O'REILLY®

Helion

Wydanie II

Rusz głową!

Android

Programowanie aplikacji

Zobacz jak wyglądają fragmenty pod mikroskopem

Unikaj kłopotliwych aktywności

Przekonaj się jak układy z ograniczeniami mogą zmienić Twoje życie

Twórz usługi nie z tego świata

Znajdź własną drogę dzięki usługom lokalizacyjnym Androida

Zabaw się z biblioteką wsparcia wzornictwa

Dawn Griffiths, David Griffiths

Tytuł oryginału: Head First Android Development: A Brain-Friendly Guide, 2nd Edition

Tłumaczenie: Piotr Rajca

ISBN: 978-83-283-4079-4

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie/andrr2
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

Spis treści (skrócony)

Spis treści (z prawdziwego zdarzenia)

Wprowadzenie

Twój mózg jest nastawiony na Androida.

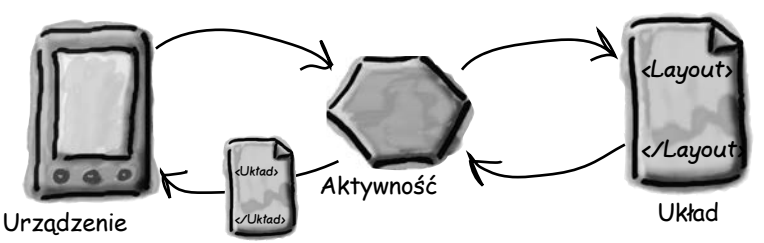
Jesteś *tu* po to, by się czegoś *nauczyć*, a Twój *mózg* robi Ci przysługę, upewniając się, że to, czego się nauczyłeś, szybko *wyleci* z pamięci. Twój mózg myśli sobie: „Lepiej zostawić miejsce na coś ważnego, na przykład na zastanowienie się nad tym, których dzikich zwierząt lepiej unikać albo czy jeżdżenie nago na snowboardzie to dobry pomysł". A zatem w jaki sposób możesz skłonić swój mózg, by myślał, że Twoje życie zależy od umiejętności pisania aplikacji na Androida?

Zaczynamy

1 Skok na głęboką wodę

Android błyskawicznie podbił świat.

Każdy chce mieć smartfon lub tablet, a urządzenia z Androidem są niezwykle popularne. W tej książce nauczymy Cię, jak **pisać własne aplikacje**, a zaczniemy od pokazania procesu przygotowania bardzo prostej aplikacji i uruchomienia jej na wirtualnym urządzeniu z Androidem. W trakcie tych prac poznasz także kilka podstawowych komponentów wszystkich aplikacji na Androida, takich jak **aktywności** i **układy**. **Jedyną rzeczą, której będziesz do tego potrzebować, jest znajomość Javy, choć wcale nie musisz być w niej mistrzem...**

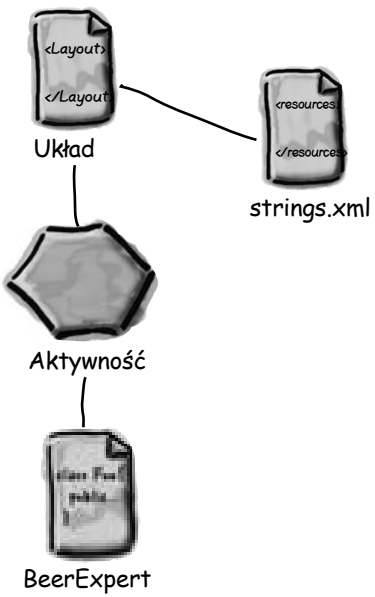


Tworzenie interaktywnych aplikacji

2 Aplikacje, które coś robią

Większość aplikacji musi w jakiś sposób reagować na poczynania użytkowników.

Z tego rozdziału dowiesz się, co zrobić, aby Twoje aplikacje były nieco bardziej interaktywne. Przekonasz się, jak zmusić aplikację, by coś ***zrobiła*** w odpowiedzi na działania użytkownika, oraz jak sprawić, by **aktywności i układy porozumiewały się ze sobą** jak starzy kumple. Przy okazji pokażemy Ci **nieco dokładniej, jak naprawdę działa Android** — poznasz plik **R**, czyli ukryty klejnot, który spaja pozostałe elementy aplikacji.

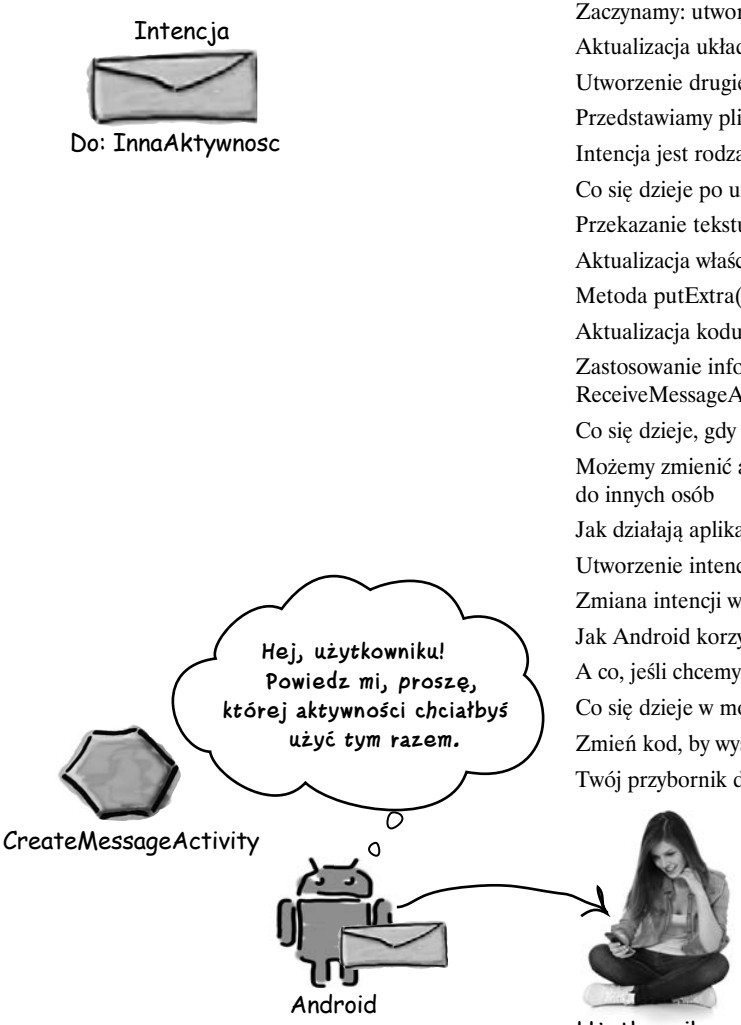


Wiele aktywności i intencji

3 Jakie są Twoje intencje?

Większość aplikacji potrzebuje więcej niż jednej aktywności.

Dotychczas mieliśmy do czynienia z aplikacjami składającymi się tylko z jednej aktywności. Kiedy jednak sprawy się komplikują, jedna aktywność zwyczajnie nie wystarczy. Dlatego w tym rozdziale pokażemy Ci, jak **tworzyć aplikacje składające się z wielu aktywności** i jak nasze aplikacje mogą porozumiewać się z innymi, wykorzystując ***intencje***. Pokażemy także, jak można używać intencji, by **wykraczać poza granice naszych aplikacji**, i jak wykorzystywać **aktywności należące do innych aplikacji dostępnych w urządzeniu do wykonywania *akcji***. To wszystko zapewni nam znacznie większe możliwości.

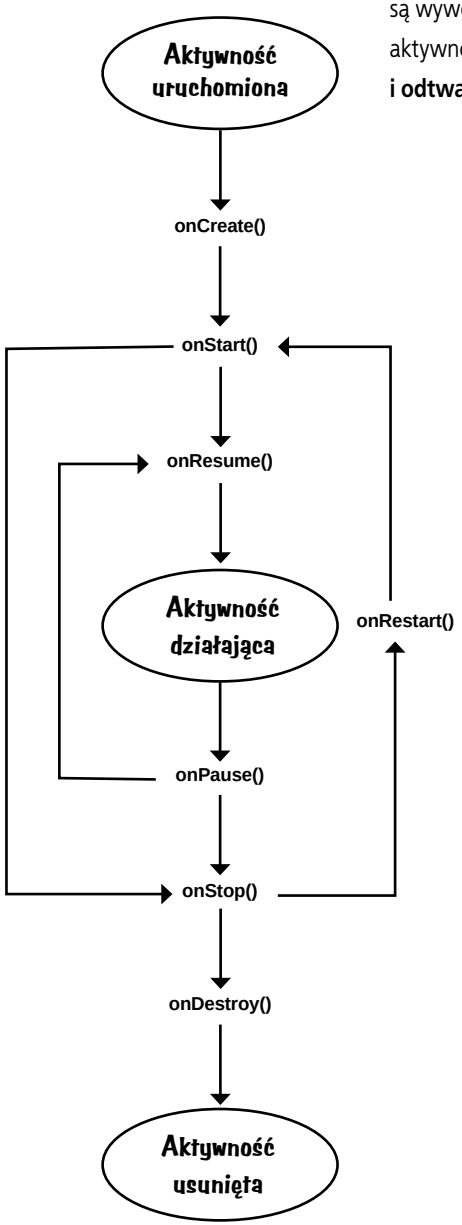


4 Cykl życia aktywności

Była sobie aktywność

Aktywności stanowią podstawę wszystkich aplikacji na Androida.

Wiesz już, jak tworzyć aktywności i jak sprawić, by jedna aktywność uruchomiła drugą, używając intencji. Ale ***co tak naprawdę dzieje się za kulisami***? W tym rozdziale nieco dokładniej opiszemy **cykl życia aktywności**. Co się dzieje, kiedy aktywność **jest tworzona i usuwana**? Jakie metody są wywoływane, gdy aktywność jest **wyświetlana i pojawia się na ekranie**, a jakie, gdy aktywność **traci miejsce wprowadzania i jest ukrywana**? W jaki sposób można **zapisywać i odtwarzać stan aktywności**? Przeczytaj, by się dowiedzieć.



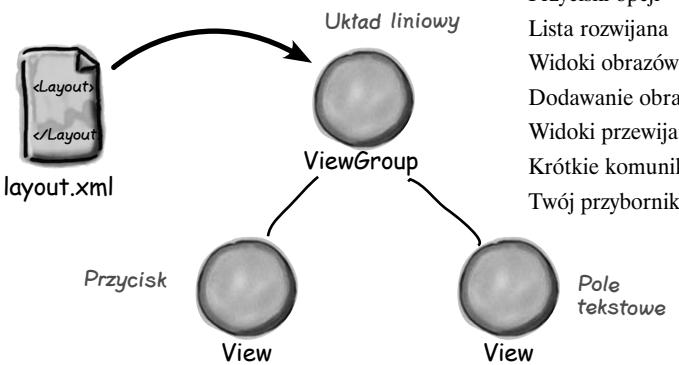
Widoki i grupy widoków

5 Podziwiaj widoki

Zobaczyłeś już, jak można rozmieszczać elementy GUI, używając układu LinearLayout, ale to jedynie wierzchołek góry logowej.

W tym rozdziale **nieco dokładniej** opiszemy *rzeczywisty sposób działania* układu liniowego. Przedstawimy także **`FrameLayout`**, nazywany również **układem ramki**, będący prostym układem używanym do rozmieszczania widoków jeden na drugim, i zabierzemy Cię na wycieczkę po **najważniejszych komponentach GUI** i **sposobach ich stosowania**. Pod koniec tego rozdziału przekonasz się, że choć wszystkie te układy i komponenty wyglądają nieco inaczej, to jednak ***mają ze sobą więcej wspólnego, niż można by przypuszczać***.

Układy FrameLayout pozwalają nakładać jedne widoki na inne. To bardzo przydatne w przypadkach, gdy chcemy stworzyć interfejs użytkownika, w którym na przykład tekst jest wyświetlany na tle obrazu.



6 Układy z ograniczeniami

Rozmieszczaj rzeczy w odpowiednich miejscach

Spójrzmy prawdzie w oczy: musisz wiedzieć, jak tworzyć piękne układy.

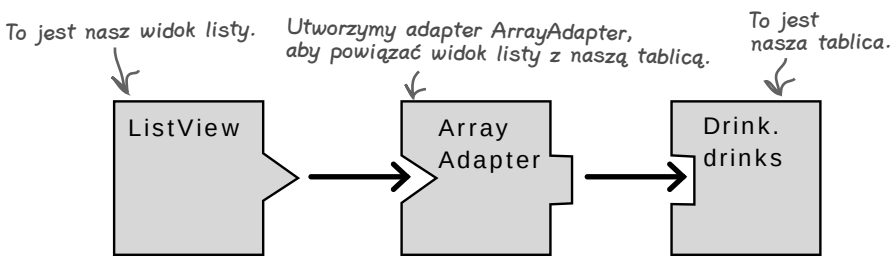
Jeśli piszesz aplikacje i chcesz, by inni ich używali, to musisz mieć pewność, że będą one **wyglądać dokładnie tak, jak sobie tego życzysz**. Jak na razie dowiedziałeś się jedynie, jak używać układów `LinearLayout` oraz `FrameLayout`, ale co zrobić, jeśli *projekt interfejsu aplikacji będzie bardziej złożony*? Aby pokazać Ci jak należy sobie radzić w takich sytuacjach, przedstawimy nowy układ dodany do Androida — układ z ograniczeniami (`ConstraintLayout`) — używany do ***wizualnego konstruowania układów na podstawie szkicu graficznego***. Pokażemy Ci także, jak **ograniczenia** pozwalają na określanie położenia widoków, w *sposób niezależny od wielkości i orientacji ekranu*. I w końcu dowiesz się też, jak oszczędzać czas, korzystając z możliwości **automatycznego przewidywania i dodawania ograniczeń**, jaką dysponuje Android Studio.

7 Widoki list i adaptery

Zorganizuj się

Chcesz wiedzieć, jaki jest najlepszy sposób na określenie struktury aplikacji?

Znasz już podstawowe elementy konstrukcyjne używane do tworzenia aplikacji, więc teraz nadszedł czas, **żebyś się lepiej zorganizował**. W tym rozdziale pokażemy Ci, jak możesz **przekształcić** zbiór pomysłów w **niesamowitą aplikację**. Zobaczysz, że **listy danych** mogą stać się kluczowym elementem projektu aplikacji i że **łączenie ich** może prowadzić do powstania aplikacji **łatwej w użyciu i zapewniającej ogromne możliwości**. Przy okazji zapoznasz się z **obiektami nasłuchującymi** i **adapterami**, dzięki którym Twoja aplikacja stanie się bardziej dynamiczna.

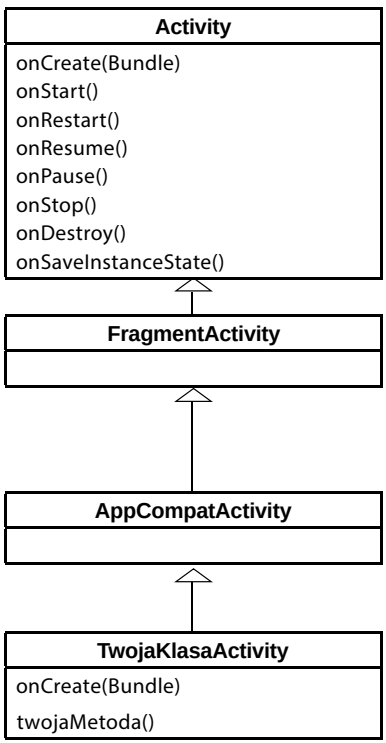


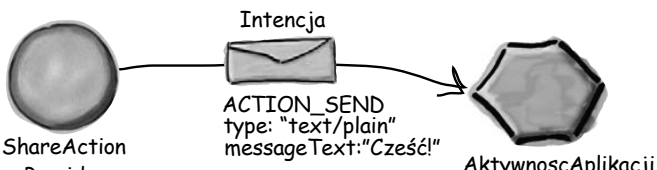
Biblioteki wsparcia i paski aplikacji

8 Na skróty

Każdy lubi chodzić na skróty.

Z tego rozdziału dowiesz się, jak korzystając z **pasków aplikacji**, wzbogacić aplikację o możliwość chodzenia na skróty. Pokażemy Ci, jak uruchamiać inne aktywności za pomocą *elementów akcji* dodawanych do pasków aplikacji, jak udostępniać treści innym aplikacjom, używając *dostawcy akcji współdzielenia*, oraz jak poruszać się w górę hierarchii aplikacji za pomocą *przycisku W górę* umieszczonego na pasku aplikacji. Jednocześnie przedstawimy Ci potężne **biblioteki wsparcia Androida**, mające kluczowe znaczenie dla zapewniania nowoczesnego wyglądu aplikacji na starszych wersjach systemu.



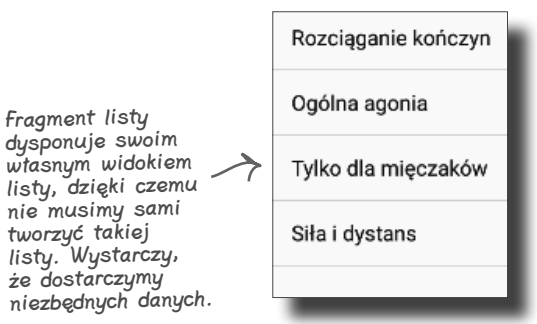


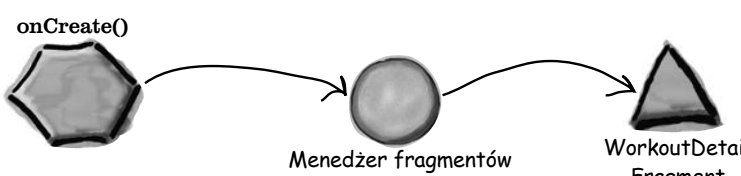
9 Fragmenty

Zadbaj o modularyzację

Wiesz już, jak tworzyć aplikacje, które działają tak samo niezależnie od tego, na jakim urządzeniu zostały uruchomione…

…ale co zrobić w przypadku, kiedy akurat chcesz, by aplikacja **wyglądała i działała inaczej** w zależności od tego, czy zostanie uruchomiona na *telefonie*, czy na *tablecie*? W tej sytuacji będziesz potrzebował **fragmentów**, modularnych komponentów, które mogą być **wielokrotnie używane w różnych aktywnościach**. Pokażemy, jak można tworzyć **proste fragmenty** oraz **fragmenty list**, jak **dodawać fragmenty do aktywności** oraz w jaki sposób zapewniać wzajemną **komunikację** pomiędzy fragmentami i aktywnościami.





10 Fragmenty dla większych interfejsów

Różne wielkości, różne interfejsy

Jak na razie uruchamialiśmy nasze aplikacje wyłącznie na urządzeniach z małymi ekranami.

