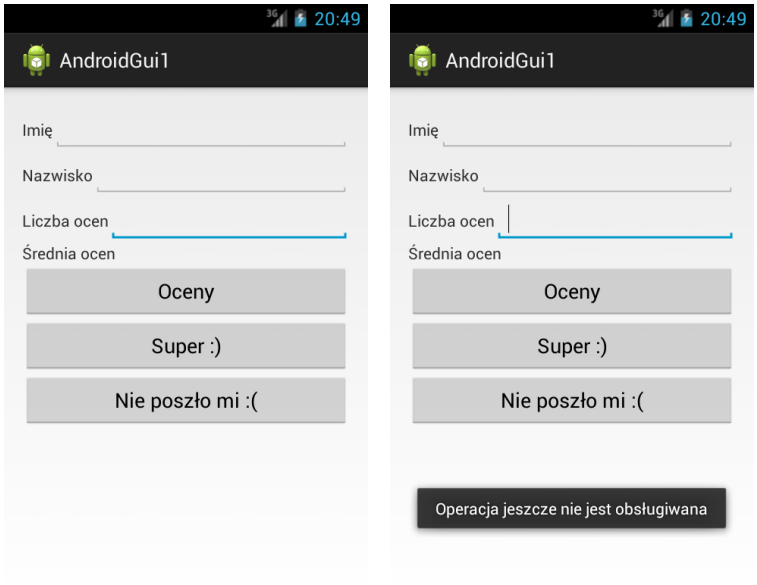
Natomiast w pliku XML opisującym układ komponentów definiujemy atrybut `onClick`, którego wartością jest nazwa metody.

Listing 2.7. Obsługa kliknięć na skróty – kod w pliku układu

```
<Button
    android:id="@+id/niePoszloPrzycisk"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:onClick="niePoszloPrzycisk"
    android:text="@string/przycisk_nie_poszlo" >
</Button>
```

Ćwiczenie 2.2 (do samodzielnego wykonania)

Wprowadź w aplikacji modyfikacje opisane w punktach 2.7.1 – 2.7.3. Oprócz plików `DaneActivity.java` i `activity_dane.xml` zmodyfikuj również plik `strings.xml`. Sprawdź działanie aplikacji.



Rys. 2.5. Efekt wykonania programu

2.7.4. Inne rodzaje zdarzeń

Istnieje wiele typów zdarzeń. Omówienie ich wszystkich wykracza poza zakres podręcznika. Można znaleźć je wszystkie w dokumentacji dostępnej na stronie [13]. Tutaj skupimy się tylko na zdarzeniach niezbędnych do wykonania ćwiczeń są to: `TextWatcher` – reagujący na zmianę zawartości w polu tesktowym (metoda rejestrująca słuchacza to: `addTextChangedListener()`), `View.OnFocusChangeListener` – reagujący na utratę focus'a (metoda rejestrująca `setOnFocusChangeListener()`) oraz `OnCheckedChangeListener` – reagujący na zaznaczenie/usunięcie zaznaczenia przycisku radiowego (metoda rejestrująca `setOnCheckedChangeListener()`). Ostatni rodzaj zdarzenia wykorzystamy w dalszej części rozdziału.

W naszej aplikacji zdarzenia dotyczące zmiany tekstu i utraty fokusu przez pole tekstowe wykorzystamy do sprawdzania poprawności danych wpisanych przez użytkownika. Gdy dane będą niepoprawne, wyświetlimy odpowiedni komunikat (zdefiniowany oczywiście w pliku `strings.xml`), a gdy dane będą poprawne pokażemy przycisk Oceny. Dlatego też zaczniemy od ukrycia wszystkich przycisków oraz etykiety zawierającej średnią. Dla elementów `sredniaEtykieta`, `ocenyPrzycisk`, `superPrzycisk`, `niePoszloPrzycisk` ustawiamy wartość atrybutu `visibility` na `gone`, tak jak to pokazano na listingu 2.8.

Listing 2.8. Ukrywanie przycisku

```
<Button
    android:id="@+id/ocenyPrzycisk"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:text="@string/przycisk_oceny"
    android:visibility="gone" >
</Button>
```

Następnie wprowadzamy do aktywności `DaneActivity` modyfikacje z listingu 2.9.

Listing 2.9. Sprawdzanie liczby ocen w reakcji na utratę fokusu pola

```
public class DaneActivity extends Activity {

    private boolean mLiczbaOcenOk = false;

    private Button mOcenyPrzycisk;
```

```
private Button mSuperPrzycisk;

private EditText mLiczbaOcenEdycja;
private String mKomunikat;

@Override
protected void onCreate(Bundle savedInstanceState) {

  // ... - część kodu usunięto

  mLiczbaOcenEdycja =
    (EditText) findViewById(R.id.liczbaOcenEdycja);
  mLiczbaOcenEdycja.setOnFocusChangeListener(
    new View.OnFocusChangeListener() {
      @Override
      public void onFocusChange(View v,
                                boolean hasFocus)
      {
        zmianaFokusuLiczbyOcen(v, hasFocus);
       }
  });
}

// ... - część kodu usunięto
private void sprawdzLiczbeOcen() {
  mLiczbaOcenOk = false;
  try {
    if (mLiczbaOcenEdycja.getText().toString().equals(""))
      mKomunikat = getString(R.string.pusta_liczba_ocen);
    else if (Integer.parseInt(
             mLiczbaOcenEdycja.getText().toString()) <= 0)
      mKomunikat = getString(R.string.za_malo_ocen);
    else if (Integer.parseInt(
             mLiczbaOcenEdycja.getText().toString()) > 10)
      mKomunikat = getString(R.string.za_duzo_ocen);
    else
      mLiczbaOcenOk = true;
  } catch (NumberFormatException e) {}
}

private void pokazPrzycisk() {
  if (mLiczbaOcenOk)
    mOcenyPrzycisk.setVisibility(Button.VISIBLE);
  else
    mOcenyPrzycisk.setVisibility(Button.GONE);
}
```

```
    private void zmianaFokusuLiczbyOcen(View v,
                                        boolean hasFocus) {
      if (hasFocus)
        return;
      sprawdzLiczbeOcen();
      if (!mLiczbaOcenOk) {
        Toast grzanka = Toast.makeText(this, mKomunikat,
                                       Toast.LENGTH_SHORT);
        grzanka.show();
      }
      pokazPrzycisk();
    }
}
```

Jak widać zastosowano tu drugą metodę obsługi zdarzeń – wykorzystującą metody pomocnicze, ponieważ zapobiega to nadmiernemu rozrostowi metody `onCreate()`. Obsługa zdarzenia (metoda `onFocusChange()`) wywołuje metodę `zmianaFokusuLiczbyOcen()`, która sprawdza, czy pole utraciło fokus, następnie wywołuje kolejną metodę `sprawdzLiczbeOcen()`. `sprawdzLiczbeOcen()` sprawdza liczbę ocen, a wynik tego sprawdzenia zapisuje w polach `mLiczbaOcenOk` oraz `mKomunikat`. Następnie `zmianaFokusuLiczbyOcen()`, w zależności od wyniku tego sprawdzenia, wyświetla odpowiedni komunikat lub wywołuje funkcję pokazującą przycisk oceny.

Ćwiczenie 2.3 (do samodzielnego wykonania)

Wprowadź do przykładowej aplikacji modyfikacje przedstawione w punkcie 2.7.4., a także dodaj sprawdzanie poprawności danych w polu `liczbaOcenEdycja` po każdej zmianie zawartości pola. Wykorzystaj interfejs `TextWatcher` i metodę `addTextChangedListener()` pola tekstowego. Interfejs `TextWatcher` ma metody: `onTextChanged()`, `beforeTextChanged()`, `afterTextChanged()` – wykorzystaj ostatnią z wymienionych.

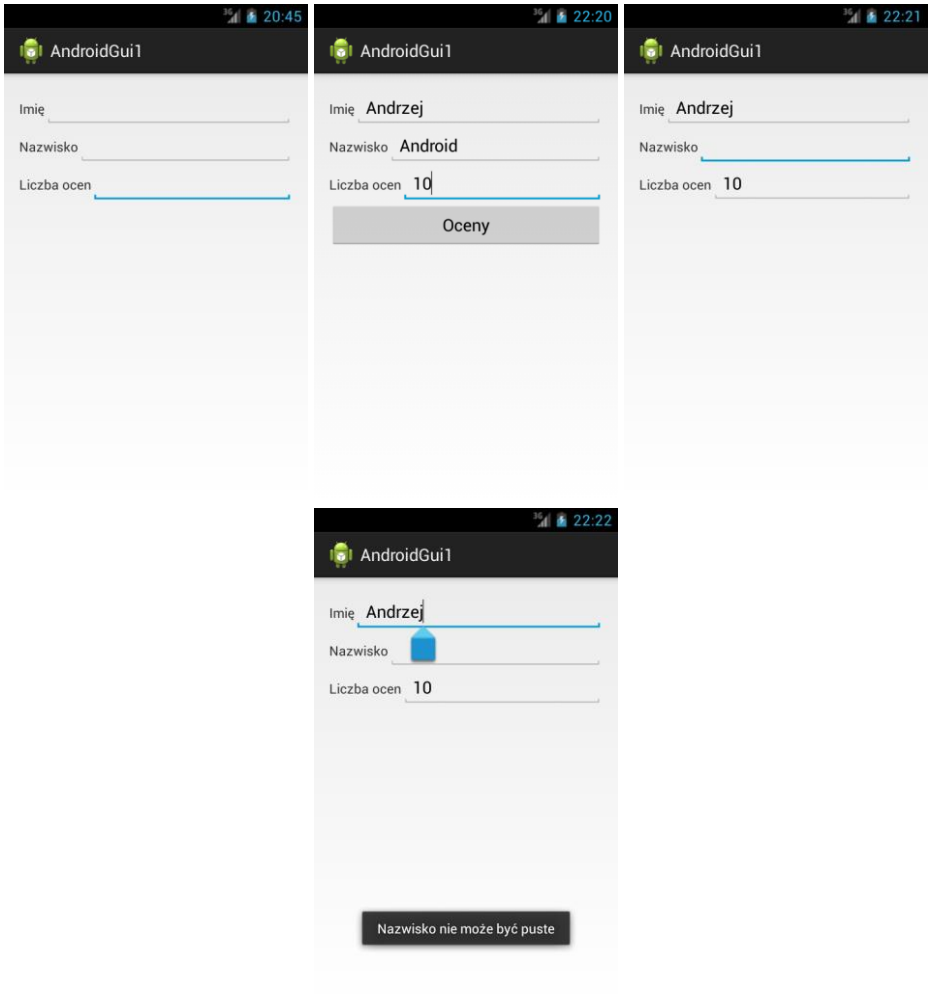
Ćwiczenie 2.4 (do samodzielnego wykonania)

Postępując analogicznie, dodaj do aplikacji sprawdzanie poprawności imienia oraz nazwiska (pola nie mogą być puste). Przycisk oceny jest widoczny tylko wtedy, gdy wszystkie dane są poprawne. W przypadku utraty fokusu przez pole zawierające nieprawidłowe dane wyświetlany jest odpowiedni komunikat. Tak jak poprzednio wykorzystaj interfejsy `TextWatcher` i `OnFocusChangeListener`. Ćwiczenie zrealizuj w sposób analogiczny do przykładu z punktu 2.7.4 tzn.:

- stwórz pola logiczne `mImieOk` i `mNazwiskoOk`

- utwórz metody pomocnicze `zmianaFokusuImienia()`, `zmianaFokusuNazwiska()`, `sprawdzImie()`, `sprawdzNazwisko()`
- dostosuj metodę `pokazPrzycisk()`, tak aby korzystała z nowych pól logicznych

Efekt wykonania ćwiczeń 2.3 i 2.4 pokazano na rysunku 2.6.

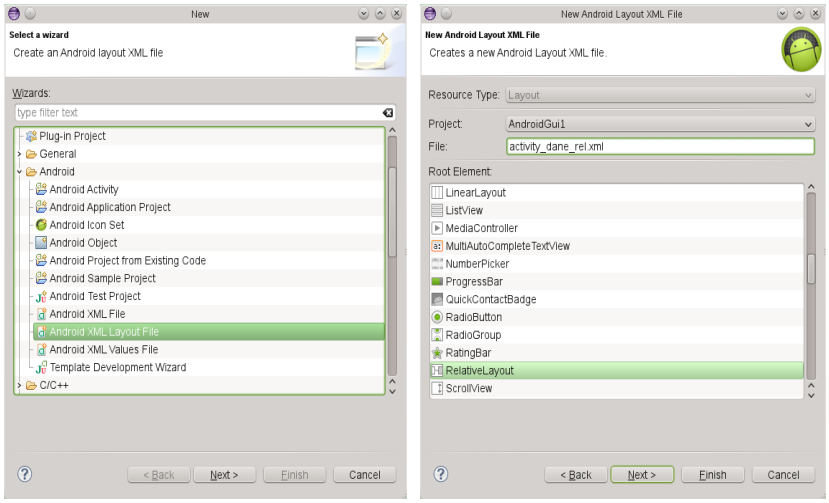


Rys 2.6. Sprawdzanie poprawności danych wprowadzanych przez użytkownika

2.8. Zaawansowane rozmieszczanie elementów – Relative Layout

Tworząc aplikacje dla Androida, mamy dostęp do wielu różnych zarządców układu elementów. Najprostszego z nich użyliśmy, tworząc układ pierwszej aktywności. Aby osiągnąć zamierzony efekt musieliśmy zagnieżdżać układy liniowe. Teraz zajmiemy się najbardziej zaawansowanym tzn. *układem względnym* (*ang. relative layout*), który pozwala uniknąć takich sytuacji. Stworzymy dla aktywności `DaneActivity` nowy układ względny, wyglądający tak jak układ z poprzednich ćwiczeń.

Aby dodać nowy układ klikamy prawym przyciskiem katalog `res/layout` w eksploratorze projektów, a następnie wybieramy: *New* | *Other* | *Android XML Layout File*.



Rys. 2.7. Tworzenie układu względnego

Wybieramy kreator nowego układu XML. Nadajemy nowemu układowi nazwę `activity_dane.xml`, a jako główny elementy wybieramy `RelativeLayout`. Na koniec zatwierdzamy ustawienia przyciskiem *Finish*.

2.8.1. Typowe właściwości

W przypadku układu względnego stosuje się więcej atrybutów niż w przypadku układu liniowego. Jak sugeruje nazwa układu, są one potrzebne do określenia położenia elementów względem siebie. Zatem zanim przystąpimy do pracy nad układem, warto zapoznać się z podstawowymi atrybutami, które wykorzystamy w ćwiczeniu dotyczącym układu względnego. Pełną listę można znaleźć na stronie [6].

Atrybuty:

- `android:layout_alignParentRight="true"`,
- `android:layout_alignParentLeft="true"`,
- `android:layout_alignParentTop="true"`,

powodują, że krawędź widoku (komponentu) zostanie zrównana z odpowiednią krawędzią obszaru zajmowanego przez układ względny.

Z kolei atrybuty:

- `android:layout_alignBaseline="@+id/identyfikatorWidoku"`,
- `android:layout_alignBottom="@+id/identyfikatorWidoku"`,
- `android:layout_alignRight="@+id/identyfikatorWidoku"`

powodują, że dana krawędź (lub linia bazowa) widoku zostanie zrównana z odpowiednią krawędzią (linią bazową) widoku, którego identyfikator podano jako wartość atrybutu.

Atrybuty

- `android:layout_toRightOf="@+id/identyfikatorWidoku"`,
- `android:layout_below="@+id/identyfikatorWidoku"`

służą do określenia położenia widoku względem innych np. poprzez podanie informacji, że widok ma się znajdować poniżej innego widoku, po lewej czy po prawej stronie. Warto też dodać, że nie wolno stosować zależności cyklicznych.

Innymi atrybutami (niezwiązanymi z układem względnym), które będziemy stosować, są:

- `android:layout_marginRight="10dp"`
- i `android:layout_marginLeft="10dp"`

Określają one, ile pustego miejsca należy dodać po zewnętrznej stronie krawędzi określonej w nazwie atrybutu. Jednostką `dp`, którą stosujemy, są tzw. piksele niezależne od gęstości (*ang. density independent pixels*). W przypadku ekranu o gęstości *160dpi*, `1dp` odpowiada jednemu pikselowi ekranu. W przypadku większej lub mniejszej gęstości Android odpowiednio przeliczy liczbę pikseli, tak aby rozmiar elementu na ekranie był taki sam.

Atrybuty, które stosowaliśmy już poprzednio, to znaczy:

- `android:layout_width="wrap_content"`
- `android:layout_height="wrap_content"`

w przypadku układu względnego mają wartość `wrap_content`, ponieważ rozmiar elementów jest określany w inny sposób.

2.8.2. Tworzenie układu względnego

Po omówieniu atrybutów możemy otworzyć plik `res/layout/activity_dane_rel.xml` w edytorze graficznym i przejść do umieszczania elementów. Umieszczamy w nim dwie etykiety i dwa pola tekstowe w sposób pokazany na rysunku 2.8. Plik XML opisujący ten układ przedstawiono na listingu 2.10. Etykietom i polom tekstowym nadajemy identyfikatory identyczne, jak te z pliku `activity_dane.xml`. Wykorzystujemy również te same łańcuchy zdefiniowane w pliku `strings.xml`.

Rys. 2.8. Częściowo ukończony układ względny

Listing 2.10. Częściowo ukończony układ względny

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

    <EditText
        android:id="@+id/imieEdycja"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentRight="true"
        android:layout_alignParentTop="true"
        android:layout_marginRight="10dp"
```

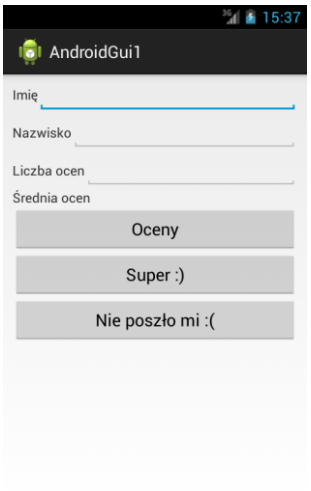
```
        android:layout_toRightOf="@+id/imieEtykieta" >

        <requestFocus />
    </EditText>

    <TextView
        android:id="@+id/imieEtykieta"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/imieEdycja"
        android:layout_alignBottom="@+id/imieEdycja"
        android:layout_alignParentLeft="true"
        android:layout_marginLeft="10dp"
        android:text="@string/etykieta_imie" />

    <TextView
        android:id="@+id/nazwiskoEtykieta"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignBaseline="@+id/nazwiskoEdycja"
        android:layout_alignBottom="@+id/nazwiskoEdycja"
        android:layout_alignParentLeft="true"
        android:layout_marginLeft="10dp"
        android:text="@string/etykieta_nazwisko" />

    <EditText
        android:id="@+id/nazwiskoEdycja"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignRight="@+id/imieEdycja"
        android:layout_below="@+id/imieEdycja"
        android:layout_toRightOf="@+id/nazwiskoEtykieta"
        android:ems="10" />

</RelativeLayout>
```

Jak widać na listingu 2.10, pierwszym dodanym elementem jest pole `imieEdycja`. Jego górna i prawa krawędź zrównana jest z krawędzią układu. Kolejnym elementem jest pole tekstowe `imieEtykieta`, jego linia bazowa i dolna krawędź zrównane są z polem tekstowym `imieEdycja`, a lewa krawędź z krawędzią układu. Kolejne pole tekstowe – `nazwiskoEdycja` umieszczone jest pod polem `imieEdycja` i po prawej stronie etykiety `nazwiskoEtykieta`. Do pola `imieEdycja` zrównana jest również prawa krawędź pola `nazwiskoEdycja`. Położenie etykiety `nazwiskoEtykieta` jest określone względem pola

`nazwiskoEdycja` (zrównane: dolna krawedź i linia bazowa) i krawędzi układu (zrównane krawędzie po lewej stronie). Elementy znajdujące się po lewej stronie mają ustawiony lewy margines o szerokości `10dp`, a elementy po prawej – prawy margines o szerokości `10dp`.

Ćwiczenie 2.5 (do samodzielnego wykonania)

Rozbuduj układ z rysunku 2.8 i listingu 2.10, tak aby wyglądał jak ten z rysunku 2.9. Użyj takich samych oznaczeń elementów jak w przypadku układu wykorzystującego układ liniowy. Ukryj przyciski i zamień stosowany układ w programie z poprzedniego ćwiczenia. Przetestuj działanie aplikacji.



Rys. 2.9. Efekt wykonania ćwiczenia

2.9. Aplikacje zawierające więcej niż jedną aktywność

Zagadnienia takie jak tworzenie projektów składających się z wielu aktywności i przekazywanie danych między nimi nie pasują najlepiej do rozdziału poświęconemu tworzeniu interfejsu użytkownika, jednak druga aktywność z listą ocen studenta będzie przydatna w demonstracji bardziej zaawansowanych zagadnień związanych właśnie z tworzeniem GUI.

Wiele osób początkujących w tworzeniu aplikacji dla Androida ma trudności z uruchomieniem projektów składających się z wielu aktywności. Typowym problemem jest brak deklaracji nowej aktywności w manifeście aplikacji. W aplikacjach przeznaczonych dla systemu Android każda aktywność – niezależnie od tego czy jest wywoływana przez zewnętrzne aplikacje – czy jest

używana tylko przez jedną aplikację, musi być wymieniona w manifeście. W przeciwnym wypadku uruchomiona aplikacja w momencie próby wywołania niezadeklarowanej aktywności zgłosi wyjątek. Na szczęście kreator nowej aktywności dodaje ją do manifestu automatycznie. Ponieważ jednak nie jest to jedyny sposób dodawania aktywności do projektu, warto pamiętać o tym potencjalnym źródle problemów.

Listing 2.11. Manifest aplikacji

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="pl.pollub.android_gui_1"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="17"
        android:targetSdkVersion="17" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="pl.pollub.android_gui_1.DaneActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name=
                   "android.intent.action.MAIN" />
                <category android:name=
                   "android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <activity
            android:name=
              "pl.pollub.android_gui_1.ListaOcenActivity"
            android:label="@string/title_activity_lista_ocen" >
        </activity>
    </application>
</manifest>
```

Listing 2.11. przedstawia manifest naszej przykładowej aplikacji. Jak widać znaczniki `<activity>` są częściami aplikacji. Główna aktywność ma dodatkowo

zdefiniowany filtr intencji (*ang. intent filter*), który w tym wypadku mówi o tym, że aplikacja ma być widoczna w menu lauchera – czyli na liście aplikacji prezentowanej użytkownikowi przez system. Druga aktywność (którą dodamy w następnym ćwiczeniu) posiada tylko minimalną liczbę atrybutów zapewniającą poprawne działanie.

Korzystając z okazji warto omówić najważniejsze elementy, jakie zawiera. Jak już widzieliśmy w rozdziale pierwszym – podczas tworzenia projektu należy podać wykorzystywane wersje API. Jak widać zapisywane są one w manifeście aplikacji. Atrybutami aplikacji zapisanymi w manifeście są też ikona, nazwa czy temat (wygląd) aplikacji.

Po utworzeniu i dodaniu aktywności do manifestu możliwe jest jej wywołanie. Przykład wywołania innej aktywności zamieszczono poniżej:

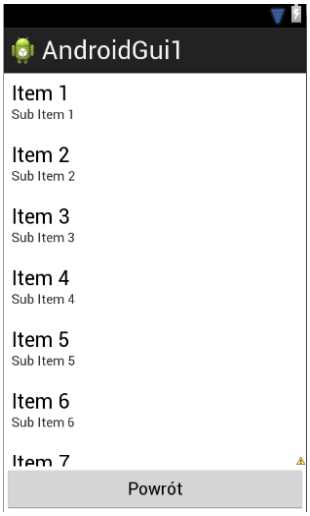
```
Intent zamiar = new Intent(BiezacaAktywnosc.this, //kontekst -
                                                  //zazwyczaj aktywność
WywolywanaAktywnosc.class);
startActivity(zamiar);
```

Rozpoczyna się ono od utworzenia obiektu klasy `Intent`. Zapisywane są w nim informacje o aktywności wywołującej. W przykładzie używamy (na razie) najprostszego wywołania na zasadzie „uruchom i zapomnij", które realizuje metoda `startActivity()`.

Ćwiczenie 2.6 (do samodzielnego wykonania)

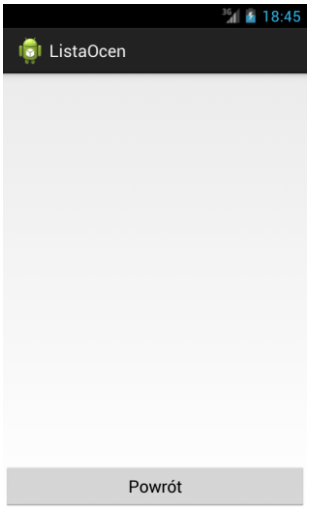
Do aplikacji przykładowej dodaj drugą aktywność o nazwie `ListaOcenActivity`, a następnie:

- w układzie elementów utworzonym wraz z aktywnością zmień układ główny ze względnego na liniowy,
- dodaj do układu komponenty `ListView` i `Button`, aby uzyskać efekt przedstawiony na rysunku 2.10. Dla elementu listy użyj identyfikatora `listaOcen`, a dla przycisku `przciskWroc`. **Wskazówka**: aby uzyskać wygląd pokazany na rysunku wykorzystuj atrybut `android:layout_weight="1"` dla elementu listy.



Rys. 2.10. Układ drugiej aktywności

- zmodyfikuj obsługę kliknięcia przycisku `ocenyPrzycisk` w aktywności `DaneActivity` (stwórz metodę `wypelnijOceny()`)
```
Intent intencja = new Intent(
  this,ListaOcenActivity.class);
startActivity(intencja);
```
- dodaj obsługę kliknięcia przycisku `przyciskWroc` w aktywności `ListaOcenActivity`. Stwórz metodę pomocniczą `przyciskZamknij()` i to ją wywołaj z klasy anonimowej obsługującej zdarzenie. Przycisk ma zamykać aktywność i powodować powrót do głównej aktywności – taki efekt można osiągnąć za pomocą metody `finish()`.
- Sprawdź działanie aplikacji. Po wpisaniu poprawnych danych i kliknięciu przycisku oceny efekt powinien być taki jak na rysunku 2.11.



Rys. 2.11. Nowa aktywność z pustą listą ocen

2.10. Adaptery

Klasy pochodne `AdapterView` czyli: `ListView`, `GridView`, `Spinner` i `Gallery` służą do wyświetlania danych pochodzących z dowolnego źródła np. z bazy danych czy karty pamięci. W takiej sytuacji zazwyczaj nie wiadomo z góry, ile komponentów będzie znajdowało się na ekranie. Muszą one być tworzone dynamicznie w trakcie działania programu. Ponadto na tego typu listach prezentowane są różne rodzaje danych w różnych układach (np. tabela w bazie może mieć różną liczbę kolumn). Połączenie między listami/galeriami (pochodnymi `AdapterView`) a źródłami danych (tablicami, bazami danych itd.) zapewniają *Adaptery* (pochodne `BaseAdapter`). Zadania adaptera są następujące:

- utworzenie komponentów (pochodnych `View`) i wypełnienie ich zewnętrznymi danymi,
- (opcjonalnie) zapewnienie przekazania danych wprowadzanych przez użytkownika do zewnętrznego źródła danych

Dostępnych jest wiele adapterów (dziedziczących po klasie `BaseAdapter`) m. in. `ArrayAdapter`, `SimpleAdapter`, `CursorAdapter` oraz `SimpleCursorAdapter`, zapewniających wypełnianie list danymi pochodzącymi z tablic lub kursorów. W przypadku, gdy lista zawiera elementy interaktywne (np. przyciski), za pomocą których użytkownik modyfikuje dane, może być konieczna implementacja własnego adaptera. My zaczniemy od najprostszego przypadku – tzn. wypełnienia listy statycznymi danymi pochodzącymi z tablicy.

2.10.1. Wypełnianie listy statycznymi danymi

Wypełnianie listy statycznymi danymi z tablicy czy listy jest dość proste, ponieważ framework Androida przewiduje odpowiedni adapter o nazwie `ArrayAdapter`. Nie daje on zbyt wielkich możliwości, jednak w większości wypadków sprawdza się bardzo dobrze.

Aby wyświetlić statyczne dane, musimy je wcześniej przygotować. Rozpoczniemy od utworzenia pojemnika na dane. Będzie to pole klasy `ListaOcenActivity` zadeklarowane następująco:

```
//może być lista innych obiektów ale muszą mieć toString
private ArrayList<Integer> mDane =
      new ArrayList<Integer>();
```

Następnym krokiem jest wypełnienie pojemnika danymi. Dlatego w metodzie `onCreate()` aktywności `listaOcenActivity` dodajemy kilka ocen za pomocą:

```
mDane.add(liczba);
```

Po dodaniu danych musimy utworzyć adapter i stworzyć połączenie między listą ocen a źródłem danych. Można to zrobić również w metodzie `onCreate()`:

Listing 2.12. Tworzenie adaptera w metodzie `onCreate()`

```
mListaOcen = (ListView) findViewById(R.id.listaOcen);
ArrayAdapter<Integer> adapter =
  new ArrayAdapter<Integer>(this,
      R.layout.wiersz_oceny_proste,
      R.id.wierszOcenyWartosc,mDane);
mListaOcen.setAdapter(adapter);
```

Sposób postępowania jest dość prosty. Na początku (jak w przypadku wszystkich odwołań do elementów GUI) musimy uzyskać referencję do obiektu listy. Następnie tworzymy nowy adapter. Parametrami konstruktora są: *kontekst*, *układ elementów* wiersza listy, *identyfikator pola*, w którym należy umieścić dane oraz *źródło danych*. Na koniec wiążemy adapter z listą.

Więcej uwagi warto poświęcić drugiemu i trzeciemu argumentowi konstruktora adaptera. Otóż Android daje nam dużą swobodę w kwestii tego, jak będzie wyglądał wiersz naszej listy. Można tam umieszczać rozmaite elementy. Aby skorzystać z tego adaptera, musimy stworzyć układ elementów wiersza

(tworzy się go dokładnie tak samo jak układ aktywności) i podać jego identyfikator do konstruktora jako drugi parametr. Wersja konstruktora, której użyjemy, pozwala nam również określić identyfikator elementu, w którym będą umieszczone dane (w innych elementach mogą się znajdować np. opisy, ikony itp.).

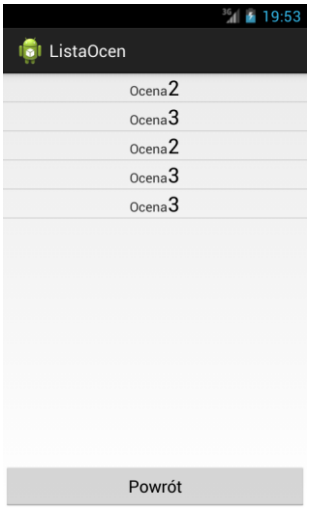
Ćwiczenie 2.7 (do samodzielnego wykonania)

Dodaj wypełnianie listy danymi statycznymi:

- utwórz układ elementów dla wiersza listy. Użyj układu liniowego o ustawieniu poziomym. Dodaj do niego dwie etykiety o identyfikatorach `wierszOcenyNazwa` oraz `wierszOcenyWartosc`. Dla pierwszej etykiety zdefiniuj odpowiedni napis w pliku `strings.xml`
- dodaj pole `mDane` do aktywności `ListaOcenActivity`,
- w metodzie `onCreate()` dodaj do pojemnika na dane 5 losowych ocen. Losowanie liczb w języku Java:

```
Random generator = new Random();
generator.nextInt(4);
```

- utwórz adapter i powiąż go z listą,
- sprawdź działanie aplikacji. Efekt powinien być podobny do tego na rysunku 2.12.



Rys. 2.12. Wypełnianie aktywności losowymi danymi

2.10.2 Przekazywanie danych do wywoływanej aktywności

Kolejnym zagadnieniem niezwiązanym bezpośrednio z tematem tworzenia interfejsu użytkownika jest przekazywanie danych do wywoływanej aktywności. Dodanie przekazywania liczby ocen z aktywności `DaneActivity` do `ListaOcenActivity` pozwoli nam na dalszą rozbudowę aplikacji i dodanie adaptera pozwalającego nie tylko na wyświetlanie, ale także na wprowadzanie danych.

Rozpoczniemy od nieznacznej modyfikacji metody uruchamiającej drugą aktywność (metoda `wypelnijOceny()` w aktywności `DaneActivity`).

Listing 2.13. Przekazywanie danych do aktywności

```
Intent intencja =
      new Intent(this,ListaOcenActivity.class);
intencja.putExtra(LICZBA_OCEN,Integer.parseInt(
      mLiczbaOcenEdycja.getText().toString()));
startActivity(intencja);
```

Jak widać powyżej (listing 2.13), modyfikacja polega na dodaniu wywołania metody `putExtra()`. Oczywiście do intencji uruchomienia aktywności można dodać więcej niż jedną daną. Istnieje wiele przeciążanych wersji metody `putExtra()`. Umożliwiają one dodawanie do intencji danych różnych typów. Wszystkie wersje tej metody posiadają dwa argumenty:

- etykietę tekstową (typu `String`) pozwalającą odróżnić dodawaną wartość od innych zapisanych w intencji,
- zapisywaną wartość.

Ponieważ używanie etykiet tekstowych (postaci „jakiś tekst”) w kilku miejscach kodu jest podatne na błędy – literówki, które nie zostaną wykryte przez kompilator, dobrą praktyką jest (i tak postąpimy w naszej aplikacji) definiowanie stałych, takich jak pokazano na listingu 2.14. Takiego identyfikatora używa się potem wszędzie tam, gdzie trzeba podać tekstowy identyfikator danej. Stałe są objęte systemem podpowiedzi środowiska Eclipse, a błędy w ich nazwach zostaną wykryte przez kompilator.

Listing 2.14. Tekstowy identyfikator danej

```
public class DaneActivity extends Activity {
      public final static String LICZBA_OCEN="liczba ocen";
      //... - część kodu usunięto
}
```

Z wymaganych modyfikacji pozostało tylko odczytywanie danych w aktywności `ListaOcenActivity`. Najbardziej typowym miejscem odczytywania danych przekazanych do aktywności jest metoda `onCreate()`. Przykład odczytania liczby ocen wprowadzonej przez użytkownika przedstawiono na listingu 2.15.

Listing 2.15. Odczytywanie danych w metodzie `onCreate()` aktywności `ListaOcenActivity`

```
if (savedInstanceState == null) {
       Bundle dodatkoweInfo = getIntent().getExtras();
       mLiczbaOcen =
         dodatkoweInfo.getInt(DaneActivity.LICZBA_OCEN);
} else
       mLiczbaOcen = 5;
```

Na początku metoda sprawdza, czy nie otrzymała zapisanego wcześniej stanu aktywności. Jeżeli go nie otrzymała, oznacza to, że jest tworzona od początku, a nie odtwarzana po zmianie konfiguracji (`savedInstanceState==null`). Jeżeli natomiast otrzymała zapisany poprzednio stan to odczytuje intencję, która spowodowała uruchomienie aktywności, a z intencji pobiera obiekt klasy `Bundle`. Z tego obiektu wydobywa liczbę ocen, korzystając z jej tekstowego identyfikatora. Wartość liczby ocen zapisywana jest w polu klasy `mLiczbaOcen`.

Ćwiczenie 2.8 (do samodzielnego wykonania)

Korzystając z informacji z punktu 2.10.2 zmodyfikuj aplikację przykładową, tak aby lista składała się z takiej liczby losowych ocen, jaką wprowadził użytkownik.

2.10.3 Adapter interaktywny – model danych

Adapter, którego używaliśmy do tej pory, nie pozwalał na modyfikację danych. Teraz zmodyfikujemy przykładową aplikację, tak aby użytkownik mógł wprowadzić oceny, a na ich podstawie była obliczana średnia.

Oceny będziemy przechowywać w obiekcie klasy `ArrayList`, którego elementami będą tym razem obiekty klasy `ModelOceny` (zamiast obiektów klasy `Integer`). Każdy z nich zawiera nie tylko wartość oceny, ale także jej nazwę (dopuszczamy tylko wartości: 2, 3, 4 oraz 5). Fragment modelu przedstawiono na listingu 2.16.

Listing 2.16. Model służący do przechowywania ocen

```
public class ModelOceny {
	private String nazwa;
	private int ocena;

	public ModelOceny(String nazwa) {
		this.nazwa = nazwa;
		ocena = 2;
	}

	public ModelOceny(String nazwa, int ocena) {
		this.nazwa = nazwa;
		this.ocena = ocena;
	}

	//... - metody dostępowe daj/ustaw
}
```

Ćwiczenie 2.9 (do samodzielnego wykonania)

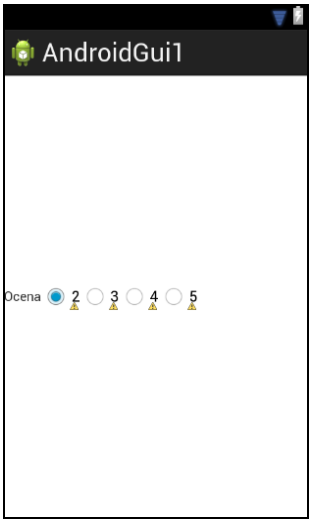
Dodaj do projektu przykładowej aplikacji klasę `ModelOceny`, której fragment znajduje się na listingu 2.16. Uzupełnij model metodami dostępowymi, których nazwy zaczynają się od słów `daj` lub `ustaw`. Metody mają pozwalać na odczytywanie/ustawianie wartości pól.

Ćwiczenie 2.10 (do samodzielnego wykonania)

Stwórz układ wiersza listy i zapisz go w pliku: `wiersz_oceny.xml`. Układ powinien:

- być liniowy o ustawieniu poziomym,
- zawierać jedną etykietę tekstową (id: `wierszOcenaEtykieta`), jedną grupę przycisków radiowych (id: `grupaRadioOceny`),
- w grupie przycisków radiowych powinny znajdować się 4 przyciski radiowe (id: `radioOcenaX`, gdzie `X` to wartość oceny), etykiety przycisków powinny odpowiadać wartościom ocen.

Efekt, który należy uzyskać w wyniku ćwiczenia przedstawiono na rys. 2.13.



Rys. 2.13. Układ wiersza interaktywnej listy ocen

2.10.4 Adapter interaktywny – tworzenie własnego adaptera

Aby móc modyfikować dane znajdujące się na liście, musimy zdefiniować własny adapter. Najprostszym rozwiązaniem jest stworzenie klasy dziedziczącej po istniejącym, podobnym adapterze. Wykorzystamy w tym celu istniejącą już klasę `ArrayAdapter` i na jej podstawie zbudujemy `InteraktywnyAdapterTablicy`. Zdefiniujemy w nim tylko: konstruktor, metodę `getView()`, dwie metody pomocnicze oraz dwa następujące pola:

```
private List<ModelOceny> listaOcen;
private Activity kontekst;
```

Konstruktor adaptera jest prosty, jego działanie sprowadza się do wywołania konstruktora klasy nadrzędnej i ustawienia wartości nowych pól. Konstruktor przedstawiono na listingu 2.17.

Listing 2.17. Konstruktor interaktywnego adaptera

```
public InteraktywnyAdapterTablicy(Activity kontekst,
            List<ModelOceny> listaOcen) {
      super(kontekst, R.layout.wiersz_oceny, listaOcen);
      this.kontekst = kontekst;
      this.listaOcen = listaOcen; }
```

Najbardziej skomplikowanym elementem nowego adaptera jest metoda `getView()`. Jej zadaniem jest (ewentualne) utworzenie i (zawsze) wypełnienie wiersza listy danymi. Dzięki modyfikacji metody `getView()` możemy zapewnić również obsługę zdarzeń polegających na wybraniu przez użytkownika określonego przycisku radiowego. Metodę `getView()` stworzoną na potrzeby naszej aplikacji przedstawiono na listingu 2.18.

