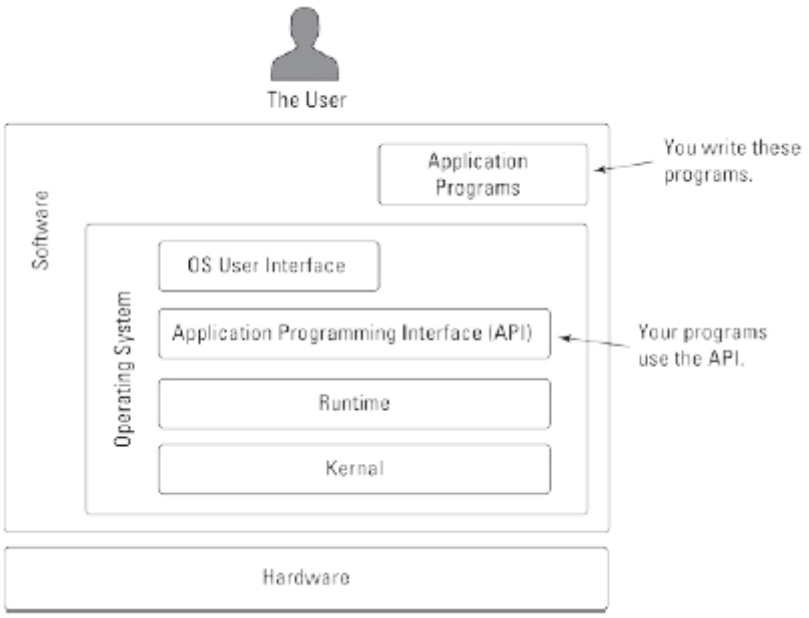
**Co to jest Flutter?**

Kilka lat temu wygrałem smartfon w loterii na konferencji deweloperów aplikacji. Co za radość wygrać coś! Doświadczenie sprawiło, że poczułem, że cały kosmos mi sprzyja. W końcu bateria telefonu stała się tak słaba, że musiałem ją ładować co godzinę. Nie zdawałem sobie sprawy, że telefon jest nadal na gwarancji, więc sam próbowałem wymienić baterię telefonu. Kupiłem nową baterię od sprzedawcy internetowego. Instrukcje mówiły mi, jak rozebrać obudowę, odpiąć połączenia obwodów i wyjąć starą baterię z uchwytu. Wszystko poszło ładnie aż do części o wyjęciu starej baterii. Instrukcje mówiły, aby wyciągnąć małą zakładkę, ale nie mogłem znaleźć zakładki. Więc przez kilka minut próbowałem uchwycić baterię. Bateria nie drgnęła, więc znalazłem mały śrubokręt i próbowałem wyrwać baterię z ciasnego otoczenia. Wtedy usłyszałem trzask, poczułem zapach dymu i zdałem sobie sprawę, że bateria telefonu się zapaliła. Przewiń do następnego popołudnia. Wędrowałem obok sklepu z elektroniką, więc wszedłem i zapytałem, czy sklepikarz może naprawić mój telefon. – Tak – powiedział. „Przynieś to następnym razem, gdy będziesz w okolicy. Mogę naprawić każdy telefon.” Powinieneś zobaczyć wyraz twarzy sklepikarza, kiedy tego samego dnia przyniosłem zwęglony, zgięty, ledwo rozpoznawalny telefon. Wciąż pamiętam tę historię baterii telefonu od początku do końca. Pamiętam radość z wygrania darmowego telefonu, szok, gdy zobaczyłem go w płomieniach i przerażenie na twarzy sklepikarza. Ale moje najpotężniejsze wspomnienie pochodzi z momentu, w którym otworzyłem etui z telefonem: wewnątrz tego małego etui zobaczyłem wystarczająco dużo obwodów, które przyprawiły mnie o zawroty głowy. Po wykonaniu pewnych prac elektrycznych we własnym domu obsługiwałem grube przewody o średnicy 10 i mocne 220-woltowe złącza. Wymieniłem karty dźwiękowe komputerów stacjonarnych, dyski twarde laptopów i dysk SSD w ciasno zapakowanym MacBooku Air. Ale ten smartfon był niesamowity. Płytka drukowana wyglądała jak mikrochip sam w sobie. Złącza były tak małe, że zastanawiałem się, w jaki sposób sygnały mogą przez nie przecisnąć. Nie ma co do tego wątpliwości: telefony komórkowe to skomplikowane bestie. Jak więc działają? Co je napędza? Co dzieje się w każdym z tych niezwykłych gadżetów?

**Sprzęt i oprogramowanie (rzeczy, które już znasz)**

Telefon komórkowy to naprawdę mały komputer. I jak każdy komputer, telefon komórkowy działa na kilku warstwach. Rysunek 1 pokazuje kilka z tych warstw.



Sprzęt to coś, czego możesz dotknąć .Sprzęt składa się z elementów takich jak obwody, pamięć i bateria. Sygnały elektryczne, które przemieszczają się po obwodach sprzętu, sprawiają, że sprzęt robi to, co chcesz. Sygnały te kodują instrukcje. Traktowane jako całość, instrukcje te nazywane są oprogramowaniem. Kiedy ludzie tworzą oprogramowanie, nie opisują każdego sygnału elektrycznego, który przechodzi przez obwody sprzętowe. Zamiast tego ludzie piszą kod źródłowy - instrukcje, które wyglądają jak instrukcje w języku angielskim. Jedna instrukcja kodu źródłowego może być skrótem dla setek lub tysięcy sygnałów elektrycznych. Zbiór instrukcji kodu źródłowego, które wykonują określone zadanie (przetwarzanie tekstu, przeglądanie stron internetowych, zarządzanie inteligentnym termostatem lub cokolwiek) nazywa się programem. Osoba, która pisze te instrukcje, jest programistą. Użytkownikiem jest osoba, która uruchamia program na własnym urządzeniu. Tak jak ludzie komunikują się za pomocą wielu języków mówionych, programiści piszą kod źródłowy przy użyciu wielu języków programowania. Jeśli tworzysz aplikacje na iPhone′a, prawdopodobnie piszesz kod w języku Swift lub języku Objective-C. Jeśli tworzysz aplikacje na Androida, prawdopodobnie napiszesz kod w Kotlin lub Javie. Tworząc aplikację Flutter, piszesz kod w języku programowania Dart. Oto kompletny program językowy Dart:

```
main() => print('Hello');
```

Ten program wyświetla na ekranie słowo Hello. Nie jest to zbyt przydatne, ale prosimy o cierpliwość. Rozróżniamy dwa rodzaje oprogramowania:

* Oprogramowanie systemu operacyjnego (OS) działa zawsze, gdy urządzenie jest włączone. Oprogramowanie systemu operacyjnego zarządza urządzeniem i zapewnia sposoby interakcji użytkownika z urządzeniem. Urządzenia firmy Apple, takie jak iPhone i iPad, działają pod kontrolą systemu operacyjnego iOS. Na telefonach i tabletach z Androidem działa system operacyjny Android (oczywiście).

* Programy użytkowe wykonują pracę, którą chcą wykonać użytkownicy. Aplikacje do wykonywania połączeń telefonicznych, aplikacje do czytania poczty e-mail, aplikacje kalendarza, przeglądarki internetowe i gry to przykłady programów użytkowych. Jako programista Flutter Twoim zadaniem jest tworzenie programów użytkowych.

Według niektórych szacunków popularny system operacyjny o nazwie Linux składa się z prawie 28 milionów instrukcji. Nikt nie jest w stanie poradzić sobie z tak dużą ilością kodu, więc systemy operacyjne są podzielone na osobne warstwy. Rysunek 1 pokazuje tylko cztery z wielu warstw typowego systemu operacyjnego:

* Jądro wykonuje najbardziej podstawowe zadania systemu operacyjnego. Jądro planuje uruchamianie aplikacji, zarządza pamięcią i plikami urządzenia, zapewnia dostęp do danych wejściowych i wyjściowych oraz wykonuje wiele innych ważnych zadań.

* Środowisko wykonawcze to zestaw kodu, który wykonuje dodatkową pracę w tle podczas działania aplikacji. Środowisko wykonawcze ma wiele kształtów i rozmiarów. Środowisko wykonawcze języka programowania C składa się ze stosunkowo niewielkiej ilości kodu. Natomiast Java , środowisko uruchomieniowe języka (wirtualna maszyna Java lub JVM) to duże oprogramowanie z wieloma ruchomymi częściami. Kiedy uruchamiasz aplikację na iOS, aplikacja korzysta ze środowiska wykonawczego Objective-C. Gdy uruchamiasz aplikację na Androida, ta aplikacja korzysta ze środowiska uruchomieniowego Androida, znanego również jako ART.

* Interfejs programowania aplikacji (API) to zestaw kodu, z którego programiści aplikacji korzystają wielokrotnie.

Na przykład API Androida ma coś o nazwie toUpperCase. Jeśli złożysz zgłoszenie na wielkie litery do „Flutter For Dummies”, otrzymasz „FLUTTER FOR DUMMIES”. Nie musisz pisać własnego kodu, aby zmienić każdą z liter. Interfejs API Androida zapewnia tę funkcjonalność. Wszystko, co musisz zrobić, to powiedzieć interfejsowi API Androida, aby zastosował funkcję toUpperCase i gotowe. Oto kilka przydatnych terminów: Zamiast mówić API, aby „zastosował swoją funkcję toUpperCase”, wywołujesz toUpperCase. To użycie słowa call wywodzi się z języka programowania FORTRAN z lat 50. XX wieku. Systemy operacyjne nie opanowały rynku API. Wszystkie rodzaje oprogramowania są dostarczane z interfejsami API. Flutter i Dart mają własne API. Interfejs API Darta ma funkcje ogólnego przeznaczenia, takie jak toUpperCase, isAtSameMomentAs i kilka innych. API Fluttera ma funkcje, które dotyczą aplikacji zorientowanych wizualnie. Na przykład, gdy chcesz wyświetlić pole, w którym użytkownik może wpisać tekst, nie musisz opisywać każdego aspektu wyglądu i zachowania pola. Zamiast tego możesz wywołać interfejs API TextField konstruktora i niech Flutter wykona za Ciebie ciężką pracę. Czasami nazywam API biblioteką. Wypożyczasz książki z biblioteki publicznej i wypożyczasz istniejący kod z interfejsów API Dart i Flutter.

W terminologii programowania Dart biblioteka słów ma nieco inne znaczenie. Nie musisz się jeszcze o to martwić. W większości opisuję fragmenty interfejsów API Darta i Fluttera, a następnie sposób, w jaki używasz tych fragmentów do tworzenia programów Flutter.

Typowe API ma tysiące sztuk. Nikt nie zapamiętuje ich wszystkich. Jeśli chcesz dodać obraz do swojej aplikacji, otwórz dokumentację Flutter i wyszukaj słowo Obraz. Strona Obraz w dokumentacji zawiera informacje o tym, jak wyświetlić obraz, jak zmienić jego rozmiar, jak kafelkować obraz i jak robić wszelkiego rodzaju inne dobre rzeczy.

* Interfejs użytkownika systemu operacyjnego to obszar, który zawiera ekran główny, ikony uruchamiania, eksplorator plików i wszelkie inne rzeczy, które użytkownicy widzą, gdy nie pracują z określoną aplikacją. Na swoim laptopie prawdopodobnie masz pulpit zamiast ekranu głównego. W ten czy inny sposób system operacyjny przedstawia opcje, które pomagają użytkownikom uruchamiać aplikacje i wykonywać inne zadania konserwacyjne. Te opcje są częścią interfejsu użytkownika systemu operacyjnego. Każda warstwa na rysunku 1 zawiera zbiór powiązanych komponentów. Pomaga to programistom skupić się na komponentach, które ich najbardziej dotyczą - na przykład:

* Interfejs API zawiera kod, który pomaga programistom w pisaniu programów użytkowych. Programista, który tworzy aplikację do zakupów online, szuka komponentów w interfejsie API.

* Warstwa Runtime zawiera kod do wydajnego uruchamiania programów.

Aby każdy kod działał szybciej, inżynierowie Apple ulepszają warstwę iOS Runtime.

Oprócz oddzielania części kodu od siebie, warstwy tworzą zorganizowane ścieżki komunikacji między częściami systemu. Ogólnie kod warstwy komunikuje się tylko z warstwami bezpośrednio nad i pod nią. Na przykład użytkownik naciska przycisk należący do aplikacji pogodowej. Aplikacja odpowiada, wywołując funkcjonalność udostępnianą przez API. Komunikacja przebiega w dół diagramu na rysunku 1, aż dotrze do sprzętu, który odpowiada, zmieniając piksele na ekranie urządzenia. Użytkownik nigdy nie komunikuje się bezpośrednio z API, a aplikacje nie mają bezpośredniego dostępu do jądra systemu operacyjnego.

**KOD, KTÓREGO MOŻESZ UŻYĆ**

We wczesnych latach osiemdziesiątych mój kuzyn pracował w firmie zajmującej się oprogramowaniem komputerowym. Firma napisała kod dla maszyn do przetwarzania tekstu. (W tamtym czasie, jeśli chciałeś pisać dokumenty bez maszyny do pisania, kupowałeś „komputer”, który nie robił nic poza

przetwarzaniem tekstu). Chris narzekał, że ciągle proszono go o pisanie tego samego starego kodu. „Najpierw piszę program do wyszukiwania i zamiany. Potem piszę sprawdzanie pisowni. Następnie piszę inny program do wyszukiwania i zamiany. Następnie inny rodzaj sprawdzania pisowni. A potem lepszy program do wyszukiwania i zastępowania". Jak Chrisowi udało się utrzymać zainteresowanie swoją pracą? A jak pracodawca Chrisa zdołał utrzymać się w biznesie? Co kilka miesięcy Chris musiał wymyślać koło na nowo – wyrzucić stary program „wyszukaj i zamień" i napisać nowy program od zera. To nieefektywne. Co gorsza, jest nudno. Przez lata profesjonaliści komputerowi szukali świętego Graala – sposobu na napisanie oprogramowania, które będzie łatwe do ponownego użycia. Nie pisz i nie przepisuj kodu wyszukiwania i zamiany. Po prostu podziel zadanie na małe kawałki. Jeden fragment kodu wyszukuje pojedynczy znak, inny fragment szuka spacji, a trzeci fragment zastępuje jedną literę inną. Kiedy masz już wszystkie elementy, po prostu złóż je, aby utworzyć program wyszukiwania i zastępowania. Później, gdy myślisz o nowej funkcji oprogramowania do przetwarzania tekstu, składasz elementy w nieco inny sposób. To rozsądne, opłacalne i o wiele przyjemniejsze. Pod koniec lat 80. nastąpiło kilka postępów w tworzeniu oprogramowania, a na początku lat 90. wiele dużych projektów programistycznych było pisanych z prefabrykowanych komponentów. Dla konkretnego projektu lub określonego języka programowania te prefabrykowane komponenty utworzyły bibliotekę kodu wielokrotnego użytku. To były narodziny nowoczesnego API. Tworząc aplikację Flutter, używasz języka programowania Dart. Dart i Flutter mają oddzielne API:

* API Darta zajmuje się zadaniami, które każdy język programowania powinien być w stanie wykonać, bez względu na to, co programiści chcą robić z tym językiem.

Na przykład API Darta pomaga programistom zaokrąglić liczbę, przyciąć ciąg znaków, opisać przedział czasu, odwrócić listę i tak dalej.

* API Fluttera zajmuje się prezentacją komponentów i obrazów na ekranie urządzenia.

Jedna część API Fluttera dotyczy przycisków, pól tekstowych, pól wyboru i tym podobnych. Kolejna część zajmuje się gestami użytkownika. Jeszcze inny obejmuje animację. Każdy program Dart, nawet najprostszy, wywołuje kod w Dart API, a każda aplikacja Flutter wywołuje zarówno Dart, jak i Flutter API. Te interfejsy API są zarówno przydatne, jak i budzące grozę. Są przydatne ze względu na wszystkie rzeczy, które możesz zrobić z kodem API. Są ogromne, ponieważ oba interfejsy API są rozbudowane. Nikt nie zapamiętuje wszystkich funkcji udostępnianych przez interfejsy API Dart i Flutter. Programiści zapamiętują funkcje, z których często korzystają, i w mgnieniu oka wyszukują funkcje, których potrzebują. Wyszukują te funkcje na stronie internetowej zwanej dokumentacją referencyjną Flutter API. Dokumentacja API (zobacz https://.api.flutter.dev) opisuje funkcje w API Dart i Flutter. Jako programista Flutter codziennie sprawdzasz dokumentację API. Możesz dodać do zakładek witrynę i ponownie ją odwiedzić, gdy tylko chcesz coś wyszukać.

**Gdzie pasuje Flutter?**

Sercem Fluttera jest API do tworzenia aplikacji. Większość aplikacji Flutter działa na urządzeniach mobilnych, ale aplikacje Flutter mogą działać również na laptopach i komputerach stacjonarnych. Flutter z pewnością nie był pierwszym API dla urządzeń mobilnych, więc dlaczego ktoś miałby rozważać używanie Fluttera do tworzenia aplikacji?

**Rozwój międzyplatformowy**

Moja ulubiona knajpa z burgerami reklamowała nową mobilną aplikację do zamawiania. Potrzebowałem aplikacji, aby szybko wyskoczyć z pociągu podmiejskiego, złapać burgera i pobiec na pobliskie spotkanie techniczne. Robiłem to kilka razy w miesiącu. Ale miałem problem: aplikacja

działała tylko na iPhonie, a ja miałem telefon z Androidem. Za kulisami twórcy aplikacji z burger jointa ciężko pracowali nad konwersją swojej aplikacji na iPhone'a na aplikację na Androida. Nie było to drobne zadanie, ponieważ API Androida nie rozpoznaje tych samych poleceń, co API iPhone'a. Przejście z jednego API do drugiego nie jest proste. Nie chodzi o wprowadzanie rutynowych zmian w kodzie. Aby dokonać konwersji z jednego rodzaju telefonu na inny, programiści przepisują tysiące (a może nawet miliony) linii kodu. Proces jest czasochłonny i kosztowny. Więc czekałem i czekałem, aż restauracja będzie miała aplikację na Androida. Byłam tak zdesperowana na pyszny cheeseburger, że w końcu się złamałam i kupiłam drugi telefon. Ale okazało się, że to zły pomysł. Gdy tylko pojawił się mój nowy iPhone, burger place wypuścił swoją błyszczącą, nową aplikację na Androida. Cała historia sprowadza się do rzeczy zwanych platformami. Ludzie rzucają się po platformie słownej, jakby słowo oznaczało wszystko i nic. Ale moim zdaniem platforma to szczególny system operacyjny wraz ze sprzętem, na którym działa system operacyjny. Co odróżnia platformę Android od jej odpowiednika na iOS? Aby utworzyć przyciski radiowe w API Androida, piszesz następujący kod:

```
< RadioGroup >

< RadioButton

android:id="@+id/radioButton1"

android:text="Red"

android:onClick="onRadioButtonClicked"/ >

< RadioButton

android:id="@+id/radioButton2"

android:text="Yellow"

android:onClick="onRadioButtonClicked "/>

< RadioButton

android:id="@+id/radioButton3"

android:text="Green"

android:onClick="onRadioButtonClicked"/ >

< /RadioGroup >
```

Spróbuj przekonwertować ten kod, aby działał na iPhonie. Interfejs API iOS nie ma przycisków radiowych, więc aby dostosować aplikację na Androida z przyciskami opcji dla iOS, piszesz kod, aby rzeczy wyglądały jak przyciski radiowe. Kodujesz również reguły, których mają przestrzegać przyciski radiowe – zasady takie jak „można wybrać tylko jeden przycisk na raz”. Jeśli nie chcesz tworzyć przycisków radiowych od zera, możesz zastąpić przyciski radiowe Androida komponentem selektora iOS, który wygląda jak stary licznik samochodowy. Tak czy inaczej, wymiana komponentów aplikacji zajmuje czas i kosztuje. Niektóre firmy rezygnują i tworzą aplikacje tylko na jedną platformę - iPhone'a lub Androida. Inne firmy zatrudniają dwa zespoły programistów - jeden do tworzenia iPhone'a, a drugi do rozwoju Androida. Jeszcze inne firmy mają jeden zespół programistów, którzy pracują nad obiema wersjami kodu. Dla menedżerów firm problem jest irytujący. Po co wydawać prawie dwa razy więcej pieniędzy i tworzyć dwie aplikacje, które robią prawie to samo? Społeczność programistów ma nazwy dla tej brzydkiej sytuacji:

* Oprogramowanie napisane dla jednej platformy nie jest kompatybilne z innymi platformami.

* Rynek telefonów komórkowych cierpi na fragmentację: rynek jest podzielony między dwa różne systemy operacyjne, a połowa Androida jest podzielona między telefony wielu dostawców.

Program, który bezpośrednio korzysta z interfejsu API systemu Android lub iOS, nazywany jest kodem natywnym, a kod natywny napisany dla systemu Android nie może działać na urządzeniu z systemem iOS. W ten sam sposób kod natywny napisany dla iOS jest bez znaczenia dla urządzenia z Androidem. Co ma robić programista? Framework to API drugiego poziomu. Co to do cholery oznacza? Framework to interfejs API, który służy jako pośrednik między programistą a innym interfejsem API. Jeśli bezpośrednie korzystanie z API Androida lub iOS jest problematyczne, przełączasz się na API frameworka. API frameworka mierzy się z problemami Androida i iOS. Struktury takie jak Flutter oferują alternatywę dla tworzenia aplikacji natywnych. Kiedy piszesz program Flutter, nie piszesz kodu specjalnie dla Androida lub iOS. Zamiast tego piszesz kod, który można przetłumaczyć na wywołania API dowolnego systemu. Oto jak tworzysz przyciski radiowe we frameworku Flutter:

```
Radio(

value: TrafficLight.Red,

groupValue: _trafficLightValue,

onChanged: _updateTrafficLight,

),

Radio(

value: TrafficLight.Yellow,

groupValue: _trafficLightValue,

onChanged: _updateTrafficLight,

),

Radio(

value: TrafficLight.Green,

groupValue: _trafficLightValue,

onChanged: _updateTrafficLight,

)
```

Twój komputer tłumaczy kod tego rodzaju na wywołania interfejsu API systemu Android lub wywołania interfejsu API systemu iOS - albo oba. To super!

**Szybki i łatwy cykl rozwoju**

Być może słyszałeś historie o początkach programowania komputerowego. Pisałem programy FORTRAN-u i sam zapisywałem je na dużej talii kart dziurkowanych. Program na 600 linii ważył około 1400 gramów. Przenosiłem swój program z automatu do kart dziurkowanych do stanowiska operatora komputera, gdzie wiecznie gburowaty operator opowiadał mi o niezwykle długim czasie realizacji pracy. Cztery godziny później otrzymywałem z powrotem gruby plik papieru z komunikatem o błędzie gdzieś pośrodku. Wróciłem do maszyny do dziurkowania kart, zrobiłem kolejną kartę z dodanym

przecinkiem w 23. kolumnie i ponownie załatwiłem całą sprawę. Nie ma co do tego wątpliwości — długi i żmudny cykl rozwoju utrudnia produktywność. W dzisiejszych czasach skrócenie czasu realizacji o kilka sekund może mieć ogromne znaczenie. Oto, co się dzieje, gdy tworzysz aplikację na urządzenia mobilne:

1. Piszesz kod lub modyfikujesz istniejący kod. Nie piszesz kodu na Androida lub iOS na żadnym telefonie. Telefony nie są wystarczająco mocne, aby wszystko edytować i inne rzeczy, które musisz zrobić. Zamiast tego tworzysz kod aplikacji na laptopie lub komputerze stacjonarnym. Ten laptop lub komputer stacjonarny jest nazywany komputerem programistycznym.

2. Wydajesz polecenie dla swojego komputera deweloperskiego, aby zbudować kod. Budowanie kodu odbywa się w kilku etapach, z których jeden nazywa się kompilacją. Kompilacja oznacza automatyczne tłumaczenie twojego programu z napisanego kodu źródłowego na szczegółowe instrukcje kodu wynikowego. Pomyśl o kodzie wynikowym jak o zbiorze zer i jedynek. Jest bardzo szczegółowy i niezwykle nieintuicyjny. Ludzie prawie nigdy nie czytają ani nie zapisują kodu wynikowego, ale w istocie rzeczy procesory reagują tylko na instrukcje kodu wynikowego. Oprócz etapu tłumaczenia, proces budowania łączy napisany przez Ciebie program z dodatkowym kodem, którego Twój program potrzebuje do uruchomienia. Na przykład, jeśli program uzyskuje dostęp do Internetu, proces kompilacji integruje kod z istniejącym kodem sieciowym. Co się potem dzieje?

3. Komputer deweloperski wdraża kod na urządzeniu docelowym. To tak zwane „urządzenie” może być prawdziwym telefonem podłączonym do komputera lub obrazem telefonu na ekranie komputera. Tak czy inaczej, twój program zaczyna działać.

4. Naciskasz przyciski, wpisujesz tekst i w inny sposób testujesz swoją aplikację, aby dowiedzieć się, czy robi to, co chcesz. Oczywiście nie robi wszystkich tych rzeczy. Więc wracasz do kroku 1 i próbujesz dalej.

Kroki 2 i 3 mogą być boleśnie powolne. W przypadku niektórych prostych aplikacji na iPhone'a i Androida obserwowałem przez kilka minut, jak mój komputer przygotowuje kod do następnego uruchomienia programu. Ta ospałość znacznie zmniejsza moją produktywność. Ale wraz z Flutterem nadchodzą dobre wieści. Flutter używa języka programowania Dart, a Dart jest wyposażony w te dwa (policz - dwa) kompilatory:

* Kompilator z wyprzedzeniem (AOT)

Dzięki kompilatorowi AOT komputer programistyczny tłumaczy cały program i udostępnia przetłumaczony kod do uruchomienia urządzeń. Żadne dalsze tłumaczenie nie ma miejsca, gdy urządzenia uruchamiają twój program. Każde urządzenie docelowe poświęca swoją moc obliczeniową na wydajne działanie kodu. Aplikacja działająca na kodzie skompilowanym przez AOT działa płynnie i wydajnie.

* Kompilator just-in-time (JIT)

Dzięki kompilatorowi JIT komputer programistyczny tłumaczy wystarczającą ilość kodu, aby uruchomić aplikację. Przekazuje ten kod do urządzenia testowego i kontynuuje tłumaczenie, gdy urządzenie testowe uruchamia aplikację. Jeśli programista naciśnie przycisk na ekranie urządzenia testowego, JIT kompilator śpieszy się z przetłumaczeniem kodu tego przycisku. Aplikacja działająca na kompilatorze JIT może wydawać się powolna, ponieważ kompilator tłumaczy kod podczas działania aplikacji. Ale używanie kompilatora JIT to świetny sposób na przetestowanie aplikacji.

Oto, co się dzieje, gdy tworzysz aplikację Flutter:

1. Piszesz jakiś kod.

2. Wydajesz polecenie dla swojego komputera deweloperskiego, aby zbudować kod.

3. Komputer deweloperski wdraża kod na urządzeniu docelowym. Znowu masz do czynienia z zauważalnym opóźnieniem.

4. Podczas testowania kodu dowiadujesz się, że nie robi on wszystkich rzeczy, które chcesz, aby robił.

5. Modyfikujesz istniejący kod, a następnie…

6. Wydajesz polecenie dla komputera deweloperskiego, aby odbudować kod.

Oto, gdzie dzieje się magia Flutter. Kompilator Darta JIT ponownie kompiluje tylko zmodyfikowaną część aplikacji i wysyła zmianę bezpośrednio na urządzenie docelowe. Zmodyfikowany kod zaczyna działać w ułamku sekundy. Oszczędzasz godziny każdego dnia, ponieważ nie czekasz, aż zmiany kodu zaczną obowiązywać.

**CO TO JEST kompilator?**

Jesteś człowiekiem. (Oczywiście, każda reguła ma wyjątki. Ale jeśli czytasz tę książkę, prawdopodobnie jesteś człowiekiem.) W każdym razie, ludzie mogą napisać i zrozumieć następujący kod źródłowy Fluttera:

```
import 'package:flutter/widgets.dart';

main() => runApp(SizedBox`());
```

Kiedy parafrazujesz kod źródłowy w języku angielskim, oto co otrzymasz:

Pobierz kod (kod z Flutter API) o nazwie widgets.dart.

Uruchom aplikację, której jedynym komponentem jest widżet pudełkowy.

Jeśli nie widzisz podobieństw między kodem Flutter a jego angielskim odpowiednikiem, nie martw się. Jak większość ludzi, możesz nauczyć się czytać i pisać kod Flutter. Jeśli się zastanawiasz, ten kod źródłowy zawiera najprostszą i najbardziej bezużyteczną aplikację Flutter na świecie. Po uruchomieniu aplikacji zobaczysz całkowicie czarny ekran. To nie jest to coś , co nazwałbyś „zabójczą aplikacją”. Kod źródłowy jest fajny, ale kod źródłowy nie jest dla wszystkich i dla wszystkiego. Procesory w komputerach i urządzeniach mobilnych nie są ludźmi. Procesory nie postępują zgodnie z instrukcjami kodu źródłowego.

Zamiast tego postępują zgodnie z tajemniczymi instrukcjami tego rodzaju:

1100100 1100101 1111000 00001010 00110000

00110011 00110101 00000000 10000100

Te zera i jedynki to w rzeczywistości kilka pierwszych słów w wersji kodu aplikacji Black Screen na telefon z Androidem. Oto aplikacja Black Screen po zinterpretowaniu przez procesor zer i jedynek:

.class public

com/allmycode/dexperiment/MainActivity

.super

io/flutter/embedding/android/FlutterActivity

```
.source MainActivity.java
.method public <init>()V
.limit registers 1
; this: v0
(Lcom/allmycode/dexperiment/MainActivity;)
.line 8
invokedirect{
v0},io/flutter/embedding/android/FlutterActivity/<; <init>()V
return-void
.end method
.method public
configureFlutterEngine(Lio/flutter/embedding/engine/.limit registers 2
; this: v0
(Lcom/allmycode/dexperiment/MainActivity;)
; parameter[0] : v1
(Lio/flutter/embedding/engine/FlutterEngine;)
.line 11
invokestatic{
v1},io/flutter/plugins/GeneratedPluginRegistrant/;
registerWith(Lio/flutter/embedding/engine/FlutterEngine;).line 12
return-void
.end method
```

Co za bałagan! Ludzie nie chcą czytać ani pisać tego rodzaju instrukcji. Te instrukcje nie są instrukcjami kodu źródłowego Dart. Są to instrukcje kodu bajtowego Dalvik. Kiedy piszesz program Flutter, piszesz instrukcje kodu źródłowego Darta. Jeśli testujesz swój program na urządzeniu z systemem Android, komputer programistyczny tłumaczy kod źródłowy na kod bajtowy. Jeśli przetestujesz swój program na iPhonie, komputer przetłumaczy Twój kod źródłowy na coś, co jest jeszcze bardziej niejasne niż kod bajtowy. Narzędziem wykonującym tłumaczenie jest kompilator. Kompilator bierze kod, który możesz napisać i zrozumieć, i tłumaczy go na kod, z którego wykonaniem ma szanse walczyć procesor. Możesz umieścić swój kod źródłowy w pliku o nazwie main.dart. Aby uruchomić aplikację na urządzeniach z systemem Android, kompilator tworzy inne pliki o nazwach MainActivity.dex i app.apk. Zwykle nie zawracasz sobie głowy przeglądaniem tych skompilowanych plików. Nie możesz nawet sprawdzać plików .dex lub .apk za pomocą zwykłego edytora. Jeśli spróbujesz otworzyć MainActivity.dex za pomocą Notatnika, TextEdit lub nawet Microsoft Word, zobaczysz tylko kropki, zawijasy i inne gobbledygook. Nikt (z wyjątkiem kilku szalonych programistów w odosobnionych laboratoriach w odległych miejscach) nie pisze kodu bajtowego Dalvik ani żadnego innego rodzaju kodu, który

procesory faktycznie rozumieją. Gdy poprosisz komputer deweloperski o uruchomienie kodu, komputer używa własnego oprogramowania (kompilatora) do tworzenia instrukcji przyjaznych dla procesora. Jedynym powodem, aby spojrzeć na kod bajtowy na tym pasku bocznym, jest zrozumienie, jak ciężko pracuje Twój komputer programistyczny. Flutter oferuje dwa sposoby wprowadzania zmian w działającej aplikacji:

* W przypadku gorącego restartu aplikacja rozpoczyna działanie od nowa, usuwając wszelkie dane wprowadzone podczas ostatniego testu, wyświetlając aplikację tak, jakbyś uruchamiała ją po raz pierwszy.

* Przy przeładowaniu na gorąco aplikacja rozpoczyna pracę od miejsca, w którym została przerwana, z nienaruszonymi ostatnio wprowadzonymi danymi, jeśli to możliwe. Jedyne zmiany to te podyktowane przez Twoje modyfikacje w kodzie.

Gorący restart i gorące przeładowanie Fluttera są niesamowicie szybkie. Sprawiają, że cykl tworzenia aplikacji staje się przyjemnością, a nie obowiązkiem.

**Świetny sposób na myślenie o tworzeniu aplikacji**

Język, którym mówisz, wpływa na sposób myślenia. Jeśli mi nie wierzysz, spójrz na hipotezę Sapir-Whorf. Na pewno znajdziesz go na swojej ulubionej stronie lingwistycznej. Języki mówione nie są ani dobre, ani złe, ale języki programowania mogą mieć dobre i złe cechy. Większość aplikacji hybrydowych jest napisana w języku programowania JavaScript. Tak, JavaScript jest jednym z najczęściej używanych języków na świecie. Ale nie, JavaScript nie zachęca do dobrego projektowania oprogramowania. Łatwo jest napisać mylący kod w JavaScript, ponieważ jego reguły są dość liberalne. Możesz napisać niechlujny kod JavaScript, a kod działa dobrze. Oznacza to, że działa dobrze, dopóki ktoś nie wprowadzi nieoczekiwanego wejścia. Kiedy tak się dzieje, masz problem z ustaleniem, jak działał Twój kod. Nawet jeśli nie jesteś zajęty naprawianiem błędów, dodawanie nowych funkcji do kodu JavaScript może być trudne i frustrujące. Miłośnicy JavaScript będą się spierać z każdym słowem w tym akapicie, ale w taki czy inny sposób JavaScript ma swoje wady. Platforma iOS firmy Apple używa języków Swift i Objective-C, podczas gdy Android używa Kotlin i Java. Objective-C datuje się na początek lat 80. i podobnie jak ja pokazuje swój wiek. Pozostałe trzy języki radzą sobie całkiem nieźle w skali dobrych funkcji językowych, ale żaden z nich nie jest tak prosty i intuicyjny jak Dart. Co więcej, zarówno iOS, jak i Android dzielą kod aplikacji na dwie oddzielne części:

* Układ: wygląd aplikacji.

* Logika: sekwencja instrukcji wykonywanych przez aplikację.

Przykład przycisku radiowego Androida we wcześniejszej części „Rozwój międzyplatformowy” nie jest ani kodem Kotlina, ani Java. To kod XML. Ma inny format i znajduje się w innym pliku niż kod, który odpowiada na wybory przycisków radiowych. Twierdzę, że oddzielenie układu od logiki to dobra rzecz. Umieszcza różne aspekty aplikacji w różnych częściach kodu. Deweloperzy mogą zarządzać każdą częścią niezależnie. Dla programisty Androida to dobra rzecz. Ale to nie jest książka na Androida. To książka Fluttera. Tak więc twierdzę, że oddzielenie układu od logiki nie jest optymalne. Dlatego Być może słyszałeś wszechogarniającą mantrę tworzenia aplikacji Flutter:

We Flutterze prawie wszystko jest widżetem.

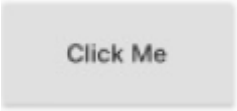
A czym jest widżet? W aplikacji mobilnej każdy przycisk jest jednym z widżetów aplikacji. Każde pole tekstowe to widżet. Sama aplikacja jest widżetem. Rozmieszczenie przycisków i pól tekstowych to widget. Animowanie obiektów z jednej części ekranu do drugiej to widżet. Tworząc aplikację Flutter,

umieszczasz widżety w innych widżetach, które z kolei znajdują się w jeszcze większej liczbie widżetów. Listing 1-1 zawiera fałszywy kod, który ilustruje tę kwestię:

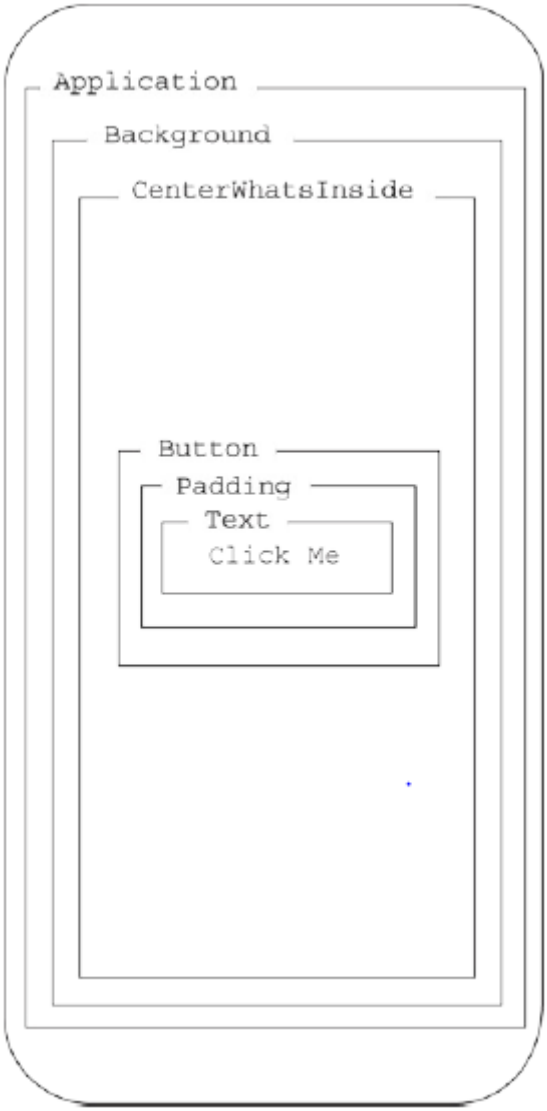
LISTING 1-1 Jak koło w kole

```
// Nie daj się nabrać na moje podstępy. To nie jest prawdziwe

Flutter code!

Application(

Background(

CenterWhateverIsInsideThis(

Button(

onPressed: print("I've been clicked."),

Padding(

Text(

"Click Me"

),

),

),

),

),

)
```

Listing 1 zawiera widżet Tekst wewnątrz widżetu Padding, który znajduje się wewnątrz widżetu Przycisk wewnątrz widżetu CenterWhateverIsInsideThis. Ten widżet CenterWhateverIsInsideThis znajduje się wewnątrz widżetu tła, który znajduje się wewnątrz widżetu aplikacji. Tworząc Listing 1, zamodelowałem go na podstawie prawdziwego kodu Fluttera. Prawdziwy kod Fluttera tworzy aplikację pokazaną na rysunku 2. Kiedy użytkownik naciśnie przycisk na rysunku 1-2, pojawią się słowa, na które kliknąłem.




Porównaj rysunki 2 i 3. Rysunek 2 przedstawia aplikację tak, jak widzi ją użytkownik. Rysunek 3 pokazuje tę samą aplikację, którą koduje programista Flutter.




Jeśli nie jesteś jeszcze programistą Flutter, widżet słów może sugerować widoczny komponent, taki jak przycisk, suwak, ikona lub coś podobnego. Ale we Flutterze rzeczy, które nie są tak naprawdę widoczne, to także widżety. Na przykład na listingu 1-1 CenterWhateverIsInsideThis jest widżetem. Posiadanie funkcji układu, takich jak widżety CenterWhateverIsInsideThis be, to potężny pomysł. Oznacza to, że programiści Flutter mogą skupić swoją uwagę na jednym nadrzędnym zadaniu - wpychaniu widżetów do innych widżetów. Flutter ma pewną prostotę i elegancję, której nie mają inne frameworki do tworzenia aplikacji. Flutter nie ma wbudowanego widżetu o nazwie CenterWhateverIsInsideThis. Ale nie bądź rozczarowany. Widżet Flutter's Center robi to, co powinien robić mój fikcyjny widżet CenterWhateverIsInsideThis.