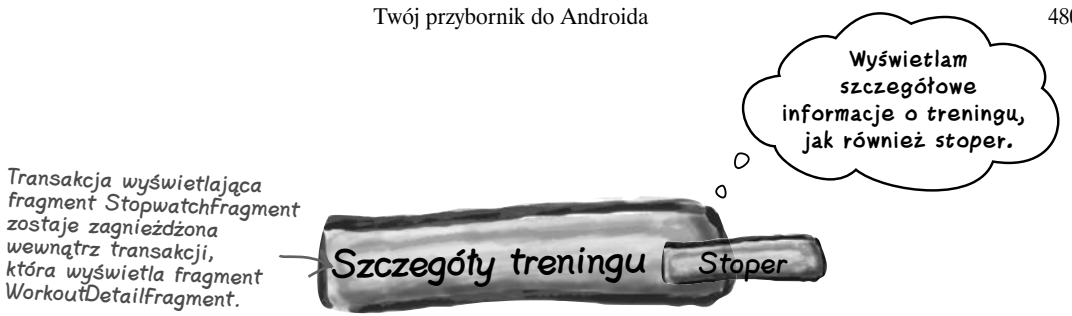
A co, jeśli użytkownicy będą mieli tablety? W tym rozdziale dowiesz się, w jaki sposób, dzięki zróżnicowaniu wyglądu i sposobu działania aplikacji w zależności od urządzenia, na jakim została ona uruchomiona, można projektować **elastyczne interfejsy użytkownika**. Pokażemy także, jak kontrolować działanie aplikacji po naciśnięciu przycisku *Wstecz*, prezentując przy okazji dwa nowe rozwiązania: **stos cofnięć** i **transakcje fragmentów**. I w końcu dowiesz się też, jak można **zapisywać i odtwarzać** stan fragmentów.

Fragmenty dynamiczne

11 Zagnieżdżanie fragmentów

Jak na razie dowiedziałeś się, jak można tworzyć i stosować fragmenty statyczne.

Ale co zrobić, jeśli zechcemy, by nasze fragmenty były nieco bardziej **dynamiczne**? Fragmenty dynamiczne mają wiele wspólnego z aktywnościami dynamicznymi, choć występują pomiędzy nimi pewne kluczowe różnice, z którymi będziemy musieli umieć odpowiednio postępować. W tym rozdziale dowiesz się, jak **przekształcać aktywności dynamiczne** na **działające fragmenty dynamiczne**. Dowiesz się także, jak korzystać z **transakcji fragmentów** w celu **zachowania stanu fragmentów**. I w końcu odkryjesz, jak można **zagnieżdżać jedne fragmenty w innych** oraz w jaki sposób **menedżer fragmentów podrzędnych** ułatwia zachowanie kontroli nad niesfornym stosem cofnięć.



Biblioteka wsparcia wzornictwa

12 Przeciągnięcie w prawo

Czy kiedykolwiek zastanawiałeś się, jak napisać aplikację z bogatym i ładnym interfejsem użytkownika?

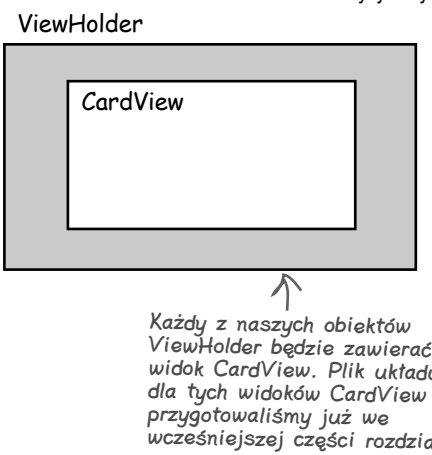
Dzięki udostępnieniu biblioteki wsparcia wzornictwa, **Android Design Support Library**, teraz znacznie łatwiej można już tworzyć aplikacje o intuicyjnym interfejsie użytkownika. W tym rozdziale przedstawimy najważniejsze informacje na ten temat. Dowiesz się, jak dodawać **karty**, dzięki którym użytkownicy będą mogli łatwiej poruszać się po aplikacji. Poznasz sposoby ***animowania pasków narzędzi***, by na żądanie mogły się zwijać lub przewijać. Nauczysz się także dodawać do aplikacji ***pływające przyciski akcji***, ułatwiające dostęp do najczęściej wykonywanych operacji. I w końcu przedstawimy Ci ***snackbar*** — mechanizm do wyświetlania krótkich, informacyjnych komunikatów, z którymi użytkownik może wchodzić w interakcje.

13 Widoki RecyclerView i CardView

Stosuj recykling

Przekonałeś się już, że kluczowym elementem większości aplikacji jest skromny widok listy.

Jednak w porównaniu z poznanymi wcześniej komponentami *Material Design* takie listy są bardzo proste. W tym rozdziale przedstawimy widok `RecyclerView` — bardziej zaawansowany typ widoku listy, który zapewnia *znacznie większą elastyczność* i *idealnie pasuje do etosu wzornictwa Material Desing*. Nauczysz się tworzyć **adaptery** dostosowane do używanych danych i dowiesz się, jak całkowicie odmieniać wygląd list, używając do tego *jedynie dwóch wierszy kodu*. Pokażemy Ci także, jak korzystać z **widoków kart** (ang. *card views*) *przestrzenną prezentację* danych zgodną z wytycznymi Material Design.



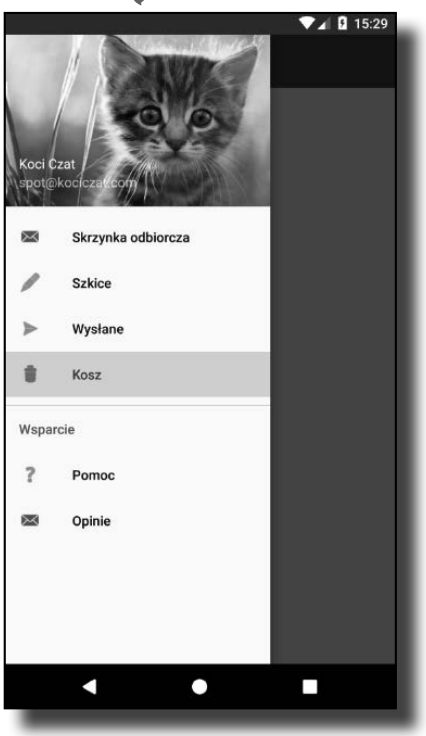
14 Szuflady nawigacyjne

Z miejsca na miejsce

Przekonałeś się już, w jaki sposób karty ułatwiają użytkownikom poruszanie się po aplikacji.

Jeśli jednak będziemy ich potrzebowali *bardzo dużo* lub jeśli *trzeba je rozdzielić na sekcje*, to **szuflada nawigacyjna** będzie Twoim nowym najlepszym przyjacielem. W tym rozdziale pokażemy Ci jak tworzyć szufladę nawigacyjną, która *wysuwa się z boku ekranu po jednym kliknięciu*. Dowiesz się, jak przygotować nagłówek takiej szuflady przy użyciu **widoku nawigacyjnego**, jak wypełnić ją **zestawem elementów o zadanej strukturze**, które pozwolą użytkownikom docierać do wszystkich głównych punktów aplikacji. I w końcu dowiesz się, jak przygotować **obiekt nasłuchujący widoku nawigacyjnego**, dzięki któremu szuflada będzie w stanie *reagować na najdelikatniejsze dotknięcia i przeciągnięcia*.

To jest aplikacja Koci Czat.

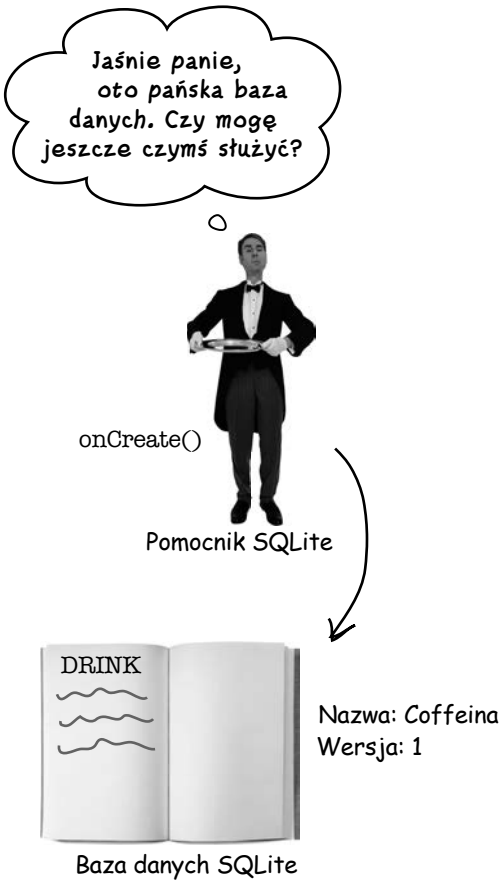


Bazy danych SQLite

15 Odpal bazę danych

Jeśli rejestrujesz najlepsze wyniki lub przesyłane komunikaty, to Twoja aplikacja będzie musiała przechowywać dane.

A w Androidzie dane są zazwyczaj bezpiecznie i trwale przechowywane w **bazach danych SQLite**. W tym rozdziale pokażemy Ci, jak **utworzyć bazę danych, dodawać do niej tabele, wypełnić ją wstępnie danymi**, a wszystko to za pomocą **pomocnika SQLite**. Dowiesz się też, w jaki sposób można bezproblemowo przeprowadzać **aktualizacje** struktury bazy danych i jak w razie konieczności wycofania zmian wrócić do jej **wcześniejszych wersji**.

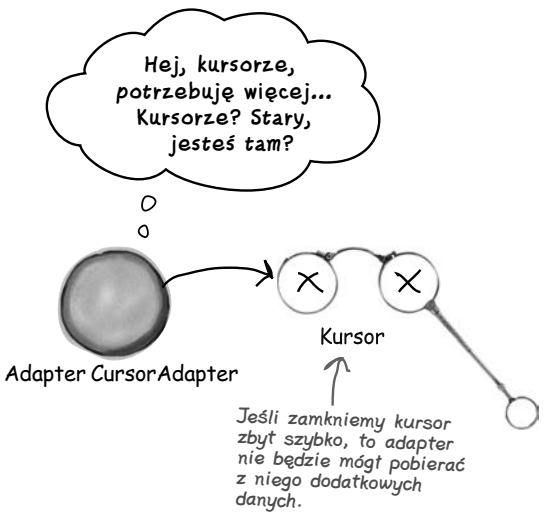


Proste kursory

16 Pobieranie danych

Jak łączysz swoje aplikacje z bazami danych SQLite?

Dotychczas dowiedziałeś się, jak tworzyć bazy danych, używając pomocnika SQLite. Kolejnym krokiem będzie uzyskanie dostępu do tych baz danych w aktywnościach. W tym rozdziale skoncentrujemy się na sposobach odczytywania danych bazy danych. Dowiesz się w nim, **jak używać kursora do odczytywania danych z bazy**. Nauczysz się także poruszać po kursorach i **uzyskiwać dostęp do umieszczonych w nich danych**. I w końcu dowiesz się, jak stosować **adaptery operujące na kursorach**, aby używać wspólnie kursorów i widoków list.

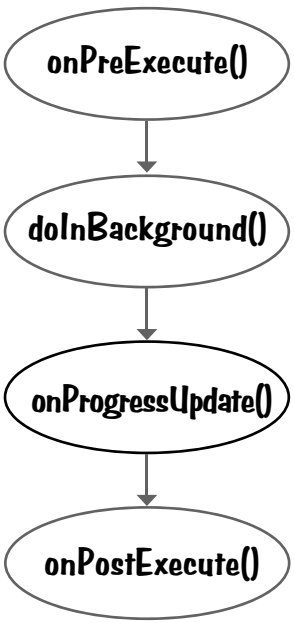


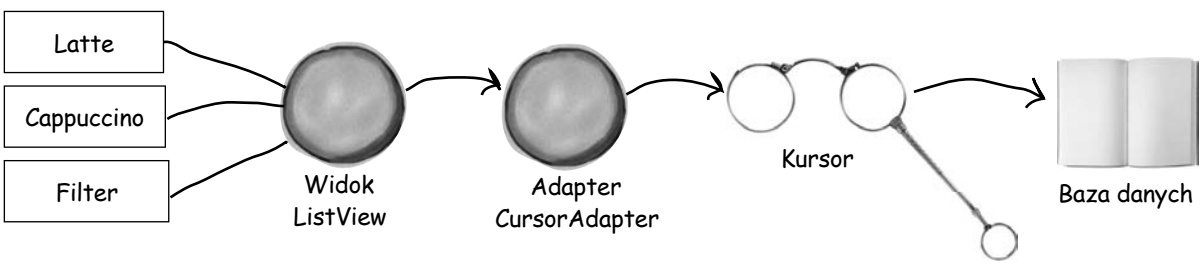
17 Kursory i zadania asynchroniczne

Pozostając w tle

Przeważająca większość aplikacji musi aktualizować swoje dane.

W poprzednim rozdziale dowiedziałeś się, jak pisać aplikacje, które odczytują dane z bazy danych SQLite. A co w przypadku, kiedy chcemy zaktualizować dane aplikacji? W tym rozdziale dowiesz się, jak sprawić, by aplikacja **reagowała na poczynania użytkownika** i **aktualizowała informacje zapisane w bazie danych**. Dowiesz się także, jak **odświeżać wyświetlone dane** po ich aktualizacji. I w końcu przekonasz się, że pisanie **wydajnego, wielowątkowego kodu** przy użyciu klasy **`AsyncTask`** pozwala zapewnić odpowiednią szybkość działania aplikacji.



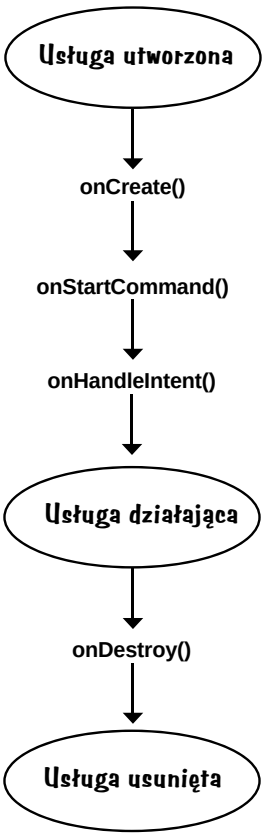


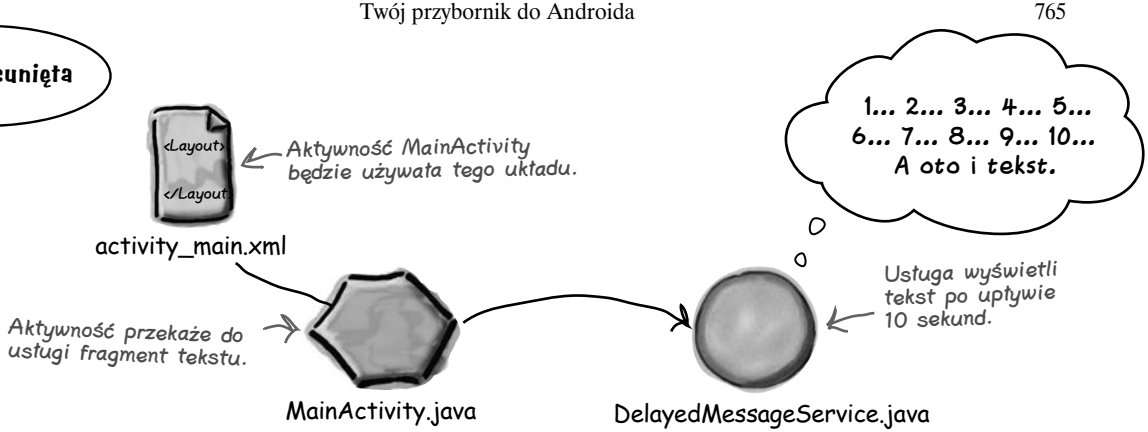
18 Usługi uruchomione

Do usług

Są operacje, które będziemy chcieli wykonywać niezależnie od tego, która aplikacja jest widoczna na ekranie.

Na przykład jeśli rozpoczniesz pobieranie pliku, to zapewne *nie będziesz chciał, by proces pobierania był przerywany, kiedy przełączysz się do innej aplikacji.* W tym rozdziale przedstawimy **usługi uruchomione**, komponenty, które *wykonują operacje w tle.* Dowiesz się, jak stworzyć usługę uruchomioną, używając klasy `IntentService`, oraz w jaki sposób cykl życia takiej usługi jest powiązany z cyklem życia aktywności. W międzyczasie odkryjesz, jak można **rejestrować komunikaty** i zadbać o to, by *użytkownik zawsze był dobrze poinformowany*, wykorzystując do tego **usługę powiadomień**.



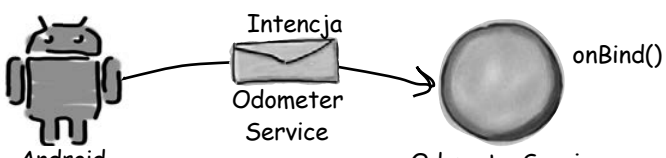


Usługi powiązane i uprawnienia

19 Powiązane ze sobą

Usługi uruchomione są doskonałe do wykonywania operacji w tle, ale co zrobić, gdy potrzebujemy usługi, która będzie bardziej interaktywna?

W tym rozdziale dowiesz się, jak tworzyć **usługi powiązane**, czyli *usługi kolejnego typu, z którymi aktywności mogą wchodzić w interakcje*. Dowiesz się także, jak ***powiązać*** usługę, kiedy to będzie potrzebne, oraz jak ją ***odłączyć*** w celu oszczędzania zasobów, kiedy nie będzie już potrzebna. Przy okazji nauczysz się korzystać z **usług lokalizacyjnych** Androida, by *pobierać z odbiornika GPS urządzenia aktualne informacje o położeniu*. I w końcu dowiesz się, jak używać **modelu uprawnień Androida**, w tym także jak obsługiwać *żądania przydzielenia uprawnień zgłaszane podczas działania aplikacji*.



Układy względne i układy siatki

A

Poznaj krewnych

Istnieją jeszcze dwa inne układy często stosowane w Androidowie.

W tej książce koncentrowaliśmy się na stosowaniu prostych *układów liniowych i układów ramek*, jak również przedstawiliśmy *nowy układ z ograniczeniami*. Jednak istnieją także dwa inne rodzaje układów, które chcielibyśmy Ci zaprezentować: **układ względny** oraz **układ siatki**. W większości zostały one zastąpione układem z ograniczeniami, niemniej jednak i tak pozostaje wiele okazji do ich stosowania, dlatego też przypuszczamy, że będą one używane jeszcze przez ładnych parę lat.

Gradle

B

Program do budowy Gradle

Większość aplikacji na Androida jest budowana przy użyciu programu narzędziowego o nazwie Gradle.

Program **Gradle** działa za kulisami: odnajduje i pobiera biblioteki, kompiluje i wdraża kod, wykonuje testy, czyści fugi i tak dalej... W większości przypadków możemy sobie nawet nie zdawać sprawy, że Gradle istnieje, gdyż Android Studio udostępnia do jego obsługi graficzny interfejs użytkownika. Jednak czasami warto zajrzeć mu pod maskę i **ręcznie go podregulować**.
W tym dodatku pokażemy Ci niektóre spośród wielu talentów i umiejętności programu Gradle.