Listing 2.18. Metoda `getView()` odpowiedzialna za utworzenie wiersza listy

```
@Override
public View getView(int numerWiersza,
                    View widokDoRecyklingu,
                    ViewGroup parent) {
  View widok = null;

  if (widokDoRecyklingu == null) {
    LayoutInflater pompka = kontekst.getLayoutInflater();
    widok = pompka.inflate(R.layout.wiersz_oceny, null);
    RadioGroup grupaOceny =
      (RadioGroup) widok.
      findViewById(R.id.grupaRadioOceny);
    grupaOceny.setOnCheckedChangeListener(
      new RadioGroup.OnCheckedChangeListener() {
        @Override
        public void onCheckedChanged(
                              RadioGroup grupaOceny,
                              int idWybranegoButtona)
        {
          aktualizujModelOceny(grupaOceny,
                               idWybranegoButtona);
        }
    });

    grupaOceny.setTag(listaOcen.get(numerWiersza));
  } else {
    widok = widokDoRecyklingu;
    RadioGroup grupaOceny =
      (RadioGroup) widok.
        findViewById(R.id.grupaRadioOceny);
    grupaOceny.setTag(listaOcen.get(numerWiersza));
  }
  TextView etykieta =
    (TextView) widok.
      findViewById(R.id.wierszOcenaEtykieta);
```

```
        etykieta.setText(
          listaOcen.get(numerWiersza).dajNazwe());
        RadioGroup grupaOceny = (RadioGroup) widok.
          findViewById(R.id.grupaRadioOceny);
        ustawOcene(grupaOceny, numerWiersza);
        return widok;
    }
```

Jak już wspomniano zadaniem metody `getView()` jest stworzenie lub ponowne ustawienie zawartości wyświetlanego elementu listy. Istniejące wiersze listy wykorzystywane są ponownie, ponieważ zazwyczaj na ekranie jest widocznych tylko kilka/kilkanaście wierszy listy. Ponowne wykorzystanie istniejących obiektów pozwala na: zaoszczędzenie pamięci (tworzonych jest mniej obiektów), przyspieszenie działania programu (zmiana danych jest szybsza). Działanie `getView()` jest następujące:

- jeżeli tworzony jest nowy element listy (`widokDoRecyklinkgu==null`):
  - element (wiersz listy) tworzony jest na podstawie układu zapisanego w pliku XML za pomocą obiektu klasy `LayoutInflater` (każdy wiersz zawiera etykietę i grupę przycisków radiowych),
  - dla grupy przycisków radiowych znajdujących się w nowym wierszu, tworzony jest słuchacz odpowiedzialny za aktualizację zewnętrznej listy ocen, w przypadku gdy użytkownik zmodyfikuje wartość zapisaną w tym wierszu (dla każdego wiersza tworzony jest osobny słuchacz odpowiedzialny za zdarzenia dotyczące tylko tego wiersza). Obsługa zdarzenia wykorzystuje metodę pomocniczą `aktualizujModelOceny()`. Warto zwrócić tutaj uwagę na wywołanie funkcji `widok.findViewById()` – jest ona wywoływana na rzecz obiektu widok – czyli na rzecz nowego wiersza listy, nie na rzecz aktywności,
  - następuje powiązanie utworzonej właśnie grupy przycisków radiowych (grupa jest elementem nowego wiersza) z obiektem w zewnętrznym źródle danych, który przechowuje ocenę. Powiązanie odbywa się poprzez ustawienie obiektu z zewnętrznej listy danych jako etykiety (`tag`) grupy przycisków radiowych – metoda `setTag()`,
- jeżeli metoda `getView()` otrzymała referencję do elementu listy „z odzysku” (`widokDoRecyklinkgu!=null`):
  - aktualizowane jest powiązanie wykorzystywanego ponownie wiersza listy z obiektem w zewnętrznym źródle danych (metoda `setTag()`),
- niezależnie od przypadku:
  - ustawiany jest tekst etykiety na podstawie nazwy oceny zapisanej

w zewnętrznej liście (podobnie jak grupa przycisków, etykieta też jest wyszukiwana w obrębie wiersza),

- za pomocą metody pomocniczej `ustawOcene()` zaznaczany jest programowo właściwy przycisk radiowy – odpowiadający wartości oceny.

Metoda pomocnicza `ustawOcene()` z listingu 2.19 wybiera odpowiedni przycisk radiowy na podstawie wartości oceny pochodzącej z modelu.

Listing 2.19. Metoda pomocnicza adaptera odpowiedzialna za zaznaczenie odpowiedniego przycisku radiowego

```
private void ustawOcene(RadioGroup grupaOceny,
                        int numerWiersza) {
      switch (listaOcen.get(numerWiersza).dajOcene()) {
      case 2:
            grupaOceny.check(R.id.radioOcena2);
            break;
      case 3:
            grupaOceny.check(R.id.radioOcena3);
            break;
      case 4:
            grupaOceny.check(R.id.radioOcena4);
            break;
      case 5:
            grupaOceny.check(R.id.radioOcena5);
            break;
      }

}
```

Metoda `aktualizujModelOceny()` z listingu 2.20 modyfikuje dane w obiekcie klasy `ArrayList`, gdy użytkownik kliknie przycisk radiowy w określonym wierszu. Ponieważ ta metoda nie otrzymuje numeru wiersza listy, wykorzystywana jest etykieta (*ang. tag*) wiążąca element źródła danych z grupą przycisków.

Listing 2.20. Metoda pomocnicza odpowiedzialna za aktualizację modelu oceny

```
private void aktualizujModelOceny(
   RadioGroup grupaOceny,int idWybranegoButtona) {
       ModelOceny element =
```

```
            (ModelOceny) grupaOceny.getTag();
        switch (idWybranegoButtona) {
        case R.id.radioOcena2:
              element.ustawOcene(2);
              break;
        case R.id.radioOcena3:
              element.ustawOcene(3);
              break;
        case R.id.radioOcena4:
              element.ustawOcene(4);
              break;
        case R.id.radioOcena5:
              element.ustawOcene(5);
              break;
        }
}
```

Aby skorzystać z nowego adaptera musimy zmodyfikować pole przechowujące dane w aktywności `ListaOcenActivity`:

```
private ArrayList<ModelOceny> mDane;
```

Następnie w metodzie `onCreate()` modyfikujemy wypełnianie pojemnika `mDane` początkowymi wartościami ocen:

Listing 2.21. Ustawianie wartości początkowych ocen

```
mDane = new ArrayList<ModelOceny>();
for (int i = 0; i < mLiczbaOcen; i++)
      mDane.add(new ModelOceny("ocena " + (i + 1)));
```

Wreszcie modyfikujemy linię kodu odpowiedzialną za utworzenie adaptera (również w metodzie `onCreate()`)

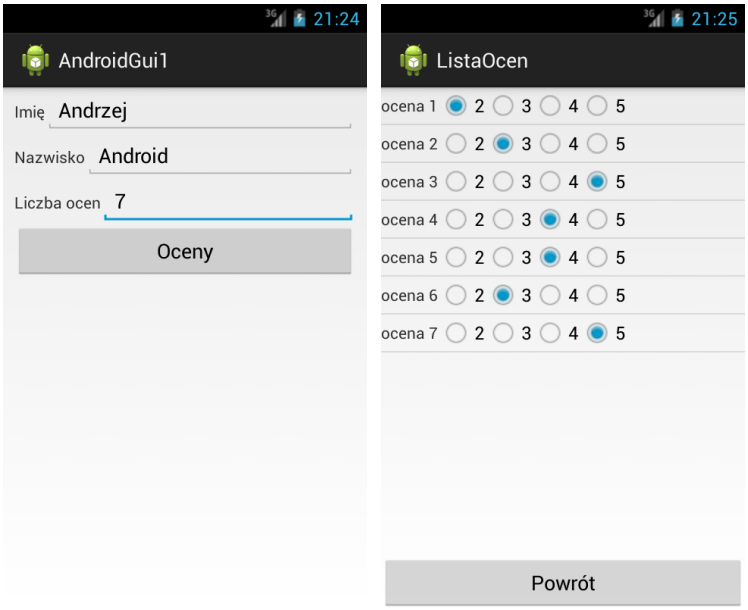
```
InteraktywnyAdapterTablicy adapter =
      new InteraktywnyAdapterTablicy(this, mDane);
```

Ćwiczenie 2.11 (do samodzielnego wykonania)

Na podstawie punktu 2.10.4:

- stwórz kompletną klasę `InteraktywnyAdapterTablicy` dziedziczącą po klasie `ArrayAdapter<ModelOceny>` i zawierającą omówione metody,
- wprowadź modyfikacje w klasie `ListaOcenActivity`, tak aby korzystała z nowego adaptera,
- sprawdź działanie aplikacji. W ramach testów sprawdź, co się stanie po obróceniu urządzenia z orientacji pionowej do poziomej, gdy wyświetlana jest lista z ocenami (obrócić emulator można za pomocą kombinacji *Ctrl+F11*). Co się dzieje, gdy w trakcie obracania urządzenia wyświetlana jest aktywność z danymi?

Efekt wykonania ćwiczenia przedstawiono na rys. 2.14.



Rys 2.14. Działanie aplikacji z interaktywnym adapterem tablicy

2.11 Zachowywanie stanu aktywności

Z ostatniego punktu ćwiczenia 2.11 wynika, że stan aktywności nie zawsze jest zachowywany. W przypadku `DaneActivity` wszystko działa jak należy,

jednak w przypadku `ListaOcenActivity` dane są tracone. Problem związany jest z tym, że domyślnie, przy zmianie konfiguracji Android niszczy aktywność i natychmiast tworzy ją on nowa. Przykładem zmiany konfiguracji jest właśnie zmiana rozdzielczości ekranu spowodowana obróceniem urządzenia przez użytkownika. Stan niektórych komponentów interfejsu użytkownika zostanie zapisany automatycznie, ale na przykład o zapisanie wartości pól klasy zawierających wyniki obliczeń musi zadbać programista.

Jest to zagadnienie związane z cyklem życia aktywności. Metoda `onSaveInstanceState()` jest wywoływana przed metodą `onStop()` i w miarę możliwości przed `onPause()`. Nie ma jednak gwarancji, że zostanie wywołana np. gdy użytkownik opuści aktywność, naciskając przycisk wstecz. Służy ona wyłącznie do przechowywania przejściowego stanu aktywności. Przechowywanie tego stanu jest konieczne do zapewnienia poprawnej obsługi zmiany konfiguracji urządzenia.

Zapisanie stanu odbywa się poprzez umieszczenie istotnych danych w obiekcie klasy `Bundle`.

```
protected void onSaveInstanceState(Bundle outState)
{
        super.onSaveInstanceState(outState);
        //zapisanie danych do outState
}
```

Obiekt `outState` wypełniony w metodzie `onSaveInstanceState()` jest przekazywany jako `savedInstanceState` do następnego wywołania metody `onCreate()`. UWAGA! Metoda `onSaveInstanceState()` nie służy do trwałego przechowywania informacji.

Poprawimy teraz naszą aplikację, tak aby wpisane przez użytkownika oceny nie były tracone w momencie, gdy użytkownik obróci urządzenie. Zaczniemy od zdefiniowania stałej tekstowej, której będziemy używać podczas zapisywania danych w obiekcie klasy `Bundle`. Stałą zdefiniujemy w aktywności `ListaOcenActivity`:

```
public final static String OCENA="ocena";
```

Aby oceny były zachowywane zdefiniujemy dwie metody (osobna metoda `spakujOceny()` przyda nam się później). Metody przedstawiono na listingu 2.22. Warto zauważyć, że zapisywanie danych do obiektu typu `Bundle` jest bardzo podobne do dodawania danych do intencji.

Listing 2.22. Zachowywanie ocen

```
@Override
protected void onSaveInstanceState(Bundle outState) {
      super.onSaveInstanceState(outState);
      spakujOceny(outState);
}

private void spakujOceny(Bundle tobolek) {
      tobolek.putInt(DaneActivity.LICZBA_OCEN,
                  mLiczbaOcen);
      for (int i = 0; i < mLiczbaOcen; i++)
            tobolek.putInt(OCENA + i,
                        mDane.get(i).dajOcene());
}
```

Pozostało nam tylko dodanie przywracania danych wpisanych przez użytkownika w metodzie `onCreate()`. Sposób przywracania danych przedstawiono na listingu 2.23. Polega on na sprawdzeniu, czy metoda `onCreate()` otrzymała wcześniej zachowany stan jako obiekt `savedInstanceState`, a następnie odczytaniu kolejnych ocen z tego obiektu.

Listing 2.23. Przywracanie ocen w metodzie `onCreate()` aktywności `ListaOcenActivity`

```
if (savedInstanceState == null) {
  // wypełnianie modelu wartościami początkowymi
  // ...
} else {
  mLiczbaOcen =
    savedInstanceState.getInt(DaneActivity.LICZBA_OCEN);
  for (int i = 0; i < mLiczbaOcen; i++)
    mDane.add(new ModelOceny("ocena " + (i + 1),
  savedInstanceState.getInt(OCENA + i)));
}
```

Ćwiczenie 2.12 (do samodzielnego wykonania)

Wprowadź modyfikacje w aktywności `ListaOcenActivity`. Sprawdzić, czy teraz dane są zachowywane przy obrocie urządzenia.

2.12. Przekazywanie wyników z aktywności

Kolejnym pobocznym zagadnieniem, które pozwoli rozbudować naszą przykładową aplikację jest przekazywanie wyników z aktywności wywoływanej do aktywności wywołującej (wcześniej dane przekazywaliśmy w przeciwną stronę).

Modyfikacje rozpoczniemy od metody `przyciskPowrot()` odpowiedzialnej za zamknięcie aktywności `ListaOcenActivity`. Tworzymy w niej nowy tobołek (obiekt klasy `Bundle`) i umieszczamy w nim oceny. Następnie tworzymy nową intencję i umieszczamy w niej tobołek z danymi. Wreszcie ustawiamy kod zakończenia aktywności `RESULT_OK` (równy -1) w nowej intencji z wynikami aktywności i kończymy aktywność za pomocą `finish()`. Istnieją inne kody zakończenia np. `RESULT_CANCEL=0`. Można też zdefiniować własny o wartości większej od zera.

Listing 2.24. Zapisywanie wyników

```
private void przyciskPowrot() {
        Bundle tobolek = new Bundle();
        spakujOceny(tobolek);
        Intent zamiar = new Intent();
        zamiar.putExtras(tobolek);
        setResult(RESULT_OK, zamiar);
        finish();
}
```

Aby aktywność wywołująca `DaneActivity` „zainteresowała” się wynikami zwracanymi przez `ListaOcenActivity`, należy zmienić sposób uruchamiania tej drugiej na `startActivityForResult()`, tak jak to pokazano na listingu 2.25. Jak widać, parametrami metody `startActivityForResult()` oprócz obiektu klasy `Intent`, jest też wartość liczbowa – `kod_zadania`, który pozwala na późniejsze określenie, w jakim celu aktywność została uruchomiona (np. dodanie nowej notatki czy edycja istniejącej notatki). Tak więc metoda `startActivityForResult()` jest stosowana w przypadku, gdy wywoływana aktywność ma zwrócić dane do aktywności wywołującej. Gdy uruchomienie innej aktywności ma się odbywać na zasadzie „uruchom i zapomnij”, można użyć metody `startActivity()`.

Listing 2.25. Nowy sposób wywoływania aktywności `ListaOcenActivity` przez `DaneActivity`

```
private void wypelnijOceny() {
  Intent intencja =
    new Intent(this,ListaOcenActivity.class);
  intencja.putExtra(LICZBA_OCEN,
    Integer.parseInt(
      mLiczbaOcenEdycja.getText().toString()));
  startActivityForResult(intencja,1);
}
```

Ostatnią modyfikacją niezbędną do zakończenia przekazywania wyników jest dodanie metody `onActivityResult()` w `DaneActivity`. Parametry tej metody pozwalają na stwierdzenie, która aktywność zakończyła wykonanie (`data` – intencja ustawiona przez `setResult()`), jaki był jej kod zakończenia (`resultCode` – ustawiony przez `setResult()`) i w jakim celu była uruchomiona (`requestCode` – przekazany do `startActivityForResult()`). Fragment metody `onActivityResult()`, jaki należy dodać do `DaneActivity` przedstawiono na listingu 2.26.

Listing 2.26. Odebranie wyników przez DaneActivity

```
protected void onActivityResult(int requestCode,
                                int resultCode,
                                Intent data) {
  super.onActivityResult(requestCode, resultCode, data);
  if (resultCode == RESULT_OK) {
    // ... ukrycie i blokowanie przycisku oceny
    Bundle tobolek = data.getExtras();
    int liczbaOcen = tobolek.getInt(LICZBA_OCEN);
    int suma = 0;
    for (int i = 0; i < liczbaOcen; i++)
      suma += tobolek.getInt(ListaOcenActivity.OCENA + i);
      // ... obliczenie i wyświetlenie średniej
      // ... pokazanie przycisku super lub nie poszło
}
```

Ćwiczenie 2.13 (do samodzielnego wykonania)

Wykonaj modyfikacje opisane w punkcie 2.13. Uzupełnij je:

- dodaj do aktywności `DaneActivity` pole `mBlokada` typu `boolean`, które

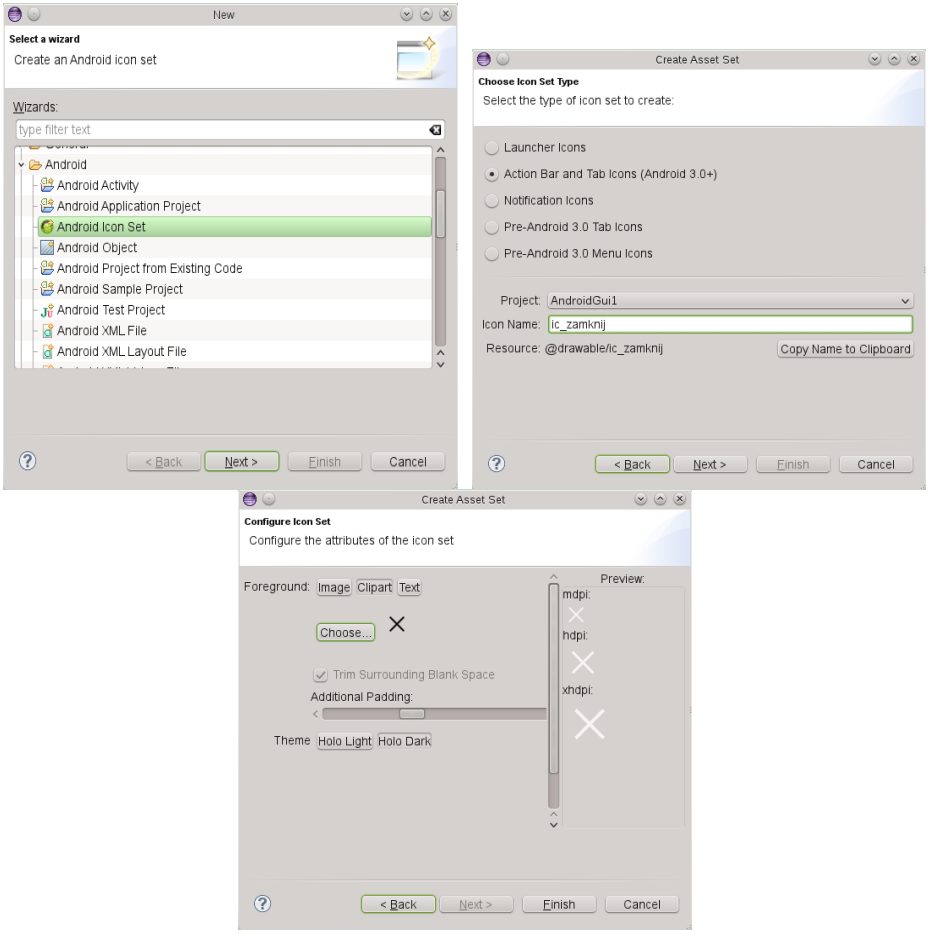
powoduje, że przycisk *Oceny* nie pojawia się po wyświetleniu obliczonej średniej (na wypadek gdyby użytkownik modyfikował pola tekstowe),
- w metodzie `onActivityResult()` dodaj ukrywanie i blokowanie przycisku *Oceny*,
- w metodzie `onActivityResult()` dodaj obliczanie i wyświetlanie średniej. Średnią umieść w etykiecie `sredniaEtykieta`. Do sformatowania liczby rzeczywistej wykorzystać można `String.format("%1.2f", srednia)`,
- również w metodzie `onActivityResult()` dodaj pokazywanie przycisku „*Super*” lub przycisku „*Nie poszło mi*”.

2.14 Tworzenie menu – pasek akcji

W tym punkcie zajmiemy się podstawami zastosowania *paska akcji* (*Action Bar*) – zamiennika menu występującego w Androidzie 3.0 i nowszych. Pokazane tu techniki są również kompatybilne ze starszymi wersjami z tą różnicą, że menu nie jest dostępne na pasku, lecz pod sprzętowym przyciskiem menu.

2.14.1 Tworzenie/dodawanie ikonek

Jest wskazane (aczkolwiek nie obowiązkowe), żeby akcje posiadały ikonki. Nie będziemy tutaj tworzyć własnych ikon od zera, ponieważ tak się składa, że ADT jest dostarczane z gotowym zestawem klipartów. Aby dodać ikonkę z klipartów (własną dodajemy tak samo), należy kliknąć prawym przyciskiem na projekcie, a następnie wybrać: *new* | *other* | *Android Icon Set*. Zostanie uruchomiony kreator pokazany na rysunku 2.15.



Rys. 2.15. Tworzenie/dodawanie ikonek

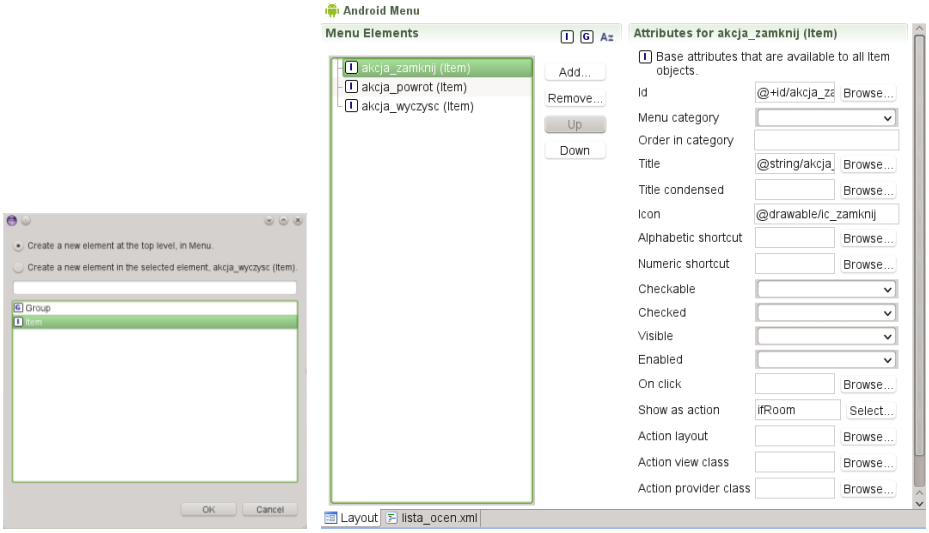
Na pierwszym etapie pracy kreatora wybieramy *Action Bar and Tab Icons*, a także nadajemy ikonie identyfikator podobny do tych, jakie stosujemy np. w układach widoków. Typowo identyfikatory zaczynają się od przedrostka `ic_`. Ponieważ zaczniemy od ikony akcji zamykania, nadajemy identyfikator `ic_zamknij`. Na kolejnym ekranie wybieramy ikonę dostępną w klipartach, a także wybieramy temat do jakiego ma być dostosowana ikona (w naszej aplikacji stosujemy *Holo Dark*). Po zakończeniu pracy kreatora w katalogach projektu `res/drawable-*` pojawią się pliki ikon dostosowane do różnych rozdzielczości. W klasie `R` pojawi się identyfikator pozwalający na korzystanie z nowej ikony.

Ćwiczenie 2.14 (do samodzielnego wykonania)

Analogicznie do przykładu z punktu 2.14.1 stwórz ikonki `ic_ok` oraz `ic_wyczysc`.

2.14.2 Tworzenie opisu paska oraz programowa obsługa akcji

Do opisu menu czy też paska akcji stosowane są typowo również pliki XML. Znajdują się one w projekcie w katalogu `res/menu`. Aby dodać nowy opis menu wystarczy wybrać ten katalog, kliknąć na nim prawym przyciskiem myszy, a następnie wybrać *New* | *Other...* | *Android XML file*. W uruchomionym kreatorze jako rodzaj pliku wybieramy Menu, a nazwę ustawiamy jako: `lista_ocen.xml`. Następnie otwieramy nowy plik w edytorze, którego wygląd pokazano na rysunku 2.16.



Rys. 2.16. Edytor menu/paska akcji

Tak jak pokazano na rys. 2.16, nową opcję dodajemy, wybierając w edytorze przycisk *Add*. W nowym oknie wybieramy z listy element `Item` i zatwierdzamy przyciskiem *Ok*. Wszystkie właściwości ustawiamy w edytorze. Do paska naszej aplikacji dodajemy 3 opcje: *zamknij*, *powrót* i *wyczyść*. Identyfikatory tych opcji to odpowiednio: `akcja_zamknij`, `akcja_powrot` i `akcja_wyczysc`. Ikonki to odpowiednio `ic_zamknij`, `ic_ok` oraz `ic_wyczysc`. Opisy akcji (*ang. title*) dodajemy do pliku `strings.xml` i odwołujemy się do nich za pomocą identyfikatorów `@string/...` . Jak zobaczymy za chwilę, opisy nie zawsze są

widoczne, lecz można je zobaczyć dzięki tzw. *długiemu kliknięciu* elementu paska akcji. Wynikowy plik XML przedstawiono na listingu 2.27.

Listing 2.27. Wynikowy plik XML opisujący pasek akcji aktywności `ListaOcenActivity`

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

    <item
        android:id="@+id/akcja_zamknij"
        android:icon="@drawable/ic_zamknij"
        android:showAsAction="ifRoom"
        android:title="@string/akcja_zamknij"/>
    <item
        android:id="@+id/akcja_powrot"
        android:icon="@drawable/ic_ok"
        android:showAsAction="ifRoom"
        android:title="@string/akcja_powrot">
    </item>
    <item
        android:id="@+id/akcja_wyczysc"
        android:icon="@drawable/ic_wyczysc"
        android:showAsAction="ifRoom"
        android:title="@string/akcja_wyczysc">
    </item>

</menu>
```

Aby aktywność wyświetlała pasek akcji, należy uzupełnić treść automatycznie wygenerowanej metody, tak jak pokazano na listingu 2.28. Za generowanie obiektów menu odpowiedzialna jest metoda `inflate()` klasy `MenuInflater`.

Listing 2.28. Tworzenie menu na podstawie pliku XML w aktywności `ListaOcenActivity`

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
      getMenuInflater().inflate(R.menu.lista_ocen, menu);
      return true;
}
```

Ostatnim etapem tworzenia paska akcji jest zapewnienie obsługi poszczególnych akcji. Służy do tego metoda `onOptionsItemSelected()`. Otrzymuje ona obiekt opisujący wybraną akcję. Z obiektu można odczytać identyfikator wybranej akcji i wykonać odpowiednie działanie. Na listingu 2.29. pokazano wersję metody dla aktywności `ListaOcenActivity`.

Listing 2.29. Metoda zapewniająca obsługę wybranej akcji w aktywności `ListaOcenActivity`

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
	switch (item.getItemId()) {
	case R.id.akcja_powrot:
		przyciskPowrot();
		break;
	case R.id.akcja_wyczysc:
		przyciskWyczysc();
		break;
	case R.id.akcja_zamknij:
		przyciskZamknij();
		break;
	}
	return true;
}
```

Metoda `onOptionsItemSelected()` korzysta z metod pomocniczych. Jedną z nich zdefiniowaliśmy wcześniej. Dwie nowe odpowiedzialne za czyszczenie danych i zamykanie aktywności (bez wysyłania wprowadzonych ocen) zamieszczono na listingu 2.30.

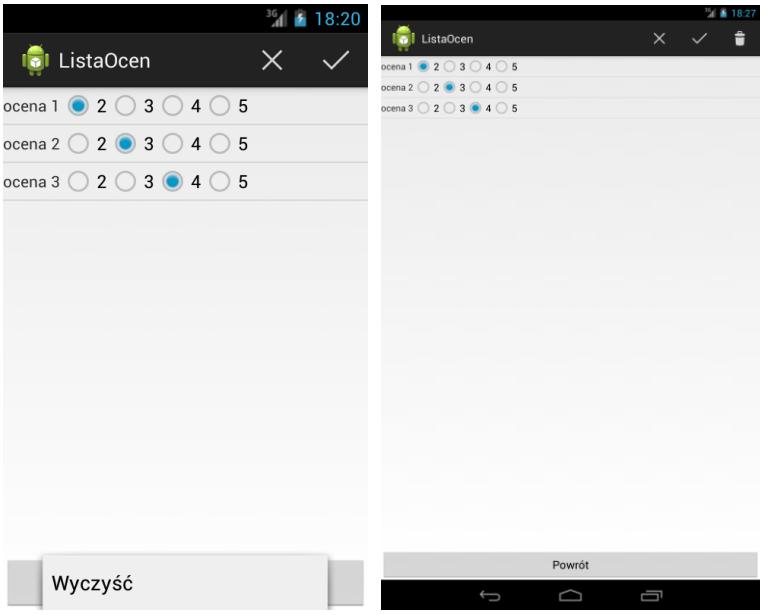
Listing 2.30. Metody pomocnicze wykorzystywane przy obsłudze akcji

```
private void przyciskWyczysc() {
	mDane.clear();
	for (int i = 0; i < mLiczbaOcen; i++)
		mDane.add(new ModelOceny(
			"ocena " + (i + 1)));
	InteraktywnyAdapterTablicy adapter =
		(InteraktywnyAdapterTablicy) mListaOcen.
			getAdapter();
	adapter.notifyDataSetChanged();
}
```

```
private void przyciskZamknij() {
        setResult(RESULT_CANCELED);
        finish();
}
```

Warto zwrócić uwagę na to, jak o zmianie danych w pojemniku `mDane` informowany jest adapter – bez wywołania metody `notifyDataSetChanged()` zawartość listy prezentowanej użytkownikowi nie zmieni się, gdy zawartość pojemnika `mData` zostanie zmodyfikowana.

Efekt dodania paska akcji został pokazany na rysunku 2.17.



Rys. 2.17. Aplikacja z paskiem akcji. Po lewej na urządzeniu ze sprzętowym przyciskiem menu, po prawej na urządzeniu z dużym ekranem bez przycisku menu

Ćwiczenie 2.15 (do samodzielnego wykonania)

Sprawdź co się stanie, jeżeli wartość atrybutu `android:showAsAction="ifRoom"` zmienimy na inne wartości np. `always`, `never`. Przetestuj te wartości na urządzeniach ze sprzętowym przyciskiem menu i bez. Co się dzieje, jeżeli dłużej przytrzymamy naciśniętą opcję na pasku akcji?

Ćwiczenie 2.16 (do samodzielnego wykonania)

Dodaj do głównej aktywności `DaneActivity` akcje pozwalające na wyczyszczenie danych, przejście do listy ocen oraz zamknięcie aplikacji. Analogicznie do powyższego przykładu użyj identyfikatorów `akcja_powrot`, `akcja_oceny`, `akcja_zamknij` (będą istotne dla następnego ćwiczenia). Również w tym wypadku utwórz metody pomocnicze `przyciskWyczysc()` i `przyciskZamknij()`.

2.14.3 Modyfikacja akcji paska w trakcie wykonania programu

Akcje znajdujące się na pasku *Action Bar* można modyfikować w trakcie wykonania programu. W naszej aplikacji akcją, która nie powinna być zawsze dostępna jest ta, która pozwala na przejście z `DaneActivity` do `ListaOcenActivity` – czyli odpowiednik przycisku *Oceny*. Za każdym razem, gdy przycisk się pojawia, powinna uaktywniać się również ta akcja.

Do zmieniania właściwości akcji służy metoda `onPrepareOptionsMenu()`. Jest ona wywoływana przez Androida, gdy zachodzi konieczność aktualizacji paska *Action Bar*. Na listingu 2.31. przedstawiono metodę, która sprawdza, czy `akcja_oceny` powinna być aktywna i ustawia jej odpowiednią właściwość (metodę `setEnabled()` użytą na listingu można zastąpić za pomocą `setVisible()`).

Listing 2.31. Ukrywanie akcji w `DataActivity`

```
@Override
public boolean onPrepareOptionsMenu(Menu menu) {
        MenuItem opcja = (MenuItem)
                menu.findItem(R.id.akcja_oceny);
        opcja.setEnabled((mImieOk && mNazwiskoOk &&
                mLiczbaOcenOk) || mBlokada);
        return super.onPrepareOptionsMenu(menu);
}
```

Metoda jest prosta. Na początku w otrzymanym obiekcie `menu`, który reprezentuje cały pasek, wyszukuje właściwą akcję. Następnie uaktywnia lub dezaktywuje ją w zależności od poprawności wprowadzonych przez użytkownika danych.

Drugą rzeczą, którą musimy się zająć, aby pasek akcji był poprawnie aktualizowany jest unieważnienie go w odpowiednim momencie. Do unieważniania zawartości paska służy metoda `invalidateOptionsMenu()`. To

ona powoduje, że Android wywoła metodę `onPrepareOptionsMenu()`. Wywołanie umieścimy w dwóch miejscach, w których modyfikujemy widoczność przycisku oceny. Miejsca te pokazano na listingu 2.32.

Listing 2.32. Wywołania metody `invalidateOptionsMenu()` w `DataActivity`

```
private void pokazPrzycisk() {
      // ... - część kodu usunięto
      invalidateOptionsMenu();
}

@Override
protected void onActivityResult(int requestCode,
                                int resultCode,
                                Intent data) {
      super.onActivityResult(requestCode, resultCode,
                             data);
      if (resultCode == RESULT_OK) {
            mBlokada = true;
            Button przyciskOceny =
              (Button) findViewById(R.id.ocenyPrzycisk);
            przyciskOceny.setVisibility(View.GONE);
            invalidateOptionsMenu();
            // ... - część kodu usunięto
      }
}
```

2.15. Fragmenty

Fragmenty są elementem wprowadzonym w *Androidzie 3.0*, który był wersją systemu przeznaczoną na *tablety* posiadające ekrany większe od ekranów telefonów. Pojawiła się wtedy potrzeba tworzenia aplikacji, które wyglądają równie dobrze na małych ekranach telefonów jak i na tabletach wyposażonych w duże ekrany. Fragmenty są właśnie odpowiedzią na to zapotrzebowanie. Umożliwiają one umieszczenie większości funkcji aplikacji we fragmentach a następnie osadzanie poszczególnych fragmentów w jednej aktywności. W ten sposób (po opracowaniu fragmentów) można niewielkim nakładem pracy przygotować aktywność w wersji dla tabletów, gdzie wszystkie funkcje mieszczą się na jednym ekranie oraz zestawu dwóch aktywności, między którymi użytkownik przełącza się podczas korzystania z aplikacji na telefonie. Innymi słowy o fragmentach można myśleć jak o mini-aktywnościach, które można łączyć w większe aktywności.

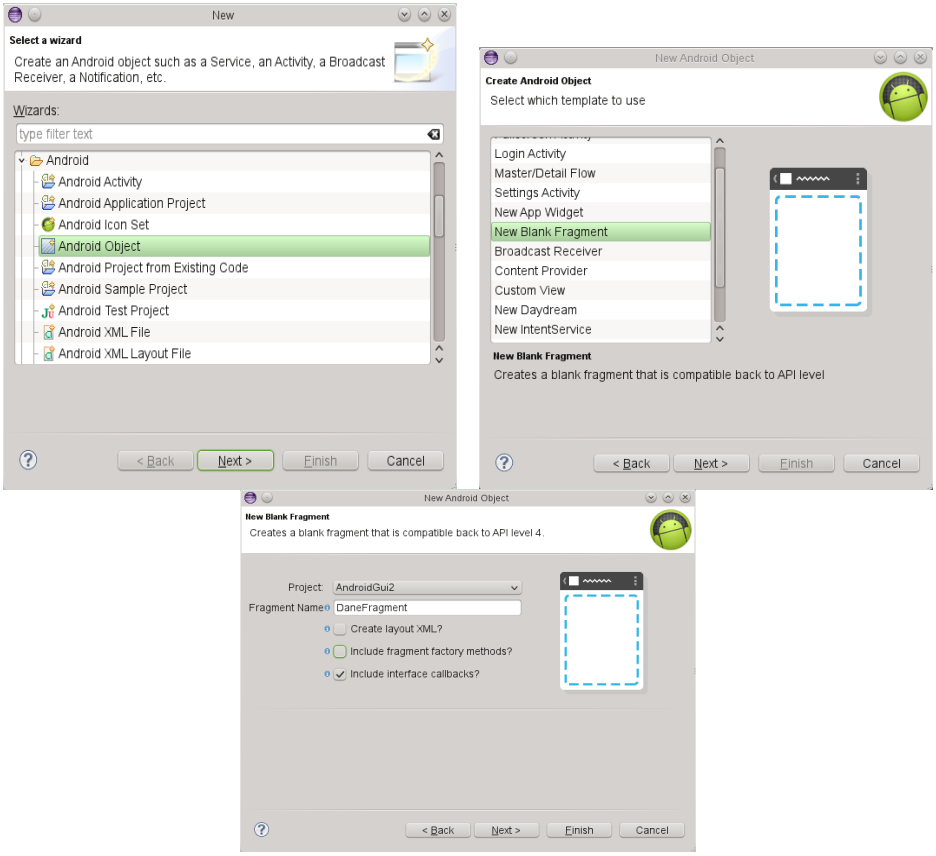
Z fragmentami zapoznamy się przerabiając naszą aplikację do obliczania średniej. Naszym celem będzie stworzenie aplikacji, która w ustawieniu pionowym urządzenia będzie korzystała z dwóch aktywności (tak jak jej obecna wersja), natomiast w ustawieniu poziomym dane studenta i jego oceny wprowadzane będą na jednym ekranie i w jednej aktywności.

2.15.1 Tworzenie fragmentów

Ponieważ w niektórych miejscach przeróbki będą dość istotne, rozpoczniemy od wykonania kopii naszego obecnego projektu. Jest to bardzo proste – wystarczy kliknąć prawym przyciskiem myszy na projekcie i z menu kontekstowego wybrać opcję *Copy* a następnie *Paste*. Projekt nazwiemy `AndroidGui2`. Następnie zmienimy nazwę pakietu naszej aplikacji – wybieramy `src/pl.pollub.android_gui_1` i w menu kontekstowym *Refactor* | *Rename.* Ustawimy nazwę `src/pl.pollub.android_gui_2`. W ten sposób powinny zostać zaktualizowane wszystkie wystąpienia nazwy pakietu w projekcie.

Większość fragmentów wymaga utworzenia układu (layout) elementów (większość – ponieważ mogą istnieć fragmenty niewidoczne). Układy takie tworzy się w ten sam sposób co układy przeznaczone dla aktywności. Dwa fragmenty, które opracujemy, będą wykorzystywały te same układy co aktywności z poprzednich ćwiczeń, tak więc ograniczymy się do zmiany nazwy istniejących plików. Zmieniamy nazwy następujących plików: `activity_dane_rel.xml` i `activity_lista_ocen.xml` odpowiednio na: `fragment_dane_rel.xml` i `fragment_lista_ocen.xml`. Usuwamy też plik `activity_dane.xml`, który nie będzie już potrzebny.

Przejdziemy teraz do tworzenia pierwszego fragmentu. Użyjemy w tym celu kreatora przedstawionego na rysunku 2.18. Uruchamiamy go, wybierając z menu kontekstowego projektu *New* | *Other...* a następnie *Android Object*.



Rys. 2.18. Tworzenie fragmentu

W trakcie pracy kreatora podajemy nazwę fragmentu: `DaneFragment`, wyłączamy tworzenie układu (ponieważ już jest gotowy) oraz tworzenie *factory methods*, które nie będą potrzebne. Po zatwierdzeniu wprowadzonych danych przyciskiem *Finish* otrzymamy kod przedstawiony na listingu 2.33.

Listing 2.33. Wygenerowany kod fragmentu

```
public class DaneFragment extends Fragment {

        private OnFragmentInteractionListener mListener;

        public DaneFragment() {}
```

```
        @Override
        public View onCreateView(LayoutInflater inflater,
                                 ViewGroup container,
                                 Bundle savedInstanceState) {
                //trzeba utworzyć obiekty layoutu
                return textView;
        }