**Dość nowej terminologii! Co dalej?**

Być może przeczytałeś tą część od początku do końca, ale żadne słowo nie skłoniło cię do dotknięcia klawiatury komputera. Jaka szkoda! Jeśli przeczytasz następną, mogę naprawić to okropne pominięcie.

**Konfigurowanie komputera do tworzenia aplikacji mobilnych**

Archimedes żył w starożytnej Grecji w II wieku pne. Jego praca nad wczesnym rozwojem matematyki była przełomowa. W ciszy i spokoju własnej wanny natknął się na ważny wzór — wzór opisujący zależność między jego wagą a ilością wody wypartej przez jego ciało. Po dokonaniu tego odkrycia krzyknął „Eureka!" i wyskoczył z wanny, żeby powiedzieć wszystkim w swoim mieście. Tunika czy bez tuniki, Archimedes chciał, aby wszyscy Grecy uczestniczyli w jego radości. Skocz do przodu o około 2200 lat. Mam 12 lat. W rozmowie na temat przesuwania mebli mój ojciec cytuje Archimedesa: „Daj mi wystarczająco dużą dźwignię, a poruszę świat". Próbuję to sobie zobrazować. Oto Ziemia, wisząca w kosmosie, a tam jest długi, długi kij z zawinięciem na dole. Zwinięcie sięga pod dużą planetę i przesuwa ją w nowe miejsce. Śmierdzący ze mnie, widziałem trzy problemy z tym scenariuszem. Po pierwsze, w jaki sposób zakotwiczyłbyś dolną część dźwigni w stałym punkcie w pustej przestrzeni? Po drugie, jak powstrzymałbyś dźwignię przed wykopaniem ogromnej dziury w jakiejś miękkiej części Ziemi? I wreszcie, jak ludzie na Ziemi przyjęliby wyrzucenie ich planety z orbity? Jak widać, byłem wrednym dzieckiem. Ale pomysł z dźwignią utkwił mi w głowie. Im dłuższa dźwignia, tym więcej korzyści z niej uzyskasz. Nie możesz samodzielnie podnieść dużego głazu bez dźwigni, ale możesz przesunąć głaz za pomocą ogromnej dźwigni. Dźwignia to narzędzie, a narzędzia to wspaniałe rzeczy. Narzędzia nie wykonują bezpośrednio rzeczy, które chcesz zrobić. Nie możesz zjeść narzędzia, przeczytać dobrego narzędzia, usłyszeć wesołej piosenki narzędzia ani zatańczyć jig z narzędziem. Ale możesz używać narzędzi do robienia jedzenia, książek, instrumentów muzycznych i parkietów tanecznych. Ta część dotyczy narzędzi - narzędzi, których używasz do tworzenia świetnych aplikacji mobilnych.

Rzeczy, których potrzebujesz

Dowiesz się, jak tworzyć aplikacje za pomocą Fluttera. Zanim zaczniesz tworzyć aplikacje, potrzebujesz kilku narzędzi programowych. Oto lista potrzebnych narzędzi:

* Zestaw deweloperski oprogramowania Flutter (SDK). Flutter SDK zawiera mnóstwo gotowych, wielokrotnego użytku kodu Flutter oraz kilka narzędzi programowych do uruchamiania i testowania aplikacji Flutter. SDK zawiera oficjalne biblioteki kodów Flutter, biblioteki kodów Dart, dokumentację, a nawet kilka przykładowych aplikacji.

* Zintegrowane środowisko programistyczne. Możesz tworzyć aplikacje Flutter za pomocą narzędzi dla geeków, tylko z klawiatury, ale w końcu zmęczysz się pisaniem i przepisywaniem poleceń. Z drugiej strony zintegrowane środowisko programistyczne (IDE) przypomina trochę edytor tekstu: edytor tekstu pomaga w komponowaniu dokumenty (notatki, wiersze i inne dzieła literatury pięknej); natomiast IDE pomaga komponować instrukcje dla procesorów.

Do tworzenia aplikacji Flutter polecam Android Studio IDE. Nie daj się zwieść słowu Android w nazwie IDE. Korzystając z Android Studio, możesz tworzyć aplikacje na iPhone′a, aplikacje internetowe i inne rodzaje aplikacji.

* Niektóre przykładowe aplikacje Flutter, które pomogą Ci zacząć

* Urządzenie do testowania kodu Fluttera

Piszesz kod, a następnie uruchamiasz go, aby sprawdzić, czy działa poprawnie. Zwykle nie działa poprawnie, dopóki nie wprowadzisz pewnych zmian. Najczęściej nie działa poprawnie, dopóki nie wprowadzisz wielu zmian. Emulator to nie to samo co symulator. Emulator to oprogramowanie, które w dużej mierze zachowuje się jak sprzęt prawdziwego, fizycznego telefonu. Symulator to oprogramowanie, które uruchamia aplikacje telefonu, nie zachowując się zbytnio jak sprzęt telefonu.

Na szczęście po uruchomieniu tych aplikacji możesz zignorować tę subtelną różnicę. Wszystkie te narzędzia działają na komputerze deweloperskim — laptopie lub komputerze stacjonarnym, którego używasz do tworzenia aplikacji Flutter. Później, gdy opublikujesz aplikację, użytkownicy uruchamiają ją na swoich urządzeniach docelowych — urządzeniach fizycznych, takich jak iPhone′y, telefony z Androidem i (kiedyś wkrótce) inteligentnych tosterach. Oto dobra wiadomość: możesz bezpłatnie pobrać całe oprogramowanie potrzebne do uruchamiania przykładów z tej książki. Oprogramowanie jest podzielone na cztery pliki do pobrania:

* Gdy odwiedzasz https://flutter.dev/docs/getstarted/install, możesz kliknąć przycisk, aby zainstalować Flutter SDK.

* Przycisk na stronie http://developer.android.com/studio umożliwia pobranie Android Studio IDE. Wraz z tym pobieraniem pojawia się emulator Androida.

* Łącze do całego kodu : www.szkolazpieklarodem.pl/Flutter.zip.

*Symulator iPhone′a, a także cały kod potrzebny do generowania aplikacji na iPhone′a, są dostarczane wraz z instalacją Xcode na komputerze Mac. Xcode jest dostępny w Macintosh App Store. (Niestety nie można programować dla iPhone′a na komputerze z systemem Windows).

W świecie tworzenia aplikacji mobilnych wszystko zmienia się bardzo szybko. Instrukcje, które piszę we wtorek, mogą być nieaktualne do czwartku rano. Twórcy Fluttera nieustannie tworzą nowe funkcje i nowe narzędzia. Stare narzędzia przestają działać, a stare instrukcje nie mają już zastosowania. Jeśli zobaczysz na ekranie coś, co nie przypomina jednego z moich zrzutów ekranu, nie rozpaczaj. Może to być coś zupełnie nowego lub dotarłeś do zakątka oprogramowania, którego nie opisuję w tej książce. Tak czy inaczej, wyślij mi e-mail, tweet lub inną formę komunikacji. (Nie próbuj wysłać gołębia pocztowego. Mój kot dotrze do niej, zanim znajdę notatkę.)

**Co robić**

To stary, znajomy refren. Najpierw dostajesz oprogramowanie. Następnie uruchamiasz oprogramowanie. Pobieranie i instalowanie rzeczy

1. Pobierz plik zawierający wszystkie przykłady programów : www.szkolazpieklarodem.pl/Flutter.zip. Większość przeglądarek internetowych zapisuje pliki w katalogu Pobrane na dysku twardym komputera. Ale Twoja przeglądarka może być skonfigurowana nieco inaczej. Tak czy inaczej, zanotuj folder zawierający pobrany plik Flutter.zip.www.szkolazpieklarodem.pl/Flutter.zip.

2. Wyodrębnij zawartość pobranego pliku w dowolne miejsce na dysku twardym komputera.

3. Odwiedź https://flutter.dev/docs/getstarted/install i pobierz Flutter SDK. Wybierz wersję oprogramowania pasującą do Twojego systemu operacyjnego (Windows, Macintosh lub cokolwiek innego).

4. Wyodrębnij zawartość pobranego pliku w dowolne miejsce na dysku twardym komputera. Wspomniana zawartość jest w rzeczywistości katalogiem pełnym rzeczy. Nazwa katalogu to flutter. Umieść swój nowy katalog fluttera w miejscu, które nie jest chronione specjalnymi uprawnieniami. Na przykład, jeśli spróbujesz wyodrębnić katalog flutter w katalogu c:\program files, system Windows wyświetli okno dialogowe Kontrola konta użytkownika i poprosi o potwierdzenie. Nie umieszczaj katalogu flutter w takim miejscu. Mówisz „folder”. Mówię „katalog”. By nie do końca zacytować Gershwina, odłóżmy to wszystko na bok, ponieważ w tej książce używam tych dwóch słów zamiennie. Osobiście lubię umieszczać katalog flutter w moim katalogu domowym. Mój komputer ma katalog o nazwie Users, a wewnątrz tego katalogu Users znajduje się katalog o nazwie barryburd. Ten katalog

Barryburd jest moim katalogiem domowym. Ten katalog domowy zawiera mój katalog Documents, mój katalog Downloads i wiele innych rzeczy. Po wyodrębnieniu zawartości pobranego pliku mój katalog domowy Barryburd ma zupełnie nowy katalog flutter. Nie musisz wyodrębniać katalogu flutter bezpośrednio w swoim katalogu domowym, ale jest to najprostsza i najbardziej niezawodna rzecz, jaką mogę zrobić.

5. Zanotuj miejsce na dysku twardym, w którym znajduje się nowy katalog flutter. Na przykład, jeśli skopiowałeś zawartość pliku .zip do katalogu /Users/janeqreader, zanotuj katalog /Users/janeqreader/flutter. To jest twoja ścieżka Flutter SDK. Aby upewnić się, że poprawnie wyodrębniłeś zawartość pobranego pliku, zajrzyj do katalogu flutter w poszukiwaniu podkatalogu o nazwie bin. Mój katalog flutter ma inne podkatalogi, nazwane dev, przykłady i pakiety. Twój przebieg może się różnić w zależności od tego, kiedy pobierzesz Flutter SDK.

6. Odwiedź http://developer.android.com/studio i pobierz Android Studio IDE. Pobierany plik to plik .exe, plik .dmg lub może coś innego.

7. Zainstaluj oprogramowanie pobrane w kroku 6. Podczas instalacji w oknie dialogowym może być dostępna opcja instalacji wirtualnego urządzenia z systemem Android (AVD). Jeśli tak, zaakceptuj tę opcję.

Android Studio nie jest jedynym IDE, które ma funkcje do tworzenia aplikacji Flutter. Niektórzy programiści wolą Virtual Studio Code (znany pieszczotliwie jako VS Code), który jest dostępny dla systemów Windows, Macintosh i Linux. A jeśli lubisz to robić zgrubnie, możesz obejść się bez IDE i używać wiersza poleceń wraz z ulubionym edytorem tekstu — Emacsem, vi lub Notatnikiem. W tej książce skupiam się na Android Studio, ale można znaleźć wiele alternatyw.

**TYCH NIEPEWNYCH ROZSZERZEŃ NAZW PLIKÓW**

Nazwy plików wyświetlane w Eksploratorze plików lub w oknie Findera mogą wprowadzać w błąd. Możesz przeglądać katalog i zobaczyć nazwę androidstudioide lub flutter_windows. Prawdziwa nazwa pliku może brzmieć android-studio-ide.exe, flutter_windows.zip lub zwykły stary flutter_windows. Rozszerzenia nazw plików, takie jak .zip, .exe, .dmg, .app i .dart, to rozszerzenia nazw plików. Brzydka prawda jest taka, że domyślnie Windows i Mac ukrywają wiele rozszerzeń nazw plików. Ta okropna funkcja często myli ludzi. Jeśli nie chcesz być zdezorientowany, zmień ustawienia systemowe komputera. Oto jak to zrobić:

* W Windows 10: W głównym menu Eksploratora plików wybierz Widok. Na wyświetlonej wstążce umieść znacznik wyboru obok opcji Rozszerzenia nazw plików.

* W systemie macOS: W menu aplikacji Finder wybierz Preferencje. W wyświetlonym oknie dialogowym wybierz kartę Zaawansowane i poszukaj opcji Pokaż wszystkie rozszerzenia plików. Upewnij się, że to pole wyboru jest zaznaczone.

Odwiedzając dowolną witrynę pobierania oprogramowania, sprawdź wymagania dotyczące pobierania, instalowania i uruchamiania tego oprogramowania. Upewnij się, że masz wystarczającą ilość pamięci i wystarczająco aktualny system operacyjny.

Tylko dla użytkowników komputerów Mac

Jeśli masz komputer Mac i chcesz tworzyć aplikacje na iPhone′a, wykonaj następujące czynności:

1. Wybierz App Store z menu Apple.

2. W polu wyszukiwania sklepu wpisz Xcode, a następnie naciśnij Enter. Wyszukiwarka App Store znajduje dziesiątki aplikacji, ale tylko jedna ma prostą nazwę Xcode.

3. Kliknij przycisk Pobierz aplikacji Xcode. W rezultacie App Store instaluje Xcode na twoim komputerze.

4. Uruchom aplikację Xcode. Przy pierwszym uruchomieniu Xcode komputer Mac instaluje dodatkowe komponenty. Jeśli chcesz, aby Twoje aplikacje działały na urządzeniach Apple, potrzebujesz tych dodatkowych składników.

**SKOMPRESOWANE PLIKI ARCHIWALNE**

Kiedy odwiedzasz stronę www.allmycode.com/Flutter i pobierasz przykłady z tej książki, pobierasz plik o nazwie FlutterForDummies_Listings.zip. Plik zip to pojedynczy plik, który koduje kilka mniejszych plików. Plik Flutter.zip koduje pliki o nazwach takich jak App0301.dart, App0302.dart i App0401.dart. Plik App0301.dart zawiera kod z Listingu 3-1 — pierwszy wpis znajduje się w Części 3. Podobnie pliki App0302.dart i App0401.dart mają kod z Listingów 3-2 i 4-1. Plik Flutter.zip również koduje folder o nazwie asset. Ten folder zawiera kopie obrazów, które pojawiają się w aplikacjach. Przykładem skompresowanego pliku archiwum jest plik .zip. Inne przykłady skompresowanych archiwów obejmują pliki .tar.gz, pliki .rar i pliki .sparsebundle. Podczas rozpakowywania pliku wyodrębniane są oryginalne pliki i foldery przechowywane w większym pliku archiwum. (W przypadku pliku .zip innym słowem na określenie dekompresji jest rozpakowanie). Podczas pobierania pliku FlutterForDummies_Listings.zip przeglądarka internetowa może automatycznie rozpakować plik. Jeśli nie, możesz poprosić komputer o rozpakowanie pliku. Oto jak:

* Na komputerze z systemem Windows kliknij dwukrotnie ikonę pliku .zip. Gdy to zrobisz, Eksplorator plików systemu Windows wyświetli pliki i foldery w skompresowanym archiwum .zip. Przeciągnij wszystkie te pliki i foldery w inne miejsce na dysku twardym komputera (miejsce poza plikiem archiwum).

* Na komputerze Mac kliknij dwukrotnie ikonę pliku .zip. Gdy to zrobisz, Mac wyodrębni zawartość pliku archiwum i wyświetli wyodrębnioną zawartość w oknie Findera.

Konfigurowanie Android Studio

Android Studio nie jest automatycznie dostarczane z obsługą Flutter, co oznacza, że musisz dodać obsługę Flutter przy pierwszym uruchomieniu IDE. Oto, co robisz.

1. Uruchom aplikację Android Studio. Przy pierwszym uruchomieniu świeżej, nowej kopii Android Studio zobaczysz ekran powitalny.

2. Wybierz Konfiguruj ⇒ Wtyczki na ekranie powitalnym. Menu rozwijane Konfiguruj znajduje się w prawym dolnym rogu ekranu powitalnego.

3. Wyszukaj wtyczkę o nazwie Flutter. Zainstaluj tę wtyczkę. Jeśli Android Studio oferuje również opcję instalacji Darta, zaakceptuj tę opcję. Po zainstalowaniu wtyczki Android Studio może wymagać ponownego uruchomienia. Oczywiście należy go ponownie uruchomić. Gdy to zrobisz, ponownie zobaczysz ekran powitalny. Teraz ekran powitalny zawiera opcję Rozpocznij nowy projekt Flutter

**Uruchamianie pierwszej aplikacji**

Zainstalowałeś Android Studio, dodałeś wtyczkę Flutter do Android Studio, a następnie ponownie uruchomiłeś Android Studio. Teraz patrzysz na ekran powitalny Android Studio. Co zrobisz następnie?

1. Połącz się z Internetem. Podczas uruchamiania Twojej pierwszej aplikacji Android Studio pobiera dodatkowe oprogramowanie.

2. Wybierz opcję Rozpocznij nowy projekt Flutter. W telefonie aplikacja to aplikacja i to wszystko. Ale na twoim komputerze programistycznym cała twoja praca jest podzielona na projekty. W celach zawodowych nie masz absolutnej racji, jeśli myślisz, że jedna aplikacja jest równa jednemu projektowi. Ale w przypadku przykładów w tej książce model „jeden projekt równa się jednej aplikacji" działa dobrze. Jeśli nie widzisz opcji Rozpocznij nowy projekt Flutter, być może wtyczka Flutter nie została poprawnie zainstalowana. Zalecam podwójne sprawdzenie instrukcji w sekcji „Konfigurowanie Android Studio" we wcześniejszej części tego rozdziału. Jeśli to nie pomoże lub utkniesz w innym miejscu tego rozdziału, wyślij mi e-mail. Mój adres e-mail jest we wstępie do książki. Po wybraniu Rozpocznij nowy projekt Flutter zobaczysz trzy okna dialogowe, jedno po drugim. Pierwszy pyta, jaki rodzaj projektu Flutter chcesz utworzyć, drugi pyta o nazwę nowej aplikacji i inne szczegóły, a trzeci tworzy coś, co nazywa się pakietem.

3. W pierwszym oknie dialogowym wybierz Flutter Application, a następnie kliknij Next. Drugie okno dialogowe ma cztery pola: Nazwa projektu, Ścieżka Flutter SDK, Lokalizacja projektu i Opis.

4. Wybierz nazwę zawierającą tylko małe litery i, jeśli chcesz, znaki podkreślenia (_). Nazwy projektów Flutter nie mogą zawierać wielkich liter, spacji ani znaków interpunkcyjnych innych niż podkreślenie. Jeśli tworzysz wiele aplikacji, śledzenie ich wszystkich może doprowadzić Cię do szału. Dlatego warto zdecydować się na formułę nazywania aplikacji, a następnie trzymać się tej formuły tak ściśle, jak to tylko możliwe. Później, gdy zaczniesz sprzedawać swoje aplikacje, możesz porzucić formułę i używać sprytnych nazw, które przyciągną uwagę ludzi.

5. W przypadku ścieżki Flutter SDK podaj ścieżkę Flutter SDK. Skopiowałeś ścieżkę Flutter SDK, wykonując krok 5 we wcześniejszej sekcji „Pobieranie i instalowanie rzeczy", prawda? Jeśli zapomniałeś, wyszukaj na dysku twardym folder o nazwie flutter. Na pewno gdzieś tam będzie.

6. Nie zmieniaj opcji lokalizacji projektu, chyba że masz ku temu konkretny powód. Nie musisz określać nowego katalogu dla każdego ze swoich projektów. Android Studio zrobi to za Ciebie automatycznie z tą lokalizacją projektu jako punktem początkowym.

7. Jako opis wpisz coś głupiego i nietuzinkowego. Zrób to teraz, póki jeszcze możesz. Kiedy tworzysz aplikacje profesjonalnie, musisz być bardziej poważny. Po kliknięciu Dalej Android Studio wyświetla okno dialogowe Ustaw nazwę pakietu.

8. Jeśli Twoja firma posiada nazwę domeny lub posiadasz własną nazwę domeny, wpisz ją w polu Domena firmy. Jeśli nie, wpisz cokolwiek lub pozostaw domyślny tekst w spokoju. Pakiet to zbiór ściśle powiązanych fragmentów kodu, a każda aplikacja Flutter należy do własnego pakietu. W świecie Flutter zwyczajowo rozpoczyna się nazwę pakietu odwrotnością nazwy domeny. Na przykład nazwa domeny mojej firmy to allmycode.com. Kiedy więc tworzę aplikację Flutter, zwykle znajduje się ona w pakiecie o nazwie com.allmycode.somethingorother. Coś lub inna część jest unikalna dla każdej z moich aplikacji. Kiedy tworzysz swój pierwszy projekt, domyślnym tekstem pola Nazwa firmy jest prawdopodobnie example.com. Kilka lat temu Internet Corporation for Assigned Names and Numbers (ICANN) odłożyła tę nazwę na bok, aby każdy mógł jej używać. Zaraz pod tym okno dialogowe wyświetla nazwę pakietu example.com.cokolwiek nazwałeś swoją aplikację. Ta domyślna nazwa pakietu jest odpowiednia, gdy tworzysz swoje pierwsze aplikacje Flutter To okno dialogowe może zawierać pola wyboru z etykietami Wygeneruj przykładową zawartość, Dołącz obsługę Kotlin dla kodu Android i Dołącz obsługę Swift dla kodu iOS. Nie przejmuj się tymi polami wyboru. Sprawdź je lub nie sprawdzaj. W przypadku Twojej pierwszej aplikacji Flutter nie ma to znaczenia.

9. Kliknij Zakończ. Jak za dotknięciem czarodziejskiej różdżki pojawia się główne okno Android Studio. Główne okno zawiera wszystkie narzędzia potrzebne do tworzenia najlepszych aplikacji Flutter. Ma nawet przykładową aplikację startową, którą uruchamiasz w kilku następnych krokach.

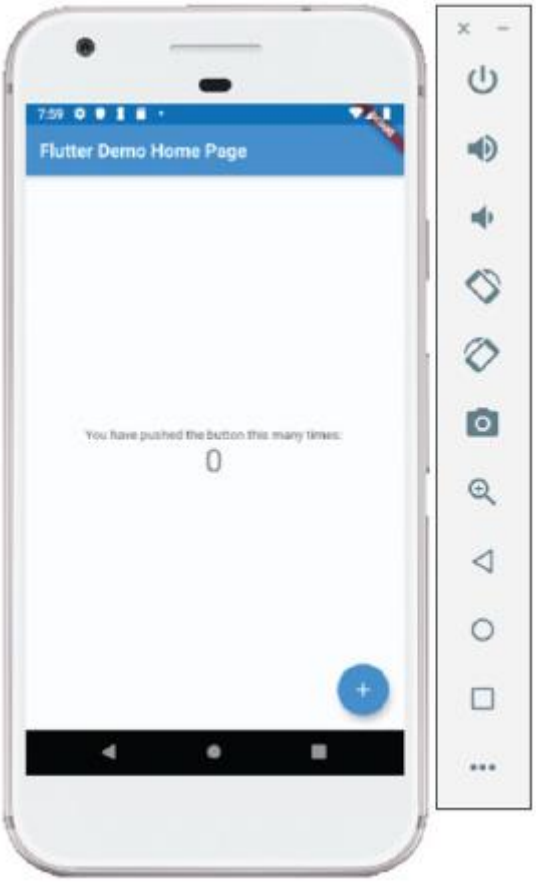
Główne okno Android Studio może początkowo wyglądać przytłaczająco. Aby pomóc ci stać się rozczarowanym (a może po prostu przeciętnym). Dwie ważne pozycje w górnej części głównego okna Android Studio:

* Selektor docelowy wyświetla tekst < brak urządzeń >.

* Ikona Uruchom to mała zielona strzałka skierowana w prawo.

To, co zrobisz dalej, zależy od komputera programistycznego i celów programistycznych. 10. Jeśli masz komputer Mac i chcesz uruchomić symulator iPhone′a, wybierz Open iOS Simulator z listy rozwijanej Target Selector. Jeśli nie masz komputera Mac lub chcesz uruchomić emulator Androida, wybierz Narzędzia ⇒ Menedżer AVD na pasku menu głównego Android Studio. W wyświetlonym oknie dialogowym poszukaj ikony zielonej strzałki po prawej stronie okna dialogowego. Kliknij ikonę zielonej strzałki. Jeśli menedżer AVD jest pusty — to znaczy, jeśli nie jest podobny do menedżera, który pokazuje urządzenie wirtualne oznaczone jako Pixel API 28 — musisz utworzyć urządzenie wirtualne Android. Wirtualne urządzenia Android nie zawsze uruchamiają się szybko. Na moim komputerze z 16 gigabajtami pamięci RAM czas uruchamiania może wynosić od dwóch do trzech minut. Na komputerze z zaledwie 4 gigabajtami pamięci RAM AVD może nigdy się nie uruchomić. Symulator iPhone′a firmy Apple jest nieco szybszy, ale nigdy nie wiadomo. Dwie dalsze sekcje tego rozdziału poświęciłem sztuczkom z emulatorem Androida i symulatorem iPhone′a — „O dodawaniu urządzeń wirtualnych” i „Podział urządzeń”. Gdy na ekranie pojawi się ekran główny urządzenia wirtualnego, możesz uruchomić przykładową aplikację Flutter.

11. Kliknij ikonę Uruchom na pasku narzędzi Android Studio. W rezultacie okno narzędzia Uruchom Android Studio pojawia się w dolnej części głównego okna. Podczas niecierpliwego oczekiwania na uruchomienie aplikacji pojawia się kilka komunikatów. Kiedy aplikacja zaczyna działać, urządzenie wirtualne (symulator lub emulator) ma ładny wyświetlacz.




Gratulacje! Twoja pierwsza aplikacja jest uruchomiona. Możesz wypróbować aplikację, klikając myszką pływający przycisk akcji aplikacji (okrągły element w prawym dolnym rogu ekranu urządzenia wirtualnego). Komunikat w środku informuje, ile razy kliknąłeś przycisk. Nie jest to najbardziej przydatna aplikacja na świecie, ale to dobry początek.

**Radzenie sobie z diabelskimi szczegółami**

Kilkadziesiąt lat temu kupiłem książkę o bazach danych ze stołu z dużymi rabatami w moim lokalnym supermarkecie. Kiedy dostałem książkę do domu, beznadziejnie utknąłem w rozdziale 1. Nie mogłem wymyślić, jak poprawnie uruchomić oprogramowanie. Walczyłem kilka godzin i poddałem się. Od tamtego dnia nie dotknąłem książki. Dlaczego piszę o tym przykrym doświadczeniu? Piszę o tym, aby zapewnić, że drapałem się po palcach, próbując uruchomić oprogramowanie. To problem, o który pyta większość czytelników, kiedy wysyłają do mnie e-maile. To naturalne, że utkniesz i potrzebujesz pomocy. We wcześniejszych sekcjach przedstawiam podstawowe kroki konfiguracji komputera i uruchomienia pierwszej aplikacji Flutter. Podstawowe kroki są fajne, ale nie działają dla wszystkich. Dlatego w tej sekcji zagłębiam się nieco. W świecie tworzenia aplikacji mobilnych wszystko zmienia się bardzo szybko. Instrukcje, które piszę we wtorek, mogą być nieaktualne do czwartku rano. Twórcy Fluttera nieustannie tworzą nowe funkcje i nowe narzędzia. Stare narzędzia przestają działać, a stare instrukcje nie mają już zastosowania. Jeśli zobaczysz na ekranie coś, co nie przypomina jednego z moich zrzutów ekranu, nie rozpaczaj.

**O instalacji Android Studio**

To, co zrobisz, aby zainstalować Android Studio, zależy od systemu operacyjnego:

* W systemie Windows: Pobrany plik to prawdopodobnie plik .exe. Kliknij dwukrotnie ikonę pliku .exe. Po dwukrotnym kliknięciu ikony pliku .exe kreator przeprowadzi Cię przez proces instalacji.

* Na komputerze Mac: Pobrany plik to prawdopodobnie plik .dmg. Kliknij dwukrotnie ikonę pliku .dmg. Po dwukrotnym kliknięciu ikony pliku .dmg zobaczysz ikonę Android Studio (znaną również jako ikona Android Studio.app). Przeciągnij ikonę Android Studio do folderu Aplikacje.

Jeśli chodzi o pliki .exe i pliki .dmg, nie daję żadnych gwarancji. Pobrany plik może być archiwum .zip lub innym egzotycznym plikiem archiwum. Jeśli nie wiesz, co robić, wyślij mi e-mail.

**O uruchomieniu Android Studio po raz pierwszy**

Czy nadszedł czas, aby uruchomić Android Studio? Ta sekcja zawiera kilka drobnych szczegółów.

* W systemie Windows: Kliknij przycisk Start i poszukaj wpisu Android Studio.

* Na komputerze Mac: naciśnij Command-spacja, aby wyświetlić Spotlight. W polu wyszukiwania Spotlight zacznij pisać Android Studio. Gdy komputer Mac wyświetli pełną nazwę Android Studio w polu wyszukiwania Spotlight, naciśnij Enter.

Jeśli Twój Mac skarży się, że Android Studio pochodzi od niezidentyfikowanego programisty, poszukaj ikony Android Studio w folderze Aplikacje. Ctrlkliknij ikonę Android Studio i wybierz Otwórz. Gdy pojawi się kolejne pole „niezidentyfikowany programista", kliknij znajdujący się przy nim przycisk Otwórz. Gdy uruchamiasz Android Studio po raz pierwszy, możesz zobaczyć okno dialogowe z propozycją zaimportowania ustawień z poprzedniej instalacji Android Studio. Możliwe, że nie masz wcześniejszej instalacji Android Studio, więc zdecydowanie, ale grzecznie odrzuć tę ofertę. Gdy opadnie kurz, Android Studio wyświetli ekran powitalny. Ekran powitalny zawiera opcje, takie jak Rozpocznij nowy projekt Android Studio, Otwórz istniejący projekt Android Studio, Konfiguruj i Uzyskaj pomoc. Cały czas widzisz ten ekran powitalny. Mówiąc nieformalnie, ekran powitalny mówi: „W tej chwili nie pracujesz nad żadnym konkretnym projektem (żadną konkretną aplikacją Flutter). Więc co chcesz robić dalej?

**O instalacji wtyczki Flutter w Android Studio**

Kiedy po raz pierwszy uruchamiasz Android Studio, zdecydowanie powinieneś zainstalować wtyczkę Android Studio do tworzenia aplikacji Flutter. Oto bliższe, bardziej szczegółowe spojrzenie na to, jak to zrobić:

1. Na ekranie powitalnym Android Studio wybierz Konfiguruj ⇒ Wtyczki. Na ekranie pojawi się okno dialogowe Plugins z trzema zakładkami. Karty są oznaczone jako Rynek, Zainstalowane i Aktualizacje .W niektórych wersjach Android Studio na ekranie powitalnym nie ma opcji Konfiguruj. W takim przypadku wybierz opcję Rozpocznij nowy projekt Android Studio na ekranie powitalnym. Zaakceptuj wszystkie ustawienia domyślne, dopóki nie zobaczysz głównego okna Android Studio. Następnie na pasku menu głównego wybierz Plik ⇒ Ustawienia ⇒ Wtyczki (w systemie Windows) lub Android Studio ⇒ Preferencje ⇒ Wtyczki (w systemie Mac).

2. Wybierz kartę Rynek. Gdy to zrobisz, Android Studio wyświetli obszerną listę dostępnych wtyczek. Będziesz chciał się zawęzić tą listę.

3. W polu wyszukiwania okna dialogowego wpisz słowo Flutter. Android Studio pokazuje kilka wtyczek ze słowem Flutter w ich tytułach. Każda wtyczka ma własny przycisk Instaluj. Poszukaj wtyczki o nazwie Flutter – nie Flutter Snippets, flutter_json_format ani nic podobnego.

4. Wybierz przycisk Instaluj dla wtyczki o nazwie Flutter. Po wyświetleniu okna dialogowego utworzonego przez prawników Google, Android Studio pyta, czy chcesz zainstalować wtyczkę Dart.

5. Wybierz Tak. Zdecydowanie chcesz zainstalować wtyczkę Dart. Po zakończeniu instalacji wtyczek Android Studio oferuje ponowne uruchomienie.

6. Uruchom ponownie Android Studio.

Po ponownym uruchomieniu ekran powitalny Android Studio ma nową opcję z etykietą Rozpocznij nowy projekt Flutter.

**O dodawaniu urządzeń wirtualnych**

Jeśli chodzi o instalowanie urządzeń wirtualnych, historie dotyczące iPhone′a i Androida są nieco inne.

* Na komputerze Apple, Windows lub Linux możesz pobrać Android Studio i pobrać dołączony do niego emulator Androida. Być może będziesz musiał trochę popracować, aby zainstalować urządzenie wirtualne Android (AVD), ale to nie jest wielka sprawa.

* Jeśli masz komputer Apple, możesz pobrać symulator iPhone′a, pobierając oprogramowanie Xcode firmy Apple.

Jeśli nie masz komputera Apple, możesz znaleźć symulatory innych firm, przeszukując Internet, ale pamiętaj, że tworzenie aplikacji na iPhone′a na czymkolwiek innym niż Mac jest trudne. W zależności od sposobu, w jaki to robisz, proces ten może być nawet nielegalny. Android rozróżnia emulator i urządzenie wirtualne Android (AVD). Oto miarka:

* Po zainstalowaniu Android Studio automatycznie otrzymujesz emulator telefonu z Androidem. Ten emulator może wypełnić lukę między sprzętem komputera programistycznego a makietą sprzętu telefonu. Ale który sprzęt telefonu kpi? Czy to Samsung Galaxy czy Sony Xperia? Jak duży jest ekran telefonu? Jaki aparat ma telefon?

* Wirtualne urządzenie Android to opis sprzętu telefonu. Emulator nie działa, chyba że utworzysz AVD do emulacji emulatora. Podczas instalowania Android Studio możesz zobaczyć opcję instalacji AVD lub nie. Jeśli tak, zaakceptuj tę opcję. Jeśli nie, to w porządku. Gdy uruchomisz Android Studio, będziesz mógł utworzyć kilka plików AVD.

Podczas instalacji Android Studio instalator może zaoferować opcję utworzenia AVD do użycia. Jeśli nie zaoferowano Ci tej opcji lub pominąłeś tę opcję, możesz utworzyć AVD za pomocą narzędzia AVD Manager. W rzeczywistości możesz utworzyć kilka dodatkowych AVD i używać kilku różnych AVD do uruchamiania i testowania aplikacji Flutter na emulatorze Androida. Aby otworzyć Menedżera AVD, przejdź do głównego paska menu Android Studio i wybierz Narzędzia ⇒ Menedżer AVD.

**IMPULSUJĄCE FIZYCZNE URZĄDZENIE Z ANDROIDEM**

Emulowane urządzenie z Androidem to tak naprawdę trzy części oprogramowania w jednym:

* Obraz systemu to kopia jednej wersji systemu operacyjnego Android. Na przykład określony obraz systemu może dotyczyć systemu Android Pie (poziom interfejsu API 28) działającego na procesorze Intel x86_64.

* Emulator wypełnia lukę między obrazem systemu a procesorem na komputerze deweloperskim. Możesz mieć obraz systemu dla procesora Atom:64, ale twój komputer programistyczny obsługuje procesor Core i5. Emulator tłumaczy instrukcje dla procesora Atom:64 na instrukcje, które może wykonać procesor Core i5.

* Wirtualne urządzenie Android (AVD) to oprogramowanie opisujące sprzęt urządzenia. AVD zawiera kilka ustawień, które przekazują emulatorowi wszystkie szczegóły dotyczące emulowanego urządzenia. Jaka jest rozdzielczość ekranu urządzenia? Czy urządzenie ma fizyczną klawiaturę? Czy ma aparat? Ile ma pamięci? Czy ma kartę SD? Wszystkie te wybory należą do podstawowego AVD.

Menu i okna dialogowe Android Studio ułatwiają pomylenie tych trzech elementów. Kiedy tworzysz nowy AVD, tworzysz nowy obraz systemu, który będzie pasował do tego AVD. Ale okna dialogowe Android Studio zacierają różnicę między AVD a obrazem systemu. Widzisz także słowo emulator, gdy poprawnym terminem jest AVD. Jeśli nie przeszkadzają Ci subtelne różnice między obrazami systemu, emulatorami i AVD, nie martw się o nie. Doświadczony programista Androida zazwyczaj ma kilka obrazów systemu i kilka AVD na komputerze programistycznym, ale tylko jeden emulator Androida. Niechętnie wymieniam instrukcje dotyczące korzystania z AVD Manager, ponieważ wygląd narzędzia AVD Manager ciągle się zmienia. Są szanse, że to, co widzisz na ekranie komputera, nie przypomina zrzutów ekranu z połowy 2019 roku na rysunkach od 2-12 do 2-15. Zamiast dawać wyraźne instrukcje, moją ogólną radą podczas tworzenia nowego AVD jest wybranie nowszych telefonów lub tabletów i poziomów API o wyższych numerach oraz zaakceptowanie ustawień domyślnych, ilekroć masz ochotę zagrać w eeny-meeny-minymoe. Po prostu klikaj Dalej, aż będziesz mógł kliknąć Zakończ. Jeśli nie podoba Ci się utworzony AVD, zawsze możesz ponownie otworzyć Menedżera AVD i wybrać inne opcje, aby utworzyć kolejny AVD. Kiedy osiągniesz poziom biegłości gdzie jesteś wybredny w kwestii charakterystyki swojego AVD, prawdopodobnie będziesz znał wiele opcji AVD Manager i będziesz mógł mądrze wybierać.

**O instalacji Fluttera**

Czasami, gdy źle się czuję, idę do lekarza. Jeśli masz problemy z uruchamianiem aplikacji i uważasz, że instalacja Fluttera jest chora, możesz zabrać Fluttera do lekarza. Oto jak:

1. W Android Studio rozpocznij nowy projekt Flutter lub otwórz istniejący projekt. Aby uzyskać pomoc, zapoznaj się z sekcją „Uruchamianie pierwszej aplikacji” v

2. Na pasku menu głównego Android Studio wybierz Narzędzia ⇒ Flutter ⇒ Flutter Doctor.

W rezultacie komputer informuje o stanie instalacji Fluttera. Raport Flutter Doctor nie zawsze jest pomocny. Niektóre ustalenia raportu mogą być fałszywymi alarmami. Inne mogą być trudne do interpretacji. Jeśli zobaczysz coś, co wygląda na użyteczną diagnozę, spróbuj. Wiele wskazówek lekarza dotyczy otwarcia okna terminala lub wiersza polecenia. Porady na ten temat znajdziesz w pasku bocznym „Twój przyjaciel, wiersz poleceń”. Treści dostarczane przez flutter doctor nie mają służyć jako substytut profesjonalnej porady medycznej. Zasięgnij porady lekarza Flutter w przypadku jakichkolwiek pytań dotyczących stanu zdrowia. Nigdy nie lekceważ profesjonalnej porady medycznej z powodu czegoś, co przeczytałeś w produkcji flutter doctor.

**Podział między urządzeniami**

Jeśli Twój komputer programistyczny ma wystarczającą moc, możesz jednocześnie uruchomić kilka urządzeń wirtualnych z systemem Android. Na komputerze Mac możesz uruchomić symulator iPhone′a, gdy działają Twoje urządzenia wirtualne z systemem Android. Ale korzystanie z urządzeń wirtualnych i fizycznych może być trudne. Ta sekcja zawiera kilka wskazówek.

**Uruchamianie aplikacji na urządzeniu z systemem Android**

Czytelnik z Minnesoty pisze:

Drogi Barry,

Postępowałem zgodnie ze wszystkimi twoimi instrukcjami. Wszystko idzie dobrze, dopóki nie spróbuję uruchomić aplikacji. Emulator Androida nie działa. Co powinienem zrobić?

Podpisano,

Wciąż mróz w Minneapolis

Cóż, panie Freezing, emulator dołączony do Android tudio pochłania mnóstwo zasobów na twoim komputerze programistycznym. Jeśli jesteś podobny do mnie i nie zawsze masz najnowszy, najpotężniejszy sprzęt, możesz mieć problemy z uruchamianiem aplikacji w emulatorze. Być może nie widzisz ekranu głównego Androida lub nie widzisz swojej aplikacji działającej mniej więcej pięć minut po uruchomieniu emulatora. Jeśli tak, oto kilka rzeczy, które możesz wypróbować:

* Spienić, spłukać, powtórzyć. Zamknij emulator i ponownie uruchom aplikację. Czasami drugi lub trzeci raz to urok. W rzadkich przypadkach moje pierwsze trzy próby kończą się niepowodzeniem, ale czwarta próba się powiedzie.

* Jeśli masz dostęp do komputera z większą ilością pamięci RAM, spróbuj uruchomić na nim swoją aplikację. Moc ma znaczenie.