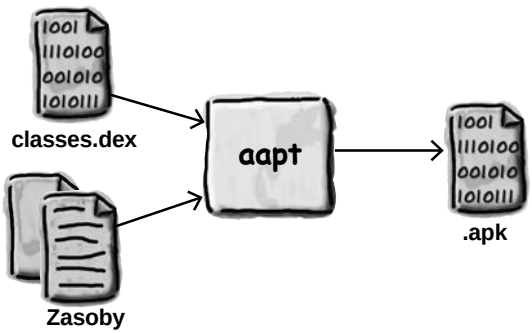
ART

C

Środowisko uruchomieniowe Androida

Czy kiedykolwiek zastanawiałeś się, jak to się dzieje, że aplikacje na Androida mogą działać na tak wielu rodzajach urządzeń?

Aplikacje na Androida są wykonywane w wirtualnej maszynie nazywanej środowiskiem uruchomieniowym Androida (ART — *Android runtime*), a nie w wirtualnej maszynie Javy firmy Oracle. Oznacza to, że będą one szybciej uruchamiane na niewielkich urządzeniach o niewielkiej mocy obliczeniowej i będą na nich działać bardziej efektywnie. W tym dodatku dowiesz się, jak działa ART.



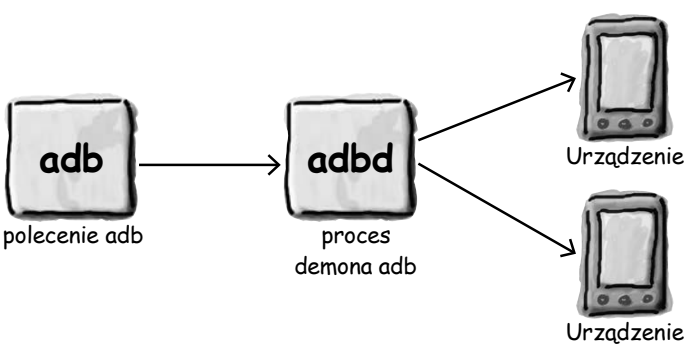
ADB

D

Android Debug Bridge

W tej książce skoncentrowaliśmy się na zaspokajaniu wszystkich potrzeb związanych z pisaniem aplikacji na Androida z wykorzystaniem IDE.

Zdarzają się jednak sytuacje, w których zastosowanie narzędzi obsługiwanych z poziomu wiersza poleceń jest po prostu przydatne. Mamy tu na myśli na przykład przypadki, gdy Android Studio nie jest w stanie zauważyć urządzenia z Androidem, choć my wiemy, że ono istnieje. W tym rozdziale przedstawimy ***Android Debug Bridge*** (w skrócie ***ADB***) — obsługiwany z poziomu wiersza poleceń program narzędziowy, którego można używać do komunikacji z emulatorem lub z rzeczywistymi urządzeniami zaopatrzonymi w Androida.



Emulator

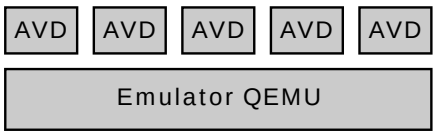
E

Przyspieszanie emulatora

Czy miałeś kiedyś wrażenie, że cały swój czas spędzasz, czekając na emulator?

Nie ma najmniejszych wątpliwości co do tego, że emulator Androida jest bardzo przydatny. Dzięki niemu możemy sprawdzić, jak nasza aplikacja będzie działała na urządzeniach innych niż te, do których mamy fizyczny dostęp. Niekiedy jednak można odnieść wrażenie, że emulator działa... wolno. W tym dodatku wyjaśnimy, dlaczego tak się dzieje. Ale to nie wszystko, damy Ci bowiem także kilka wskazówek, jak **przyspieszyć jego działanie**.

Wszystkie wirtualne urządzenia z Androidem działają na emulatorze nazywanym QEMU. →



Pozostałości

F

Dziesięć najważniejszych zagadnień (których nie opisaliśmy)

Nawet po tym wszystkim, co opisaliśmy w tej książce, wciąż pozostaje wiele innych interesujących zagadnień.

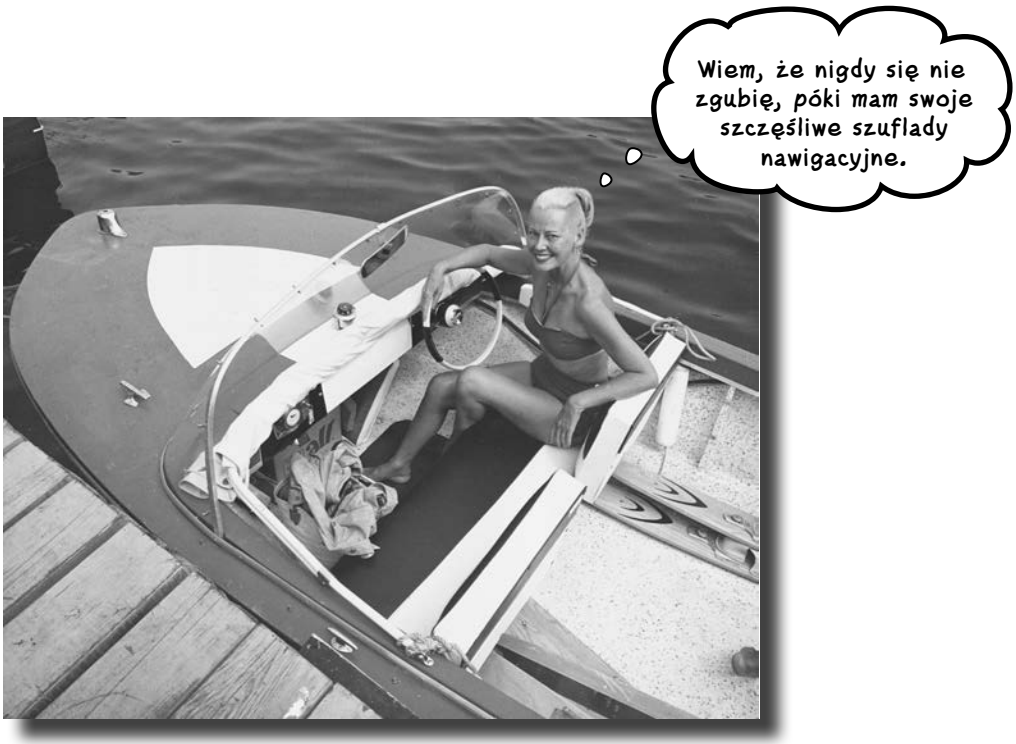
Jest jeszcze kilka dodatkowych spraw, o których musisz się dowiedzieć. Czulibyśmy się nie w porządku, gdybyśmy je pominęli, a jednocześnie chcieliśmy oddać w Twoje ręce książkę, którą dasz radę podnieść bez intensywnego treningu na siłowni. Dlatego zanim odłożysz tę książkę, przeczytaj **kilka dodatkowych zagadnień opisanych w tym dodatku**.

S

Skorowidz

14. Szuflady nawigacyjne

Z miejsca na miejsce

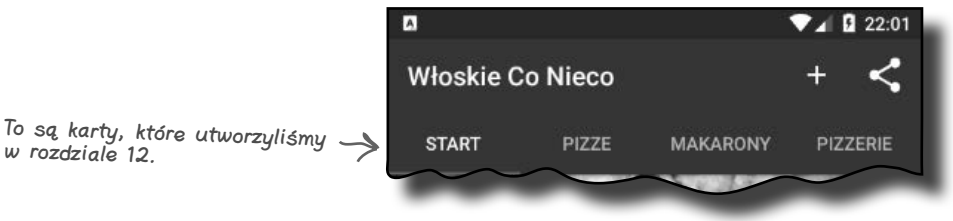


Zobaczyłeś się już, w jaki sposób karty ułatwiają użytkownikom poruszanie się po aplikacji.

Jeśli jednak będziemy ich potrzebowali *bardzo dużo* lub jeśli *trzeba je rozdzielić na sekcje*, to **szuflada nawigacyjna** będzie Twoim nowym najlepszym przyjacielem. W tym rozdziale pokażemy Ci, jak tworzyć szufladę nawigacyjną, która *wysuwa się z boku ekranu po jednym kliknięciu*. Dowiesz się, jak przygotować nagłówek takiej szuflady przy użyciu **widoku nawigacyjnego**, jak wypełnić ją **zestawem elementów o zadanej strukturze**, które pozwolą użytkownikom docierać do wszystkich głównych punktów aplikacji. I w końcu dowiesz się, jak przygotować **obiekt nasłuchujący widoku nawigacyjnego**, dzięki któremu szuflada będzie w stanie *reagować na najdelikatniejsze dotknięcia i przeciągnięcia*.

Widoki kart zapewniają łatwą nawigację...

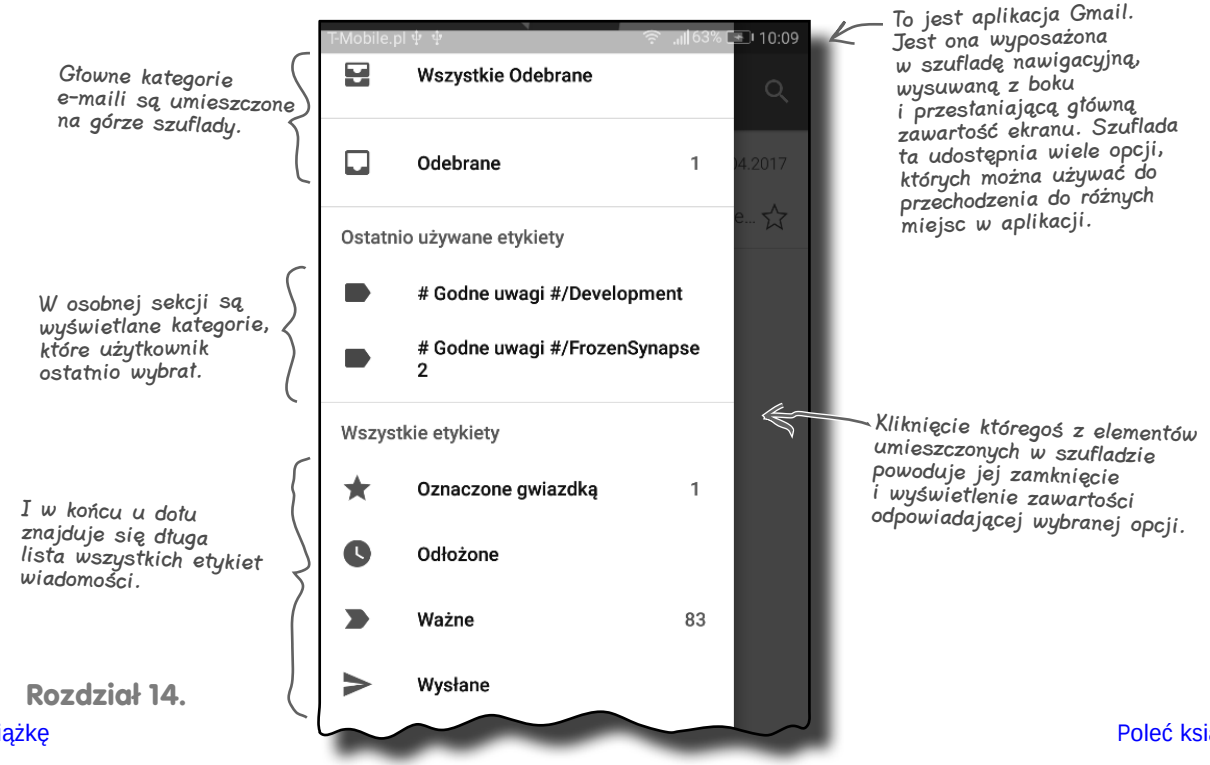
W rozdziale 12. przedstawiliśmy układ kart jako sposób ułatwiania użytkownikom poruszania się po aplikacji. W tym rozdziale dodaliśmy do naszej aplikacji *Włoskie Co Nieco* ekran startowy oraz karty dla wszystkich głównych kategorii: *Pizze*, *Makarony* oraz *Pizzerie*:



Układy kart świetnie się sprawdzają w sytuacjach, gdy mamy niewielką liczbę ekranów kategorii, znajdujących się na tym samym poziomie hierarchii aplikacji. Ale co zrobić w przypadkach, gdy chcemy używać dużej liczby kart bądź też grupować karty w sekcje?

...ale szuflady nawigacyjne pozwalają wyświetlać więcej opcji

Jeśli chcemy zapewnić użytkownikom możliwość wybierania spośród dużej liczby opcji bądź też grupowania ich w sekcje, to lepszym rozwiązaniem niż karty mogą się okazać **szuflady nawigacyjne**. Szuflada nawigacyjna to panel wysuwany spoza pionowej krawędzi ekranu i zawierający odnośniki do innych miejsc aplikacji; pozwala on także grupować te odnośniki w sekcje. Na przykład aplikacja Gmail używa szuflady nawigacyjnej zawierającej sekcje takie jak kategorie e-maili oraz wszystkie etykiety:



Planujemy utworzenie szuflady nawigacyjnej w nowej aplikacji pocztowej

W tym rozdziale planujemy utworzenie szuflady nawigacyjnej w nowej aplikacji do obsługi poczty elektronicznej, którą nazwiemy Koci Czat. Nasza szuflada nawigacyjna będzie zawierać nagłówek (prezentujący obrazek i trochę tekstu) oraz zestaw opcji. Podstawowe opcje będą zapewniać dostęp do skrzynki odbiorczej, szkiców, wiadomości wysłanych oraz kosza. Dostępna będzie także odrębna sekcja wsparcia, z odnośnikami do systemu pomocy oraz ekranu do przesyłania informacji zwrotnych.

Szuflada nawigacyjna składa się z kilku różnych komponentów. Przedstawimy je wszystkie na następnej stronie.

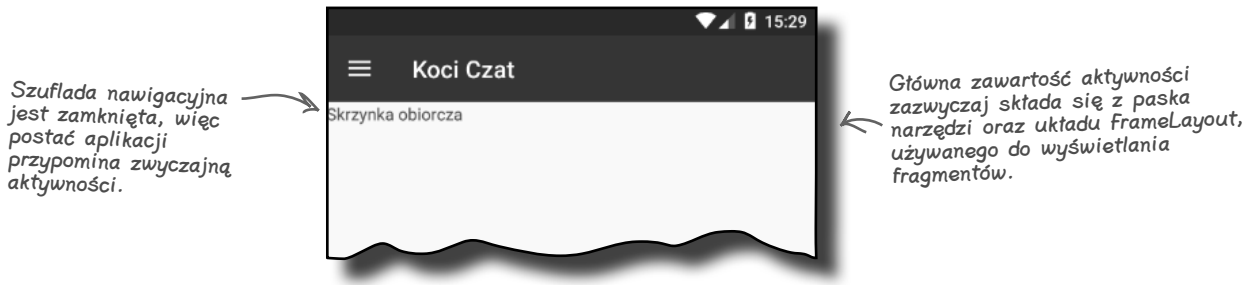
Szuflady nawigacyjne rozmontowane na czynniki pierwsze

Szufladę nawigacyjną tworzymy poprzez dodanie do układu aktywności **układu szuflady**. Definiuje on szufladę, którą można otwierać i zamykać, a dodatkowo musi ona zawierać dwa widoki:

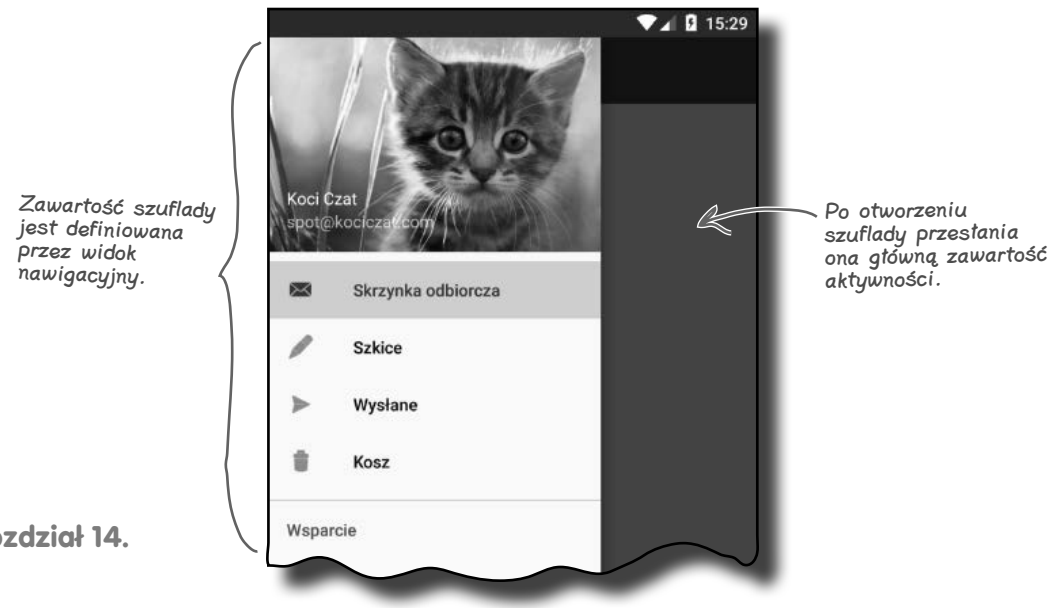
1. **Widok do prezentacji głównej zawartości aplikacji.**
   Zazwyczaj jest to układ zawierający pasek narzędzi oraz układ ramki, którego można używać do wyświetlania fragmentów.

2. **Widok do prezentacji zawartości szuflady.**
   Zazwyczaj jest to widok nawigacyjny, kontrolujący przeważającą większość możliwości funkcjonalnych szuflady.

Gdy szuflada jest zamknięta, jej układ przypomina układ standardowej aktywności — prezentuje wyłącznie układ głównej zawartości:



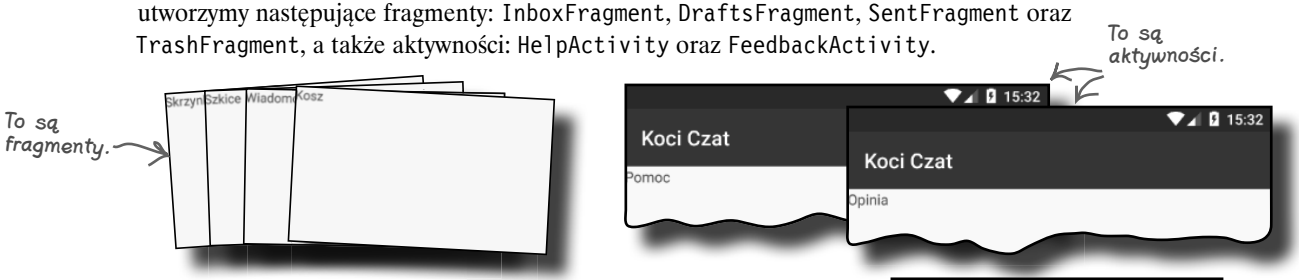
Po otworzeniu szuflady nawigacyjnej zostaje ona wysunięta w celu wyświetlenia jej zawartości i częściowo przesłania główną zawartość aktywności. Zawartość szuflady zazwyczaj składa się z widoku nawigacyjnego prezentującego obrazek stanowiący nagłówek oraz listę opcji. Kliknięcie jednej z tych opcji powoduje uruchomienie nowej aktywności bądź też wyświetlenie odpowiedniego fragmentu w układzie `FrameLayout` aktywności.