        //onButtonPressed - niepotrzebna
        @Override
        public void onAttach(Activity activity) {
                super.onAttach(activity);
                try {
                        mListener =
                          (OnFragmentInteractionListener) activity;
                } catch (ClassCastException e) {
                        throw new ClassCastException(
                          activity.toString()
                          + " must implement "
                          + "OnFragmentInteractionListener");
                }
        }
        @Override
        public void onDetach() {
                super.onDetach();
                mListener = null;
        }
        public interface OnFragmentInteractionListener {
                //trzeba zmienić metody
        }
}
```

Wygenerowany kod składa się z kilku najważniejszych metod. Pierwszą z nich jest `onCreateView()`. Jest to metoda, której zadaniem jest utworzenie obiektów reprezentujących widoki (komponenty GUI). Zazwyczaj odbywa się to poprzez „napompowanie" układu opisanego plikiem XML, za pomocą obiektu `inflater`. Druga metoda `onAttach()` zapewnia obsługę sytuacji, w której fragment jest łączony z konkretną aktywnością. Zapisywana jest w niej referencja do podłączanej aktywności. Referencja ta – zapisana w polu `mListener` – jest istotnym elementem mechanizmu umożliwiającego komunikację między aktywnością a fragmentem. Metoda `onDetach()` jest wywoływana w momencie odłączenia fragmentu od aktywności. Ostatnim ważnym elementem w wygenerowanym automatycznie kodzie jest interfejs `OnFragmentInteractionListener` – on również jest elementem mechanizmu

komunikacji z aktywnością. W tym momencie pojawia się pytanie – jak działa ten mechanizm? Przekazywanie danych czy informacji o zdarzeniach z aktywności do fragmentu jest banalne – wystarczy we fragmencie zdefiniować metody, które będą wywoływane przez aktywność. Każda aktywność, w której umieścimy fragment, będzie mogła wywołać te metody. Większy problem istnieje w przypadku komunikacji w drugą stronę, jeżeli weźmiemy pod uwagę, że fragment ma być uniwersalny – tzn. musi istnieć możliwość osadzenia tego fragmentu w wielu różnych aktywnościach. Rozwiązaniem tego problemu jest właśnie wspomniany wewnętrzny interfejs. To w nim deklarujemy metody, które będzie wywoływał fragment w aktywności. Są to swego rodzaju komunikaty, które fragment „wysyła" do aktywności. Fragment wywołuje metody zdefiniowane w interfejsie wyłącznie za pośrednictwem pola `mListener`. Wtedy wystarczy, żeby aktywność, w której osadziliśmy fragment implementowała wewnętrzny interfejs fragmentu.

Kończąc to wprowadzenie, warto jeszcze wspomnieć, że kreator generuje fragmenty korzystające z bibliotek zgodności. Na razie nie będziemy z nich korzystać więc musimy zmienić pakiet importowany w pliku `DaneFragment.java` z:

```
import android.support.v4.app.Fragment;
```

na

```
import android.app.Fragment;
```

Wykorzystaniem biblioteki zgodności zajmiemy się później.

Przejdziemy teraz do tworzenia kodu fragmentu `DaneFragment`. Większość kodu możemy przenieść z `DaneActivity`, jednak uda się to tylko częściowo. Ze względu na inną organizację aplikacji – wiele metod jest zmodyfikowanych. Istotne elementy kodu nowego fragmentu przedstawiono na dość obszernym listingu 2.34.

Listing 2.34. Listing fragmentu DaneFragment

```
public class DaneFragment extends Fragment {

        private InterakcjaFragmentuDanychListener mListener;

        public final static String LICZBA_OCEN = "liczba ocen";

        private boolean mLiczbaOcenOk = false;
        private boolean mImieOk = false;
        private boolean mNazwiskoOk = false;
        private boolean mWynik = false;
```

```
private int mLiczbaOcen = -1;
private String mImie;
private String mNazwisko;

private Button mOcenyPrzycisk;
private Button mSuperPrzycisk;

private EditText mLiczbaOcenEdycja;
private String mKomunikat;

private EditText mImieEdycja;
private EditText mNazwiskoEdycja;

public interface InterakcjaFragmentuDanychListener {
      public void zmianaDanych(int liczbaOcen);
}

public DaneFragment() { }

@Override
public View onCreateView(LayoutInflater inflater,
                         ViewGroup container,
                         Bundle savedInstanceState) {
      View widok =inflater.inflate(
            R.layout.fragment_dane_rel, container,
            false);
      setHasOptionsMenu(true);
      return widok;
}

@Override
public void onActivityCreated(Bundle savedInstanceState) {
      super.onActivityCreated(savedInstanceState);
      // UWAGA - dopisane getView()
      mOcenyPrzycisk = (Button)
            getView().findViewById(R.id.ocenyPrzycisk);
      mOcenyPrzycisk.setOnClickListener(
            new View.OnClickListener() {
              @Override
              public void onClick(View v) {
                    klikniecie PrzyciskuOceny();
              }
      });

      //... - dla mSuperPrzycisk analogicznie
```

```
        mLiczbaOcenEdycja =
          (EditText) getView().findViewById(
            R.id.liczbaOcenEdycja);
        mLiczbaOcenEdycja
              .setOnFocusChangeListener(
                new View.OnFocusChangeListener() {
                  @Override
                  public void onFocusChange(View v,
                    boolean hasFocus) {
                    zmianaFokusuLiczbyOcen(v, hasFocus);
                  }
                });

        mLiczbaOcenEdycja.addTextChangedListener(
          new TextWatcher() {
            @Override
            public void onTextChanged(CharSequence s,
              int start, int before, int count) { }

            @Override
            public void beforeTextChanged(CharSequence s,
              int start, int count, int after) { }
              @Override
              public void afterTextChanged(Editable s) {
                sprawdzLiczbeOcen();
                // UWAGA – usunięto - pokazPrzycisk();
              }
        });

        //... - dla mImieEdycja i mNazwiskoEdycja
        //      analogicznie
}

//... - onAttach() i onDetach() pozostają niezmodyfikowane

private void sprawdzImie() {
        String noweImie = mImieEdycja.getText().toString();
        mImieOk = !noweImie.equals("");
        if (!noweImie.equals(mImie)) {
              mImie = noweImie;
              mListener.zmianaDanych(mLiczbaOcen);
        }
}
```

```
    private void sprawdzNazwisko() {
            String noweNazwisko =
                    mNazwiskoEdycja.getText().toString();
            mNazwiskoOk = !noweNazwisko.equals("");
            if (!noweNazwisko.equals(mNazwisko)) {
                    noweNazwisko = mNazwisko;
                    mListener.zmianaDanych(mLiczbaOcen);
            }
    }
    private void sprawdzLiczbeOcen() {
            mLiczbaOcenOk = false;
            int nowaLiczbaOcen = -1;
            if (mLiczbaOcenEdycja.getText().
                toString().equals("")) {
                    mKomunikat =
                      getString(R.string.pusta_liczba_ocen);
                    nowaLiczbaOcen = -1;
            } else {
                    try {
                      nowaLiczbaOcen =
                        Integer.parseInt(
                          mLiczbaOcenEdycja.getText().
                            toString());
                      if (nowaLiczbaOcen <= 0)
                        mKomunikat =
                          getString(R.string.za_malo_ocen);
                      else if (nowaLiczbaOcen > 10)
                        mKomunikat =
                          getString(R.string.za_duzo_ocen);
                      else
                        mLiczbaOcenOk = true;
                    } catch (NumberFormatException e) {
                      nowaLiczbaOcen = -1;
                    }
            }
            if (mLiczbaOcenOk) {
                    if (nowaLiczbaOcen != mLiczbaOcen)
                      mListener.zmianaDanych(nowaLiczbaOcen);
                    mLiczbaOcen = nowaLiczbaOcen;
            } else {
                    mLiczbaOcen = -1;
                    mListener.zmianaDanych(mLiczbaOcen);
            }
    }
```

```
public boolean czyWszystkieDaneOk() {
	return mImieOk && mNazwiskoOk && mLiczbaOcenOk;
}

private void zmianaFokusuImienia(View v, boolean hasFocus)
{
	if (hasFocus)
		return;
	sprawdzImie();
	if (!mImieOk)
		Toast.makeText(getActivity(),
		  getString(R.string.puste_imie),
		  Toast.LENGTH_SHORT).show();
}

private void zmianaFokusuNazwiska(View v,
                                  boolean hasFocus) {
	if (hasFocus)
		return;
	sprawdzNazwisko();
	if (!mNazwiskoOk)
		Toast.makeText(getActivity(),
		  getString(R.string.puste_nazwisko),
		  Toast.LENGTH_SHORT).show();
}
private void zmianaFokusuLiczbyOcen(View v,
                                    boolean hasFocus) {
	if (hasFocus)
		return;
	sprawdzLiczbeOcen();
	if (!mLiczbaOcenOk) {
		Toast grzanka = Toast.makeText(getActivity(),
		  mKomunikat, Toast.LENGTH_SHORT);
		grzanka.show();
	}
}

public void pokazPrzyciskOceny() {
	if (czyWszystkieDaneOk())
		mOcenyPrzycisk.setVisibility(Button.VISIBLE);
}

public void kliknieciePrzyciskuOceny() {
	Intent intencja = new Intent(getActivity(),
		ListaOcenActivity.class);
```

```
        intencja.putExtra(DaneFragment.LICZBA_OCEN,
                mLiczbaOcen);
        startActivityForResult(intencja, 1);
    }

    @Override
    public void onActivityResult(int requestCode,
        int resultCode, Intent data)
    {
        super.onActivityResult(requestCode,
                resultCode, data);
        if (resultCode == Activity.RESULT_OK) {
                Bundle oceny = data.getExtras();
                DaneFragment daneFragment =
                  (DaneFragment) getFragmentManager()
                    .findFragmentById(R.id.daneFragment);
                daneFragment.
                  obliczSredniaOcenIPokazWynik(oceny);
        }
    }
    // zmienione this na getActivity(), dodane getActivity().
    private void kliknieciePrzyciskuSuper() {
        Toast.makeText(getActivity(),
                getString(R.string.zaliczenie),
                Toast.LENGTH_LONG).show();
        getActivity().finish();
    }
    public void obliczSredniaOcenIPokazWynik(Bundle oceny) {
        //... -  obliczanie i wyświetlanie średniej jak w
        //        wersji bez fragmentów
        Button przyciskSuper =
                (Button) getView().findViewById(
                  R.id.superPrzycisk);
        Button przyciskNiePoszlo =
                (Button) getView().findViewById(
                  R.id.niePoszloPrzycisk);
        if (srednia >= 3.0) {
                przyciskSuper.setVisibility(View.VISIBLE);
                przyciskNiePoszlo.setVisibility(View.GONE);
        } else {
          przyciskSuper.setVisibility(View.GONE);
          przyciskNiePoszlo.setVisibility(View.VISIBLE);
        }
        mWynik=true;
        blokujPolaPrzyciskiIOpcje();
    }
```

```
public void blokujPolaPrzyciskiIOpcje() {
        getView().findViewById(R.id.ocenyPrzycisk).
                setVisibility(View.GONE);
        getView().
                findViewById(R.id.imieEdycja).
                  setEnabled(false);
        //... -  nazwiskoEdycja i liczbaOcenEdycja
        //       analogicznie
}
```

Przejdziemy teraz do przyjrzenia się najważniejszym elementom nowego fragmentu. W metodzie `onCreateView()` tworzymy całą hierarchię widoków za pomocą metody `inflate()`. Jej dwa najważniejsze argumenty to id układu, który należy „napompować" oraz pojemnik, który zawiera fragment.

Kolejna metoda to `onActivityCreated()`. Jest ona wywoływana w momencie, gdy aktywność zawierająca fragment skończy wykonywać swoją metodę `onCreate()`. Wtedy mamy pewność, że aktywność istnieje i można się do niej odwołać. W tej metodzie ustawiamy obsługę zdarzeń takich jak kliknięcie przycisku czy zmiana zawartości pola tekstowego. Odbywa się to praktycznie tak samo jak w przypadku aktywności. Najważniejszą różnicą jest dodane wywołanie metody `getView()` przed `findViewById()` tzn. `getView().findViewById()` zamiast samego `findViewById()`. Wynika to z faktu, iż fragment w odróżnieniu od aktywności nie zawiera tej metody. Dlatego elementów musimy szukać w widoku, który stanowi korzeń (główny element) hierarchii widoków fragmentu. Właśnie ten widok jest zwracany przez `getView()`.

W dalszej kolejności na listingu pojawiają się metody pomocnicze `sprawdzImie()`, `sprawdzNazwisko()` oraz `sprawdzLiczbeOcen()`. Choć występowały one w aktywności `DaneActivity`, tutaj zostały zmodyfikowane. Teraz metody śledzą zmiany odpowiednich danych (porównują nową i starą wartość) i w wypadku, gdy jakaś wartość się zmieniła informują o tym aktywność (za pomocą metody `zmianaDanych()` wewnętrznego interfejsu).

Warto też zwrócić uwagę, że fragmenty mają możliwość bezpośredniego wywoływania innych aktywności i odbierania od nich wyników (metody `klikniecieOcenyPrzyciskuOceny()` i `onActivityResult()`).

Kolejna drobna lecz istotna modyfikacja to metoda pomocnicza zapewniająca obsługę przycisku Super (`klikniecieOcenyPrzyciskuSuper()`). Jej zadaniem jest wyświetlenie komunikatu i zamknięcie aktywności – aby uzyskać dostęp do aktywności stosujemy metodę `getView()`.

Ostatnia istotna zmiana w funkcjach aplikacji jest taka, że przy wyświetlaniu wyniku wszystkie elementy umożliwiające wprowadzanie danych są

blokowane – służy do tego metoda pomocnicza `blokujPolaPrzyciskiIOpcje()`. Pozostałe modyfikacje to tylko efekt reorganizacji kodu (np. rozbicia fragmentu kodu na metody pomocnicze).

W podobny sposób tworzymy drugi fragment, który nazwiemy `ListaOcenFragment`. Najważniejsze jego elementy zamieszczono na listingu 2.35.

Listing 2.35. Listing fragmentu `ListaOcenFragment`

```
public class ListaOcenFragment extends Fragment {

      private InterakcjaFragmentuListyOcenListener mListener;

      private int mLiczbaOcen = -1;
      private ListView mListaOcen;
      private ArrayList<ModelOceny> mDane;

      public final static String OCENA = "ocena";

      public interface InterakcjaFragmentuListyOcenListener {
            public void ocenyWypelnione(Bundle oceny);
      }

      public ListaOcenFragment() { }

      @Override
      public View onCreateView(LayoutInflater inflater,
            ViewGroup container,Bundle savedInstanceState) {
              View widok = inflater.
              inflate(R.layout.fragment_lista_ocen,
                      container, false);
            mDane = new ArrayList<ModelOceny>();
            // musi być tu (nie w onAttach()) bo inaczej
            // aktywność nie jest dostępna
            InteraktywnyAdapterTablicy adapter =
              new InteraktywnyAdapterTablicy(
                getActivity(), mDane);
            mListaOcen =
              (ListView) widok.findViewById(R.id.listaOcen);
            mListaOcen.setAdapter(adapter);
            return widok;
      }

      @Override
```

```
public void onActivityCreated(Bundle savedInstanceState) {
	super.onActivityCreated(savedInstanceState);
	// ustawienie obsługi kliknięcia przyciskWroc –
	// analogicznie jak we FragmentDane
}

public void ustawLiczbeOcen(int liczbaOcen) {
	int staraLiczbaOcen = mLiczbaOcen;
	mLiczbaOcen = liczbaOcen;
	if (mLiczbaOcen == -1)
		zablokujElementy();
	else
		odblokujElementy();
	if (staraLiczbaOcen != mLiczbaOcen)
		kliknieciePrzyciskuWyczysc();
}

//... - onAttach() i onDetach() pozostają niezmodyfikowane

public void kliknieciePrzyciskuWyczysc() {
	mDane.clear();
	for (int i = 0; i < mLiczbaOcen; i++)
		mDane.add(new ModelOceny(
		  "ocena " + (i + 1)));
	InteraktywnyAdapterTablicy adapter =
	  (InteraktywnyAdapterTablicy) mListaOcen.
	    getAdapter();
	adapter.notifyDataSetChanged();
}

public void kliknieciePrzyciskuPowrot() {
	Bundle tobolek = new Bundle();
	spakujOceny(tobolek);
	zablokujElementy();
	mListener.ocenyWypelnione(tobolek);
}

private void spakujOceny(Bundle tobolek) {
	tobolek.putInt(
	  DaneFragment.LICZBA_OCEN, mLiczbaOcen);
	for (int i = 0; i < mLiczbaOcen; i++)
	  tobolek.putInt(OCENA + i,
	    mDane.get(i).dajOcene());
}

public void zablokujElementy() {
```

```
            radioButtonyAktywne(false);
            Button przyciskWroc =
              (Button) getView().
                findViewById(R.id.przyciskWroc);
            przyciskWroc.setVisibility(View.GONE);
        }

        public void odblokujElementy() {
            radioButtonyAktywne(true);
            Button przyciskWroc =
              (Button) getView().
                findViewById(R.id.przyciskWroc);
            przyciskWroc.setVisibility(View.VISIBLE);
        }

        private void radioButtonyAktywne(boolean aktywne) {
          for (int i = 0; i < mListaOcen.getChildCount(); ++i) {
          mListaOcen.getChildAt(i).
            findViewById(R.id.grupaRadioOceny).
              findViewById(R.id.radioOcena2).setEnabled(aktywne);
          //... - pozostałe radio buttony - analogicznie
          }
        }
}
```

Tutaj również w metodzie `onCreateView()` umieszczamy kod tworzący hierarchię obiektów reprezentujących widoki. Dodatkowo tworzymy w niej adapter i łączymy go z listą. W metodzie `onActivityCreated()` ustawiamy obsługę zdarzeń.

Wśród metod pomocniczych nowością jest metoda `radioButtonyAktywne()`, która za pomocą parametru logicznego pozwala zdecydować, czy przyciski radiowe na liście ocen będą aktywne czy nie. Metoda ta będzie potrzebna do blokowania elementów GUI po obliczeniu wyników. Pozostałe metody to efekt reorganizacji kodu.

Ćwiczenie 2.17 (do samodzielnego wykonania)

Korzystając ze zdobytej wiedzy i przykładu z punktu 2.15.1 stwórz kompletny kod dwóch opisanych fragmentów.

2.15.2. Tworzenie aktywności korzystającej z fragmentów

Teraz gdy dysponujemy już obydwoma fragmentami, możemy przejść do tworzenia nowej głównej aktywności naszej aplikacji. Nazwiemy ją tak samo jak w poprzedniej wersji aplikacji tzn. `DaneActivity`. W kreatorze wybieramy

tworzenie *pustej aktywności* (*ang. blank activity*), nowy układ elementów nazwiemy `activity_glowna.xml`.

Listing 2.36. Główna aktywność aplikacji

```
public class DaneActivity extends Activity implements
        DaneFragment.InterakcjaFragmentuDanychListener,

    ListaOcenFragment.InterakcjaFragmentuListyOcenListener {

    private DaneFragment dajDaneFragment() {
        return
          (DaneFragment) getFragmentManager().
            findFragmentById(R.id.daneFragment);
    }

    private ListaOcenFragment dajListaOcenFragment() {
        ListaOcenFragment listaOcenFragment =
          (ListaOcenFragment) getFragmentManager().
            findFragmentById(R.id.listaOcenFragment);
        if (listaOcenFragment != null)
          if (listaOcenFragment.isAdded())
            return listaOcenFragment;
        return null;
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_glowna);
    }

    @Override
    public void zmianaDanych(int liczbaOcen) {
        ListaOcenFragment
          listaOcenFragment = dajListaOcenFragment();
        if (listaOcenFragment != null) {
          if (dajDaneFragment().czyWszystkieDaneOk())
            listaOcenFragment.ustawLiczbeOcen(liczbaOcen);
          else
            listaOcenFragment.zablokujElementy();
        } else
            dajDaneFragment().pokazPrzyciskOceny();
    }
```

```
        @Override
        public void ocenyWypelnione(Bundle oceny) {
          dajDaneFragment().obliczSredniaOcenIPokazWynik(oceny);
        }

        // MUSI BYĆ PUBLICZNA I MUSI BYĆ W AKTYWNOŚCI (BO TO
        // KONTEKST)
        public void niePoszloPrzycisk(View v) {
               Toast.makeText(this, getString(R.string.warunek),
                 Toast.LENGTH_LONG).show();
               finish();
        }
}
```

Jak widać kod aktywności jest dużo bardziej zwięzły niż w poprzedniej wersji aplikacji i różni się od niej w istotny sposób.

Na początku znalazły się dwie metody pomocnicze `dajDaneFragment()` oraz `dajListaOcenFragment()`. Ich zadaniem jest zwracanie referencji do odpowiednich fragmentów. Prostsza z nich `dajDaneFragment()` za pomocą managera fragmentów odszukuje fragment na podstawie jego identyfikatora (identyfikator jest zdefiniowany w układzie XML aktywności – zajmiemy się nim za chwilę). W przypadku tego fragmentu nigdy nie powinien wystąpić problem z jego znalezieniem, ponieważ zawsze znajduje się on w aktywności głównej (niezależnie od ustawienia). Druga metoda `dajListaOcenFragment()` jest bardziej skomplikowana. Na początku podobnie jak w metodzie `dajDaneFragment()` odszukuje ona fragment z listą ocen na podstawie id. Ten fragment jednak nie zawsze jest częścią głównej aktywności – dzieje się tak, gdy uruchomimy aplikację w ustawieniu pionowym. Dlatego metoda sprawdza, czy uzyskana referencja jest różna on null. Drugim problemem są zmiany konfiguracji. Jeżeli np. aplikacja zostanie uruchomiona w ustawieniu poziomym do aktywności (zgodnie z naszymi założeniami), zostaną dodane dwa fragmenty. Gdy urządzenie zostanie obrócone do ustawienia pionowego, w aktywności zostanie jeden widoczny fragment `DaneFragment`. Jednak gdy spróbujemy uzyskać referencję do `ListaOcenFragment`, otrzymamy wartość różną on `null`, mimo że nie jest on widoczny. Reasumując, w głównej aktywności fragment `ListaOcenFragment` może istnieć bądź nie w zależności od historii aplikacji. Funkcja pomocnicza `dajListaOcenFragment()` rozwiązuje ten problem, i jeżeli fragment jest widoczny, zwraca referencję do niego. W przeciwnym wypadku zwraca `null`.

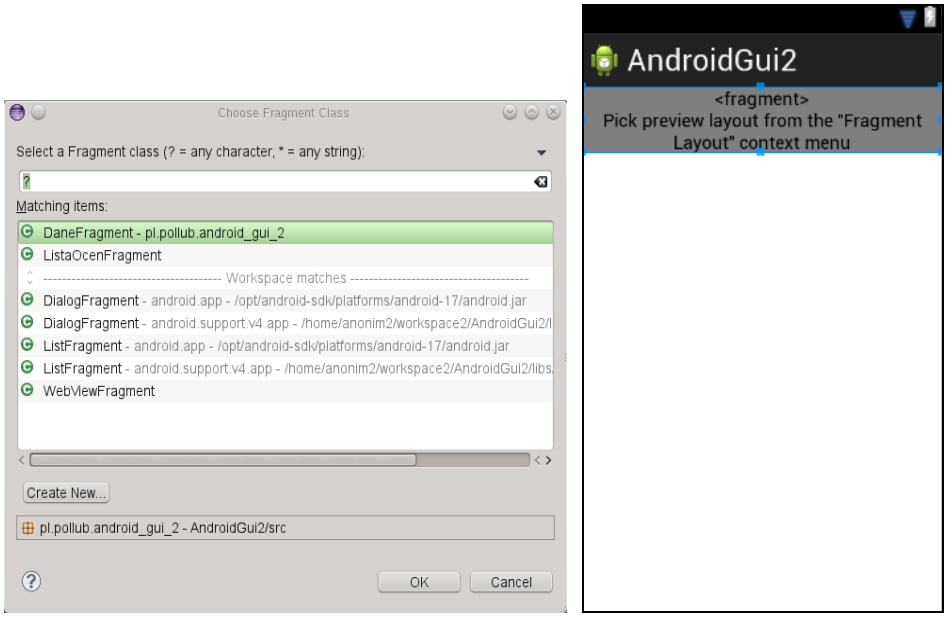
Kolejną istotną metodą jest `zmianaDanych()` – jest ona zdefiniowana w interfejsie `DaneFragment.InterakcjaFragmentuDanychListener`, który nasza aktywność implementuje. Dzięki tej metodzie fragment `DaneFragment` może powiadomić aktywność `DaneActivity` o zmianie imienia, nazwiska bądź

liczby ocen. Powstaje pytanie – po co aktywności ta informacja? Krótka odpowiedź mogłaby brzmieć: żeby przekazać ją dalej. Otóż aby zapewnić dużą elastyczność w powtórnym wykorzystaniu fragmentów, komunikację między nimi należy realizować za pośrednictwem aktywności. Fragment nie powinien przekazywać informacji bezpośrednio do innego fragmentu. Powinien wysłać ją do aktywności, a aktywność powinna „podjąć decyzję" co z nią zrobić. Taki właśnie sposób komunikacji zastosujemy. Aktywność za pośrednictwem metody `zmianaDanych()` dowiaduje się o zmianie, następnie sprawdza, czy istnieje i jest widoczny fragment z listą ocen (czyli czy ustawienie jest poziome). Jeżeli tak jest, to sprawdza poprawność danych i w zależności od wyniku umożliwia, bądź blokuje możliwość wprowadzania ocen. Jeżeli fragment z listą nie jest widoczny (czyli ustawienie jest pionowe), to w zależności od poprawności danych zostanie wyświetlony przycisk umożliwiający przejście do drugiej aktywności służącej do wprowadzania ocen (utworzymy ją za chwilę).

Aktywność implementuje również drugi interfejs – `ListaOcenFragment.InterakcjaFragmentuListyOcenListener`. Należy do niego metoda `ocenyWypelnione()`, dzięki której aktywność może zareagować na zakończenie wprowadzania ocen przez użytkownika.

Ostatnie zagadnienie związane z kodem aktywności dotyczy metody obsługującej przycisk „nie poszło mi". Jak widać, ta metoda nie została umieszczona we fragmencie. Jest tak, ponieważ obsługę tego przycisku zdefiniowaliśmy w układzie XML poprzez ustawienie atrybutu `android:onClick="niePoszloPrzycisk"`. Atrybut ten powoduje, że w momencie kliknięcia przycisku metoda o zadanej nazwie jest wyszukiwana w bieżącym kontekście, a jak pamiętamy kontekstem jest aktywność, a nie fragment. Gdybyśmy umieścili tą metodę we fragmencie, nie zostałaby znaleziona.

Skoro dysponujemy już kodem nowej aktywności, zajmiemy się teraz jej układem zapisanym w pliku XML. Zaczniemy od zmiany układu względnego (*RelativeLayout*) na najprostszy liniowy (*LinearLayout*) o ustawieniu pionowym. Następnie z palety układów (*Layouts*) wybierzemy Fragment i umieścimy w układzie aktywności. W momencie dodawania fragmentu pojawi się okno przedstawione z lewej strony rysunku 2.19. Możemy w nim wybrać fragment, który chcemy dodać do naszej aktywności. Identyfikator (`id`) fragmentu ustawimy na `daneFragment`.



Rys. 2.19. Dodawanie fragmentu do układu

Po dodaniu fragmentu w edytorze graficznym widoczny będzie układ przedstawiony po prawej stronie rysunku 2.19. Zgodnie z informacją zamieszczoną w szarej ramce wybieramy z menu kontekstowego fragmentu: *Fragment Layout | fragment_dane_rel*. W edytorze powinien się pojawić podgląd układu fragmentu. Listing 2.37 przestawia kompletny plik XML opisujący układ nowej aktywności.

Listing 2.37. Plik `activity_glowna.xml` opisujący układ `DaneActivity`

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

    <fragment
        android:id="@+id/daneFragment"
        android:name="pl.pollub.android_gui_2.DaneFragment"
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        tools:layout="@layout/fragment_dane_rel" />
</LinearLayout>
```

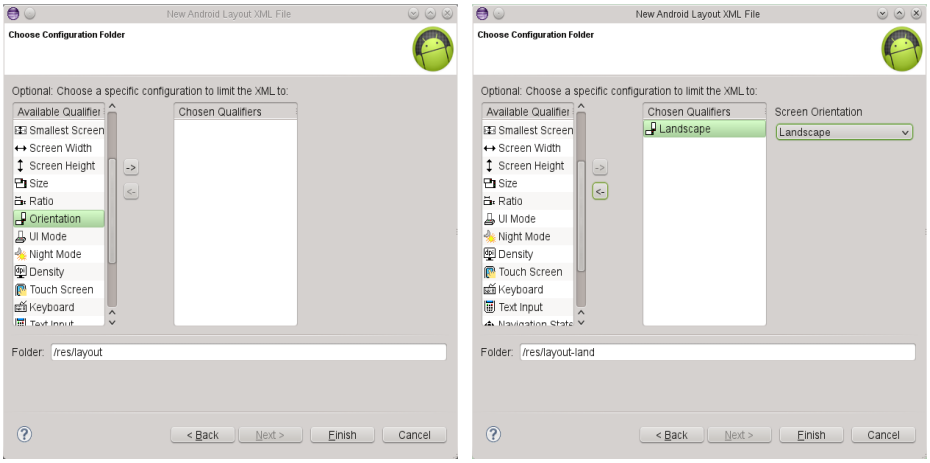
Na listingu warto zwrócić uwagę na dwa atrybuty: `android:name = ”pl.pollub.android_gui_2.DaneFragment”` oraz `tools:layout = ”@layout/fragment_dane_rel”`. Pierwszy z nich wskazuje klasę fragmentu, który jest osadzony w aktywności, a drugi układ, który należy wyświetlać, w podglądzie interfejsu graficznego tej aktywności.

Ćwiczenie 2.18 (do samodzielnego wykonania)

Utwórz 2-gi układ o nazwie `activity_glowna.xml` dla aktywności `DaneActivity` (nazwa celowo jest identyczna). Umieść go w katalogu `/res/layout-land`. Wykorzystaj układ liniowy o ustawieniu poziomym. Dodaj do niego oba utworzone fragmenty. Powinny one znajdować się obok siebie. Zastosuj następujące identyfikatory fragmentów: `daneFragment` oraz `listaOcenFragment`.

Wskazówki:

- nie twórz katalogu `res/layout-land` ręcznie, lecz w trakcie pracy kreatora układu XML wybierz odpowiedni kwalifikator, tak jak to pokazano na rysunku 2.20. Zauważ, że kwalifikatorów, które można stosować jest dużo więcej,



Rys. 2.20. Wybór kwalifikatora

- aby oba fragmenty zajmowały taką samą ilość miejsca w układzie, wykorzystaj atrybut `layout_weight`

```
android:layout_weight="1"
```

2.15.3 Tworzenie aktywności dla określonej konfiguracji

W naszej aplikacji mamy już jedną aktywność. Gdy urządzenie jest ustawione pionowo, wykorzystywany jest domyślny plik `res/layout/activity_glowna.xml`. Układ opisany przez ten plik zawiera tylko fragment pozwalający na wpisanie imienia, nazwiska oraz liczby ocen. Gdy urządzenie jest ustawione poziomo, Android przełączy się automatycznie na plik `res/layout-land/activity_glowna.xml`, w którym osadzone są dwa fragmenty. Katalog `res/layout-land` zawiera pliki specyficzne dla ustawienia poziomego. Jeżeli urządzenie jest ustawione poziomo, a jakiegoś pliku nie ma w tym katalogu, wykorzystywane są pliki z domyślnego `res/layout`. Podobna zasada odnosi się do wszystkich innych katalogów zasobów – wiele z nich zawiera różne kwalifikatory, często istnieje też katalog bez kwalifikatorów z plikami domyślnymi.

W nowej wersji naszej aplikacji do obliczania średniej brakuje jeszcze możliwości wprowadzania ocen w przypadku, gdy urządzenie ustawione jest pionowo. Musimy zatem stworzyć nową wersję aktywności `ListaOcenActivity`. Aktywność ta powinna jednak działać wyłącznie wtedy, gdy urządzenie ustawione jest pionowo. Jeżeli użytkownik, w trakcie wprowadzania ocen, obróci urządzenie do ustawienia poziomego, aktywność powinna się automatycznie zakończyć, a aplikacja powinna przejść do głównej aktywności zawierającej oba fragmenty.

Listing 2.38. Aktywność `ListaOcenActivity` działająca tylko w ustawieniu pionowym

```
public class ListaOcenActivity extends Activity implements

      ListaOcenFragment.InterakcjaFragmentuListyOcenListener {

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);

            if (getResources().getConfiguration().
              orientation ==
              Configuration.ORIENTATION_LANDSCAPE)
            {
                  finish();
```

```
                // żeby onCreate skończyło się natychmiast
                return;
            }

            setContentView(R.layout.activity_lista_ocen);

            Bundle dodatkoweInfo = getIntent().getExtras();
            int liczbaOcen = dodatkoweInfo.
              getInt(DaneFragment.LICZBA_OCEN);
            ListaOcenFragment listaOcenFragment =
              (ListaOcenFragment) getFragmentManager().
                findFragmentById(R.id.listaOcenFragment);
            listaOcenFragment.ustawLiczbeOcen(liczbaOcen);
        }

        @Override
        public void ocenyWypelnione(Bundle oceny) {
            Intent intencja = new Intent();
            intencja.putExtras(oceny);
            setResult(RESULT_OK, intencja);
            finish();
        }
}
```

Listing 2.38. przedstawia kompletny kod aktywności `ListaOcenActivity`. Jak widać jest dość krótki. Najważniejszym elementem jest sprawdzanie konfiguracji na początku metody `onCreate()`. Z konfiguracji odczytywane jest ustawienie urządzenia. Jeżeli jest ono poziome, aktywność jest kończona i następuje powrót do aktywności głównej – `DaneActivity`. Warto zauważyć, że po wywołaniu metody `finish()` znajduje się słowo kluczowe `return`. Bez niego aktywność zostałaby zamknięta dopiero po wykonaniu `onCreate()` do końca.

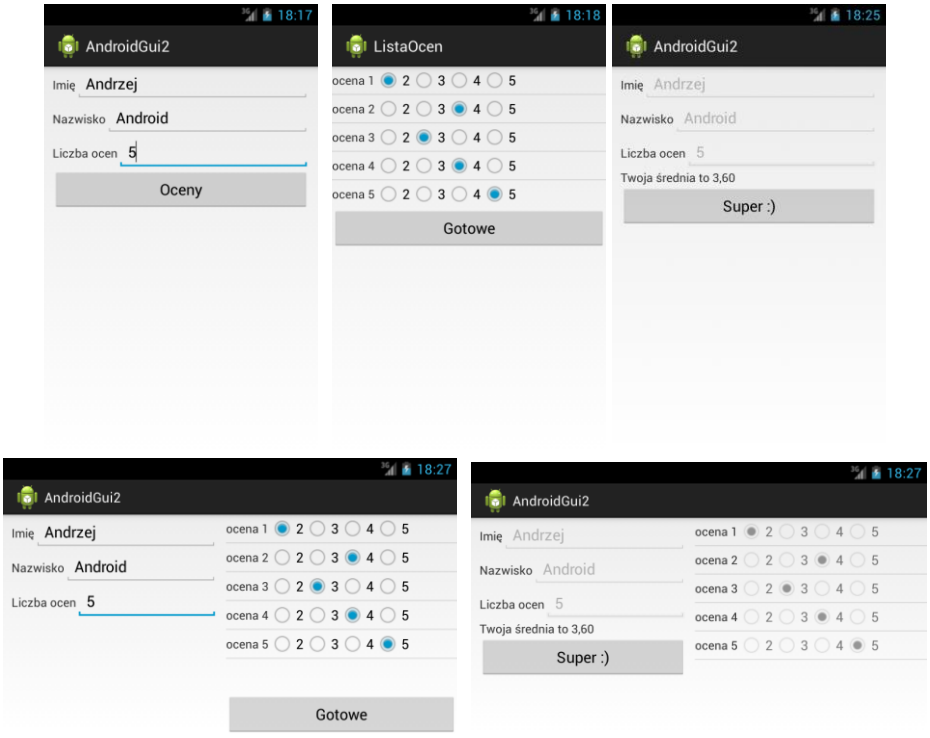
Ten sam kod zapewni nam obsługę sytuacji, w której użytkownik podczas edycji listy ocen w ustawieniu pionowym obróci urządzenie. Stanie się to dzięki temu, że Android przy zmianie konfiguracji niszczy aktywność, a następnie tworzy ją od nowa. Aktywność w trakcie ponownego uruchamiania stwierdzi, że urządzenie jest ustawione poziomo i natychmiast się zakończy, wracając do aktywności głównej.

Innym istotnym elementem jest metoda `ocenyWypelnione()`. Jest ona implementacją interfejsu:
`ListaOcenFragment.InterakcjaFragmentuListyOcenListner`

Służy do przekazywania z fragmentu do aktywności informacji o tym, że użytkownik zakończył wprowadzanie ocen. Warto zauważyć, że reakcja aktywności z listą ocen jest inna, niż reakcja aktywności z danymi na to samo zdarzenie. Jest to dobry przykład uzasadniający, dlaczego warto stosować podejście z wykorzystaniem interfejsów do komunikacji fragment – aktywność.

Ćwiczenie 2.19 (do samodzielnego wykonania)

Utwórz układ o nazwie `activity_lista_ocen.xml` i umieść go w katalogu `res/layout`. Wykorzystaj układ liniowy, pionowy z jednym fragmentem. Layout jest przeznaczony dla aktywności `ListaOcenActivity`, która – jak wyjaśniono powyżej – będzie wykorzystywana tylko w pionie. Jako dodawany fragment wybierz `ListaOcenFragment`. Jako `id` ustaw `listaOcenFragment`. Sprawdź działanie nowej wersji aplikacji wykorzystującej fragmenty. Wygląd aplikacji w różnych sytuacjach z różnymi ustawieniami ekranu przedstawiono na rysunku 2.21.



Rys. 2.21. Efekt wykonania aplikacji wykorzystującej fragmenty. W kolejności: wprowadzone poprawne dane (imię, nazwisko, liczba ocen), edycja ocen, obliczone wyniki, wprowadzanie wszystkich danych, obliczone wyniki

2.15.4 Pasek akcji (Action Bar) i fragmenty

Aby dodać do nowej wersji pasek akcji, będziemy musieli wprowadzić szereg modyfikacji w poszczególnych plikach. Rozpoczniemy jednak od upewnienia się, czy w folderach `res/drawable*` dostępne są stworzone w starej wersji ikonki a w katalogu `res/menu` pliki `dane.xml` i `lista_ocen.xml`. Jeżeli nie, to tworzymy ikonki i pliki XML opisujące menu tak samo jak w przypadku wersji bez fragmentów. Przejdziemy teraz do omawiania modyfikacji poszczególnych plików. Modyfikacje przedstawiono na listingach 2.39 – 2.42.

Listing 2.39. Modyfikacje dodające pasek akcji do fragmentu `DaneFragment`

```
public class DaneFragment extends Fragment {
       //... - część kodu usunięto

       public interface InterakcjaFragmentuDanychListener {
              //... -  część kodu usunięto
              public void menuZamknij();
              public void menuOk();
              public void menuWyczysc();
       }

       @Override
       public View onCreateView(LayoutInflater inflater,
                                ViewGroup container,
                                Bundle savedInstanceState) {
              //... -  część kodu usunięto
              setHasOptionsMenu(true);
              //... -  część kodu usunięto
       }

       @Override
       public void onCreateOptionsMenu(Menu menu,
                                       MenuInflater inflater) {
              inflater.inflate(R.menu.dane, menu);
              super.onCreateOptionsMenu(menu, inflater);
       }

       @Override
       public void onPrepareOptionsMenu(Menu menu) {
              MenuItem opcja =
                (MenuItem) menu.findItem(R.id.akcja_oceny);
              opcja.setVisible(czyWszystkieDaneOk() && !mWynik
                && getResources().getConfiguration().
                orientation ==
```

```
            Configuration.ORIENTATION_PORTRAIT);
        opcja =
          (MenuItem) menu.findItem(R.id.akcja_wyczysc);
        opcja.setVisible(!mWynik);
        super.onPrepareOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
        case R.id.akcja_oceny:
              mListener.menuOk();
              break;
        case R.id.akcja_wyczysc:
              mListener.menuWyczysc();
              break;
        case R.id.akcja_zamknij:
              mListener.menuZamknij();
              break;
        }
        return super.onOptionsItemSelected(item);
    }

    public void pokazPrzyciskOceny() {
        //... -  część kodu usunięto
        getActivity().invalidateOptionsMenu();
    }

    public void kliknieciePrzyciskuWyczysc() {
        mLiczbaOcenOk = false;
        mImieOk = false;
        mNazwiskoOk = false;
        mLiczbaOcen = -1;
        mImie="";
        mNazwisko="";
        mWynik=false;

        mImieEdycja.getEditableText().clear();
        mNazwiskoEdycja.getEditableText().clear();
        mLiczbaOcenEdycja.getEditableText().clear();
    }

    public void kliknieciePrzyciskuZamknij() {
        getActivity().finish();
    }
```

```
        public void blokujPolaPrzyciskiIOpcje() {
                //... -  część kodu usunięto
                getActivity().invalidateOptionsMenu();
        }
}
```

Modyfikacje rozpoczniemy od pliku `DaneFragment.java`. Zmiany do wprowadzenia znajdują się na listingu 2.39. Pierwszą z nich jest modyfikacja interfejsu zdefiniowanego wewnątrz. Będzie on teraz służył również do przekazywania informacji na temat zdarzeń związanych z paskiem akcji.