* Jeśli nie masz dostępu do komputera z większą ilością pamięci RAM, zamknij wszystkie niepotrzebne programy na komputerze programistycznym i spróbuj ponownie uruchomić aplikację.

* Wypróbuj inny AVD. Sekcja „O dodawaniu urządzeń wirtualnych”, opisuje sposób dodawania nowego AVD do systemu. AVD z obrazem systemu x86 jest lepszy niż AVD z obrazem armeabi. (Na szczęście, gdy okno dialogowe pozwala wybrać między x86 a armeabi, nie musisz wiedzieć, co oznaczają x86 lub armeabi.)

* Zmagaj się z technologią wirtualizacji. Możesz nie chcieć zaczynać tej króliczej nory. Kiedy działa na procesorze Intel x86, emulator Androida próbuje użyć czegoś, co nazywa się Intel Virtualization Technology (VT) z Intel Hardware Accelerated Execution Manager (HAXM). Jeśli Twój komputer nie jest całkowicie wygodny w konfiguracji VTand-HAXM, prawdopodobnie będziesz mieć problemy z używaniem emulatora Androida. Nie rozpaczaj! Spróbuj zainstalować obraz systemu armeabi. Wreszcie, jeśli twój komputer może używać VT i HAXM oraz jeśli chcesz dostosować te elementy na swoim komputerze, śmiało. Tylko nie obwiniaj mnie, jeśli miesiąc później nagle przypomnisz sobie, że Twoim celem było poznanie Fluttera.

Poprzednia wypunktowana lista opisuje kilka rozwiązań problemów z emulatorem Android Studio. Niestety, żadna z kul na tej liście nie jest srebrną kulą. Jeśli wypróbowałeś te sztuczki i nadal masz problemy, możesz spróbować porzucić emulator dostarczany z Android Studio i uruchamiać aplikacje na „prawdziwym” urządzeniu.

**Testowanie aplikacji na urządzeniu fizycznym**

Możesz ominąć urządzenia wirtualne i przetestować swoje aplikacje na fizycznym telefonie, tablecie, a może nawet inteligentnym dzbanku do kawy. Aby to zrobić, musisz przygotować urządzenie fizyczne, przygotować komputer programistyczny, a następnie połączyć je ze sobą. To dość pracochłonne, ale po zrobieniu tego za pierwszym razem staje się znacznie łatwiejsze. W tej sekcji opisano kroki, które należy wykonać

Przygotowanie do testowania na fizycznym urządzeniu z Androidem

Aby przetestować swoją aplikację na urządzeniu z Androidem, wykonaj następujące czynności:

1. Na urządzeniu z Androidem włącz Opcje programisty. Na wielu urządzeniach z Androidem możesz to zrobić, wybierając Ustawienia ⇒ Informacje. Na liście Informacje stuknij pozycję Numer kompilacji siedem razy. (Tak, siedem razy.) Następnie naciśnij przycisk Wstecz, aby powrócić do listy ustawień. Na liście Ustawienia dotknij System ⇒ Opcje programisty. Niektóre osoby zgłosiły, że po siedmiokrotnym dotknięciu elementu numeru kompilacji pomaga trzykrotne obrócenie króliczej łapy nad głową. Jak dotąd nie udało mi się powtórzyć tych wyników.

2. Na liście Opcje programisty włącz debugowanie USB. Oto, co wyświetla jedno z moich urządzeń, gdy zadzieram z tym ustawieniem: Debugowanie USB jest przeznaczone do celów programistycznych. Używaj go do kopiowania danych między komputerem a urządzeniem, instalowania aplikacji na urządzeniu bez powiadomienia i odczytywania danych dziennika. Stewardzi Androida ostrzegają mnie, że opcja debugowania USB może narazić moje urządzenie na złośliwe oprogramowanie. Na moim urządzeniu cały czas włączam debugowanie USB. Ale jeśli martwisz się o bezpieczeństwo, wyłącz debugowanie USB, gdy nie używasz urządzenia do tworzenia aplikacji.

3. (Tylko dla użytkowników systemu Windows) Odwiedź

https://developer.android.com/studio/run/oemusb.html, aby pobrać sterownik USB urządzenia z systemem Android. Zainstaluj sterownik na komputerze deweloperskim z systemem Windows. Wykonując następny krok, miej oko na ekran swojego urządzenia z Androidem.

4. Za pomocą kabla USB podłącz urządzenie do komputera deweloperskiego. Nie wszystkie kable USB są sobie równe. Niektóre kable, zwane kablami do transmisji danych, mają druty i metal w miejscach, w których inne kable, zwane kablami do ładowania, zawierają tylko plastik. Spróbuj użyć dowolnego kabla USB dostarczonego z urządzeniem. Jeśli tak jak ja nie możesz znaleźć kabla dołączonego do urządzenia lub nie wiesz, który kabel był dołączony do urządzenia, wypróbuj więcej niż jeden kabel. Gdy znajdziesz kabel, który działa, oznacz go odpowiednim kablem. (Jeśli kabel zawsze działa, oznacz go jako „stabilny kabel”). Po podłączeniu kabla na ekranie urządzenia z systemem Android zostanie wyświetlone wyskakujące okno dialogowe. Wyskakujące okienko z pytaniem, czy chcesz zezwolić na debugowanie USB.

5. Tak, zezwól na debugowanie USB. Jeśli go nie szukasz, możesz przegapić wyskakujące okienko, aby umożliwić debugowanie USB. Pamiętaj, aby szukać tego wyskakującego okienka po podłączeniu urządzenia. Jeśli na pewno nie widzisz wyskakującego okienka, i tak możesz być w porządku. Ale jeśli pojawi się wiadomość i nie odpowiesz na nią, na pewno nie będzie dobrze.

**SPRAWDZANIE POŁĄCZENIA I PRZERWANIE POŁĄCZENIA**

Aby dowiedzieć się, czy Twój telefon z Androidem jest prawidłowo podłączony do komputera programistycznego, wykonaj następujące kroki:

1. Otwórz Terminal na komputerze Mac lub wiersz poleceń  w systemie Windows.

2. Użyj polecenia cd, aby przejść do katalogu narzędzi platformy Androida.

Jestem rootin'-tootin', dwupięściowym użytkownikiem komputera. Piszę na komputerze

cd

%HOMEDRIVE%%HOMEPATH%\AppData\Local\Android\Sdk\platformtools

Piszę na komputerze Mac

cd ~/Library/Android/sdk/platformtools/

3. Wpisz urządzenia adb. (Na komputerze Mac wpisz ./adb devices). Jeśli odpowiedź komputera zawiera bardzo długą liczbę szesnastkową (na przykład 2885046445FF097), liczba ta reprezentuje podłączone urządzenie. Na przykład, gdy podłączony jest jeden konkretny telefon, mój komputer odpowiada

emulator-5554 device

emulator-5556 device

2885046445FF097 device

Jeśli widzisz słowo nieautoryzowane obok długiej liczby szesnastkowej, prawdopodobnie nie odpowiedziałeś OK na pytanie „Zezwalać na debugowanie USB?" w kroku 5 wcześniejszej sekcji „Przygotowanie do testowania na fizycznym urządzeniu z Androidem". Jeśli odpowiedź komputera nie zawiera długiej liczby szesnastkowej, być może przegapiłeś łódź na jednym z pozostałych kroków w tej wcześniejszej sekcji. W końcu będziesz chciał odłączyć urządzenie od komputera deweloperskiego. Poszukaj odniesienia do urządzenia w Eksploratorze plików lub Finderze. * Jeśli nie widzisz odniesienia, prawdopodobnie możesz wyrwać kabel USB urządzenia z komputera.

* Jeśli widzisz odniesienie, spróbuj wysunąć urządzenie. Jeśli spróbujesz wysunąć urządzenie i zobaczysz przerażający komunikat Nie można bezpiecznie usunąć urządzenia, zacznij od wykonania kroków 1 i 2 na tym pasku bocznym. Następnie wykonaj jedną z następujących czynności:

Na komputerze Mac wpisz

./adb kill-server

a następnie naciśnij Enter.

W systemie Windows wpisz

adb kill-server

a następnie naciśnij Enter.

Następnie zobaczysz przyjazny komunikat Bezpieczne usuwanie sprzętu.

Przygotowanie do testowania na iPhonie

Aby przetestować swoją aplikację na iPhonie (lub nawet iPadzie), musisz używać komputera Apple. Jeśli masz komputer Mac, wykonaj następujące kroki:

1. Odwiedź https://brew.sh i postępuj zgodnie z instrukcjami, aby zainstalować Homebrew na swoim komputerze. Homebrew to menedżer pakietów oprogramowania innej firmy dla systemów macOS i Linux. Możesz go użyć do zainstalowania wszelkiego rodzaju oprogramowania, nie tylko narzędzi programistycznych iPhone′a.

2. Otwórz aplikację Terminal na komputerze Mac.

3. W oknie aplikacji Terminal wpisz kolejno następujące polecenia:

brew update

brew install --HEAD usbmuxd

brew link usbmuxd

brew install --HEAD libimobiledevice

brew install ideviceinstaller ios-deploy

cocoapods

pod setup

Czy to nie było zabawne? Uzyskanie odpowiedzi zajmuje dużo czasu, a po drodze prawdopodobnie zobaczysz przerażające komunikaty ostrzegawcze.

Instrukcje zawarte w tym kroku są aktualne na dzień 23 sierpnia 2019 r., godz. 10:05 czasu wschodnioamerykańskiego. Po tym momencie nie składam żadnych obietnic. Jeśli utkniesz, zajrzyj do sieci lub wyślij mi e-mail.

4. Odwiedź stronę developer.apple.com i zarejestruj się, aby uzyskać bezpłatne członkostwo w programie Apple dla programistów. Po wykonaniu tych trzech kroków komputer programistyczny jest gotowy do pracy. Wykonaj te czynności, gdy chcesz przetestować nową aplikację Flutter na fizycznym iPhonie:

1. Podłącz fizyczny telefon do komputera programistycznego za pomocą kabla USB do transmisji danych. Nie wszystkie kable są takie same. Apple umieszcza zastrzeżony chip w każdym z kabli iPhone′a. Jeśli kupisz kabel od zewnętrznego dostawcy, możesz nie mieć możliwości przeniesienia aplikacji na telefon za jego pomocą.

2. W Android Studio otwórz swój nowy projekt Flutter.

3. Poszukaj okna narzędzia Projekt — panelu wyświetlającego drzewo plików i folderów. Okno narzędzia Projekt znajduje się po lewej stronie głównego okna Android Studio.

4. Rozwiń jedną z najwyższych gałęzi drzewa, aby znaleźć gałąź o nazwie iOS.

5. Kliknij prawym przyciskiem myszy gałąź iOS. W wynikowym menu kontekstowym wybierz Flutter ⇒ Otwórz moduł iOS w Xcode. W rezultacie uruchamia się Xcode. Po lewej stronie okna Xcode znajduje się drzewo plików i folderów.

6. W drzewie plików i folderów wybierz Runner.

7. Wybierz kartę Podpisywanie i możliwości w górnej części okna Xcode. Karta Podpisywanie i uprawnienia zawiera listę rozwijaną Zespół.

8. Z listy rozwijanej Zespół wybierz opcję Dodaj konto. W rezultacie pojawi się okno dialogowe Konta. Dzięki Apple ID automatycznie należysz do zespołu programistów — Twojego osobistego zespołu, którego jedynym członkiem jesteś Ty.

9. Zrób wszystko, co musisz zrobić w oknie dialogowym Konta, a następnie zamknij okno dialogowe. W rezultacie wrócisz do zakładki Podpisywanie i możliwości.

10. Z listy rozwijanej Zespół wybierz własny zespół.

11. Zamknij Xcode. Możesz iść.

Testowanie na dowolnym urządzeniu fizycznym (Android lub iPhone)

Gdy będziesz gotowy do przetestowania aplikacji na urządzeniu fizycznym i podłączysz urządzenie do komputera programistycznego, spójrz na listę rozwijaną Target Selector na pasku narzędzi Android

Studio. Gdy komputer deweloperski prawidłowo komunikuje się z urządzeniem fizycznym, nazwa urządzenia pojawia się jako jeden z elementów tej listy rozwijanej. Wybierz ten element, a następnie kliknij ikonę Uruchom.

**Korzystanie z Android Studio**

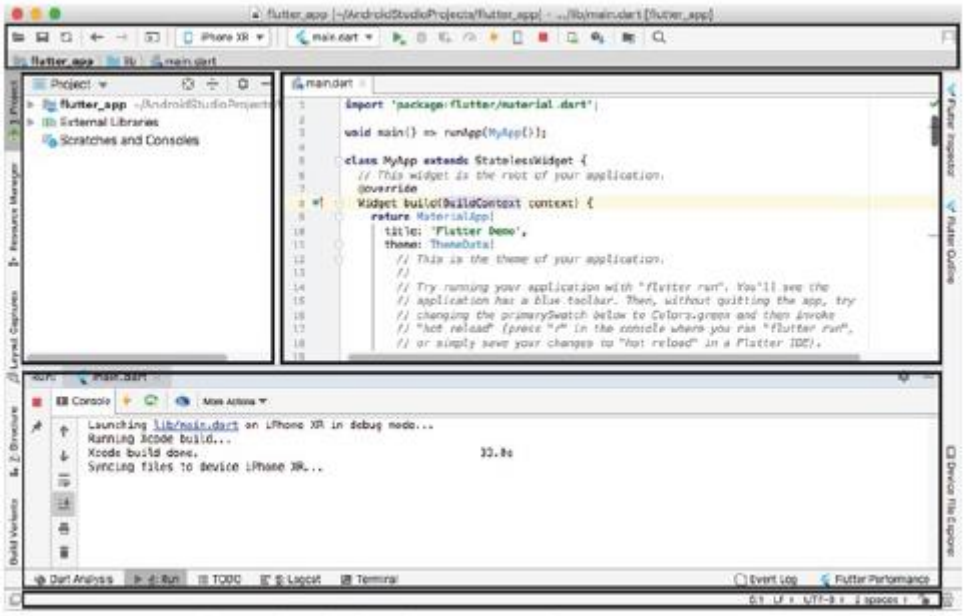
Android Studio to dostosowana wersja IntelliJ IDEA — IDE ogólnego przeznaczenia z narzędziami do programowania w Javie, C/C++, PHP, modelowania, zarządzania projektami, testowania, debugowania i wielu innych. W tej sekcji znajdziesz przegląd głównego okna Android Studio. Koncentruję się na najbardziej przydatnych funkcjach, które pomagają tworzyć aplikacje Flutter, ale pamiętaj, że Android Studio ma setki funkcji i wiele sposobów uzyskiwania dostępu do każdej z nich.

**Uruchamianie**

Każda aplikacja Flutter należy do projektu. Możesz mieć dziesiątki projektów na dysku twardym komputera. Kiedy uruchamiasz Android Studio, każdy z twoich projektów jest otwarty lub zamknięty. Otwarty projekt pojawia się w oknie (własnym oknie) na ekranie komputera. Zamknięty projekt nie pojawia się w oknie. Kilka Twoich projektów może być otwartych w tym samym czasie. Możesz przełączać się między projektami, przechodząc od okna do okna. Często nazywam otwarte okno projektu głównym oknem Android Studio. Może to być nieco mylące, ponieważ przy kilku projektach jednocześnie otwartych jest kilka głównych okien. W pewnym sensie żadne z tych okien nie jest bardziej „główne” niż inne. Kiedy piszę główne okno, mam na myśli okno, nad którym aktualnie pracujesz we Flutterze. Jeśli Android Studio jest uruchomione i żaden projekt nie jest otwarty, Android Studio wyświetla ekran powitalny. Na ekranie powitalnym mogą być wyświetlane ostatnio zamknięte projekty. Jeśli tak, możesz otworzyć projekt, klikając jego nazwę na ekranie powitalnym. W przypadku istniejącej aplikacji, której nie ma na liście ostatnich projektów, możesz kliknąć opcję Otwórz istniejący projekt Android Studio na ekranie powitalnym. Jeśli masz jakieś otwarte projekty, Android Studio nie wyświetla ekranu powitalnego. W takim przypadku możesz otworzyć inny projekt, wybierając Plik ⇒ Otwórz lub Plik ⇒ Otwórz ostatnie w oknie otwartego projektu. Aby zamknąć projekt, możesz wybrać Plik ⇒ Zamknij projekt lub możesz zrobić to, co zwykle robisz, aby zamknąć jedno z okien na komputerze. (Na komputerze PC kliknij X w prawym górnym rogu okna. Na komputerze Mac kliknij mały czerwony przycisk w lewym górnym rogu okna). Android Studio zapamiętuje, które projekty były otwarte od jednego uruchomienia do drugiego. Jeśli jakieś projekty są otwarte po zamknięciu Android Studio, zostaną one ponownie otwarte (z widocznymi głównymi oknami) przy następnym uruchomieniu Android Studio. Możesz zmienić to zachowanie (tak, aby przy każdym uruchomieniu pojawiał się tylko ekran powitalny Studio Androida). W Android Studio na komputerze z systemem Windows zacznij od wybrania Plik ⇒ Ustawienia ⇒ Wygląd i zachowanie ⇒ Ustawienia systemowe. W Android Studio na komputerze Mac wybierz Android Studio ⇒ Preferencje ⇒ Wygląd i zachowanie ⇒ Ustawienia systemowe. W obu przypadkach usuń zaznaczenie pola wyboru Ponownie otwórz ostatni projekt przy uruchamianiu.

**Główne okno**

Główne okno Android Studio jest podzielone na kilka obszarów. Niektóre z tych obszarów mogą pojawiać się i znikać na twoje polecenie. Następnie następuje opis obszarów z rysunku 2.18, przechodząc od góry głównego okna do dołu.




Obszary widoczne na ekranie komputera mogą różnić się od obszarów na rysunku. Zazwyczaj jest to w porządku. Możesz sprawić, by obszary pojawiały się i znikały, wybierając określone opcje menu, w tym opcję Widok na głównym pasku menu Android Studio. Możesz także kliknąć małe przyciski narzędzi na krawędziach głównego okna.

**Górna część głównego okna**

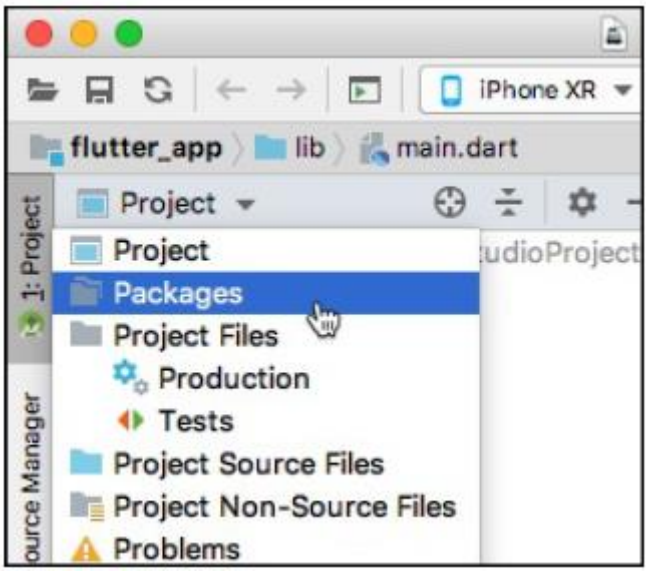
Górny obszar zawiera pasek narzędzi i pasek nawigacji.

* Pasek narzędzi zawiera przyciski czynności, takie jak Otwórz i Zapisz wszystko. Zawiera również selektor celu i ikonę biegu. Selektor docelowy to lista rozwijana, której domyślną opcją jest <brak urządzeń>. Na rysunku 2.18 selektor celu wyświetla nazwę iPhone XR. Ikona Uruchom to coś, co wygląda jak zielony przycisk Odtwórz. Możesz przeczytać więcej o tych przedmiotach wcześniej.

* Pasek nawigacyjny wyświetla ścieżkę do jednego z plików w twoim projekcie Flutter. Projekt Flutter zawiera wiele plików iw dowolnym momencie pracujesz nad jednym z nich. Pasek nawigacyjny wskazuje ten plik.

**Okno narzędzia Projekt**

Poniżej menu głównego i pasków narzędzi widoczne są dwa różne obszary. Obszar po lewej stronie zawiera okno narzędzia Projekt, które służy do nawigowania między plikami w aplikacji na Androida. W dowolnym momencie okno narzędzia Projekt wyświetla jeden z kilku możliwych widoków. Na przykład na rysunku 2.18 okno narzędzia Projekt wyświetla swój widok Projekt. Na rysunku 2.19 klikam listę rozwijaną i wybieram widok Packages (zamiast widoku Project).




W widoku Pakiety jest wyświetlanych wiele takich samych plików jak w widoku Projekt, ale w widoku Pakiety pliki są pogrupowane inaczej. W przypadku większości instrukcji zawartych w tej książce zakładam, że okno narzędzia Projekt jest w widoku domyślnym; mianowicie Widok projektu. Jeśli Android Studio nie wyświetla okna narzędzia Projekt, poszukaj przycisku narzędzia Projekt — małego przycisku wyświetlającego słowo Projekt na lewej krawędzi głównego okna. Kliknij ten przycisk narzędzia Projekt. (Ale poczekaj! Co jeśli nie możesz znaleźć małego przycisku Projekt? W takim przypadku przejdź do głównego menu Android Studio i wybierz Okno ⇒ Przywróć domyślny układ.)

**Obszar edytora**

Obszar po prawej stronie okna narzędzia Projekt to obszar Edytora. Kiedy edytujesz plik programu Dart, edytor wyświetla tekst pliku. (Patrz rysunek 2.18.) Tekst można wpisywać, wycinać, kopiować i wklejać tak samo, jak w innych edytorach tekstu. Obszar edytora może mieć kilka zakładek. Każda karta zawiera plik otwarty do edycji. Aby otworzyć plik do edycji, kliknij dwukrotnie gałąź pliku w oknie narzędzia Projekt. Aby zamknąć plik, kliknij mały x obok nazwy pliku na karcie Edytor.

**Dolny obszar**

Poniżej okna narzędzia Projekt i obszaru Edytora znajduje się kolejny obszar zawierający kilka okien narzędzi. Gdy nie używasz żadnego z tych okien narzędzi, możesz nie widzieć tego dolnego obszaru. W dolnym obszarze najczęściej używanym oknem narzędzia jest okno narzędzia Uruchom. (Patrz dolna część rysunku 2-18.) Okno narzędzia Uruchom pojawia się automatycznie po kliknięciu ikony Uruchom. To okno narzędzia wyświetla informacje o uruchomieniu aplikacji Flutter. Jeśli Twoja aplikacja nie działa poprawnie, okno narzędzia Uruchom może zawierać przydatne informacje diagnostyczne. Możesz wymusić wyświetlanie innych okien narzędzi w dolnym obszarze, klikając przyciski narzędzi u dołu okna Android Studio. Na przykład po kliknięciu przycisku narzędzia Terminal Android Studio wyświetla polecenie systemu Windows

**Monituj, aplikacja Terminal Mac lub inny określony tekstowy ekran poleceń.**

Określony przycisk narzędzia może się nie pojawić, jeśli nic nie można z nim zrobić. Na przykład przycisk narzędzia Uruchom może się nie pojawić, dopóki nie naciśniesz ikony Uruchom. Nie martw się o to. Przycisk narzędzia pojawia się zawsze, gdy jest potrzebny.

**Pasek stanu**

Pasek stanu znajduje się na dole okna Android Studio. Pasek stanu informuje, co się teraz dzieje. Na przykład, jeśli kursor znajduje się na 37. znaku 11. wiersza w edytorze, gdzieś w wierszu stanu zobaczysz godzinę 11:37. Gdy każesz Android Studio uruchomić aplikację, pasek stanu zawiera najnowszy komunikat okna narzędzia Uruchom.

Oprócz obszarów, o których wspominam w tej sekcji, w razie potrzeby mogą pojawić się inne obszary. Obszar można zamknąć, klikając jego ikonę Ukryj.

**Uruchamianie przykładowych programów**

Możesz uruchomić dowolny z tych programów jako część aplikacji Android Studio Flutter. Ta sekcja zawiera wszystkie szczegóły.

1. Uruchom Android Studio. Do uruchomienia pierwszej aplikacji potrzebne jest połączenie z Internetem. To, co zrobisz dalej, zależy od tego, co zobaczysz po uruchomieniu Android Studio.

2. Jeśli zobaczysz ekran powitalny Android Studio, wybierz Rozpocznij nowy projekt Flutter. Jeśli zobaczysz inne okno Android Studio z opcją Plik na pasku menu głównego, wybierz Plik ⇒ Nowy ⇒ Nowy projekt Flutter na pasku menu głównego. Tak czy inaczej, pojawi się pierwsze okno dialogowe do tworzenia nowego projektu Flutter.

3. Utwórz nowy projekt Flutter, wykonując kroki od 3 do 9 z wcześniejszej części „Uruchamianie pierwszej aplikacji".

4. W oknie narzędzi projektu Android Studio poszukaj folderu o nazwie lib. Jeśli potrzebujesz pomocy w znalezieniu tego okna narzędzia, zapoznaj się z sekcją „Okno narzędzia projektu" we wcześniejszej części tego rozdziału. Okno narzędzia Projekt zawiera drzewo folderów i plików. Rozwiń jedną z najwyższych gałęzi drzewa, aby znaleźć folder lib. Ten folder lib zawiera kod Dart twojego projektu.

5. Kliknij prawym przyciskiem myszy gałąź main.dart drzewa, a następnie wybierz Usuń. Jeśli Android Studio wyświetli monit o potwierdzenie, kliknij OK. W ten czy inny sposób daj main.dart dawną frajdę.

6. Upewnij się, że rozpakowałeś plik FlutterForDummies_Listings.zip. Jeśli nie masz pewności, gdzie znaleźć plik FlutterForDummies_Listings.zip, poszukaj najpierw w folderze o nazwie Pobrane. Większość przeglądarek internetowych domyślnie umieszcza rzeczy w Pobranych plikach. Safari na komputerze Mac zwykle automatycznie rozpakowuje archiwa .zip, a przeglądarki Windows (Internet Explorer, Firefox, Chrome i inne) nie rozpakowują automatycznie archiwów .zip.

7. W Eksploratorze plików lub Finderze przejdź do nieskompresowanego folderu FlutterForDummies_Listings. W tym folderze poszukaj przykładu, który chcesz uruchomić. Jeśli zajrzysz do nieskompresowanego pliku do pobrania, zauważysz pliki o nazwach App0301.dart, App0302.dart i tak dalej. Z kilkoma wyjątkami numery w tych nazwach plików to numery rozdziałów, po których następują numery list. Na przykład w nazwie App0602.dart cyfra 06 oznacza rozdział 6, a cyfra 02 drugą listę kodów w tym rozdziale. W tym eksperymencie sugeruję odszukanie pliku App0201.dart.

8. Kliknij prawym przyciskiem myszy wybrany plik App####.dart. Następnie w wynikowym menu kontekstowym wybierz Kopiuj.

9. Kliknij prawym przyciskiem myszy pusty folder lib nowego projektu. W wynikowym menu kontekstowym wybierz Wklej. Jeśli Android Studio wyświetli okno dialogowe z propozycją wklejenia do określonego katalogu, upewnij się, że pełna nazwa katalogu kończy się na lib. Następnie naciśnij OK. Teraz możesz uruchomić jeden z przykładów. Idź po to!

Czasami możesz mieć więcej niż jeden plik w folderze lib swojego projektu i więcej niż jedną aplikację w swoim projekcie. Jeśli to zrobisz, naciśnięcie ikony Uruchom może nie uruchomić aplikacji, która pojawia się w obszarze edytora Android Studio. Aby uruchomić aplikację wyświetlaną w obszarze edytora, poszukaj karty tej aplikacji u góry obszaru edytora. Po kliknięciu tej karty prawym przyciskiem myszy zobaczysz opcję, taką jak Uruchom „App0201.dart". Wybierz tę opcję i obserwuj przebieg programu.

**Ciesz się powtórkami**

Gdy po raz drugi uruchomisz konkretny przykład , nie musisz wykonywać wszystkich kroków z poprzedniej sekcji. Łatwo jest uruchamiać przykład w kółko. Możesz zrobić zmiany w kodzie, a następnie ponownie kliknij ikonę Uruchom. Jest to wszystko co musisz zrobić. Jeśli zamknąłeś projekt i chcesz go uruchomić ponownie, po prostu ponownie otwórz projekt w Android Studio i kliknij ikonę Uruchom.

**Jeśli jesteś wybredny…**

Po wykonaniu czynności opisanych w poprzedniej sekcji możesz zobaczyć niektóre znaczniki błędów (faliste, czerwone podkreślenia) w projekcie okno narzędzia. Przykładowy projekt Flutter w Android Studio opisuje coś o nazwie MyApp, ale kod, który ty skopiowałeś do folderu lib nie wspomina o MyApp. Możesz uruchamiać ten projekt w kółko bez naprawiania pliku , falistym, czerwonym podkreśleniem. Ale jeśli chcesz je naprawić, po prostu wykonaj następujące kroki:

1. W oknie narzędzia Projekt rozwiń gałąź oznakowaną test. Wewnątrz tej gałęzi znajdziesz plik o nazwie widget_test.dart.

2. Usuń plik widget_test.dart.

Faliste, czerwone podkreślenia zniknęły. Problem rozwiązany!

Aplikacje nasze to aplikacje do ćwiczeń. Nikt nie uruchamia tych aplikacji, aby wykonać prawdziwą pracę. Kiedy tworzysz prawdziwą aplikację, nigdy nie wolno ignorować kodu w folderze testowym. Testowanie jest istotną częścią procesu tworzenia oprogramowania. Dokładne testowanie sprawia, że programy działają niezawodnie. Innym sposobem na pozbycie się falistych, czerwonych podkreśleń jest wskoczenie do wehikułu czasu i ponowne wykonanie instrukcji z sekcji „Uruchamianie przykładowych programów". Jeśli zignorujesz krok 5 i nie usuniesz main.dart, nie dostaniesz tych czerwonych podkreśleń. Ale być może będziesz musiał poradzić sobie z dwoma innymi problemami. Zachowanie ikony Uruchom może być nieco mylące. Dodatkowo możesz stworzyć wyrwę w kontinuum czasoprzestrzennym i stać się swoim własnym dziadkiem.

**Czy te kroki konfiguracji były zabawne, czy co?**

Zawsze boję się konfiguracji oprogramowania, która nie jest całkowicie trywialna. Każdy komputer jest inny, a instrukcje instalacji aplikacji nie mogą obejmować każdego możliwego scenariusza. Ale po uruchomieniu oprogramowania czuję się podekscytowany. W końcu mogę zacząć korzystać z oprogramowania i czerpać korzyści z pracy nad etapami konfiguracji. Jeśli całe to zamieszanie w tym rozdziale doprowadziło Cię do punktu, w którym jesteś gotowy do nauki Fluttera, świetnie!

**"Helo" od Fluttera**

Słowo Hello  jest względną nowością w języku angielskim. Jego pierwsze znane użycie w druku miało miejsce w Norwich, Connecticut, Courier w 1826 roku. Alexander Graham Bell, wynalazca telefonu, uważał, że rozmowy telefoniczne powinny zaczynać się od terminu Ahoy! ale najwyraźniej Thomas Edison wolał Hello, a wczesne książki telefoniczne zalecały wybór Edisona. Według legendy pierwszy program komputerowy, który drukował tylko "Witaj, świecie!" został napisany przez Briana Kernighana jako część dokumentacji języka programowania BCPL. Pierwszy publiczny występ takiego programu miał miejsce w książce Kernighana i Ritchiego z 1972 roku, The C Programming Language. W dzisiejszych czasach termin program Hello world lub po prostu program Hello odnosi się do każdego prostego kodu dla czyjegoś pierwszego kontaktu z nowym językiem lub nowym frameworkiem. Ten rozdział zawiera prosty program Flutter "Hello world" i kilka ozdób. Możesz uruchomić kod, przeanalizować go, zmienić i dobrze się z nim bawić.

**Pierwsze rzeczy na pierwszym miejscu**

Listing 1 zawiera Twoją pierwszą aplikację Flutter.

```
import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Text("Hello world!"),

),

);

}

}
```

Jeśli wolisz samodzielnie wpisać kod, wykonaj następujące czynności:

1. Utwórz nowy projekt Flutter. Jak zwykle Android Studio tworzy dla Ciebie plik pełen kodu Dart. Nazwa pliku to main.dart.

2. Upewnij się, że kod main.dart pojawił się w edytorze Android Studio. Jeśli nie, rozwiń drzewo w oknie narzędzia Projekt po lewej stronie głównego okna Android Studio. Poszukaj gałęzi lib, aw gałęzi lib gałęzi main.dart. Kliknij dwukrotnie tę gałąź main.dart.

3. W edytorze Android Studio usuń cały kod main.dart.

4. W edytorze Android Studio wpisz kod, który widzisz na Listingu 1.

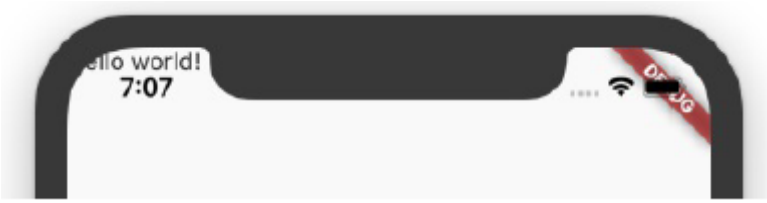
Jeśli zmienisz małą literę w słowie na wielką literę, możesz zmienić znaczenie słowa. ZMIANA przypadku może sprawić, że całe słowo przestanie mieć znaczenie i stanie się bez znaczenia. W pierwszym wierszu listingu 1 nie można zastąpić importu importem. JEŚLI TO ZROBISZ, CAŁY PROGRAM PRZESTAJE DZIAŁAĆ. Spróbuj i przekonaj się sam!

Rysunek  przedstawia gotowy produkt

```
main.dart
import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Material(
        child: Text("Hello world!"),
      ),
    );
  }
}
```

5. Uruchom nową aplikację.

Rysunek 3-2 pokazuje, co widzisz po uruchomieniu aplikacji Flutter z Listingu 1. Aplikacja wygląda dość źle, ale przynajmniej możesz zobaczyć mały Hello world! w lewym górnym rogu ekranu. Kosmetycznymi aspektami aplikacji zajmę się w dalszej części .




W edytorze Android Studio możesz zobaczyć czerwone znaczniki. Jeśli tak, najedź kursorem na znacznik i przeczytaj wyświetlone wyjaśnienie. Niektóre wyjaśnienia są łatwe do zrozumienia; inni nie. Im więcej masz praktyki w interpretowaniu tych komunikatów, tym bardziej jesteś wprawny w rozwiązywaniu problemów. Inną rzeczą, którą możesz spróbować, jest wybranie zakładki Analiza rzutów u dołu głównego okna Android Studio. Ta karta zawiera listę wielu miejsc w twoim projekcie, które zawierają wątpliwy kod. W przypadku dowolnego elementu na liście czerwona ikona oznacza błąd - coś, co należy naprawić. (Jeśli tego nie naprawisz, plik app nie można uruchomić.) Każda inna ikona koloru oznacza ostrzeżenie - coś, co nie zapobiegnie uruchomieniu kodu, ale może być warte rozważenia. W następnych kilku sekcjach rozbiorę kod z Listingu 1. Analizuję kod z wielu punktów widzenia. Wyjaśniam, co robi kod, dlaczego robi to, co robi i co może robić inaczej.

O czym to jest?

Kiedy spojrzysz na Listing 1, możesz zobaczyć słowa, znaki interpunkcyjne i wcięcia, ale to nie jest to, co widzą doświadczeni programiści Fluttera. Widzą ogólny zarys. Widzą wielkie idee w pełnych zdaniach. Rysunek 3-3 pokazuje, jak wygląda Listing 1 dla doświadczonego programisty.

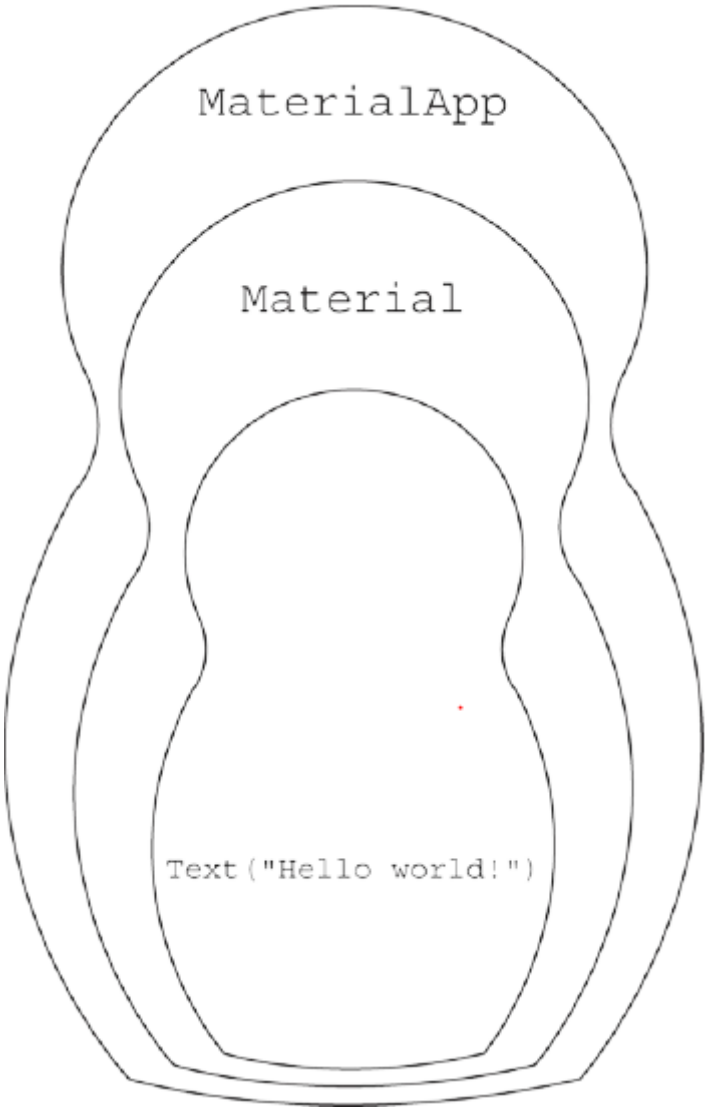
```
import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Material(
        child: Text("Hello world!"),
      ),
    );
  }
}
```

This Flutter app is a Material Design app. *Material Design* is a set of specifications describing the way an app looks.

An app's `home` is the app's starting screen – the initial page of the app. This app's `home` is something that looks like a piece of `Material`.

The piece of `Material` has one thing on it – one `child`. That `Child` is a bunch of `Text`. It's the words `Hello world!`

Program Flutter jest jak zestaw rosyjskich lalek matrioszek. Jest to rzecz w rzeczy w innej rzeczy i tak dalej, aż dojdziesz do punktu końcowego.




Listing 1 zawiera tekst wewnątrz elementu Material, który z kolei znajduje się wewnątrz aplikacji MaterialApp. Słowa Text, Material i MaterialApp rozpoczynają polecenia konstruowania rzeczy. W terminologii języka Dart słowa Text, Material i MaterialApp to nazwy wywołań konstruktora. Oto wewnętrzna historia:

Kod

Tekst("Witaj świecie!")

jest wywołaniem konstruktora. Kiedy Flutter wykonuje ten kod, konstruuje obiekt Text. Ten obiekt Text zawiera słowa Witaj, świecie!

Kod

Material(child: Text("Hello world!"),

)

jest kolejnym wywołaniem konstruktora. Kiedy Flutter wykonuje ten kod, konstruuje obiekt Material. Ten obiekt Material zawiera wspomniany wcześniej obiekt Text

Obiekt Material ma pewne cechy charakterystyczne dla materiału fizycznego, takiego jak kawałek tkaniny. Ma określony kształt. Może być wyniesiony z powierzchni pod nim. Możesz go przesunąć lub uszczypnąć. To prawda, że tło na rysunku 3.2 nie przypomina kawałka tkaniny. Ale imitowanie faktury materiału już nie jest celem Material Design. Celem Material Design jest stworzenie języka do opisywania stanu komponentów na ekranie użytkownika i opisywania, w jaki sposób te komponenty są ze sobą powiązane.

Kod:

MaterialApp(

home: Material(

child: Text("Hello world!"),

),

)

to kolejne wywołanie konstruktora. Kiedy Flutter wykonuje ten kod, konstruuje aplikację MateralApp, której ekranem startowym jest obiekt Material.