Oto co zamierzamy zrobić

Zamierzamy zaimplementować szufladę nawigacyjną w aplikacji KociCzat. Cały proces będzie się składać z czterech głównych etapów:

1. **Utworzenia prostych fragmentów i aktywności dla treści aplikacji.**
Chcemy, by po kliknięciu jednej z opcji dostępnych w szufladzie nawigacyjnej został wyświetlony odpowiedni fragment lub została uruchomiona odpowiednia aktywność. Na potrzeby aplikacji utworzymy następujące fragmenty: `InboxFragment`, `DraftsFragment`, `SentFragment` oraz `TrashFragment`, a także aktywności: `HelpActivity` oraz `FeedbackActivity`.



2. **Utworzenia nagłówka szuflady nawigacyjnej.**
Na potrzeby nagłówka szuflady nawigacyjnej utworzymy plik układu *nav_header.xml*; będzie on zawierał obrazek i tekst.

3. **Utworzenia opcji szuflady nawigacyjnej.**
W celu określenia opcji wyświetlanych w szufladzie przygotujemy menu *menu_nav.xml*.

4. **Utworzenia szuflady nawigacyjnej.**
Szufladę nawigacyjną dodamy do głównej aktywności aplikacji i zadbamy o wyświetlanie jej nagłówka i opcji. Następnie napiszemy kod, który będzie kontrolować działanie szuflady.

A zatem zaczynajmy.

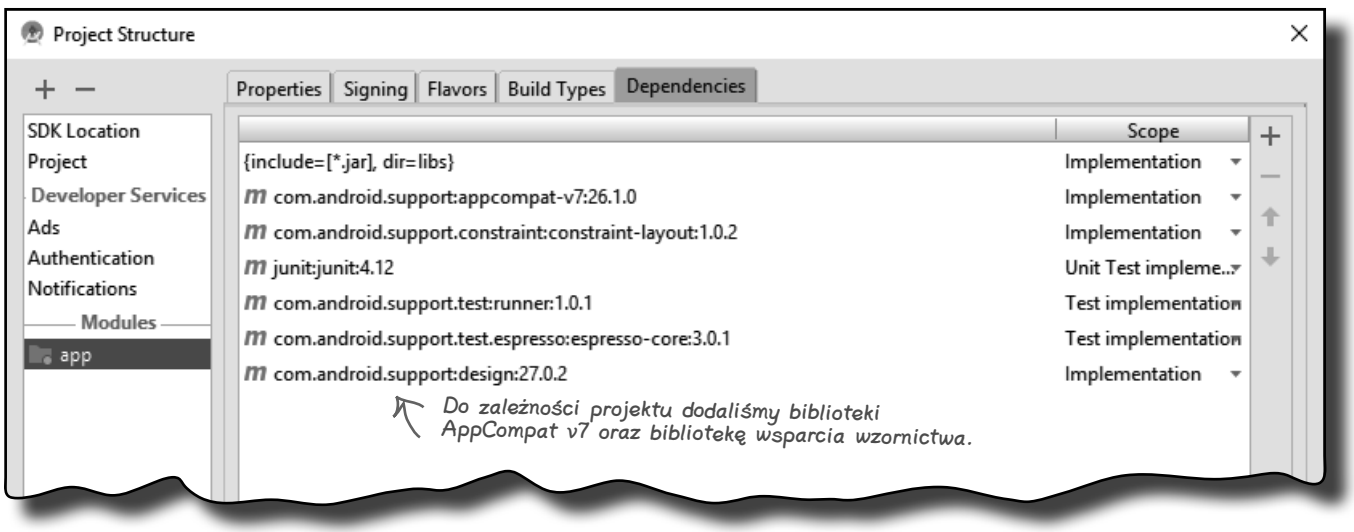
Utworzenie projektu Koci Czat

- → ☐ **Fragmenty i aktywności**
- ☐ **Nagłówek**
- ☐ **Opcje**
- ☐ **Szuflada**

Zanim zaczniemy, musimy utworzyć nowy projekt na potrzeby prac nad aplikacją. Utwórz zatem nowy projekt aplikacji na Androida, nadaj jej nazwę `KociCzat` i użyj domeny firmowej `hfad.com`, dzięki czemu pakiet aplikacji będzie miał nazwę `com.hfad.kociczat`. Jako minimalny poziom SDK wybierz API poziomu 19., dzięki czemu aplikacja będzie działać na większości urządzeń. Podczas tworzenia aplikacji dodaj do niej pustą aktywność, nadaj jej nazwę `MainActivity`, a używany przez nią plik układu nazwij `activity_main`. **Koniecznie zaznacz pole wyboru *Backwards Compatibility (AppCompat)*.**

Dodaj bibliotekę AppCompat v7 oraz bibliotekę wsparcia wzornictwa

W aplikacji będziemy używać komponentów i motywów z biblioteki wsparcia AppCompat v7 oraz z biblioteki wsparcia wzornictwa, dlatego musimy dodać je jako zależności naszego projektu. W tym celu wybierz z menu głównego Android Studio opcję *File/Project Structure*, kliknij moduł *app* i przejdź na kartę *Dependencies*. Po wyświetleniu karty z zależnościami projektu kliknij przycisk „+”, umieszczony u dołu lub z prawej strony okna dialogowego. Następnie wybierz opcję *Library Dependency* i z listy dostępnych bibliotek wybierz *Design Library*. Te same czynności powtórz, by dodać bibliotekę AppCompat v7, o ile Android Studio nie dodało jej automatycznie podczas tworzenia projektu. W końcu kliknij przycisk *OK*, aby zapisać zmiany.



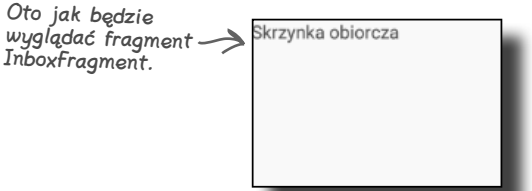
Teraz zajmiemy się przygotowaniem czterech prostych fragmentów dla: skrzynki odbiorczej, szkiców, wiadomości wysłanych oraz kosza. Zastosujemy je w dalszej części rozdziału, kiedy napiszemy kod obsługujący działanie szuflady nawigacyjnej.

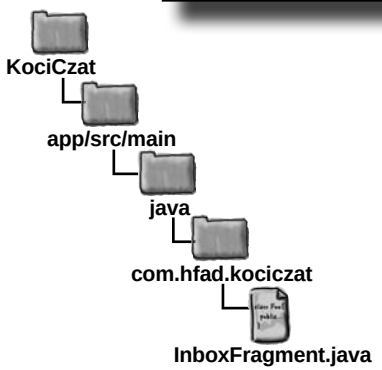
Utworzenie fragmentu InboxFragment

- → ☐ **Fragmenty i aktywności**
- ☐ **Nagłówek**
- ☐ **Opcje**
- ☐ **Szuflada**

Fragment `InboxFragment` będziemy wyświetlać, kiedy użytkownik kliknie opcję *Skrzynka odbiorcza* w szufladzie nawigacyjnej. Zaznacz zatem pakiet `com.hfad.kociczat` w katalogu *app/src/main/java* i wybierz z menu głównego opcję *File/New.../Fragment/Fragment (Blank)*. Tworzonemu fragmentowi nadaj nazwę `InboxFragment`, a jego plikowi układu nazwę `fragment_inbox`. Następnie zaktualizuj kod w pliku *InboxFragment.java*, by był taki sam jak ten przedstawiony poniżej:

Oto jak będzie wyglądać fragment InboxFragment.



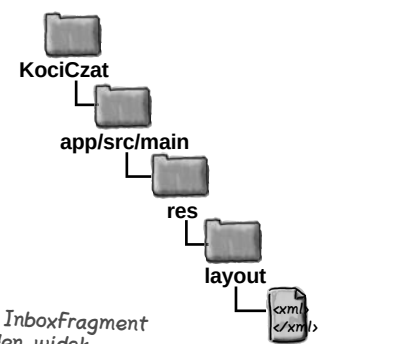



```
package com.hfad.kociczat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class DraftsFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_drafts, container, false);
    }
}
```

Wszystkie fragmenty używają klasy Fragment z biblioteki wsparcia.

A oto kod pliku *fragment_inbox.xml* (zaktualizuj swoją wersję tego pliku, tak by była identyczna z naszą):

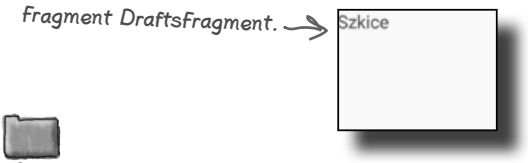
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.kociczat.InboxFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Skrzynka odbiorcza" />
</LinearLayout>
```

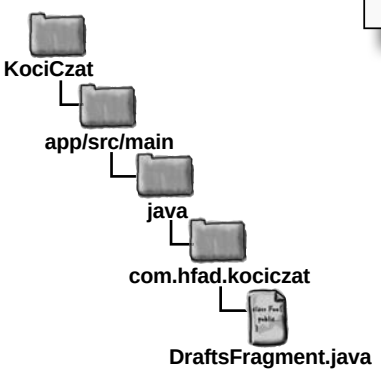


Układ fragmentu InboxFragment zawiera tylko jeden widok TextView. Dodaliśmy go jedynie po to, aby można było łatwo określić, kiedy będzie on wyświetlany.

Utworzenie fragmentu DraftsFragment

Fragment `DraftsFragment` będziemy wyświetlać, kiedy użytkownik kliknie opcję *Szkice* w szufladzie nawigacyjnej. Zaznacz zatem pakiet `com.hfad.kociczat` w katalogu *app/src/main/java* i wybierz z menu głównego opcję *File/New.../Fragment/Fragment (Blank)*. Tworzonemu fragmentowi nadaj nazwę `DraftsFragment`, a jego plikowi układu nazwę `fragment_drafts`. Następnie zaktualizuj kod w pliku *DraftsFragment.java*, by był taki sam jak ten przedstawiony poniżej:

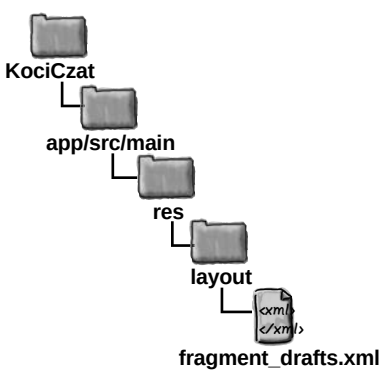





```
package com.hfad.kociczat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class DraftsFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_drafts, container, false);
    }
}
```

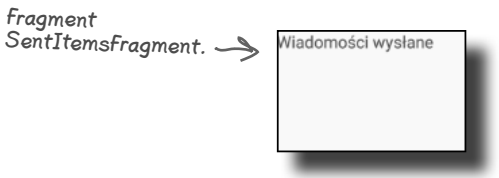
A oto kod pliku *fragment_drafts.xml* (zaktualizuj swoją wersję tego pliku, tak by była identyczna z naszą):

```
<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"
    tools:context="com.hfad.kociczat.DraftsFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Szkice" />
</LinearLayout>
```

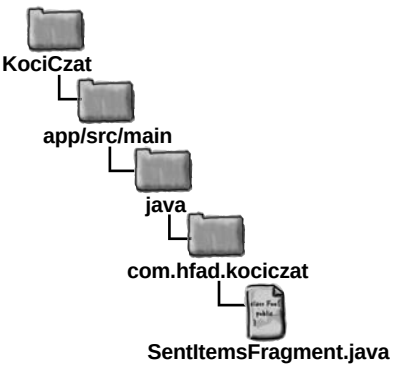


Utworzenie fragmentu SentItemsFragment

Fragment `SentItemsFragment` będziemy wyświetlać, kiedy użytkownik kliknie opcję *Wysłane* w szufladzie nawigacyjnej. Zaznacz zatem pakiet `com.hfad.kociczat` w katalogu *app/src/main/java* i wybierz z menu głównego opcję *File/New.../Fragment/Fragment (Blank)*. Tworzonemu fragmentowi nadaj nazwę `SentItemsFragment`, a jego plikowi układu nazwę `fragment_sent_items`. Następnie zaktualizuj kod w pliku *SentItemsFragment.java*, by był taki sam jak ten przedstawiony poniżej:

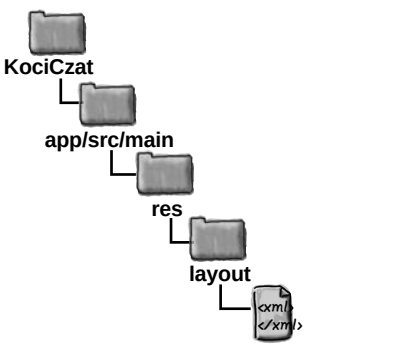



```
package com.hfad.kociczat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class SentItemsFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_sent_items, container, false);
    }
}
```



A oto kod pliku *fragment_sent_items.xml* (zaktualizuj swoją wersję tego pliku, tak by była identyczna z naszą):

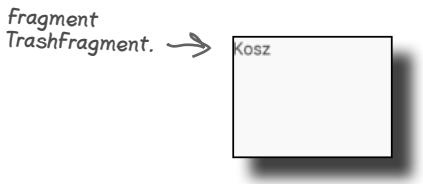
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.kociczat.SentItemsFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Wiadomości wysłane" />
</LinearLayout>
```

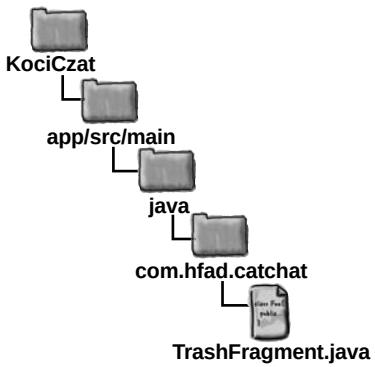


Utworzenie fragmentu TrashFragment

- → ☐ **Fragmenty i aktywności**
- ☐ **Nagłówek**
- ☐ **Opcje**
- ☐ **Szuflada**

Fragment `TrashFragment` będziemy wyświetlać, kiedy użytkownik kliknie opcję *Kosz* w szufladzie nawigacyjnej. Zaznacz zatem pakiet `com.hfad.kociczat` w katalogu *app/src/main/java* i wybierz z menu głównego opcję *File/New.../Fragment/Fragment (Blank)*. Tworzonemu fragmentowi nadaj nazwę `TrashFragment`, a jego plikowi układu nazwę `fragment_trash`. Następnie zaktualizuj kod w pliku *TrashFragment.java*, by był taki sam jak ten przedstawiony poniżej:

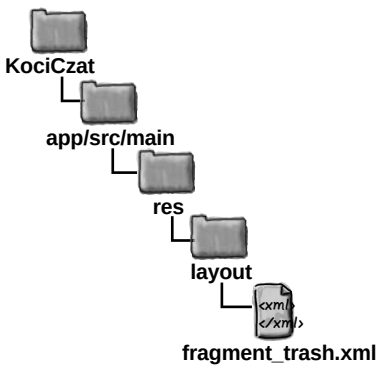





```
package com.hfad.kociczat;

import android.os.Bundle;
import android.support.v4.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class TrashFragment extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
                             Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_trash, container, false);
    }
}
```

Teraz zastąp także kod w pliku *fragment_trash.xml*:

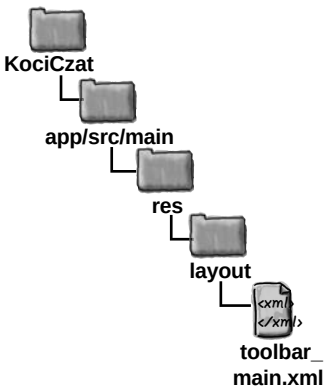
```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.kociczat.TrashFragment">

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Kosz" />
</LinearLayout>
```



W ten sposób utworzyliśmy już wszystkie fragmenty, których będziemy potrzebować. W kolejnym kroku zajmiemy się przygotowaniem paska narzędzi, który będziemy mogli wykorzystać w aktywnościach.

Przygotowanie układu paska narzędzi

Teraz zajmiemy się przygotowaniem paska narzędzi. Jego kod zapiszemy w odrębnym pliku układu, tak byśmy mogli łatwo dołączyć go do układu każdej z aktywności (nimi zajmiemy się w dalszej kolejności). W eksploratorze Android Studio przejdź do widoku *Project*, wybierz z katalogu *app/src/res/main/layout*, a następnie wybierz z menu opcję *File/New/Layout resource file*. Kiedy zostaniesz poproszony o podanie nazwy, wpisz `toolbar_main`. W końcu kliknij przycisk *OK*.



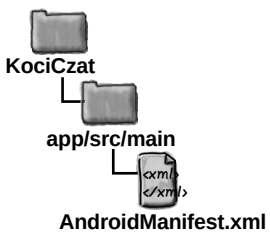
Kiedy wykonasz powyższe czynności, otwórz plik *toolbar_main.xml* i zastąp w nim kod wygenerowany przez Android Studio kodem zamieszczonym poniżej:

To jest ten sam kod paska zadań, którego używaliśmy w poprzednich rozdziałach.

```
<android.support.v7.widget.Toolbar
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="?attr/actionBarSize"
    android:background="?attr/colorPrimary"
    android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar" />
```

Zanim będziemy mogli użyć tego paska narzędzi w którejkolwiek z naszych aktywności, musimy zmienić wykorzystywany w nich motyw. Odpowiednie zmiany należy wprowadzić w zasobie stylów aplikacji.

Otwórz plik manifestu, *AndroidManifest.xml*, i upewnij się, że wartością atrybutu `theme` jest `"@style/AppTheme"`. Może się zdarzyć, że Android Studio samo ustawi wartość tego atrybutu; jeśli jednak tak się nie stanie, to będziesz musiał zmodyfikować ją samodzielnie, tak jak pokazaliśmy w poniższym przykładzie:




```
<?xml version="1.0" encoding="utf-8"?>
<manifest ...>
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            ...
        </activity>
    </application>
</manifest>
```

Może się zdarzyć, że Android Studio samo ustawi tę wartość.

Aktualizacją stylu zajmiemy się na następnej stronie.

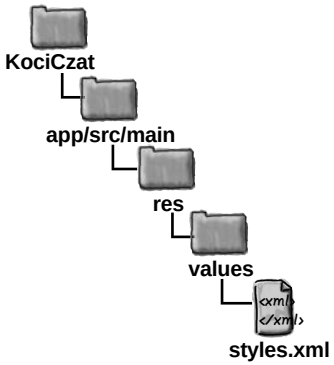
Aktualizacja motywu aplikacji

- ☐ **Fragmenty i aktywności**
- ☐ **Nagłówek**
- ☐ **Opcje**
- ☐ **Szuflada**

W kolejnym kroku zmodyfikujemy styl `AppTheme`, tak by korzystał on z motywu `"Theme.AppCompat.Light.NoActionBar"`. Poza tym zmienimy parę kolorów używanych w domyślnym motywie.

W pierwszej kolejności otwórz katalog *app/src/main/res/values* i sprawdź, czy Android Studio utworzyło w nim plik o nazwie *styles.xml*. Jeśli taki plik nie istnieje, to go utwórz. W tym celu zaznacz katalog *values* i wybierz z menu opcję *File/New.../Values resource file*. Jako nazwę pliku wpisz `styles` i kliknij przycisk *OK*.