Kolejną modyfikacją jest dodanie wywołania metody `setHasOptionsMenu()` do `onCreateView()`. Domyślnie pasek akcji we fragmentach jest wyłączony i jeżeli chcemy, aby fragment dodawał swoje akcje do paska, musimy zadbać o uaktywnienie tej opcji.

Dalej dodajemy kolejne metody odpowiedzialne za dodanie akcji do paska oraz za obsługę poszczególnych opcji. Najważniejszą różnicą w stosunku do wersji aplikacji bez fragmentów jest to, że komunikacja odbywa się za pośrednictwem aktywności. Oznacza to, że w momencie gdy użytkownik wybierze akcję z paska, wywołana zostaje metoda `onOptionsItemSeleted()`. Ona z kolei wywoła za pośrednictwem referencji `mListener` metodę z aktywności. Dopiero metoda z aktywności, w zależności od kontekstu, uruchomi odpowiednie metody w jednym bądź dwóch fragmentach.

Listing 2.40. Modyfikacje dodające pasek akcji do fragmentu `ListaOcenFragment`

```
public class ListaOcenFragment extends Fragment {

        //... -  część kodu usunięto

        public interface InterakcjaFragmentuListyOcenListener {
                //... -  część kodu usunięto
                public void menuZamknij();
                public void menuOk();
                public void menuWyczysc();
        }

        //... -  część kodu usunięto

        @Override
        public void onCreateOptionsMenu(Menu menu,
                                        MenuInflater inflater) {
                inflater.inflate(R.menu.lista_ocen, menu);
                super.onCreateOptionsMenu(menu, inflater);
```

```
        }

        @Override
        public boolean onOptionsItemSelected(MenuItem item) {
                switch (item.getItemId()) {
                case R.id.akcja_powrot:
                        mListener.menuOk();
                        break;
                case R.id.akcja_wyczysc:
                        mListener.menuWyczysc();
                        break;
                case R.id.akcja_zamknij:
                        mListener.menuZamknij();
                        break;
                }
                return super.onOptionsItemSelected(item);
        }
}
```

Modyfikacje w pliku `ListaOcenFragment.java` (listing 2.40) są bardzo podobne do tych z `DaneActivity.java`. Tutaj również stosujemy komunikację przez aktywność. Warto zauważyć, że w `onCreateView()` nie pojawia się wywołanie metody `setHasOptionsMenu()`. Zostanie to wyjaśnione za chwilę.

Listing 2.41. Modyfikacje zapewniające obsługę paska akcji do aktywności `DaneActivity`

```
public class DaneActivity extends Activity implements
        DaneFragment.InterakcjaFragmentuDanychListener,
        ListaOcenFragment.InterakcjaFragmentuListyOcenListener {
        //... - część kodu usunięto
        @Override
        public void menuZamknij() {
                finish();
        }
        @Override
        public void menuOk() {
                ListaOcenFragment listaOcenFragment =
                  dajListaOcenFragment();
                if (listaOcenFragment != null)
                  listaOcenFragment.klikniecePrzyciskuPowrot();
                else
                  dajDaneFragment().klikniecePrzyciskuOceny();
        }
        @Override
```

```
        public void menuWyczysc() {
                dajDaneFragment().kliknieciePrzyciskuWyczysc();
                ListaOcenFragment listaOcenFragment =
                  dajListaOcenFragment();
                if (listaOcenFragment != null)
                  //bo to nie przycisk fragmentu listy ocen lecz
                  //fragmentu danych liczba ocen jest kasowana więc
                ..//trzeba usunąć też liczbę ocen
                  listaOcenFragment.ustawLiczbeOcen(-1);
        }
}
```

Modyfikacje w pliku `DaneActivity.java` (listing 2.41) sprowadzają się do zaimplementowania nowych metod interfejsów

`DaneFragment.InterakcjaFragmentuDanychListener`

oraz

`ListaOcenFragment.InterakcjaFragmentuListyOcenListener`.

Warto zauważyć, że metody `menuOk()` oraz `menuWyczysc()` wykonują różne czynności w zależności od tego, które fragmenty są widoczne. Na przykład `menuOk()` w ustawieniu poziomym (widoczne dwa fragmenty) powoduje obliczenie średniej, natomiast w ustawieniu pionowym (widoczny jeden fragment) powoduje przejście do aktywności z listą ocen. Jest to kolejny przykład, dlaczego przekazywanie danych między fragmentami warto realizować za pośrednictwem aktywności.

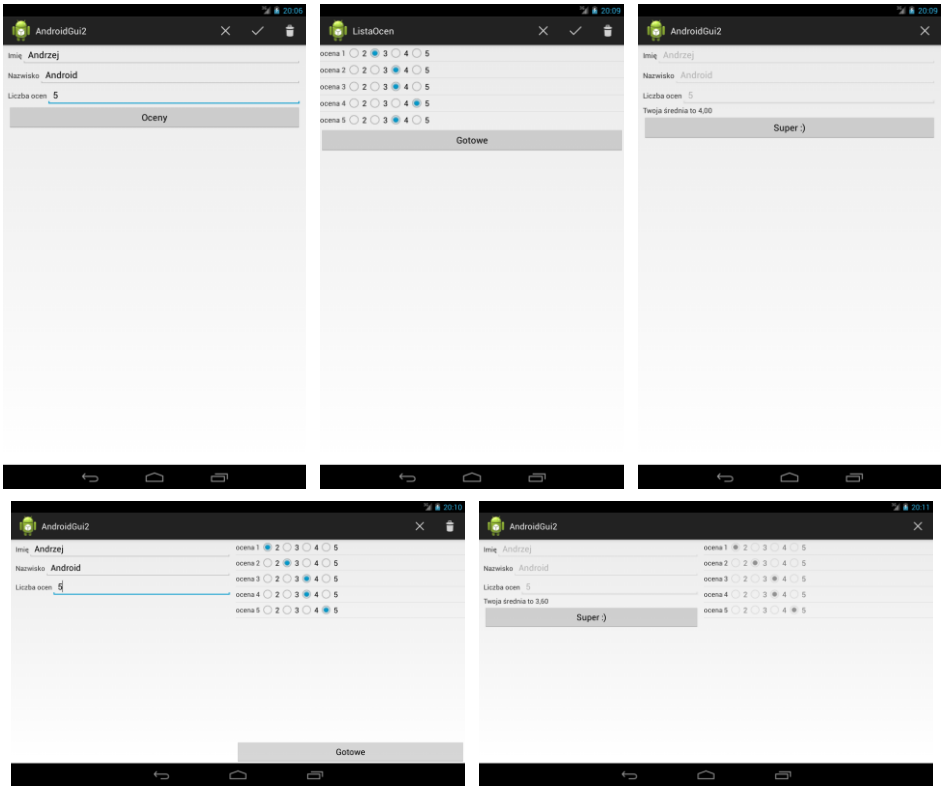
Listing 2.42. Modyfikacje zapewniające obsługę paska akcji do aktywności `ListaOcenActivity`

```
public class ListaOcenActivity extends Activity implements
        ListaOcenFragment.InterakcjaFragmentuListyOcenListener {

          @Override
          protected void onCreate(Bundle savedInstanceState) {
               //... - część kodu usunięto
               listaOcenFragment.setHasOptionsMenu(true);
          }
          //... - część kodu usunięto
          @Override
          public void menuZamknij() {
```

```
            finish();
        }
        @Override
        public void menuOk() {
            ListaOcenFragment listaOcenFragment =
                (ListaOcenFragment) getFragmentManager().
                  findFragmentById(R.id.listaOcenFragment);
            listaOcenFragment.kliknieciePrzyciskuPowrot();
        }
        @Override
        public void menuWyczysc() {
            ListaOcenFragment listaOcenFragment =
                (ListaOcenFragment) getFragmentManager().
                  findFragmentById(R.id.listaOcenFragment);
          listaOcenFragment.kliknieciePrzyciskuWyczysc();
        }
}
```

Ostatnimi modyfikacjami są te przedstawione na listingu 2.42. Również one polegają główne na zaimplementowaniu nowych metod interfejsu służącego do komunikacji z fragmentem. Jednak istnieje jeszcze jedna różnica – pasek akcji fragmentu `ListaOcenFragment` jest włączany tylko przez aktywność `ListaOcenActivity`. Tak więc w ustawieniu pionowym każdy z fragmentów dodaje swoje akcje do paska odpowiadającej mu aktywności (`DaneFragment` do `DaneActivity`, a `ListaOcenFragment` do `ListaOcenActivity`). W ustawieniu poziomym tylko `DaneFragment` dodaje akcje do paska `DaneActivity` i to aktywność dba o to, aby w razie potrzeby uruchamiać odpowiednie metody fragmentu z listą ocen. `ListaOcenFragment` nie dodaje swoich akcji.

Poniżej, na rysunku 2.22 przedstawiono efekty działania aplikacji po dodaniu paska akcji. Warto zaznaczyć, że na niektórych urządzeniach (konfiguracjach emulatora) wybrane opcje mogą być dostępne pod sprzętowym przyciskiem menu lub ekranowym przyciskiem menu opcji. Takie rozwiązanie stosuje się w wypadku, gdy na pasku nie ma dostatecznie dużo miejsca.



Rys 2.22. Aplikacja wykorzystująca fragmenty i pasek akcji. Widoczność opcji zależy od konfiguracji i etapu, na którym znajduje się aplikacja

2.15.5. Fragmenty na starszych urządzeniach

Jak wspomniano wcześniej fragmenty są elementem, który został wprowadzony w Androidzie 3.0. Odpowiada on poziomowi API równemu 11. W kolejnych wersjach systemu wprowadzane były również inne nowości. Niechęć niektórych producentów w publikowaniu aktualizacji do nowszych wersji Androida dla swoich urządzeń (czasami usprawiedliwiona brakiem możliwości technicznych) doprowadziła do zjawiska nazywanego *fragmentacją*. Polega ono na tym, że w użyciu znajduje się wiele urządzeń z różnymi wersjami systemu. Prowadziło to do sytuacji, w której programiści stawali przed wyborem – napisać aplikację o szerokich możliwościach korzystającą z dobrodziejstw najnowszego API Androida, czy napisać aplikację działającą na możliwie dużej gamie urządzeń.

Na szczęście programiści Google zauważyli ten problem i stworzyli tzw. *bibliotekę wsparcia* (*ang. support library*). Nie jest to całkowite

rozwiązanie problemu różnic między różnymi wersjami Androida, jednak umożliwia korzystanie z wielu nowości na starszych urządzeniach. Biblioteka wsparcia ma wiele wersji dla różnych poziomów API. Warto zapoznać się ze stroną [15], na której opisano sposób instalacji i konfiguracji biblioteki wsparcia oraz ze stroną [14], na której opisano możliwości biblioteki.

Do tej pory naszą aplikację tworzyliśmy wykorzystując 17. poziom API systemu Android. Z biblioteką wsparcia zapoznamy się, czyniąc naszą aplikację zdolną do pracy na Androidzie 1.6 – czyli 4. poziomem API. Oczywiście przeróbkom poddamy wersję korzystającą z fragmentów.

Zanim przejdziemy do modyfikacji wykonamy kopię projektu i zmienimy jej nazwę na `AndroidGui2Compat`. Do zapewnienia kompatybilności z 4. poziomem API będzie potrzebna nam biblioteka `android-support-v4.jar`. Powinna zostać dodana automatycznie podczas tworzenia projektu i znajdować się w katalogu `libs` w naszym projekcie. Gdyby z jakichś przyczyn tak się nie stało, to należy:

- uruchomić program *SDK Manager* i sprawdzić czy zainstalowana jest *Android Support Library* z grupy pakietów o nazwie *Extras*
- utworzyć katalog `/libs` w głównym katalogu projektu
- skopiować plik `android-support-v4.jar` z katalogu `android-sdk/extras/android/compatibility/v4`

Gdy biblioteka znajduje się już w projekcie, należy odszukać ją w drzewie projektu i kliknąć plik `libs/android-support-v4.jar` prawym przyciskiem myszy, a następnie wybrać *Build Path* | *Add to Build Path*. W projekcie powinna pojawić się sekcja *Referenced Libraries* a w niej `android-support-v4.jar`

W celu uniknięcia konfliktów odpowiedniki klas z nowych wersji API zostały umieszczone w pakietach `android.support.*` zamiast w pakietach `android.*`. Często również nazwy klas zawierają `Compat` a nazwy metod `Support`. Nie jest to jednak reguła przestrzegana w 100%. Przykładem może być tu `FragmentActivity`, która jest odpowiednikiem klasy `Activity` z nowszych wersji API.

Widać zatem, że do zapewnienia kompatybilności ze starszą wersją API nie wystarczy dołączenie biblioteki wsparcia do projektu. Będą konieczne modyfikacje polegające na zamianie „zwykłych” klas i metod na ich odpowiedniki z biblioteki wsparcia. Szczęśliwie modyfikacje te nie są zbyt skomplikowane i można wykonać je bardzo szybko.

Należy:

- zamienić wszystkie wystąpienia klasy `Activity` na `FragmentActivity`. Należy również zmienić importowany pakiet z `android.app.Activity` na `android.support.v4.app.FragmentActivity`. Na przykład:

```
import android.support.v4.app.FragmentActivity;
```

```
public class DaneActivity extends FragmentActivity //...
```

- zamienić wszystkie wystąpienia importu pakietu `android.app.Fragment` na import pakietu `android.support.v4.app.Fragment`,

```
import android.support.v4.app.Fragment;
```

- zamienić wszystkie wywołania metody `getFragmentManager()` na `getSupportFragmentManager()`. Na przykład:

```
(DaneFragment) getSupportFragmentManager().
       findFragmentById(R.id.daneFragment);
```

- zamienić wszystkie wywołania metody `invalidateOptionsMenu()` na `supportInvalidateOptionsMenu()`,

```
getActivity().supportInvalidateOptionsMenu();
```

- zmodyfikować manifest, tak aby minimalna wersja SDK wynosiła 4

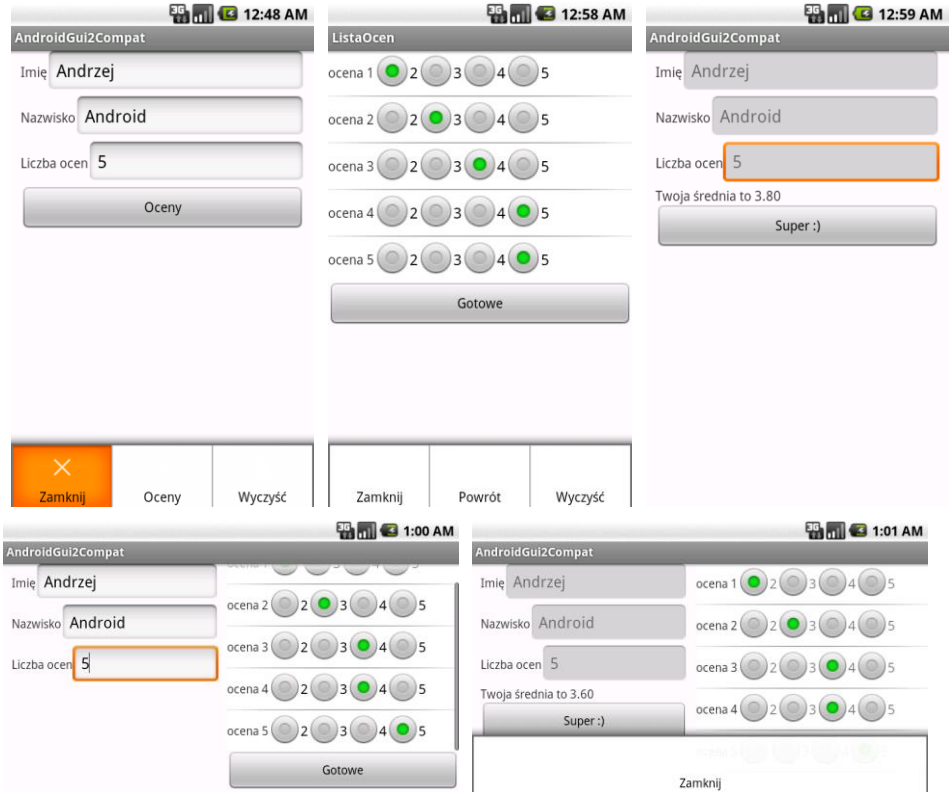
```
<uses-sdk
     android:minSdkVersion="4"
     android:targetSdkVersion="17" />
```

Na koniec trzeba dodać, że choć aplikacja będzie działać na starszych wersjach systemu, to mogą występować różnice w jej wyglądzie. Wersje starsze niż 3.0 nie posiadały paska akcji, więc akcje będą dostępne pod sprzętowym przyciskiem menu.

Co ciekawe, aplikacja po konwersji uruchomiona na urządzeniu z nowszą wersją systemu wygląda i działa tak samo, jak wersja nie korzystająca z biblioteki wsparcia (np. dostępny jest pasek akcji).

Ćwiczenie 2.20 (do samodzielnego wykonania)

Korzystając ze wskazówek z punktu 2.15.5 przeprowadź konwersję aplikacji korzystającej z fragmentów, tak aby działała nawet na systemie Android 1.6. Sprawdź działanie aplikacji nie tylko na starszym systemie, lecz także na wersji 4.2. Efekt działania aplikacji przedstawiono na rysunku 2.23.



Rys 2.23. Aplikacja wykorzystująca fragmenty pracująca na Androidzie 1.6

3. Trwałe przechowywanie danych

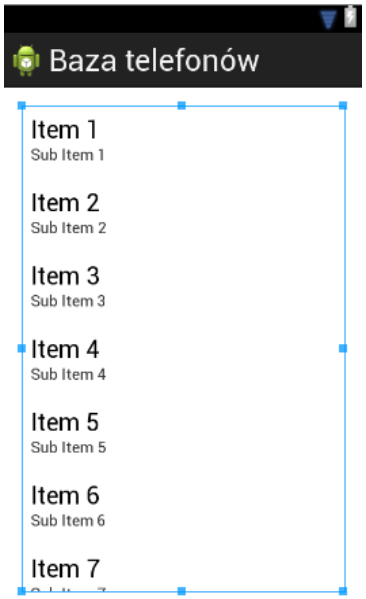
W tym rozdziale zajmiemy się ważnym zagadnieniem, jakim jest trwałe przechowywanie danych. Trwałość oznacza, że dane zostaną zachowane między uruchomieniami aplikacji. W poprzednich rozdziałach zapisywaliśmy tylko tymczasowy stan aplikacji, co pozwalało na jego przywrócenie po restarcie aktywności.

Różne sposoby przechowywania danych poznamy na przykładzie aplikacji pozwalającej na utworzenie prostej bazy telefonów – oczywiście pracujących pod kontrolą Androida. Zanim jednak przejdziemy do poznawania wbudowanej w androida bazy danych, czy zasad dostępu do plików, musimy stworzyć szkielet projektu, z którym będziemy pracować w tym rozdziale. Będzie to przy okazji przypomnienie materiału dotyczącego tworzenia GUI.

3.1 Szkielet przykładowego projektu

Rozpoczniemy od utworzenia projektu `AndroidDane`. Aplikacja będzie korzystała z najnowszego dostępnego API. Wygląd aplikacji ustawimy na *Holo Dark*, a pakiet, w którym będziemy umieszczać klasy, nazwiemy: `pl.pollub.android_dane`. Tworząc projekt, przy okazji, utworzymy również pierwszą aktywność. Nazwiemy ją: `ListaTelActivity` natomiast jej układ: `activity_lista_tel.xml`

Przejdziemy teraz do modyfikacji układu głównej aktywności. Zmienimy układ z domyślnego układu względnego (*ang.* `RelativeLayout`) na liniowy (*ang.* `LinearLayout`) o ustawieniu pionowym. Następnie z palety widoków (komponentów) złożonych (*ang. composite*) przeciągamy widok listy (`ListView`). Ustawiamy jego identyfikator na `@android:id/list`. Pozostawimy również w układzie domyślnie dodaną etykietę (element `TextView`), jednak jej identyfikator zmienimy na `@android:id/empty`. Warto zwrócić uwagę, że tym razem identyfikatory ustawiamy nieco inaczej. Nie dodajemy własnych za pomocą `@+id/...`, lecz korzystamy z `@android:id/...`, czyli zestawu identyfikatorów zdefiniowanych w systemie. Dzięki temu klasa `ListActivity`, którą zastosujemy zamiast zwykłej aktywności, będzie mogła odszukać te elementy. Na koniec zmienimy tekst etykiety na `@string/brak_telefonow` (oczywiście w pliku `strings.xml` musimy dodać odpowiedni napis z informacją o braku telefonów w bazie). Efekt, do którego dążymy przedstawiono na rysunku 3.1.



Rys. 3.1. Układ aktywności `ListaTelActivity`

Od razu utworzymy też drugą aktywność, która będzie służyła do dodawania oraz edycji telefonów. Rozpoczniemy od utworzenia drugiej pustej aktywności o nazwie `EdycjaTelActivity` i układzie `activity_edycja_tel.xml`. Rozpoczniemy od zmiany głównego układu ze względnego na tabelaryczny (`TableLayout`). Ponieważ jest to nowy typ układu, poświęcimy mu trochę więcej uwagi.

3.1.1. Układ tabelaryczny

Jak wskazuje nazwa, układ ten pozwala łatwo uporządkować elementy w sposób przypominający tabelę. Sam układ `TableLayout` oraz wiersz tabeli `TableRow` można znaleźć w edytorze graficznym w części `Layouts`. Definiowanie tego układu przypomina tworzenie tabel w HTMLu. Elementy znajdują się w wierszach. Standardowo liczba kolumn dostosowana jest do liczby komponentów. Jeden komponent może zajmować więcej niż jedną kolumnę. Rozmiar układu tabelarycznego określany jest przez element nadrzędny (cały ekran czy innego zarządcę). Elementy potomne układu powinny mieć szerokość/wysokość ustawioną jako „`wrap_content`”.

Listing 3.1. Fragment układu XML aktywności `EdycjaTelActivity`

```
<TableLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/TableLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    android:shrinkColumns="*"
    android:stretchColumns="2"
    tools:context=".EdycjaTelActivity" >

    <TableRow
        android:id="@+id/tableRow1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        <TextView
            android:id="@+id/textView1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/producent_napis" />

        <EditText
            android:id="@+id/producentEdycja"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_span="2" >

            <requestFocus />
        </EditText>
    </TableRow>

    <TableRow
        android:id="@+id/tableRow2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" >

        <TextView
            android:id="@+id/textView2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
```

```
            android:text="@string/model_napis" />

        <EditText
            android:id="@+id/modelEdycja"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_span="2" />
    </TableRow>

</TableLayout>
```

Listing 3.1 przedstawia fragment układu aktywności `EdycjaTelActivity`. Warto w nim zwrócić uwagę na następujące atrybuty:

- Atrybut `android:shrinkColumns="*"` określa, które kolumny mogą zostać zwężone, aby tabela zmieściła się w obszarze zadanym przez element nadrzędny. „*” oznacza wszystkie kolumny, można też wymienić numery wybranych kolumn po przecinku. Kolumny numerowane są od 0.
- Atrybut `android:stretchColumns="2"` określa, które kolumny mogą zostać rozciągnięte, aby wypełnić wolne miejsce. Wartością może być „*” lub lista numerów wymienionych po przecinku.
- Atrybut `android:layout_span="2"` określa ile kolumn ma zajmować dany element (np. pole tekstowe). Domyślnie każdy komponent zajmuje tylko jedną kolumnę.

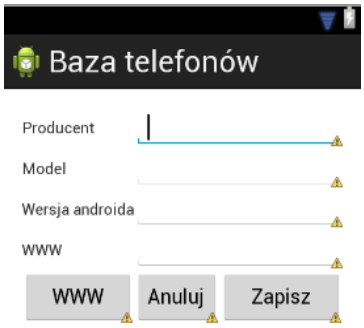
Ćwiczenie 3.1 (do samodzielnego wykonania)

Korzystając z informacji z punktu 3.1.1 stwórz kompletny układ dla aktywności `EdycjaTelActivity`. W kolejnych wierszach powinny znajdować się:

- etykieta `TextView` z napisem o identyfikatorze producent_napis (napis umieszczony w pliku `strings.xml`), pole tekstowe `EditText` o identyfikatorze `producentEdycja`,
- etykieta z napisem o identyfikatorze `model_napis`, pole tekstowe o identyfikatorze `modelEdycja`,
- etykieta z napisem o identyfikatorze `android_napis`, pole tekstowe o identyfikatorze `androidEdycja`,
- etykieta z napisem o identyfikatorze `www_napis`, pole tekstowe o identyfikatorze `wwwEdycja`,
- trzy przyciski o identyfikatorze napisu i identyfikatorze samego przycisku odpowiednio: `www_przycisk` / `wwwPrzycisk`, `anuluj_przycisk` / `anulujPrzycisk`, `zapisz_przycisk` / `zapiszPrzycisk`.

Pola tekstowe powinny zajmować dwie ostatnie kolumny i wypełniać całą wolną przestrzeń. Efekt wykonania ćwiczenia przedstawiono na rysunku 3.2.
Wskazówki

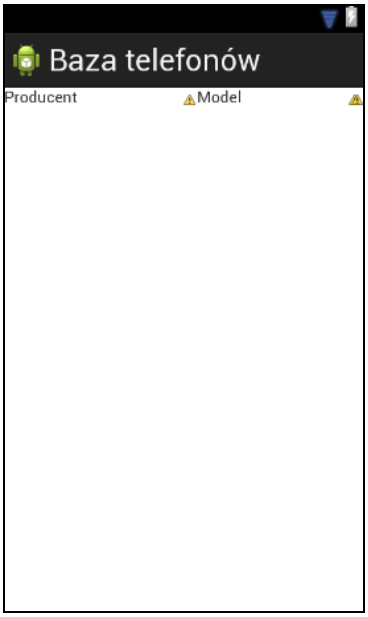
- z pól tekstowych należy usunąć domyślnie wstawiany przez edytor graficzny atrybut `android:ems="10"`, żeby rozmiar pola był dopasowany do rozmiarów układu.
- niektóre atrybuty takie jak `android:layout_span` najłatwiej dodać bezpośrednio w pliku XML



Rys. 3.2. Układ elementów aktywności `EdycjaTelActivity`

Ćwiczenie 3.2 (do samodzielnego wykonania)

Stwórz układ pojedynczego wiersza listy. Plik z układem nazwij `wiersz_listy.xml`. Jako główny element ustaw TableLayout. W układzie umieść jeden wiersz o identyfikatorze `wierszTabeli`. Do wiersza dodaj dwie etykiety tekstowe o identyfikatorach: `producentText` i `modelText`. Włącz rozciąganie i zmniejszanie wszystkich kolumn – tak aby wiersz wypełniał całą szerokość ekranu. Wynik wykonania ćwiczenia przedstawiono na rysunku 3.3.



Rys. 3.3. Układ wiersza listy telefonów

3.2. Korzystanie z bazy danych SQLite

Android pozwala każdej aplikacji na tworzenie baz *SQLite*. ***Jak łatwo się domyślić na podstawie nazwy, baza wykorzystuje język SQL.*** Nie ma dodatkowych ograniczeń, niż te wprowadzane przez samą bazę *SQLite* (www.sqlite.org). Baza jest dostępna z poziomu wszystkich klas danej aplikacji, ale tylko z poziomu aplikacji, która ją stworzyła.

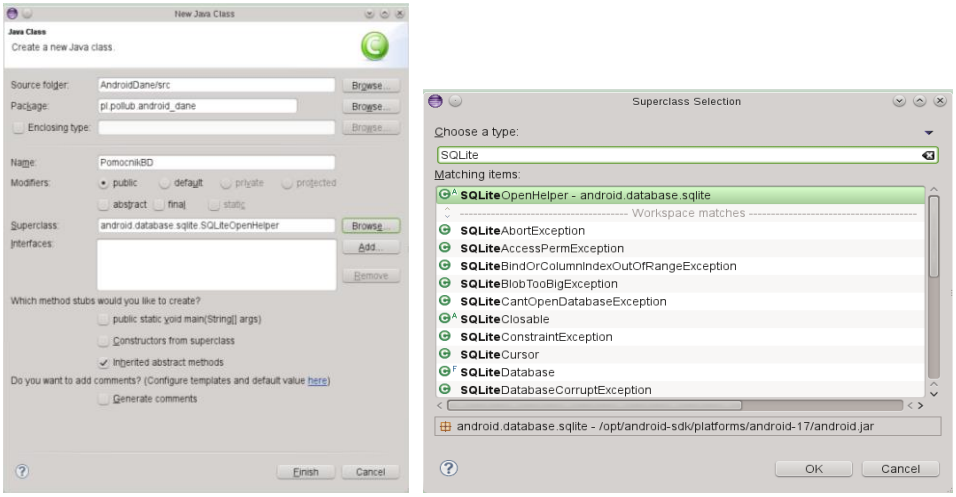
3.2.1. Tworzenie klasy pomocnika

Aby korzystać z wbudowanej bazy *SQLite*, wygodnie jest skorzystać z klasy `SQLiteOpenHelper`, która ułatwia tworzenie bazy i zarządzanie nią. Ponieważ klasa `SQLiteOpenHelper` zawiera metody abstrakcyjne, należy stworzyć własną klasę potomną. Musi ona przeciążać metody `onCreate()` oraz `onUpgrade()` (te metody są abstrakcyjne), dodatkowo można zdefiniować też `onDowngrade()` i `onOpen()`. Wymienione metody są wywoływane odpowiednio w momencie:

- tworzenia bazy,
- aktualizacji do nowszej wersji (gdy zmieni się wersja bazy danych – dzięki temu można uwzględnić ewentualne zmiany w strukturze bazy),
- przywracania starszej wersji,

- otwierania bazy. W przypadku metody `onOpen()` należy sprawdzić, czy baza jest otwierana w trybie tylko-do-odczytu za pomocą funkcji `isReadOnly()`.

Klasę pomocnika utworzymy, wykorzystując kreator nowych klas (rys. 3.4). Wybieramy *menu kontekstowe katalogu src | New | Class*. Nową klasę nazwiemy `PomocnikBD` i umieścimy w standardowym pakiecie naszej aplikacji `pl.pollub.android_dane`. W polu *superclass* jako klasę bazową wybieramy: `android.database.sqlite.SQLiteOpenHelper`. Dodatkowo upewnimy się, czy zaznaczona jest opcja tworzenia dziedziczonych metod abstrakcyjnych (*Inherited abstract methods*), dzięki czemu metody `onCreate()` i `onUpgrade()` zostaną dodane automatycznie.



Rys. 3.4. Dodawanie klasy pomocnika za pomocą kreatora

Listing 3.2. Fragment klasy pomocnika bazy danych

```
public class PomocnikBD extends SQLiteOpenHelper {
       private Context mKontekst;
       public final static int WERSJA_BAZY = 1;
       public final static String NAZWA_BAZY = "baza_telefonow";
       public final static String NAZWA_TABELI = "telefony";
       public final static String ID = "_id";
       public final static String PRODUCENT = "producent";
       public final static String MODEL = "model";
       public final static String ANDROID = "android";
       public final static String WWW = "www";
```

```
        public final static String ETYKIETA = "PomocnikBD";
        public final static String TW_BAZY =
           "CREATE TABLE " + NAZWA_TABELI + "("
          + ID + " integer primary key autoincrement, " +
           PRODUCENT + " text not null," + MODEL +
           " text not null," + ANDROID + " text not null," + WWW +
           " text);";

        private static final String KAS_BAZY =
           "DROP TABLE IF EXISTS " + NAZWA_TABELI;

        public PomocnikBD(Context context) {
               super(context, NAZWA_BAZY, null, WERSJA_BAZY);
               mKontekst = context;
        }

        @Override
        public void onCreate(SQLiteDatabase db) {
               //tworzenie bazy
        }

        @Override
        public void onUpgrade(SQLiteDatabase db,
                              int oldVersion, int newVersion) {
               //aktualizacja bazy
        }

}
```

Listing 3.2 przedstawia szkielet klasy pomocniczej obsługującej bazę danych. Ponieważ wiele funkcji do obsługi bazy wymaga podawania nazw tabel czy kolumn:

- zdefiniujemy w klasie pomocniczej statyczne pola zawierające wspomniane nazwy
- konsekwentnie będziemy korzystać z tych pól w odwołaniach do tabel/kolumn

Dzięki temu literówka w takim wywołaniu metody `query()`

```
Cursor kursor = mBD.query(true, PomocnikBD.NAZWS_TABELI, ...
```

zostanie przez kompilator wychwycona, natomiast w tym

```
Cursor kursor = mBD.query(true, "nazws_tabeli", ...
```

spowoduje błąd w trakcie wykonywania aplikacji.

Do tworzenia/kasowania tabel wykorzystamy metodę `execSQL()` klasy `SQLiteDatabase`. Pozwala ona wykonywać polecenia niezwracające danych, tak więc wykorzystamy ją do utworzenia bazy (`db` jest parametrem metod `onCreate()` i `onUpgrade()`)

```
db.execSQL(TW_BAZY);
```

Stała `TW_BAZY` została zdefiniowana w klasie pomocnika i zawiera polecenie `CREATE TABLE`. W naszej aplikacji utworzymy prostą tabelę zawierającą kilka kolumn takich jak: identyfikator, producent, model, wersja Androida oraz strona WWW. Szczególną uwagę warto zwrócić na kolumnę o nazwie `„_id”`. W zasadzie każda tabela powinna zawierać takie pole, ponieważ wiele klas frameworku Androida zakłada istnienie tej kolumny.

Metodę `execSQL()` wykorzystamy także do przeprowadzenia jej aktualizacji

```
db.execSQL(KAS_BAZY);
onCreate(db);
```

Jak widać aktualizacja jest dość prosta – polega na skasowaniu bieżącej bazy i utworzeniu jej od nowa (w „prawdziwej” aplikacji należałoby zadbać o przeniesienie danych). Warto zaznaczyć, że w ten sposób nie można wykonać np. polecenia `SELECT` – ponieważ ono zwraca dane.

Ćwiczenie 3.3 (do samodzielnego wykonania)

Korzystając z informacji z punktu 3.2.1 stwórz kompletną klasę `PomocnikBD` w projekcie z aplikacją *„Baza telefonów”*.

3.2.2. Korzystanie z bazy danych

Zanim rozpoczniemy korzystanie z bazy musimy ją otworzyć. Otwieranie bazy jest bardzo proste. Należy stworzyć obiekt klasy pomocniczej (w naszym

wypadku PomocnikBD), a następnie wywołać na jego rzecz metodę getWritebleDatabase() – jeżeli baza ma umożliwiać zapis i odczyt, lub getReadableDatabase() – jeżeli baza ma być otwarta w trybie tylko-do-odczytu. Metody get....Database() zwracają obiekt klasy SQLiteDatabase. Warto zaznaczyć, że można ją wywoływać dowolną liczbę razy, we wszystkich miejscach, w których baza jest potrzebna [8].

```
PomocnikBD mPomocnikBD=new PomocnikBD(this);
SQLiteDatabase mBD=mPomocnikBD.getWritableDatabase();
```

Gdy baza danych nie jest już potrzebna, należy ją zamknąć za pomocą metody close().

```
mBD.close();
```

Operacją bardzo często wykonywaną na bazach danych jest SELECT. Do wykonywania zapytań służy funkcja query() klasy SQLiteDatabase. Wymaga ona podania m.in. nazwy tabeli, nazw kolumn oraz klauzuli WHERE. Jeżeli jakiś argument nie jest potrzebny, ustawiamy jego wartość na null. Jak widać w poniższym przykładzie, tabelę łańcuchów można skonstruować w miejscu. Warto też zwrócić uwagę na to, że klauzula WHERE może (ale nie musi) być rozbita na 2 parametry. Pierwszy z nich (oznaczony where) zawiera łańcuch np. „kolumna1=? and kolumna2=?” a kolejny (whereArgs) tablicę wartości do wstawienia w miejsce „?”. Wartości są wstawiane w miejsce „?” w takiej kolejności, w jakiej są podane. Drugi sposób przekazywania parametrów klauzuli WHERE jest zalecany i bezpieczniejszy – wartości, które są wstawiane w miejsce znaków zapytania, nie są interpretowane jako część zapytania, co uniemożliwia wstrzyknięcie kodu SQL.

```
Cursor kursor = mBD.query(true, //distinct
  PomocnikBD.NAZWA_TABELI,  //tabela
   new String[]{PomocnikBD.ID,PomocnikBD.KOLUMNA1,
  PomocnikBD.KOLUMNA2}, //kolumny
  null,  //where
   null,  //whereArgs - argumenty zastępujące "?" w where
   null,  //group by
   null,  //having
   null,  //order by
   null); //limit
```

Wynikiem działania metody `query()` jest kursor. Ponieważ po wykonaniu zapytania nie wskazuje on na pierwszy rekord, przed odczytaniem danych należy wykonać instrukcję:

```
kursor.moveToFirst();
```

W dalszej kolejności można odczytać wartości poszczególnych pól, lecz najpierw należy odczytać z kursora indeks kolumny (parametrem metody `getColumnIndexOrThrow()` jest łańcuch zawierający nazwę kolumny)

```
int indeksKolumny=
   kursor.getColumnIndexOrThrow(PomocnikBD.KOLUMNA1);
```

a potem wartość pola rekordu

```
String wartosc=kursor.getString(indeksKolumny);
```

Po odczytaniu pól można przejść do następnego rekordu

```
kursor.moveToNext();
```

Przed odczytaniem kolejnego rekordu należy sprawdzić, czy nie odczytano już wszystkich rekordów zwróconych przez zapytanie. Służy do tego metoda `kursor.isAfterLast()`. Po odczytaniu wszystkich wartości kursor należy zamknąć

```
kursor.close();
```

Kolejną ważną operacją jest dodawanie rekordów. Wymaga ono, w pierwszej kolejności, utworzenia i wypełnienia obiektu klasy `ContentValues`

```
ContentValues wartosci = new ContentValues();
wartosci.put(PomocnikBD.KOLUMNA1,"wartość pola 1");
wartosci.put(PomocnikBD.KOLUMNA2,"wartość pola 2");
```

a następnie wywołania metody `insert()` klasy `SQLiteDatabase`

```
mIdWiersza=mBD.insert(PomocnikBD.NAZWA_TABELI, //tabela
                      null, //nullColumnHack
                      wartosci); //wartości
```

Funkcja zwraca ID (z kolumny `_id`) wstawionego rekordu lub -1 jeżeli wystąpił błąd.

Aktualizacja rekordów przebiega analogicznie do dodawania

```
ContentValues wartosci = new ContentValues();
wartosci.put(PomocnikBD.KOLUMNA1,"wartość pola 1");
wartosci.put(PomocnikBD.KOLUMNA2,"wartość pola 2");
mBD.update(PomocnikBD.NAZWA_TABELI, //tabela
    wartosci, //wartości
    PomocnikBD.ID+"="+mIdWiersza, //where
     null); //whereArgs (zastępują "?" w parametrze where)
```

Warto zwrócić uwagę, że klauzula `WHERE` również w tym przypadku może być rozbita na 2 parametry.

Ostatnią operacją jest usuwanie rekordów. Wymaga ona tylko podania nazwy tabeli oraz klauzuli `WHERE`.

```
mBD.delete(PomocnikBD.NAZWA_TABELI, //tabela
  PomocnikBD.ID+"="+mIdWiersza, //where
  null); //whereArgs (zastępują "?" w parametrze where)
```

3.2.3. Tworzenie dostawcy treści

Dostawcy treści (*ang. content providers*) w Androidzie standaryzują dostęp do danych tzn. udostępniają je w sposób podobny do bazy *SQLite*, nawet jeżeli źródłem danych jest np. zwykły plik. Jak już wspomniano, bazy danych *SQLite* są dostępne wyłącznie w ramach aplikacji, która ją stworzyła. Za pośrednictwem dostawców treści można również udostępniać dane innym aplikacjom. Ich inną zaletą jest kompatybilność z wieloma elementami frameworku Androida i zalecanym sposobem tworzenia aplikacji. Tak więc nawet jeśli nie chcemy udostępniać danych poza naszą aplikację, a źródłem danych jest baza *SQLite* , to wbrew temu co możemy przeczytać na stronie [2] to i tak musimy stworzyć dostawcę treści. Alternatywą jest korzystanie z przestarzałego i niezalecanego API.

Dane w dostawcach treści identyfikowane są za pomocą jednorodnych identyfikatorów zasobów – URI (*ang. uniform resource identifier*), a dostępne za pomocą typowych bazodanowych metod: `query()`, `insert()`, `update()` i `delete()`. W naszej aplikacji stworzymy klasę TelefonyProvider opisującą dostawcę udostępniającego dane za pośrednictwem następujących URI: „`content://pl.pollub.android_dane.TelefonyProvider/telefony`” lub „`content://pl.pollub.android_dane.TelefonyProvider/telefony/#`” gdzie # to identyfikator wiersza.