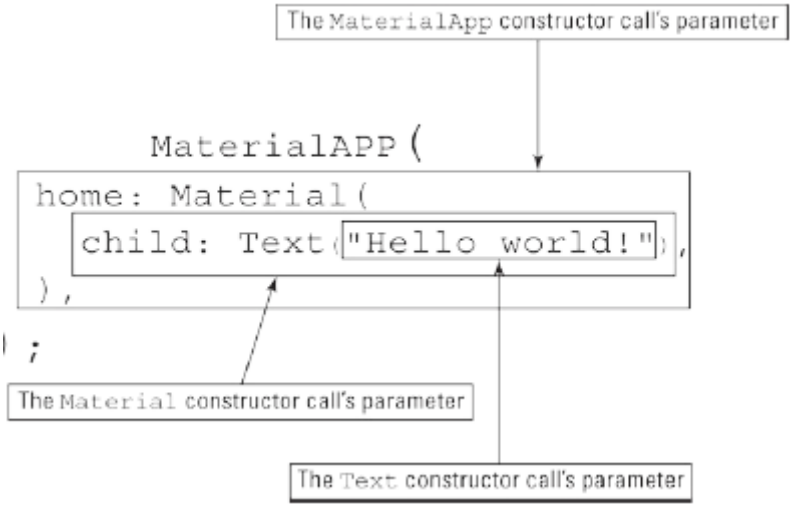
Oto sposób podsumowania tego wszystkiego:

Na listingu 1 obiekt MaterialApp ma obiekt Material, a obiekt Material ma obiekt Text. W tym zdaniu ważne jest pozornie niewinne użycie słów "ma". Aby zrozumieć kod z listingu 3.1, musisz wiedzieć, gdzie zaczynają się i kończą pary nawiasów. Ale znalezienie dopasowań między nawiasami otwierającymi i zamykającymi nie zawsze jest łatwe. Aby w tym pomóc , Androi Studio ma kilka sztuczek w swoim wirtualnym rękawie. Jeśli umieścisz kursor w pobliżu znaku nawiasu, Android Studio podświetli pasujący nawias. Ponadto możesz odwiedzić okno dialogowe Ustawienia lub Preferencje Android Studio. (W systemie Windows wybierz Plik ? Ustawienia. Na komputerze Mac wybierz Android Studio ? Preferencje.) W tym oknie dialogowym wybierz Edytor ? Ogólne ? Wygląd i zaznacz pole wyboru Pokaż etykiety zamykające w kodzie źródłowym Dart. Po zamknięciu okna dialogowego Android Studio wyświetla komentarze oznaczające końce wielu wywołań konstruktorów. (Zwróć uwagę na etykiety //Material i // MaterialApp na rysunku)

```
main.dart
import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Material(
        child: Text("Hello world!"),
      ), // Material
    ); // MaterialApp
  }
}
```

**Parametry konstruktora**

Każde wywołanie konstruktora ma listę parametrów (zwykle nazywaną listą parametrów). Na listingu 1 lista parametrów każdego konstruktora zawiera tylko jeden parametr




Wywołania konstruktora mogą mieć wiele parametrów lub nie mieć żadnych parametrów. Weźmy na przykład wywołanie tekstowe z listingu 1. W tym kodzie parametr "Witaj, świecie!" dostarcza informacje do Dart - informacje specyficzne dla widżetu Tekst tworzonego przez Dart. Spróbuj zmienić Text("Witaj świecie!") na Text("Witaj świecie!", textScaleFactor: 4.0). Gdy zapiszesz nowy kod, Android Studio wykona gorący restart, który zmieni wygląd aplikacji w twoim emulatorze.


Rozdział 1 opisuje różnicę między funkcjami gorącego restartu i gorącego przeładowania Fluttera. Obie funkcje stosują aktualizacje do aplikacji, gdy aplikacja jest uruchomiona. Aby wykonać gorący restart, po prostu zapisz swój kod. Aby wykonać przeładowanie na gorąco, naciśnij ikonę Uruchom w górnej części głównego okna Android Studio.

**Wywołanie konstruktora**

Text("Hello world!", textScaleFactor: 4.0)

zawiera dwa rodzaje parametrów:

"Witaj świecie!" jest parametrem pozycyjnym.

Parametr pozycyjny to parametr, którego znaczenie zależy od jego pozycji na liście parametrów. Podczas tworzenia nowego obiektu tekstowego znaki, które mają być wyświetlane, muszą zawsze znajdować się na początku listy. Możesz się o tym przekonać, zmieniając wywołanie konstruktora na następujący, nieprawidłowy kod:

Text(textScaleFactor: 4.0, "Hello world!")

// Bad code!!

W tym kodzie pozycyjne "Witaj, świecie!" parametr nie znajduje się na pierwszym miejscu listy. Jeśli więc wpiszesz tę linię w edytorze Android Studio, edytor oznaczy tę linię brzydkim czerwonym wskaźnikiem błędu. Szybki! Zmień go z powrotem, aby komunikat "Witaj, świecie!" parametr jest na pierwszym miejscu! Ty nie chcesz, aby Android Studio zrobiło na Tobie złe wrażenie

textScaleFactor: 4.0 to nazwany parametr.

Nazwany parametr to parametr, którego znaczenie zależy od słowa przed dwukropkiem. Wywołanie konstruktora Text może mieć wiele różnych nazwanych --parametrów, takich jak textScaleFactor, style i maxLinie. Nazwane parametry można zapisywać w dowolnej kolejności, o ile występują one po dowolnym parametrze pozycyjnym. Gdy podasz parametr textScaleFactor, parametr ten mówi Flutterowi, jak duży powinien być tekst. Jeśli nie podasz parametru textScaleFactor, Flutter użyje domyślnego współczynnika 1,0. Rozmiar tekstu zależy od kilku rzeczy, takich jak textScaleFactor i rozmiar czcionki parametru stylu. Na przykład poniższy kod tworzy Hello world! dwa razy większy niż na powyższym rysunku

Text("Hello world!", textScaleFactor: 4.0,

style: TextStyle(fontSize: 28.0))

Aplikacja pokazana na powyższym rysunku ma już textScaleFactor 4.0. Ale ma domyślny rozmiar czcionki, który wynosi 14,0. Ponieważ 28,0 to dwa razy 14,0, parametr fontSize: 28,0 podwaja rozmiar

tekstu. Uwaga na temat interpunkcji W Dart przecinkami oddziela się parametry konstruktora. W przypadku wszystkich oprócz najprostszych list parametrów kończysz je przecinkiem.

```
return MaterialApp(

home: Material(

child: Text("Hello world!"), // Trailing

comma after the child parameter

), // Trailing

comma after the home parameter

);
```

Bez końcowych przecinków Twój kod działa zgodnie z oczekiwaniami. Ale w następnej sekcji dowiesz się, jak sprawić, by Twój kod wyglądał dobrze w Android Studio. I bez końcowych przecinków, Android Studio nie daje z siebie wszystkiego.

**Nie ustępuj - po prostu wcinaj**

Spójrz jeszcze raz na Listing 1 i zwróć uwagę na wcięcia niektórych wierszy. Zgodnie z ogólną zasadą, jeśli jedna rzecz jest podporządkowana innej rzeczy, jej linia kodu jest wcięta bardziej niż ta inna rzecz. Na przykład na listingu 1 obiekt MaterialApp zawiera obiekt Material, więc wiersz home: Material ma większe wcięcie niż zwracany wiersz MaterialApp. Oto dwa fakty, o których należy pamiętać:

W programie Dart konieczne jest wcięcie.

Czekać! Jakie są te dwa fakty?

Jeśli zmienisz wcięcie w programie Dart, program nadal będzie działał. Oto poprawna przeróbka kodu z Listingu 1.

```
// Don't do this. It's poorly indented code.

import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Text("Hello world!"),

),

);

}

}
```

Gdy poprosisz Android Studio o uruchomienie tego słabo wciętego kodu, działa. Android Studio sumiennie uruchamia kod na Twoim urządzeniu wirtualnym lub fizycznym. Ale uruchomienie tego kodu nie jest wystarczająco dobre. Ten słabo wcięty kod jest ohydny. To prawie niemożliwe do odczytania. Wcięcie lub jego brak nie wskazuje struktury programu. Musisz przebrnąć przez słowa, aby odkryć, że widżet Materiał znajduje się w widżecie MateralApp. Zamiast pokazywać strukturę aplikacji na pierwszy rzut oka, ten kod sprawia, że twoje oczy błąkają się bez celu po morzu pozornie niezwiązanych ze sobą poleceń. Dobra wiadomość jest taka, że nie musisz uczyć się tworzenia wcięć w kodzie. Android Studio może wykonać wcięcie za Ciebie. Oto jak:

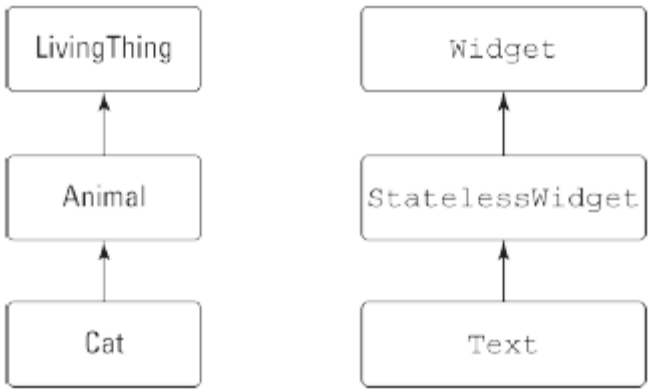
1. Otwórz okno dialogowe Ustawienia lub Preferencje Android Studio.

W systemie Windows wybierz Plik ? Ustawienia.

Na Macu wybierz Android Studio ? Preferencje.

2. W tym oknie dialogowym wybierz Języki i struktury ? Flutter, a następnie zaznacz pole wyboru Formatuj kod przy zapisywaniu. Znacznik wyboru mówi Androidowi Studio, aby poprawiał wcięcia kodu za każdym razem, gdy zapisujesz swoją pracę. Skoro już to robisz, równie dobrze możesz zaznaczyć pole wyboru w następnym polu - polu wyboru Organizuj importy przy zapisywaniu.

3. Wybierz OK, aby zamknąć okno dialogowe. Chazza! Kiedy uruchamiasz kod - lub po prostu zapisujesz kod - Android Studio naprawia wcięcia kodu. Jeśli chcesz mieć większą kontrolę nad zachowaniem Android Studio, nie majstruj w oknie dialogowym Ustawienia lub Preferencje. Zamiast tego, gdy chcesz naprawić wcięcie, umieść kursor w panelu Edytora, a następnie wybierz Kod ? Przeformatuj kod z głównego menu Android Studio. Tak czy inaczej, proszę odpowiednio wciąć kod.




W ten sam sposób każda instancja klasy Text Fluttera jest z definicji instancją klasy StatelessWidget Fluttera. Z kolei każda instancja klasy StatelessWidget jest instancją klasy Widget Fluttera. Tak więc każda instancja Text jest również instancją Widget.

* We Flutter prawie każdy obiekt jest w taki czy inny sposób instancją klasy Widget. Nieformalnie widżet jest elementem na ekranie użytkownika. Flutter przenosi tę ideę na inny poziom, z każdą częścią interfejs użytkownika (instancja Text, instancja Material, a nawet instancja MaterialApp) jest samodzielnym widżetem. - Na listingu 1 App0301 to nazwa klasy. w linii main() => runApp(App0301()); termin App0301() to kolejne wywołanie konstruktora. To wywołanie konstruuje instancję klasy App0301. Linia class App0301 rozszerza StatelessWidget

* a cały poniższy kod to deklaracja klasy App0301. Deklaracja mówi Dartowi, jaka to klasa i jakie rzeczy możesz z nią robić. W szczególności słowo "rozszerza się" w tym pierwszym wierszu powoduje, że dowolne wystąpienie klasy App0301 jest wystąpieniem klasy tatelessWidget. To wszystko, co musisz zrobić, aby instancje App0301 były instancjami klasy StatelessWidget.

Teraz masz kilka terminów o subtelnie różnych znaczeniach - klasa, obiekt, instancja i widżet. Na listingu 1 kod Text("Witaj, świecie!") coś konstruuje, ale dokładnie co konstruuje ten kod?

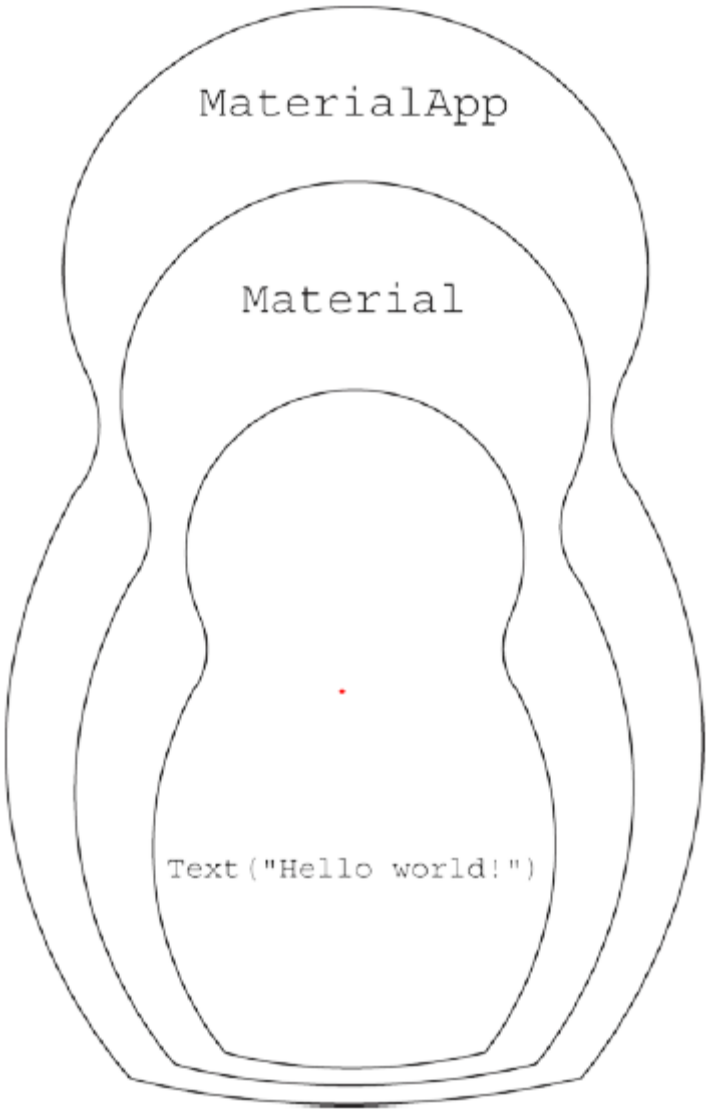
* Z punktu widzenia języka Dart Text("Witaj świecie!") konstruuje obiekt. W terminologii Dart nazywa się to instancją klasy Text.

* Z punktu widzenia Fluttera, Text("Witaj świecie!") tworzy widżet.

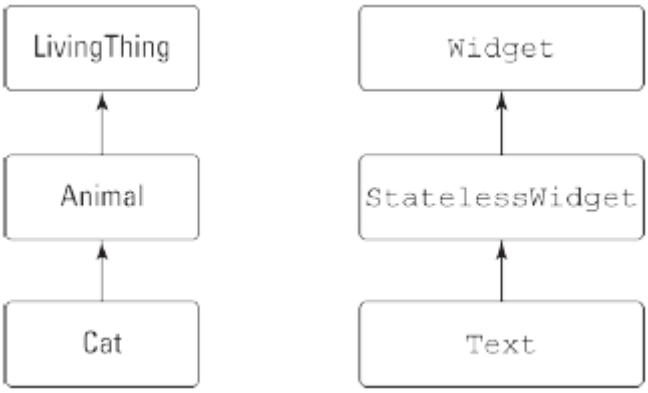
Jest to instancja klasy Text, a więc (…wina przez asocjację…) instancja klasy StatelessWidget oraz instancja klasy Widget.

**Krótki traktat o "wewnętrzności"**

W programie Dart możesz znaleźć widżety w innych widżetach. (Patrz rysunek 3.4.) W tym samym programie Dart znajdziesz -klasy w innych klasach. (Patrz rysunek 3.9.) Te dwa rodzaje "wewnętrzności" nie są tożsame. W rzeczywistości te dwa rodzaje "wewnętrzności" mają ze sobą niewiele wspólnego. Na rysunku 3.3 widżet Tekst jest dzieckiem widżetu Materiał. Nie oznacza to, że instancja Text jest również instancją klasy Material. Aby zrozumieć różnicę, pomyśl o dwóch rodzajach relacji: relacjach "jest" i relacjach "ma". Relacje, które opisuję w artykule "Co to jest wszystko o?" sekcja to relacje "ma". Na listingu 3.1 obiekt MaterialApp zawiera w sobie obiekt Material, a obiekt Material zawiera w sobie obiekt Text. Nie ma nic specjalnego w związkach "ma". Na podwórku mogą istnieć relacje "ma". Kot ma mysz, a mysz ma kawałek sera. Relacje, które opisałem we wcześniejszej sekcji "Klasy, obiekty i widżety", to relacje "jest". W każdym programie Flutter każdy obiekt Text jest obiektem StatelessWidget, a z kolei każdy obiekt StatelessWidget jest obiektem Widget. Na podwórku każdy kot jest zwierzęciem, a z kolei każde zwierzę jest żywą istotą. Nie miałoby sensu twierdzenie, że kot to mysz lub że obiekt materialny jest obiektem tekstowym. W ten sam sposób niepoprawne jest twierdzenie, że każdy kot ma zwierzę lub że każdy obiekt Text ma obiekt StatelessWidget. Te dwa rodzaje relacji - "ma" i "jest" - są zupełnie różne. Jeśli masz ochotę na bardziej formalną terminologię niż "ma" i "jest a", mam dla ciebie kilka: Łańcuch rzeczy połączonych relacją "ma" nazywany jest hierarchią kompozycji. Choć może to być frywolne, diagram na rysunku 3.4 ilustruje hierarchię kompozycji.




Łańcuch rzeczy połączony relacją "jest" nazywany jest hierarchią dziedziczenia. Diagramy na rysunku są częścią hierarchii klas Fluttera. Czy nie czujesz się lepiej teraz, gdy masz te fantazyjne terminy do rzucania?



We Flutterze prawie wszystko nazywa się "widżetem". Wiele klas to widżety. Gdy klasa jest widżetem, instancje klasy (dowolne obiekty zbudowane z tej klasy) są również nazywane widżetami.

**Dokumentacja jest twoim przyjacielem**

Być może zadajesz sobie pytanie, jak zapamiętasz wszystkie te nazwy: Text, StatelessWidget, MaterialApp i prawdopodobnie tysiące innych. Przykro mi to mówić, zadajesz złe pytanie. Nic nie zapamiętujesz. Kiedy używasz imienia wystarczająco często, zapamiętujesz je w sposób naturalny. Jeśli nie pamiętasz nazwy, sprawdzasz ją w dokumentacji Flutter online. (Czasami nie jesteś pewien, gdzie szukać nazwy, której szukasz. W takim przypadku musisz trochę poszperać.) Aby zobaczyć, co mam na myśli, skieruj przeglądarkę internetową na stronę https://api.flutter.dev /flutter/widgets/Textclass.html. Gdy to zrobisz, zobaczysz stronę z informacjami o klasie Tekst, przykładowym kodem i innymi rzeczami.


IGURE 3-10: Useful info about the Text class.

W prawym górnym rogu strony znajduje się lista konstruktorów tekstu. Na rysunku 3.10 są dwie możliwości: tekstowa i bogata. Jeśli wybierzesz łącze Tekst, zostanie wyświetlona strona z opisem wywołania konstruktora tekstu. Ta strona zawiera listę parametrów w wywołaniu konstruktora i zawiera inne przydatne informacje.

# Text constructor

```
const Text(
    String data, {
    Key key,
    TextStyle style,
    StrutStyle strutStyle,
    TextAlign textAlign,
    TextDirection textDirection,
    Locale locale,
    bool softWrap,
    TextOverflow overflow,
    double textScaleFactor,
    int maxLines,
    String semanticsLabel,
    TextWidthBasis textWidthBasis
})
```

Creates a text widget.

If the `style` argument is null, the text will use the style from

The `data` parameter must not be null.

## Implementation

Na stronie na rysunku  zauważ, że wszystkie parametry konstruktora oprócz jednego są ujęte w parę nawiasów klamrowych. Parametr, którego nie ma w nawiasach klamrowych (mianowicie dane typu String) jest jedynym parametrem pozycyjnym konstruktora. Każdy z parametrów w nawiasach klamrowych (w tym double textScaleFactor) jest nazwanym parametrem. Zawsze możesz liczyć na dokumentację Fluttera, która powie ci, jakiego rodzaju obiekty możesz, a jakich nie możesz umieszczać wewnątrz innych obiektów. Na przykład następujący kod jest skazany na niepowodzenie:

```
return MaterialApp(

child: Text("Hello world!"), // Don't do

this!

);
```

Jest to skazane na niepowodzenie, ponieważ zgodnie z dokumentacją Flutter konstruktor MaterialApp nie ma parametru o nazwie child.

**Sprawianie, że rzeczy wyglądają ładniej**

Aplikacja pokazana na rysunku 2 wygląda dość źle. Słowa Witaj, świecie! są przyciśnięte do lewego górnego rogu ekranu. Na szczęście Flutter oferuje prosty sposób na rozwiązanie tego problemu:

otaczasz widżet Tekst widżetem Centrum. Jak sama nazwa wskazuje, widżet Centrum wyśrodkowuje wszystko, co się w nim znajduje. Słowo Centrum jest nazwą klasy, więc każdy obiekt zbudowany z tej klasy jest nazywany egzemplarz tej klasy. W określeniu takim jak "Widżet centralny" słowo widżet sugeruje, że coś takiego jak Centrum (coś, co pomaga zarządzać układem ekranu) jest pewnego rodzaju komponentem. Komponentem jest fragment tekstu na ekranie, komponentem jest fragment materiału na ekranie, a komponentem jest również obiekt środkowy. Mimo że widżet Centrum nie świeci się gdzieś na stronie ekranu, widżet Center-er nadal jest komponentem. Częścią wielkiej siły Fluttera jest to, że Flutter traktuje wszystkie rzeczy w ten sam sposób. Kiedy tak wiele rzeczy jest widżetami, tak wiele rzeczy może służyć jako parametry w konstruktorach innych rzeczy. Ludzie, którzy wymyślają nazwy funkcji programistycznych, nazywają to funkcją komponowalności, a komponowalność to bardzo przyjemna funkcja. Istnieje kilka sposobów na otoczenie kodu widżetu Tekst kodem widżetu Centrum. Jednym ze sposobów jest umieszczenie kursora gdzieś w edytorze Android Studio, rozpoczęcie pisania i miej nadzieję, że poprawnie poruszasz się po gąszczu nawiasów. Lepszym sposobem jest wykonanie następujących czynności:

1. Umieść kursor na słowie Tekst w edytorze.

2. Naciśnij klawisze Alt+Enter.  W rezultacie pojawi się lista rozwijana.

3. Z listy rozwijanej wybierz Center Widget.

Listing 2 pokazuje, co otrzymujesz.

```
import 'package:flutter/material.dart';

main() => runApp(App0302());

class App0302 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Center(

child: Text("Hello world!"),

),

),

);

}

}
```

Na listingu 2 widżet Material ma element podrzędny widżetu Center, który z kolei ma element podrzędny widget Text. Widżet Tekst można traktować jako następcę widżetu Materiał. Flutter obsługuje ponowne uruchamianie na gorąco. Po dodaniu kodu Centrum do programu w edytorze Android Studio, zapisz zmiany, naciskając Ctrl+S (w Windows) lub Cmd+S (w Mac). Jeśli program z Listingu 1 był już uruchomiony, Flutter zastosuje zmiany i niemal natychmiast zaktualizuje ekran emulatora. W niektórych sytuacjach gorący restart nie działa. Zamiast aktualizować aplikację, Android Studio wyświetla komunikat o błędzie. Jeśli tak się stanie, spróbuj przeładować na gorąco. (Naciśnij

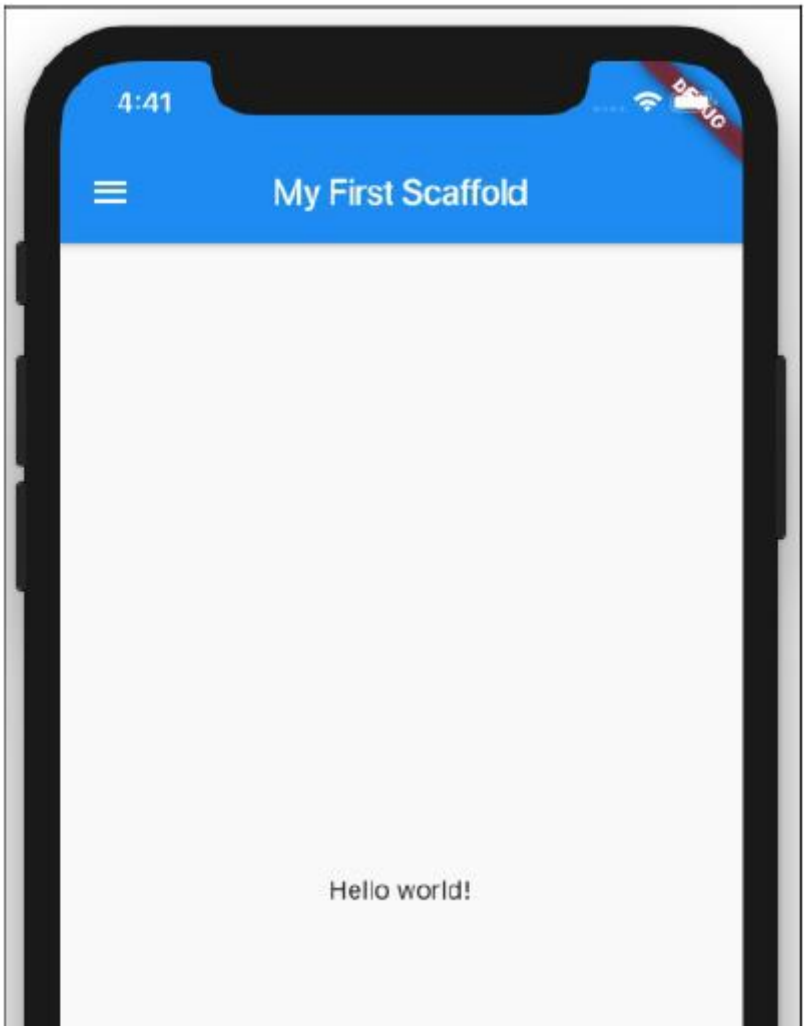
ikonę Uruchom w górnej części głównego okna Android Studio.) A co, jeśli przeładowanie na gorąco się nie powiedzie? W takim przypadku naciśnij ikonę Stop - czerwoną kwadratową ikonę, która znajduje się w tym samym rzędzie co ikona Uruchom. Po naciśnięciu ikony Zatrzymaj działanie aplikacji kończy się całkowicie. Naciśnięcie ikony Uruchom w celu rozpoczęcia od nowa może rozwiązać problem. Rysunek 12 pokazuje, co otrzymasz po uruchomieniu kodu z listingu 2.


**Tworzenie rusztowania**

Widżet Tekst na rysunku wygląda na bardzo samotny. Dodajmy trochę fanfar do podstawowej aplikacji. Listing 3 zawiera kod; Rysunki 13 i 14 przedstawiają nowy ekran.

```
import 'package:flutter/material.dart';

main() => runApp(App0303());

class App0303 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(
```

```
home: Scaffold(
appBar: AppBar(
title: Text("My First Scaffold"),
),
body: Center(
child: Text("Hello world!"),
),
drawer: Drawer(
child: Center(
child: Text("I'm a drawer."),
),
),
),
);
}
}
```


4:41
My First Scaffold
Hello world!


Strona główna aplikacji MaterialApp nie musi być widżetem Material. Na listingu 3.3 domem jest rusztowanie. Kiedy firmy budują drapacze chmur, tworzą rusztowania - tymczasowe drewniane konstrukcje wspierające pracowników na wysokich stanowiskach. W programowaniu rusztowanie jest strukturą, która zapewnia podstawową, często używaną funkcjonalność. - Konstruktor Scaffold z listingu 3 ma trzy parametry - pasek aplikacji, treść i szufladę. Na rysunkach 13 i 14 pasek aplikacji to ciemny obszar u góry ekranu. Ciało - to duży biały region zawierający Centrum z widżetem Tekst. Na rysunku 14 szuflada to duży biały obszar, który pojawia się, gdy użytkownik przesuwa palcem od lewej krawędzi ekranu. Szuflada pojawia się również, gdy użytkownik naciśnie ikonę "hamburgera" - trzy poziome kreski w pobliżu lewego górnego rogu ekranu. Ciało to nic specjalnego. Jest bardzo podobny do całego ekranu we wcześniejszych przykładach. Ale appBar i szuflada są nowe. AppBar i szuflada to dwie rzeczy, które możesz mieć podczas tworzenia rusztowania. Inne rzeczy udostępniane przez widżety Scaffold to paski nawigacyjne, pływające przyciski, dolne arkusze, przyciski stopki i inne. Listingi 1 i 2 mają widżety Materiał, a Listing 3 - Rusztowanie. Te widżety tworzą tła dla odpowiednich aplikacji. Jeśli usuniesz widżet Materiał z Listingu 1 lub 2, na ekranie Twojej aplikacji pojawi się brzydki bałagan. Otrzymujesz duże czerwone litery z żółtymi podkreśleniami na czarnym tle. To samo stanie się, gdy usuniesz rusztowanie z Listingu 3. Istnieją inne widżety, które mogą stanowić tło dla twoich aplikacji, ale najczęściej używane są materiały i rusztowania.

**Dodanie poprawek wizualnych**

Wypróbuj ten eksperyment: Zmień parametr appBar z Listingu 3 na fragment kodu z Listingu 4.

appBar: AppBar(

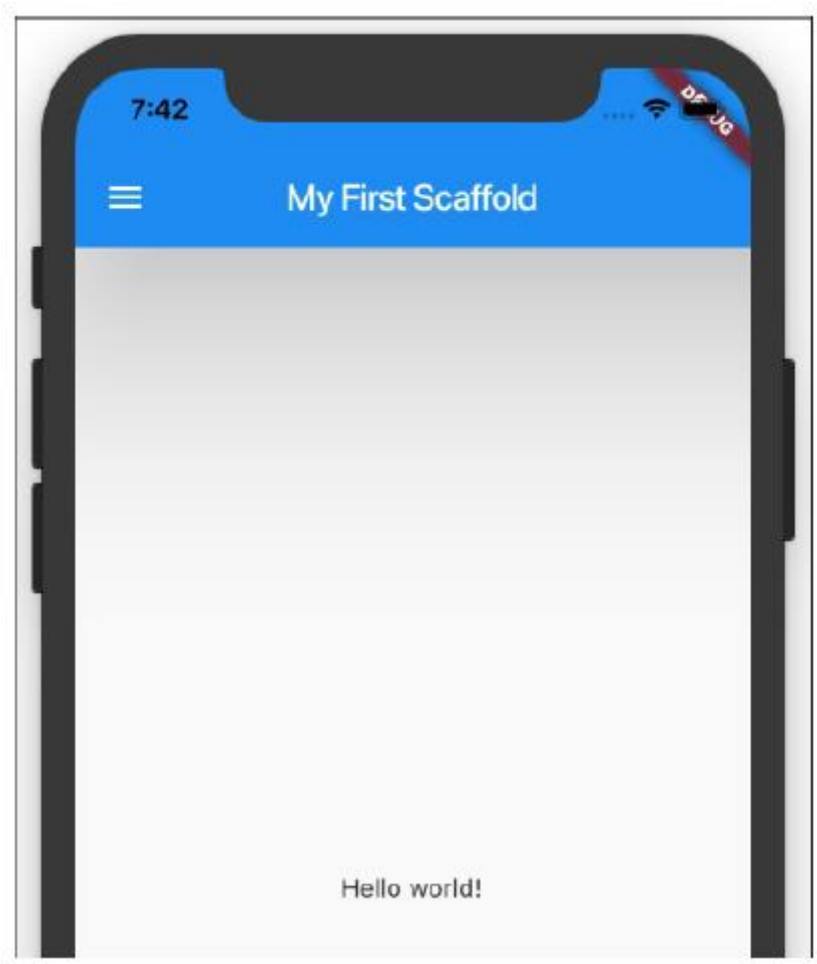
title: Text("My First Scaffold"),

elevation: 100,

brightness: Brightness.light,

)

Na rysunku próbuję pokazać efekt dodania parametrów elewacji i jasności do wywołania konstruktora AppBar. Może mi się nie udać, ponieważ efekt parametru elewacji jest subtelny.




W języku Google Material Design wyobrażasz sobie, że tło spoczywa na jakiejś płaskiej powierzchni, a inne komponenty są uniesione nad tło o pewną liczbę pikseli. W przypadku paska aplikacji domyślna wysokość wynosi 4, ale można zmienić wysokość paska za pomocą… poczekaj na to… parametru wysokości. Rzędna komponentu wpływa na kilka aspektów jego wyglądu. Ale w tej sekcji najbardziej oczywistą zmianą jest prawdopodobnie cień pod paskiem aplikacji. Możesz nie być w stanie dostrzec różnicy między cieniami na rysunkach 13 i 15, ale kiedy uruchamiasz kod na urządzeniu wirtualnym lub fizycznym, pasek AppBar z podniesieniem: 100 rzuca dość duży cień.  Być może zastanawiasz się, co oznacza 100 na wysokości: 100. Czy to milimetry, piksele, punkty czy lata świetlne? W rzeczywistości oznacza to "100 pikseli niezależnych od gęstości" - lub w skrócie "100 dps". Bez względu na to, jaki ekran posiada użytkownik, jeden dp to 1/160 cala. Tak więc wzniesienie: 100 oznacza 100/160 cala (lepiej znane jako pięć ósmych cala). Parametr jasności widżetu AppBar to jeszcze inna sprawa. Efekt dodania jasności: Brightness.light powie Flutterowi, że ponieważ pasek aplikacji jest jasny, tekst i ikony

na górze paska powinny być ciemne.  Ciemny tekst i ikony są dobrze widoczne na tle jasnego paska aplikacji.

**Funkcja enum Darta**

Interesującą cechę języka programowania Dart kryje Listing 3-4. Słowo Jasność odnosi się do czegoś, co nazywa się enum (wymawiane "ee-noom"). Słowo enum jest skrótem od wyliczenia. Wyliczenie to zbiór wartości, takich jak Brightness.light i Brightness.dark. Zwróć uwagę, jak na listingu 3.4 odwołujesz się do wartości wyliczenia. Nie używasz wywołania konstruktora. Zamiast tego używasz nazwy wyliczenia (np. Jasność), po której następuje kropka i unikatowa część nazwy wartości (np. jasny lub ciemny). Flutter ma wiele innych wbudowanych wyliczeń. Na przykład wyliczenie Orientation ma wartości Orientation.portrait i Orientation.landscape. Wyliczenie AnimationStatus ma wartości AnimationStatus.forward, AnimationStatus.reverse, AnimationStatus.completed i AnimationStatus.dismissed.

**Pozdrowienia ze słonecznej Kalifornii!**

Firma Google ogłosiła Material Design na swojej konferencji dla programistów w 2014 roku. Pierwsza wersja tego języka projektowania dotyczyła głównie urządzeń z Androidem, ale wersja 2 obejmowała niestandardowe branding dla iPhone'ów i innych urządzeń z systemem iOS. Widżet Material firmy Flutter działa na iPhone'ach z automatycznymi adaptacjami specyficznymi dla platformy. Możesz uruchomić dowolny z przykładów MaterialApp z tej książki na iPhonie, a także na telefonach z Androidem, ale jeśli chcesz mieć strategię projektowania iPhonefirst, możesz użyć Flutt

```
import 'package:flutter/cupertino.dart';

void main() => runApp(App0305());

class App0305 extends StatelessWidget {

Widget build(BuildContext context) {

return CupertinoApp(

home: CupertinoPageScaffold(

navigationBar:

CupertinoNavigationBar(),

child: Center(

child: Text("Hello world!"),

),

),

);
```

Listing 5 jest bardzo podobny do swojego kuzyna Material Design, Listing 3. Ale zamiast widżetów MaterialApp, Scaffold i AppBar, Listing 5 zawiera widżety CupertinoApp, CupertinoPageScaffold i CupertinoNavigationBar. Zamiast importować "package:flutter/material.dart", Listing 5 importuje package:flutter/cupertino.dart. (Ta deklaracja importu udostępnia bibliotekę widżetów Cupertino firmy Flutter do wykorzystania przez resztę kodu aukcji). Widżety Material Design i Cupertino firmy Flutter nie są do siebie całkowicie równoległe. Na przykład wywołanie konstruktora Scaffold na listingu

3.3 ma parametr body. Zamiast tego parametru wywołanie konstruktora CupertinoPageScaffold na listingu 3.5 ma parametr potomny. W razie wątpliwości sprawdź oficjalne strony dokumentacji Fluttera, aby dowiedzieć się, które nazwy parametrów należą do wywołań konstruktora poszczególnych widżetów. Możesz mieszać i dopasowywać widżety Material Design i Cupertino w tej samej aplikacji. Możesz nawet dostosować styl projektowania swojej aplikacji do różnych rodzajów telefonów. Możesz nawet umieścić kod następującego rodzaju w swojej aplikacji:

```
if (Platform.isAndroid) {

// Do Android-specific stuff

} if (

Platform.isIOS) {

// Do iOS-specific stuff

}
```

**Dodanie kolejnego widżetu**

Kiedy skończą ci się tematy do rozmowy, możesz zapytać ludzi o ich rodziny. Czasami dowiadujesz się ciekawych faktów, innym razem słyszysz listy skarg, a czasem dostajesz długi, nudny monolog. W ten czy inny sposób wypełnia niezręczną ciszę. Jeśli chodzi o zrozumienie relacji rodzinnych, jestem powolnym uczniem. Ktoś opowiada mi o teściowej żony jego drugiego kuzyna, a ja muszę przerwać rozmowę, żeby narysować diagram mentalny. W przeciwnym razie jestem po prostu zdezorientowany. Moje własne drzewo genealogiczne jest raczej proste. To była mama, tata i ja. Ludzie pytają mnie, czy byłam samotna jako jedynaczka. "Na pewno nie!" Mówię. "Jako jedynak nie musiałem się dzielić". Ta dyskusja o rodzinach jest moim wątpliwym wstępem do tematu widżetów kolumnowych. W poprzednich przykładach widżet Tekst był jedynakiem. Ale w końcu widżet Tekst musi nauczyć się udostępniać. (W przeciwnym razie widżet Tekst zostanie zepsuty, tak jak ja.) Jak umieścić dwoje dzieci na ciele rusztowania? Możesz ulec pokusie, aby spróbować tego:// DON'T DO THIS:

```
body: Center(

child: Text("Hello world!"),

child: AnotherWidget(&hellip;

)
```

Ale wywołanie konstruktora nie może mieć dwóch parametrów o tej samej nazwie. Więc co możesz zrobić? Flutter ma widżet Kolumna. Konstruktor widżetu Kolumna ma parametr potomny. Elementy podrzędne widżetu kolumny ustawiają się jeden pod drugim na ekranie. To brzmi obiecująco! Listing 6 zawiera kod, a rysunek 3-16 przedstawia wynikowy ekran.

```
import 'package:flutter/material.dart';

main() => runApp(App0306());

class App0306 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(
```

```
home: Scaffold(
appBar: AppBar(
title: Text("Adding Widgets"),
),
body: Column(
children: [
Text(
"Hello world!",
textScaleFactor: 2.0,
),
Text("It's lonely for me inside
this phone.")
],
),
),
);
}
}
```

Wywołanie konstruktora Column ma parametr child, a wartością parametru children jest lista. W języku programowania Dart lista to grupa obiektów. Pozycja każdego obiektu na liście jest nazywana indeksem. Wartości indeksu zaczynają się od 0 i idą w górę. Jednym ze sposobów tworzenia listy jest umieszczanie obiektów w nawiasach kwadratowych.