Następnie zaktualizuj zawartość pliku *styles.xml*, by była identyczna z poniższym fragmentem kodu:

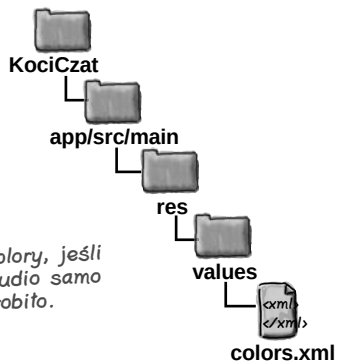



```
<resources>
    <style name="AppTheme" parent="Theme.AppCompat.Light.NoActionBar">
        <item name="colorPrimary">@color/colorPrimary</item>
        <item name="colorPrimaryDark">@color/colorPrimaryDark</item>
        <item name="colorAccent">@color/colorAccent</item>
    </style>
</resources>
```

Ten motyw usuwa domyślny pasek aplikacji (zastępujemy go paskiem narzędzi).

Android Studio mogło samo określić te kolory.

Styl `AppTheme` używa zasobów kolorów, a te powinny zostać zdefiniowane w pliku *colors.xml*. W pierwszej kolejności upewnij się, że Android Studio utworzyło ten plik w katalogu *app/src/main/res/values* (a jeśli go nie utworzyło, to zrób to samodzielnie). Następnie zapisz w pliku *colors.xml* poniższy fragment kodu:




```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="colorPrimary">#3F51B5</color>
    <color name="colorPrimaryDark">#303F9F</color>
    <color name="colorAccent">#FF4081</color>
</resources>
```

Dodaj te kolory, jeśli Android Studio samo tego nie zrobiło.

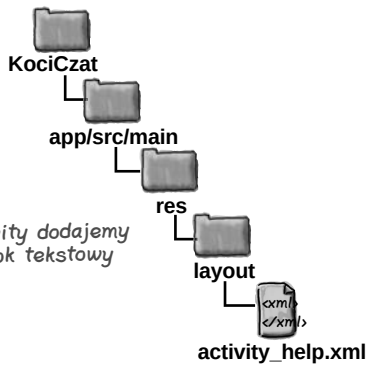
Skoro już przygotowaliśmy style pozwalające na zastosowanie paska narzędzi, utworzymy dwie aktywności: jedną na potrzeby systemu pomocy aplikacji oraz drugą do podawania opinii. Aktywności te będziemy uruchamiać, kiedy użytkownik wybierze odpowiednie opcje z szuflady nawigacyjnej.

Utworzenie aktywności HelpActivity

Zaczniemy od utworzenia aktywności `HelpActivity`. Zaznacz pakiet `com.hfad.kociczat`, a następnie wybierz z menu opcję *File/New*. Potem wybierz opcję utworzenia pustej aktywności, nadaj jej nazwę `HelpActivity`, a używanemu przez nią plikowi układu nazwę `activity_help`. Upewnij się, że klasa aktywności będzie należeć do pakietu `com.hfad.kociczat` oraz że **jest zaznaczone pole wyboru *Backwards Compatibility (AppCompat)***. Następnie zaktualizuj swoją wersję pliku *activity_help.xml*, tak by zawierał poniższy kod:


```
<?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"
    tools:context="com.hfad.kociczat.HelpActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Pomoc" />
</LinearLayout>
```

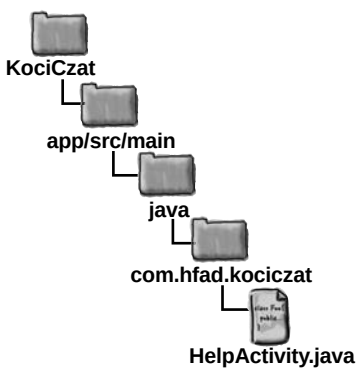
Do aktywności HelpActivity dodajemy pasek narzędzi oraz widok tekstowy ze słowem „Pomoc".



Teraz zaktualizuj plik *HelpActivity.java*, zapisując w nim poniższy kod:

```
package com.hfad.kociczat;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;

public class HelpActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_help);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
}
```

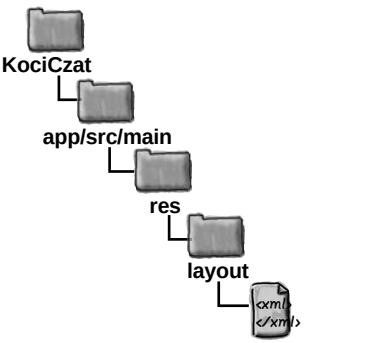
Używamy motywu AppCompat, więc aktywność musi dziedziczyć po klasie AppCompatActivity.



Utworzenie aktywności FeedbackActivity

Teraz zaznacz pakiet `com.hfad.kociczat` i wybierz z menu opcję *File|New*. Następnie wybierz opcję utworzenia pustej aktywności, nadaj jej nazwę `FeedbackActivity`, a używanemu przez nią plikowi układu nazwę `activity_feedback`. Upewnij się, że klasa aktywności będzie należeć do pakietu `com.hfad.kociczat` oraz że **jest zaznaczone pole wyboru *Backwards Compatibility (AppCompat)***. Następnie zaktualizuj swoją wersję pliku *activity_feedback.xml*, tak by zawierał poniższy kod:

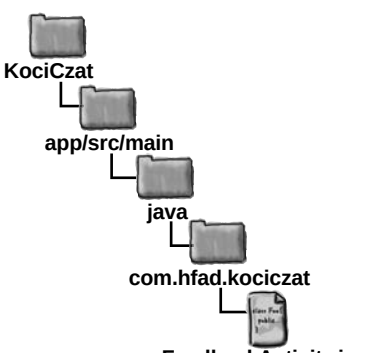

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context="com.hfad.kociczat.FeedbackActivity">

    <include
        layout="@layout/toolbar_main"
        android:id="@+id/toolbar" />

    <TextView
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:text="Opinia" />
</LinearLayout>
```



Teraz zaktualizuj plik *FeedbackActivity.java*, zapisując w nim poniższy kod:

```
package com.hfad.kociczat;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;

public class FeedbackActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_feedback);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
}
```

Także ta aktywność musi dziedziczyć po klasie AppCompatActivity.

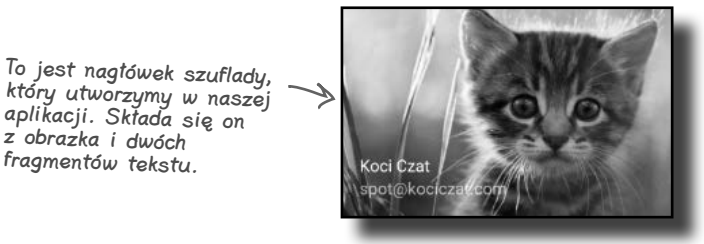


Musimy przygotować szufladę nawigacyjną

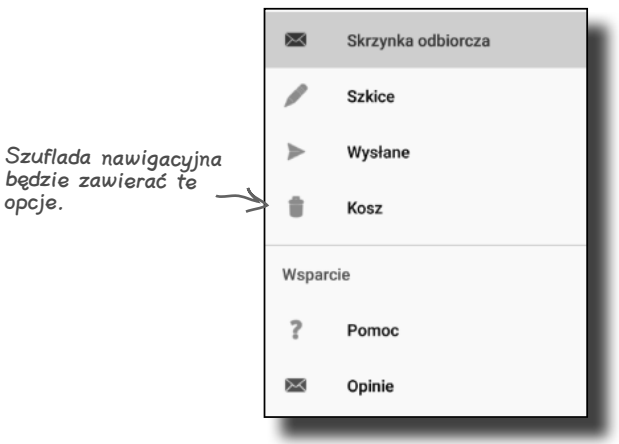
Dodaliśmy już do projektu wszystkie fragmenty i aktywności; będą one powiązane z opcjami, które znajdą się w szufladzie nawigacyjnej. Teraz zajmiemy się utworzeniem samej szuflady.

Szuflada nawigacyjna składa się z dwóch odrębnych komponentów:

- **Nagłówka szuflady.**
  To układ wyświetlany na samej górze szuflady nawigacyjnej. Zazwyczaj zawiera on obrazek i jakiś tekst, na przykład zdjęcie użytkownika i jego adres poczty elektronicznej.



- **Zestawu opcji.**
  Zestaw opcji jest wyświetlany w szufladzie nawigacyjnej poniżej jej nagłówka. Kiedy użytkownik kliknie jedną z tych opcji, ekran skojarzony z daną opcją zostanie wyświetlony, bądź to jako fragment w aktywności zawierającej szufladę, bądź jako nowa aktywność.



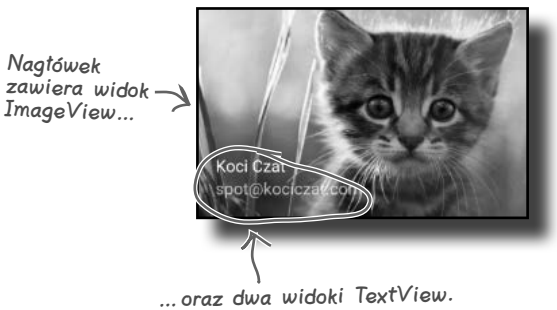
Zajmiemy się teraz przygotowaniem obu tych komponentów, a następnie użyjemy ich w aktywności `MainActivity`, aby zaimplementować szufladę nawigacyjną. Zaczniemy od nagłówka szuflady.

Utworzenie nagłówka szuflady nawigacyjnej

- ☑ Fragmenty i aktywności
- → ☐ Nagłówek
- ☐ Opcje
- ☐ Szuflada

Nagłówek szuflady nawigacyjnej składa się z prostego układu, który dodamy do nowego pliku układu. Utworzymy w tym celu nowy plik, o nazwie *nav_header.xml*. A zatem, aby to zrobić, zaznacz w Android Studio katalog *app/src/main/res/layout*, po czym wybierz opcję *File/New/Layout resource file*. Kiedy zostaniesz poproszony o podanie nazwy pliku, wpisz `nav_header`.

Nasz układ nagłówka składa się z trzech widoków: `ImageView` oraz dwóch `TextView`. Oznacza to, że do projektu musimy dodać plik obrazka jako zasób graficzny oraz dwa zasoby łańcuchowe. Zaczniemy od pliku obrazka.



Dodanie pliku obrazka

Aby dodać obrazek, w pierwszej kolejności w eksploratorze Android Studio musisz przełączyć się do widoku *Project*, o ile nie zrobiłeś tego już wcześniej. Następnie sprawdź, czy w projekcie istnieje katalog o nazwie *app/src/main/res/drawable*. Jeśli go nie ma, to zaznacz katalog *app/src/main/res*, wybierz z menu opcję *File/New...*, a następnie opcję pozwalającą utworzyć nowy katalog zasobów aplikacji na Androida. Gdy zostaniesz poproszony o podanie nazwy katalogu, wpisz `drawable` i kliknij *OK*.

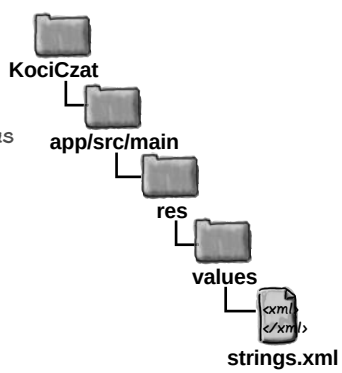
Po utworzeniu katalogu *drawable* w przykładach dołączonych do książki odszukaj plik *kitten_small.jpg* — będzie on umieszczony w analogicznym podrozdziale *drawable*, w katalogu *rozdzial_14* — i skopiuj go do katalogu *drawable* swojego projektu.

Dodanie zasobów łańcuchowych

Kolejnym krokiem będzie dodanie zasobów łańcuchowych, których będziemy używać do określenia tekstów wyświetlanych w nagłówku. Otwórz plik *app/src/main/res/values/strings.xml* i dodaj do niego dwa poniższe zasoby:

```
<resources>
    ...
    <string name="app_name">KociCzat</string>
    <string name="user_name">spot@kociczat.com</string>
</resources>
```

Android Studio mogło dodać ten zasób automatycznie podczas tworzenia projektu.



Skoro dodaliśmy już zasoby, możemy się zająć pisaniem kodu układu. Już doskonale potrafisz napisać cały niezbędny kod, dlatego na następnej stronie przedstawimy go w całości.

Kompletny kod pliku nav_header.xml

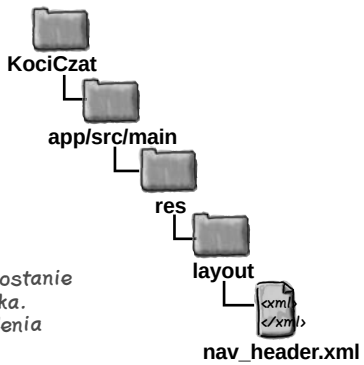
- ☑ Fragmenty i aktywności
- → ☑ Nagłówek
- ☐ Opcje
- ☐ Szuflada

Poniżej przedstawiliśmy kompletny kod pliku *nav_header.xml*; zaktualizuj plik w swoim projekcie, tak by był identyczny z naszym:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="180dp"
    android:theme="@style/ThemeOverlay.AppCompat.Dark" >

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:scaleType="centerCrop"
        android:src="@drawable/kitten_small" />

    <LinearLayout
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:orientation="vertical"
        android:gravity="bottom|start"
        android:layout_margin="16dp" >

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/app_name"
            android:textAppearance="@style/TextAppearance.AppCompat.Body1" />

        <TextView
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:text="@string/user_name" />
    </LinearLayout>
</FrameLayout>
```

Jawnie określamy, że wysokość układu wynosi 180dp, tak by nie zabierał on zbyt dużo miejsca w szufladzie nawigacyjnej.

Tło obrazka jest dość ciemne, więc używamy tego wiersza, który sprawi, że tekst będzie jasny.



Ten układ LinearLayout zostanie wyświetlony na tle obrazka. Używamy go do wyświetlenia tekstów u dołu obrazka.

To jest wbudowany styl, który powoduje, że wyświetlany w nim tekst jest nieco bardziej wytłuszczony. Jest on dostępny w bibliotece wsparcia AppCompat.

Skoro już utworzyliśmy nagłówek szuflady nawigacyjnej, możemy przejść do przygotowania listy opcji.

Szuflada pobiera opcje z menu

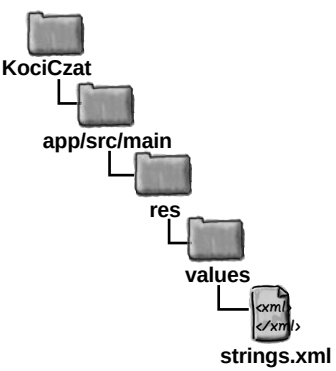
- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- → ☐ Opcje
- ☐ Szuflada

Szuflada nawigacyjna pobiera listę swoich opcji z pliku zasobu menu. Kod realizujący tę operację jest bardzo podobny do tego, którego używaliśmy do dodawania opcji do paska aplikacji.

Zanim przyjrzymy się kodowi służącemu do dodawania opcji do szuflady nawigacyjnej, musimy dodać do projektu plik zasobu menu. W tym celu zaznacz w Android Studio katalog *app/src/main/res* i z menu głównego wybierz opcję *File/New*. Następnie wybierz opcję pozwalającą na utworzenie nowego pliku zasobu. W efekcie zostaniesz poproszony o podanie nazwy pliku oraz typu zasobu. Jako nazwę wpisz `menu_nav`, a na liście typów zasobów wybierz opcję *Menu*. Upewnij się także, że w polu *Directory* jest wybrany katalog *menu*. Po kliknięciu przycisku *OK* Android Studio utworzy plik zasobu.

W kolejnym kroku dodamy do projektu zasoby łańcuchowe określające tytuły poszczególnych opcji, dzięki czemu będziemy mogli ich użyć w dalszej części rozdziału. Otwórz plik *strings.xml* i dodaj do niego poniższe zasoby:

```
<resources>
    ...
    <string name="nav_inbox">Skrzynka odbiorcza</string>
    <string name="nav_drafts">Szkice</string>
    <string name="nav_sent">Wysłane</string>
    <string name="nav_trash">Kosz</string>
    <string name="nav_support">Wsparcie</string>
    <string name="nav_help">Pomoc</string>
    <string name="nav_feedback">Opinie</string>
</resources>
```

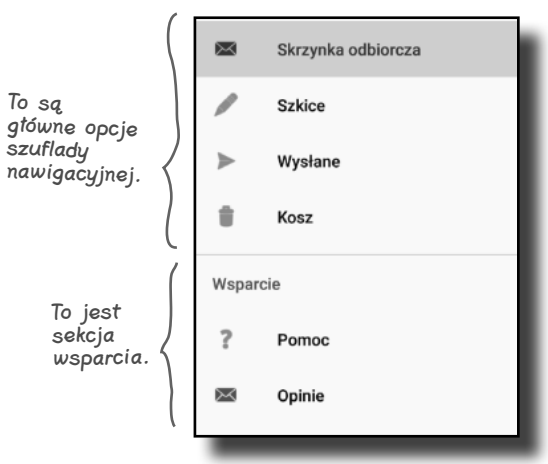


Teraz możemy już przystąpić do pisania kodu menu.

Tworzone menu ma mieć dwie sekcje

Jak już zaznaczyliśmy wcześniej, chcemy podzielić opcje w szufladzie nawigacyjnej na dwie sekcje. Pierwsza z nich będzie zawierać opcje pozwalające użytkownikom na przejście w kilka głównych miejsc aplikacji, a konkretnie do: skrzynki odbiorczej, szkiców, wiadomości wysłanych oraz kosza. Poniżej będzie umieszczona odrębna sekcja zawierająca opcje związane z systemem pomocy oraz opiniami.

Zacznijmy od dodania opcji głównych.

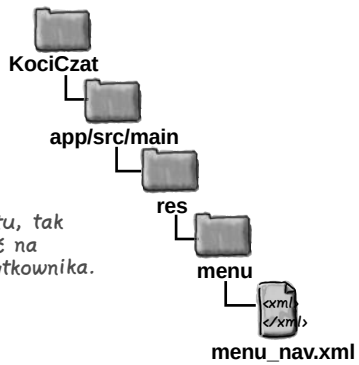


Dodawaj opcje w kolejności, w jakiej mają być wyświetlane

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- → ☐ Opcje
- ☐ Szuflada

Podczas projektowania zestawu opcji mających się znaleźć w szufladzie nawigacyjnej zazwyczaj te z nich, które najprawdopodobniej będą najczęściej klikane przez użytkownika, powinny się znaleźć na samej górze listy. W naszym przykładzie będą to opcje: skrzynki odbiorczej, szkiców, wiadomości wysłanych i kosza.

Poszczególne elementy należy dodawać do pliku zasobu w takiej kolejności, w jakiej później mają być wyświetlane w szufladzie nawigacyjnej. Dla każdego z elementów należy podać identyfikator, dzięki któremu będzie można odwoływać się do niego w kodzie Javy, oraz tytuł określający tekst, jaki chcemy wyświetlić w szufladzie. Dodatkowo można także określić ikonę, która zostanie wyświetlona obok tytułu opcji. W ramach przykładu poniżej pokazaliśmy kod dodający do szuflady nawigacyjnej opcję „Skrzynka odbiorcza”:




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

    <item
        android:id="@+id/nav_inbox"
        android:icon="@android:drawable/sym_action_email"
        android:title="@string/nav_inbox" />

    ...