Przejdziemy zatem do tworzenia naszego dostawcy. Rozpoczniemy od dodania klasy `TelefonyProvider` dziedziczącej po klasie abstrakcyjnej `ContentProvider`. Automatycznie zostaną dodane metody, które należy zaimplementować. Szablon dostawcy przedstawiono na listingu 3.3.

Listing 3.3. Szablon dostawcy treści

```
public class TelefonyProvider extends ContentProvider {
	@Override
	public int delete(Uri uri, String selection,
			String[] selectionArgs) {
		return 0;
	}
	@Override
	public String getType(Uri uri) {
		return null;
	}
	@Override
	public Uri insert(Uri uri, ContentValues values) {
		return null;
	}
	@Override
	public boolean onCreate() {
		return false;
	}
	@Override
	public Cursor query(Uri uri, String[] projection,
			String selection,
			String[] selectionArgs,
			String sortOrder) {
		return null;
	}
	@Override
	public int update(Uri uri, ContentValues values,
			String selection,
			String[] selectionArgs) {
		return 0;
	}
}
```

Jak widać są to metody podobne do tych z klasy `SQLiteDatabase`. Dodamy teraz do klasy kilka pól i stałych, których będziemy używać w metodach, które zaimplementujemy.

Listing 3.4. Pola i stałe w klasie dostawcy

```
public class TelefonyProvider extends ContentProvider {

       private PomocnikBD mPomocnikBD;

       private static final String IDENTYFIKATOR =
                    "pl.pollub.android_dane.TelefonyProvider";

       public static final Uri URI_ZAWARTOSCI =
         Uri.parse("content://" + IDENTYFIKATOR +"/"+
         PomocnikBD.NAZWA_TABELI);

       private static final int CALA_TABELA = 1;
       private static final int WYBRANY_WIERSZ = 2;

       private static final UriMatcher sDopasowanieUri =
         new UriMatcher(UriMatcher.NO_MATCH);
       static {
              sDopasowanieUri.addURI(IDENTYFIKATOR,
                PomocnikBD.NAZWA_TABELI,
                CALA_TABELA);
              sDopasowanieUri.addURI(IDENTYFIKATOR,
                PomocnikBD.NAZWA_TABELI + "/#",
                WYBRANY_WIERSZ);
       }
       //... - część kodu usunięto
}
```

Pola i stałe używane w klasie `TelefonyProvider` przedstawiono na listingu 3.4. Pole `mPomocnikBD` będzie przechowywać referencję do obiektu pomocnika (obiektu klasy `PomocnikBD`), za pomocą którego uzyskamy dostęp do bazy danych.

Stała `IDENTYFIKATOR` (*ang. authority*) pozwala na odróżnienie naszego dostawcy od innych. Identyfikator, a także stała „`content://`” oraz nazwa tabeli z bazy danych wchodzą w skład stałej `URI_ZAWARTOSCI`. To za jej pomocą będziemy się odwoływać do naszego dostawcy.

Kolejne stałe: `CALA_TABELA` i `WYBRANY_WIERSZ` będą stosowane wewnętrznie i pozwolą nam na rozróżnienie dwóch rodzajów URI, które będzie rozumiał nasz dostawca.

Do pola statycznego `sDopasowanieUri` przypisujemy obiekt klasy `UriMatcher`. Następnie definiujemy w tym obiekcie dwa rozpoznawane przez dostawcę URI. Tak jak wspomniano wcześniej pierwszy z nich będzie miał postać:
„`content://pl.pollub.android_dane.TelefonyProvider/telefony`”
a drugi
„`content://pl.pollub.android_dane.TelefonyProvider/telefony/#`”.
Warto zwrócić uwagę, że do pierwszej postaci przypisujemy stałą `CALA_TABELA` a do drugiej `WYBRANY_WIERSZ`. Dzięki temu, gdy za pomocą `sDopasowanieUri.match()` będziemy analizować URI, to w przypadku pierwszej postaci metoda zwróci stałą `CALA_TABELA`, a w przypadku drugiej zwróci `WYBRANY_WIERSZ`.

Listing 3.5. Metoda `onCreate()` tworząca dostawcę

```
@Override
public boolean onCreate() {
       mPomocnikBD = new PomocnikBD(getContext());
       return false;
}
```

Listing 3.5 przedstawia metodę odpowiedzialną za utworzenie dostawcy. Jak widać jest ona bardzo prosta i polega jedynie na utworzeniu obiektu pomocnika. Metoda ta powinna wykonywać się możliwie szybko, więc otwieranie bazy danych zostawimy na później. Przejdziemy teraz do tworzenia jednej z kluczowych metod naszego dostawcy. Jest to metoda `query()`.

Listing 3.6. Metoda `query()` oraz metoda pomocnicza dodająca wartość klucza

```
@Override
public Cursor query(Uri uri, String[] projection,
                    String selection,
                    String[] selectionArgs,
                    String sortOrder) {
  int typUri = sDopasowanieUri.match(uri);
  SQLiteDatabase baza =
    mPomocnikBD.getWritableDatabase();
     Cursor kursorTel = null;
  switch (typUri) {
    case CALA_TABELA:
      kursorTel = baza.query(false,
       PomocnikBD.NAZWA_TABELI,
```

```
            projection, selection, selectionArgs, null, null,
            sortOrder, null, null);
          break;
        case WYBRANY_WIERSZ:
          kursorTel = baza.query(false,
            PomocnikBD.NAZWA_TABELI, projection,
            dodajIdDoSelekcji(selection, uri),
            selectionArgs, null, null, sortOrder, null,
            null);
          break;
        default:
          throw new IllegalArgumentException("Nieznane URI: "
            + uri);
      }

          kursorTel.setNotificationUri(
            getContext().getContentResolver(), uri);
          return kursorTel;
    }

    private String dodajIdDoSelekcji(String selekcja, Uri uri)
    {
      if (selekcja!=null && !selekcja.equals(""))
        selekcja = selekcja + " and " + PomocnikBD.ID + "="
          + uri.getLastPathSegment();
      else
        selekcja = PomocnikBD.ID + "=" +
          uri.getLastPathSegment();
      return selekcja;
    }
```

Metoda `query()` dostawcy (listing 3.6) jest dość podobna do metody `query()` klasy `SQLiteDatabase`. Część argumentów się pokrywa, jednak występują także różnice. Pierwszą z nich jest argument `uri`. Określa on jakich danych oczekujemy od dostawcy. Kolejny – `projection` (projekcja) – określa, które kolumny z tabeli nas interesują. Jest to odpowiednik parametru `columns` metody `query()` bazy danych. Kolejne `selection` (selekcja) i `selectionArgs` (argumenty selekcji) pozwalają na wybranie określonych wierszy, które zostaną zwrócone przez dostawcę. Są one odpowiednikami argumentów `where` i `whereArgs`. Ostatni – `sortOrder` (porządek sortowania) jest odpowiednikiem `orderBy` i określa w jakiej kolejności, dostawca powinien zwrócić dane. Wartości argumentów: `projection`, `selection`, `selectionArgs` oraz `sortOrder` określa się tak samo jak w przypadku bezpośredniego korzystania z bazy danych.

Przejdziemy teraz do analizy treści metody `query()` naszego dostawcy. Na początku identyfikujemy rodzaj `uri` i zapisujemy wynik tej analizy (jak wspomniano wcześniej będzie to jedna ze stałych: `CALA_TABELA` lub `WYBRANY_WIERSZ`. Kolejnym krokiem jest uzyskanie dostępu do bazy danych. Następnie w zależności od zidentyfikowanego typu wykonujemy odpowiednie działanie:

- gdy odczytujemy dane z całej tabeli – po prostu przekazujemy parametry z wywołania metody `query()` dostawcy do metody `query()` bazy.
- gdy odczytujemy dane jednego wiersza – do argumentu `selection` dodajemy wartość klucza, który określa żądany wiersz. Wykorzystujemy do tego metodę pomocniczą `dodajIdDoSelekcji()`. Potrzeba modyfikacji selekcji wynika z tego, że numer wiersza przekazywany jest standardowo poprzez URI, a nie jak w przypadku języka SQL określany w klauzuli `WHERE`.
- Gdy URI nie zostało rozpoznane zgłaszany jest wyjątek.

W wyniku wykonania zapytania otrzymujemy kursor. Wykorzystujemy metodę `setNotificationUri()` kursora ustawiającą `uri`, które będzie można monitorować pod kątem zmiany danych (loader może zarejestrować obserwatora, który będzie powiadamiany o modyfikacji danych [16]). Na koniec metoda zwraca kursor uzyskany z bazy.

Metoda pomocnicza `dodajIdDoSelekcji()` sprawdza, czy łańcuch opisujący selekcję istnieje. Jeżeli tak jest, to dopisuje do niego warunek „ `and _id = #` „, gdzie `#` to numer wiersza występujący na końcu `uri`. W przeciwnym wypadku – po prostu ustawia warunek „`_id = #`”.

Listing 3.7. Metoda `update()` dostawcy treści

```
@Override
public int update(Uri uri, ContentValues values,
    String selection, String[] selectionArgs) {
  int typUri = sDopasowanieUri.match(uri);
  SQLiteDatabase baza = mPomocnikBD.getWritableDatabase();

  int liczbaZaktualizowanych = 0;
  switch (typUri) {
    case CALA_TABELA:
      liczbaZaktualizowanych =
        baza.update(PomocnikBD.NAZWA_TABELI,
                    values, selection, selectionArgs);
      break;
    case WYBRANY_WIERSZ:
      liczbaZaktualizowanych =
```

```
            baza.update(PomocnikBD.NAZWA_TABELI,
              values, dodajIdDoSelekcji(selection, uri),
              selectionArgs);
          break;
        default:
          throw new IllegalArgumentException("Nieznane URI: "
            + uri);
      }

      getContext().getContentResolver().
        notifyChange(uri, null);
      return liczbaZaktualizowanych;
    }
```

Metoda `update()` dostawcy treści została przedstawiona na listingu 3.7. Jak widać, jej argumenty jeszcze bardziej niż w przypadku `query()` przypominają argumenty metody `update()` bazy danych. Jedyna różnica polega na tym, że w przypadku dostawcy jako pierwszy parametr podajemy `uri`, a w przypadku bazy – nazwę tabeli. Pozostałe parametry są swoimi odpowiednikami i są to kolejno: `values` – nowe wartości pól, `selection` – łańcuch opisujący warunki, które muszą spełniać aktualizowane wiersze, `selectionArg` – potrzebny w przypadku wykorzystywania selekcji w postaci „`pole = ?`”. Argumenty działają tak jak w przypadku omówionej wcześniej metody `update()` z bazy danych.

Działanie metody `update()` jest podobne do metody `query()`. Na początku identyfikowany jest rodzaj `uri`, a baza danych otwierana jest do zapisu. Następnie w przypadku operacji dotyczącej całej tabeli parametry są po prostu przekazywane do metody `update()` bazy danych. Gdy operacja dotyczy jednego wiersza, do argumentu `selection` dodawany jest identyfikator wiersza określony w `uri`. W przypadku nieprawidłowego `uri` zgłaszany jest wyjątek. Pod koniec metoda informuje obiekt zapewniający dostęp do dostawców treści o zmianie danych (dzięki temu wspomniany wcześniej obserwator może zareagować na zmianę danych).

Ostatnią rzeczą o jakiej musimy pamiętać, tworząc nowego dostawcę treści, jest modyfikacja manifestu. Musimy upewnić się, że nasz nowy dostawca został w nim zadeklarowany, tak jak to pokazano na listingu 3.8.

Listing 3.8. Manifest aplikacji uwzględniający nowego dostawcę

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="pl.pollub.android_dane"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="17"
        android:targetSdkVersion="17" />
    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
          android:name="pl.pollub.android_dane.ListaTelActivity"
          android:label="@string/app_name" >
          <intent-filter>
            <action android:name="android.intent.action.MAIN" />

            <category android:name=
              "android.intent.category.LAUNCHER" />
          </intent-filter>
        </activity>
        <provider
            android:name=
              "pl.pollub.android_dane.TelefonyProvider"
            android:authorities=
              "pl.pollub.android_dane.TelefonyProvider" >
        </provider>
    </application>
</manifest>
```

Ćwiczenie 3.4 (do samodzielnego wykonania)

Korzystając z informacji z punktów 3.2.2 i 3.2.3 stwórz kompletną klasę `TelefonyProvider` w projekcie z aplikacją *„Baza telefonów”*. W szczególności uzupełnij metodę `insert()` oraz `delete()`. Metoda `insert()` obsługuje wyłącznie `uri` odnoszące się do całej tabeli (w trakcie dodawania nie podajemy numeru wiersza). Metoda `delete()` obsługuje oba rodzaje `uri`. Pamiętaj, że obie metody muszą powiadamiać obiekt klasy `ContentResolver` o zmianie danych (tak jak `update()`).

3.2.4 Wypełnianie listy danymi z dostawcy treści

Jakkolwiek z dostawców treści można korzystać na wiele sposobów, zalecaną metodą wypełniania list jest korzystanie z tzw. *loaderów*. Zostały one wprowadzone w Androidzie 3.0, ale są również dostępne w bibliotece wsparcia, więc ich użycie nie ogranicza aplikacji do nowszych wersji systemu. Z loaderów można korzystać w aktywnościach i fragmentach w celu asynchronicznego ładowania danych. Ponieważ dostawcy treści mogą pobierać dane z różnych źródeł (w tym dość wolnych), działanie *loaderów* w tle (poza wątkiem interfejsu użytkownika) rozwiązuje problem blokowania się GUI. Nie ma konieczności samodzielnego implementowania osobnego wątku. Pierwszą, prostą wersję, aktywności `ListaTelActivity` przedstawiono na listingu 3.9.

Listing 3.9. Pierwsza wersja aktywności `ListaTelActivity` korzystająca z dostawcy treści

```
public class ListaTelActivity extends ListActivity implements
      LoaderCallbacks<Cursor> {

      private SimpleCursorAdapter mAdapterKursora;

      @Override
      protected void onCreate(Bundle savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_lista_tel);
            wypelnijPola();
      }

      private void wypelnijPola() {
            getLoaderManager().initLoader(0, null, this);
            // utworzenie mapowania między kolumnami tabeli a
            //kolumnami wyświetlanej listy
            String[] mapujZ = new String[] {
              PomocnikBD.PRODUCENT, PomocnikBD.MODEL };
            int[] mapujDo = new int[] { R.id.producentText,
                                        R.id.modelText };
            // adapter wymaga aby wyniku zapytania znajdowała
            //się kolumna _id
            mAdapterKursora = new SimpleCursorAdapter(this,
              R.layout.wiersz_listy, null, mapujZ, mapujDo, 0);
            setListAdapter(mAdapterKursora);
      }
```

```
        @Override
        public Loader<Cursor> onCreateLoader(int id, Bundle args)
        {
                String[] projekcja = { PomocnikBD.ID,
                                       PomocnikBD.PRODUCENT,
                                       PomocnikBD.MODEL };
                CursorLoader loaderKursora = new CursorLoader(this,
                             TelefonyProvider.URI_ZAWARTOSCI,
                             projekcja, null, null, null);
                return loaderKursora;
        }

        @Override
        public void onLoadFinished(Loader<Cursor> loader,
                                   Cursor dane) {
                mAdapterKursora.swapCursor(dane);
        }
        @Override
        public void onLoaderReset(Loader<Cursor> loader) {
                mAdapterKursora.swapCursor(null);
        }
}
```

Jak widać, do wyświetlenia wyników wykorzystamy aktywność dziedziczącą po `ListActivity`. Jest to standardowa aktywność (`Activity`) z wbudowanym elementem typu `ListView` i kilkoma dodatkowymi funkcjami ułatwiającymi obsługę list. Dlatego w układzie, który zdefiniowaliśmy wcześniej, użyliśmy standardowych identyfikatorów `"@android:id/list"` i `"android:id/empty"` – dzięki temu aktywność będzie mogła je odszukać.

W aktywności `ListaTelActivity` definiujemy tylko jedno pole `mAdapterKursora` klasy `SimpleCursorAdapter`. Klasa ta pozwala na powiązanie kursora i widoku listy (`ListView`).

Metoda `onCreate()` ustawia układ elementów i wypełnia listę za pomocą metody pomocniczej – `wypelnijPola()`. Metoda ta rozpoczyna od zainicjowania loadera za pomocą metody `initLoader()`. Jej parametry to: identyfikator loadera, argumenty dla loadera, a także referencja do obiektu klasy implementującej interfejs `LoaderCallbacks<D>`. Identyfikator ustawiamy na 0 (korzystamy tylko z jednego loadera), nie przekazujemy do loadera żadnych argumentów (tzn. drugi argument jest równy `null`), a interfejs zaimplementujemy w aktywności `ListaTelActivity`. Po inicjalizacji loadera tworzone są tablice określające powiązanie pomiędzy nazwami nazw kolumn a identyfikatorami etykiet tekstowych, w których będą wyświetlane dane. Pierwszą tablicą jest tablica łańcuchów `mapujZ`, a drugą tablica liczb

całkowitych `mapujDo`. Jak widać na liście będziemy wyświetlać tylko dwie kolumny: nazwę producenta i nazwę modelu telefonu. Następnie metoda `wypelnijPola()` tworzy prosty adapter kursora – obiekt klasy `SimpleCursorAdapter` – przekazując do klasy odpowiednie parametry m.in. identyfikator układu wiersza listy i utworzone właśnie tablice. Na koniec w elemencie listy ustawiany jest nowy adapter (wykorzystujemy do tego charakterystyczną dla `ListActivity` metodę `setListAdapter()`).

Na wspomniany wcześniej interfejs `CursorCallbacks<D>` składają się trzy metody: `onCreateLoader()`, `onLoadFinished()` i `onLoaderReset()`. Pierwsza z nich jest odpowiedzialna za utworzenie loadera. W naszej aplikacji tworzy ona tablicę łańcuchów określającą projekcję (używamy stałych z klasy `PomocnikBD` do wybrania odpowiednich kolumn), a następnie tworzy nowy obiekt klasy `CursorLoader`. Jak widać używamy tutaj URI zdefiniowanego jako stała w klasie `TelefonyProvider`. Kolejna metoda `onLoadFinished()` wywoływana w momencie, gdy dane zostały wczytane, ustawia kursor z nowymi danymi w adapterze kursora (wtedy dane pojawią się na liście). Ostatnia z metod – `onLoaderReset()` wywoływana jest w momencie, gdy loader jest resetowany (kursor przekazany metodzie `onLoadFinished()` jest zamykany [11]). Kursor w adapterze ustawiany jest wtedy na `null`, ponieważ dane nie są już dostępne.

3.2.5 Debugowanie bazy danych

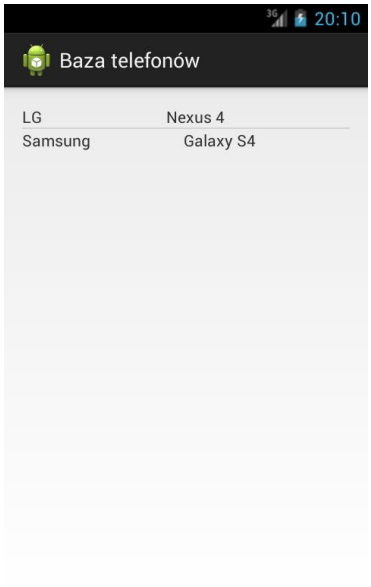
Czasami zachodzi konieczność sprawdzenia poprawności danych zapisywanych przez aplikację lub ich ręcznego umieszczenia w bazie. Aby uzyskać dostęp do bazy danych wykorzystywanej przez aplikację należy:

- uruchomić emulator,
- w wierszu poleceń wykonać polecenie `adb devices` – powinna wtedy pojawić się lista pracujących emulatorów lub urządzeń podłączonych przez USB,
- w wierszu poleceń wykonać polecenie `adb -s id shell`, gdzie `id` to identyfikator (np. `emulator-5554`) urządzenia, na którym znajduje się baza danych – powinna zostać udostępniona powłoka (shell) systemu android na urządzeniu,
- na urządzeniu wykonać polecenie `sqlite3 /data/data/nazwa.pakietu.aplikacji/databases/nazwa_bazy` – baza danych powinna zostać otworzona, można wykonywać polecenia SQL, `.help` powoduje wyświetlenie pomocy.

Ćwiczenie 3.5 (do samodzielnego wykonania)

Korzystając z informacji zawartych w punkcie 3.2.4 uzupełnij kod aktywności `ListaTelActivity`. Następnie korzystając z punktu 3.2.5, dodaj ręcznie kilka telefonów do bazy danych (`insert into … values (null,`

„...);). Sprawdź działanie aplikacji. Przykładowy efekt wykonania aplikacji przedstawiono na rysunku 3.5.



Rys. 3.5. Lista telefonów

3.2.6. Zapisywanie/odczytywanie danych za pośrednictwem dostawcy

Często danych nie umieszczamy na przewijanej liście. Z taką sytuacją mamy do czynienia np. wtedy, gdy chcemy dodać nowy telefon do naszej bazy, wyświetlić szczegóły już istniejącego, bądź też zmodyfikować telefon już znajdujący się w bazie. Nie będziemy wtedy korzystać z loaderów, lecz użyjemy dostawcy treści bezpośrednio.

Przejdziemy teraz do modyfikacji szkieletu aktywności `EdycjaTelActivity`, tak aby pozwalała na dodawanie i wyświetlanie szczegółowych danych poszczególnych telefonów.

Listing 3.10. Aktywność `EdycjaTelActivity` – korzystanie z dostawcy treści

```
public class EdycjaTelActivity extends Activity {
      private long mIdWiersza;
      private EditText mProducentEdycja;
      private EditText mModelEdycja;
```

```
private EditText mAndroidEdycja;
private EditText mWWWEdycja;


@Override
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_edycja_tel);

  //... - część kodu usunięto
  // odczytanie referencji pól tekstowych na podstawie id

  mIdWiersza = -1;
  if (savedInstanceState != null)
    mIdWiersza =
      savedInstanceState.getLong(PomocnikBD.ID);
  else {
    Bundle tobolek = getIntent().getExtras();
    if (tobolek != null)
      mIdWiersza = tobolek.getLong(PomocnikBD.ID);
  }

if (mIdWiersza != -1)
  wypelnijPola();

  //... - część kodu usunięto
  // ustawienie obsługi kliknięcia przycisków: Zapisz,
  //Anuluj
}

@Override
protected void onSaveInstanceState(Bundle outState) {
  super.onSaveInstanceState(outState);
  outState.putLong(PomocnikBD.ID, mIdWiersza);
}

private boolean sprawdzNapisy() {
 return !(mProducentEdycja.getText().toString().equals("")
        || mModelEdycja.getText().toString().equals("")
        || mAndroidEdycja.getText().toString().equals("")
        || mWWWEdycja.getText().toString().equals(""));
}
```

```
private void wypelnijPola() {
  String projekcja[] = { PomocnikBD.PRODUCENT,
                         PomocnikBD.MODEL,
                        PomocnikBD.ANDROID,
                        PomocnikBD.WWW };
  Cursor kursorTel = getContentResolver().query(
    ContentUris.withAppendedId(
      TelefonyProvider.URI_ZAWARTOSCI,
      mIdWiersza), projekcja, null, null, null);
  kursorTel.moveToFirst();
  int indeksKolumny =
    kursorTel.getColumnIndexOrThrow(PomocnikBD.PRODUCENT);
  String wartosc = kursorTel.getString(indeksKolumny);
  mProducentEdycja.setText(wartosc);
  mModelEdycja.setText(kursorTel.getString(
   kursorTel.getColumnIndexOrThrow(PomocnikBD.MODEL)));
  mAndroidEdycja.setText(kursorTel.getString(
    kursorTel.getColumnIndexOrThrow(PomocnikBD.ANDROID)));
  mWWWEdycja.setText(kursorTel.getString(
    kursorTel.getColumnIndexOrThrow(PomocnikBD.WWW)));
  kursorTel.close();
}

private void klknieciePrzyciskuZapisz() {
  if (sprawdzNapisy()) {
    ContentValues wartosci = new ContentValues();
    wartosci.put(PomocnikBD.PRODUCENT,
      mProducentEdycja.getText().toString());
    wartosci.put(PomocnikBD.MODEL,
      mModelEdycja.getText().toString());
    wartosci.put(PomocnikBD.ANDROID,
      mAndroidEdycja.getText().toString());
    wartosci.put(PomocnikBD.WWW,
      mWWWEdycja.getText().toString());
    if (mIdWiersza == -1) {
      Uri uriNowego = getContentResolver().insert(
        TelefonyProvider.URI_ZAWARTOSCI, wartosci);
    mIdWiersza = Integer.parseInt(
      uriNowego.getLastPathSegment());
  }
   else {
   int zmienionychWierszy = getContentResolver().update(
     ContentUris.withAppendedId(
       TelefonyProvider.URI_ZAWARTOSCI, mIdWiersza),
        wartosci, null, null);
  }
```

```
            setResult(RESULT_OK);
            finish();
          }
          else Toast.makeText(this,
           getString(R.string.wypelnij_pola_komunikat),
           Toast.LENGTH_SHORT).show();
        }

        private void kliknieciePrzyciskuAnuluj() {
          setResult(RESULT_CANCELED);
          finish();
        }
}
```

Listing 3.10 przedstawia istotne fragmenty aktywności `EdycjaTelActivity`. Na początku znajdują się pola. Pierwsze z nich przechowuje identyfikator wyświetlanego telefonu. Identyfikator ten jest tożsamy z wartością klucza głównego w tabeli `telefony` przechowywanej w bazie SQLite oraz z numerem wiersza w dostawcy treści. Warto zauważyć, że identyfikatory w Androidzie typowo mają typ `long`. Kolejne pola to referencje do obiektów reprezentujących pola tekstowe.

W metodzie `onCreate()` typowo ustawiamy układ aktywności i dokonujemy inicjalizacji. Fragmentem, na który warto zwrócić uwagę jest ustawianie identyfikatora wiersza przechowywanego w polu `mIdWiersza`. Jego wartość początkowa to -1. Jest ona zmieniana w zależności od sytuacji. Jeżeli aktywność została zrestartowana (np. po obróceniu urządzenia), to wtedy wartość identyfikatora odczytywana jest z obiektu `savedInstanceState` za pomocą metody `getLong()`. Jeżeli nie ma zachowanego stanu, oznacza to, że aktywność `EdycjaTelActivity` została wywołana przez inną aktywność. Wtedy w intencji, która uruchomiła `EdycjaTelActivity`, sprawdzamy obecność dodatkowych danych za pomocą `getExtras()`. Jeżeli są one dostępne, to z przekazanych parametrów próbujemy odczytać wartość identyfikatora. Następnie sprawdzamy czy wartość identyfikatora jest różna od -1, co oznacza, że musimy wczytać dane z dostawcy treści. Jeżeli identyfikatora nie udało się ustalić, oznacza to, że dodajemy nowy telefon do bazy.

Kolejna metoda `onSaveInstanceState()` jest odpowiedzialna za zachowanie identyfikatora wiersza. Warto tu przypomnieć, że o ile np. zawartość pól tekstowych jest zachowywana automatycznie, o tyle o zachowanie pól klasy musimy zadbać sami.

Metoda `sprawdzNapisy()` jest metodą pomocniczą, która sprawdza, czy we wszystkie pola tekstowe zostały wpisane jakieś dane.

Jedną z dwóch najbardziej rozbudowanych metod w tej aktywności jest metoda `wypelnijPola()`. Rozpoczyna ona od utworzenia tablicy projekcja

określającej, które kolumny chcemy pobrać z dostawcy (w tym wypadku wszystkie z wyjątkiem identyfikatora). Następnie za pośrednictwem obiektu klasy `ContentResolver` odpytujemy dostawcę treści. Metodzie `query()` przekazujemy tylko dwa niepuste argumenty: URI i projekcję. Warto zwrócić uwagę na to, w jaki sposób tworzymy URI. Wykorzystujemy tutaj klasę `ContentUris` i jej statyczną metodę `withAppendedId()`. Jej parametrami jest bazowe URI zdefiniowane w dostawcy treści oraz identyfikator wiersza, który chcemy dodać do URI. W wyniku wykonania zapytania otrzymujemy kursor, który przestawiamy na pierwszy element za pomocą `moveToFirst()`, odczytujemy z kursora indeksy kolejnych kolumn na podstawie ich nazw (`getColumnIndexOrThrow()`), odczytujemy też wartość pola, korzystając z odczytanych indeksów (`getString()`) i wreszcie ustawiamy odczytane wartości w polach tekstowych (`setText()`). Na koniec zamykamy kursor uzyskany z dostawcy.

Kolejną rozbudowaną metodą jest `kliknieciePrzyciskuZapisz()`. Jest to metoda pomocnicza, która jak nazwa wskazuje, obsługuje kliknięcie przycisku *Zapisz*. Na początku sprawdzana jest poprawność danych w polach tekstowych. Jeżeli dane są niekompletne, wyświetlany jest komunikat, jeżeli dane są poprawne tworzony jest obiekt klasy `ContentValues`. W obiekcie tym umieszczane są wartości zapisywane za pośrednictwem dostawcy. Do dodawania wartości służy metoda `put()`, której parametrami są nazwa pola i wartość pola. Po dodaniu wszystkich wartości sprawdzamy jaki jest identyfikator wiersza przechowywany w naszej aktywności. Jeżeli jest równy `-1`, to dodajemy nowy telefon za pomocą metody `insert()`, a w przeciwnym wypadku aktualizujemy wybrany wiersz za pomocą metody `update()`. Jako parametry metody `insert()` przekazujemy URI zdefiniowane w klasie dostawcy oraz obiekt zawierający wartości. Wynikiem wykonania `insert()` jest URI opisujące nowy wiersz. Odczytujemy z niego identyfikator nowego wiersza za pomocą `getLastPathSegment()` i zachowujemy go w polu `mIdWiersza`. W przypadku aktualizacji za pomocą metody `update()` dodajemy do URI identyfikator aktualizowanego wiersza. Nowe URI i nowe wartości są parametrami `update()`. Po zapisaniu danych ustawiamy wynik wykonania aktywności na `RESULT_OK` i kończymy aktywność.

Ostatnia z metod aktywności `EdycjaTelActivity` to `kliknieciePrzyciskuAnuluj()`. Jest to również metoda pomocnicza. Jest ona bardzo prosta, ustawia wynik wykonania aktywności na `RESULT_CANCELED` i kończy aktywność. Wszystkie zmiany dokonane w aktywności przepadają.

Listing 3.11. Modyfikacje `ListaTelActivity` pozwalające na uruchamianie drugiej aktywności

```
public class ListaTelActivity extends ListActivity implements
        LoaderCallbacks<Cursor>{

      //... - część kodu usunięto

      @Override
      public boolean onCreateOptionsMenu(Menu menu) {
        //... - część kodu usunięto
      }

      //... - część kodu usunięto

      @Override
      protected void onListItemClick(ListView l, View v,
                                     int position, long id) {
        super.onListItemClick(l, v, position, id);
        Intent zamiar =
          new Intent(this, EdycjaTelActivity.class);
        zamiar.putExtra(PomocnikBD.ID, id);
        startActivityForResult(zamiar, 0);
      }

      @Override
      public boolean onMenuItemSelected(int featureId,
                                        MenuItem item) {
        //... - część kodu usunięto
      }

      //... - część kodu usunięto

      private void tworzElement() {
        Intent zamiar =
          new Intent(this, EdycjaTelActivity.class);
                    zamiar.putExtra(PomocnikBD.ID, (long) -1);
                    startActivityForResult(zamiar, 0);
      }

      @Override
      protected void onActivityResult(int requestCode,
                           int resultCode, Intent data) {
        super.onActivityResult(requestCode, resultCode, data);
        getLoaderManager().restartLoader(0, null, this);
      }
}
```

Przejdziemy teraz do wprowadzenia modyfikacji w aktywności `ListaTelActivity` przedstawionych w listingu 3.11. Pozwolą one na uruchomienie `EdycjaTelActivity` w celu utworzenia nowego lub zmodyfikowania istniejącego telefonu. Trzy metody z listingu wymagają omówienia. Pierwsza z nich to `onListItemClick()`. Jest to kolejna metoda charakterystyczna dla `ListActivity`. Upraszcza ona obsługę zdarzenia polegającego na kliknięciu elementu listy. W metodzie tej tworzymy nową intencję, w której za pomocą metody `putExtra()` zapisujemy otrzymany przez parametr identyfikator wiersza (czyli równocześnie numer wiersza w dostawcy i wartość klucza w tabeli bazy danych). Następnie uruchamiamy drugą aktywność. Dzięki tej metodzie możliwe będzie przejście z listy do modyfikacji wybranego telefonu.

Drugą ważną metodą jest `onActivityResult()`. Jest ona wywoływana po powrocie z innej aktywności. Ponieważ dane telefonów mogły zostać zmodyfikowane restartujemy loader odpowiedzialny za wypełnienie listy. Parametry metody `restartLoader()` są identyczne jak metody `initLoader()`.

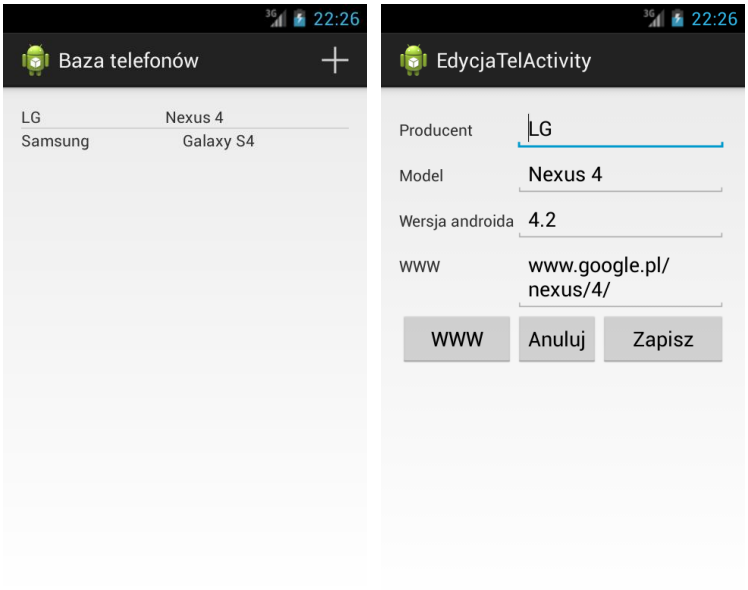
Ostatnia istotna metoda to `tworzElement()`. Jest to metoda pomocnicza, którą wykorzystamy w obsłudze akcji (umieszczonej na pasku akcji) pozwalającej na dodanie nowego telefonu do bazy. Podobnie jak `onListItemClick()` tworzy intencję i uruchamia drugą aktywność, jednak w tym wypadku ustawiamy identyfikator wiersza na -1 (co w drugiej aktywności oznacza dodawanie nowego telefonu).

Ćwiczenie 3.6 (do samodzielnego wykonania)

Wprowadź modyfikacje omówione w punkcie 3.2.6, a następnie uzupełnij brakujące fragmenty kodu. W szczególności:

- dodaj do aplikacji pasek akcji do aktywności `ListaTelActivity`. Plik XML opisujący akcje umieść w pliku `pasek_akcji_lista_tel.xml`. Jako identyfikatora akcji użyj `dodajMenu`. Stwórz dla akcji odpowiednią ikonkę. Uzupełnij metody `onCreateOptionsMenu()` i `onMenuItemSelected()`,
- uzupełnij metodę `onCreate()` aktywności `EdycjaTelActivity`.

Efekt wykonania aplikacji przedstawiono na rysunku 3.6.



Rysunek 3.6. Dodawanie/modyfikacja telefonów

3.2.7. Edycja wielu elementów listy

Operacją, którą do tej pory się nie zajęliśmy, jest kasowanie wierszy za pośrednictwem dostawcy. Służy do tego metoda `delete()`. Do zrealizowania tej funkcji w naszej aplikacji wykorzystamy *pasek kontekstowy* (*ang. contextual action bar*), który zgodnie z wytycznymi tworzenia aplikacji dla Androida zastępuje zwykłe menu kontekstowe [3]. Pasek będzie związany z listą i uaktywniany przez tzw. długie kliknięcie elementu. Po wyświetlaniu paska możliwe będzie wybranie kolejnych elementów listy, a następnie ich usunięcie lub anulowanie całej operacji. Wymagane modyfikacje przedstawiono na listingu 3.12.

Listing 3.12. Modyfikacje dodające hurtowe usuwanie telefonów

```
public class ListaTelActivity extends ListActivity implements
             LoaderCallbacks<Cursor>{

      //... - część kodu usunięto
```

```
@Override
protected void onCreate(Bundle savedInstanceState) {

  //... - część kodu usunięto

  ListView listaTelefonow = getListView();
  listaTelefonow.setChoiceMode(
    ListView.CHOICE_MODE_MULTIPLE_MODAL);
  listaTelefonow.setMultiChoiceModeListener(
    new MultiChoiceModeListener() {
      @Override
      public boolean onPrepareActionMode(ActionMode mode,
                                         Menu menu) {
        return false;
      }
      @Override
      public void onDestroyActionMode(ActionMode mode) {
      }
      @Override
      public boolean onCreateActionMode(ActionMode mode,
                                        Menu menu) {
        MenuInflater pompka = mode.getMenuInflater();
        pompka.inflate(R.menu.pasek_kontekstowy_listy,
                       menu);
        return true;
      }
      @Override
      public boolean onActionItemClicked(ActionMode mode,
                                         MenuItem item) {
        switch (item.getItemId()) {
          case R.id.kasujMenu:
            kasujZaznaczone();
            return true;
        }
        return false;
      }
      @Override
      public void onItemCheckedStateChanged(
        ActionMode mode, int position, long id,
        boolean checked) {
      }
  });
}