**RZECZY STRINGOWE**

W języku programowania Dart elementy ujęte w cudzysłowy (jak w przypadku "Witaj świecie!") nazywane są łańcuchami znaków. To grupa postaci, jedna po drugiej. Oto kilka przydatnych faktów na temat ciągów znaków: Aby utworzyć ciąg, możesz użyć podwójnego lub pojedynczego cudzysłowu. Innymi słowy, "Witaj, świecie!" to to samo co "Witaj świecie!". Łatwo jest umieścić pojedynczy cudzysłów wewnątrz podwójnego cudzysłowu. Zapoznaj się z ciągiem "Czuję się samotny w tym telefonie". na Listingu 3-6. Łatwo jest umieścić podwójny cudzysłów wewnątrz pojedynczego cudzysłowu. Na przykład następujący prawidłowy ciąg znaków:

'"Tak!" powiedziała.'

Używając znaków ukośnika odwrotnego (\), możesz umieścić dowolny rodzaj cudzysłowu wewnątrz dowolnego rodzaju łańcucha. Oto dwa przykłady:

"Czuję się samotny w tym telefonie".

"\"Tak!\" powiedziała."

łańcuch może rozciągać się na kilka wierszy, jeśli użyjesz potrójnych cudzysłowów. Oba te przykłady są prawidłowym kodem Dart:

'''A zwycięzcą jest …

Karol Wan

Doren!'''

"""A zwycięzcą jest …

Karol Wan

Doren!" """

Aby wkleić ciągi jeden po drugim, użyj znaku plus (+) lub spacji. Oba te przykłady są prawidłowym kodem Dart:

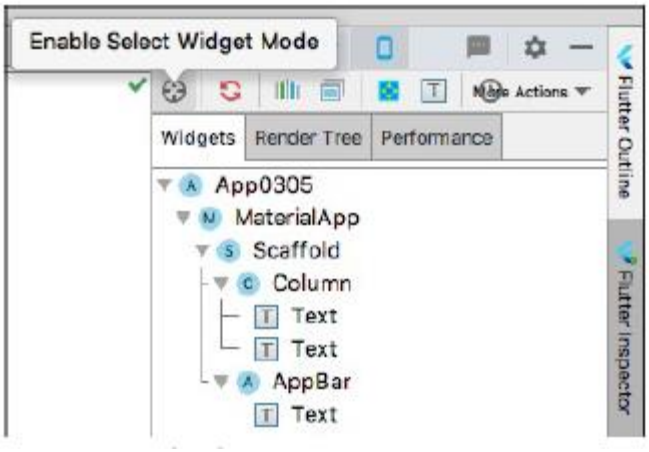
"Witaj" + "świat!"

"Witaj świecie!"

Wyśrodkowanie tekstu (część 1)

Rysunek wygląda dziwnie, ponieważ słowa są przyciśnięte do lewego górnego rogu.


W tej sekcji przeprowadzę Cię przez kilka kroków, aby zdiagnozować ten problem i go naprawić. 1. Gdy aplikacja z listingu  działa, poszukaj na prawym brzegu okna Android Studio przycisku paska narzędzi z napisem Flutter Inspector. Kliknij ten przycisk paska narzędzi. W rezultacie pojawia się Flutter Inspector.




2. W lewym górnym rogu narzędzia Flutter Inspector znajdź ikonę Włącz tryb wyboru widżetu. Kliknij tę ikonę.

3. Wybierz zakładkę Widżety Flutter Inspector.

4. W drzewie widżetów wybierz Kolumna. W rezultacie urządzenie, na którym działa Twoja aplikacja, dodaje wyróżnienie i niewielką etykietę do widżetu Kolumna na ekranie.

5. Dla zabawy wybierz kilka innych gałęzi w drzewie widżetów Flutter Inspector. Możesz określić granice prawie każdego z nich swoje widżety za pomocą tej techniki.

Podświetlenie na rysunku 3.20 informuje, że widżet Kolumna nie jest wyśrodkowany wewnątrz swojego nadrzędnego widżetu Scaffold i nie jest wystarczająco szeroki, aby wypełnić cały widżet Scaffold. Aby to naprawić, umieść widżet Kolumna wewnątrz widżetu Centrum. Umieść kursor na słowie Kolumna w edytorze Android Studio, a następnie postępuj zgodnie z instrukcjami na początku wcześniejszej sekcji "Sprawianie, że rzeczy wyglądają ładniej". Listing 7 pokazuje, co otrzymujesz.

```
import 'package:flutter/material.dart';

main() => runApp(App0307());

class App0307 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text("Adding Widgets"),

),

body: Center(

child: Column(

children: [

Text(
```

"Hello world!",

textScaleFactor: 2.0,

),

Text("It's lonely for me inside

this phone.")

],

),

),

),

);

}

}

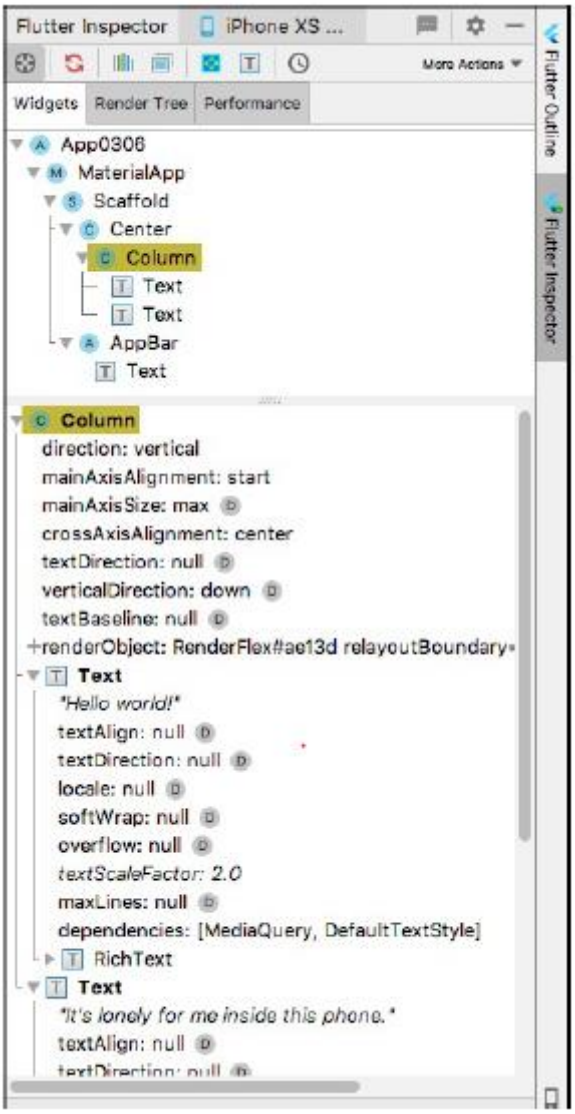
Kiedy zapiszesz zmiany, Android Studio wykona gorący restart i zobaczysz nowy i ulepszony ekran na rysunku

<img src="images/flut25.png">


**Wyśrodkowanie tekstu (część 2)**

Widżety Tekst na rysunku są wyśrodkowane w poziomie, ale nie w pionie. Aby wyśrodkować je w pionie, możesz majstrować przy widżecie Flutter's Center, ale jest o wiele prostszy sposób.

1. W narzędziu Flutter Inspector Android Studio wybierz widżet Kolumna. Dolny panel Flutter Inspector wyświetla wszystkie właściwości dowolnego wybranego widżetu. Czekać! Co to jest "właściwość"? Każdy obiekt ma właściwości, a każda właściwość każdego obiektu ma wartość. Na przykład każda instancja klasy Text Fluttera ma właściwość textScaleFactor. Na listingu wywołanie konstruktora ustawia właściwość textScaleFactor instancji Text na wartość 2,0. Wywołania konstruktora nie są jedynym sposobem ustawiania właściwości obiektów. Na rysunku dolny panel Flutter Inspector pokazuje wartości właściwości kierunku widżetu Column, jego właściwości mainAxisAlignment i wielu innych właściwości. Ponadto dwoje dzieci Text pojawia się w dolnym panelu Flutter Inspector.




2. W dolnym panelu najedź kursorem na właściwość mainAxisAlignment widżetu Kolumna. Gdy to zrobisz, Android Studio wyświetli wyskakujące okienko wyjaśniające znaczenie właściwości mainAxisAlignment. Tekst w tym wyskakującym okienku pochodzi automatycznie z oficjalnej dokumentacji Fluttera. Główną osią słupa jest niewidoczna linia biegnąca od góry do dołu słupa.

3. Ponownie w dolnym panelu najedź kursorem na słowo początek we właściwości mainAxisAlignment widżetu Kolumna. Nowe wyskakujące okienko mówi, że możesz zastąpić start dowolną wartością end, center, spaceBetween, spaceAround lub spaceEvenly.

4. W edytorze Android Studio dodaj parametr mainAxisAlignment do konstruktora widżetu Column.

```
import
'package:flutter/material.dart';
main() => runApp(App0308());
class App0308 extends
StatelessWidget {
Widget build(BuildContext context)
{
return MaterialApp(
home: Scaffold(
appBar: AppBar(
title: Text("Adding
Widgets"z),
),
body: Center(
child:
Column(mainAxisAlignment:
MainAxisAlignment.center,children: [
Text(
"Hello world!",
textScaleFactor:
2.0,
),
Text("It's lonely for
me inside this phone)
],
),
),
),
);
}
```

}import

Na listingu mainAxisAlignment to nazwa parametru, MainAxisAlignment to nazwa wyliczenia, a MainAxisAlignment.center to jedna z wartości wyliczenia.

5. Zapisz zmiany w edytorze, aby wykonać gorący restart. Na urządzeniu, na którym działa Twoja aplikacja, widżety tekstowe są wyśrodkowane w poziomie i w pionie.

Przykład w tej sekcji ilustruje aspekty widżetu Kolumna Fluttera, który wyświetla rzeczy od góry do dołu. Nie powinno dziwić, że Flutter ma widżet Row, który wyświetla rzeczy z boku na bok. Większość faktów dotyczących widżetu Kolumna odnosi się również do widżetu Wiersz. (Cóż, są prawdziwe, kiedy leżysz zamiast siedzieć prosto). Ponadto Flutter ma widżet ListView. Widżet ListView wyświetla rzeczy w dowolny sposób - od góry do dołu lub z boku na bok. Ponadto widżet ListView ma własną funkcję przewijania. Możesz umieścić 100 elementów na ListView, mimo że tylko 20 elementów mieści się na ekranie. Gdy użytkownik przewija ekran, elementy przesuwają się poza ekran, podczas gdy inne elementy przesuwają się dalej.

**Wyświetlanie obrazu**

Słowa są ładne, ale zdjęcia ładniejsze. W tej sekcji umieszczasz obraz na ekranie aplikacji Flutter.

1. W Android Studio rozpocznij nowy projekt Flutter. Nazwałem mój projekt app0308, ale nie musisz używać tej nazwy.

2. W oknie narzędzia projektu Android Studio kliknij prawym przyciskiem myszy nazwę projektu. W rezultacie pojawia się menu kontekstowe.

3. Z menu kontekstowego wybierz Nowy ? Katalog. (Patrz rysunek 3-26.) W rezultacie pojawi się okno dialogowe Nowy katalog. Jak wygodnie!

4. W oknie dialogowym wpisz nazwę zasobów, a następnie naciśnij klawisz Enter. Szczerze mówiąc, możesz nazwać ten nowy katalog prawie tak, jak chcesz. Ale jeśli nie nazwiesz tego aktywa, zdezorientujesz innych programistów Fluttera.

5. Sprawdź okno Project Tool, aby upewnić się, że drzewo projektu ma nową gałąź zasobów . Doświadczeni programiści Flutter tworzą podkatalog images nowego katalogu zasobów. Nie będę się tym teraz przejmować.

6. Znajdź plik obrazu. Wyszukaj plik obrazu na dysku twardym komputera deweloperskiego. Szukaj nazw plików kończących się na .png, .jpg, .jpeg lub .gif. Jeśli Twój Eksplorator plików lub Finder nie wyświetla rozszerzeń nazw plików (takich jak .png, .jpg, .jpeg lub .gif w przypadku plików graficznych)

7. Skopiuj plik obrazu w Eksploratorze plików lub programie +-Finder komputera programistycznego. Oznacza to, że kliknij prawym przyciskiem myszy nazwę pliku obrazu. W wyświetlonym menu kontekstowym wybierz opcję Kopiuj.

8. Korzystając z okna Project Tool w Android Studio, wklej plik obrazu do katalogu asset. Oznacza to, że kliknij prawym przyciskiem myszy gałąź zasobów. W wynikowym menu kontekstowym wybierz Wklej. W wyświetlonym oknie dialogowym wpisz nazwę pliku obrazu, a następnie naciśnij klawisz Enter. Kiedy to wszystko zrobiłem, nazwałem plik MyImage.png, ale nie musisz używać tej nazwy.

9. Otwórz plik pubspec.yaml swojego projektu. Mówiąc dokładniej, kliknij dwukrotnie gałąź pubspec.yaml w drzewie okna Narzędzia projektu. Oto zabawny fakt: rozszerzenie .yaml oznacza jeszcze jeden język znaczników.

10. W pliku pubspec.yaml poszukaj porad dotyczących dodawania zasobów do projektu. Porada może wyglądać mniej więcej tak:

```
# To add assets to your application,

# add an assets section, like this:

# assets:

# - images/a_dot_burr.jpeg

# - images/a_dot_ham.jpeg
```

(Jeśli się zastanawiasz, nazwy plików a_dot_burr.jpeg i a_dot_ham.jpeg odnoszą się do Aarona Burra i Alexandra Hamiltona. Te nazwy plików występują wiele razy w oficjalnej dokumentacji Flutter. Flutter to technologia stojąca za aplikacją mobilną dla musicalu Hamilton na Broadwayu .) W pliku .yaml hashtag (#) mówi komputerowi, aby ignorował wszystko w pozostałej części wiersza. Tak więc w tej części pliku .yaml żadna z linii nie ma żadnego efektu.

11. Usuń hashtagi z dwóch linii. W drugim wierszu zmień nazwę pliku obrazu na wybraną w kroku 8. Kiedy to zrobię, mój plik pubspec.yaml zawiera następujący tekst:

```
# To add assets to your application,

# add an assets section, like this:

assets:

- MyImage.png

# - images/a_dot_ham.jpeg
```

Używam nazwy MyImage.png zamiast images/MyImage.png, ponieważ w kroku 5 nie utworzyłem katalogu images. Często zapominam wprowadzić niezbędne zmiany w pliku pubspec.yaml. Staraj się nie zapomnieć o tym kroku. Kiedy zapomnisz (a prawie wszyscy to robią), wróć i edytuj plik pubspec.yaml projektu.

12. Zastąp cały kod w pliku main.dart kodem z Listingu 3-9. Użyj własnej nazwy klasy i nazwy pliku zamiast moich nazw App0309 i MyImage.png.

```
import

'package:flutter/material.dart';

main() => runApp(App0309());

class App0309 extends

StatelessWidget {

Widget build(BuildContext context)

{

return MaterialApp(

home: Scaffold(

appBar: AppBar(
```

```
title: Text("My First

Image"),

),

body: Center(

child:

Image.asset('MyImage.png'),

),

),

);

}

}
```

13. Let rip.

Oznacza to, że uruchom kod na urządzeniu wirtualnym lub fizycznym.

W tym momencie chcę całkowicie wyjaśnić jedną rzecz: nie jestem narcyzem. Powodem, dla którego używam tego rysunku, jest fascynacja rekurencją. Lubię mieć odniesienie do tej książki w tej książce. (Poza tym jestem trochę narcyzem). Flutter ma klasę Image, a klasa Image ma kilka różnych konstruktorów. Konstruktor Image.asset pobiera plik z miejsca w katalogu projektu Flutter. Aby pobrać obraz z Internetu, wywołujesz inny konstruktor - konstruktor Image.network. Aby uzyskać obraz z dowolnego miejsca na dysku twardym (gdzieś poza katalogiem projektu Flutter), możesz wywołać konstruktor Image.file. Każdy z tych konstruktorów jest nazywany konstruktorem nazwanym. W każdym przypadku elementy po kropce (.asset, .network i .file) to nazwa konkretnego konstruktora.

**Hej, poczekaj chwilę…**

Omówiono kilka podstawowych idei związanych z tworzeniem aplikacji Dart i Flutter. Zaczynasz od programu Hello World i wprowadzasz w nim kilka zmian. Robiąc to wszystko, budujesz słownik przydatnych pojęć - takich jak klasy, konstruktory, wyliczenia i widżety. Zrobiłeś to wszystko, podczas gdy ja sprytnie odwróciłem twoją uwagę od kilku linijek programu Hello World. Co robią pierwsze cztery wiersze programu Hello World? Dlaczego coś zwracasz, gdy tworzysz MaterialApp? Odpowiedzi na te i podobne pytania znajdują się w następnym rozdziale. Na co czekasz? Czytaj!K<Kbr>

? "Szczęśliwe szlaki dla ciebie / Do ponownego spotkania" ?

NAPISANE PRZEZ DALE EVANS, ŚPIEWANE PRZEZ

ROY ROGERS I DALE EVANS NA

"POKAZ ROYA ROGERSA", 1944-1957

„Helo” od Fluttera

Słowo cześć jest względną nowością w języku angielskim. Jego pierwsze znane użycie w druku miało miejsce w Norwich, Connecticut, Courier w 1826 roku. Alexander Graham Bell, wynalazca telefonu, uważał, że rozmowy telefoniczne powinny zaczynać się od terminu Ahoy! ale najwyraźniej Thomas Edison wolał Hello, a wczesne książki telefoniczne zalecały wybór Edisona. Według legendy pierwszy program komputerowy, który drukował tylko „Witaj, świecie!” został napisany przez Briana Kernighana jako część dokumentacji języka programowania BCPL. Pierwszy publiczny występ takiego programu miał miejsce w książce Kernighana i Ritchiego z 1972 roku, The C Programming Language. W dzisiejszych czasach termin program Hello world lub po prostu program Hello odnosi się do każdego prostego kodu dla czyjegoś pierwszego kontaktu z nowym językiem lub nowym frameworkiem. Ten rozdział zawiera prosty program Flutter „Hello world” i kilka ozdób. Możesz uruchomić kod, przeanalizować go, zmienić i dobrze się z nim bawić.

<>Pierwsze rzeczy na pierwszym miejscu

Listing 1 zawiera Twoją pierwszą aplikację Flutter.

```
import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(
```

child: Text("Hello world!"),

),

);

}

}

Jeśli wolisz samodzielnie wpisać kod, wykonaj następujące czynności:

1. Utwórz nowy projekt Flutter. Jak zwykle Android Studio tworzy dla Ciebie plik pełen kodu Dart. Nazwa pliku to main.dart.

2. Upewnij się, że kod main.dart pojawił się w edytorze Android Studio. Jeśli nie, rozwiń drzewo w oknie narzędzia Projekt po lewej stronie głównego okna Android Studio. Poszukaj gałęzi lib, aw gałęzi lib gałęzi main.dart. Kliknij dwukrotnie tę gałąź main.dart.

3. W edytorze Android Studio usuń cały kod main.dart.

4. W edytorze Android Studio wpisz kod, który widzisz na Listingu 1.

Jeśli zmienisz małą literę w słowie na wielką literę, możesz zmienić znaczenie słowa. ZMIANA przypadku może sprawić, że całe słowo przestanie mieć znaczenie i stanie się bez znaczenia. W pierwszym wierszu listingu 3.1 nie można zastąpić importu importem. JEŚLI TO ZROBISZ, CAŁY PROGRAM PRZESTAJE DZIAŁAĆ. Spróbuj i przekonaj się sam!

Rysunek  przedstawia gotowy produkt

<img src=”images/flut7.png”>

```
main.dart
1   import 'package:flutter/material.dart';
2
3   main() => runApp(App0301());
4
5   class App0301 extends StatelessWidget {
6     Widget build(BuildContext context) {
7       return MaterialApp(
8         home: Material(
9           child: Text("Hello world!"),
10          ),
11        );
12    }
13  }
14
```

5. Uruchom nową aplikację.

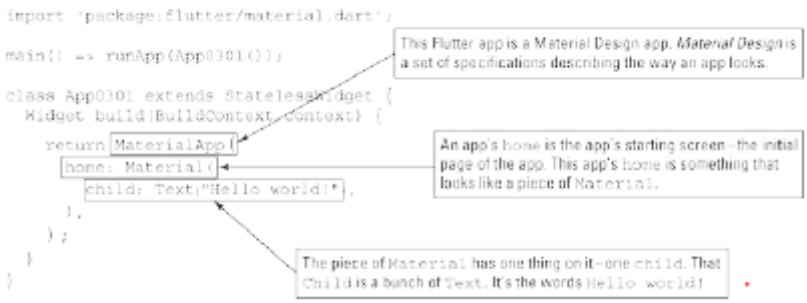
Rysunek 3-2 pokazuje, co widzisz po uruchomieniu aplikacji Flutter z Listingu 1. Aplikacja wygląda dość źle, ale przynajmniej możesz zobaczyć mały Hello world! w lewym górnym rogu ekranu. Kosmetycznymi aspektami aplikacji zajmę się w dalszej części .

<img src=”images/flut8.png”>

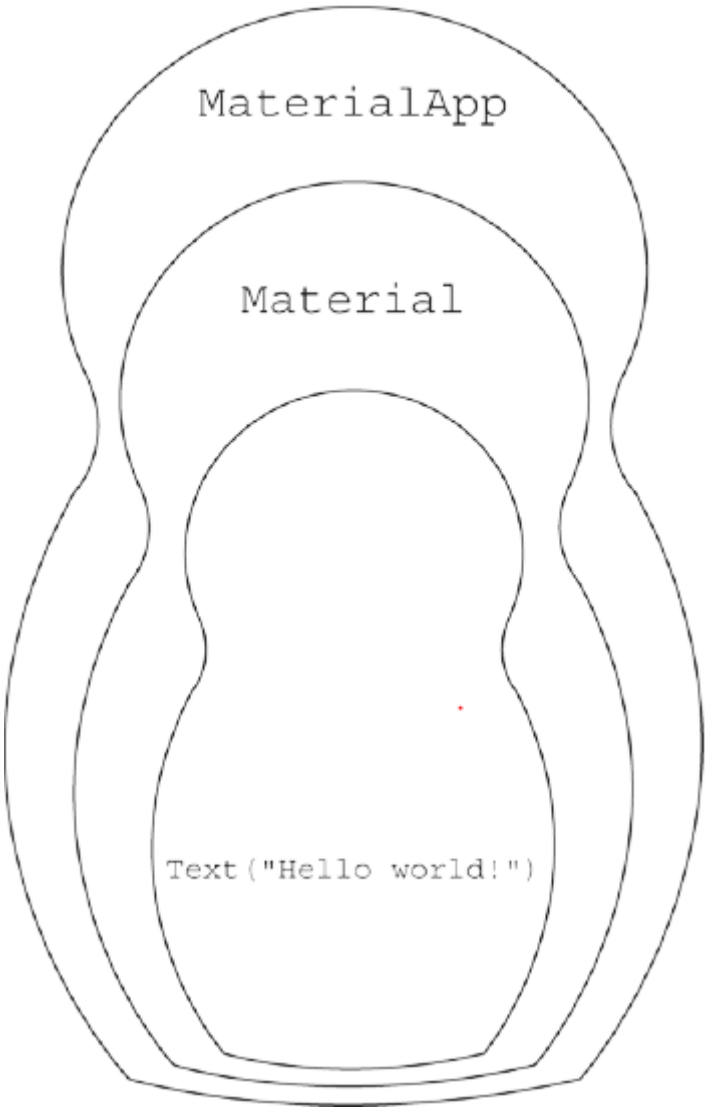

W edytorze Android Studio możesz zobaczyć czerwone znaczniki. Jeśli tak, najedź kursorem na znacznik i przeczytaj wyświetlone wyjaśnienie. Niektóre wyjaśnienia są łatwe do zrozumienia; inni nie. Im więcej masz praktyki w interpretowaniu tych komunikatów, tym bardziej jesteś wprawny w rozwiązywaniu problemów. Inną rzeczą, którą możesz spróbować, jest wybranie zakładki Analiza rzutów u dołu głównego okna Android Studio. Ta karta zawiera listę wielu miejsc w twoim projekcie, które zawierają wątpliwy kod. W przypadku dowolnego elementu na liście czerwona ikona oznacza błąd — coś, co należy naprawić. (Jeśli tego nie naprawisz, plik app nie można uruchomić.) Każda inna ikona koloru oznacza ostrzeżenie — coś, co nie zapobiegnie uruchomieniu kodu, ale może być warte rozważenia. W następnych kilku sekcjach rozbiorę kod z Listingu 3-1. Analizuję kod z wielu punktów widzenia. Wyjaśniam, co robi kod, dlaczego robi to, co robi i co może robić inaczej.

O czym to jest?

Kiedy spojrzysz na Listing 1, możesz zobaczyć słowa, znaki interpunkcyjne i wcięcia, ale to nie jest to, co widzą doświadczeni programiści Fluttera. Widzą ogólny zarys. Widzą wielkie idee w pełnych zdaniach. Rysunek 3-3 pokazuje, jak wygląda Listing 1 dla doświadczonego programisty.




Program Flutter jest jak zestaw rosyjskich lalek matrioszek. Jest to rzecz w rzeczy w innej rzeczy i tak dalej, aż dojdziesz do punktu końcowego.




Listing 1 zawiera tekst wewnątrz elementu Material, który z kolei znajduje się wewnątrz aplikacji MaterialApp. Słowa Text, Material i MaterialApp rozpoczynają polecenia konstruowania rzeczy. W terminologii języka Dart słowa Text, Material i MaterialApp to nazwy wywołań konstruktora. Oto wewnętrzna historia:

Kod

Tekst("Witaj świecie!")

jest wywołaniem konstruktora. Kiedy Flutter wykonuje ten kod, konstruuje obiekt Text. Ten obiekt Text zawiera słowa Witaj, świecie!

Kod

Material(child: Text("Hello world!"),

)

jest kolejnym wywołaniem konstruktora. Kiedy Flutter wykonuje ten kod, konstruuje obiekt Material. Ten obiekt Material zawiera wspomniany wcześniej obiekt Text

Obiekt Material ma pewne cechy charakterystyczne dla materiału fizycznego, takiego jak kawałek tkaniny. Ma określony kształt. Może być wyniesiony z powierzchni pod nim. Możesz go przesunąć lub uszczypnąć. To prawda, że tło na rysunku 3.2 nie przypomina kawałka tkaniny. Ale imitowanie faktury materiału już nie jest celem Material Design. Celem Material Design jest stworzenie języka do opisywania stanu komponentów na ekranie użytkownika i opisywania, w jaki sposób te komponenty są ze sobą powiązane.

Kod:

```
MaterialApp(

home: Material(

child: Text("Hello world!"),

),

)
```

to kolejne wywołanie konstruktora. Kiedy Flutter wykonuje ten kod, konstruuje aplikację MateralApp, której ekranem startowym jest obiekt Material.

Oto sposób podsumowania tego wszystkiego:

Na listingu 1 obiekt MaterialApp ma obiekt Material, a obiekt Material ma obiekt Text. W tym zdaniu ważne jest pozornie niewinne użycie słów „ma”. Aby zrozumieć kod z listingu 3.1, musisz wiedzieć, gdzie zaczynają się i kończą pary nawiasów. Ale znalezienie dopasowań między nawiasami otwierającymi i zamykającymi nie zawsze jest łatwe. Aby w tym pomóc , Androi Studio ma kilka sztuczek w swoim wirtualnym rękawie. Jeśli umieścisz kursor w pobliżu znaku nawiasu, Android Studio podświetli pasujący nawias. Ponadto możesz odwiedzić okno dialogowe Ustawienia lub Preferencje Android Studio. (W systemie Windows wybierz Plik ⇒ Ustawienia. Na komputerze Mac wybierz Android Studio ⇒ Preferencje.) W tym oknie dialogowym wybierz Edytor ⇒ Ogólne ⇒ Wygląd i zaznacz pole wyboru Pokaż etykiety zamykające w kodzie źródłowym Dart. Po zamknięciu okna dialogowego Android Studio wyświetla komentarze oznaczające końce wielu wywołań konstruktorów. (Zwróć uwagę na etykiety //Material i // MaterialApp na rysunku)

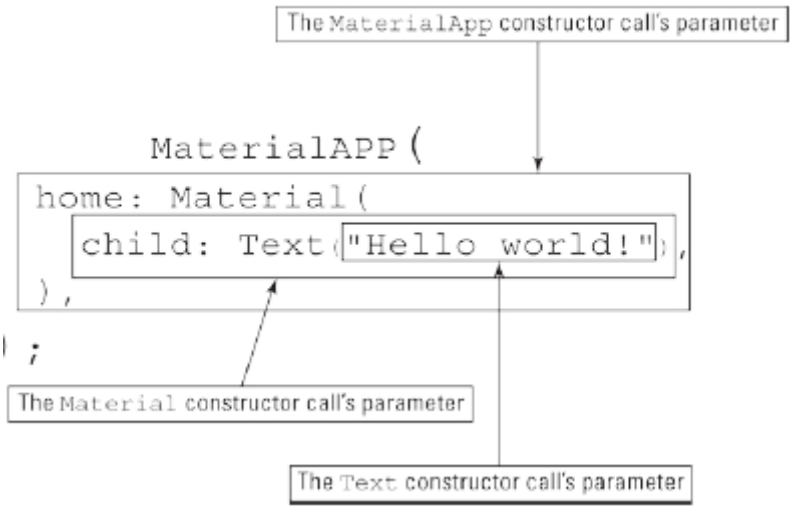
<img src=”images/flut11.png”>

```
main.dart
import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Material(
        child: Text("Hello world!"),
      ), // Material
    ); // MaterialApp
  }
}
```

Parametry konstruktora<br<

Każde wywołanie konstruktora ma listę parametrów (zwykle nazywaną listą parametrów). Na listingu 1 lista parametrów każdego konstruktora zawiera tylko jeden parametr<.,br>

<img src=”images/flut12.png”>




Wywołania konstruktora mogą mieć wiele parametrów lub nie mieć żadnych parametrów. Weźmy na przykład wywołanie tekstowe z listingu 3.1. W tym kodzie parametr „Witaj, świecie!” dostarcza informacje do Dart — informacje specyficzne dla widżetu Tekst tworzonego przez Dart. Spróbuj zmienić Text("Witaj świecie!") na Text("Witaj świecie!", textScaleFactor: 4.0). Gdy zapiszesz nowy kod, Android Studio wykona gorący restart, który zmieni wygląd aplikacji w twoim emulatorze.

<img src=”images/flut13.png”>


Rozdział 1 opisuje różnicę między funkcjami gorącego restartu i gorącego przeładowania Fluttera. Obie funkcje stosują aktualizacje do aplikacji, gdy aplikacja jest uruchomiona. Aby wykonać gorący restart, po prostu zapisz swój kod. Aby wykonać przeładowanie na gorąco, naciśnij ikonę Uruchom w górnej części głównego okna Android Studio.

Wywołanie konstruktora

Text("Hello world!", textScaleFactor: 4.0)

zawiera dwa rodzaje parametrów:

"Witaj świecie!" jest parametrem pozycyjnym.

Parametr pozycyjny to parametr, którego znaczenie zależy od jego pozycji na liście parametrów. Podczas tworzenia nowego obiektu tekstowego znaki, które mają być wyświetlane, muszą zawsze znajdować się na początku listy. Możesz się o tym przekonać, zmieniając wywołanie konstruktora na następujący, nieprawidłowy kod:

Text(textScaleFactor: 4.0, "Hello world!")

// Bad code!!

W tym kodzie pozycyjne „Witaj, świecie!” parametr nie znajduje się na pierwszym miejscu listy. Jeśli więc wpiszesz tę linię w edytorze Android Studio, edytor oznaczy tę linię brzydkim czerwonym wskaźnikiem błędu. Szybki! Zmień go z powrotem, aby komunikat „Witaj, świecie!” parametr jest na pierwszym miejscu! Ty nie chcesz, aby Android Studio zrobiło na Tobie złe wrażenie

textScaleFactor: 4.0 to nazwany parametr.

Nazwany parametr to parametr, którego znaczenie zależy od słowa przed dwukropkiem. Wywołanie konstruktora Text może mieć wiele różnych nazwanych --parametrów, takich jak textScaleFactor, style i maxLinie. Nazwane parametry można zapisywać w dowolnej kolejności, o ile występują one po dowolnym parametrze pozycyjnym. Gdy podasz parametr textScaleFactor, parametr ten mówi Flutterowi, jak duży powinien być tekst. Jeśli nie podasz parametru textScaleFactor, Flutter użyje domyślnego współczynnika 1,0. Rozmiar tekstu zależy od kilku rzeczy, takich jak textScaleFactor i rozmiar czcionki parametru stylu. Na przykład poniższy kod tworzy Hello world! dwa razy większy niż na powyższym rysunku

Text("Hello world!", textScaleFactor: 4.0,

style: TextStyle(fontSize: 28.0))

Aplikacja pokazana na powyższym rysunku ma już textScaleFactor 4.0. Ale ma domyślny rozmiar czcionki, który wynosi 14,0. Ponieważ 28,0 to dwa razy 14,0, parametr fontSize: 28,0 podwaja rozmiar tekstu. Uwaga na temat interpunkcji W Dart przecinkami oddziela się parametry konstruktora. W przypadku wszystkich oprócz najprostszych list parametrów kończysz je przecinkiem.

```
return MaterialApp(

home: Material(

child: Text("Hello world!"), // Trailing

comma after the child parameter

), // Trailing

comma after the home parameter

);
```

Bez końcowych przecinków Twój kod działa zgodnie z oczekiwaniami. Ale w następnej sekcji dowiesz się, jak sprawić, by Twój kod wyglądał dobrze w Android Studio. I bez końcowych przecinków, Android Studio nie daje z siebie wszystkiego.

Nie ustępuj — po prostu wcinaj

Spójrz jeszcze raz na Listing 3-1 i zwróć uwagę na wcięcia niektórych wierszy. Zgodnie z ogólną zasadą, jeśli jedna rzecz jest podporządkowana innej rzeczy, jej linia kodu jest wcięta bardziej niż ta inna rzecz. Na przykład na listingu 3.1 obiekt MaterialApp zawiera obiekt Material, więc wiersz home: Material ma większe wcięcie niż zwracany wiersz MaterialApp. Oto dwa fakty, o których należy pamiętać:

W programie Dart konieczne jest wcięcie.

Czekać! Jakie są te dwa fakty?

Jeśli zmienisz wcięcie w programie Dart, program nadal będzie działał. Oto poprawna przeróbka kodu z Listingu 1.

```
// Don’t do this. It’s poorly indented code.

import 'package:flutter/material.dart';

main() => runApp(App0301());

class App0301 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Text("Hello world!"),

),

);

}

}
```

Gdy poprosisz Android Studio o uruchomienie tego słabo wciętego kodu, działa. Android Studio sumiennie uruchamia kod na Twoim urządzeniu wirtualnym lub fizycznym. Ale uruchomienie tego kodu nie jest wystarczająco dobre. Ten słabo wcięty kod jest ohydny. To prawie niemożliwe do

odczytania. Wcięcie lub jego brak nie wskazuje struktury programu. Musisz przebrnąć przez słowa, aby odkryć, że widżet Materiał znajduje się w widżecie MateralApp. Zamiast pokazywać strukturę aplikacji na pierwszy rzut oka, ten kod sprawia, że twoje oczy błąkają się bez celu po morzu pozornie niezwiązanych ze sobą poleceń. Dobra wiadomość jest taka, że nie musisz uczyć się tworzenia wcięć w kodzie. Android Studio może wykonać wcięcie za Ciebie. Oto jak:

1. Otwórz okno dialogowe Ustawienia lub Preferencje Android Studio.

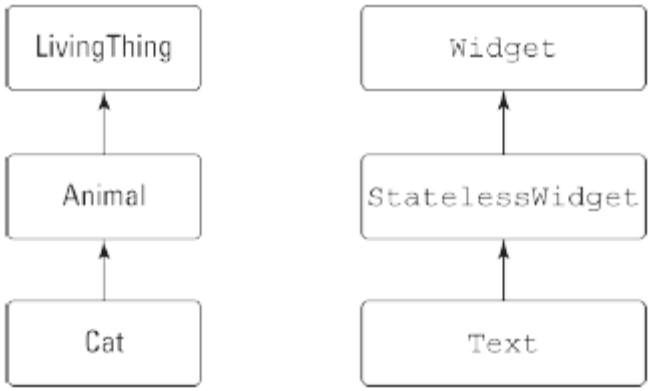
W systemie Windows wybierz Plik ⇒ Ustawienia.

Na Macu wybierz Android Studio ⇒ Preferencje.

2. W tym oknie dialogowym wybierz Języki i struktury ⇒ Flutter, a następnie zaznacz pole wyboru Formatuj kod przy zapisywaniu. Znacznik wyboru mówi Androidowi Studio, aby poprawiał wcięcia kodu za każdym razem, gdy zapisujesz swoją pracę. Skoro już to robisz, równie dobrze możesz zaznaczyć pole wyboru w następnym polu — polu wyboru Organizuj importy przy zapisywaniu.

3. Wybierz OK, aby zamknąć okno dialogowe. Chazza! Kiedy uruchamiasz kod — lub po prostu zapisujesz kod — Android Studio naprawia wcięcia kodu. Jeśli chcesz mieć większą kontrolę nad zachowaniem Android Studio, nie majstruj w oknie dialogowym Ustawienia lub Preferencje. Zamiast tego, gdy chcesz naprawić wcięcie, umieść kursor w panelu Edytora, a następnie wybierz Kod ⇒ Przeformatuj kod z głównego menu Android Studio. Tak czy inaczej, proszę odpowiednio wciąć kod.

<img src=”images/flut14.png”>




W ten sam sposób każda instancja klasy Text Fluttera jest z definicji instancją klasy StatelessWidget Fluttera. Z kolei każda instancja klasy StatelessWidget jest instancją klasy Widget Fluttera. Tak więc każda instancja Text jest również instancją Widget.

* We Flutter prawie każdy obiekt jest w taki czy inny sposób instancją klasy Widget. Nieformalnie widżet jest elementem na ekranie użytkownika. Flutter przenosi tę ideę na inny poziom, z każdą częścią interfejs użytkownika (instancja Text, instancja Material, a nawet instancja MaterialApp) jest samodzielnym widżetem. — Na listingu 3.1 App0301 to nazwa klasy. w linii main() => runApp(App0301()); termin App0301() to kolejne wywołanie konstruktora. To wywołanie konstruuje instancję klasy App0301. Linia class App0301 rozszerza StatelessWidget

* a cały poniższy kod to deklaracja klasy App0301. Deklaracja mówi Dartowi, jaka to klasa i jakie rzeczy możesz z nią robić. W szczególności słowo „rozszerza się” w tym pierwszym wierszu powoduje, że

dowolne wystąpienie klasy App0301 jest wystąpieniem klasy tatelessWidget. To wszystko, co musisz zrobić, aby instancje App0301 były instancjami klasy StatelessWidget.

Teraz masz kilka terminów o subtelnie różnych znaczeniach — klasa, obiekt, instancja i widżet. Na listingu 1 kod Text("Witaj, świecie!") coś konstruuje, ale dokładnie co konstruuje ten kod?

* Z punktu widzenia języka Dart Text("Witaj świecie!") konstruuje obiekt. W terminologii Dart nazywa się to instancją klasy Text.