</menu>
```

Musisz określić identyfikator elementu, tak by kod aktywności mógł odpowiedzieć na kliknięcie danego elementu przez użytkownika.

To jest wbudowany zasób graficzny, którego można użyć do wyświetlenia ikony e-maila.

To jest tekst, który zostanie wyświetlony w szufladzie nawigacyjnej.

W powyższym kodzie zastosowaliśmy jedną z wbudowanych ikon dostępnych w systemie Android: `"@android:drawable/sym_action_email"`. System Android zawiera wbudowany zestaw ikon, których można używać w tworzonych aplikacjach. O tym, że chcemy użyć jednej z tych ikon informuje początkowy fragment identyfikatora: `"@android:drawable"`. Pełną listę dostępnych ikon można wyświetlić, rozpoczynając wpisywanie identyfikatora w Android Studio:

Jak można grupować elementy?

Oprócz dodawania do szuflady nawigacyjnej pojedynczych elementów można je także dodawać w grupach. Grupy definiuje się przy użyciu znacznika `<group>`, jak pokazano poniżej:

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

Tutaj umieszczamy wszystkie elementy, które chcemy dodać do grupy.

Rozwiązanie to jest przydatne, gdy chcemy zastosować jakiś atrybut dla całej grupy elementów. Na przykład aby wyróżnić w szufladzie element wybrany przez użytkownika, można dodać do znacznika `<group>` atrybut `android:checkableBehavior` o wartości `"single"`. Ten sposób działania szuflady jest przydatny, jeśli planujemy wyświetlanie ekranów poszczególnych opcji jako fragmentów podmienianych aktywności, do której została dodana szuflada nawigacyjna (w naszym przykładzie jest to aktywność `MainActivity`), gdyż ułatwia określenie obecnie wybranej opcji:

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

Użycie tej wartości atrybutu oznacza, że w grupie będzie wyróżniona pojedyncza opcja (ta, którą wybrał użytkownik).

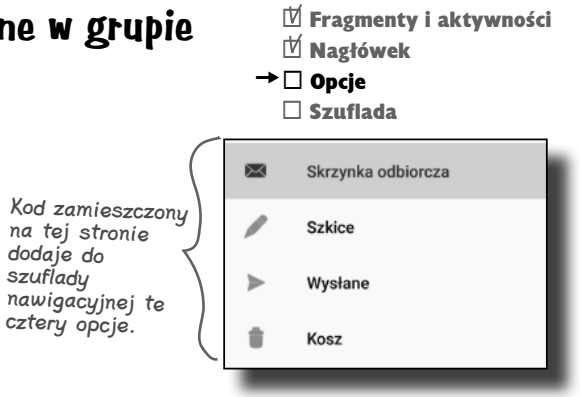
Wybrany element szuflady nawigacyjnej można domyślnie wyróżnić, dodając do niego atrybut `android:checked` o wartości `"true"`. Poniższy przykład pokazuje, w jaki sposób można wyróżnić element „Skrzynka odbiorcza”:

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_inbox"
            android:icon="@android:drawable/sym_action_email"
            android:title="@string/nav_inbox"
            android:checked="true" />
        ...
    </group>
</menu>
```

Zastosowanie tego atrybutu powoduje domyślne wyróżnienie danego elementu w szufladzie nawigacyjnej.

Na następnej stronie pokażemy pełny kod definiujący cztery początkowe elementy szuflady nawigacyjnej.

Opcje pierwszej sekcji zostaną umieszczone w grupie

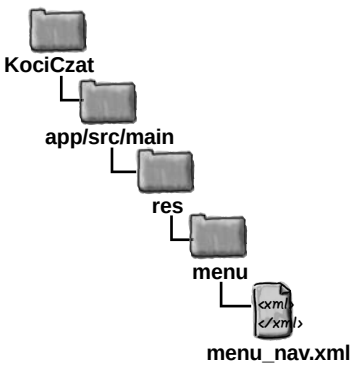


Opcje dla skrzynki odbiorczej, szkiców, wiadomości wysłanych oraz kosza dodamy do naszego pliku zasobu menu w formie jednej grupy i domyślnie wyróżnimy pierwszą z nich. Opcje te umieścimy w grupie, ponieważ ekrany dla każdej z nich są fragmentami wyświetlanymi w ramach aktywności `MainActivity`.

Poniżej przedstawiliśmy kodu menu; zaktualizuj swoją wersję pliku *menu_nav.xml*, by była identyczna z naszą:

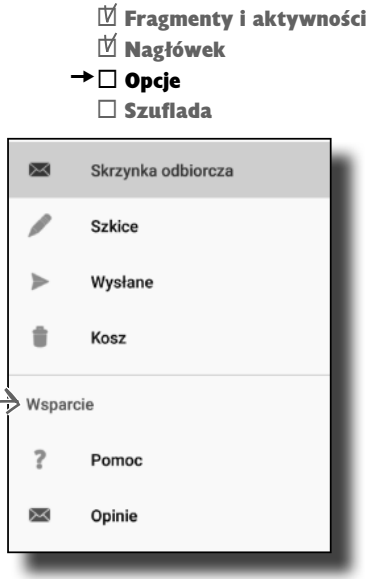
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_inbox"
            android:icon="@android:drawable/sym_action_email"
            android:title="@string/nav_inbox"
            android:checked="true" />
        <item
            android:id="@+id/nav_drafts"
            android:icon="@android:drawable/ic_menu_edit"
            android:title="@string/nav_drafts" />
        <item
            android:id="@+id/nav_sent"
            android:icon="@android:drawable/ic_menu_send"
            android:title="@string/nav_sent" />
        <item
            android:id="@+id/nav_trash"
            android:icon="@android:drawable/ic_menu_delete"
            android:title="@string/nav_trash" />
    </group>

</menu>
```

Dodaj tę grupę i cztery umieszczone w niej opcje do pliku zasobu menu, aby zostały wyświetlone w szufladzie nawigacyjnej.



W ten sposób uporaliśmy się z pierwszą grupą opcji. Kolejnymi zajmiemy się w dalszej kolejności.

Sekcję wsparcia dodamy jako podmenu



Drugi zestaw elementów tworzonej szuflady nawigacyjnej ma postać odrębnej sekcji. Znajdują się w niej nagłówek „Wsparcie” oraz dwie opcje, „Pomoc” i „Opinie”, które użytkownik będzie mógł klikać.

Tworzenie tej sekcji rozpoczniemy od dodania nagłówka „Wsparcie”, który zdefiniujemy jako osobny element. Ponieważ ma to być nagłówek, zdefiniujemy w nim jedynie tytuł — pominiemy ikonę oraz identyfikator, gdyż ten element szuflady nie będzie musiał reagować na kliknięcia:

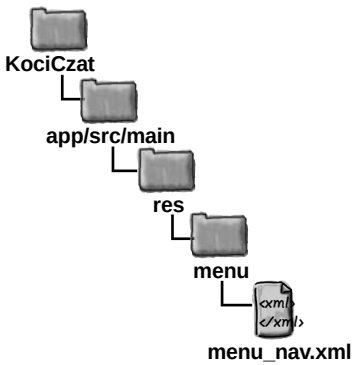
```
...
    <item android:title="@string/nav_support">
    </item>
...
```

Ten element dodaje do szuflady nawigacyjnej nagłówek „Wsparcie”.

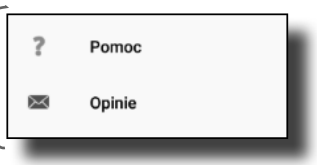
Chcemy, by opcje zapewniające dostęp do systemu pomocy i opinii były wyświetlone w sekcji wsparcia, dlatego dodamy je wewnątrz tej sekcji jako niezależne opcje umieszczone w podmenu:

```
...
    <item android:title="@string/nav_support">
        <menu>
            <item
                android:id="@+id/nav_help"
                android:icon="@android:drawable/ic_menu_help"
                android:title="@string/nav_help"/>
            <item
                android:id="@+id/nav_feedback"
                android:icon="@android:drawable/sym_action_email"
                android:title="@string/nav_feedback" />
        </menu>
    </item>
...
```

Ten blok kodu definiuje podmenu wewnątrz sekcji Wsparcie.



Ten kod powoduje dodanie do szuflady tych dwóch elementów.



Warto zwrócić uwagę na to, że tych opcji nie dodawaliśmy w formie jednej grupy; a zatem kiedy użytkownik kliknie jedną z nich, nie zostanie ona wyróżniona w szufladzie nawigacyjnej. Zastosowaliśmy takie rozwiązanie, ponieważ obie te opcje będą wyświetlane jako aktywności, a nie fragmenty prezentowane wewnątrz aktywności zawierającej szufladę nawigacyjną.

Pełny kod menu przedstawimy na następnej stronie.

- [x] Fragmenty i aktywności
- [x] Nagłówek
- [x] Opcje
- [ ] Szuflada

Kompletny kod pliku menu_nav.xml

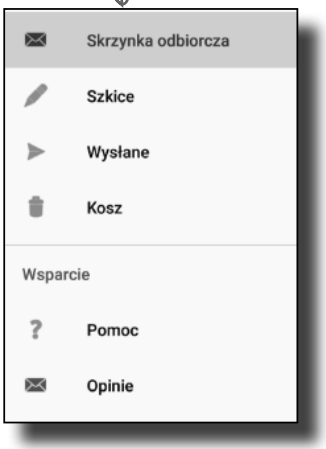
Oto kompletny kod pliku *menu_nav.xml*; zaktualizuj ten plik w swoim projekcie, tak by był identyczny z naszym:

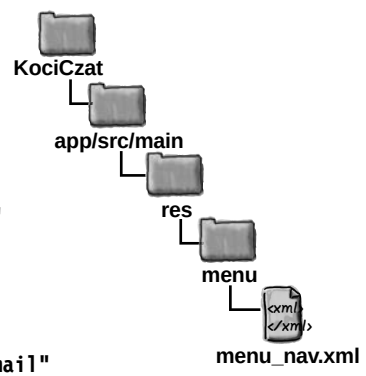
```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <group android:checkableBehavior="single">
        <item
            android:id="@+id/nav_inbox"
            android:icon="@android:drawable/sym_action_email"
            android:title="@string/nav_inbox"
            android:checked="true" />
        <item
            android:id="@+id/nav_drafts"
            android:icon="@android:drawable/ic_menu_edit"
            android:title="@string/nav_drafts" />
        <item
            android:id="@+id/nav_sent"
            android:icon="@android:drawable/ic_menu_send"
            android:title="@string/nav_sent" />
        <item
            android:id="@+id/nav_trash"
            android:icon="@android:drawable/ic_menu_delete"
            android:title="@string/nav_trash" />
    </group>
    <item android:title="@string/nav_support">
        <menu>
            <item
                android:id="@+id/nav_help"
                android:icon="@android:drawable/ic_menu_help"
                android:title="@string/nav_help"/>
            <item
                android:id="@+id/nav_feedback"
                android:icon="@android:drawable/sym_action_email"
                android:title="@string/nav_feedback" />
        </menu>
    </item>
</menu>
```

To są główne opcje szuflady.

To jest sekcja wsparcia.

Kod zamieszczony na tej stronie tworzy pełne menu.





Skoro dodaliśmy już do układu menu oraz nagłówek szuflady nawigacyjnej, możemy przejść do utworzenia jej samej.

Jak utworzyć szufladę nawigacyjną?

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

Szufladę nawigacyjną tworzymy poprzez zastosowanie układu `DrawerLayout` jako głównego elementu układu aktywności. Układ `DrawerLayout` musi zawierać dwa elementy: pierwszym z nich ma być widok bądź też grupa widoków określające zawartość aktywności, a drugim widok `NavigationView`, definiujący szufladę:

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical" >
        ...
    </LinearLayout>

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/nav_header"
        app:menu="@menu/menu_nav" />
</android.support.v4.widget.DrawerLayout>
```

Układ DrawerLayout definiuje szufladę.

Określamy identyfikator układu, by móc odwoływać się do niego w kodzie aktywności.

Pierwszym widokiem umieszczonym w DrawerLayout jest układ określający główną zawartość aktywności. Jest ona widoczna, gdy szuflada nawigacyjna jest ukryta.

Widok NavigationView definiuje zawartość szuflady.

Ten atrybut umieszcza szufladę przy początkowej krawędzi aktywności (w przypadku języków, w których piszemy od lewej do prawej, będzie to lewa krawędź ekranu).

To jest układ nagłówka szuflady.

A to plik zasobu menu definiującego opcje szuflady.

Wygląd szuflady nawigacyjnej określają dwa kluczowe atrybuty elementu `<NavigationView>`: `headerLayout` oraz `menu`.

Atrybut `app:headerLayout` określa układ, który powinien zostać zastosowany jako nagłówek szuflady (w naszym przykładzie jest to plik *nav_header.xml*). Jest to atrybut opcjonalny.

Z kolei atrybut `app:menu` służy do określania pliku zasobu menu zawierającego opcje szuflady (w naszej aplikacji jest to plik *menu_nav.xml*). Jeśli pominiemy ten atrybut, szuflada nawigacyjna nie będzie zawierać żadnych elementów.

Kompletny kod układu aktywności activity_main.xml

Dodamy teraz do układu aktywności `MainActivity` szufladę nawigacyjną, korzystającą z przygotowanego wcześniej układu nagłówka oraz pliku zasobu menu. Główna zawartość układu będzie się składać z paska narzędzi oraz układu `FrameLayout`. Tego układu użyjemy w dalszej części rozdziału do wyświetlania fragmentów.

Poniżej przedstawiliśmy kod pliku *activity_main.xml*; zaktualizuj zawartość tego pliku w swoim projekcie, tak by była identyczna z tą ukazaną przez nas:

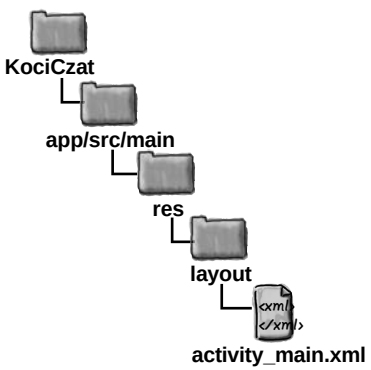
```
<?xml version="1.0" encoding="utf-8"?>
<android.support.v4.widget.DrawerLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:id="@+id/drawer_layout"
    android:layout_width="match_parent"
    android:layout_height="match_parent" >

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

        <include
            layout="@layout/toolbar_main"
            android:id="@+id/toolbar" />

        <FrameLayout
            android:id="@+id/content_frame"
            android:layout_width="match_parent"
            android:layout_height="match_parent" />
    </LinearLayout>

    <android.support.design.widget.NavigationView
        android:id="@+id/nav_view"
        android:layout_width="wrap_content"
        android:layout_height="match_parent"
        android:layout_gravity="start"
        app:headerLayout="@layout/nav_header"
        app:menu="@menu/menu_nav" />
</android.support.v4.widget.DrawerLayout>
```

Głównym elementem układu jest DrawerLayout.

Układ ma zdefiniowany identyfikator, dzięki czemu będziemy mogli odwoływać się do niego w kodzie aktywności.

Ten fragment definiuje główną zawartość aktywności.

Na główną zawartość aktywności składają się komponent Toolbar oraz układ FrameLayout, w którym będziemy wyświetlali fragmenty.



Widok NavigationView definiuje wygląd szuflady nawigacyjnej oraz przeważającą część jej zachowania. Określamy identyfikator tego widoku, gdyż będziemy musieli odwoływać się do niego w kodzie aktywności.

Tu używamy utworzonych wcześniej plików zasobów: układu określającego postaci nagłówka oraz menu definiującego zawartość listy opcji.

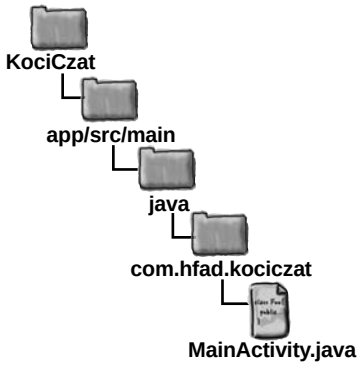
Zanim uruchomimy aplikacje, by przekonać się, jak wygląda nasza szuflada nawigacyjna, zmienimy jeszcze kod aktywności `MainActivity`, aby bezpośrednio po utworzeniu aktywności wyświetlany był w niej fragment `InboxFragment`.

Dodanie fragmentu InboxFragment do układu aktywności MainActivity

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

Tworząc plik zasobu menu, zdefiniowaliśmy opcję skrzynki odbiorczej jako domyślnie wyróżnioną. Dlatego powinniśmy teraz wyświetlić fragment `InboxFragment` podczas tworzenia aktywności, tak by jej zawartość odpowiadała zaznaczonej opcji szuflady nawigacyjnej. Dodatkowo użyjemy paska narzędzi jako paska aplikacji aktywności, tak by został na nim wyświetlony tytuł aplikacji.

Poniżej przedstawiliśmy naszą wersję pliku *MainActivity.java*; zmodyfikuj ten sam plik w swoim projekcie, by był identyczny z naszym:




```
package com.hfad.kociczat;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        Fragment fragment = new InboxFragment();
        FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
        ft.add(R.id.content_frame, fragment);
        ft.commit();
    }
}
```

Upewnij się, że aktywność dziedziczy po klasie AppCompatActivity, gdyż używamy motywu AppCompat oraz fragmentów z biblioteki wsparcia.

Ustawiamy komponent Toolbar jako pasek aplikacji aktywności.

Stosujemy transakcję fragmentu, by wyświetlić instancję InboxFragment.