//... - część kodu usunięto
```

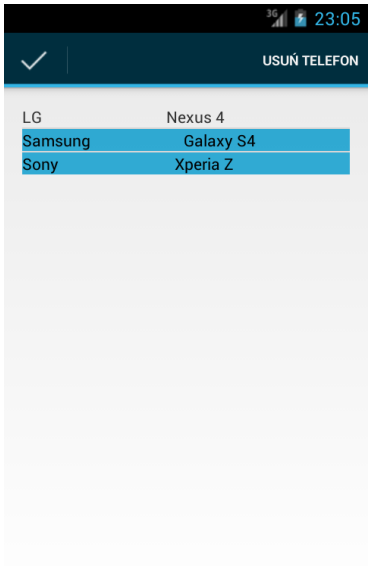
```
    private void kasujZaznaczone() {
      long zaznaczone[] = getListView().getCheckedItemIds();
      for (int i = 0; i < zaznaczone.length; ++i) {
        getContentResolver().delete(
          ContentUris.withAppendedId(
            TelefonyProvider.URI_ZAWARTOSCI, zaznaczone[i]),
          null, null);
      }
    }
    //... - część kodu usunięto
}
```

W pierwszej kolejności zmienimy metodę `onCreate()`. Musimy włączyć w niej odpowiedni tryb zaznaczania elementów listy za pomocą `setChoiceMode()`. Jest on określony stałą `ListView.CHOICE_MODE_MULTIPLE_MODAL`. Następnie tworzymy słuchacza implementującego interfejs `MultiChoiceModeListener` a odpowiedzialnego za obsługę *trybu akcji* (*ang. action mode*). Spośród pięciu metod interfejsu musimy zaimplementować dwie, a trzy pozostają puste. W pierwszej z nich – `onCreateActionMode()` tworzymy obiekty paska kontekstowego dokładnie w ten sam sposób, jak w przypadku omawianego w poprzednim rozdziale „zwykłego" paska akcji. Oczywiście wymagany jest tu plik XML opisujący akcje dostępne na pasku kontekstowym. Druga metoda interfejsu to `onActionItemClick()`, której zadaniem jest obsłużenie wybranej przez użytkownika akcji. Również ona działa analogicznie do metody `onMenuItemSelected()` „zwykłego" paska akcji.

W reakcji na wybór akcji usuwania telefonu/telefonów wywołujemy metodę pomocniczą `kasujZaznaczone()`. Metoda ta odczytuje identyfikatory zaznaczonych elementów listy za pomocą `getListView().getCheckedIds()`. W ten sposób otrzymuje tablicę o elementach typu `long` zawierającą identyfikatory wierszy / wartości klucza w bazie danych. Następnie w pętli wywołuje metodę `delete()` dostawcy treści, korzystając z obiektu klasy `ContentResolver()`. URI przekazywane do metody `delete()` zawiera identyfikator kasowanego wiersza. Jest ono tworzone za pomocą `withAppendedId()` z klasy `ContentUris`.

Ćwiczenie 3.7 (do samodzielnego wykonania)

Korzystając z informacji zawartych w punkcie 3.2.7 dodaj obsługę kasowania wybranych telefonów z listy. Pamiętaj o utworzeniu pliku XML opisującego pasek kontekstowy. Jest to plik tworzony analogicznie do plików opisujących „zwykły" pasek akcji. Plik nazwij `pasek_kontekstowy_listy.xml`. Identyfikatorem akcji powinno być: `kasujMenu`. Wygląd paska kontekstowego przedstawia rysunek 3.7.



Rys. 3.7. Zaznaczanie elementów listy i pasek kontekstowy

3.2.8 Wywoływanie i przekazywanie danych do innych aplikacji

Zajmiemy się teraz zagadnieniem niezwiązanym bezpośrednio z tematem dostawców treści i baz danych lecz pasującym do naszej aplikacji a w szczególności do danych, które przechowujemy w bazie. Chodzi tutaj o adres strony opisującej dany model telefonu. Czasami w systemie istnieje aplikacja robiąca dokładnie to czego potrzebujemy np. wyświetla obrazek, wysyła plik, wysyła e-mail, czy jak w naszym wypadku wyświetla strony WWW. Implementowanie tej funkcji w naszej aplikacji byłoby niepotrzebną komplikacją i dublowaniem funkcjonalności. Zwiększałoby to też niepotrzebnie rozmiar aplikacji. Android oferuje standardowy sposób wywoływania i przekazywania danych do innej aplikacji. Mogą to być dane różnych typów np. nazwa pliku do wysłania, nazwa obrazka do wyświetlenia czy adres do otworzenia. Do określenia działania, które chcemy wykonać służą *akcje*. My użyjemy akcji `VIEW` (ich listę można znaleźć na stronie [4]). Inną często używaną akcją jest `SEND`. Akcje definiuje się w intencji, którą następnie się uruchamia. Android automatycznie wybiera i prezentuje użytkownikowi listę aplikacji, które potrafią obsłużyć daną akcję. Tak więc w przypadku adresu strony pojawi się lista przeglądarek, a w przypadku wysyłania pojawi się lista aplikacji umożliwiających wysyłanie – nie jest istotne czy jest to wysyłanie pliku jako załącznika e-maila czy wysyłanie pliku przez bluetooth.

Tak jak wspomniano w naszej aplikacji użyjemy akcji `VIEW` do uruchomienia przeglądarki WWW. Konieczne jest zatem przekazanie adresu strony z naszej aplikacji do wywoływanej przeglądarki.

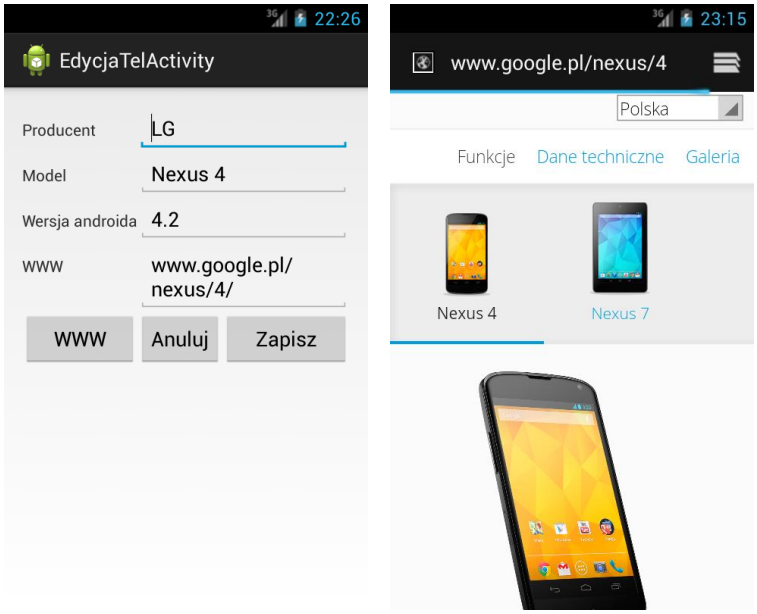
Listing 3.13. Metoda pomocnicza obsługująca kliknięcie przycisku WWW

```
private void kliknieciePrzyciskuWWW() {
  if (!mWWWEdycja.getText().toString().equals("")) {
    String adres = mWWWEdycja.getText().toString();
    if (!adres.startsWith("http://") &&
        !adres.startsWith("https://"))
      adres = "http://" + adres;
    Intent zamiarPrzegladarki = new Intent(
      "android.intent.action.VIEW", Uri.parse(adres));
    startActivity(zamiarPrzegladarki);
  } else {
    Toast.makeText(this,
      getString(R.string.wypelnij_pola_komunikat),
    Toast.LENGTH_SHORT).show();
  }
}
```

Listing 3.13 przedstawia metodę pomocniczą obsługującą kliknięcie przycisku WWW. Rozpoczyna się ona od sprawdzenia czy w polu tekstowym WWW wpisano cokolwiek. Jeżeli tak to sprawdzany jest początek napisu, czy rozpoczyna się od „`http://`” lub „`https://`”. Jeżeli nie to doklejane jest „`http://`” na początku łańcucha. Następnie tworzona jest nowa intencja. Warto zwrócić uwagę, że w przeciwieństwie do innych miejsc, w których w naszej aplikacji korzystamy z intencji, tutaj nie podajemy konkretnej nazwy klasy do uruchomienia, lecz tak jak już wspomniano określamy akcję do wykonania. Jest nią `android.intent.action.VIEW`. Ważne jest, aby podawać akcję razem z przestrzenią nazw (w przykładzie przestrzeń nazw to `android.intent.action`). Drugim parametrem konstruktora intencji jest jednorodny identyfikator zasobu czyli URI. Identyfikator ten tworzymy za pomocą statycznej metody `parse()` z klasy `Uri`. Po utworzeniu intencji uruchamiamy ją za pomocą standardowego wywołania `startActivity()`.

Ćwiczenie 3.8 (do samodzielnego wykonania)

Korzystając z punktu 3.2.8 dodaj obsługę przycisku WWW w aktywności `EdycjaTelActivity`. Sprawdź działanie aplikacji. Czy można powrócić z przeglądarki WWW do naszej aplikacji za pomocą przycisku *wstecz*? Wynik działania aplikacji przedstawiono na rysunku 3.8.



Rys. 3.8. Uruchamianie innej aplikacji

3.3. Korzystanie z ustawień współdzielonych

Przejdziemy teraz do drugiego sposobu trwałego przechowywania danych, jakim są *ustawienia współdzielone*. Nie umożliwiają one składowania skomplikowanych struktur danych, ale nadają się doskonale do przechowywania wszelkich ustawień aplikacji. Może to być np. nazwa użytkownika w jakiejś usłudze, ustawienia synchronizacji danych czy preferencje odnośnie wyglądu aplikacji. W ustawieniach współdzielonych można przechowywać proste ustawienia, które da się wyrazić za pomocą napisów, liczb czy wartości logicznych. Są one przechowywane jako pary: tekstowy klucz oraz wartość. Ich zaletą jest to, że ustawianie/odczytywanie wartości nie wymaga tworzenia żadnych rozbudowanych klas. Dostępne są dla wszystkich komponentów aplikacji.

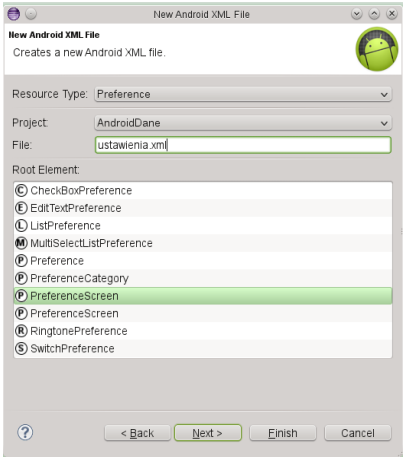
Oprócz samych ustawień współdzielonych zajmiemy się też ich modyfikacją.

W większości aplikacji należy udostępnić użytkownikowi możliwość edycji przynajmniej części ustawień przechowywanych przez naszą aplikację. Oznacza to konieczność utworzenia nowej aktywności, która umożliwi zmienianie ustawień. Na szczęście Android oferuje aktywności `PreferenceActivity` oraz `PreferenceFragment` pozwalające na łatwe uzupełnienie aplikacji o taką funkcję.

Przechowywanie ustawień sprawdzimy, dodając do naszej aplikacji możliwość ręcznego lub automatycznego eksportowania danych z bazy do pliku CSV. Naszymi ustawieniami będą: nazwa pliku, do którego dane są eksportowane oraz informacja, czy eksport ma następować automatycznie. Samym eksportem zajmiemy się w następnym punkcie, w którym omówimy podstawy pracy z plikami w pamięci masowej.

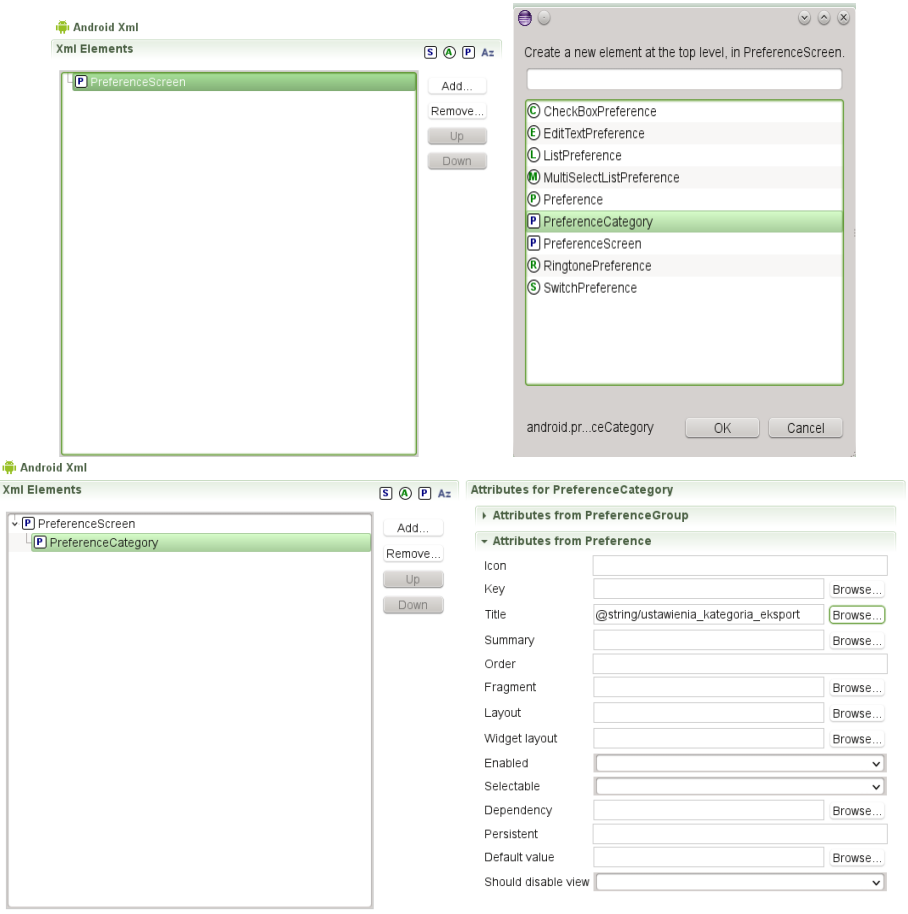
3.3.1 Tworzenie aktywności do modyfikacji ustawień

Rozpoczniemy od utworzenia aktywności umożliwiającej edycję ustawień. Najpierw musimy dodać, jak to zazwyczaj bywa w przypadku aplikacji dla Androida, plik XML opisujący ustawienia naszej aplikacji. Będzie to plik `ustawienia.xml` w katalogu `res/xml` naszego projektu. Zatem *w menu kontekstowym projektu* wybieramy *new* | *other* | *Android XML file*. W kreatorze (rysunek 3.9) zmieniamy *typ zasobu* (*ang. resource type*) na: *ustawienia* (*ang. preference*). Jako nazwę pliku ustalamy wspomniane: *ustawienia.xml*, a jako *element główny* (*ang. root element*) wybieramy: *ekran ustawień* (*ang. PreferenceScreen*). Kończymy kreator wybierając *finish*.



Rys. 3.9. Kreator ekranu ustawień

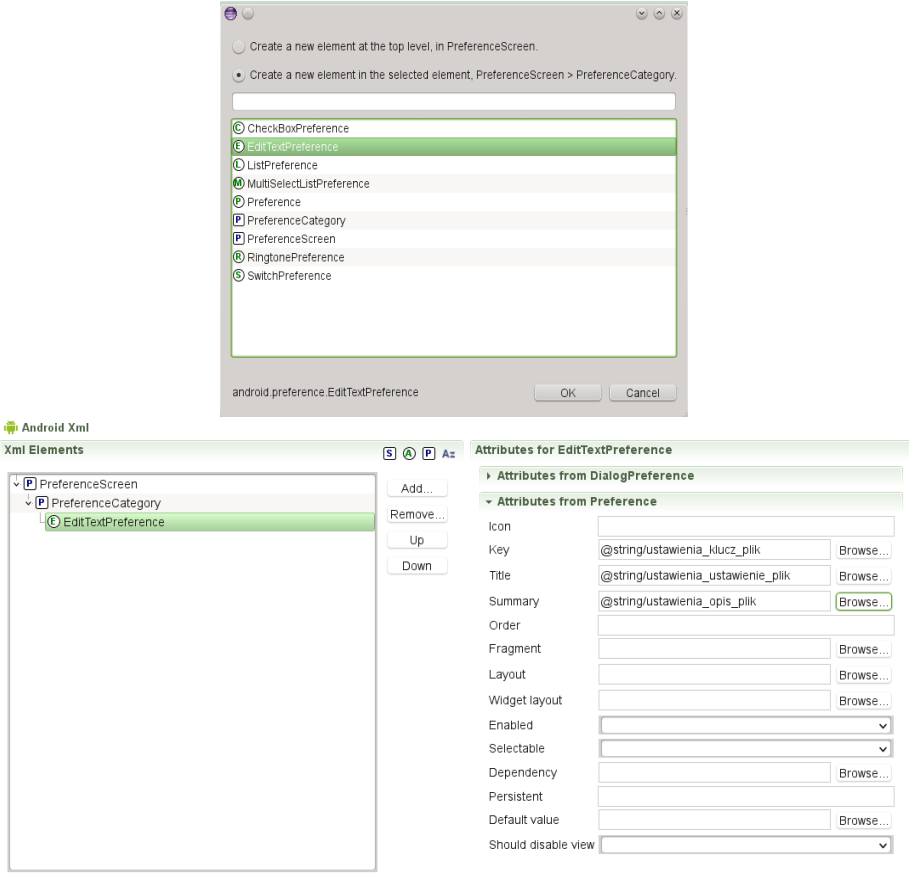
Po utworzeniu pliku XML przechodzimy do jego edycji. Po dwukrotnym kliknięciu pliku powinien uruchomić się edytor przedstawiony na rysunku 3.10. Początkowo nasz plik zawiera tylko element *PreferenceScreen*. Ustawienia w Androidzie pogrupowane są w kategorie. Każda kategoria ma na ekranie ustawień własny nagłówek. Ponieważ w naszej aplikacji nie znajduje się zbyt wiele ustawień, dodamy tylko jedną kategorię. W tym celu w edytorze wybieramy element *PreferenceScreen*, a następnie klikamy przycisk *Add*. W pojawiającym się oknie zaznaczamy *kategorię ustawień* (*ang. preference category*) i zatwierdzamy przyciskiem *Ok*. Następnie, w edytorze wybieramy nowy element i ustawiamy jego tytuł (oczywiście używamy odwołania do łańcucha w pliku `strings.xml`)



Rys. 3.10. Dodawanie kategorii ekranu ustawień

Gdy mamy już potrzebną kategorię, przystępujemy do dodawania poszczególnych opcji. Zajmiemy się teraz pierwszą z nich – czyli ustawieniem przechowującym nazwę pliku potrzebną do wyeksportowania danych. Wybieramy kategorię, do której chcemy dodać ustawienie i akceptujemy przyciskiem *Add*. Pojawi się okno jak na rysunku 3.11. Korzystamy w nim z ustawienia wykorzystującego *pole tekstowe* (*EditTextPreference*) i zatwierdzamy za pomocą *Ok*. Następnie przechodzimy do modyfikowania właściwości ustawienia. Wartości nadamy *kluczowi* (*ang. key*), *tytułowi* (*ang. title*) i *opisowi* (*ang. summary*). We wszystkich wypadkach wykorzystamy odwołania do pliku `strings.xml`. Są to:

- `ustawienia_klucz_plik`,
- `ustawienia_ustawienie_plik`,
- `ustawienia_opis_plik`.



Rys. 3.11. Dodawanie ustawień w określonej kategorii

Listing 3.14 przedstawia plik XML opisujący dodane przez nas: kategorię i ustawienie.

Listing 3.14. Utworzony plik `res/xml/ustawienia.xml`

```
<?xml version="1.0" encoding="utf-8"?>
<PreferenceScreen
xmlns:android="http://schemas.android.com/apk/res/android" >

    <PreferenceCategory
      android:title="@string/ustawienia_kategoria_eksport" >
        <EditTextPreference
            android:key="@string/ustawienia_klucz_plik"
            android:summary="@string/ustawienia_opis_plik"
            android:title="@string/ustawienia_ustawienie_plik"
        />
    </PreferenceCategory>

</PreferenceScreen>
```

Ćwiczenie 3.9 (do samodzielnego wykonania)

Utwórz plik `ustawienia.xml`, tak jak to opisano w punkcie 3.3.1, a następnie dodaj nowe ustawienie typu `CheckBoxPreference`. Ustaw klucz, tytuł i opis nowego ustawienia. Wykorzystaj plik `strings.xml`. Jako identyfikatory poszczególnych właściwości wykorzystaj odpowiednio: `ustawienia_klucz_auto`, `ustawienia_ustawienie_auto` oraz `ustawienia_opis_auto`.

3.3.2 Tworzenie aktywności ustawień

Po dodaniu pliku `ustawienia.xml` utworzymy wykorzystującą go aktywność. W menu kontekstowym katalogu `src` wybieramy *New* | *Class*. W kreatorze nowej klasy ustawiamy nazwę nowej aktywności na: `UstawieniaActivity` oraz klasę nadrzędną na: `android.preference.PreferenceActivity`.

Listing 3.15. Aktywność `UstawieniaActivity`

```
public class UstawieniaActivity extends PreferenceActivity {

      public static class UstawieniaFragment extends
          PreferenceFragment {
```

```
    @Override
    public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
        addPreferencesFromResource(R.xml.ustawienia);
    }
  }

  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    getFragmentManager().beginTransaction().
      replace(android.R.id.content,
              new UstawieniaFragment()).commit();
  }

}
```

Do utworzonego szkieletu aktywności musimy wprowadzić elementy pokazane na listingu 3.15. Jak widać, nie są one zbyt skomplikowanie. Pierwszym z tych elementów jest klasa wewnętrzna `UstawieniaFragment` dziedzicząca po `PreferenceFragment`. W klasie `UstawieniaFragment` przeciążamy tylko metodę `onCreate`, w której oprócz wywołania `onCreate()` z klasy nadrzędnej, ustawiamy jedynie opis ustawień z pliku XML za pomocą `addPreferencesFromResource()`. Podobnie postępujemy w przypadku metody `onCreate()` aktywności `UstawieniaActivity`. W tej metodzie jednak programowo zastępujemy zawartość aktywności fragmentem `UstawieniaFragment`. Ważne jest, aby pamiętać o dodaniu nowej aktywności do manifestu aplikacji.

Warto sobie w tym miejscu zadać pytanie – jakie są korzyści z zastosowania aktywności pochodnej `PreferenceActivity`? Są one następujące:

- jak widać nie musieliśmy samodzielnie tworzyć układu XML. Zostanie on wygenerowany automatycznie i będzie podobny do ekranów ustawień z innych aplikacji, do których użytkownik jest przyzwyczajony,
- podstawowy szkielet aktywności ustawień jest standardowy i może być wykorzystany w każdej aplikacji bez przeróbek (o ile nie są wymagane zaawansowane sposoby prezentacji ustawień czy reagowanie na zdarzenia),
- wszystkie zdefiniowane w pliku XML ustawienia zachowywane są automatycznie, wykorzystywana jest zdefiniowana dla danego ustawienia wartość klucza.

Zamiast sposobu tworzenia aktywności ustawień można użyć *kreatora Settings Activity*. Generuje on jednak dość dużo kodu, który na dodatek używa API oznaczonego przez Google jako przestarzałe. Ponadto na urządzeniach z ekranami mniejszymi niż 10 cali efekt działania kodu z kreatora jest taki sam jak naszego.

Ćwiczenie 3.10 (do samodzielnego wykonania)

Stwórz aktywność `UstawieniaActivity`, tak jak to opisano w punkcie 3.3.2. Pamiętaj o dodaniu nowej aktywności do manifestu aplikacji. Dodaj opcję do paska akcji pozwalającą na uruchomienie aktywności ustawień (identyfikator akcji: `ustawieniaMenu`). Dodaj odpowiednią ikonkę (identyfikator: `ic_ustawienia`). Do obsługi opcji uruchamiającej aktywność ustawień stwórz metodę pomocniczą `uruchomUstawienia()`.

3.3.3. Korzystanie z ustawień współdzielonych w kodzie programu

Przejdziemy teraz do umieszczenia odwołań do ustawień współdzielonych w kodzie naszej aplikacji. Użyjemy najprostszego sposobu polegającego na zapisywaniu danych w domyślnych ustawieniach współdzielonych. Obiekt klasy `SharedPreferences` uzyskamy za pomocą metody `getDefaultSharedPreferences()` (w razie potrzeby można utworzyć inne zestawy ustawień za pomocą metody aktywności `getSharedPreferences()`). Sposób uzyskania tego obiektu przedstawiono na listingu 3.16.

Listing 3.16. Wykorzystanie z ustawień współdzielonych w `ListaTelActivity`

```
public class ListaTelActivity extends ListActivity implements
             LoaderCallbacks<Cursor>{

      //... - część kodu usunięto

      private SharedPreferences mUstawienia;

      @Override
      protected void onCreate(Bundle savedInstanceState) {
        //... - część kodu usunięto
        mUstawienia =
          PreferenceManager.getDefaultSharedPreferences(this);
        if (mUstawienia.
          getString(getString(R.string.ustawienia_klucz_plik),
                              "").equals("")) {
          Editor edytorUstawien = mUstawienia.edit();
        edytorUstawien.putString(
```

```
            getString(R.string.ustawienia_klucz_plik),
            getString(R.string.ustawienia_domylsny_plik));
          edytorUstawien.commit();
          }
          //... - część kodu usunięto
        }

        //... - część kodu usunięto

        private void menuEksportujDoCSV() {
          //... - część kodu usunięto
          Log.d("LTA", "eksportujDoCSV() - nazwa pliku: " +
                nazwaPliku);
        }

        @Override
        protected void onActivityResult(int requestCode,
                                        int resultCode, Intent data) {
          super.onActivityResult(requestCode, resultCode, data);
          Log.d("LTA", "onActivityResult()");
          getLoaderManager().restartLoader(0, null, this);
          if (resultCode == RESULT_OK) {
            //... - część kodu usunięto
          }
        }
}
```

Po uzyskaniu obiektu ustawień `mUstawienia` odczytujemy z niego za pomocą metody `getString()` nazwę pliku, do którego będziemy eksportować dane. Metod `getNazwaTypu()` jest w klasie `SharedPreferences` wiele. Ich argumentami są klucz (wartość tekstowa) wykorzystany do zapisania wartości oraz wartość domyślna używana w wypadku, gdyby dane ustawienie nie zostało wcześniej zachowane. Warto tu przypomnieć, że wartości klucza zdefiniowaliśmy w pliku `strings.xml` – stąd odwołanie do zasobów `getString(R.string....)`. Wartością domyślną jest łańcuch pusty. Jeżeli z ustawień zostanie odczytany łańcuch pusty, to przystępujemy do zapisania wartości domyślnej, jaką jest „`lista_telefonow.csv`” (znów zdefiniowana w `strings.xml`). Robimy to z dwóch powodów: po pierwsze aby przećwiczyć zachowywanie wartości z poziomu kodu programu, po drugie aby użytkownikowi po otworzeniu aktywności ustawień była od razu prezentowana domyślna wartość.

Zapisywanie wartości przebiega następująco: za pomocą obiektu `mUstawienia` uzyskujemy dostęp do edytora – czyli obiektu klasy `Editor`. Następnie zapisujemy wartość za pomocą metody `putString()` (istnieją

oczywiście metody pozwalające na zapisywanie danych innych typów). Jej parametrem są: klucz i wartość. Po wprowadzeniu wartości zatwierdzamy wprowadzone zmiany, wywołując metodę `commit()` (możemy zapisać kilka wartości po kolei, a dopiero potem zatwierdzić wszystkie zmiany).

Ćwiczenie 3.11

Wprowadź w naszej przykładowej aplikacji zmiany opisane w punkcie 3.3.3. Dodatkowo:

- dodaj opcję na pasku zadań pozwalającą na uruchomienie eksportu danych (identyfikator akcji: `eksportMenu`, identyfikator ikonki: `ic_eksport`). Obsługa wybrania opcji powinna korzystać z metody pomocniczej `menuEksportujDoCSV()`
- uzupełnij metodę `menuEksportujDoCSV()`, tak aby odczytywała nazwę pliku zapisaną w ustawieniach i wyświetlała ją na wyjściu debugowania (*LogCat*) za pomocą:
  ```
  Log.d("LTA", "menuEksportujDoCSV() - nazwa pliku: " +
        nazwaPliku);
  ```
  (eksportowaniem do pliku zajmiemy się w punkcie 3.4)
- uzupełnij metodę `onActivityResult()`, tak aby sprawdzana była wartość ustawienia dotyczącego automatycznego eksportu do pliku. Jeżeli wynik zakończonej aktywności to `RESULT_OK` i włączony jest automatyczny eksport, to powinna być wywoływana metoda `menuEksportujDoCSV()`.

Sprawdź działanie aplikacji.

3.4 Zapisywanie/odczytywanie danych z pliku

Ostatnim sposobem przechowywania danych jakim się zajmiemy, jest korzystanie z plików w pamięci masowej urządzenia. Tak jak już zasygnalizowano w poprzednim punkcie, zrealizujemy w naszej aplikacji funkcję eksportu danych z bazy do pliku CSV. Dodatkowo aplikację uzupełnimy funkcją importu danych z pliku.

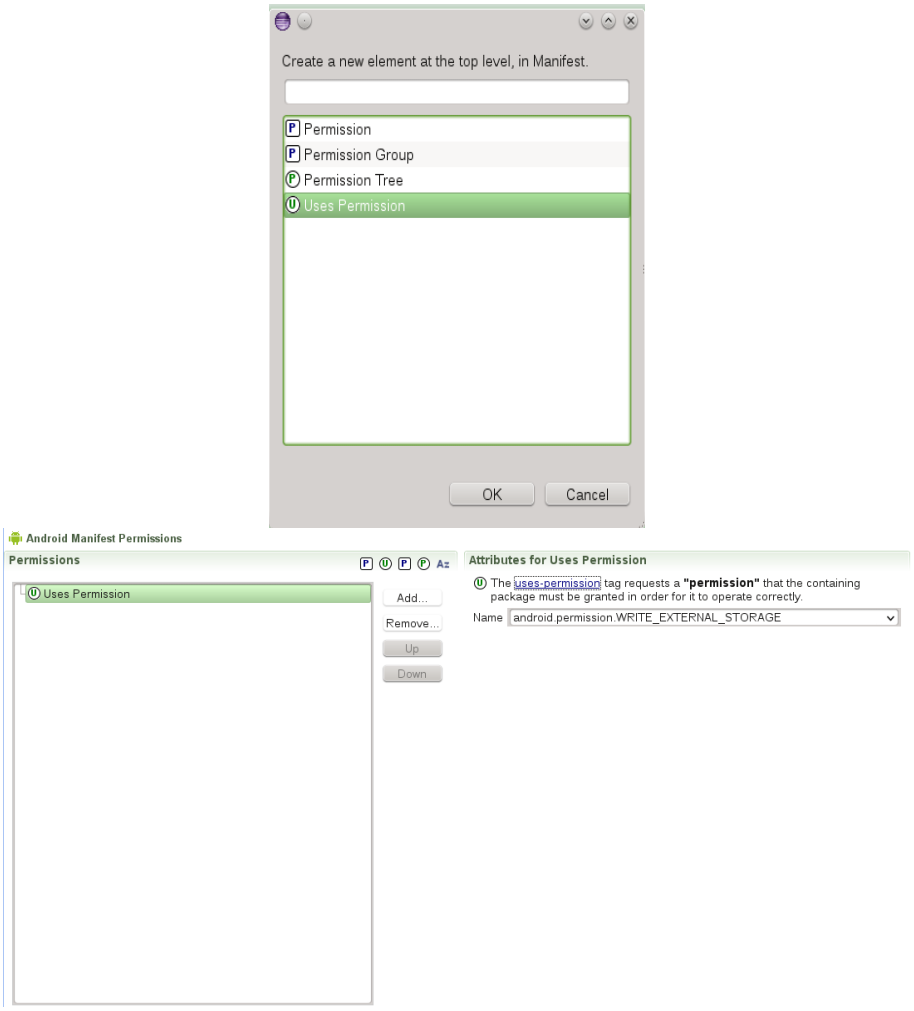
Zanim jednak przejdziemy do korzystania z plików, musimy zająć się zagadnieniem z tym związanym, jakim są uprawnienia aplikacji.

3.4.1 System uprawnień aplikacji

Android kontroluje aplikacje na wiele sposobów. Pakiety `apk` zawierające aplikacje muszą być cyfrowo podpisane (w przeciwnym razie nie zostaną uruchomione). Każda aplikacja ma własne konto użytkownika, które wykorzystuje do działania. Jest to ciekawe rozwiązanie, ponieważ w typowym systemie działającym na standardowym komputerze aplikacje uruchamiane

przez jednego użytkownika działają, wykorzystując jego pojedyncze konto. W Androidzie aplikacje są rozdzielone m.in. za pomocą różnych kont.

Kolejnym elementem pozwalającym na kontrolę aplikacji jest system uprawnień. Do tej pory tworzyliśmy aplikacje, które nie wymagały żadnych ponadstandardowych uprawnień. Wynikało to z faktu, iż wszystkie dane zapisywane były w prywatnej przestrzeni aplikacji odizolowanej od innych. Teraz jednak będziemy zapisywać i odczytywać pliki z karty SD (lub obszaru pamięci urządzenia, który ją emuluje). Dostęp do karty mają wszystkie aplikacje. Tak więc użytkownik instalując naszą aplikację, będzie musiał zezwolić jej na odczyt/zapis na karcie SD. Uprawnienia, których potrzebuje nasza aplikacja to `READ_EXTERNAL_STORAGE` i `WRITE_EXTERNAL_STORAGE`. Istnieją oczywiście inne uprawnienia takie jak: odczytywanie zgrubnej lokalizacji, odczytywanie dokładnej lokalizacji czy dostęp do aparatu. Ich pełną listę można znaleźć na stronie [5].



Rys. 3.12. Dodawanie uprawnień wykorzystywanych przez aplikację uprawnień

Uprawnienia, których wymaga aplikacja, określa się w manifeście aplikacji. Przejdziemy zatem to edytora manifestu, a następnie do zakładki *Permissions*. W edytorze (rys. 3.12) wybieramy przycisk *Add*, a w oknie, które się pojawi, wybieramy *Uses Permission*. Ważne jest to, żeby nie pomylić elementów *Uses Permission* i *Permission*, ponieważ znacznik `Permission` służy do deklarowania własnych uprawnień, których potrzebują inne aplikacje, aby korzystać z naszej. Natomiast `Uses Permission` określa uprawnienia potrzebne naszej aplikacji. Po dodaniu elementu `Uses Permission` ustawiamy jego atrybut `name` na

`android.permission.WRITE_EXTERNAL_STORAGE`. W manifeście powinien pojawić się nowy element (listing 3.17).

Listing 3.17. Fragment manifestu określający wymagane przez aplikację uprawnienia

```
<?xml version="1.0" encoding="utf-8"?>
<manifest
xmlns:android="http://schemas.android.com/apk/res/android"
    package="pl.pollub.android_dane"
    android:versionCode="1"
    android:versionName="1.0" >
    <uses-sdk
        android:minSdkVersion="17"
        android:targetSdkVersion="17" />

    <uses-permission
android:name="android.permission.WRITE_EXTERNAL_STORAGE" />

    <application
        <!-- część kodu usunięto -->
</manifest>
```

Warto zauważyć, że zażądaliśmy w manifeście tylko uprawnienia `WRITE_EXTERNAL_STORAGE`. Wynika to z faktu, że aplikacja, która uzyska uprawnienie do zapisu na karcie SD, automatycznie otrzymuje prawo do odczytu jej zawartości.

Na koniec warto jeszcze wyjaśnić, co się stanie, gdy aplikacja będzie próbowała wykonać operację, do której nie ma uprawnień. Odpowiedź jest dość krótka – na etapie tworzenia/kompilacji – nic, w momencie próby wykonania tej operacji – zostanie zgłoszony wyjątek.

3.4.2 Zapis do pliku

Zapis do pliku w Androidzie jest dość prosty i odbywa się w sposób typowy dla języka Java. Jedynym charakterystycznym elementem API Androida, który wykorzystamy jest metoda `getExternalStorageDirectory()` klasy `Environment`.

Listing 3.18. Zapis danych do pliku

```
private void menuEksportujDoCSV() {
  String nazwaPliku = //... - część kodu usunięto
  File katalogSD =
    Environment.getExternalStorageDirectory();
    File plik = new File(katalogSD + File.separator +
      nazwaPliku);
  BufferedWriter pisarz = null;
  try {
    pisarz = new BufferedWriter(new FileWriter(plik));
    String projekcja[] = { PomocnikBD.PRODUCENT,
      PomocnikBD.MODEL, PomocnikBD.ANDROID,
      PomocnikBD.WWW };
    Cursor kursorTel = getContentResolver().query(
      TelefonyProvider.URI_ZAWARTOSCI, projekcja,
      null, null, null);
    int indexProd =
      kursorTel.getColumnIndex(PomocnikBD.PRODUCENT);
    //... - część kodu usunięto
    String sepWar = ",";

    kursorTel.moveToFirst();
    while (!kursorTel.isAfterLast()) {
      pisarz.write(kursorTel.getString(indexProd) +
      sepWar);
      //... - część kodu usunięto
      kursorTel.moveToNext();
    }
    kursorTel.close();
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if (pisarz != null) {
      try {
        pisarz.close();
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
  }
}
```

Listing 3.18 przedstawia fragmenty metody `menuEksportujDoCSV()`, której zadaniem jest zapisanie danych do pliku. Metoda rozpoczyna od odczytania

nazwy pliku z ustawień współdzielonych. Następnie używa `getExternalStorageDirectory()` w celu uzyskania lokalizacji do karty SD (w zależności od wersji Androida i konkretnego urządzenia ta ścieżka może być różna). Dalej tworzony jest obiekt klasy `File` reprezentujący plik, do którego eksportujemy dane. Następnie tworzymy obiekt pisarz klasy `BufferedWriter` połączony z obiektem klasy `FileWriter`, który będzie odpowiedzialny za zapis do naszego pliku. Następnie za pomocą dostawcy treści odczytujemy dane z bazy. Otrzymujemy je w postaci kursora. Przeglądamy kursor i poszczególne wartości zapisujemy za pomocą metody pisarza – `write()`. Do kolejnych wartości dodajemy przecinek (ponieważ eksportujemy do pliku CSV). Do ostatniej wartości w wierszu dodajemy znak nowej linii. Po zakończeniu eksportu zamykamy kursor i pisarza.

Ćwiczenie 3.12 (do samodzielnego wykonania)

Wprowadź modyfikacje metody `menuEksportujDoCSV()` przedstawione w punkcie 3.4.2. Uzupełnij usunięte fragmenty. Sprawdź działanie aplikacji. Przykładowy wynik przedstawiono na listingu 3.19.

Listing 3.19. Przykładowy wynik eksportu

```
LG,Nexus 4,4.2,www.google.pl/nexus/4/
Samsung,Galaxy S4,4.2,www.samsung.com/global/microsite/galaxys4/
Sony,Xperia Z,4.2,www.sonymobile.com/pl/products/phones/xperia-z
```

3.4.3. Tworzenie okien dialogowych

Zanim przejdziemy do wczytywania danych z pliku i ich importowania do bazy, będziemy potrzebowali nazwy pliku do zaimportowania. Prostym sposobem wskazania pliku jest podanie jego nazwy w oknie dialogowym.

Zgodnie z nowym i zalecanym sposobem tworzenia okien dialogowych w Androidzie utworzymy klasę, która dziedziczy po klasie `DialogFragment`. Klasa `DialogImportFragment` będzie reprezentowała nasz dialog pozwalający na wprowadzenie nazwy pliku. Najważniejsze elementy nowego fragmentu przedstawiono na listingu 3.20.

Listing 3.20. Fragment realizujący okno dialogowe

```
public class DialogImportFragment extends DialogFragment {

    public interface DialogImportListener {
      public void plikWprowadzonyWDialogu(
        DialogFragment dialog);
    }

    private DialogImportListener mListener;

    @Override
    public void onAttach(Activity activity) {
      //... - część kodu usunięto
    }

    @Override
    public void onDetach() {
      //... - część kodu usunięto
    }

    @Override
    public Dialog onCreateDialog(Bundle savedInstanceState) {
      AlertDialog.Builder budowniczy =
        new AlertDialog.Builder(getActivity());
      LayoutInflater pompka =
        getActivity().getLayoutInflater();

      budowniczy.setView(
        pompka.inflate(R.layout.dialog_import_pliku, null))
        .setTitle(R.string.dialog_tytul)
        .setPositiveButton(R.string.dialog_przycisk_importuj,
        new DialogInterface.OnClickListener() {
          @Override
          public void onClick(DialogInterface dialog,
                              int id) {
            mListener.plikWprowadzonyWDialogu(
              DialogImportFragment.this);
          }
      }).setNegativeButton(R.string.dialog_przycisk_anuluj,
         new DialogInterface.OnClickListener() {
           public void onClick(DialogInterface dialog, int id)
           {
             DialogImportFragment.this.getDialog().cancel();
           }
      });
```

```
        return budowniczy.create();
    }
}
```

Jak widać na listingu 3.20 `DialogImportFragment` rozpoczyna się od elementów typowych dla fragmentów tzn.: interfejsu `DialogImportListener`, pola `mListener` wskazującego na aktywność implementującą ten interfejs, oraz dwóch metod `onAttach()` i `onDetach()` modyfikujących wartość pola `mListnener`.

Najbardziej istotnym fragmentem listingu 3.20 jest metoda `onCreateDialog()`, ponieważ to ona jest odpowiedzialna za skonstruowanie dialogu. Na początku korzystamy ze statycznej metody `Builder` klasy `AlertDialog` do uzyskania obiektu budowniczego, z pomocą którego stworzymy dialog. Drugim obiektem pomocniczym jest obiekt pompka klasy `LayoutInflater`. Użyjemy go do utworzenia widoków na podstawie pliku XML układu. Gdy dysponujemy już obiektami pomocniczymi, możemy przystąpić do budowania okna dialogowego. Na początku korzystamy z metody `setView()` budowniczego. Ustawia ona hierarchię widoków jako zawartość centralnej części okna dialogowego. Tę hierarchię wygenerujemy za pomocą metody `inflate()` obiektu pompka. Następnie ustawimy tytuł okna dialogowego za pomocą metody `setTitle()` a w dalszej kolejności opisy i reakcje na kliknięcie przycisku zatwierdzającego (`setPositiveButton()`) i przycisku anulującego (`setNegativeButton()`). Za obsługę kliknięcia przycisków odpowiada interfejs `DialogInterface.OnClickListener`. W metodzie `onClick()` przycisku zatwierdzającego korzystamy z pola `mListener` i metody `plikWprowadzonyWDialogu()` do powiadomienia aktywności o zdarzeniu. W metodzie `onClick()` przycisku anulującego uzyskujemy dostęp do dialogu, a następnie wywołujemy jego standardową metodę `cancel()`. Warto w tym momencie zwrócić uwagę na to, że metody układają się w łańcuch. Zamiast wywołań

```
budowniczy.setView(...);
budowniczy.setTitle(...);
...
```

mamy

```
budowniczy.setView(...).setTitle(...)...
```

Jest to możliwe, ponieważ każda z tych metod zwraca obiekt klasy `AlertDialog.Builder`. Metoda `onCreateDialog()` zwraca wynik działania budowniczego uzyskany z jego metody `create()`.

Listing 3.21. Przekazywanie informacji z dialogu do aktywności

```
public class ListaTelActivity extends ListActivity implements
             LoaderCallbacks<Cursor>, DialogImportListener {