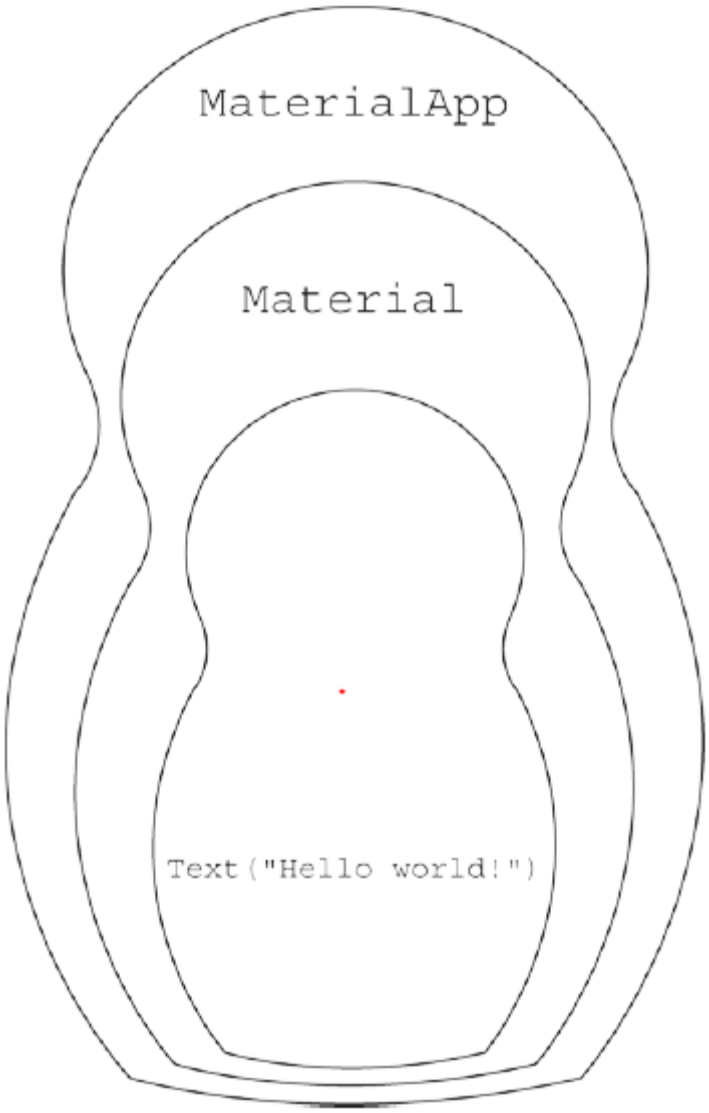
* Z punktu widzenia Fluttera, Text("Witaj świecie!") tworzy widżet.

Jest to instancja klasy Text, a więc (…wina przez asocjację…) instancja klasy StatelessWidget oraz instancja klasy Widget.

Krótki traktat o „wewnętrzności”

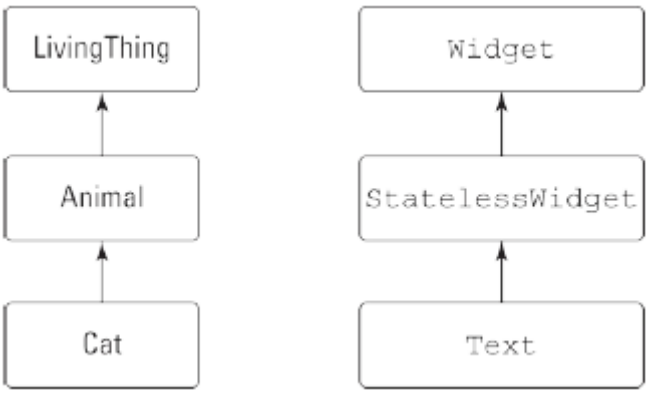
W programie Dart możesz znaleźć widżety w innych widżetach. (Patrz rysunek 3.4.) W tym samym programie Dart znajdziesz -klasy w innych klasach. (Patrz rysunek 3.9.) Te dwa rodzaje „wewnętrzności” nie są tożsame. W rzeczywistości te dwa rodzaje „wewnętrzności” mają ze sobą niewiele wspólnego. Na rysunku 3.3 widżet Tekst jest dzieckiem widżetu Materiał. Nie oznacza to, że instancja Text jest również instancją klasy Material. Aby zrozumieć różnicę, pomyśl o dwóch rodzajach relacji: relacjach „jest” i relacjach „ma”. Relacje, które opisuję w artykule „Co to jest wszystko o?" sekcja to relacje „ma”. Na listingu 3.1 obiekt MaterialApp zawiera w sobie obiekt Material, a obiekt Material zawiera w sobie obiekt Text. Nie ma nic specjalnego w związkach „ma”. Na podwórku mogą istnieć relacje „ma”. Kot ma mysz, a mysz ma kawałek sera. Relacje, które opisałem we wcześniejszej sekcji „Klasy, obiekty i widżety”, to relacje „jest”. W każdym programie Flutter każdy obiekt Text jest obiektem StatelessWidget, a z kolei każdy obiekt StatelessWidget jest obiektem Widget. Na podwórku każdy kot jest zwierzęciem, a z kolei każde zwierzę jest żywą istotą. Nie miałoby sensu twierdzenie, że kot to mysz lub że obiekt materialny jest obiektem tekstowym. W ten sam sposób niepoprawne jest twierdzenie, że każdy kot ma zwierzę lub że każdy obiekt Text ma obiekt StatelessWidget. Te dwa rodzaje relacji — „ma” i „jest” — są zupełnie różne. Jeśli masz ochotę na bardziej formalną terminologię niż „ma” i „jest a”, mam dla ciebie kilka:  Łańcuch rzeczy połączonych relacją „ma” nazywany jest hierarchią kompozycji. Choć może to być frywolne, diagram na rysunku 3.4 ilustruje hierarchię kompozycji.

<img src=”images/flut15.png”>




łańcuch rzeczy połączony relacją „jest” nazywany jest hierarchią dziedziczenia. Diagramy na rysunku są częścią hierarchii klas Fluttera. Czy nie czujesz się lepiej teraz, gdy masz te fantazyjne terminy do rzucania?

<img src=”images/flut16.png”>




We Flutterze prawie wszystko nazywa się „widżetem”. Wiele klas to widżety. Gdy klasa jest widżetem, instancje klasy (dowolne obiekty zbudowane z tej klasy) są również nazywane widżetami.

Dokumentacja jest twoim przyjacielem

Być może zadajesz sobie pytanie, jak zapamiętasz wszystkie te nazwy: Text, StatelessWidget, MaterialApp i prawdopodobnie tysiące innych. Przykro mi to mówić, zadajesz złe pytanie. Nic nie zapamiętujesz. Kiedy używasz imienia wystarczająco często, zapamiętujesz je w sposób naturalny. Jeśli nie pamiętasz nazwy, sprawdzasz ją w dokumentacji Flutter online. (Czasami nie jesteś pewien, gdzie szukać nazwy, której szukasz. W takim przypadku musisz trochę poszperać.) Aby zobaczyć, co mam na myśli, skieruj przeglądarkę internetową na stronę https://api.flutter.dev /flutter/widgets/Textclass.html. Gdy to zrobisz, zobaczysz stronę z informacjami o klasie Tekst, przykładowym kodem i innymi rzeczami.

<img src=”images/flut17.png”>


IGURE 3-10: Useful info about the Text class.

W prawym górnym rogu strony znajduje się lista konstruktorów tekstu. Na rysunku 3.10 są dwie możliwości: tekstowa i bogata. Jeśli wybierzesz łącze Tekst, zostanie wyświetlona strona z opisem wywołania konstruktora tekstu. Ta strona zawiera listę parametrów w wywołaniu konstruktora i zawiera inne przydatne informacje.

<img src=”images/flut18.png”>

# Text constructor

```
const Text(
    String data, {
    Key key,
    TextStyle style,
    StrutStyle strutStyle,
    TextAlign textAlign,
    TextDirection textDirection,
    Locale locale,
    bool softWrap,
    TextOverflow overflow,
    double textScaleFactor,
    int maxLines,
    String semanticsLabel,
    TextWidthBasis textWidthBasis
})
```

Creates a text widget.

If the `style` argument is null, the text will use the style from

The `data` parameter must not be null.

## Implementation

Na stronie na rysunku  zauważ, że wszystkie parametry konstruktora oprócz jednego są ujęte w parę nawiasów klamrowych. Parametr, którego nie ma w nawiasach klamrowych (mianowicie dane typu String) jest jedynym parametrem pozycyjnym konstruktora. Każdy z parametrów w nawiasach klamrowych (w tym double textScaleFactor) jest nazwanym parametrem. Zawsze możesz liczyć na dokumentację Fluttera, która powie ci, jakiego rodzaju obiekty możesz, a jakich nie możesz umieszczać wewnątrz innych obiektów. Na przykład następujący kod jest skazany na niepowodzenie:

```
return MaterialApp(

child: Text("Hello world!"), // Don’t do

this!

);
```

Jest to skazane na niepowodzenie, ponieważ zgodnie z dokumentacją Flutter konstruktor MaterialApp nie ma parametru o nazwie child.

Sprawianie, że rzeczy wyglądają ładniej

Aplikacja pokazana na rysunku 3.2 wygląda dość źle. Słowa Witaj, świecie! są przyciśnięte do lewego górnego rogu ekranu. Na szczęście Flutter oferuje prosty sposób na rozwiązanie tego problemu:

otaczasz widżet Tekst widżetem Centrum. Jak sama nazwa wskazuje, widżet Centrum wyśrodkowuje wszystko, co się w nim znajduje. Słowo Centrum jest nazwą klasy, więc każdy obiekt zbudowany z tej klasy jest nazywany egzemplarz tej klasy. W określeniu takim jak „Widżet centralny” słowo widżet sugeruje, że coś takiego jak Centrum (coś, co pomaga zarządzać układem ekranu) jest pewnego rodzaju komponentem. Komponentem jest fragment tekstu na ekranie, komponentem jest fragment materiału na ekranie, a komponentem jest również obiekt środkowy. Mimo że widżet Centrum nie świeci się gdzieś na stronie ekranu, widżet Center-er nadal jest komponentem. Częścią wielkiej siły Fluttera jest to, że Flutter traktuje wszystkie rzeczy w ten sam sposób. Kiedy tak wiele rzeczy jest widżetami, tak wiele rzeczy może służyć jako parametry w konstruktorach innych rzeczy. Ludzie, którzy wymyślają nazwy funkcji programistycznych, nazywają to funkcją komponowalności, a komponowalność to bardzo przyjemna funkcja. Istnieje kilka sposobów na otoczenie kodu widżetu Tekst kodem widżetu Centrum. Jednym ze sposobów jest umieszczenie kursora gdzieś w edytorze Android Studio, rozpoczęcie pisania i miej nadzieję, że poprawnie poruszasz się po gąszczu nawiasów. Lepszym sposobem jest wykonanie następujących czynności:

1. Umieść kursor na słowie Tekst w edytorze.

2. Naciśnij klawisze Alt+Enter.  W rezultacie pojawi się lista rozwijana.

3. Z listy rozwijanej wybierz Center Widget.

Listing 2 pokazuje, co otrzymujesz.

```
import 'package:flutter/material.dart';

main() => runApp(App0302());

class App0302 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Center(

child: Text("Hello world!"),

),

),

);

}

}
```

Na listingu 2 widżet Material ma element podrzędny widżetu Center, który z kolei ma element podrzędny widget Text. Widżet Tekst można traktować jako następcę widżetu Materiał. Flutter obsługuje ponowne uruchamianie na gorąco. Po dodaniu kodu Centrum do programu w edytorze Android Studio, zapisz zmiany, naciskając Ctrl+S (w Windows) lub Cmd+S (w Mac). Jeśli program z Listingu 1 był już uruchomiony, Flutter zastosuje zmiany i niemal natychmiast zaktualizuje ekran emulatora. W niektórych sytuacjach gorący restart nie działa. Zamiast aktualizować aplikację, Android Studio wyświetla komunikat o błędzie. Jeśli tak się stanie, spróbuj przeładować na gorąco. (Naciśnij

ikonę Uruchom w górnej części głównego okna Android Studio.) A co, jeśli przeładowanie na gorąco się nie powiedzie? W takim przypadku naciśnij ikonę Stop — czerwoną kwadratową ikonę, która znajduje się w tym samym rzędzie co ikona Uruchom. Po naciśnięciu ikony Zatrzymaj działanie aplikacji kończy się całkowicie. Naciśnięcie ikony Uruchom w celu rozpoczęcia od nowa może rozwiązać problem. Rysunek 3.12 pokazuje, co otrzymasz po uruchomieniu kodu z listingu 2.

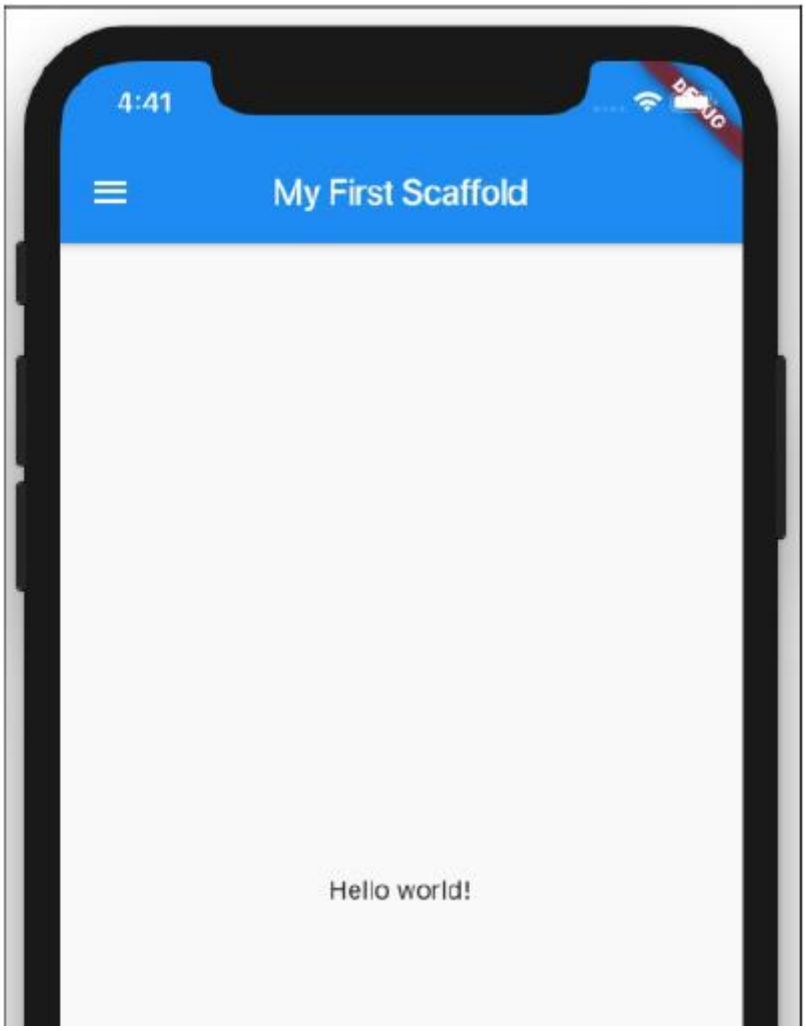
<img src=”images/flut19.png”>


Tworzenie rusztowania

Widżet Tekst na rysunku wygląda na bardzo samotny. Dodajmy trochę fanfar do podstawowej aplikacji. Listing 3 zawiera kod; Rysunki 13 i 14 przedstawiają nowy ekran.

```
import 'package:flutter/material.dart';

main() => runApp(App0303());

class App0303 extends StatelessWidget {

Widget build(BuildContext context) {
```

```
return MaterialApp(
home: Scaffold(
appBar: AppBar(
title: Text("My First Scaffold"),
),
body: Center(
child: Text("Hello world!"),
),
drawer: Drawer(
child: Center(
child: Text("I’m a drawer."),
),
),
),
);
}
}
```



4:41
My First Scaffold
Hello world!


Strona główna aplikacji MaterialApp nie musi być widżetem Material. Na listingu 3.3 domem jest rusztowanie. Kiedy firmy budują drapacze chmur, tworzą rusztowania — tymczasowe drewniane konstrukcje wspierające pracowników na wysokich stanowiskach. W programowaniu rusztowanie jest strukturą, która zapewnia podstawową, często używaną funkcjonalność. — Konstruktor Scaffold z listingu 3.3 ma trzy parametry — pasek aplikacji, treść i szufladę. Na rysunkach 3-13 i 3-14 pasek aplikacji to ciemny obszar u góry ekranu. Ciało - to duży biały region zawierający Centrum z widżetem Tekst. Na rysunku 3.14 szuflada to duży biały obszar, który pojawia się, gdy użytkownik przesuwa palcem od lewej krawędzi ekranu. Szuflada pojawia się również, gdy użytkownik naciśnie ikonę „hamburgera" — trzy poziome kreski w pobliżu lewego górnego rogu ekranu. Ciało to nic specjalnego. Jest bardzo podobny do całego ekranu we wcześniejszych przykładach. Ale appBar i szuflada są nowe. AppBar i szuflada to dwie rzeczy, które możesz mieć podczas tworzenia rusztowania. Inne rzeczy udostępniane przez widżety Scaffold to paski nawigacyjne, pływające przyciski, dolne arkusze, przyciski stopki i inne. Listingi 1 i2 mają widżety Materiał, a Listing 3 — Rusztowanie. Te widżety tworzą tła dla odpowiednich aplikacji. Jeśli usuniesz widżet Materiał z Listingu 1 lub 2, na ekranie Twojej aplikacji pojawi się brzydki bałagan. Otrzymujesz duże czerwone litery z żółtymi podkreśleniami na czarnym tle. To samo stanie się, gdy usuniesz rusztowanie z Listingu 3-3. Istnieją inne widżety, które mogą stanowić tło dla twoich aplikacji, ale najczęściej używane są materiały i rusztowania.

Dodanie poprawek wizualnych

Wypróbuj ten eksperyment: Zmień parametr appBar z Listingu 3 na fragment kodu z Listingu 4.

appBar: AppBar(

title: Text("My First Scaffold"),

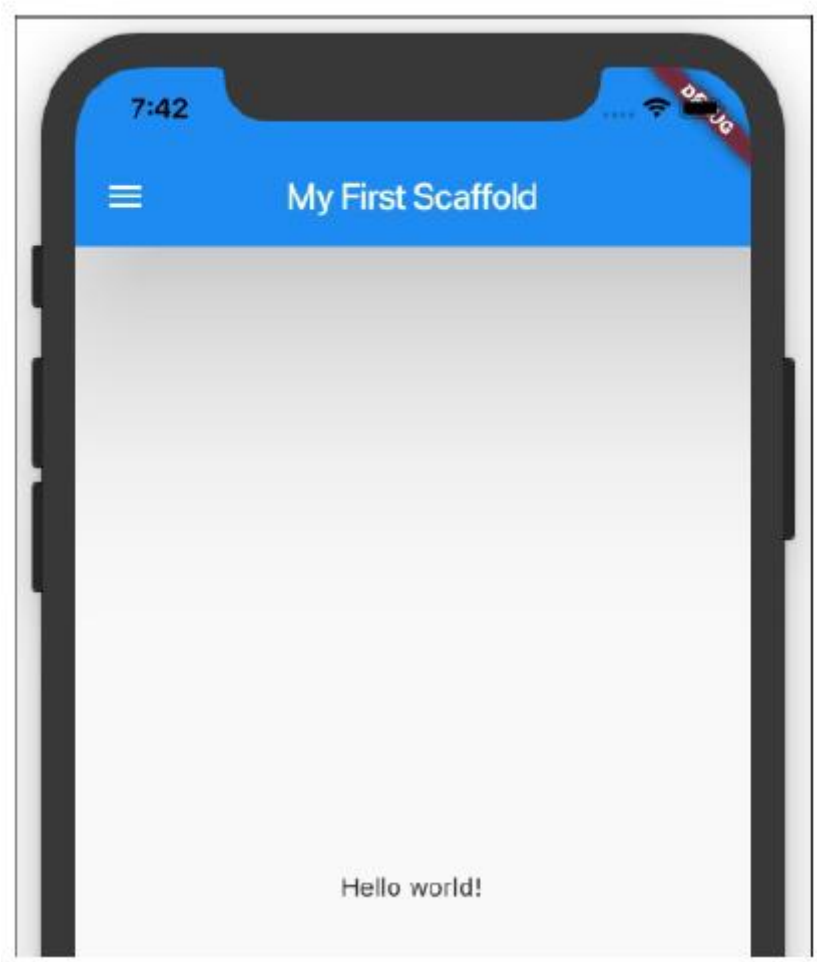
elevation: 100,

brightness: Brightness.light,

)

Na rysunku próbuję pokazać efekt dodania parametrów elewacji i jasności do wywołania konstruktora AppBar. Może mi się nie udać, ponieważ efekt parametru elewacji jest subtelny.

<img src=”images/flut22png”>




W języku Google Material Design wyobrażasz sobie, że tło spoczywa na jakiejś płaskiej powierzchni, a inne komponenty są uniesione nad tło o pewną liczbę pikseli. W przypadku paska aplikacji domyślna wysokość wynosi 4, ale można zmienić wysokość paska za pomocą… poczekaj na to… parametru wysokości. Rzędna komponentu wpływa na kilka aspektów jego wyglądu. Ale w tej sekcji najbardziej oczywistą zmianą jest prawdopodobnie cień pod paskiem aplikacji. Możesz nie być w stanie dostrzec różnicy między cieniami na rysunkach 3-13 i 3-15, ale kiedy uruchamiasz kod na urządzeniu wirtualnym lub fizycznym, pasek AppBar z podniesieniem: 100 rzuca dość duży cień. Być może zastanawiasz się, co oznacza 100 na wysokości: 100. Czy to milimetry, piksele, punkty czy lata świetlne? W rzeczywistości oznacza to „100 pikseli niezależnych od gęstości” — lub w skrócie „100 dps”. Bez względu na to, jaki ekran posiada użytkownik, jeden dp to 1/160 cala. Tak więc wzniesienie: 100 oznacza 100/160 cala

(lepiej znane jako pięć ósmych cala). Parametr jasności widżetu AppBar to jeszcze inna sprawa. Efekt dodania jasności: Brightness.light powie Flutterowi, że ponieważ pasek aplikacji jest jasny, tekst i ikony na górze paska powinny być ciemne.  Ciemny tekst i ikony są dobrze widoczne na tle jasnego paska aplikacji.

Funkcja enum Darta

Interesującą cechę języka programowania Dart kryje Listing 3-4. Słowo Jasność odnosi się do czegoś, co nazywa się enum (wymawiane „ee-noom”). Słowo enum jest skrótem od wyliczenia. Wyliczenie to zbiór wartości, takich jak Brightness.light i Brightness.dark. Zwróć uwagę, jak na listingu 3.4 odwołujesz się do wartości wyliczenia. Nie używasz wywołania konstruktora. Zamiast tego używasz nazwy wyliczenia (np. Jasność), po której następuje kropka i unikatowa część nazwy wartości (np. jasny lub ciemny). Flutter ma wiele innych wbudowanych wyliczeń. Na przykład wyliczenie Orientation ma wartości Orientation.portrait i Orientation.landscape. Wyliczenie AnimationStatus ma wartości AnimationStatus.forward, AnimationStatus.reverse, AnimationStatus.completed i AnimationStatus.dismissed.

Pozdroienia ze słonecznej Kalifornii!

Firma Google ogłosiła Material Design na swojej konferencji dla programistów w 2014 roku. Pierwsza wersja tego języka projektowania dotyczyła głównie urządzeń z Androidem, ale wersja 2 obejmowała niestandardowe branding dla iPhone'ów i innych urządzeń z systemem iOS. Widżet Material firmy Flutter działa na iPhone'ach z automatycznymi adaptacjami specyficznymi dla platformy. Możesz uruchomić dowolny z przykładów MaterialApp z tej książki na iPhonie, a także na telefonach z Androidem, ale jeśli chcesz mieć strategię projektowania iPhonefirst, możesz użyć Flutt

```
import 'package:flutter/cupertino.dart';

void main() => runApp(App0305());

class App0305 extends StatelessWidget {

Widget build(BuildContext context) {

return CupertinoApp(

home: CupertinoPageScaffold(

navigationBar:

CupertinoNavigationBar(),

child: Center(

child: Text("Hello world!"),

),

),

);
```

Listing 5 jest bardzo podobny do swojego kuzyna Material Design, Listing 3. Ale zamiast widżetów MaterialApp, Scaffold i AppBar, Listing 5 zawiera widżety CupertinoApp, CupertinoPageScaffold i CupertinoNavigationBar. Zamiast importować „package:flutter/material.dart”, Listing 5 importuje package:flutter/cupertino.dart”. (Ta deklaracja importu udostępnia bibliotekę widżetów Cupertino

firmy Flutter do wykorzystania przez resztę kodu aukcji). Widżety Material Design i Cupertino firmy Flutter nie są do siebie całkowicie równoległe. Na przykład wywołanie konstruktora Scaffold na listingu 3.3 ma parametr body. Zamiast tego parametru wywołanie konstruktora CupertinoPageScaffold na listingu 3.5 ma parametr potomny. W razie wątpliwości sprawdź oficjalne strony dokumentacji Fluttera, aby dowiedzieć się, które nazwy parametrów należą do wywołań konstruktora poszczególnych widżetów. Możesz mieszać i dopasowywać widżety Material Design i Cupertino w tej samej aplikacji. Możesz nawet dostosować styl projektowania swojej aplikacji do różnych rodzajów telefonów. Możesz nawet umieścić kod następującego rodzaju w swojej aplikacji:

if (Platform.isAndroid) {

// Do Android-specific stuff

} if (

Platform.isIOS) {

// Do iOS-specific stuff

}

Dodanie kolejnego widżetu

Kiedy skończą ci się tematy do rozmowy, możesz zapytać ludzi o ich rodziny. Czasami dowiadujesz się ciekawych faktów, innym razem słyszysz listy skarg, a czasem dostajesz długi, nudny monolog. W ten czy inny sposób wypełnia niezręczną ciszę. Jeśli chodzi o zrozumienie relacji rodzinnych, jestem powolnym uczniem. Ktoś opowiada mi o teściowej żony jego drugiego kuzyna, a ja muszę przerwać rozmowę, żeby narysować diagram mentalny. W przeciwnym razie jestem po prostu zdezorientowany. Moje własne drzewo genealogiczne jest raczej proste. To była mama, tata i ja. Ludzie pytają mnie, czy byłam samotna jako jedynaczka. "Na pewno nie!" Mówię. „Jako jedynak nie musiałem się dzielić”. Ta dyskusja o rodzinach jest moim wątpliwym wstępem do tematu widżetów kolumnowych. W poprzednich przykładach widżet Tekst był jedynakiem. Ale w końcu widżet Tekst musi nauczyć się udostępniać. (W przeciwnym razie widżet Tekst zostanie zepsuty, tak jak ja.) Jak umieścić dwoje dzieci na ciele rusztowania? Możesz ulec pokusie, aby spróbować tego:// DON’T DO THIS:

body: Center(

child: Text("Hello world!"),

child: AnotherWidget(…

)

Ale wywołanie konstruktora nie może mieć dwóch parametrów o tej samej nazwie. Więc co możesz zrobić? Flutter ma widżet Kolumna. Konstruktor widżetu Kolumna ma parametr potomny. Elementy podrzędne widżetu kolumny ustawiają się jeden pod drugim na ekranie. To brzmi obiecująco! Listing 6 zawiera kod, a rysunek 3-16 przedstawia wynikowy ekran.

import 'package:flutter/material.dart';

main() => runApp(App0306());

class App0306 extends StatelessWidget {

Widget build(BuildContext context) {

```
return MaterialApp(
home: Scaffold(
appBar: AppBar(
title: Text("Adding Widgets"),
),
body: Column(
children: [
Text(
"Hello world!",
textScaleFactor: 2.0,
),
Text("It’s lonely for me inside
this phone.")
],
),
),
);
}
}
```

Wywołanie konstruktora Column ma parametr child, a wartością parametru children jest lista. W języku programowania Dart lista to grupa obiektów. Pozycja każdego obiektu na liście jest nazywana indeksem. Wartości indeksu zaczynają się od 0 i idą w górę. Jednym ze sposobów tworzenia listy jest umieszczanie obiektów w nawiasach kwadratowych.

RZECZY STRINGOWE

W języku programowania Dart elementy ujęte w cudzysłowy (jak w przypadku „Witaj świecie!”) nazywane są łańcuchami znaków. To grupa postaci, jedna po drugiej. Oto kilka przydatnych faktów na temat ciągów znaków: Aby utworzyć ciąg, możesz użyć podwójnego lub pojedynczego cudzysłowu. Innymi słowy, „Witaj, świecie!” to to samo co "Witaj świecie!". Łatwo jest umieścić pojedynczy cudzysłów wewnątrz podwójnego cudzysłowu. Zapoznaj się z ciągiem „Czuję się samotny w tym telefonie”. na Listingu 3-6. Łatwo jest umieścić podwójny cudzysłów wewnątrz pojedynczego cudzysłowu. Na przykład następujący prawidłowy ciąg znaków:

'"Tak!" powiedziała.'

Używając znaków ukośnika odwrotnego (\), możesz umieścić dowolny rodzaj cudzysłowu wewnątrz dowolnego rodzaju łańcucha. Oto dwa przykłady:

„Czuję się samotny w tym telefonie”.

"\"Tak!\" powiedziała."

Łańcuch może rozciągać się na kilka wierszy, jeśli użyjesz potrójnych cudzysłowów. Oba te przykłady są prawidłowym kodem Dart:

'''A zwycięzcą jest …

Karol Wan

Doren!'''

"""A zwycięzcą jest …

Karol Wan

Doren!" """

Aby wkleić ciągi jeden po drugim, użyj znaku plus (+) lub spacji. Oba te przykłady są prawidłowym kodem Dart:

„Witaj” + „świat!”

"Witaj świecie!"

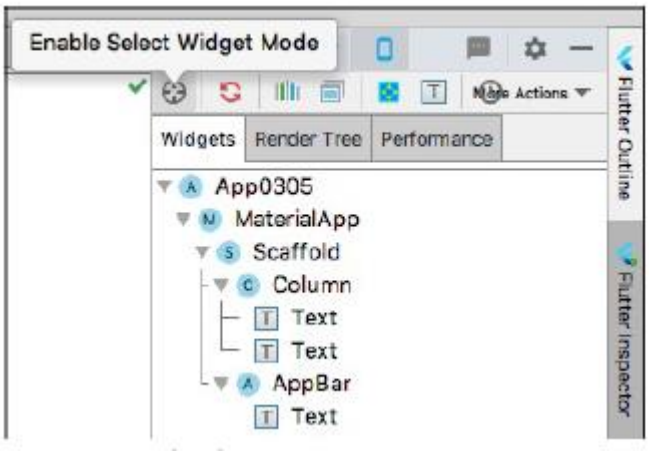
Wyśrodkowanie tekstu (część 1)

Rysunek wygląda dziwnie, ponieważ słowa są przyciśnięte do lewego górnego rogu.

<img src=”images/flut23png”>

W tej sekcji przeprowadzę Cię przez kilka kroków, aby zdiagnozować ten problem i go naprawić. 1. Gdy aplikacja z listingu  działa, poszukaj na prawym brzegu okna Android Studio przycisku paska narzędzi z napisem Flutter Inspector. Kliknij ten przycisk paska narzędzi. W rezultacie pojawia się Flutter Inspector.

<img src=”images/flut24png”>




2. W lewym górnym rogu narzędzia Flutter Inspector znajdź ikonę Włącz tryb wyboru widżetu. Kliknij tę ikonę.

3. Wybierz zakładkę Widżety Flutter Inspector.

4. W drzewie widżetów wybierz Kolumna. W rezultacie urządzenie, na którym działa Twoja aplikacja, dodaje wyróżnienie i niewielką etykietę do widżetu Kolumna na ekranie.

5. Dla zabawy wybierz kilka innych gałęzi w drzewie widżetów Flutter Inspector. Możesz określić granice prawie każdego z nich swoje widżety za pomocą tej techniki.

Podświetlenie na rysunku 3.20 informuje, że widżet Kolumna nie jest wyśrodkowany wewnątrz swojego nadrzędnego widżetu Scaffold i nie jest wystarczająco szeroki, aby wypełnić cały widżet Scaffold. Aby to naprawić, umieść widżet Kolumna wewnątrz widżetu Centrum. Umieść kursor na słowie Kolumna w edytorze Android Studio, a następnie postępuj zgodnie z instrukcjami na początku wcześniejszej sekcji „Sprawianie, że rzeczy wyglądają ładniej”. Listing 3-7 pokazuje, co otrzymujesz.

```
import 'package:flutter/material.dart';

main() => runApp(App0307());

class App0307 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text("Adding Widgets"),
```

```
),
body: Center(
child: Column(
children: [
Text(
"Hello world!",
textScaleFactor: 2.0,
),
Text("It's lonely for me inside
this phone.")
],
),
),
),
);
}
}
```

Kiedy zapiszesz zmiany, Android Studio wykona gorący restart i zobaczysz nowy i ulepszony ekran na rysunku

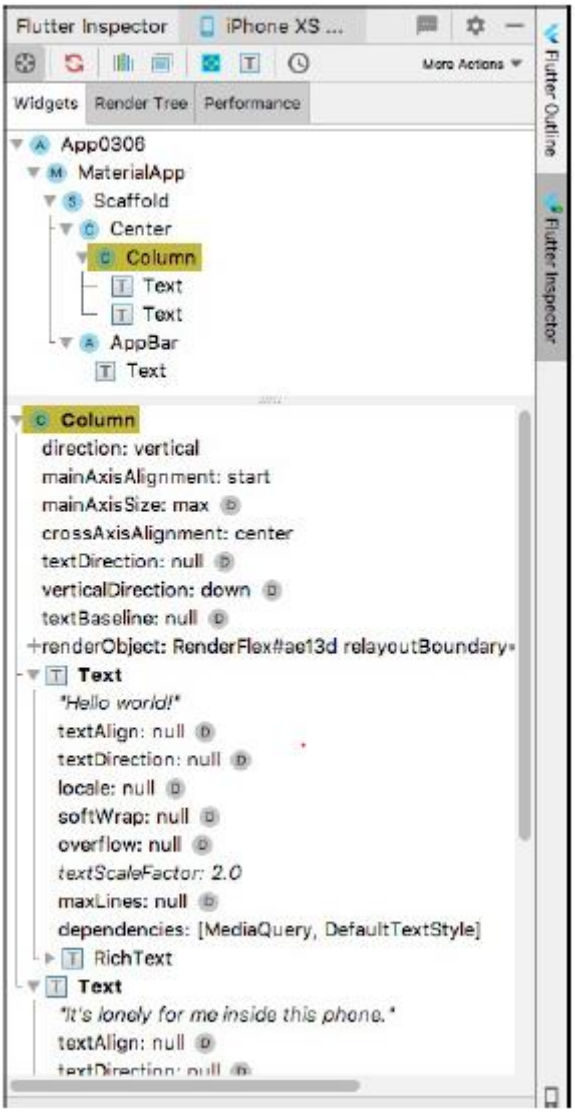
<img src=”images/flut25.png”>


Wyśrodkowanie tekstu (część 2)

Widżety Tekst na rysunku są wyśrodkowane w poziomie, ale nie w pionie. Aby wyśrodkować je w pionie, możesz majstrować przy widżecie Flutter's Center, ale jest o wiele prostszy sposób.

1. W narzędziu Flutter Inspector Android Studio wybierz widżet Kolumna. Dolny panel Flutter Inspector wyświetla wszystkie właściwości dowolnego wybranego widżetu. Czekać! Co to jest „właściwość”? Każdy obiekt ma właściwości, a każda właściwość każdego obiektu ma wartość. Na przykład każda instancja klasy Text Fluttera ma właściwość textScaleFactor. Na listingu wywołanie konstruktora ustawia właściwość textScaleFactor instancji Text na wartość 2,0. Wywołania konstruktora nie są jedynym sposobem ustawiania właściwości obiektów. Na rysunku dolny panel Flutter Inspector pokazuje wartości właściwości kierunku widżetu Column, jego właściwości mainAxisAlignment i wielu innych właściwości. Ponadto dwoje dzieci Text pojawia się w dolnym panelu Flutter Inspector.

<img src=”images/flut26.png”>




2. W dolnym panelu najedź kursorem na właściwość mainAxisAlignment widżetu Kolumna. Gdy to zrobisz, Android Studio wyświetli wyskakujące okienko wyjaśniające znaczenie właściwości mainAxisAlignment. Tekst w tym wyskakującym okienku pochodzi automatycznie z oficjalnej dokumentacji Fluttera. Główną osią słupa jest niewidoczna linia biegnąca od góry do dołu słupa.

3. Ponownie w dolnym panelu najedź kursorem na słowo początek we właściwości mainAxisAlignment widżetu Kolumna. Nowe wyskakujące okienko mówi, że możesz zastąpić start dowolną wartością end, center, spaceBetween, spaceAround lub spaceEvenly.

4. W edytorze Android Studio dodaj parametr mainAxisAlignment do konstruktora widżetu Column.

```
import

'package:flutter/material.dart';

main() => runApp(App0308());

class App0308 extends

StatelessWidget {
```

```
Widget build(BuildContext context)
{
return MaterialApp(
home: Scaffold(
appBar: AppBar(
title: Text("Adding
Widgets"z),
),
body: Center(
child:
Column(mainAxisAlignment:
MainAxisAlignment.center,children: [
Text(
"Hello world!",
textScaleFactor:
2.0,
),
Text("It's lonely for
me inside this phone.")
],
),
),
),
);
}
}import
```

Na listingu mainAxisAlignment to nazwa parametru, MainAxisAlignment to nazwa wyliczenia, a MainAxisAlignment.center to jedna z wartości wyliczenia.

5. Zapisz zmiany w edytorze, aby wykonać gorący restart. Na urządzeniu, na którym działa Twoja aplikacja, widżety tekstowe są wyśrodkowane w poziomie i w pionie.

Przykład w tej sekcji ilustruje aspekty widżetu Kolumna Fluttera, który wyświetla rzeczy od góry do dołu. Nie powinno dziwić, że Flutter ma widżet Row, który wyświetla rzeczy z boku na bok. Większość faktów dotyczących widżetu Kolumna odnosi się również do widżetu Wiersz. (Cóż, są prawdziwe, kiedy leżysz zamiast siedzieć prosto). Ponadto Flutter ma widżet ListView. Widżet ListView wyświetla rzeczy w dowolny sposób — od góry do dołu lub z boku na bok. Ponadto widżet ListView ma własną funkcję przewijania. Możesz umieścić 100 elementów na ListView, mimo że tylko 20 elementów mieści się na ekranie. Gdy użytkownik przewija ekran, elementy przesuwają się poza ekran, podczas gdy inne elementy przesuwają się dalej.

Wyświetlanie obrazu

Słowa są ładne, ale zdjęcia ładniejsze. W tej sekcji umieszczasz obraz na ekranie aplikacji Flutter.

1. W Android Studio rozpocznij nowy projekt Flutter. Nazwałem mój projekt app0308, ale nie musisz używać tej nazwy.

2. W oknie narzędzia projektu Android Studio kliknij prawym przyciskiem myszy nazwę projektu. W rezultacie pojawia się menu kontekstowe.

3. Z menu kontekstowego wybierz Nowy ⇒ Katalog. (Patrz rysunek 3-26.) W rezultacie pojawi się okno dialogowe Nowy katalog. Jak wygodnie!

4. W oknie dialogowym wpisz nazwę zasobów, a następnie naciśnij klawisz Enter. Szczerze mówiąc, możesz nazwać ten nowy katalog prawie tak, jak chcesz. Ale jeśli nie nazwiesz tego aktywa, zdezorientujesz innych programistów Fluttera.

5. Sprawdź okno Project Tool, aby upewnić się, że drzewo projektu ma nową gałąź zasobów . Doświadczeni programiści Flutter tworzą podkatalog images nowego katalogu zasobów. Nie będę się tym teraz przejmować.

6. Znajdź plik obrazu. Wyszukaj plik obrazu na dysku twardym komputera deweloperskiego. Szukaj nazw plików kończących się na .png, .jpg, .jpeg lub .gif. Jeśli Twój Eksplorator plików lub Finder nie wyświetla rozszerzeń nazw plików (takich jak .png, .jpg, .jpeg lub .gif w przypadku plików graficznych)

7. Skopiuj plik obrazu w Eksploratorze plików lub programie +-Finder komputera programistycznego. Oznacza to, że kliknij prawym przyciskiem myszy nazwę pliku obrazu. W wyświetlonym menu kontekstowym wybierz opcję Kopiuj.

8. Korzystając z okna Project Tool w Android Studio, wklej plik obrazu do katalogu asset. Oznacza to, że kliknij prawym przyciskiem myszy gałąź zasobów. W wynikowym menu kontekstowym wybierz Wklej. W wyświetlonym oknie dialogowym wpisz nazwę pliku obrazu, a następnie naciśnij klawisz Enter. Kiedy to wszystko zrobiłem, nazwałem plik MyImage.png, ale nie musisz używać tej nazwy.

9. Otwórz plik pubspec.yaml swojego projektu. Mówiąc dokładniej, kliknij dwukrotnie gałąź pubspec.yaml w drzewie okna Narzędzia projektu. Oto zabawny fakt: rozszerzenie .yaml oznacza jeszcze jeden język znaczników.