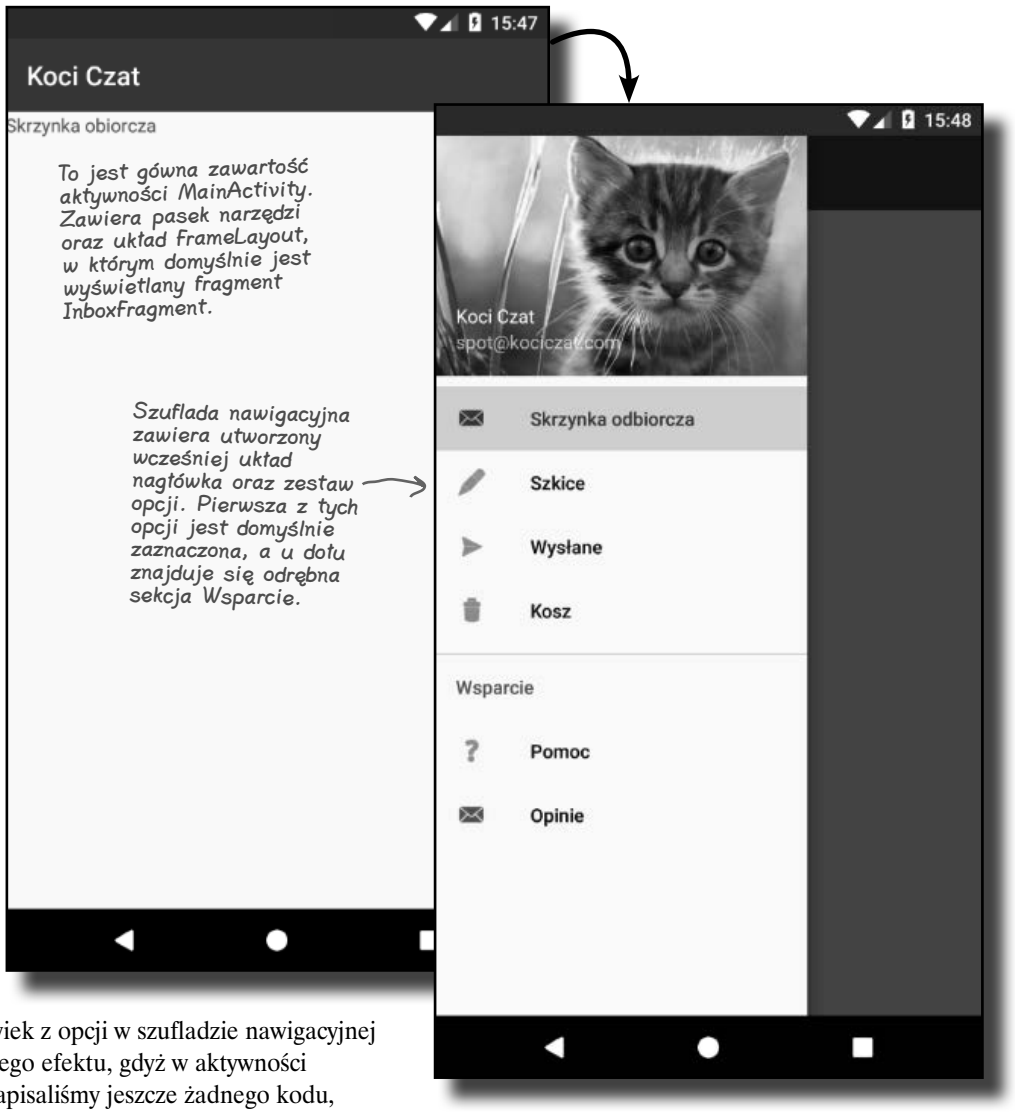
A teraz sprawdźmy, co się stanie po uruchomieniu aplikacji.

Jazda próbna aplikacji

- [x] Fragmenty i aktywności
- [x] Nagłówek
- [x] Opcje
- [ ] Szuflada

Po uruchomieniu aplikacji w aktywności `MainActivity` zostaje wyświetlony fragment `InboxFragment`. Kiedy wykonamy gest przeciągnięcia od lewej krawędzi ekranu do jego środka (przynajmniej w przypadku języków takich jak polski, w których piszemy od strony lewej do prawej), zostanie wyświetlona szuflada nawigacyjna. Szuflada ta zawiera układ nagłówka oraz listę opcji zdefiniowaną w pliku zasobu menu. Pierwsza z tych opcji jest automatycznie wyróżniona:

W językach, w których pisze się od strony prawej do lewej, szuflada jest wyświetlana przy prawej krawędzi ekranu.



Kliknięcie którejkolwiek z opcji w szufladzie nawigacyjnej nie daje jeszcze żadnego efektu, gdyż w aktywności `MainActivity` nie napisaliśmy jeszcze żadnego kodu, który by kontrolował działanie szuflady. Zajmiemy się tym w następnej kolejności.

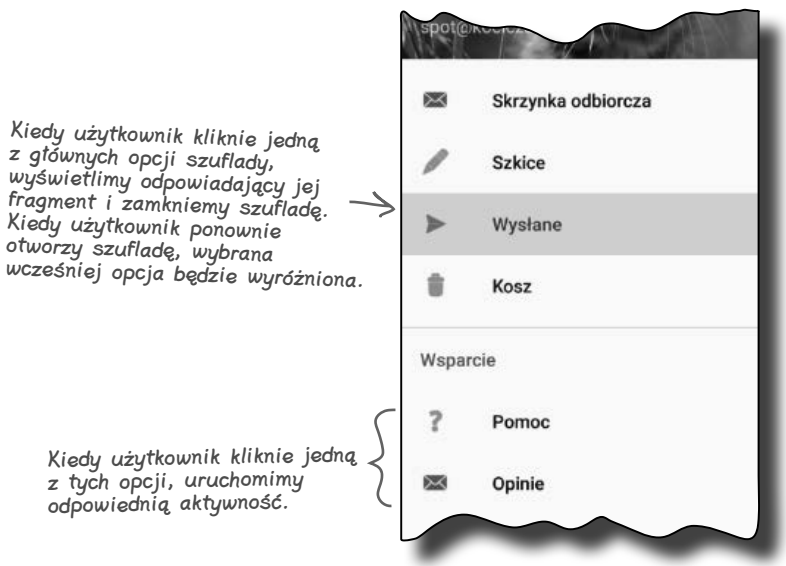
Co musi robić kod aktywności?

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

Kod aktywności musi wykonać trzy operacje:

1. **Dodać przełącznik szuflady.**
Będzie on stanowić wizualny sygnał dla użytkownika informujący o tym, że aktywność zawiera szufladę nawigacyjną. Przełącznik ten ma postać ikony „hamburgera” umieszczonej na pasku narzędzi, którą można kliknąć, aby otworzyć szufladę.

2. **Zapewnić, że szuflada będzie reagować na kliknięcia.**
Kiedy użytkownik kliknie jedną z opcji w szufladzie nawigacyjnej, wyświetlimy odpowiedni fragment lub odpowiednią aktywność i zamkniemy szufladę.



3. **Zamykać szufladę, kiedy użytkownik naciśnie klawisz Wstecz.**
Jeśli szuflada będzie widoczna, to po kliknięciu przycisku *Wstecz* zamkniemy ją. Jeśli szuflada będzie zamknięta, to kliknięcie przycisku *Wstecz* da standardowe efekty.

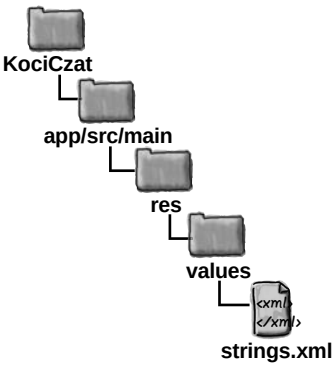
Zaczniemy od dodania przełącznika szuflady.

Dodanie przełącznika szuflady

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

Pierwszą rzeczą, jaką musimy zrobić, jest dodanie przełącznika szuflady, dzięki któremu będziemy mogli ją wyświetlić, klikając ikonę na pasku narzędzi.

Zaczniemy od dodania dwóch zasobów łańcuchowych, których użyjemy do opisania akcji „otwórz szufladę” oraz „zamknij szufladę”, wymaganych w celu ułatwiania dostępu. Dodaj poniższe dwa zasoby do pliku *strings.xml*:




```
<string name="nav_open_drawer">Otwórz szufladę nawigacyjną</string>
<string name="nav_close_drawer">Zamknij szufladę nawigacyjną</string>
```

Przełącznik szuflady nawigacyjnej konstruujemy w metodzie `onCreate()` aktywności, tworząc nową instancję klasy **`ActionBarDrawerToggle`** i dodając ją do układu `DrawerLayout`. Najpierw pokażemy Ci kod realizujący tę operację, a później, w dalszej części rozdziału, dodamy go do kodu aktywności `MainActivity`.

Konstruktor `ActionBarDrawerToggle` ma aż pięć parametrów: bieżącą aktywność, układ `DrawerLayout`, obiekt paska narzędzi oraz dwa zasoby łańcuchowe opisujące czynności otwierania i zamykania szuflady (zasoby te dodaliśmy powyżej):

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
...
DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this,
        drawer,
        toolbar,
        R.string.nav_open_drawer,
        R.string.nav_close_drawer);
```

Bieżąca aktywność.

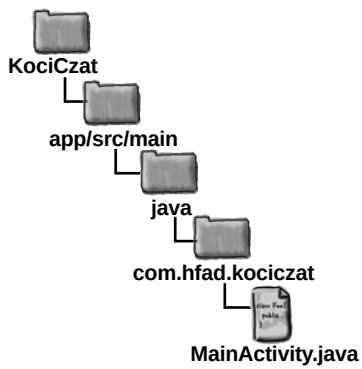
Układ DrawerLayout używany w aktywności.

Pasek narzędzi aktywności.

Ta instrukcja dodaje ikonę „hamburgera” do paska narzędzi.

Te łańcuchy znaków są niezbędne ze względu na wymogi dotyczące ułatwiania dostępu.

Po utworzeniu przełącznika szuflady musimy go dodać do układu — w tym celu należy wywołać metodę `addDrawerListener()` klasy `DrawerLayout` i przekazać w jej wywołaniu utworzoną instancję przełącznika:




```
drawer.addDrawerListener(toggle);
```

Następnie należy jeszcze wywołać metodę `syncState()`, aby zsynchronizować stan ikony na pasku narzędzi ze stanem szuflady nawigacyjnej. To konieczne, gdyż stan ikony zmienia się, gdy klikniemy ją, by otworzyć szufladę:

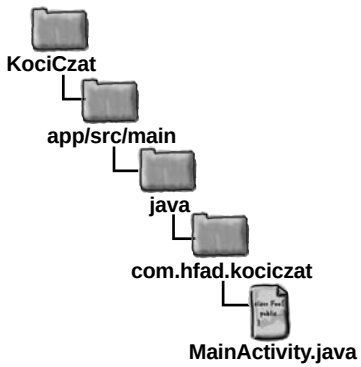
```
toggle.syncState();
```

Już niebawem dodamy cały ten kod do metody `onCreate()` aktywności `MainActivity`.

Reagowanie na klikanie elementów szuflady

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

Teraz zajmiemy się zapewnieniem odpowiedniej reakcji aktywności `MainActivity` na klikanie opcji umieszczonych w szufladzie. W tym celu zaimplementujemy w niej interfejs **`NavigationView.OnNavigationItemSelectedListener`**. W ten sposób sprawimy, że każde kliknięcie jednej z opcji dostępnych w szufladzie nawigacyjnej spowoduje wywołanie nowej metody, `onNavigationItemSelected()`, którą zaimplementujemy w aktywności `MainActivity`. Użyjemy tej metody, by wyświetlać ekran odpowiedni dla wybranej opcji.



W pierwszej kolejności, używając poniższego kodu, zaimplementujemy interfejs w kodzie aktywności `MainActivity`. W ten sposób aktywność stanie się obiektem nasłuchującym zdarzeń generowanych przez widok `NavigationView`:

```
...
import android.support.design.widget.NavigationView;

public class MainActivity extends AppCompatActivity
                          implements NavigationView.OnNavigationItemSelectedListener {
    ...
}
```

Implementacja tego interfejsu oznacza, że aktywność może odpowiadać na kliknięcia opcji w szufladzie nawigacyjnej.

Kolejną czynnością będzie zarejestrowanie obiektu nasłuchującego, czyli aktywności `MainActivity`, w widoku `NavigationView`. Dzięki temu aktywność będzie informowana o tym, kiedy użytkownik kliknie jedną z opcji dostępnych w szufladzie nawigacyjnej. W tym celu w metodzie `onCreate()` aktywności musimy pobrać referencję do obiektu `NavigationView` i wywołać jego metodę `setNavigationItemSelectedListener()`:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    ...
    NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
    navigationView.setNavigationItemSelectedListener(this);
}
```

To wywołanie rejestruje aktywność jako obiekt nasłuchujący zdarzeń generowanych przez widok NavigationView, dzięki czemu zostanie ona poinformowana, kiedy użytkownik kliknie jedną z opcji dostępnych w szufladzie nawigacyjnej.

I w końcu pozostaje nam zaimplementowanie metody `onNavigationItemSelected()`.

Implementacja metody onNavigationItemSelected()

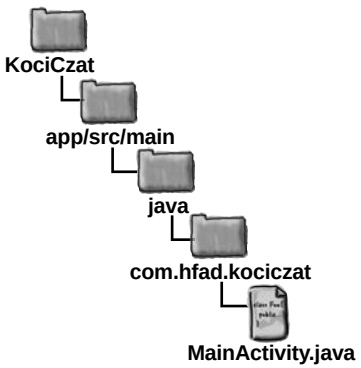
- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

Metoda `onNavigationItemSelected()` jest wywoływana, kiedy użytkownik kliknie jeden z elementów wyświetlonych w szufladzie nawigacyjnej. Ma ona jeden parametr — obiekt `MenuItem`, reprezentujący kliknięty element menu — i zwraca wartość logiczną określającą, czy dany element szuflady należy wyróżnić:

Ta metoda zostaje wywołana po każdym kliknięciu elementu szuflady nawigacyjnej. Jej parametrem jest kliknięty element.

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    // kod obsługujący kliknięcie elementu szuflady
}
```

W naszej aplikacji kod w tej metodzie musi wyświetlać odpowiedni ekran, odpowiadający kliknięte mu elementowi szuflady. Jeśli element szuflady będzie powiązany z aktywnością, to musimy ją uruchomić przy użyciu intencji. Jeśli natomiast kliknięty element jest powiązany z fragmentem, to wystarczy, że wyświetlimy go w układzie `FrameLayout` aktywności `MainActivity`, używając do tego transakcji fragmentu.



Ogólnie rzecz biorąc, w przypadku wyświetlania fragmentów w reakcji na klikanie opcji w szufladzie nawigacyjnej zazwyczaj nie umieszcza się przy tym transakcji na stosie cofnięć, jak robiliśmy to wcześniej. Wynika to z tego, że klikając przycisk *Wstecz*, użytkownik nie będzie oczekiwał wyświetlania kolejno fragmentów odpowiadających wszystkim wybieranym wcześniej opcjom szuflady nawigacyjnej. Dlatego przeważnie używany jest kod taki jak ten przedstawiony poniżej:

```
FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
ft.replace(R.id.content_frame, fragment);
ft.commit();
```

To jest ten sam kod transakcji fragmentu, którego używaliśmy wcześniej, z tą różnicą, że nie dodajemy transakcji do stosu cofnięć aktywności.

I w końcu musimy zamknąć szufladę. W tym celu musimy pobrać referencję do układu `DrawerLayout` i wywołać metodę `closeDrawer()` uzyskanego obiektu:

```
DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
drawer.closeDrawer(GravityCompat.START);
```

Używamy stałej GravityCompat.START, gdyż umieściliśmy szufladę przy początkowej krawędzi aktywności. Gdybyśmy umieścili szufladę przy końcowej krawędzi, musielibyśmy użyć stałej GravityCompat.END.

Wykonanie powyższego fragmentu kodu spowoduje wysunięcie szuflady poza początkową krawędź aktywności.

Teraz już wiesz wszystko, co trzeba wiedzieć, żeby napisać kod metody `onNavigationItemSelected()`. Napiszesz tę metodę w ramach ćwiczenia zamieszczonego na kilku kolejnych stronach.

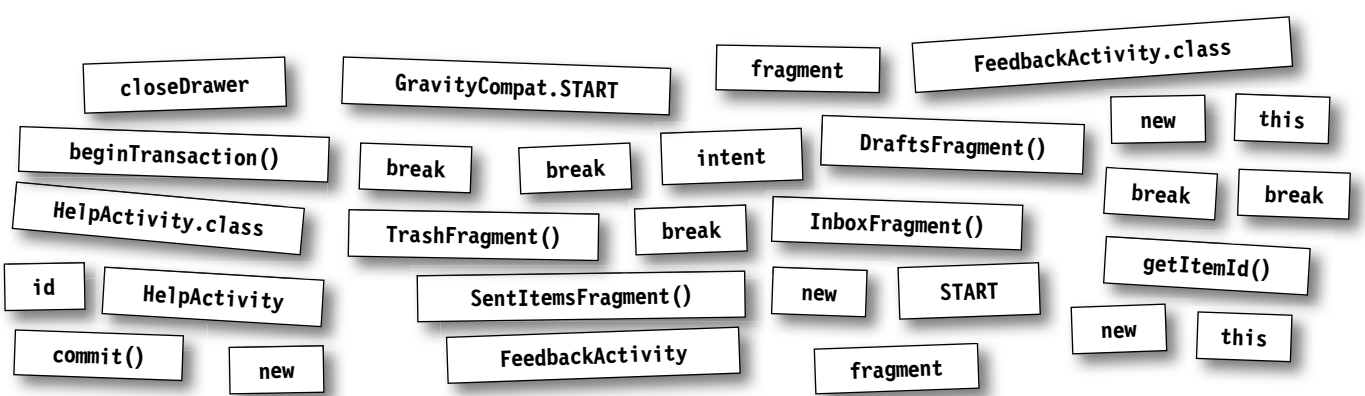
Magnesiki z kodem

Po kliknięciu przez użytkownika któregoś z elementów w szufladzie nawigacyjnej chcemy wyświetlić ekran skojarzony z tym elementem. Jeśli element będzie skojarzony z fragmentem, wyświetlimy go w układzie `content_frame`. Jeśli element będzie skojarzony z aktywnością, to będziemy chcieli ją uruchomić. Na końcu musimy zamknąć szufladę nawigacyjną.

Ciekawe, czy będziesz potrafił uzupełnić kod zamieszczony na tej oraz następnej stronie. Nie będziesz musiał użyć wszystkich dostępnych magnesików.

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {


    int id = item. ..................... ;
    Fragment fragment = null;
    Intent intent = null;


    switch( .......... ){
        case R.id.nav_drafts:
            fragment = ................................... ;
            ......... ;
        case R.id.nav_sent:
            fragment = ..................................... ;
            ..........;
        case R.id.nav_trash:
            fragment = ..................................... ;
            .......... ;
        case R.id.nav_help:
            intent = new Intent( ........ , ................. );
            .......... ;
```

```
            case R.id.nav_feedback:
                intent = new Intent( ......... , ..................... );
                ..........;
            default:
                fragment = ......................... ;
        }
        if ................. != null) {
            FragmentTransaction ft = getSupportFragmentManager(). ................. ;
            ft.replace(R.id.content_frame, ................. );
            ft. ................. ;
        } else {
            startActivity( ................ );
        }

        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);

        drawer. ............. ( ................................ );
        return true;
    }
```



Magnesiki z kodem. Rozwiązanie

Po kliknięciu przez użytkownika któregoś z elementów w szufladzie nawigacyjnej chcemy wyświetlić ekran skojarzony z tym elementem. Jeśli element będzie skojarzony z fragmentem, wyświetlimy go w układzie `content_frame`. Jeśli element będzie skojarzony z aktywnością, to będziemy chcieli ją uruchomić. Na końcu musimy zamknąć szufladę nawigacyjną.

Ciekawe, czy będziesz potrafił uzupełnić kod zamieszczony na tej oraz następnej stronie. Nie będziesz musiał użyć wszystkich dostępnych magnesików.

```
@Override
public boolean onNavigationItemSelected(MenuItem item) {

    int id = item. .. getItemId() . ;
    Fragment fragment = null;
    Intent intent = null;

    switch( .. id .. ){
        case R.id.nav_drafts:
            fragment = .. new .. DraftsFragment() ..;
            break ;
        case R.id.nav_sent:
            fragment = .. new . SentItemsFragment() . ;
            break .;
        case R.id.nav_trash:
            fragment = ... new . TrashFragment() .. ;
            break . ;
        case R.id.nav_help:
            intent = new Intent( this , HelpActivity.class . );
            break . ;
```

Pobieramy identyfikator wybranego elementu.