      //... - część kodu usunięto

      @Override
      public void plikWprowadzonyWDialogu(DialogFragment dialog)
      {
        EditText nazwaImportowanegoEdycja =
          (EditText) dialog.getDialog()
          .findViewById(R.id.nazwaImportowanegoEdycja);
        String nazwaPliku =
          nazwaImportowanegoEdycja.getText().toString();
        Log.d("LTA", "plikWprowadzonyWDialogu() " + nazwaPliku);
      }
}
```

Listing 3.21 przedstawia modyfikację, którą należy wprowadzić w aktywności `ListaTelActivity`. Jest nią zaimplementowanie interfejsu `DialogImportListener` zdefiniowanego wewnątrz naszego fragmentu `DialogImportFragment`. Warto zwrócić uwagę na to, jak można odczytać wartość pola tekstowego znajdującego się nie w aktywności lecz w oknie dialogowym. Najpierw z fragmentu dialogu odczytujemy właściwy dialog (za pomocą `getDialog()`) i wyszukujemy w nim pole tekstowe o identyfikatorze `nazwaImportowanegoEdycja`. Z pola tekstowego pobieramy napis w standardowy sposób – za pomocą wywołań `getText().toString()`. Oczywiście można było też zdefiniować odpowiednią metodę w `DialogImportFragment`.

Listing 3.22. Pokazywanie okna dialogowego

```
      public final static String DIALOG_IMPORT =
        "dialog import";
      private void menuImportujZCSV() {
        DialogFragment fragmentDialoguImportu =
```

```
            new DialogImportFragment();
        fragmentDialoguImportu.show(getFragmentManager(),
                                    DIALOG_IMPORT);
    }
```

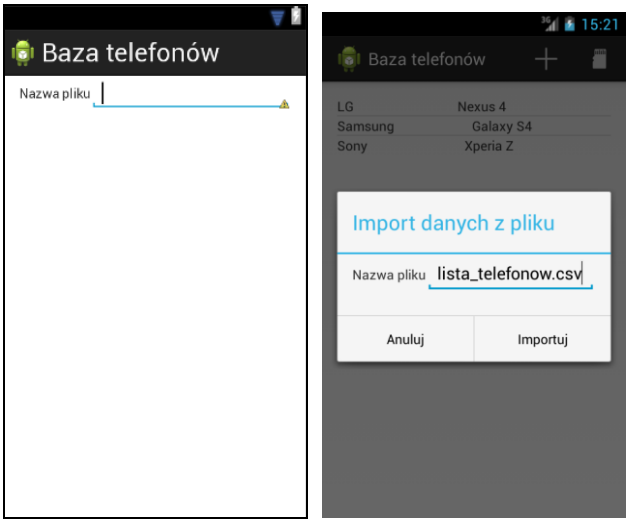
Ostatnią kwestią jaką musimy się zająć, jest wyświetlanie okna dialogowego. Na listingu 3.22 znajduje się metoda pomocnicza, która wyświetla okno dialogowe. Jest ona bardzo prosta. Najpierw tworzony jest obiekt fragmentu, a następnie nowy fragment jest pokazywany za pomocą metody `show()`. Drugi argument tej metody to etykieta, która będzie identyfikowała fragment w menedżerze fragmentów.

Ćwiczenie 3.13 (do samodzielnego wykonania)

Dodaj dialog pozwalający na wprowadzenie nazwy importowanego pliku, korzystając z modyfikacji omówionych w punkcie 3.4.3. W szczególności:

- uzupełnij metody `onAttach()` i `onDetach()` fragmentu,
- stwórz układ dla okna dialogowego taki jak przedstawiony na rysunku 3.12. Dla pola tekstowego użyj identyfikatora `nazwaImportowanegoEdycja`,
- dodaj akcję paska zadań pozwalającą na importowanie danych, stwórz odpowiednią ikonę

Wynik działania aplikacji przedstawiono na rysunku 3.13.



Rys. 3.13. Układ okna dialogowego oraz wygląd okna dialogowego w trakcie pracy aplikacji

3.4.4. Odczyt z pliku

Zajmiemy się teraz ostatnim zagadnieniem tego rozdziału, jakim jest odczyt informacji z pliku. Na listingu 3.23 przedstawiono metodę `plikWprowadzonyWDialogu()` interfejsu `DialogImportListener`. Reaguje ona na wybranie w oknie dialogowym przycisku zatwierdzającego – *Importuj*.

Listing 3.23. Odczytywanie informacji z pliku

```
@Override
public void plikWprowadzonyWDialogu(DialogFragment dialog)
{
  EditText nazwaImportowanegoEdycja =
    (EditText) dialog.getDialog()
    .findViewById(R.id.nazwaImportowanegoEdycja);
  String nazwaPliku =
    nazwaImportowanegoEdycja.getText().toString();

  File katalogSD =
    Environment.getExternalStorageDirectory();
  File plik = new File(katalogSD + File.separator +
    nazwaPliku);

  if (!plik.exists()) {
    Toast.makeText(this, R.string.plik_nie_istnieje,
                   Toast.LENGTH_LONG).show();
  } else {
    BufferedReader czytnik = null;
    try {
      czytnik = new BufferedReader(new FileReader(plik));
      String linia;
      linia = czytnik.readLine();
      while (linia != null) {
        String wartosci[] = linia.split(",");

        //... - część kodu usunięto

        linia = czytnik.readLine();
      }
    } catch (Exception e) {
      e.printStackTrace();
    } finally {
      if (czytnik != null) {
        try {
          czytnik.close();
```

```
            } catch (IOException e) {
              e.printStackTrace();
            }
          }
        }
      }
    }
```

Na początku, podobnie jak w przypadku zapisu, odczytujemy nazwę pliku, lokalizację karty SD i tworzymy obiekt klasy `File` opisujący importowany plik. Następnie upewniamy się, czy plik istnieje. Jeżeli nie, to wyświetlamy odpowiedni komunikat. Jeżeli wszystko jest w porządku rozpoczynamy wczytywanie pliku. Tworzymy obiekt czytnik klasy `BufferedReader` połączony z obiektem `FileReader`, który jest odpowiedzialny za czytanie pliku. Następnie odczytujemy kolejne linie i jeżeli wczytana linia jest różna od `null`, to dzielimy ją za pomocą metody `split()` na części rozdzielone przecinkami. Za pomocą dostawcy treści wstawiamy wczytany rekord do bazy danych. Na koniec zamykamy czytnik metodą `close()`.

Ćwiczenie 3.14 (do samodzielnego wykonania)

Zmodyfikuj metodę `plikWprowadzonyWDialogu()` w aktywności `ListaTelActivity` zgodnie z opisem z punktu 3.4.4. Uzupełnij metodę zapisywanie danych do bazy za pośrednictwem dostawcy treści. Sprawdź działanie aplikacji.

4. Podstawy wielozadaniowości

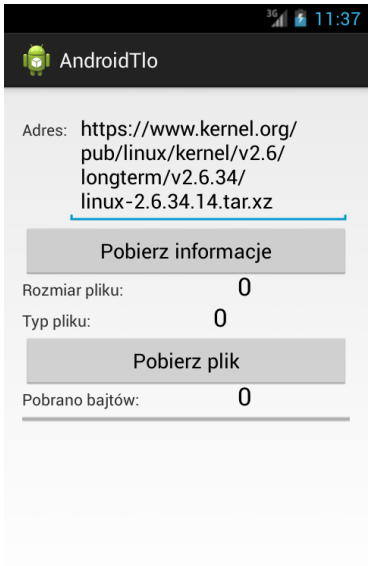
W rozdziale poznamy różne techniki wykonywania zadań w tle. Oprócz standardowych mechanizmów Javy framework Androida dostarcza wielu charakterystycznych dla siebie elementów ułatwiających wykonywanie zadań poza głównym procesem aplikacji. Najważniejsze z nich poznamy na przykładzie aplikacji pozwalającej na pobieranie plików z internetu za pośrednictwem protokołu http. Rozpoczniemy od ćwiczenia, w którym utworzymy kod będący punktem wyjścia do kolejnych rozważań.

Ćwiczenie 4.1 (do samodzielnego wykonania)

Utwórz aktywność o nazwie `PobieranieActivity`. Układ aktywności zapisany w pliku `pobieranie_activity.xml` powinien być podobny do tego z rysunku 4.1. Powinien zawierać następujące elementy:

- pole tekstowe do wprowadzania adresu o identyfikatorze `adresEdycja`, w polu powinien znajdować się domyślny adres. Jego wartość powinna być zapisana w pliku `strings.xml` i posiadać identyfikator `adres_pliku_domyslny`
- przycisk o identyfikatorze `pobierzInfoPrzycisk`,
- etykieta tekstowa o identyfikatorze `rozmiarWartoscEtykieta` i wartości domyślnej zapisanej w pliku `strings.xml`. Wartość domyślna to 0 a jej identyfikator to `rozmiar_pliku_wartosc`,
- etykieta tekstowa o identyfikatorze `typWartoscEtykieta` i wartości domyślnej zapisanej w pliku `strings.xml`. Wartość domyślna to 0 a jej identyfikator to `typ_pliku_wartosc`,
- przycisk o identyfikatorze `pobierzPrzycisk`,
- etykieta tekstowa o identyfikatorze `pobranoBajtowEtykieta` i wartości domyślnej zapisanej w pliku strings.xml. Wartość domyślna to 0 a jej identyfikator to `pobrano_bajtow_wartosc`,
- pasek postępu o identyfikatorze `postepPasek`.

Oprócz wymienionych elementów do aktywności można dodać np. etykiety opisujące poszczególne elementy.



Rys. 4.1. Układ aktywności pobierania plików

4.1. Wykonywanie krótkich zadań w tle – klasa AsyncTask

W Androidzie nie należy wykonywać zadań, długotrwałych w głównym wątku aplikacji. Wynika to z faktu, iż główny wątek aplikacji obsługuje interfejs użytkownika. Gdy wątek ten jest zajęty aplikacja nie reaguje na polecenia użytkownika. Jeżeli wątek główny jest obciążony zbyt długo aplikacja wyświetla tzw. komunikat *ANR* (*ang. application not responding*) informujący o tym, że aplikacja nie odpowiada i proponujący jej zamknięcie. Przykładem operacji, których nie należy wykonywać w wątku głównym są operacje sieciowe, którymi zajmiemy się w tym rozdziale. Do wykonywania zadań w tle poza głównym wątkiem można używać wielu mechanizmów m. in. standardowych wątków Javy. Jednak ciekawszym przykładem będzie klasa `AsyncTask` z frameworku Androida. Zalecane jest stosowanie klasy `AsyncTask` do wykonywania zadań trwających co najwyżej kilka sekund. Do wykonywania dłuższych zadań mogą służyć np. usługi, którymi zajmiemy się później.

Jak już wspomniano klasę `AsyncTask` wykorzystamy wykonywania operacji sieciowych a konkretnie do pobrania informacji o pliku znajdującym się pod wpisanym przez użytkownika adresem. Sposób wykorzystania tej klasy oraz sposób nawiązywania połączenia sieciowego przedstawiono na listingu 4.1.

Listing 4.1. Wykorzystanie klasy `AsyncTask`

```
public class PobieranieActivity extends Activity {

      @Override
      protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.pobieranie_activity);
        //... - obsługa kliknięcia przycisku pobierzInfoPrzycisk
      }

      private class InfoOPliku {
        public int mRozmiar;
        public String mTyp;
      }

      private class PobierzInfoTask extends
        AsyncTask<String, Void, InfoOPliku> {
        @Override
        protected InfoOPliku doInBackground(String... params) {
          HttpURLConnection polaczenie = null;
          InfoOPliku infoOPliku = null;
          try {
            URL url = new URL(params[0]);
            polaczenie =
              (HttpURLConnection) url.openConnection();
            polaczenie.setRequestMethod("GET");
            polaczenie.setDoOutput(true);
            infoOPliku = new InfoOPliku();
            infoOPliku.mRozmiar = polaczenie.getContentLength();
            infoOPliku.mTyp = polaczenie.getContentType();
          } catch (Exception e) {
            e.printStackTrace();
          } finally {
            if (polaczenie != null)
              polaczenie.disconnect();
          }
          return infoOPliku;
        }
        @Override
        protected void onPostExecute(InfoOPliku result) {
          ustawInfoOPliku(result);
          super.onPostExecute(result);
        }
      }
```

```
    private void ustawInfoOPliku(InfoOPliku infoOPliku) {
      //... - ustawianie informacji o pliku w etykietach
    }

    protected void pobierzInfo() {
      TextView adresEdycja =
        (TextView) findViewById(R.id.adresEdycja);
      PobierzInfoTask pobierzInfoTask = new PobierzInfoTask();
      pobierzInfoTask.execute(
        new String[] { adresEdycja.getText().toString() });
    }
}
```

Pierwszym interesującym elementem z listingu 4.1 jest prywatna klasa wewnętrzna `InfoOPliku`, którą utworzymy w aktywności `PobieranieActivity`. Posłuży ona nam do przechowywania informacji dotyczących pobieranego pliku.

Kolejną, również prywatną klasą wewnętrzną jest `PobierzInfoTask` dziedzicząca po klasie `AsyncTask`. Warto zauważyć, że `AsyncTask` jest tylko szablonem klasy, który wymaga podania konkretnych typów w tzw. nawiasach ostrych. W naszym przypadku te typy to: `String`, `Void` oraz `InfoOPliku`. Pierwszy typ (`String`) to typ parametrów przekazywanych do metody `doInBackgound()`. Drugi (`Void`) to typ informacji o postępie wykonywanej czynności, które można przesyłać do głównego wątku za pomocą metody `publishProgress()`. Ponieważ nie będziemy wyświetlać informacji o postępie – wybraliśmy `Void`. Ostatni typ (`InfoOPliku`) to typ wyniku zwracanego przez metodę `doInBackground()`.

W klasie `PobierzInfoTask` zaimplementowaliśmy dwie metody: `doInBackground()` i `onPostExecute()`. Pierwsza z nich jest wykonywana w osobnym wątku i właśnie tam należy umieścić kod operacji, którą chcemy wykonać. W przykładzie operacje wykonywane w osobnym wątku to po kolei:

- utworzenie obiektu reprezentującego URL na podstawie pierwszego łańcucha przekazanego do metody,
- otworzenie połączenia (za pomocą metody `openConnection()` obiektu `url`). Warto zauważyć, że zwrócony obiekt jest rzutowany na typ `HttpURLConnection`,
- ustawienie parametrów połączenia (ustawienie metody „`GET`” oraz umożliwienie wysyłania danych przez połączenie),
- utworzenie obiektu klasy `InfoOPliku` i ustawienie w nim informacji odczytanych z obiektu reprezentującego połączenie,
- zamknięcie połączenia,
- zwrócenie obiektu `infoOPliku`.

Druga metoda `onPostExecute()` jest wywoływana gdy zakończy się `doInBackground()`. Metoda `onPostExecute()` jest wywoływana w głównym wątku aplikacji (w wątku GUI) i dzięki temu możemy bezpiecznie wyświetlić użytkownikowi wyniki operacji wykonanej w tle (GUI nie powinno się aktualizować z innych wątków). Jak widać otrzymuje ona przez parametr wynik działania metody `doInBackground()` – w naszym wypadku jest to obiekt klasy `InfoOPliku` zwrócony przez `doInBackground()`. Jej działanie jest proste i sprowadza się do wywołania metody `ustawInfoOPliku()` z `PobieranieActivity`. Metoda `ustawInfoOPliku()` jak łatwo się domyślić powinna w etykietach `rozmiarWartoscEtykieta` i `rozmiarTypEtykieta` umieścić odpowiednie wartości.

Ostatnia z metod – `pobierzInfo()` – jest metodą pomocniczą, którą wykorzystamy programując obsługę kliknięcia przycisku *Pobierz informacje*. Jak widać jest ona dość prosta i wykonuje 4 proste czynności:

- odczytuje adres URL z pola tekstowego `adresEdycja`,
- tworzy nowy obiekt klasy `pobierzInfoTask`,
- tworzy jednoelementową tablicę łańcuchów, w której umieszcza odczytany URL,
- wywołuje metodę `execute()` klasy `pobierzInfoTask` i przekazuje utworzoną tablicę jako parametr.

Ćwiczenie 4.2 (do samodzielnego wykonania)

Wprowadź do aktywności `PobieranieActivity` modyfikacje z punktu 4.1. Uzupełnij kod o obsługę kliknięcia przycisku `pobierzInfoPrzycisk`. Obsługa kliknięcia przycisku powinna używać metody pomocniczej `pobierzInfo()`. Uzupełnij również metodę `ustawInfoOPliku()`, tak aby ustawiała pobrane wartości w przeznaczonych do tego etykietach. Przykładowy efekt wykonania aplikacji przedstawiono na rysunku 4.2. Sprawdź działanie aplikacji. Sprawdź również, co się stanie po obróceniu urządzenia. Pamiętaj o tym, żeby do manifestu dodać informację o tym, że aplikacja korzysta z dostępu do internetu (element `<uses-permission>` i uprawnienie `INTERNET`).

Rys. 4.2. Pobieranie informacji o pliku

Ćwiczenie 4.3 (do samodzielnego wykonania)

Dodaj do aktywności 4 następujące stałe (będą służyły jako identyfikatory zachowywanych wartości):

```
public final static String ROZMIAR_PLIKU =
  "rozmiar pliku";
public final static String TYP_PLIKU =
  "typ pliku";
public final static String POBRANO_BAJTOW =
  "pobrano bajtow";
public final static String PASEK_POSTEPU =
  "pasek postepu";
```

a następnie dodaj do aktywności metody `onSaveInstanceState()` oraz `onRestoreInstanceState()` (najprościej z menu kontekstowego edytora kodu wybrać *Source* | *Override/Implement Methods*) i uzupełnij je o kod zachowujący wartości:

- etykiety `rozmiarWartoscEtykieta`,

- etykiety `typWartoscEtykieta`,
- etykiety `pobranoBajtowWartoscEtkieta`,
- postępu wyświetlany na pasku postępu (można go odczytać za pomocą metody `getProgress()` a ustawić za pomocą `setProgress()` klasy `ProgressBar`).

Sprawdź działanie aplikacji.

4.2. Proste usługi – dziedziczące po IntentService

Elementem frameworku Android pozwalającym na uzyskanie prawdziwej wielozadaniowości jest klasa `Service` i jej pochodne. Klasa Service reprezentuje chęć aplikacji do wykonywania długotrwałych czynności np. odtwarzania muzyki lub świadczenia usług na rzecz innych aplikacji [7]. Domyślnie usługi dziedziczące po klasie `Service` nie uruchamiają własnych wątków czy procesów i działają w głównym wątku aplikacji. W przypadku operacji, które mogłyby zablokować główny wątek na dłużej (np. operacje sieciowe) usługa powinna uruchomić swój własny wątek.

W tym punkcie zajmiemy się najprostszą z usług mianowicie usługą dziedziczącą po `IntentService`. Pochodne klasy `IntentService` służą do obsługi asynchronicznych żądań zgłaszanych do usługi za pomocą metody `startService()`. Żądania są automatycznie kolejkowane i uruchamiane w osobnym wątku (nie ma potrzeby samodzielnej implementacji tych mechanizmów). Ten sposób obsługi zadań działających w tle dość dobrze pasuje do naszego przykładu – czyli pobierania plików.

4.2.1 Tworzenie usługi

Ponieważ, jak wspomniano, przykład dotyczy pobierania plików rozpoczniemy od utworzenia klasy pomocniczej, w której będziemy mogli przechowywać dane dotyczące postępu pobierania. Klasa `postepInfo` została przedstawiona na listingu 4.2.

Listing 4.2. Klasa pomocnicza z informacjami o postępie pobierania

```
public class PostepInfo
{
	public static final int POBIERANIE_OK=0;
	public static final int POBIERANIE_TRWA=1;
	public static final int POBIERANIE_BLAD=2;


	public int mPobranychBajtow;
```

```
        public int mRozmiar;
        public int mWynik;

        public PostepInfo()
        {
                mPobranychBajtow=0;
                mRozmiar=0;
                mWynik=POBIERANIE_TRWA;
        }
}
```

Kasa `PostepInfo` jest prosta. Zawiera:

- definicje kilku stałych (`POBIERANIE_OK`, `POBIERANIE_TRWA` i `POBIERANIE_BLAD`), które reprezentują wynik operacji pobierania pliku,
- pola przechowujące liczbę pobranych bajtów, rozmiar pliku i wynik pobierania
- konstruktor ustawiający wartości początkowe.

Przejdziemy teraz do tworzenia klasy implementującej usługę. Jej treść przedstawiono na listingu 4.3. Jej głównym elementem jest metoda `onHandleIntent()`. Jest ona uruchamiana w momencie, gdy usługa otrzymuje zadanie do wykonania. Usługi dziedziczące po klasie `IntentService` mogą wykonywać tylko jedno zadanie w danym momencie (zadania są kolejkowane automatycznie).

Listing 4.3. Usługa pobierania pliku

```
public class PobieranieService extends IntentService {

        public PobieranieService() {
          super("usługa pobierania plików");
        }

        public final static String ADRES = "adres pliku";

        public final static int ROZMIAR_BLOKU = 32 * 1024; // 32kB

        private PostepInfo mPostepInfo = new PostepInfo();
        private String mTypPliku;
        String mAdres;
```

```
@Override
protected void onHandleIntent(Intent intent) {
  mAdres = intent.getStringExtra(ADRES);
  Log.d("PS", "adres: " + mAdres);
  HttpURLConnection polaczenie = null;
  InputStream strumienZSieci = null;
  FileOutputStream strumienDoPliku = null;
  try {
    URL url = new URL(mAdres);
    File plikRoboczy = new File(url.getFile());
    File plikWyjsciowy = new File(
      Environment.getExternalStorageDirectory() +
      File.separator+ plikRoboczy.getName());
    Log.d("PS", "plik: " + plikWyjsciowy);
    if (plikWyjsciowy.exists())
      plikWyjsciowy.delete();

    //... - tworzenie połączenia

    mPostepInfo.mPobranychBajtow = 0;
    mPostepInfo.mRozmiar = //... - pobieranie długości
                           //pliku
    mPostepInfo.mWynik = PostepInfo.POBIERANIE_TRWA;
    mTypPliku = //... - pobieranie typu pliku

    DataInputStream czytnik = new DataInputStream(
      polaczenie.getInputStream());
    strumienDoPliku =
      new FileOutputStream(plikWyjsciowy.getPath());

    byte bufor[] = new byte[ROZMIAR_BLOKU];
    int pobrano = czytnik.read(bufor, 0, ROZMIAR_BLOKU);
    while (pobrano != -1) {
      strumienDoPliku.write(bufor, 0, pobrano);
      mPostepInfo.mPobranychBajtow += pobrano;
      pobrano = czytnik.read(bufor, 0, ROZMIAR_BLOKU);
    }
    mPostepInfo.mWynik = PostepInfo.POBIERANIE_OK;
  } catch (Exception e) {
    e.printStackTrace();
    mPostepInfo.mWynik = PostepInfo.POBIERANIE_BLAD;
  } finally {
    if (strumienZSieci != null) {
      try {
        strumienZSieci.close();
      } catch (IOException e) {
```

```
            e.printStackTrace();
          }
        }
        if (strumienDoPliku != null) {
          try {
            strumienDoPliku.close();
          } catch (IOException e) {
            e.printStackTrace();
          }
        }
        if (polaczenie != null)
          polaczenie.disconnect();
        }
      }
}
```

Zajmiemy się teraz budową naszej usługi. Jak widać na listingu 4.3 konstruktor naszej usługi po prostu wywołuje konstruktor klasy nadrzędnej przekazując jako parametr nazwę usługi. W dalszej kolejności deklarujemy stałe: `ADRES` (potrzebną do przekazywania adresu z aktywności do usługi), `ROZMIAR_BLOKU` (określającą w jakich porcjach będziemy pobierać plik). Następnie definiujemy pola klasy przechowujące informacje o postępie pobierania bieżącego pliku (`mPostepInfo`), jego adresie (`mAdres`) i typie (`mTypPliku`).

Większość kodu usługi mieści się w metodzie `onHandleIntent()`. Rozpoczyna się ona od odczytania adresu pobieranego pliku z intencji, którą usługa będzie otrzymywać z aktywności. Dalej ustawiane na `null` są wartości zmiennych reprezentujących strumienie danych (pozwalających na pobieranie z sieci oraz pozwalających na zapis w pliku). Następnie tworzymy zmienne reprezentujące plik docelowy i sprawdzamy czy taki plik już istnieje. Jeżeli tak to jest on kasowany. Wszystko to odbywa się przy wykorzystaniu charakterystycznej dla Javy (nie tylko dla Androida) klasy `File`. Jedynym wyjątkiem jest tu specyficzna dla Androida metoda `getExternalStorageDirectory()` klasy `Environment`. Zwraca ona ścieżkę do karty SD urządzenia (lub w przypadku urządzeń bez slotu kart SD – do katalogu w pamięci wewnętrznej urządzenia pełniącego funkcję pamięci masowej). Nasza aplikacja będzie zapisywać pliki w głównym katalogu karty pamięci. Po sprawdzeniu pliku docelowego otwierane jest połączenie sieciowe w sposób identyczny jak przy sprawdzaniu parametrów pliku (rozmiaru i typu). Po utworzeniu połączenia ustawiane są wartości początkowe pól przechowujących informacje o postępie pobierania pliku i rozpoczyna się tworzenie strumieni. Strumień `czytnik` typu `DataInputStream` jest połączony ze strumieniem danych z połączenia sieciowego. `strumienDoPliku` typu `FileOutputStream` jest połączony z plikiem na karcie SD. Dalej tworzony jest bufor o rozmiarze

ROZMIAR_BLOKU i rozpoczyna się faktyczne pobieranie pliku. Za pomocą metody read() obiektu czytnik usługa próbuje pobrać do bufora porcję danych. Parametry metody read() to:

- bufor, do którego należy pobrać dane,
- numer elementu, od którego należy w buforze zapisywać dane,
- liczba bajtów do pobrania.

Metoda read() zwraca liczbę pobranych bajtów (może ona się różnić od wartości przekazanej za pomocą ostatniego parametru – np. w przypadku gdy rozmiar pobieranego pliku nie jest wielokrotnością wartości ROZMIAR_BLOKU). Gdy nie uda się już nic pobrać – tzn gdy skończył się strumień – metoda zwraca -1. Po pobraniu pierwszej porcji danych metoda w pętli na przemian zapisuje pobrane dane do pliku i próbuje pobrać kolejną porcję danych. Zapisywanie danych do pliku jest realizowane za pomocą metody write() obiektu strumienDoPliku. Parametry metody write() są analogiczne do parametrów read() z tą różnicą, że ostatnim parametrem jest liczba bajtów do zapisania (zapisujemy dokładnie tyle bajtów ile udało się pobrać). Pętla w trakcie działania aktualizuje informacje o liczbie pobranych bajtów zapisanych w obiekcie mPostepInfo. Po zakończeniu pętli wynik pobierania (pole mWynik w obiekcie mPostepInfo) ustawiany jest na zdefiniowaną wcześniej wartość POBIERANIE_OK. Metoda onHandleIntent() kończy się zapewnieniem obsługi sytuacji wyjątkowych, które mogą wystąpić w trakcie pobierania plików oraz sekcją odpowiedzialną za „sprzątanie". W naszym przykładzie stosujemy bardzo proste rozwiązanie kwestii wyjątków polegające na przechwyceniu wszystkich i ustawieniu wyniku pobierania na POBIERANIE_BLAD. W bardziej rozbudowanej aplikacji można byłoby rozróżniać rodzaje wyjątków i zapewnić wyświetlanie użytkownikowi odpowiedniej informacji o problemie (np. błąd połączenia sieciowego czy brak miejsca na karcie SD). „Sprzątanie" polega na tym, że wszystkie strumienie są zamykane.

Gdy dysponujemy już kodem usługi możemy zaprogramować jej uruchomienie. Sposób uruchamiania usługi jest bardzo podobny do sposobu uruchamiania aktywności. Jedyną różnicą jest użycie metody startService() zamiast startActivity(). Listing 4.4 pokazuje sposób uruchamiania usługi w naszej aplikacji.

Listing 4.4. Uruchamianie usługi z poziomu aktywności

```
public class PobieranieActivity extends Activity {

      @Override
      protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.pobieranie_activity);
```

```
        //... - część kodu usunięto

        Button pobierzPrzycisk =
          (Button) findViewById(R.id.pobierzPrzycisk);
        //... - obsługa kliknięcia przycisku pobierzPrzycisk

        //... - część kodu usunięto
      }

      protected void pobierzPlik() {
        pobierzInfo();
        EditText adresEdycja =
          (EditText) findViewById(R.id.adresEdycja);
        Intent zamiarPobierania =
          new Intent(this, PobieranieService.class);
        zamiarPobierania.putExtra(PobieranieService.ADRES,
          adresEdycja.getText().toString());
        startService(zamiarPobierania);
      }
      //... - część kodu usunięto
}
```

Ostatnią czynnością, o której trzeba pamiętać podczas tworzenia usługi jest (podobnie jak w przypadku aktywności) dodanie jej do manifestu aplikacji. Poniżej przedstawiono deklarację usługi w manifeście aplikacji.

```
<application
    <!-- część kodu usunięto -->
    <service android:name="PobieranieService" >
    </service>
</application>
```

Ćwiczenie 4.4 (do samodzielnego wykonania)

Wykonaj modyfikacje przedstawione w punkcie 4.2.1 tj. :

- dodaj do projektu klasę pomocniczą `PostepInfo`,
- dodaj do projektu klasę usługi `PobieranieService`,
- uzupełnij kod usługi w miejscach wskazanych komentarzem,
- dodatkowo korzystając z klasy `Log`, możesz dodać do usługi wyświetlanie informacji o postępach pobierania pliku (na razie będzie to jedyne źródło informacji o pracy usługi),
- dodaj do aktywności `PobieranieActivity` obsługę przycisku

`pobierzPrzycisk`. Obsługa pobieranie powinna korzystać z metody pomocniczej `pobierzPlik()`,
- dodaj do aktywności `PobieranieActivity` metodę pomocniczą `pobierzPlik()`. Metoda uruchamia usługę pobierania pliku,
- do manifestu dodaj deklarację usługi
- dodaj do manifestu informację o tym, że aplikacja oprócz dostępu do internetu wymaga także uprawnień do zapisywania plików na karcie pamięci (element `<uses-permission>` i uprawnienie `WRITE_EXTERNAL_STORAGE`).

Sprawdź działanie aplikacji.

4.2.2 Łatwe przekazywanie obiektów między elementami aplikacji – interfejs Parcelable

Do tej pory w obiektach, które żartobliwie nazywamy tobołkami (obiektami klasy `Bunlde`), przekazywaliśmy typy standardowe takie jak wartości całkowite czy napisy. Czasami jednak wygodnie byłoby przekazać cały obiekt. W tym celu można zaimplementować w obiekcie interfejs `Parcelable`, dzięki któremu możliwe jest proste zapisanie stanu obiektu do tobołka a po drugiej stronie kanału komunikacyjnego utworzenie jego „klona”. Nie jest to dokładnie przekazywanie tej samej instancji obiektu, lecz utworzenie drugiego obiektu znajdującego się w dokładnie takim samym stanie. Implementację interfejsu `Parcelable` na przykładzie klasy `PostepInfo` przedstawiono na listingu 4.5. Przykład wykorzystania tego interfejsu omówimy w następnym punkcie.

Listing 4.5. Implementacja interfejsu `Parcelable`

```
public class PostepInfo implements Parcelable
{
       //... - część kodu usunięto

       public PostepInfo(Parcel dane) {
         mPobranychBajtow=dane.readInt();
         mRozmiar=dane.readInt();
         mWynik=dane.readInt();
       }

       @Override
       public int describeContents() {
         return 0;
       }
       @Override
       public void writeToParcel(Parcel dest, int flags) {
```

```
        dest.writeInt(mPobranychBajtow);
        dest.writeInt(mRozmiar);
        dest.writeInt(mWynik);
    }

    public static final Parcelable.Creator<PostepInfo>
      CREATOR = new Parcelable.Creator<PostepInfo>() {

        @Override
        public PostepInfo createFromParcel(Parcel source) {
          return new PostepInfo(source);
        }

        @Override
        public PostepInfo[] newArray(int size) {
          return new PostepInfo[size];
        }
    };
}
```

Jak widać implementacja interfejsu `Parcelable` wymaga dodania do klasy następujących elementów:

- konstruktora, którego parametrem jest obiekt klasy `Parcel`. Konstruktor na podstawie danych zawartych w tej paczce ma za zadanie utworzyć odtworzyć stan obiektu (utworzyć „klon”). Jak widać na przykładzie do odczytywania wykorzystywane są metody `read???()` pozwalające na odczytywanie wartości określonych typów. Wartości są odczytywane w takiej kolejności jak zostały zapisane tzn. jeżeli zapisując stan obiektu w paczce (*ang. parcel*) umieściliśmy liczbę pobranych bajtów, rozmiar pliku a na koniec wynik pobierania to w takiej samej kolejności musimy je odczytać,
- metody `describeContents()`, która zwraca wartość opisującą zawartość paczki (zazwyczaj metoda zwraca 0),
- metody `writeToParcel()`, która do paczki (parametru typu `Parcel`) zapisuje stan obiektu. Do zapisywania służą metody `write???()`. Istnieje wiele metod pozwalających na zapisywanie rozmaitych typów. Należy pamiętać, żeby kolejność zapisywania wartości zgadzała się z kolejnością ich odczytywania w opisanym wyżej konstruktorze,
- Obiektu `CREATOR` klasy anonimowej implementującej interfejs `Parcelable.Creator<NazwaKlasy>`. Jakkolwiek może on wyglądać skomplikowanie to w praktyce jest on bardzo szablonowy i jedynym elementem wymagającym zmiany jest `NazwaKlasy` (w naszym wypadku jest to `PostepInfo`).

4.2.3 Odbiorcy komunikatów

W wielu aplikacjach ich składniki muszą się ze sobą komunikować. Odbiorcy komunikatów stanowią dobry kanał przesyłania danych – komunikatów – pomiędzy elementami jednej lub wielu aplikacji. Umożliwiają np. reagowanie na zdarzenia takie jak zakończenie ładowania systemu czy zmiana stanu baterii. Możliwe jest tworzenie i wysyłanie własnych komunikatów, co wykorzystamy do przesyłania danych z usługi do aktywności. Będziemy przesyłać informacje na temat postępów pobierania pliku wykorzystując klasę `PostepInfo` i zaimplementowany w niej interfejs `Parcelable`. Rozpoczniemy od zaimplementowania części wysyłającej komunikaty w naszej usłudze `PobieranieService`. Implementacja ta została przedstawiona na listingu 4.6.

Listing 4.6. Wysyłanie komunikatów z usługi

```
public class PobieranieService extends IntentService {

      //... - część kodu usunięto

      public final static String POWIADOMIENIE =
        "pl.pollub.android_tlo.odbiornik";
      public final static String POSTEP_INFO =
        "info o postepie";

      //... - część kodu usunięto

      int mPoprzedniePrzocenty;
      int procenty() {
        return (int) ((double) mPostepInfo.mPobranychBajtow
          / (double) mPostepInfo.mRozmiar * 100);
      }
      void wyslijWiadomosc() {
        int noweProcenty = procenty();
        if (noweProcenty != mPoprzedniePrzocenty
            || mPostepInfo.mWynik == PostepInfo.POBIERANIE_BLAD
            || mPostepInfo.mWynik == PostepInfo.POBIERANIE_OK) {
          Intent zamiar = new Intent(POWIADOMIENIE);
          zamiar.putExtra(POSTEP_INFO, mPostepInfo);
          sendBroadcast(zamiar);
          mPoprzedniePrzocenty = noweProcenty;
        }
      }
}
```

Pierwszym nowym elementem są dwie stałe. Stała `POWIADOMIENIE` będzie stanowić identyfikator naszych komunikatów pozwalający na odróżnienie go od innych. Będziemy z niej korzystać tworząc nowe komunikaty i rejestrując naszego odbiorcę. Kolejną stałą jest `POSTEP_INFO`, która stanowi opis elementu umieszczanego w intencji. Dalej zdefiniowana jest metoda pomocnicza `procenty()` obliczająca jaki procentowo fragment pliku już pobrano, na podstawie danych zachowanych w obiekcie `mPostepInfo`. Faktyczne wysyłanie komunikatu ma miejsce w metodzie `wyslijWiadomosc()`. Na początku sprawdza ona czy zmieniła się liczba procent pobranego pliku lub wystąpił jakiś błąd (nie warto zbyt często wysyłać komunikatów). Jeżeli warto zaktualizować dane Tworzona jest nowa intencja a w niej umieszczany jest obiekt `mPostepInfo` (możemy dodać ten obiekt za pomocą `putExtra()` ponieważ klasa `PostepInfo` implementuje interfejs `Parcelable`). Komunikat wysyłany jest za pomocą metody `sendBroadcast()`. Jak łatwo zauważyć cały mechanizm tworzenia komunikatu jest bardzo podobny do uruchamiania innej aktywności czy uruchamiania usługi. Na koniec zapisujemy właśnie wysłaną wartość procentową, która będzie nam potrzebna do kolejnego porównania.

Przejdziemy teraz do sposobu odbierania komunikatów. Część aktywności `PobieranieActivity` odpowiedzialną za odbieranie komunikatów przedstawiono na listingu 4.7. Odbieranie komunikatów wymaga stworzenia klasy dziedziczącej po `BroadcastReveiver`.

Listing 4.7. Odbieranie rozgłoszeń w aktywności `PobieranieActivity`

```
public class PobieranieActivity extends Activity {

      //... - część kodu usunięto

      private BroadcastReceiver mOdbiorcaRozgloszen =
         new BroadcastReceiver() {

        @Override
        public void onReceive(Context context, Intent intent) {
          Bundle tobolek = intent.getExtras();
          PostepInfo postepInfo =
            tobolek.getParcelable(
              PobieranieService.POSTEP_INFO);
          aktualizujPostep(postepInfo);
        }

      };
```

```
    protected void aktualizujPostep(PostepInfo postepInfo) {
      ProgressBar pasekPostepu =
        (ProgressBar) findViewById(R.id.postepPasek);
      pasekPostepu.setProgress(
        (int) ((double) postepInfo.mPobranychBajtow
               / (double) postepInfo.mRozmiar * 100.0));
      TextView pobranoBajtowWartoscEtykieta =
        (TextView) findViewById(
          R.id.pobranoBajtowWartoscEtykieta);
       pobranoBajtowWartoscEtykieta.setText(
      Integer.toString(postepInfo.mPobranychBajtow));
      //... - część kodu usunięto
    }

    @Override
    protected void onPause() {
      super.onPause();
      unregisterReceiver(mOdbiorcaRozgloszen);
    }