10. W pliku pubspec.yaml poszukaj porad dotyczących dodawania zasobów do projektu. Porada może wyglądać mniej więcej tak:

# To add assets to your application,

# add an assets section, like this:

# assets:

# - images/a_dot_burr.jpeg

# - images/a_dot_ham.jpeg

(Jeśli się zastanawiasz, nazwy plików a_dot_burr.jpeg i a_dot_ham.jpeg odnoszą się do Aarona Burra i Alexandra Hamiltona. Te nazwy plików występują wiele razy w oficjalnej dokumentacji Flutter. Flutter to technologia stojąca za aplikacją mobilną dla musicalu Hamilton na Broadwayu .) W pliku .yaml hashtag (#) mówi komputerowi, aby ignorował wszystko w pozostałej części wiersza. Tak więc w tej części pliku .yaml żadna z linii nie ma żadnego efektu.

11. Usuń hashtagi z dwóch linii. W drugim wierszu zmień nazwę pliku obrazu na wybraną w kroku 8. Kiedy to zrobię, mój plik pubspec.yaml zawiera następujący tekst:

# To add assets to your application,

# add an assets section, like this:

assets:

- MyImage.png

# - images/a_dot_ham.jpeg

Używam nazwy MyImage.png zamiast images/MyImage.png, ponieważ w kroku 5 nie utworzyłem katalogu images. Często zapominam wprowadzić niezbędne zmiany w pliku pubspec.yaml. Staraj się nie zapomnieć o tym kroku. Kiedy zapomnisz (a prawie wszyscy to robią), wróć i edytuj plik pubspec.yaml projektu.

12. Zastąp cały kod w pliku main.dart kodem z Listingu 3-9. Użyj własnej nazwy klasy i nazwy pliku zamiast moich nazw App0309 i MyImage.png.

```
import

'package:flutter/material.dart';

main() => runApp(App0309());

class App0309 extends

StatelessWidget {

Widget build(BuildContext context)

{

return MaterialApp(

home: Scaffold(

appBar: AppBar(

title: Text("My First

Image"),

),

body: Center(
```

```
child:
Image.asset('MyImage.png'),
),
),
);
}
}
```

13. Let'er rip.

Oznacza to, że uruchom kod na urządzeniu wirtualnym lub fizycznym.

W tym momencie chcę całkowicie wyjaśnić jedną rzecz: nie jestem narcyzem. Powodem, dla którego używam tego rysunku, jest fascynacja rekurencją. Lubię mieć odniesienie do tej książki w tej książce. (Poza tym jestem trochę narcyzem). Flutter ma klasę Image, a klasa Image ma kilka różnych konstruktorów. Konstruktor Image.asset pobiera plik z miejsca w katalogu projektu Flutter. Aby pobrać obraz z Internetu, wywołujesz inny konstruktor — konstruktor Image.network. Aby uzyskać obraz z dowolnego miejsca na dysku twardym (gdzieś poza katalogiem projektu Flutter), możesz wywołać konstruktor Image.file. Każdy z tych konstruktorów jest nazywany konstruktorem nazwanym. W każdym przypadku elementy po kropce (.asset, .network i .file) to nazwa konkretnego konstruktora.

**Hej, poczekaj chwilę….**

W tym rozdziale omówiono kilka podstawowych idei związanych z tworzeniem aplikacji Dart i Flutter. Zaczynasz od programu Hello World i wprowadzasz w nim kilka zmian. Robiąc to wszystko, budujesz słownik przydatnych pojęć — takich jak klasy, konstruktory, wyliczenia i widżety. Zrobiłeś to wszystko, podczas gdy ja sprytnie odwróciłem twoją uwagę od kilku linijek programu Hello World. Co robią pierwsze cztery wiersze programu Hello World? Dlaczego coś zwracasz, gdy tworzysz MaterialApp? Odpowiedzi na te i podobne pytania znajdują się w następnym rozdziale. Na co czekasz? Czytaj!K<Kbr>

♪ „Szczęśliwe szlaki dla ciebie / Do ponownego spotkania” ♪

NAPISANE PRZEZ DALE EVANS, ŚPIEWANE PRZEZ

ROY ROGERS I DALE EVANS NA

„POKAZ ROYA ROGERSA”, 1944–1957

**Witam ponownie**

Część 3 dotyczy prostego programu Hello world. Dla wygody skopiuję jedną wersję kodu tutaj, na Listingu 4-1. LIST 4-1 Jeszcze jedno spojrzenie na pierwszy program Hello

```
import 'package:flutter/material.dart';

main() => runApp(App0401());

class App0401 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Center(child: Text("Hello

world!")),

),

);

}

}
```

Skoncentrujemy się na środku programu — aplikacji MaterialApp i wszystkim, co się w niej znajduje. Pozwalam ci radośnie ignorować inne części programu. W szczególności pozwalam ci ignorować wszystko, co ma związek z rzeczami zwanymi „funkcjami”. Ten rozdział stanowi kontynuację wycieczki po programie Hello World i umieszcza jego witryny na tych „funkcjonalnych” rzeczach.

**Tworzenie i używanie funkcji**

Oto eksperyment: uruchom aplikację, której kod jest pokazany na listingu 2.

**LISTING 2 Słowa, słowa, słowa**

```
import 'package:flutter/material.dart';

main() => runApp(App0402());

class App0402 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Center(child:

Text(highlight("Look at me"))),

),

);
```

```
}

}

highlight(words) {

return "*** " + words + " ***";

}
```

Rysunek 1 przedstawia dane wyjściowe aplikacji z Listingu 2.


Listing 2 zawiera deklarację funkcji i wywołanie funkcji.

**Deklaracja funkcji**

Pomyśl o przepisie — zbiorze instrukcji dotyczących przygotowania konkretnego posiłku. Deklaracja funkcji jest jak przepis: to zestaw instrukcji do wykonania określonego zadania. Na listingu 4.2 ten zestaw instrukcji brzmi: „Utwórz ciąg zawierający gwiazdki, po których następują słowa, po których następuje więcej gwiazdek, i zwróć gdzieś ten ciąg”. Większość przepisów ma nazwy, takie jak Makaron z Serem lub Potrójne Ciasto Czekoladowe. Funkcja na dole Listingu 4-2 również ma swoją nazwę: Jej nazwa to highlight . Nie ma nic specjalnego w wyróżnieniu nazwy.

Podświetlenie nazwy funkcji znajduje się w części deklaracji zwanej nagłówkiem. Instrukcje funkcji (return "*** " + words + „***”) znajdują się w części deklaracji zwanej body.

Przepis na makaron z serem znajduje się w książce lub na stronie internetowej. Recepta nic nie daje. Jeśli nikt nie korzysta z przepisu, przepis pozostaje uśpiony. To samo dotyczy deklaracji funkcji. Deklaracja z listingu 2 sama w sobie niczego nie robi. Deklaracja po prostu tam jest.

**Wywołanie funkcji**

W końcu ktoś może powiedzieć: „Proszę, zrób makaron z serem na obiad”, a następnie ktoś postępuje zgodnie z instrukcjami przepisu Makaron z serem. Tak czy inaczej, proces zaczyna się, gdy ktoś wypowie (a może tylko pomyśli) nazwę przepisu. Wywołanie funkcji to kod, który mówi: „Proszę wykonać instrukcje określonej deklaracji funkcji”. Wyobraź sobie telefon lub inne urządzenie, na którym działa kod z listingu 2. Gdy telefon napotka funkcję podświetlenia wywołania („Spójrz na mnie”), telefon zostanie odwrócony od swojego podstawowego zadania — zadania tworzenia aplikacji z jej widżetami Materiał, Środek i Tekst. Telefon wykonuje objazd, aby wykonać instrukcje zawarte w treści funkcji podświetlenia. Po ustaleniu, że powinien utworzyć „*** Spójrz na mnie ***”, telefon powraca do swojego podstawowego zadania, dodając widżet Tekst z „*** Spójrz na mnie ***” do widżetu Centrum, dodając Wyśrodkuj widżet do widżetu Materiał i tak dalej. Wywołanie funkcji składa się z nazwy funkcji (takiej jak podświetlenie nazwy na listingu 2), po której następuje informacja z ostatniej chwili (na przykład „Spójrz na mnie” na listingu 2). Czekać! Co w poprzednim zdaniu oznaczają niektóre informacje z ostatniej chwili? Czytaj.

**Parametry i wartość zwracana**

Załóżmy, że twój przepis na makaron z serem służy jednej osobie i wymaga dwóch uncji niegotowanego makaronu łokciowego.

Zaprosiłeś 100 osób na swoje kameralne wieczorne spotkanie. W takim przypadku potrzebujesz 200 uncji niegotowanego makaronu łokciowego. W pewnym sensie przepis mówi: „Aby znaleźć potrzebną liczbę uncji niegotowanego makaronu, pomnóż liczbę porcji przez 2”. Ta liczba porcji to informacja z ostatniej chwili. Osoba, która napisała przepis, nie wie, ilu osobom będziesz służyć. Podajesz liczbę porcji, gdy zaczynasz przygotowywać mac-and-cheese. Cała recepta polega na pomnożeniu tej liczby przez 2. W podobny sposób deklaracja funkcji podświetlania na listingu 4.2 mówi: „Aby znaleźć wartość zwracaną przez tę funkcję, połącz gwiazdki, po których następują słowa, które chcesz otrzymać podświetlony, po którym następuje więcej gwiazdek”. Deklaracja funkcji jest jak czarna skrzynka. Dajesz mu jakieś wartości. Funkcja robi coś z tymi wartościami, aby obliczyć nową wartość. Następnie funkcja zwraca tę nową wartość.

Nadajesz wartości funkcji z listą parametrów funkcji. Jak każde wywołanie konstruktora, każde wywołanie funkcji ma listę parametrów. Każdy parametr dostarcza informację, której ma użyć funkcja. W wywołaniu funkcji highlight("Look at me") przekazuje wartość "Look at me" do deklaracji funkcji podświetlenia. Wewnątrz deklaracji funkcji nazwa słowa oznacza „Spójrz na mnie”, więc wyrażenie "*** " + words + " ***" stands for "*** Look at me ***".

Zwracasz wartość z funkcji za pomocą instrukcji return.

Na listingu 2 wiersz zwraca „*** ” + words + „ ***”; jest zwrotem. Ponownie wyobraź sobie telefon, na którym działa kod z listingu 4.2. Po wykonaniu tej instrukcji return dzieje się tak:

* Telefon przestaje wykonywać dowolny kod w treści funkcji podświetlenia.

* Telefon zastępuje całe wywołanie funkcji zwróconą wartością

Center(child:

Text(highlight("Look at me")))

skutecznie staje się

Center(child: Text("*** Look at

me ***"))

Kontynuuje wykonywanie dowolnego kodu, który wykonywał, zanim został przekierowany przez wywołanie funkcji. Kontynuuje tam, gdzie skończył, tworząc widżety Center, Material i MaterialApp.

Książka kucharska może zawierać tylko jeden przepis na frykasy z kurczaka, ale możesz postępować zgodnie z przepisem tyle razy, ile chcesz. W ten sam sposób określona funkcja ma tylko jedną deklarację, ale aplikacja może zawierać wiele wywołań tej funkcji. Aby zobaczyć to w działaniu, spójrz na Listing 2 i zmień parametr potomny kodu w następujący sposób:

child: Column(mainAxisAlignment:

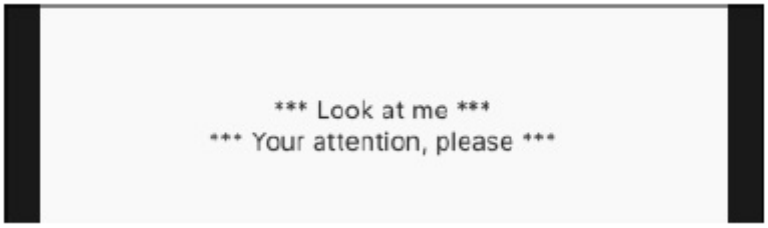
MainAxisAlignment.center, children: [

Text(highlight("Look at me")),

Text(highlight("Your attention, please"))

])

Nowe dziecko zawiera dwa wywołania funkcji podświetlania, każde z własną wartością parametru. Wynikowa aplikacja jest tym, co widzisz na rysunku




**RATOWANIE PLANETY**

W rozdziale 3 radzę kończyć każdą listę parametrów końcowym przecinkiem. W większości przypadków to dobra rada. Ale w przypadku książek drukowanych liczba stron ma duże znaczenie. Aby zachować tę książkę w rozsądnych rozmiarach, zrobiłem to pominięto kilka końcowych przecinków tu i tam. Na przykład fragment kodu z listingu 2 wygląda następująco:

```
home: Material(

child: Center(child:

Text(highlight("Look at me"))),

)
```

Gdy wybierzesz Kod ⇒ Przeformatuj kod, Android Studio sformatuje Twój kod zgodnie z oficjalnymi wytycznymi Darta. (Dart używa narzędzia o nazwie Dartfmt). Kiedy Android Studio formatuje fragment z Listingu 2, fragment ma tylko trzy wiersze. Linia środkowa kończy się trzema nawiasami zamykającymi. Ale zamiast trzech nawiasów pod rząd, mogę oddzielić zamykające nawiasy przecinkami. Kiedy to robię, Android Studio formatuje kod w ten sposób:

```
home: Material(

child: Center(

child: Text(

highlight("Look at me"),

),

),

)
```

Narzędzie Dartfmt interpretuje przecinek jako sygnał rozpoczęcia nowej linii kodu. Podwaja to liczbę wierszy we fragmencie kodu. Czuję się winny, że umieściłem tak wiele linii na tym pasku bocznym! Więc rób tak, jak mówię, a nie tak, jak ja robię. Pamiętaj, że wiele przykładów w tej książce pomija końcowe przecinki. Przykłady działają poprawnie, ale styl kodu jest nie do zniesienia. Dodaj końcowe przecinki, aby zachować zgodność z rygorystycznymi wytycznymi Darta.

**Programowanie w Dart: małe rzeczy**

„Dart jest nudny”. Tak powiedział Faisal Abid podczas prezentacji na DevFest NYC 2017. Nie mówił bzdur o Dart. Wyjaśniał jedynie, że Dart jest bardzo podobny do wielu innych języków programowania. Jeśli napisałeś kilka programów w Javie, C++ lub JavaScript, funkcje Darta są ci całkiem znajome. Napotkasz kilka niespodzianek, ale nie za dużo. Kiedy uczysz się tworzyć aplikacje Flutter, nie chcesz, aby nowy, skomplikowany język programowania stanął Ci na drodze. Tak więc nudny język, taki jak Dart, jest właśnie tym, czego potrzebujesz. Ta sekcja przedstawia kilka nieciekawych faktów na temat języka programowania Dart. Postaraj się nie zasnąć podczas czytania.

**Oświadczenia i deklaracje**

Instrukcja to fragment kodu, który nakazuje Dartowi zrobienie czegoś. Jeśli uważasz, że ta definicja jest niejasna, to w porządku

Teraz. W każdym razie na listingu 2 wiersz

return "*** " + words + " ***";

jest instrukcją, ponieważ nakazuje Dartowi zwrócić wartość z wykonania funkcji podświetlania. W przeciwieństwie do instrukcji, głównym celem deklaracji jest zdefiniowanie czegoś. Na przykład deklaracja funkcji podświetlania na listingu 4.2 określa, co powinno się stać, jeśli i kiedy funkcja podświetlenia zostanie wywołana. Oświadczenia i deklaracje nie są całkowicie od siebie oddzielone. Na listingu 4.2 deklaracja funkcji podświetlania zawiera jedną instrukcję — instrukcję return. Deklaracja funkcji może zawierać kilka instrukcji. Na przykład poniższa deklaracja zawiera trzy instrukcje:

highlight2(words) {

print("Wha' da' ya' know!");

print("You've just called the highlight2

function!");

return "*** " + words + " ***";

}

Pierwsze dwie instrukcje (wywołania funkcji drukowania Darta) wysyłają tekst do okna narzędzia Uruchom Android Studio. Trzecia instrukcja (instrukcja return) sprawia, że highlight("SLook at me") ma wartość "***Look at me***".

Używaj funkcji print Dart tylko do testowania kodu. Usuń wszystkie wywołania drukowania przed opublikowaniem aplikacji. Jeśli tego nie zrobisz, możesz napotkać pewne problemy. W najlepszym przypadku wywołania nie służą żadnemu celowi i mogą spowolnić działanie Twojej aplikacji. W najgorszym przypadku możesz wydrukować poufne dane i pokazać je złośliwym hakerom.

**Funkcja pisania w Dart**

Co oznacza „pięć”? Możesz mieć pięcioro dzieci, ale możesz też mieć pięć stóp wzrostu. Przy piątce dzieci dokładnie wiesz, ile masz dzieci. (W przeciwieństwie do przeciętnej amerykańskiej rodziny, nie możesz mieć 2,5 dzieci.) Ale jeśli masz pięć stóp wzrostu, naprawdę możesz mieć pięć stóp i pół cala wzrostu. Albo możesz mieć cztery stopy jedenaście i trzy czwarte cala wzrostu i nikt nie będzie się o to spierać. Co jeszcze może oznaczać „pięć”? Elektrownie jądrowe mogą zostać poddane ocenie podatności na pożar wywołanej pożarem, znanej również jako pięć. W tym przypadku „pięć” nie ma

nic wspólnego z liczbą. To tylko f-iv-e. Znaczenie wartości zależy od typu wartości. Rozważ trzy wbudowane typy języka Dart: int, double i String.

* Int to liczba całkowita, bez cyfr po prawej stronie przecinka dziesiętnego. Jeśli piszesz

int ileDzieci = 5;

w programie Dart 5 oznacza „dokładnie pięć”.

* Double to liczba ułamkowa z cyframi po prawej stronie przecinka dziesiętnego. Jeśli piszesz

podwójna wysokość = 5;

w programie Dart 5 oznacza „tak blisko pięciu, jak chcesz zmierzyć”.

* Stringto zbiór znaków. Jeśli używasz pojedynczych cudzysłowów (lub podwójnych cudzysłowów) i piszesz

Naciśnięcie klawisza ciągu = „5”;

w programie Dart „5” oznacza „znak, który wygląda jak wielka litera S, ale którego górna połowa ma spiczaste zakręty”.

Typ wartości określa, co możesz z nią zrobić. Rozważ wartości 86 i „86”.

Pierwsza, 86, to liczba. Możesz dodać do niego kolejny numer.

86 + 1 to 87

Drugi, „86”, to ciąg znaków. Nie możesz dodać do niego liczby, ale możesz dodać do niego kolejny ciąg.

„86” + „1” to „861”

W niektórych językach można połączyć dowolną wartość z inną wartością i uzyskać jakiś wynik. Nie możesz tego zrobić w Dart. Język programowania Dart jest bezpieczny pod względem typów.

Wartość zmiennej nie jest taka sama w każdym programie Dart. W rzeczywistości wartość zmiennej może nie być taka sama w różnych częściach programu Dart. Weźmy na przykład następujący wiersz kodu:

int howManyChildren = 5;

Ta linia jest nazywana deklaracją zmiennej. Linia definiuje zmienną o nazwie howManyChildren, której typ to int. Wiersz inicjuje tę zmienną wartością 5. Kiedy Dart napotka ten wiersz, howManyChildren oznacza liczbę 5. Później, w tym samym programie, Dart może wykonać następujący wiersz:

howManyChildren = 6;

Ta linia jest nazywana instrukcją przypisania. Linia sprawia, że howManyChildren odnosi się do 6 zamiast 5. Gratulujemy narodzin nowego dziecka! Czy to dziewczyna czy chłopak? Wyrażenie jest częścią programu Dart, która oznacza wartość. Wyobraź sobie, że Twój kod zawiera następujące deklaracje zmiennych:

int numberOfApples = 7;

int numberOfOranges = 10;

Jeśli zaczniesz od tych dwóch deklaracji, każdy wpis w lewej kolumnie tabeli 4-1 jest wyrażeniem.

Język Dart ma instrukcje i wyrażenia. Oświadczenie to polecenie zrobienia czegoś; wyrażenie to kod, który ma wartość. Na przykład instrukcja print("Cześć"); coś robi. (Wyświetla Hello w oknie narzędzia Uruchom Android Studio.) Wyrażenie 3 + 7 * 21 ma wartość. (Jego wartość to 150.)

Możesz zastosować toString Darta do dowolnego wyrażenia

Dart zapewnia szybki sposób określenia typu określonego wyrażenia. Aby to zobaczyć, zmień deklarację funkcji podświetlania na listingu 2 w następujący sposób:

```
highlight(words) {

print(20 / 7);

print((20 / 7).runtimeType);

return "*** " + words + " ***";

}
```

Po uruchomieniu aplikacji w oknie narzędzia Uruchom Android Studio pojawiają się następujące wiersze:

```
flutter: 2.857142857142857

flutter: double
```

Wartość 20/7 to 2,857142857142857, a wartość (20/7).runtimeType jest podwójna.

**Dwa w cenie jednego**

W Dart niektóre instrukcje pełnią podwójną rolę zarówno jako instrukcje, jak i wyrażenia. W ramach eksperymentu zmień funkcję podświetlania na listingu 4.2, aby wyglądała tak:

```
highlight(words) {

int numberOfKazoos;

print(numberOfKazoos);

print(numberOfKazoos = 94);

return "*** " + words + " ***";

}
```

Android Studio wyświetla ostrzeżenie, że zmienna numberOfKazoos nie jest używana, ale to w porządku. To tylko eksperyment. Oto, co widzisz w oknie narzędzia Uruchom Android Studio po uruchomieniu tego kodu:

```
flutter: null

flutter: 94
```

Linia int numberOfKazoos; jest deklaracją zmiennej bez inicjalizacji. To uczciwa gra w języku programowania Dart. Kiedy Dart wykonuje print(numberOfKazoos); zobaczysz flutter: null w oknie narzędzia Uruchom. Z grubsza mówiąc, null oznacza „nic”. W tym momencie programu zmienna numberOfKazoos została zadeklarowana, ale nie nadano jej jeszcze wartości, więc numberOfKazoos nadal ma wartość null. Wreszcie, gdy Dart wykonuje print(numberOfKazoos = 94); zobaczysz

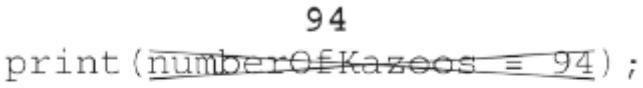
trzepotanie: 94 w oknie narzędzia Uruchom. Aha! Kod numberOfKazoos = 94 jest zarówno stwierdzeniem, jak i wyrażeniem!

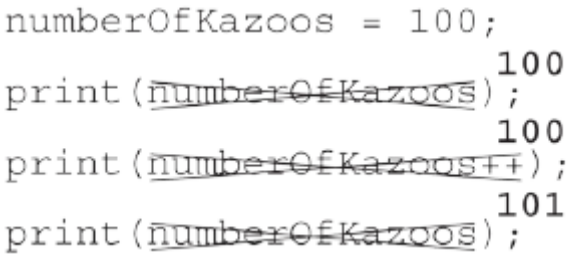
Dlatego:

* Jako stwierdzenie, numberOfKazoos = 94 powoduje, że wartość numberOfKazoos wynosi 94.

* Jako wyrażenie, wartość numberOfKazoos = 94 wynosi 94.

Z tych dwóch faktów drugi jest trudniejszy do strawienia dla ludzi. (Znałem kilku doświadczonych programistów, którzy myśleli o tym w niewłaściwy sposób.) Aby wykonać print(numberOfKazoos = 94); Dart potajemnie zastępuje 94 wyrażeniem numberOfKazoos = 94, jak pokazano na rysunku 7.




Innymi słowy, wartość numberOfKazoos = 94 to 94. Tak więc, oprócz robienia czegoś, kod numberOfKazoos = 94 ma również wartość. Dlatego liczbaOfKazoos = 94 jest zarówno instrukcją, jak i wyrażeniem. Proste instrukcje przypisania nie są jedynymi rzeczami, które podwajają się jako wyrażenia. Wypróbuj ten kod dla rozmiaru:

numberOfKazoos = 100;

print(numberOfKazoos);

print(numberOfKazoos++);

print(numberOfKazoos);

The code’s output is

flutter: 100

flutter: 100

flutter: 101

Jeśli środkowa linia wydruku Cię zaskakuje, nie jesteś sam. Jako stwierdzenie, numberOfKazoos++ dodaje 1 do wartości numberOfKazoos, zmieniając wartość ze 100 na 101. Ale, jak

wyrażenie, wartość numberOfKazoos++ wynosi 100, a nie 101.

Oto pocieszająca myśl. Zanim Dart wykona ostatnią instrukcję print(numberOfKazoos), wartość

numberOfKazoos już się zmieniło na 101. Uff!

Jako stwierdzenie, ++numberOfKazoos (ze znakiem plus z przodu) robi to samo co numberOfKazoos++: dodaje 1 do wartości numberOfKazoos. Ale jako wyrażenie wartość ++numberOfKazoos nie jest taka sama jak wartość numberOfKazoos++. Spróbuj. Zobaczysz.

Dart ma kilka innych instrukcji, których wartościami są wyrażenia.

Na przykład poniższy kod drukuje flutter: 15 dwukrotnie:

```
int howManyGiraffes = 10;

print(howManyGiraffes += 5);

print(howManyGiraffes);
```

A następujący kod drukuje flutter: 5000 dwa razy:

```
int rabbitCount = 500;

print(rabbitCount *= 10);

print(rabbitCount);
```

**Słowo kluczowe var Darta**

Czasami możesz chcieć utworzyć zmienną, której typ można zmienić. Aby to zrobić, zadeklaruj zmienną za pomocą słowa kluczowego var Darta i pomiń inicjalizację w deklaracji. Na przykład następujący kod nie zadziała:

```
int x = 7;

print(x);

x = "Someone's trying to turn me into a

String"; // You can’t do this

print(x);
```

Ale poniższy kod działa dobrze:

```
var x;

x = 7;

print(x);

x = "I've been turned into a String"; // Dart

is happy to oblige

print(x);
```

Innym powodem używania var jest unikanie długich, skomplikowanych nazw typów.

**PRZERWAMY NA KILKA KOMENTARZY**

Być może zauważyłeś, że w niektórych przykładach kodu z tego rozdziału coś zaczyna się od dwóch ukośników (//). Dwa ukośniki oznaczają początek komentarza. Komentarz jest częścią tekstu programu. Jednak w przeciwieństwie do deklaracji, wywołań konstruktora i innych tego typu elementów celem

komentarza jest pomoc ludziom w zrozumieniu kodu. Komentarz jest częścią dokumentacji dobrego programu. Język programowania Dart ma trzy rodzaje komentarzy:

**Komentarze na koniec linii**

Komentarz końca wiersza rozpoczyna się dwoma ukośnikami i kończy na końcu wiersza tekstu. Tak więc w poniższym fragmencie kodu tekst // Dart chętnie się zobowiąże jest komentarzem na końcu wiersza:

x = "Zostałem zamieniony w String";

// Dart chętnie się zobowiąże

Cały tekst w komentarzu na końcu wiersza jest przeznaczony wyłącznie dla ludzkiego oka. Żadna informacja od dwóch ukośników do końca wiersza nie jest tłumaczona przez kompilator Darta.

**Blokuj komentarze**

Komentarz blokowy zaczyna się od /* i kończy na */. Komentarz blokowy może obejmować kilka wierszy. Na przykład następujący kod jest komentarzem blokowym:

/* Tymczasowe komentowanie tego kodu.

Oznacza to, że pomijając te stwierdzenia, aby zobaczyć, co się stanie:

x = „Ktoś próbuje mnie przemienić w łańcuch";

print(x); */

Po raz kolejny żadne informacje między /* a */ nie są tłumaczone przez kompilator.

**Komentarze dokumentu**

Komentarz dokumentu na końcu wiersza rozpoczyna się trzema ukośnikami (///). Blokowy komentarz do dokumentu zaczyna się od /** i kończy na */.

Komentarz do dokumentu powinien być czytany przez ludzi, którzy nigdy nawet nie spojrzą na kod Darta. Ale to nie ma sensu. Jak możesz zobaczyć komentarz do dokumentu, jeśli nigdy nie patrzysz na kod? Cóż, pewien program o nazwie dartdoc (co jeszcze?) może znaleźć dowolne komentarze do dokumentów w programie i przekształcić te komentarze w ładnie wyglądającą stronę internetową. Co jest lepsze — komentarze do dokumentów na końcu wiersza czy komentarze do dokumentów blokowych? Profesjonalni programiści Darta przedkładają komentarze do dokumentów na końcu wiersza nad komentarze do dokumentów blokowych. Wyśmiewają komentarze do dokumentów blokowych i pukają komentarze do dokumentów blokowych. Nie umieszczają żadnych akcji w komentarzach do dokumentów blokowych. Nie blokują komentarzy do dokumentów. Ich zdaniem komentarz do dokumentu na końcu wiersza jest świetny, ale cały pomysł z komentarzem do dokumentu blokowego to bzdura. Jeszcze jedna myśl na temat komentarzy w ogóle: w rozdziale 3 opisuję sposób wyświetlania etykiet zamykających w edytorze Android Studio.

home: Material(

child: Text("Hello world!"),

), // Material

Czy ten końcowy // Material wygląda dla ciebie jak komentarz? Cóż, to właściwie nie jest komentarz. (Przepraszam.) Etykiety zamykające należą do szerszej kategorii przedmiotów zwanej dekoracją kodu. Kiedy Android Studio tworzy dekorację kodu, nie dodaje dekoracji do tekstu programu. Wyświetla tylko tę dekorację w edytorze. Jeśli sprawdzisz tekst programu za pomocą Notatnika lub TextEdit, nie zobaczysz dekoracji kodu.

**Typy wbudowane**

W programie Dart każda wartość ma swój typ. Dart ma dziesięć wbudowanych typów.

**Typy, które nie są wbudowane**

Oprócz typów z tabeli 2 każda klasa jest typem. Na przykład na listingu 4.1 App0401 jest nazwą typu. Jest to typ zdefiniowany na listingu 4.1. Możesz dodać linię do Listingu 4-1, która sprawia, że zmienna odwołuje się do instancji klasy App0401. Oto jeden z takich wierszy:

Aplikacja0401 mojaAplikacja = Aplikacja0401();

Podobnie jak wiele innych deklaracji zmiennych, ten wiersz ma nazwę typu (App0401), po której następuje nowa nazwa zmiennej (myApp), po której następuje inicjalizacja. Inicjalizacja sprawia, że myApp odwołuje się do nowo skonstruowanej instancji App0401. Język Dart jest dostarczany z biblioteką pełną standardowego kodu wielokrotnego użytku. Formalna nazwa takiej biblioteki to interfejs programowania aplikacji (API). API Darta ma deklaracje wielu klas. Na przykład instancje klasy DateTime programu Dart to momenty w czasie, a instancje klasy Duration to interwały czasowe. Podobnie zestaw narzędzi Flutter zawiera bogate w funkcje API. Na listingu 4.1 Widget, StatelessWidget, BuildContext, MaterialApp, Material, Center i Text to nazwy klas w interfejsie API Flutter.

**Korzystanie z deklaracji importowych**

Biada mi! Nie mogę przeczytać książki Flutter For Dummies, chyba że pójdę do lokalnej biblioteki i sprawdzę kopię. To samo dotyczy klas bibliotecznych Darta i Fluttera (no, prawie). Nie możesz używać klas MaterialApp lub Material Fluttera, chyba że zaczniesz swój program od

import 'package:flutter/material.dart';

Jeśli usuniesz tę linię, nie będziesz mógł nawet używać żadnej z klas widgetów Fluttera (StatelessWidget, Widget, Center i Text, by wymienić tylko kilka). To dlatego, że kiedy importujesz „package:flutter/material.dart”, automatycznie importujesz również „package:flutter/widgets.dart”. Stosunkowo niewielka liczba klas API Darta, jak wspomniana wcześniej klasa DateTime, należy do pakietu o nazwie dart.core. Możesz uruchomić swój program za pomocą wiersza

import „dart:core”;

ale nic ci to nie da. Zajęcia z pakietu dart.core są zawsze importowane, niezależnie od tego, czy o to poprosisz, czy nie.

Nikt (i mam na myśli nikogo) nie zapamiętuje nazw wszystkich klas w bibliotekach Dart czy Flutter.

**Wariacje na temat z Die Flutter Mouse**

W tej sekcji przedstawiono kilka alternatywnych sposobów tworzenia deklaracji funkcji. Listing 4.3 zawiera pierwszy przykład.

**LISTING 3 Pomieszanie z deklaracjami funkcji**

```
import 'package:flutter/material.dart';

main() {

runApp(App0403());

}

class App0403 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Center(child:

Text(highlight("Look at me"))),

),

);

}

}

highlight(words) => "*** $words ***";
```

BLING SWÓJ STRING

Listing 4-2 zawiera następujący kod:

```
"*** " + words+ " ***"
```

Zestawienie znaków plus i cudzysłowów może utrudniać odczytanie kodu. Aby ułatwić Ci życie, Dart ma interpolację strun. W przypadku interpolacji łańcucha znak dolara ($) oznacza „Tymczasowo zignoruj otaczające cudzysłowy i znajdź wartość następującej zmiennej”. Dlatego na listingu 3 wyrażenie „*** $words ***” oznacza „*** Spójrz na mnie ***” — ten sam ciąg znaków, co na listingu 2. Nie jesteś pod wrażeniem interpolacji łańcucha? Przyjrzyj się poniższej funkcji i zastanów się, co o niej myślisz:

```
// The function call

getInstructions1(8, "+", ";", "'")

// The function’s declaration

getInstructions1(howMany, char1, char2,

char3) {

return "Password: " +

howMany.toString() +

" characters; Don't use " +

char1 +
```

" " +

char2 +

" or " +

char3;

}

Niezły bałagan, prawda? Wartość zwracana przez funkcję getInstructions1 to Hasło: 8 znaków; nie używaj +; lub ' Ilekroć próbuję napisać taki kod, zapominam o spacjach, cudzysłowach lub innych elementach. Oto jak uzyskać tę samą wartość zwracaną za pomocą interpolacji łańcucha:

// The function call

Text(getInstructions2(8, "+", ";", "'")

// The function’s declaration

getInstructions2(howMany, char1, char2,

char3) {

return "Password: $howMany characters;

Don't use $char1 $char2 or $char3";

}

Ta nowa funkcja, getInstructions2, jest łatwiejsza do utworzenia i łatwiejsza do zrozumienia niż funkcja getInstructions1. Korzystając z interpolacji łańcucha, możesz pójść o krok dalej. Oto, co możesz zrobić, dodając nawiasy klamrowe do miksu:

// The function call

getInstructions3(8, "+", ";", "'")

// The function’s declaration

getInstructions3(howMany, char1, char2,

char3) {

return "Password: ${howMany + 1}

characters; Don't use $char1 $char2 or

$char3";

}

Ta nowa funkcja getInstructions3 zwraca hasło: 9 znaków; nie używaj +; lub ' Interpolacja ciągów może obsłużyć wszystkie rodzaje wyrażeń — wyrażenia arytmetyczne, wyrażenia logiczne i inne.

Przebieg kodu z Listingu 3 jest taki sam jak z Listingu 2. (Patrz rysunek 1.) W pewnym sensie Listing 3 zawiera ten sam program, co Listing 2. Notacja dla rzeczy jest nieco inna, ale same rzeczy są takie same. Na listingu 3 deklaracja funkcji podświetlenia

podświetl(słowa) => "*** $words ***";

jest skrótem bardziej rozbudowanej deklaracji podświetlenia z listingu 2. Gdy treść deklaracji funkcji zawiera tylko jedną instrukcję, możesz użyć tej szybkiej i łatwej notacji z grubą strzałką (=>).

Wracając do listingu 2, do zadeklarowania funkcji main używam notacji grubej strzałki. Aby pokazać, że potrafię to zrobić, „odtłuszczam” tę deklarację na Listingu 3. Każdy program Dart ma funkcję o nazwie main. Kiedy zaczynasz uruchamiać program, Dart szuka deklaracji głównej funkcji programu i rozpoczyna wykonywanie wszelkich instrukcji znajdujących się w treści deklaracji. W aplikacji Flutter stwierdzenie takie jak

```
runApp(App0403());
```

mówi Dartowi, aby skonstruował instancję App0403, a następnie uruchomił tę instancję. Funkcja runApp jest częścią API Fluttera.

Wpisz nazwy w deklaracjach funkcji

Listing 4 dodaje kilka nazw typów do kodu z Listingu 2.

**LISTING 4 Lepiej być bezpiecznym niż żałować**

```
import 'package:flutter/material.dart';

void main() => runApp(App0404());

class App0404 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

child: Center(child:

Text(highlight("Look at me"))),

),

);

}

}

String highlight(String words) {

return "*** $words ***";

}
```

Na listingu 4, String i void dodają do kodu pożądaną redundancję. Wystąpienie String w (String Words) mówi Dartowi, że w każdym wywołaniu funkcji podświetlania parametr Words musi mieć typ String. Uzbrojony w te dodatkowe informacje o łańcuchu, Dart kaszle i wypluwa złe wywołanie funkcji, takie jak highlight(19) To jest złe, ponieważ 19 to liczba, a nie ciąg. Możesz się kłócić i powiedzieć: „Nigdy nie popełnię błędu i nie wstawię numeru w wywołaniu funkcji podświetlenia”. A moja odpowiedź brzmi: „Tak, zrobisz to i ja, i każdy inny programista na ziemi”. Podczas pisania kodu błędy są nieuniknione. Sztuką jest złapać je raczej wcześniej niż później. Pod koniec Listingu 4 podświetlenie String mówi o tym Dartowi

wartość zwrócona przez funkcję podświetlenia musi być łańcuchem. Jeśli przypadkowo napiszesz następujący kod, Dart będzie narzekał jak nikt inny:

String highlight(String words) {

return 99; //Bad code!

}

Przepraszam szefie. Wartość 99 nie jest ciągiem znaków.

Kontynuując naszą podróż po listingu 4, stwierdzenie void main nie oznacza, że „funkcja main musi zwracać wartość typu void”. Dlaczego nie? Można umieścić nazwę typu przed deklaracją grubej strzałki. Czym więc różni się void main? Mówiąc najprościej, void nie jest typem. W pewnym sensie void oznacza „brak typu”. Słowo void przypomina Dartowi, że ta główna funkcja nie powinna zwracać niczego użytecznego. Spróbuj zadeklarować void main i umieścić instrukcję return w treści deklaracji:

void main() {

runApp(App0404());

return 0; // Bad

}

Jeśli to zrobisz, edytor Android Studio doda czerwone znaki do twojego kodu. Dart mówi: „Przepraszam, Bud. Nie możesz tego zrobić.

**Nazywanie parametrów**

Rozdział 3 rozróżnia parametry pozycyjne konstruktorów i parametry nazwane. Całe zamieszanie związane z rodzajami parametrów dotyczy również funkcji. Na przykład funkcja podświetlania na listingu 4.4 ma jeden parametr — parametr pozycyjny.

highlight("Spójrz na mnie") // Wywołanie funkcji

Wyróżnienie ciągu znaków (słowa ciągu) { // The

deklaracja funkcji

zwróć "*** $słowa ***";

}

Jeśli chcesz, możesz zamienić słowa w nazwany parametr. Wystarczy otoczyć parametr nawiasami klamrowymi:

podświetl (słowa: „Spójrz na mnie”) // A

wywołanie funkcji

Wyróżnienie ciągu ({Słowa ciągu}) { // The

deklaracja funkcji

zwróć "*** " + słowa + " ***";

}

Możesz nawet mieć funkcję z parametrami pozycyjnymi i nazwanymi. Na liście parametrów wszystkie parametry pozycyjne muszą znajdować się przed dowolnym z nazwanych parametrów. Na przykład poniższy kod wyświetla +++Spójrz na mnie!+++.

zaznacz ( // A

wywołanie funkcji

"Spójrz na mnie",

interpunkcja: „!”,

symbole: „+++”,

)

Wyróżnienie ciągu ( // The

deklaracja funkcji

słowa ciągów, {

interpunkcja ciągów znaków,

symbole ciągów,

}) {

zwróć symbole + słowa + znaki interpunkcyjne +

symbolika;

}

A co z funkcją budowania?

Listing 4 zawiera znajomo wyglądający kod:

klasa App0404 rozszerza StatelessWidget {

Kompilacja widżetu (kontekst BuildContext) {

zwróć aplikację materiału (

Oto kilka faktów:

* W tym kodzie build to nazwa funkcji, a build widżetu (kontekst BuildContext) to nagłówek deklaracji funkcji. Funkcja kompilacji robi dokładnie to, co sugeruje jej nazwa. To coś buduje. Mówiąc ściślej, buduje widżet, którego treścią jest cała aplikacja Flutter.