Zapisujemy w zmiennej fragment instancję fragmentu, który chcemy wyświetlić.

Tworzymy intencję do uruchomienia aktywności HelpActivity, o ile została wybrana opcja Pomoc.

```
        case R.id.nav_feedback:
            intent = new Intent( this . , FeedbackActivity.class );
            break ;
        default:
            fragment = . new .. InboxFragment() ... ;
    }
    if ( . fragment .. != null) {
        FragmentTransaction ft = getSupportFragmentManager(). . beginTransaction() ... ;
        ft.replace(R.id.content_frame, .. fragment . );
        ft. .. commit() . ;
    } else {
        startActivity( .. intent );
    }

    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);

    drawer. . closeDrawer ..(. GravityCompat.START ..);
    return true;
}
```

Jeśli kliknięto opcję Opinia, musimy uruchomić intencję FeedbackActivity.

Domyślnie wyświetlamy fragment InboxFragment, gdyż jest on powiązany z pierwszą opcją w szufladzie nawigacyjnej.

Jeśli musimy wyświetlić fragment, to robimy to, używając transakcji fragmentu.

Jeśli musimy wyświetlić aktywność, uruchamiamy ją używając utworzonej wcześniej intencji

W końcu zamykamy szufladę.

Już niebawem dodamy ten kod do pliku MainActivity.java.

Tych magnesików nie trzeba było używać.

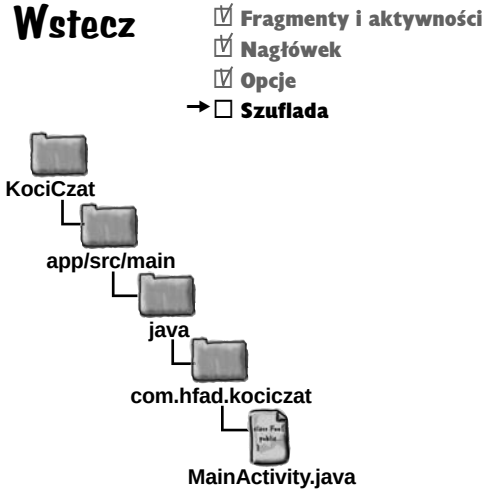
HelpActivity

FeedbackActivity

START

Zamknięcie szuflady po naciśnięciu przycisku Wstecz

I w końcu przesłonimy czynności wykonywane po naciśnięciu przycisku *Wstecz*. Jeśli użytkownik naciśnie ten przycisk, kiedy szuflada nawigacyjna będzie otworzona, zamkniemy ją. Jeśli natomiast szuflada już będzie zamknięta, to naciśnięcie przycisku *Wstecz* da standardowe rezultaty.



W celu obsługi przycisku *Wstecz* zaimplementujemy w aktywności metodę `onBackPressed()`, która jest wywoływana za każdym razem, kiedy użytkownik naciśnie ten przycisk. Poniżej przedstawiliśmy kod tej metody:

Ta metoda jest wywoływana po naciśnięciu przycisku Wstecz.

```
@Override
public void onBackPressed() {
    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    if (drawer.isDrawerOpen(GravityCompat.START)) {
        drawer.closeDrawer(GravityCompat.START);
    } else {
        super.onBackPressed();
    }
}
```

Jeśli szuflada nawigacyjna jest obecnie otworzona, to ją zamykamy.

W przeciwnym razie, jeśli szuflada jest zamknięta, wywołujemy metodę onBackPressed() klasy bazowej.

I to już wszystkie zmiany, które należy wprowadzić w kodzie aktywności `MainActivity`. Na kilku następnych stronach przedstawimy jej pełny kod.

Nie istnieją głupie pytania

P: Czy zawartość szuflady musi być określana przy użyciu widoku NavigationView?

O: Nie, jednak takie rozwiązanie jest *znacznie* prostsze. Przed udostępnieniem biblioteki wsparcia wzornictwa powszechnie stosowaną praktyką było wykorzystanie do tego widoku listy. Wciąż można stosować takie rozwiązanie, jednak wymaga ono napisania znacznie bardziej rozbudowanego kodu.

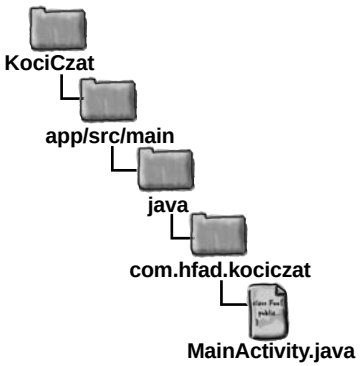
P: Czy aktywność może zawierać więcej niż jedną szufladę nawigacyjną?

O: Aktywność może zawierać po jednej szufladzie nawigacyjnej dla każdej pionowej krawędzi układu. W celu dodania drugiej szuflady do układu `DrawerLayout` trzeba dodać drugi widok `NavigationView` poniżej pierwszego.

Kompletny kod aktywności MainActivity

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

Poniżej przedstawiliśmy pełny kod aktywności *MainActivity.java*; zaktualizuj swoją wersję tego pliku, by była identyczna z naszą:




```
package com.hfad.kociczat;

import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.support.v7.widget.Toolbar;
import android.support.v4.app.Fragment;
import android.support.v4.app.FragmentTransaction;
import android.support.v4.widget.DrawerLayout;
import android.support.v7.app.ActionBarDrawerToggle;
import android.support.design.widget.NavigationView;
import android.view.MenuItem;
import android.content.Intent;
import android.support.v4.view.GravityCompat;

public class MainActivity extends AppCompatActivity
                    implements NavigationView.OnNavigationItemSelectedListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        ActionBarDrawerToggle toggle =
                new ActionBarDrawerToggle(this,
                                          drawer,
                                          toolbar,
                                          R.string.nav_open_drawer,
                                          R.string.nav_close_drawer);

        drawer.addDrawerListener(toggle);
        toggle.syncState();
```

Używamy tych dodatkowych klas, więc musimy je zaimportować.

Zaimplementowanie tego interfejsu oznacza, że aktywność może nasłuchiwać zdarzeń kliknięć.

Ten fragment kodu dodaje przełącznik szuflady.

Dalsza część kodu znajduje się na następnej stronie.

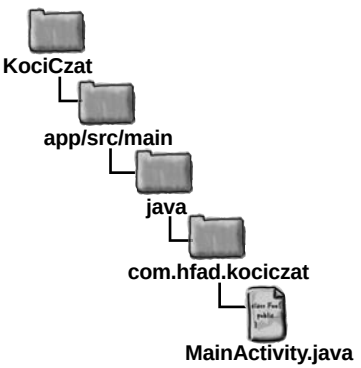
MainActivity.java (ciąg dalszy)

- ☑ Fragmenty i aktywności
- ☑ Nagłówek
- ☑ Opcje
- → ☐ Szuflada

```
    NavigationView navigationView = (NavigationView) findViewById(R.id.nav_view);
    navigationView.setNavigationItemSelectedListener(this);

    Fragment fragment = new InboxFragment();
    FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
    ft.add(R.id.content_frame, fragment);
    ft.commit();
}

@Override
public boolean onNavigationItemSelected(MenuItem item) {
    int id = item.getItemId();
    Fragment fragment = null;
    Intent intent = null;
    switch(id){
        case R.id.nav_drafts:
            fragment = new DraftsFragment();
            break;
        case R.id.nav_sent:
            fragment = new SentItemsFragment();
            break;
        case R.id.nav_trash:
            fragment = new TrashFragment();
            break;
        case R.id.nav_help:
            intent = new Intent(this, HelpActivity.class);
            break;
        case R.id.nav_feedback:
            intent = new Intent(this, FeedbackActivity.class);
            break;
        default:
            fragment = new InboxFragment();
    }
```

To wywołanie rejestruje aktywność w widoku NavigationView jako obiekt nasłuchujący zdarzeń.

Ta metoda jest wywoływana, kiedy użytkownik kliknie jeden z elementów umieszczonych w szufladzie nawigacyjnej.



Dalsza część kodu znajduje się na następnej stronie.

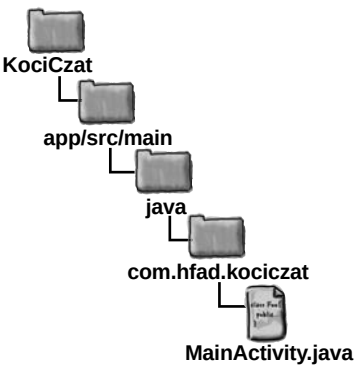
MainActivity.java (ciąg dalszy)


```
        if (fragment != null) {
            FragmentTransaction ft = getSupportFragmentManager().beginTransaction();
            ft.replace(R.id.content_frame, fragment);
            ft.commit();
        } else {
            startActivity(intent);
        }

        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        drawer.closeDrawer(GravityCompat.START);
        return true;
    }

    @Override
    public void onBackPressed() {
        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        if (drawer.isDrawerOpen(GravityCompat.START)) {
            drawer.closeDrawer(GravityCompat.START);
        } else {
            super.onBackPressed();
        }
    }
}
```

Wyświetlamy odpowiedni fragment lub odpowiednią aktywność, zależnie od opcji wybranej przez użytkownika w szufladzie nawigacyjnej.

To wywołanie zamyka szufladę po wybraniu jednej z opcji.

Kiedy użytkownik naciśnie przycisk Wstecz, zamykamy szufladę, jeśli będzie otworzona.

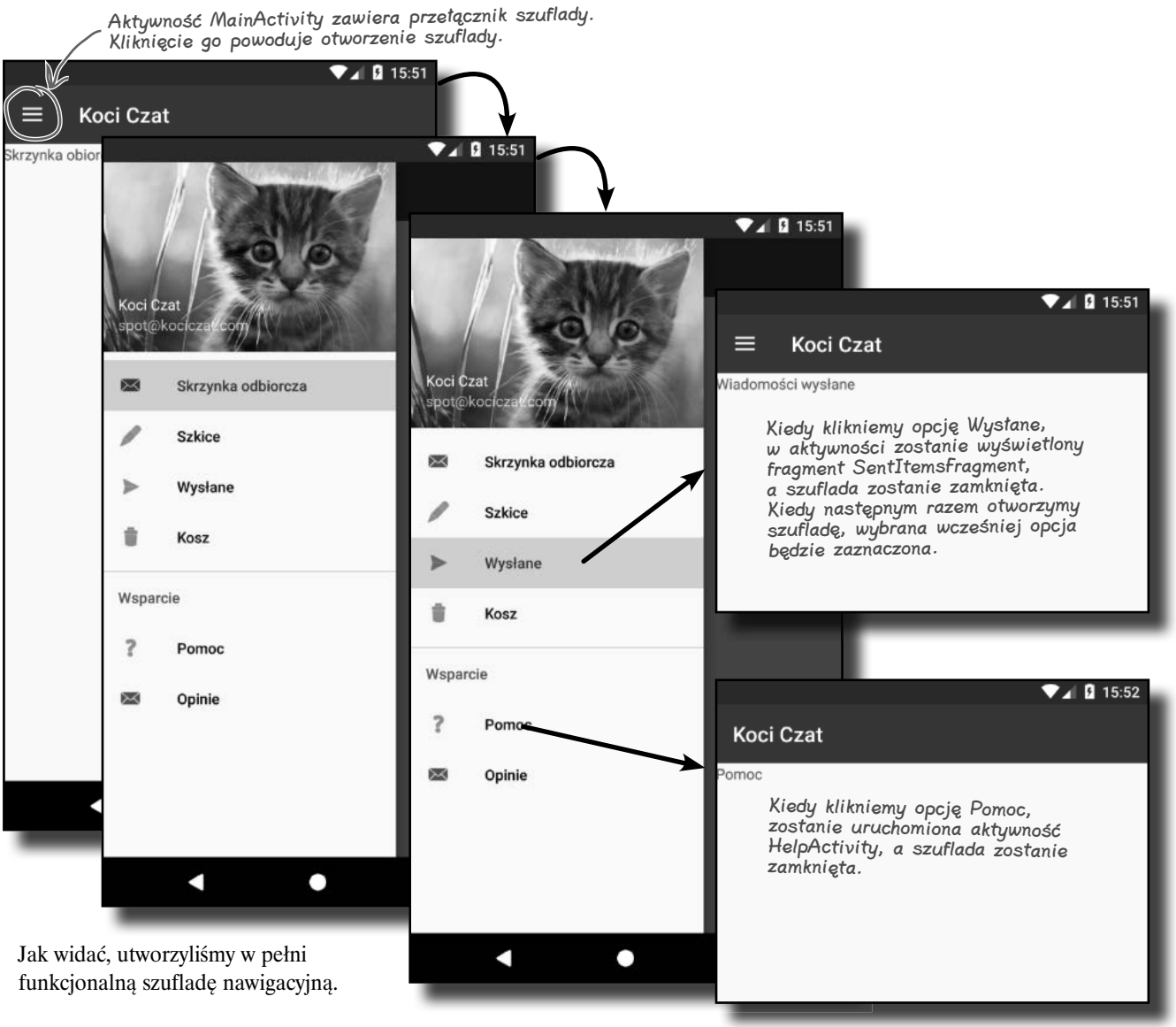


Sprawdźmy teraz, co się stanie po uruchomieniu naszej aplikacji.

Jazda próbna aplikacji

- Fragmenty i aktywności
- Nagłówek
- Opcje
- → Szuflada

Po uruchomieniu aplikacji ikona przełącznika szuflady zostaje wyświetlona na pasku narzędzi. Kliknięcie tej ikony powoduje otworzenie szuflady. Kliknięcie jednej z pierwszych czterech opcji widocznych w szufladzie powoduje wyświetlenie w aktywności `MainActivity` odpowiedniego fragmentu i zamknięcie szuflady; kiedy użytkownik ponownie wyświetli szufladę, wybrana wcześniej opcja będzie wyróżniona. Kliknięcie którejś z dwóch ostatnich opcji widocznych w szufladzie powoduje uruchomienie odpowiedniej aktywności.



Jak widać, utworzyliśmy w pełni funkcjonalną szufladę nawigacyjną.

Twój przybornik do Androida

Opanowałeś już rozdział 14. i dodałeś do swojego przybornika z narzędziami szuflady nawigacyjne

Pełny kod przykładowej aplikacji prezentowanej w tym rozdziale możesz pobrać z serwera FTP wydawnictwa Helion: ftp://ftp.helion.pl/przyklady/andrr2.zip

CELNE SPOSTRZEŻENIA

- Używaj szuflady nawigacyjnej, jeśli chcesz udostępnić użytkownikom znaczną liczbę skrótów lub pogrupować je w sekcje.
- Aby utworzyć szufladę nawigacyjną, dodaj do układu aktywności układ **`DrawerLayout`**. Pierwszym elementem tego układu musi być widok definiujący główną zawartość aktywności; zazwyczaj jest to układ z komponentami `Toolbar` i `FrameLayout`. Drugi element definiuje zawartość szuflady, przy czym zazwyczaj jest to widok `NavigationView`.
- Widok **`NavigationView`** pochodzi z biblioteki wsparcia wzornictwa. Obsługuje on większość możliwości funkcjonalnych szuflady.
- Nagłówek szuflady określa się, tworząc kolejny układ i podając jego identyfikator w atrybucie `headerLayout` widoku `NavigationView`.
- Elementy dodajemy do menu, tworząc zasób menu i podając jego identyfikator w atrybucie `menu` widoku `NavigationView`.
- Elementy w pliku menu dodajemy w takiej kolejności, w jakiej mają być widoczne w szufladzie.
- Jeśli chcesz wyróżnić element szuflady kliknięty przez użytkownika, to dodaj elementy menu do grupy i dodaj do niej atrybut `checkableBehavior` o wartości `"single"`.
- Użyj obiektu **`ActionBarDrawerToggle`**, aby wyświetlić na pasku narzędzi aktywności ikonę „hamburgera”. Stanowi ona wizualny sygnał informujący o tym, że aktywność udostępnia szufladę nawigacyjną. Kliknięcie tej ikony otwiera szufladę.
- Aby odpowiadać na kliknięcia opcji szuflady, zaimplementuj w aktywności interfejs **`NavigationView.OnNavigationItemSelectedListener`**. Następnie zarejestruj aktywność w widoku `NavigationView` jako obiekt nasłuchujący i zaimplementuj w niej metodę `onNavigationItemSelected()`.
- Aby zamknąć szufladę, należy wywołać metodę `closeDrawer()` obiektu `DrawerLayout`.

Skorowidz

F

G

H

I

J

K

N

O

P

R

S

Ś

T

O'REILLY®

Rusz głową!

Android

Programowanie aplikacji

Od poprzedniego wydania tej książki minęło parę lat, a kariera Androida wciąż jest dynamiczna! Kompleksowość, otwarty kod źródłowy, modułowa architektura, znakomita elastyczność — to wszystko sprawia, że lawinowo rośnie rzesza ludzi, którzy wybierają właśnie tę platformę. Liczbę urządzeń pracujących pod kontrolą Androida podaje się w miliardach, a najpewniej będzie ich o wiele więcej. To nie tylko telefony, komputery i tablety, ale także telewizory, inteligentne lodówki czy pralki, a nawet sztuczne satelity. Umiejętność efektywnego programowania dla Androida i dobry pomysł na świetną aplikację skazują dewelopera na sukces!

Sięgnij po ten nietypowy podręcznik! Możesz uznać jego formę i sposób przekazywania treści za dziwny, ale prędko się przekonasz, że jest wyjątkowo skuteczny. Twój mózg się zaangażuje i błyskawicznie przyswoi techniki programowania dla Androida. A to wszystko dzięki nowatorskiemu podejściu autorów, którzy uznali, że najszybciej uczymy się wtedy, gdy uwzględniamy specyfikę działania własnego mózgu! Dowiesz się, jak przygotować sobie warsztat pracy, czyli Android Studio. Od razu zaprojektujesz strukturę aplikacji i zbudujesz dobry interfejs. Będziesz swobodnie posługiwać się aktywnościami, intencjami, usługami. Poznasz narzędzia: Gradle, ART i ADB, dowiesz się, jak wykorzystywać bazy danych SQLite. A potem będzie jeszcze ciekawiej...

W tej książce znajdziesz między innymi:

- zasady tworzenia aplikacji interaktywnych
- istotne koncepcje, w tym: aktywności, intencje, usługi, układy i fragmenty
- biblioteki wsparcia, zadania asynchroniczne
- bazy danych i kursory
- uprawnienia i zarządzanie uprawnieniami

**Neurony płoną.
Emocje szaleją.
Oto powstaje apka
dla Androida!**

Dawn Griffiths i David Griffiths są z wykształcenia matematykami, zawodowo — znakomitymi programistami o ogromnym doświadczeniu i autorami kilku książek z serii *Rusz głową!* — a prywatnie — dobranym małżeństwem. Dawn w wolnych chwilach zastanawia się nad kolejną książką, doskonali tai-chi, biega i podróżuje. David niegdyś pracował jako instruktor zwinnych metod programowania. Dziś wolne chwile najbardziej lubi spędzać z żoną.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

KOD KORZYŚCI
Sięgnij po więcej!

ISBN 978-83-283-4079-4

9 788328 340794

INFORMATYKA W NAJLEPSZYM WYDANIU

Cena: 119,00 zł