    @Override
    protected void onResume() {
      super.onResume();
      registerReceiver(mOdbiorcaRozgloszen, new IntentFilter(
                       PobieranieService.POWIADOMIENIE));
    }
}
```

Jak widać pierwszym elementem jest obiekt `mOdbiorcaRozgloszen`. Jest to obiekt klasy anonimowej dziedziczącej po `BroadcastReceiver`. W klasie tej definiujemy tylko jedną metodę `onReceive()`. Ma ona dwa parametry. Jednym z nich jest intencja, z której możemy odczytać obiekt klasy `Bundle`, z którego z kolei możemy wydobyć obiekt klasy `PostepInfo` za pomocą `getParcelable()`. Gdy już dysponujemy obiektem z informacją o postępie pobierania pliku wywołujemy metodę pomocniczą `aktualizujPostep()`.

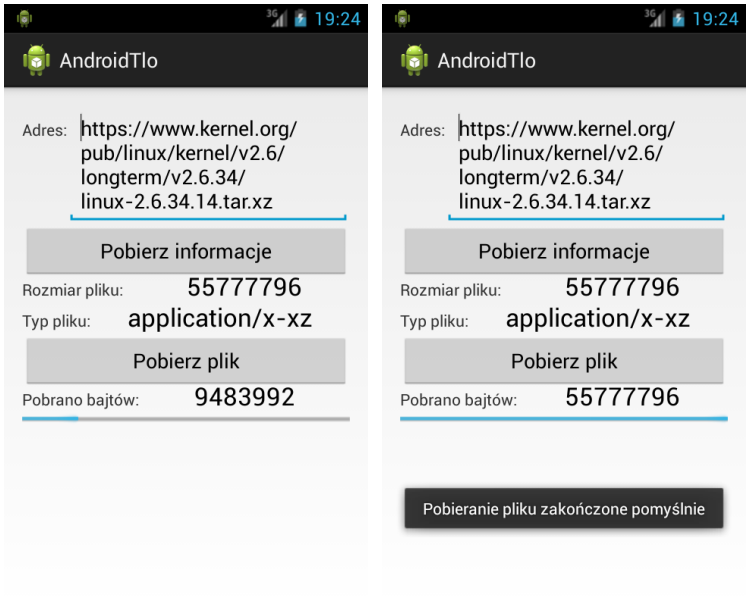
Nie będziemy się tutaj zajmować metodą pomocniczą. Jej zadaniem jest po prostu umieszczenie informacji z obiektu klasy `PostepInfo` w odpowiednich komponentach GUI znajdujących się w naszej aktywności.

Dużo bardziej istotne z punktu widzenia odbierania komunikatów jest to co znajduje się w metodach `onPause()` i `onResume()` aktywności `PobieranieActivity`. Mianowicie – w metodzie `onResume()` (uruchamianej po tym jak aktywność jest wznawiana) rejestrujemy naszego odbiorcę rozgłoszeń za pomocą `registerReceiver()`. Dzięki temu aktywność będzie odbierała komunikaty wysyłane z usługi pobierającej plik. Metoda `registerReceiver()`

posiada dwa parametry – pierwszym z nich jest obiekt odbiorcy rozgłoszeń a drugim filtr intencji, który pozwala na wybranie z rozsyłanych komunikatów tych, które są opisane identyfikatorem `POWIADOMIENIE`. W metodzie `onPause()` (uruchamianej gdy aktywność traci fokus) wyrejestrowujemy odbiorcę za pomocą `unregisterReceiver()`. Dzięki temu, aktywność przestanie odbierać komunikaty w momencie, gdy użytkownik nie będzie nimi zainteresowany.

Ćwiczenie 4.5 (do samodzielnego wykonania)

Zaimplementuj interfejs `Parcelable` w klasie `PostepInfo` zgodnie z informacjami z punktu 4.2.2. Następnie zmodyfikuj klasy `PobieranieService` i `PobieranieActivity` zgodnie z informacjami z punktu 4.2.3. Uzupełnij metodę `onHandleIntent()` w klasie `PobieranieService` o ustawianie wartości początkowej pola `mPoprzedniePrzecenty` i wywołania metody `wyslijWiadomosc()`. Metoda ta powinna być wywoływana za każdym razem, gdy zmienią się wartości zapisane w obiekcie `mPostepInfo`. Uzupełnij metodę `aktualizujPostep()` w klasie `PobieranieActivity`, tak aby wyświetlała komunikat (`Toast`), gdy pobieranie zakończyło się pomyślnie lub zakończyło się błędem (komunikat powinien oczywiście być odpowiedni do sytuacji). Przykładowy efekt działania aplikacji pokazano na rysunku 4.3.



Rys 4.3. Stan pobierania pliku aktualizowany na podstawie komunikatów z usługi (w trakcie pobierania oraz po zakończeniu pobierania)

4.2.4 Wykorzystanie powiadomień

Każdy kto używał telefonu bądź tabletu z Androidem, z całą pewnością zetknął się z powiadomieniami wyświetlanymi na pasku znajdującym się u góry ekranu. Powiadomienia te są doskonałym miejscem do wyświetlania informacji przez usługi pracujące w tle. Dlatego dodamy teraz do naszej aplikacji wyświetlanie powiadomień o postępie pobierania pliku. Powiadomienia te będą się wyświetlać na pasku niezależnie od tego, czy aktywność `PobieranieActivity` naszej aplikacji będzie widoczna czy też nie. Poprzez kliknięcie powiadomienia będzie można powrócić do aktywności `PobieranieActivity`. Sposób tworzenia i aktualizacji powiadomień przedstawiono na listingu 4.8.

Listing 4.8. Wysyłanie powiadomień z usługi `PobieranieService`

```
NotificationManager mMenedzerPowiadomien;
int mIdPowiadomienia;
Notification.Builder mBudowniczyPowiadomien;
Intent mZamiarPowiadomienia;

void przygotujPowiadomienie() {
  mZamiarPowiadomienia = new Intent(this,
    PobieranieActivity.class);
  //... - zapisanie adresu, rozmiaru pliku i typu pliku w
  //zamiarze

  TaskStackBuilder budowniczyStosu =
    TaskStackBuilder.create(this);
  budowniczyStosu.addParentStack(
    PobieranieActivity.class);
  budowniczyStosu.addNextIntent(mZamiarPowiadomienia);
  PendingIntent zamiarOczekujacy =
    budowniczyStosu.getPendingIntent(0,
      PendingIntent.FLAG_UPDATE_CURRENT);

  mMenedzerPowiadomien =
    (NotificationManager)
    getSystemService(NOTIFICATION_SERVICE);
  mIdPowiadomienia = 1;
  mBudowniczyPowiadomien = new Notification.Builder(this);
  mBudowniczyPowiadomien.setContentTitle(
    getString(R.string.powiadomienie_tytul))
    .setProgress(1, 0, false)
    .setContentIntent(zamiarOczekujacy)
```

```
        .setSmallIcon(R.drawable.ic_launcher)
        .setOngoing(true);
      mMenedzerPowiadomien.notify(mIdPowiadomienia,
        mBudowniczyPowiadomien.build());
    }

    void aktualizujPowiadomienie() {
      mBudowniczyPowiadomien.setProgress(mPostepInfo.mRozmiar,
        mPostepInfo.mPobranychBajtow, false);
      if (mPostepInfo.mWynik == PostepInfo.POBIERANIE_OK
          || mPostepInfo.mWynik == PostepInfo.POBIERANIE_BLAD)
      {
        mZamiarPowiadomienia.putExtra(POSTEP_INFO,
                                      mPostepInfo);
        TaskStackBuilder budowniczyStosu =
          TaskStackBuilder.create(this);
        budowniczyStosu.addParentStack(
          PobieranieActivity.class);
        budowniczyStosu.addNextIntent(mZamiarPowiadomienia);
        PendingIntent zamiarOczekujacy =
          budowniczyStosu.getPendingIntent(0,
            PendingIntent.FLAG_UPDATE_CURRENT);
        mBudowniczyPowiadomien.setOngoing(false)
          .setAutoCancel(true)
          .setContentIntent(zamiarOczekujacy);
      }
      mMenedzerPowiadomien.notify(mIdPowiadomienia,
      mBudowniczyPowiadomien.build());
    }
```

Omówimy teraz kolejne elementy nowego kodu usługi przedstawione na listingu 4.8. Rozpoczynamy od dodania kilku nowych pól:

- `mManagerPowiadomien` – referencja do menedżera pozwalającego na umieszczanie powiadomień na pasku,
- `mIdPowiadomienia` – identyfikator powiadomienia, pozwalający na odróżnienie danego powiadomienia od innych wysłanych przez aplikację. Ponieważ nasza aplikacja wysyła tylko jedno powiadomienie (a potem je aktualizuje) `mIdPowiadomienia` będzie po prostu równe 1,
- `mBudowniczyPowiadomien` – referencja do obiektu pozwalającego na łatwe tworzenie powiadomień,
- `mZamiarPowiadomienia` – intencja powiązana z powiadomieniem pozwalająca na przejście do aktywności `PobieranieActivity`.

W dalszej kolejności dodajemy dwie metody. Pierwszą z nich jest `przygotujPowiadomienie()`. Zadaniem metody jest utworzenie,

przygotowanie i umieszczenie (po raz pierwszy) powiadomienia na pasku powiadomień. Metoda rozpoczyna od utworzenia intencji (`mZamiarPowiadomienia`), która uruchomi aktywność `PobieranieActivity`.

Kolejnym elementem jest utworzenie budowniczego stosu – obiektu klasy `TaskStackBuilder`. Posłuży on do zbudowania tzw. *back-stack* dla aplikacji. Na czym polega *back-stac*k? Otóż jak już wiemy aplikacje składają się z aktywności między którymi porusza się użytkownik. W momencie gdy użytkownik przechodzi do nowej aktywności bieżąca odkładana jest na stos (*back-stack*). Gdy nowa aktywność jest widoczna a użytkownik naciśnie przycisk *wstecz* – nowa aktywność zostanie zakończona a poprzednia, której używał, zostanie zdjęta ze stosu i wyświetlona na ekranie. Stworzenie odpowiedniego stosu pozwala na zapewnienie zachowania aplikacji, które będzie się wydawało użytkownikom logiczne – tzn. po przejściu do konkretnej aktywności poprzez powiadomienie kliknięcie przycisku wstecz spowoduje przejście do aktywności, która „standardowo jest poprzednia" w ramach aplikacji zamiast do aktywności, którą użytkownik oglądał poprzednio. Tworzenie stosu rozpoczynamy od wywołania metody `addParentStack()`, która dodaje stos aktywności będącej rodzicem uruchamianej aktywności. Aby w przypadku skomplikowanych aplikacji metoda dawała zamierzony efekt konieczne jest zdefiniowanie wspomnianego rodzica w manifeście aplikacji (należy zdefiniować atrybut `android:parentActivityName` dla uruchamianej aktywności). Ponieważ aktywność `PobieranieActivity` jest główną (i jedyną) aktywnością aplikacji nie musimy definiować rodzica. Dalej dodajemy (w naszym wypadku) jedną, utworzoną wcześniej, intencję `mZamiarPowiadomienia` do tworzonego stosu z pomocą `addNextIntent()` (gdybyśmy dodali więcej intencji – po przejściu do aplikacji za pomocą powiadomienia zostałaby uruchomiona ta znajdująca się na szczycie stosu). Po utworzeniu stosu – za pomocą budowniczego – tworzymy *intencję oczekującą* (*ang. pending intent*) `zamiarOczekujacy` – będzie nam ona potrzebna przy tworzeniu powiadomienia.

Po uzyskaniu oczekującej intencji przystępujemy do konstruowania powiadomienia. Rozpoczynamy od utworzenia pomocniczego obiektu `mBudowniczyPowiadomien` klasy `Notification.Builder`. Za pomocą metod `setContentTitle()`, `setProgress()`, `setSmallIcon()` i `setOngoing()` ustawiamy właściwości powiadomienia. Są to kolejno:

- tytuł powiadomienia,
- zaawansowanie postępu (parametry `setProgress()` to: wartość maksymalna, wartość bieżąca, informacja czy postęp da się określić liczbowo czy nie),
- ikona powiadomienia,
- informacja o tym, czy powiadomienie dotyczy procesu, który właśnie się dzieje czy też zakończonej już czynności.

Po ustawieniu wszystkich właściwości powiadomienia możemy je utworzyć za pomocą `mBudowniczyPowiadomien.build()` i od razu wyświetlić za pomocą menedżera powiadomień `mMenedzerPowiadomien` i jego metody `notify()`.

Drugą metodą, którą dodamy do usługi jest `aktualizujPowiadomienie()`. Jej zadaniem jest aktualizowanie wyświetlonego już powiadomienia. Metoda wykorzystuje istniejący już obiekt `mBudowniczyPowiadomien` (utworzony przez `przygotujPowiadomienie()`). Zmienia w nim informacje na temat postępu, a jeżeli pobieranie się zakończyło (pomyślnie lub z błędem) zmienia dodatkowo rodzaj powiadomienia na takie, które dotyczy zakończonej czynności (`setOngoing(false)`), włącza automatyczne usuwanie powiadomienia w momencie gdy użytkownik je wybierze (`setAutoCancel(true)`), oraz tworzy nowy *back-stack* ponieważ do uruchamianej intencji `mZamiarPowiadomienia` został dodany obiekt `mPostepInfo`. Wreszcie buduje powiadomienie i aktualizuje informację wyświetlaną użytkownikowi.

Aby zapewnić poprawne odtworzenie stanu aktywności, gdy przechodzimy do niej za pomocą powiadomienia, musimy jeszcze w metodzie `onCreate()` aktywności `PobieranieActivity` zaimplementować odczytywanie informacji o postępie pobierania, tak jak to pokazano na listingu 4.9. Odbywa się to tak samo jak w przypadku, w którym wywołujemy aktywność „normalnie” i chcemy odczytać przekazane do niej dane.

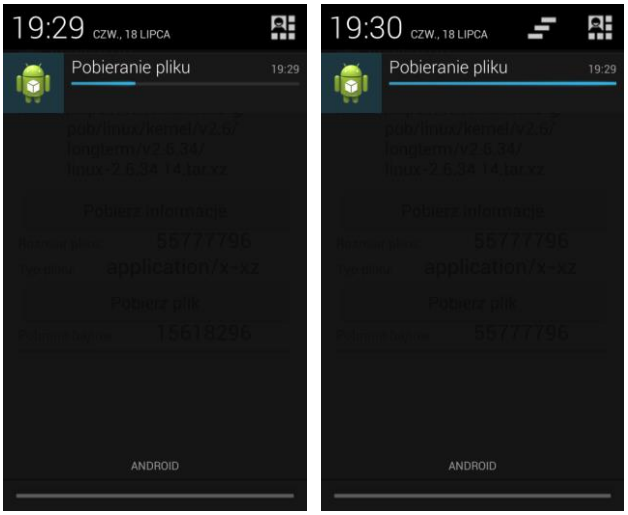
Listing 4.9. Odbieranie danych z intencji przekazanej przez powiadomienie

```
protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.pobieranie_activity);

  Bundle tobolek = getIntent().getExtras();
  if (tobolek != null) {

    //... - odczytywanie adresu, rozmiaru pliku i typu
    //pliku

    if (tobolek.containsKey(
        PobieranieService.POSTEP_INFO))
      aktualizujPostep((PostepInfo)tobolek
        .getParcelable(PobieranieService.POSTEP_INFO));
  }

  //... - część kodu usunięto

}
```

Ćwiczenie 4.6 (do samodzielnego wykonania)

Dodaj do usługi `PobieranieService` metody `przygotujPowiadomienie()` i `aktualizujPowiadomienie()` przedstawione w punkcie 4.2.4. Zadbaj o to, aby metoda `przygotujPowiadomienie()` była wywoływana tylko raz, gdy rozpoczyna się pobieranie pliku. Metoda `aktualizujPowiadomienie()` powinna być wywoływana w momencie, gdy zmienia się liczba procent pobranego pliku (nie należy generować zbyt wielu powiadomień – przy każdej pobranej 32kB porcji danych). Sprawdź działanie aplikacji. Przykładowy efekt działania pokazano na rysunku 4.4.

Dodatkowo sprawdź, czy po przejściu do ekranu domowego można powrócić do aplikacji, wybierając powiadomienie? Jeżeli tak, to czy stan aktywności jest prawidłowy?



Rys. 4.4. Powiadomienie ze statusem pobierania pliku (w trakcie pobierania i po jego zakończeniu)

Ćwiczenie 4.7 (do samodzielnego wykonania)

Zmodyfikuj metodę `przygotujPowiadomienie()` w klasie `PobieranieService`, tak aby zamiarze powiadomienia był zapisywany: adres pobieranego pliku, jego rozmiar i typ. W metodzie `onCreate()` klasy `PobieranieActivity` dokonaj zmian, które spowodują odczytanie i ustawienie wymienionych danych w odpowiednich etykietach. Wykorzystaj informacje z punktu 4.2.4.

5. Wykorzystanie usług frameworku Android

W niniejszym rozdziale na prostym przykładzie poznamy usługi frameworku Android dostępne w systemie operacyjnym. Usługi te często opierają się na wykorzystaniu chmury firmy Google. My posłużymy się dwiema popularnymi usługami jakimi są Mapy Google oraz zamienianie mowy na tekst. Zrealizujemy aplikację, do której można powiedzieć jaki adres nas interesuje a ona wyświetli go na mapie.

5.1. Wykorzystanie map

Korzystanie usług frameworku Androida rozpoczniemy – tak jak w poprzednich rozdziałach od utworzenia projektu, w którym będziemy budować naszą aplikację. Tym razem projekt nazwiemy `AndroidMapy` natomiast pakiet aplikacji będzie nazywał się `pl.pollub.android_mapy`. Jak okaże się za chwilę nazwa pakietu w tym wypadku będzie bardzo istotna. Przy okazji tworzenia projektu możemy utworzyć aktywność `MapaActivity` z układem o nazwie `activity_mapa.xml`.

5.1.1. Przygotowanie do użycia Google Map v2

Do korzystania z Map Google we własnej aplikacji niezbędny jest tzw. `API_KEY`. Warto zaznaczyć, że w chwili powstawania niniejszego podrzędnika emulator nie umożliwia korzystania z API map w wersji drugiej. Niezbędne jest do tego rzeczywiste urządzenie z Androidem w wersji minimum 2.2.

Aby uzyskać `API_KEY` należy [9]:

- Odczytać odcisk palca klucza używanego do podpisywania aplikacji (każda kompilacja aplikacji jest podpisywana cyfrowo – nawet ta uruchamiana na emulatorze). My odczytamy tzw. *debug key*, używany do podpisywania aplikacji w trakcie pracy nad kodem w środowisku Eclipse. Odcisk palca używanego klucza można uzyskać za pomocą polecenia:

```
keytool -list -v -keystore ~/.android/debug.keystore
  -alias androiddebugkey -storepass android
  -keypass android
```

  Należy zanotować odcisk SHA1.

- Przejść na stronę: https://code.google.com/apis/console/ i zalogować się za pomocą swojego konta Google.

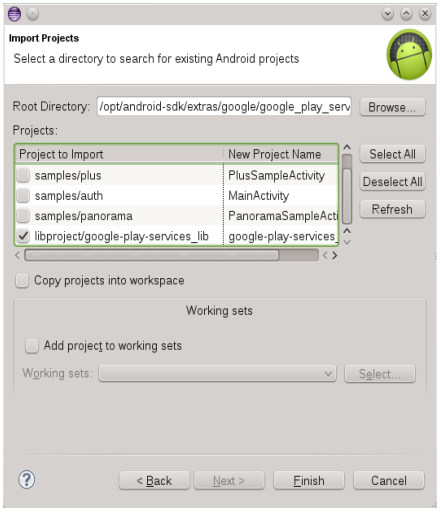
- Po zalogowaniu utworzyć nowy projekt wybierając *API Project* (w lewym górnym rogu) i *Create*
- Na liście usług (opcja Services po lewej stronie) należy odszukać i uaktywnić opcję Google *Maps Android API v2*
- Przejść do sekcji *API Access* (opcja po lewej stronie).
- Wygenerować nowy klucz za pomocą opcji *Create New Android Key...*. W trakcie tworzenia klucza będzie niezbędny odczytany wcześniej odcisk klucza i nazwa pakietu aplikacji. Podaje się je w następującej postaci:

```
XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX:XX;pl.p
ollub.android_mapy
```

- Uzyskany klucz powinien mieć postać:
  `XXXXXXXXXX-XXXXXXXXXXXXXXXXXXXXXXXXXXX`

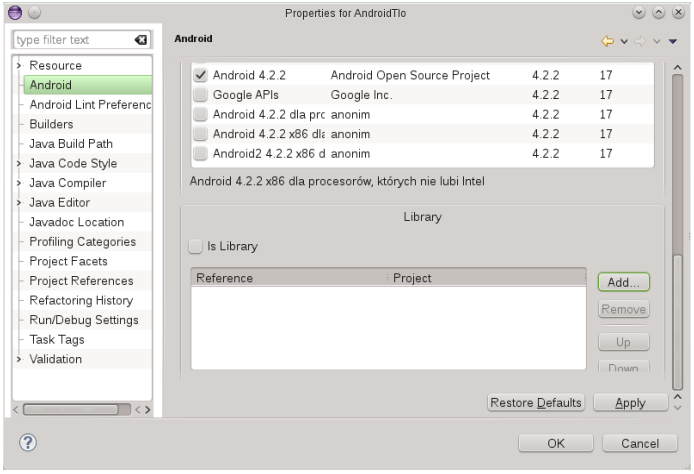
Kolejnym etapem przygotowań jest umożliwienie naszej aplikacji korzystania z usług Google Play. W tym celu należy:

- zainstalować pakiet *Google Play services*. Instaluje się go za pomocą *Android SDK Manager* (polecenie `android` z SDK). Pakiet znajduje się w sekcji *Extras*,
- zaimportować bibliotekę usług Google Play do przestrzeni roboczej środowiska Eclipse czyli:
  - wybrać menu *File* | *Import* | *Android* | *Existing Android Code Into Workspace*,
  - w oknie dialogowym wybrać katalog:
    `android-sdk/extras/google/google_play_services`
  - w kolejnym oknie – oknie importu projektów (rys. 5.1) – wybrać
    `libproject/google-pay-services_lib`
    i zatwierdzić wybór



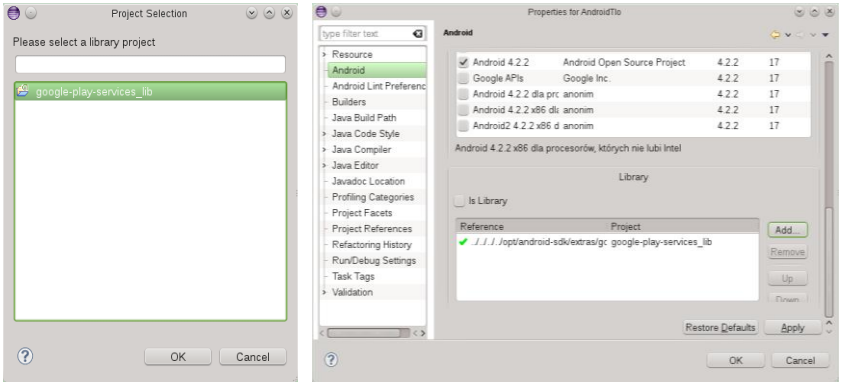
Rys. 5.1. Wybór importowanego projektu

- dodać zaimportowaną bibliotekę jako zależność naszej aplikacji. Aby tego dokonać:
  - klikamy projekt `AndroidMapy` prawym przyciskiem myszy i wybieramy *Properties* | *Android*. Pojawi się okno przedstawione na rysunku 5.2.



Rys. 5.2. Okno właściwości projektu związanych z Androidem

- w oknie właściwości w sekcji z nagłówkiem *Library* wybieramy przycisk *Add...* i w nowym oknie wybieramy `google-play-services_lib` – czyli wcześniej zaimportowany projekt. W oknie właściwości projektu pojawi się dodana zależność (rys. 5.3)



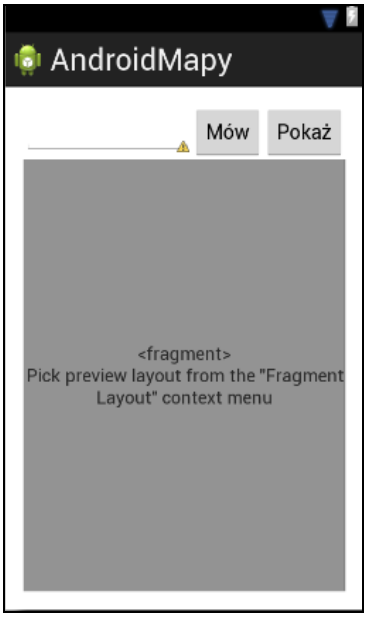
Rys. 5.3. Dodawanie zależności

Teraz możemy w naszym projekcie użyć Map Google.

Ćwiczenie 5.1 (do samodzielnego wykonania)

Zmodyfikuj układ `activity_mapa.xml` tak aby przypominał ten z rysunku 5.4. Dla pola tekstowego i dwóch przycisków użyj następujących identyfikatorów: `adresEdycja`, `mowPrzycisk` oraz `pokazPrzycisk`. Elementem wyświetlającym mapę jest fragment, który należy umieścić w układzie za pomocą następującej deklaracji:

```
<fragment
    android:id="@+id/mapa"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:layout_below="@+id/pokazPrzycisk"
    class="com.google.android.gms.maps.MapFragment" />
```



Rys. 5.4. Wygląd układu `activity_mapa.xml`

5.1.2. Używanie Google Map API v2

API Map Google posiada dużo większe możliwości niż tylko wyświetlanie mapy. Jedną z nich jest zaznaczanie na mapie interesujących miejsc. Można je zaznaczać za pomocą znaczników (*ang. marker*). Zmodyfikujemy teraz naszą aplikację tak aby wyświetlała lokalizację Instytutu Informatyki Politechniki Lubelskiej. Listing 5.1 zawiera niezbędny kod.

Listing 5.1. Wyświetlanie markerów na mapie

```
public class MapaActivity extends Activity {

       static final LatLng II_PL =
         new LatLng(51.235395, 22.553085);
       private GoogleMap mMapa;

       @Override
       protected void onCreate(Bundle savedInstanceState) {
         super.onCreate(savedInstanceState);
         setContentView(R.layout.activity_mapa);
```

```
        mMapa = ((MapFragment) getFragmentManager()
                               .findFragmentById(R.id.mapa))
                               .getMap();
        mMapa.addMarker(new MarkerOptions().position(II_PL)
          .title(getString(R.string.lokalizacja_ii_nazwa))
          .snippet(getString(R.string.lokalizacja_ii_opis))
          .icon(BitmapDescriptorFactory
          .fromResource(R.drawable.ic_launcher)));
        mMapa.moveCamera(
          CameraUpdateFactory.newLatLngZoom(II_PL, 15));
        mMapa.animateCamera(CameraUpdateFactory.zoomTo(10),
                            1000, null);
    }
}
```

Jak widać rozpoczynamy od utworzenia obiektu klasy `LatLng`, który przechowuje współrzędne Instytutu. Tworzymy też pole pomocnicze, w którym będziemy przechowywać referencję do mapy. Samo dodawanie markera odbywa się w metodzie `onCreate()`. Po uzyskaniu referencji do pola do mapy dodajemy do niej tworzony w miejscu znacznik. Ustawiamy pozycję (metoda `position()`), tytuł (`title()`), opis (`snippet()`) i niestandardową ikonkę znacznika (`icon()`). Jako ikonę znacznika wykorzystamy ikonę naszej aplikacji. Następnie przesuwamy widok mapy tak aby był wyśrodkowany na nowym markerze (metoda `moveCamera()`). Ustawiamy też poziom szczegółowości/przybliżenia mapy (metoda `animateCamera()`).

Aby nasza aplikacja zaczęła działać musimy wprowadzić modyfikacje manifestu. Ponieważ jest ich dużo na listingu 5.2 zamieszczona została jego pełna.

Listing 5.2. Manifest aplikacji korzystającej z Map Google

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android=
      "http://schemas.android.com/apk/res/android"
    package="pl.pollub.android_mapy"
    android:versionCode="1"
    android:versionName="1.0" >

    <uses-sdk
        android:minSdkVersion="17"
        android:targetSdkVersion="17" />
    <uses-feature
        android:glEsVersion="0x00020000"
```

```
        android:required="true" />

    <uses-permission android:name=
       "android.permission.INTERNET" />
    <uses-permission android:name=
      "android.permission.ACCESS_NETWORK_STATE" />
    <uses-permission android:name=
      "android.permission.WRITE_EXTERNAL_STORAGE" />
    <uses-permission android:name=
    "com.google.android.providers.gsf.permission.READ_GSERVICES"
    />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme" >
        <activity
            android:name="pl.pollub.android_mapy.MapaActivity"
            android:label="@string/app_name" >
            <intent-filter>
                <action android:name=
                  "android.intent.action.MAIN" />

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

        <meta-data
            android:name="com.google.android.maps.v2.API_KEY"
            android:value=
              "XXXXXXXXXX-XXXXXXXXXXXXXXXXXXXXXXXXXXX" />
    </application>

</manifest>
```

Jak widać występuje w nim dość dużo nowych elementów. Pierwszym z nich jest znacznik uses-feature, w którym informujemy, że nasza aplikacja wymaga aby urządzenie obsługiwało standard OpenGL ES 2.0. Dalej wymieniamy cztery uprawnienia potrzebne naszej aplikacji. Są to kolejno:

- dostęp do internetu (Map Google pobierają niezbędne fragmenty map z internetu),
- dostęp do informacji o stanie sieci (umożliwia sprawdzenie stanu

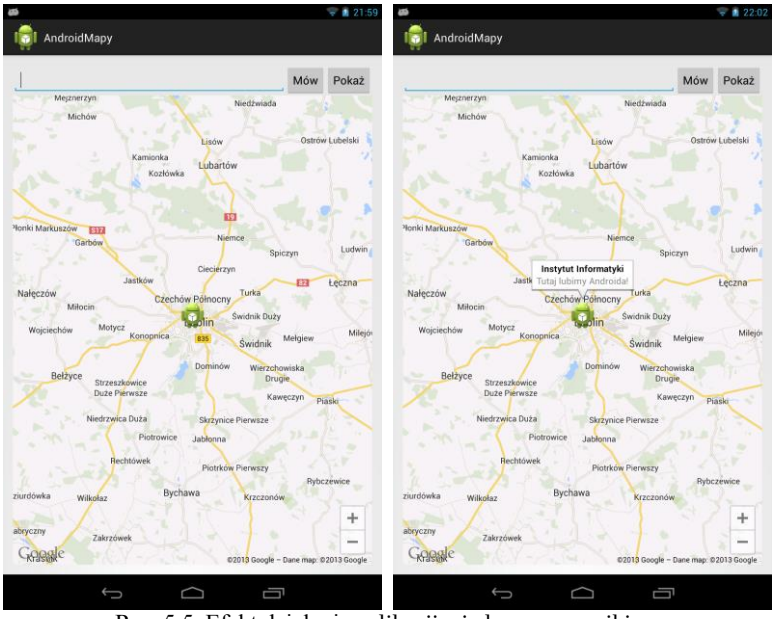
połączenia internetowego niezbędnego do pobierania fragmentów map),
- dostęp do webowych usług Google (`com.google.android.providers.gsf.permission.READ_GSERVICES`),
- możliwość zapisywania danych na karcie pamięci (Mapy Google buforują pobrane dane).

Ostatnim nowym elementem jest znacznik meta-data zagnieżdżony w znaczniku application. To w meta danych umieszczamy uzyskany wcześniej klucz API Map Google.

Ćwiczenie 5.2 (do samodzielnego wykonania)

Wprowadź do naszej przykładowej aplikacji elementy opisane w punkcie 5.1.2 i sprawdź działanie aplikacji (w tym wyświetlanie opisów). Efekt powinien być podobny do tego na rysunku 5.5. Następnie dodaj do mapy kilka znaczników wskazujących miejsca, które dla Ciebie są interesujące.

Wskazówka. Współrzędne geograficzne punktu można odczytać np. z Map Google dostępnych poprzez przeglądarkę WWW. Po odnalezieniu interesującego punktu należy kliknąć w tym miejscu na mapie prawym przyciskiem myszy. Z menu kontekstowego należy wybrać opcję: Co tu jest?. Na mapie powinna pojawić się zielona strzałka. Po najechaniu kursorem na strzałkę pojawi się prostokątna etykieta zawierająca współrzędne.



Rys. 5.5. Efekt działania aplikacji z jednym znacznikiem

5.1.3. Zamiana adresu na współrzędne geograficzne

Kolejną usługą, z której skorzystamy jest zamiana adresu na współrzędne geograficzne. Z usługi tej można skorzystać za pośrednictwem klasy `Geocoder`. Sposób wykorzystania klasy przedstawiono na listingu 5.3.

Listing 5.3. Zamiana adresu na współrzędne geograficzne

```
protected void pokazAdresNaMapie() {
  String adres = mAdresEdycja.getText().toString();
  Geocoder koder = new Geocoder(this);
  try {
    List<Address> adresy =
      koder.getFromLocationName(adres, 1);
    if (adresy.size() > 0) {
      double szerokosc = adresy.get(0).getLatitude();
      double dlugosc = adresy.get(0).getLongitude();
      LatLng nowaLokalizacja =
        new LatLng(szerokosc, dlugosc);

      //... - przesunięcie widoku

    } else {

      //... - wyświetlenie komunikatu o błędzie

    }
  } catch (IOException e) {
    e.printStackTrace();
  }
}
```

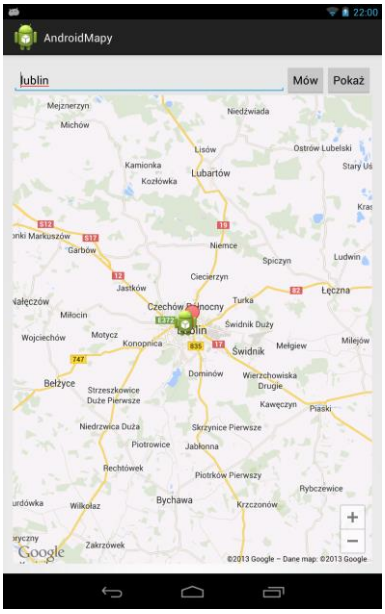
Na listingu przedstawiono metodę pomocniczą `pokazNaMapie()`. Na początku pobieramy adres wprowadzony w polu tekstowym a następnie tworzymy obiekt koder klasy `Geocoder`. Następnie za pomocą metody `getFromLocationName()` próbujemy zamienić adres w postaci tekstowej na współrzędne geograficzne. Pierwszym parametrem metody jest analizowany adres, natomiast drugim liczba wyników które ma zwrócić metoda (wprowadzony adres może nie być jednoznaczny – w bardziej zaawansowanej aplikacji można by wyświetlić użytkownikowi listę najlepszych dopasowań). W dalszej kolejności sprawdzamy czy udało się zamienić adres na współrzędne tzn. czy liczba wyników jest większa od zera. Jeżeli tak to przystępujemy do dodania nowego znacznika na mapie. W przeciwnym razie wyświetlimy komunikat o błędzie.

Ćwiczenie 5.3 (do samodzielnego wykonania)

Wykorzystując informacje z punktu 5.1.3 dodaj do naszej aplikacji funkcję pozwalającą na dodanie znacznika w miejscu określonym przez adres z pola tekstowego. W szczególności:

- dodaj do klasy `MapaActivity` pole prywatne `mAdresEdycja`. Zadbaj o ustawienie jego wartości początkowej w metodzie `onCreate()`,
- dodaj obsługę kliknięcia przycisku `pokazPrzycisk`. Powinna ona korzystać z metody pomocniczej `pokazAdresNaMapie()`,
- uzupełnij metodę `pokazAdresNaMapie()` tak aby dodawała do mapy nowy znacznik i przesuwała widok do nowego znacznika. Skorzystaj ze standardowej ikonki znacznika,
- uzupełnij metodę `pokazAdresNaMapie()` tak aby wyświetlała komunikat (`Toast`) informujący użytkownika, że nie udało się ustalić współrzędnych adresu.

Sprawdź działanie aplikacji. Wynik powinien być podobny do tego przedstawionego na rysunku 5.6.



Rys. 5.6. Znacznik dodany na podstawie adresu

5.2. Rozpoznawanie mowy

Ostatnią usługą, którą zajmiemy się w tym rozdziale jest usługa rozpoznawania mowy. Usługa ta działa bardzo sprawnie – nie tylko uwzględnia język polski, ale nawet radzi sobie z nazwami własnymi np. ulic. Dodawanie rozpoznawania mowy do aplikacji może wydawać się zadaniem bardzo skomplikowanym. Jest jednak wbrew pozorom bardzo proste. Wszytko dzięki wykorzystaniu standardowego mechanizmu Androida jakim są intencje. Sposób uruchomienia intencji rozpoznawania mowy przestawiono na listingu 5.4.

Listing 5.4. Użycie intencji rozpoznawania mowy

```
public void mowAdres() {
  Intent zamiar =
    new Intent(RecognizerIntent.ACTION_RECOGNIZE_SPEECH);
  zamiar.putExtra(RecognizerIntent.EXTRA_PROMPT,
                  getString(R.string.co_mowic));

  // podpowiedź
  // 1.LANGUAGE_MODEL_WEB_SEARCH : dla krótkich wyrażeń
  // 2.LANGUAGE_MODEL_FREE_FORM : gdy nie wiadomo co
  //będzie wyszukiwane
  zamiar.putExtra(RecognizerIntent.EXTRA_LANGUAGE_MODEL,
  RecognizerIntent.LANGUAGE_MODEL_WEB_SEARCH);
  zamiar.putExtra(RecognizerIntent.EXTRA_MAX_RESULTS, 1);
  startActivityForResult(zamiar, 0);
}
```

Jak widać sposób postępowania jest analogiczny do tego, którego używaliśmy we wcześniejszym rozdziale do uruchomienia przeglądarki. Tym razem naszą intencją jest `ACTION_RECOGNIZE_SPEECH`. Dla tej intencji tworzymy nowy obiekt zamiar. Następnie umieszczamy w tym obiekcie dane wspomagające proces rozpoznawania mowy. Pierwszą daną jaką dodajemy jest tekst wyświetlany użytkownikowi w trakcie gdy Android będzie nagrywał wypowiedź. W naszym wypadku będzie to podpowiedź informująca użytkownika o tym, że ma powiedzieć adres. Następnie dodajemy do zamiaru informacje o tym, że rozpoznawane wyrażenie będzie krótkie (połączenie parametrów `EXTRA_LANGUAGE_MODEL` i `LANGUAGE_MODEL_WEB_SEARCH`) i że oczekujemy tylko jednego wyniku rozpoznawania mowy. Na końcu metody `mowAdres()` używamy metody `startActivityForResult()` do wykonania zamiaru, który utworzyliśmy.

Ponieważ użyliśmy metody `startActivityForResult()` wyniki wykonania zamiaru zostaną zwrócone przez framework za pomocą wywołania metody `onActivityResult()`. Szablon tej metody przedstawiono na listingu 5.5.

Listing 5.5. Odczytanie wyników rozpoznawania mowy

```
@Override
protected void onActivityResult(int requestCode,
    int resultCode, Intent data) {
  if (resultCode == RESULT_OK) {
    ArrayList<String> dopasowania = data
     .getStringArrayListExtra(
        RecognizerIntent.EXTRA_RESULTS);
    if (!dopasowania.isEmpty()) {
      //... - odczytanie tekstu z listy, ustawienie tekstu
      //w polu adresu, zaznaczenie i wyświetlenie adresu
      //na mapie
    }
  } else
    //... - komunikat o błędzie rozpoznawania mowy
  super.onActivityResult(requestCode, resultCode, data);
}
```

Jak widać odczytanie wyników rozpoznawania mowy jest bardzo proste. Są one zwracane w intencji data jako lista łańcuchów tekstowych. Odczytujemy je z intencji za pomocą metody `getStringArrayListExtra()`. Identyfikatorem listy jest stała `RecognizerIntent.EXTRA_RESULTS`.

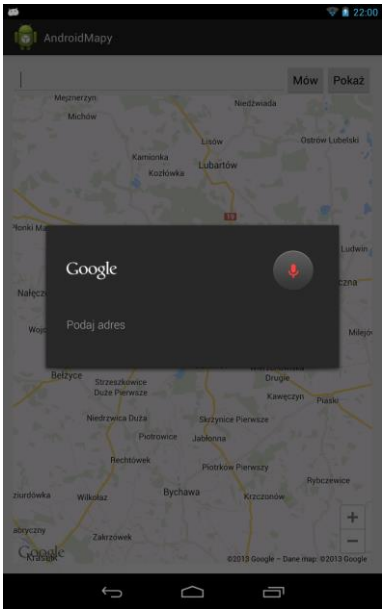
Ćwiczenie 5.4 (do samodzielnego wykonania)

Wykorzystując informacje z punktu 5.2 dodaj do naszej aplikacji funkcję rozpoznawania mowy. Powinna ona działać następująco:

- użytkownik klika przycisk `mowPrzycisk`,
- pojawia się okno takie jak na rysunku 5.7,
- użytkownik podaje adres,
- adres jest umieszczany w polu tekstowym i automatycznie pokazywany na mapie.

Realizując ćwiczenie:

- uzupełnij metodę `onActivityResult()` tak aby aplikacja wyświetlała rozpoznany adres w polu tekstowym i zaznaczała go na mapie,
- pamiętaj o dodaniu obsługi przycisku `mowPrzycisk`.



Rys. 5.7. Okienko rozpoznawania mowy

Bibliografia

[1] *Adding Platforms and Packages* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/sdk/installing/adding-packages.html
[2] *Creating a Content Provider* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/guide/topics/providers/content-provider-creating.html
[3] *Creating Contextual Menus* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/guide/topics/ui/menus.html#context-menu
[4] *Dokumentacja klasy Intent* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/reference/android/content/Intent.html
[5] *Dokumentacja klasy Manifest.permission* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/reference/android/Manifest.permission.html
[6] *Dokumentacja klasy RelativeLayout* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/reference/android/widget/RelativeLayout.html
[7] *Dokumentacja klasy Service* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/reference/android/app/Service.html
[8] *Dokumentacja klasy SQLiteOpenHelper* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html#getWritableDatabase%28%29)
[9] *Google Maps Android API v2 – Getting Started* [dostęp lipiec 2013]. Dostępny w World Wide Web: https://developers.google.com/maps/documentation/android/start
[10] *Installing the Eclipse Plugin* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/sdk/installing/installing-adt.html
[11] *Loaders* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/guide/components/loaders.html
[12] Setting Up the ADT Bundle [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/sdk/installing/bundle.html
[13] *Spis pakietów w API Androida* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/reference/packages.html
[14] *Support Library Features* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/tools/support-library/features.html
[15] *Support Library Setup* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://developer.android.com/tools/support-library/setup.html

[16] *What's the mechanism of setNotificationUri?* [dostęp lipiec 2013]. Dostępny w World Wide Web: http://stackoverflow.com/questions/11802823/whats-the-mechanism-of-setnotificationuri