* Funkcja kompilacji zwraca wartość typu Widget. Cytując z rozdziału 3: „Bycie przykładem jednej klasy może sprawić, że automatycznie staniesz się przykładem większej klasy”. W rzeczywistości każda instancja klasy MaterialApp jest automatycznie instancją klasy StatefulWidget, która z kolei jest automatycznie instancją klasy Widget. A więc masz to — każda aplikacja MaterialApp jest widżetem. Dlatego instrukcja return funkcji kompilacji może zwracać obiekt MaterialApp.

* Jedyny parametr funkcji ma typ BuildContext.

Kiedy Dart buduje widżet, Dart tworzy obiekt BuildContext i przekazuje go do funkcji budowania widżetu. Obiekt BuildContext zawiera informacje o widżecie i relacji widżetu z innymi widżetami w programie.

Na listingu 4 deklaracja funkcji budowania znajduje się w definicji klasy App0404, ale deklaracja funkcji podświetlania nie znajduje się w definicji żadnej klasy. W pewnym sensie ta funkcja budowania „należy” do instancji klasy App0404. Funkcja należąca do klasy lub instancji klasy ma specjalną nazwę. To się nazywa metoda.

**Sprawianie, że rzeczy się dzieją**

Jest 20 października 1952 roku. W Kenii brytyjski gubernator kolonialny ogłasza stan wyjątkowy. W Filadelfii rodzi się aktorka Melanie Mayron (wnuczka Frances Goodman). W Stanach Zjednoczonych odcinek „Kocham Lucy” staje się pierwszym odcinkiem telewizyjnym, który został wyemitowany więcej niż raz. Co? "Kocham Lucy"? Tak, „Kocham Lucy”. Do tego dnia powtórki telewizyjne (znane również jako „powtórki”) nie istniały. Wszystko w telewizji było zupełnie nowe. Od tego czasu powtarzające się emisje programów telewizyjnych stały się normą. Tak wiele treści telewizyjnych to powtórka ze starego wideo, że nadawcy nie reklamują już „nowego odcinka”. Zamiast tego ogłaszają emisję „zupełnie nowego odcinka”. Słowo nowy już nie wystarcza. Zwykłe produkty gospodarstwa domowego nie są nowe; są „nowe i ulepszone”. Oczywiście reklamowanie rzeczy jako „nowych”, „najlepszych” lub „najnowszych” może przynieść odwrotny skutek. W rzeczywistości wszelkiego rodzaju hype może przynieść odwrotny skutek. Rozważmy przypadek kremu do golenia Swell firmy Stanley. W 1954 roku Stanley’s był liderem rynku. Rok później, kiedy sprzedaż spadała, reklamodawcy zmienili jego nazwę na Neat New Shaving Cream firmy Stanley. Rok później stał się on doskonałym kremem do golenia firmy Stanley. Sprzedaż produktu była dobra przez kilka następnych lat. Ale na początku lat 60. sprzedaż spadła, a reklamodawcy Stanleya byli w kropce. Co może być lepszego niż „Doskonały krem do golenia”? Lepszy niż najlepszy krem do golenia? Po kilku długich spotkaniach geniusz w dziale marketingu wymyślił Sensational Shocking Pink Shaving Cream firmy Stanley — jaskrawą mieszankę mydła, gliceryny, środków zmiękczających skórę, czerwonego barwnika numer 2 i prawdopodobnie trochę wolnoschnącego kleju. To był koniec kolejki. Pomysł golenia różowym kremem nie spodobał się konsumentom, a firma Stanleya zbankrutowała. Konsumenci mówili o oślizgłym mydle Stanleya, rubinowym śmietniku Stanleya i, co najgorsze, o obrzydliwym łajnie Stanleya. Możesz zapytać: „Co u licha ma wspólnego krem do golenia Stanleya z tworzeniem aplikacji Flutter?” Chodzi mi o to, że przereklamowanie produktu wiąże się z niebezpieczeństwem, a przereklamowanie koncepcji tworzenia aplikacji wcale nie jest lepsze. W rozdziałach 3 i 4 używam żarliwych terminów do opisania strategii programowania Fluttera, jego konstruktorów, funkcji i innych dobrych rzeczy. Ale tutaj, w rozdziale 5, rzucam oszczerstwa na te wprowadzające przykłady, ponieważ żaden z nich nie pozwala użytkownikowi zmienić czegokolwiek na ekranie. Aplikacja, która zawsze wyświetla ten sam stary tekst, jest nudna, a użytkownicy ocenią ją bez gwiazdek. Ciekawa aplikacja wchodzi w interakcję z użytkownikiem. Ekran aplikacji zmienia się, gdy użytkownik wpisuje tekst, stuka przycisk, przesuwa suwak lub robi coś innego, aby uzyskać użyteczną odpowiedź z aplikacji. Sprawienie, by coś się działo, jest niezbędne do każdego rodzaju tworzenia aplikacji mobilnych. Tak więc w tym rozdziale zaczniesz uczyć się, jak sprawić, by coś się wydarzyło.

**Wciśnijmy wszyscy pływający przycisk akcji**

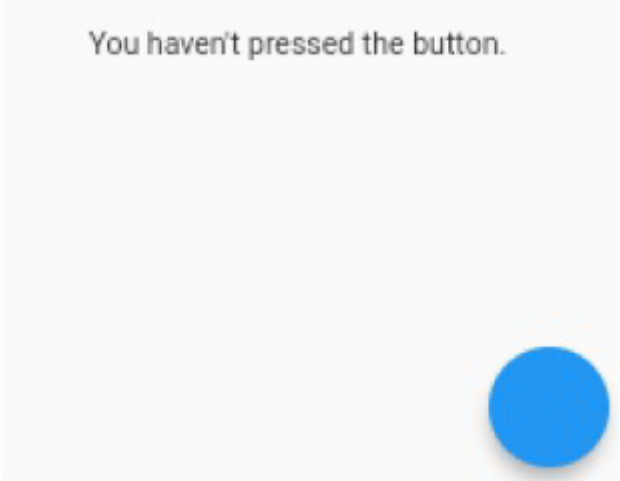
Gdy tworzysz nowy projekt Flutter, Android Studio tworzy dla Ciebie plik main.dart. Plik main.dart zawiera uroczą małą aplikację startową. Listing 1 zawiera pomniejszoną wersję tej aplikacji startowej.

**LISTING 1 Naciśnij przycisk; Zmień ekran**

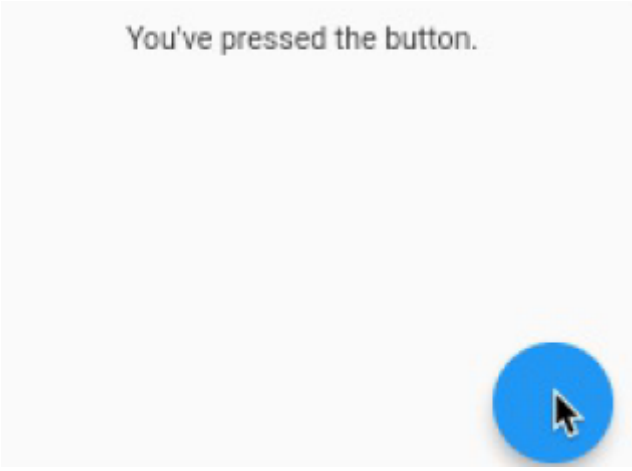
```
import 'package:flutter/material.dart';

void main() => runApp(App0501());

class App0501 extends StatelessWidget {

Widget build(BuildContext context) {

return MaterialApp(
```

```
home: MyHomePage(),
);
}
}
class MyHomePage extends StatefulWidget {
_MyHomePageState createState() =>
_MyHomePageState();
}
class _MyHomePageState extends State {
String _pressedOrNot = "You haven't pressed
the button.";
void _changeText() {
setState(_getNewText);
}
void _getNewText() {
_pressedOrNot = "You've pressed the
button.";
}
Widget build(BuildContext context) {
return Scaffold(
body: Center(
child: Text(
_pressedOrNot,
),
),
floatingActionButton:
FloatingActionButton(
onPressed: _changeText,
));
}
}
```

Kod z listingu 1 oddaje istotę aplikacji startowej w wersji Android Studio z października 2019 r. Zanim przeczytasz tę książkę, twórcy Flutter mogli całkowicie zmienić aplikację startową. Jeśli elementy z listingu 1 w niewielkim stopniu przypominają aplikację startową, którą otrzymujesz podczas tworzenia nowego projektu, nie martw się. Po prostu rób to, co robiłeś. Oznacza to, że usuń cały kod main.dart Android Studio i zastąp go kodem z Listingu 1. Po uruchomieniu aplikacji na listingu 1 zobaczysz tekst „Nie nacisnąłeś przycisku" oraz w prawym dolnym rogu ekranu niebieskie kółko.




To niebieskie kółko nazywa się pływającym przyciskiem akcji. Jest to jeden z widżetów, które można dodać do rusztowania. Gdy klikniesz pływający przycisk akcji tej aplikacji, słowa na ekranie zmienią się na „Nacisnąłeś przycisk".




W końcu! Aplikacja Flutter sprawia, że coś się dzieje!

Aby zrozumieć, co się dzieje, musisz wiedzieć o dwóch rodzajach widżetów. Aby poznać ich nazwy, przeczytaj tytuł następnej sekcji.

**Widżety bezstanowe i widżety stanowe**

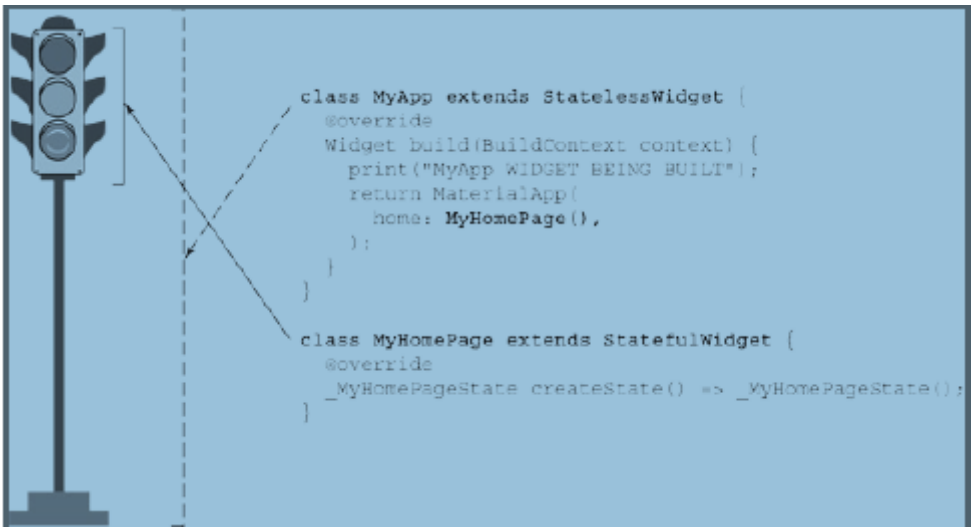
Niektóre systemy mają właściwości, które mogą zmieniać się w czasie. Weźmy na przykład zwykłą, codzienną sygnalizację świetlną. Jeśli działa prawidłowo, jest czerwony, żółty lub zielony. Wyobraź sobie, że spieszysz się do pracy i zatrzymujesz się na czerwonym świetle. Możesz narzekać pod nosem: „Jestem zirytowany, że ta sygnalizacja świetlna jest czerwona. Chciałbym, żeby stan tego systemu zmienił się na zielony." Stan systemu jest właściwością systemu, która może zmieniać się w czasie.

Ta wskazówka nie ma nic wspólnego z Flutterem. Jeśli spotkasz kogoś z innego kraju, zapytaj go o kolor środkowej żarówki na sygnalizacji świetlnej. Podczas krótkiej rozmowy z pięcioma osobami uzyskałem kolor żółty, bursztynowy, złoty i pomarańczowy. Zobacz, ile różnych nazw kolorów możesz zebrać.

Aplikacja z listingu 1 ma stronę główną (o nazwie MyHomePage), która znajduje się w jednym z dwóch stanów. Jeden stan pokazano na rysunku 1. Jest to stan, w którym widżet Tekst wyświetla komunikat „Nie nacisnąłeś przycisku". Drugi stan pokazano na rysunku 2. Jest to stan, w którym widżet Tekst wyświetla komunikat „Nacisnąłeś przycisk". Na listingu 1 pierwszy wiersz deklaracji klasy MyHomePage to

class MyHomePage extend StatefulWidget

Chcesz, aby wygląd widżetu MyHomePage mógł się szybko zmieniać, więc deklarujesz obiekty MyHomePage jako widżety stanowe. Każda instancja MyHomePage ma swój stan — coś, co może się zmieniać w czasie. Natomiast klasa App0501 z listingu 1 jest widżetem bezstanowym. Sama aplikacja (App0501) polega na swojej stronie głównej, aby śledzić wyświetlany tekst. Tak więc aplikacja nie musi pamiętać, czy jest w takim czy innym stanie. Nic w instancji App0501 nie zmienia się podczas wykonywania tego kodu. Pomyśl jeszcze raz o sygnalizacji świetlnej. Część z żarówkami spoczywa na słupku, który jest trwale przymocowany do podłoża. Cały zespół — słup, żarówki i wszystko inne — nie zmienia się. Ale prądy płynące przez żarówki zmieniają się mniej więcej co 30 sekund. Masz to. Cały zespół jest niezmienny i bezstanowy, ale część tego zespołu — część odpowiedzialna za wyświetlanie kolorów — zmienia się i jest stanowa.




**Widżety mają metody**

Na listingu 1 deklaracja klasy App0501 zawiera funkcję o nazwie build. Funkcja zdefiniowana wewnątrz deklaracji klasy nazywana jest metodą. Klasa App0501 ma metodę kompilacji. To dobrze, ponieważ w kodzie StatelessWidget jest drobny druk. Zgodnie z tym drobnym drukiem, każda klasa rozszerzająca StatelessWidget musi zawierać deklarację metody budowania. Metoda budowania bezstanowego widżetu mówi Flutterowi, jak zbudować widżet. Metoda opisuje między innymi wygląd i zachowanie widżetu. Za każdym razem, gdy uruchamiasz program z Listingu 1, Flutter wywołuje metodę build klasy App0501. Ta metoda kompilacji konstruuje instancję MaterialApp, która z kolei konstruuje instancję MyHomePage. I tak dalej. Od tego momentu instancja MaterialApp nie zmienia się. Tak, rzeczy wewnątrz instancji MaterialApp zmieniają się, ale sama instancja się nie zmienia. Jak często Twoje

miasto buduje nowy zespół sygnalizacji świetlnej? Tam, gdzie mieszkam, mogę zobaczyć, jak jeden pojawia się mniej więcej co dwa lata. Metalowa część sygnalizacji świetlnej nie jest zaprojektowana do regularnej wymiany. Planiści miejscy nazywają metodę budowania zespołu sygnalizacji świetlnej tylko wtedy, gdy konstruują nowe światło. To samo dotyczy bezstanowych widżetów we Flutterze. Widżet bezstanowy nie jest przeznaczony do zmiany. Gdy bezstanowy widżet wymaga zmiany, Flutter zastępuje widżet. A co z widżetami stanowymi? Czy mają metody kompilacji? Cóż, robią i nie. Każdy stanowy widżet musi mieć metodę createState. Metoda createState tworzy instancję klasy State Fluttera, a każda klasa State ma własną metodę budowania. Innymi słowy, stanowy widżet nie tworzy się sam. Zamiast tego stanowy widżet tworzy stan, a stan buduje się sam

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: MyHomePage(),
    );
  }
}
```

A stateless widget builds itself.

```
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}
```

A stateful widget creates a state, and

```
class _MyHomePageState extends State<MyHomePage> {
  String _pressedOrNot = "You haven't pressed the button.";

  void _changeText() {
    setState(() {
      _pressedOrNot = "You've pressed the button.";
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
        body: Center(
          child: Text(
            _pressedOrNot,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: _changeText,
        ));
  }
}
```

... the state builds itself.

**„MÓWIĘ DO CIEBIE, BEZPAŃSTWOWY WIDŻECIE — MUSISZ MIEĆ METODĘ BUDOWANIA!”**

Każda klasa rozszerzająca StatelessWidget musi mieć metodę kompilacji. API Fluttera wymusza tę regułę. Ale nie wierz mi na słowo. Tymczasowo wykomentuj deklarację metody kompilacji z listingu 1. To znaczy zmień deklarację App0501 tak, aby wyglądała tak:

klasa App0501 rozszerza StatelessWidget {

// Kompilacja widżetu (kontekst BuildContext) {

// zwróć MaterialApp(

// strona główna: MojaStronaGłówna(),

// );

// }

}

Gdy to zrobisz, zobaczysz czerwone znaki w edytorze Android Studio. Czerwone znaki oznaczają, że program zawiera błąd; mianowicie, że App0501 nie ma własnej metody kompilacji.

Aby szybko dodać komentarz do kilku wierszy kodu, przeciągnij mysz tak, aby podświetlenie dotykało każdego z tych wierszy. Następnie, jeśli używasz systemu Windows, naciśnij Ctrl-/. Jeśli używasz komputera Mac, naciśnij Cmd-/.

W jaki sposób Dart egzekwuje wymagania dotyczące metody kompilacji? Jako początkujący programista nie musisz znać odpowiedzi. Możesz pominąć resztę tego paska bocznego i wesoło iść swoją drogą. Ale jeśli jesteś ciekawy i nie masz nic przeciwko zrobieniu małego objazdu w nauce, spróbuj tego: W edytorze Android Studio kliknij prawym przyciskiem myszy słowo StatelessWidget. W wyświetlonym menu kontekstowym wybierz opcję Przejdź do ⇒ Deklaracja. Gotowe! A

otworzy się nowa zakładka zawierająca deklarację klasy StatelessWidget w edytorze. Jeśli zignorujesz większość kodu w deklaracji klasy StatelessWidget, zobaczysz coś takiego:

abstract class StatelessWidget extends

Widget {

// A bunch of code that you don't have

to worry about, followed by …

Widget build(BuildContext context);

}

Pierwsze słowo, abstract, ostrzega Darta, że ta deklaracja klasy zawiera metody (czyli funkcje) bez treści. A właściwie wiersz

Kompilacja widżetu (kontekst BuildContext);

jest nagłówkiem metody bez ciała. Zamiast ciała jest tylko a średnik. Nie będziesz zdziwiony, gdy dowiesz się, że StatelessWidget jest przykładem klasy abstrakcyjnej, a metoda budowania tej klasy jest metodą abstrakcyjną. Mając to na uwadze, przedstawiam wam te dwa fakty:

* Nie możesz wywołać konstruktora dla klasy abstrakcyjnej.

Widżet Text można utworzyć, pisząc Text("Hello"), ponieważ klasa Text nie jest abstrakcyjna. Ale nie możesz skonstruować StatelessWidget, pisząc StatelessWidget(). Ma to sens, ponieważ w deklaracji StatelessWidget metoda budowania nie jest w pełni zdefiniowana. *Jeśli rozszerzasz klasę abstrakcyjną, musisz podać pełną deklarację dla każdej metody abstrakcyjnej tej klasy.

Deklaracja klasy StatelessWidget zawiera następujący wiersz:

Kompilacja widżetu (kontekst BuildContext);

Z tego powodu klasa App0501 na listingu 5.1 musi zawierać pełną deklarację metody budowania. Co więcej deklaracja musi określać parametr typu BuildContext. Rzeczywiście, metoda kompilacji należąca do App0501 spełnia swoje zadanie:

```
Widget build(BuildContext context) {

return MaterialApp(

home: MyHomePage(),

);

}
```

Dzięki w pełni zdefiniowanej metodzie kompilacji klasa App0501 nie jest abstrakcyjna. To dobrze, ponieważ na początku Listingu 5-1 znajduje się wiersz zawierający wywołanie konstruktora App0501().

Stan typowej sygnalizacji świetlnej zmienia się co 30 sekund lub co kilka minut, a zatem stan światła jest odbudowywany. W ten sam sposób metoda build, która należy (pośrednio) do widżetu stanowego, jest wielokrotnie wywoływana podczas wykonywania programu. Do tego służą stanowe widżety. To zwinne stworzenia, których wygląd można łatwo zmienić. Z kolei widżet bezstanowy jest jak słup sygnalizacji świetlnej. Jest to sztywna konstrukcja przeznaczona do jednorazowego użytku.

**Nie zwracaj uwagi na ramy za kurtyną**

Program wyświetlający przyciski i inne ładnie wyglądające rzeczy ma graficzny interfejs użytkownika. Taki interfejs jest powszechnie nazywany GUI (wymawiane „lepki", jak w „To masło orzechowe jest naprawdę lepkie"). W wielu programach GUI rzeczy dzieją się za kulisami. Podczas działania kodu Twojej aplikacji w tle działa wiele innych kodów. Kiedy uruchamiasz aplikację Flutter, kod napisany przez twórców Flutter jest stale uruchamiany, aby wspierać kod Twojej własnej aplikacji. Ten kod wsparcia w tle należy do platformy Flutter.

Listing 1 zawiera deklaracje funkcji main, build, createState, _getNewText i _changeText, ale kod z listingu 1 nie wywołuje żadnej z tych funkcji. Zamiast tego kod platformy Flutter wywołuje te funkcje, gdy urządzenie uruchamia aplikację. Oto szczegółowy opis:

Język Dart wywołuje główną funkcję, gdy zaczyna działać kod z Listingu 1.

Główna funkcja konstruuje instancję App0501 i wywołuje runApp, aby wszystko działało. Następnie …

Framework Flutter wywołuje funkcję kompilacji instancji App0501.

Funkcja build konstruuje instancję MyHomePage. Następnie …

Framework Flutter wywołuje funkcję createState instancji MyHomePage.

Funkcja createState konstruuje instancję _myHomePageState. Następnie …

Framework Flutter wywołuje funkcję kompilacji instancji _myHomePageState.

Funkcja budowania konstruuje rusztowanie zawierające Centrum z widżetem Tekst i a Widżet FloatingActionButton.

Aby zrozumieć konstruktor widżetu Tekst, spójrz na kilka wierszy kodu:

```
String _pressedOrNot = "You haven't pressed the
```

```
button.";

// Later in the listing …

child: Text(

_pressedOrNot,

),
```

Początkowo wartość zmiennej _pressedOrNot to „Nie nacisnąłeś przycisku”. Tak więc, gdy aplikacja zaczyna działać, widżet Tekst posłusznie wyświetla komunikat „Nie nacisnąłeś przycisku”. Ale kod pływającego przycisku akcji to inna historia.

```
void _changeText() {

setState(_getNewText);

}

void _getNewText() {

_pressedOrNot = "You've pressed the button.";

}

// Later in the listing …

floatingActionButton:

FloatingActionButton(

onPressed: _changeText,

)
```

Konstruktor elementu FloatingActionButton ma parametr onPressed, a wartość tego parametru to _changeText. O co w tym wszystkim chodzi? Parametr onPressed mówi Flutterowi: „Jeśli i kiedy użytkownik naciśnie przycisk, urządzenie wywoła funkcję _changeText”. W rzeczywistości wiele rzeczy dzieje się, gdy użytkownik naciska pływający przycisk akcji. W kilku następnych sekcjach zobaczysz niektóre szczegóły.

**Wielkie wydarzenie**

W programowaniu GUI zdarzenie to coś, co się dzieje – coś, co może wymagać pewnego rodzaju odpowiedzi. Przykładem zdarzenia jest naciśnięcie przycisku. Inne przykłady zdarzeń to przychodzące połączenie telefoniczne, przeniesienie urządzenia do nowej lokalizacji GPS lub fakt, że jedna aplikacja potrzebuje informacji z innej aplikacji. Procedura obsługi zdarzeń to funkcja wywoływana, gdy wystąpi zdarzenie. Na listingu 5.1 funkcja _changeText jest procedurą obsługi zdarzenia onPressed przycisku. Sam kod onPressed: _changeText nie wywołuje funkcji _changeText. Zamiast tego kod ten rejestruje funkcję _changeText jako oficjalną procedurę obsługi naciśnięć przycisków pływających akcji. Wywołanie funkcji _changeText wyglądałoby tak: _changeText(). Wywołanie zakończy się otwierającymi i zamykającymi nawiasami. Kod onPressed: _changeText bez nawiasów nie wywołuje funkcji _changeText. Ten kod nakazuje urządzeniu zapamiętać, że nazwa procedury obsługi zdarzenia onPressed przycisku to _changeText. Urządzenie korzysta z tych informacji wtedy i tylko wtedy, gdy użytkownik naciśnie przycisk.

**Oddzwoń**

Telefon dzwoni cztery razy. Nikt nie odpowiada, ale słyszę nagrany komunikat. „To jest Steve Hayes — redaktor naczelny w John Wiley and Sons. Przepraszam, że nie ma mnie tutaj, aby odebrać telefon. Zostaw wiadomość, a ja skontaktuję się z Tobą tak szybko, jak to możliwe”. <beep>

„Cześć, Steve. To jest Barry. Rękopis Flutter For Dummies idzie dobrze, ale spóźni się o kilka miesięcy. Zadzwoń do mnie, abyśmy mogli omówić nowy rozkład jazdy. Nie dzwoń do mnie na mój zwykły numer telefonu. Zamiast tego zadzwoń do mojego hotelu w Taha'a w Polinezji Francuskiej. Numer to +689 49 55 55 55. Pa! Mój numer telefonu w Taha'a to numer oddzwonienia. W ten sam sposób funkcje _changeText i _getNewText na listingu 1 są wywołaniami zwrotnymi. Linia onPressed: _changeText mówi frameworkowi: „Zadzwoń do mnie, wywołując moją funkcję _changeText”. I linia setState(_getNewText) mówi frameworkowi „Oddzwoń do mnie, wywołując moją funkcję _getNewText”.

**Wywołania zwrotne są przydatne**

Być może masz napisane programy, które nie mają wywołań zwrotnych. Kiedy twój program zaczyna działać, system wykonuje pierwszą linię kodu i kontynuuje wykonywanie instrukcji, aż dojdzie do ostatniej linii kodu. Od początku do końca wszystko przebiega zgodnie z planem. (Cóż, w najlepszych okolicznościach wszystko przebiega zgodnie z planem). Wywołanie zwrotne dodaje do programu element niepewności. Kiedy odbędzie się wydarzenie? Kiedy zostanie wywołana funkcja? Gdzie jest kod wywołujący funkcję? Programy z wywołaniami zwrotnymi są trudniejsze do zrozumienia niż programy bez wywołań zwrotnych. Dlaczego potrzebujesz wywołań zwrotnych? Czy możesz uciec bez nich? Aby pomóc odpowiedzieć na to pytanie, pomyśl o swoim zwykłym, codziennym budziku. Przed pójściem spać każesz budzikowi wysłać dźwięk do twoich uszu (oddzwanianie), gdy wybije godzina 9:00. zdarzenie ma miejsce:

on9am: _rattleMyEarDrums,

Gdybyś nie polegał na oddzwonieniu, musiałbyś samodzielnie śledzić czas przez całą noc. Podobnie jak Bart i Lisa Simpson na tylnym siedzeniu samochodu, ciągle pytasz: „Czy jest 9 rano? już? Czy jest 9 rano? już? Czy jest 9 rano? już?" Na pewno nie przespałbyś nocy. Z tego samego powodu, gdyby program Flutter musiał sprawdzać co sto milisekund ostatnie naciśnięcie przycisku, nie miałby zbyt wiele czasu na zrobienie czegokolwiek innego. Dlatego potrzebujesz wywołań zwrotnych w programach Flutter.

Programowanie z wywołaniami zwrotnymi nazywa się programowaniem sterowanym zdarzeniami. Jeśli program nie używa wywołań zwrotnych, a zamiast tego wielokrotnie sprawdza naciśnięcia przycisków i inne podobne rzeczy, ten program odpytuje. W niektórych sytuacjach ankietowanie jest nieuniknione. Ale kiedy możliwe jest programowanie sterowane zdarzeniami, jest ono znacznie lepsze niż odpytywanie.

**Zarys kodu**

Dobrym sposobem patrzenia na kod jest mrużenie oczu, tak aby większość kodu była rozmazana i nieczytelna. Część, którą nadal możesz przeczytać, jest częścią ważną. Rysunek 5-5 zawiera moją w większości niewyraźną wersję kodu z Listingu 1.

```
class _MyHomePageState extends State {
  String _pressedOrNot = "You haven't pressed the button.";

  void _changeText() {
    setState(_getNewText);
  }

  void _getNewText() {
    _pressedOrNot = "You've pressed the button.";
  }

  Widget build(BuildContext context) {
    return Scaffold(
        body: Center(
          child: Text(
            _pressedOrNot,
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: _changeText,
        ));
  }
}
```

② _changeText calls setState. Behind the scenes, setState calls _getNewText.

③ _getNewText changes something about the home page's state.

④ Behind the scenes, setState calls build, which redraws the home page.

① This line registers _changeText as the callback function for a button press event.

Zgodnie z rysunkiem , jest to strategia zarządzania stanem z Listingu 1:

1. Zarejestruj _changeText jako funkcję wywołania zwrotnego i poczekaj, aż użytkownik naciśnie pływający przycisk akcji. Kiedy w końcu użytkownik naciśnie pływający przycisk akcji, …

2. Wywołaj funkcję _changeText setState i przekaż _getNewText jako jedyny parametr w wywołaniu funkcji setState. Funkcja setState wywołuje _getNewText. Kiedy to nastąpi,…

3. Funkcja _getNewText robi wszystko, co ma wspólnego z jakimś tekstem. Funkcja setState powoduje również, że framework Flutter wywołuje build. Kiedy to nastąpi,…

4. Rzeczy na ekranie użytkownika są przebudowywane. Przebudowany ekran wyświetla nowy tekst.

Nie ma nic specjalnego w strategii zarządzania stanami z Listingu 1. Możesz skopiować i wkleić tę strategię do wielu innych programów. Rysunek 5-6 przedstawia ogólną ideę.

```
class _MyHomePageState extends State {
  blah, blah, blah, blabbity blah, blah, blah;

  void _handlerFunction() {
    setState(_getNewInfo);
  }

  void _getNewInfo() {
    blah, blah, blah, don't read this;
  }

  Widget build(BuildContext context) {
    send email if(
        you've noticed(
          that I've(
            changed this text,
          ),
        ),
        Yada, yada, yada, sis boom bah(
          onEvent: _handlerFunction,
        ));
  }
```

② _handlerFunction calls setState. Behind the scenes, setState calls _getNewInfo.

③ _getNewInfo changes something about the home page's state.

④ Behind the scenes, setState calls build, which redraws the home page.

① This line registers _handlerFunction as the callback function for a certain kind of event.

Zgodnie z rysunkiem , te kroki tworzą strategię zarządzania stanem:

1. Zarejestruj funkcję jako funkcję wywołania zwrotnego dla zdarzenia i poczekaj, aż to zdarzenie nastąpi. Na rysunku nazwa funkcji wywołania zwrotnego to _handlerFunction. Podobnie jak wszystkie tego typu funkcje, funkcja _handlerFunction nie przyjmuje żadnych parametrów i zwraca wartość void.

Kiedy w końcu impreza się odbędzie…

2. Wywołaj funkcję wywołania zwrotnego setState i przekaż inną funkcję jako jedyny parametr w wywołaniu funkcji setState. Na rysunku 5.6 nazwa tej innej funkcji to _getNewInfo. Podobnie jak wszystkie tego typu funkcje, funkcja _getNewInfo nie przyjmuje żadnych parametrów i zwraca wartość void. Funkcja setState wywołuje _getNewInfo (lub inną nazwę, której użyłeś, inną niż _getNewInfo). Kiedy to nastąpi,…

3. Funkcja _getNewInfo zmienia coś w stanie widżetu. Funkcja setState powoduje również, że framework Flutter wywołuje build. Kiedy to nastąpi,…

4. Rzeczy na ekranie użytkownika są przebudowywane. Przebudowany ekran wyświetla widżet w nowym stanie. I tak to idzie.

**Daj spokój, co się naprawdę dzieje?**

Kiedy uruchamiasz program z graficznym interfejsem użytkownika, wiele rzeczy dzieje się za kulisami. Jeśli chcesz, możesz spojrzeć na kod frameworka, ale ten kod może być dość złożony. Poza tym, z każdym przyzwoitym frameworkiem, nie powinieneś czytać jego własnego kodu. Powinieneś być w stanie wywoływać funkcje i konstruktory frameworka, znając tylko to, co znajduje się w dokumentacji frameworka. Wiem na pewno, że po uruchomieniu Listingu 5-1 wywołanie setState skutkuje wywołaniem _getNewText. Wiem to, ponieważ kiedy komentuję wywołanie setState, tekst się nie zmienia. Ale, przyznaję, nigdy nie czułem się całkowicie komfortowo z magią jakiegokolwiek frameworka GUI. Chcę poznać wewnętrzne mechanizmy frameworka, nawet jeśli jest to tylko ogólny zarys. (Mam to samo ze wszystkim. Nie jestem pewien, czy światło gaśnie, kiedy zamykam drzwi lodówki.)

**CO ZROBIĆ, GDY WYWOŁASZ setState**

Wypróbuj ten eksperyment: Zmodyfikuj funkcję _changeText z Listingu 5-1 w następujący sposób:

```
void _changeText() {

_getNewText();

setState(_doNothing);

}

void _doNothing() {}
```

Przenieś odwołanie do _getNewText poza funkcję setState. Po tym ruchu zmiana tekstu następuje przed wywołaniem setState, więc setState nie musi wywoływać _getNewText. Oczywiście nadal musisz podać setState funkcję do wywołania, więc podajesz jej funkcję _doNothing. Ta funkcja _doNothing utrzymuje zajętość setState podczas przygotowań do wywołania metody build. Czy zmodyfikowany kod działa? W przykładzie z tego rozdziału tak jest. Ale generalnie taka zmiana to zły pomysł. Umieszczenie _getNewText wewnątrz wywołania setState gwarantuje, że przypisanie do _pressedOrNot i wywołanie build nastąpią razem. W bardziej skomplikowanym programie wywołanie kompilacji może być opóźnione, a wyniki mogą być dziwne. Oto kolejna rzecz do rozważenia: na listingu 1 funkcja _getNewText zawiera jedną prostą instrukcję przypisania. Ale wyobraź sobie aplikację, która

wykonuje długie, czasochłonne obliczenia przed wyświetleniem wyniku tych obliczeń. Aktualizacja ekranu jest w trzech częściach:

1. Wykonaj obliczenia.

2. Zmień wyświetlany tekst tak, aby zawierał wynik obliczeń.

3. Zbuduj wywołanie struktury, aby odświeżyć ekran.

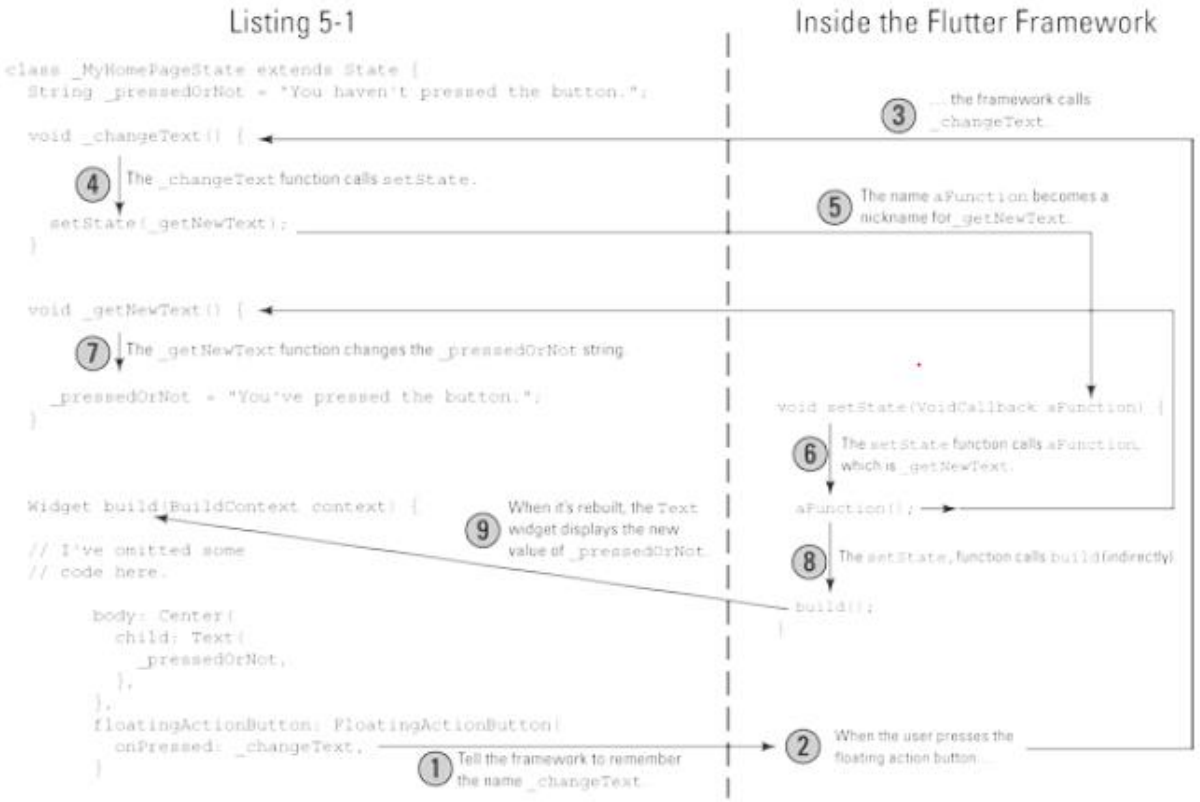
W takim przypadku eksperci Flutter zalecają następujący podział pracy:

* Wykonaj długie, czasochłonne obliczenia przed wywołaniem setState.

* Dokonaj zmiany tekstu w parametrze, gdy wywołasz setState.

Innymi słowy, trzymaj kod wykonujący ciężkie operacje poza wywołaniem setState, ale umieść kod zmieniający wartości stanu wewnątrz wywołania setState. To dobra rada.

W tym celu przedstawiam rysunek . Rysunek podsumowuje opis obsługi zdarzeń w kilku poprzednich sekcjach. Ilustruje on niektóre działania z Listingu 1, w tym podsumowanie kodu w funkcji setState. Nie popełnij błędu: rysunek to uproszczony widok tego, co dzieje się, gdy Flutter obsługuje zdarzenie, ale rysunek ten może ci się przydać. Nauczyłem się kilku rzeczy po prostu rysując figurę.




**Ulepszanie Twojej aplikacji**

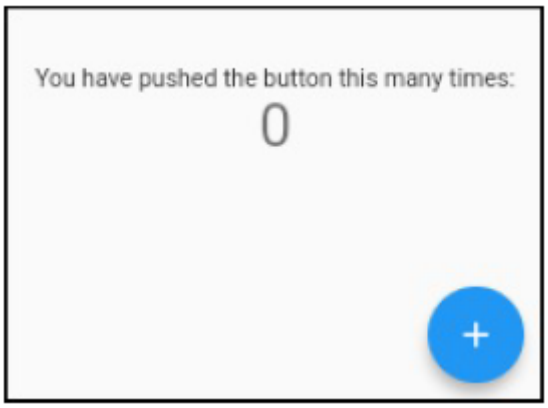
Kod na listingu 1 to uproszczona wersja aplikacji startowej Android Studio. To miłe, ale może chcesz dowiedzieć się więcej o aplikacji startowej. W tym celu Listing 2 zawiera kilka dodatkowych funkcji — funkcji, które poprawiają wygląd i zachowanie prostego programu demonstracyjnego Flutter.

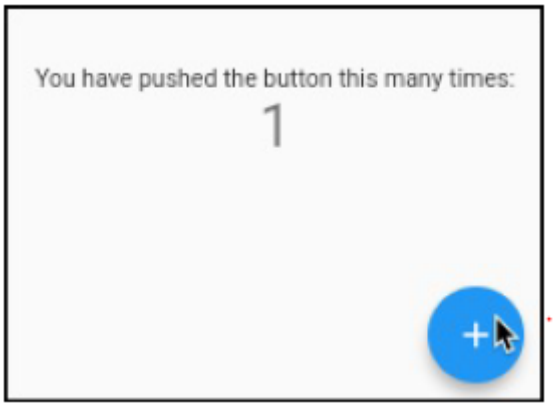
**LISTING 2 Przybliżanie się do aplikacji startowej Android Studio**

```
import 'package:flutter/material.dart';
void main() => runApp(App0502());
class App0502 extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: ’Flutter Demo’,
theme: ThemeData(
primarySwatch: Colors.blue,
),
home: MyHomePage(),
);
}
}
class MyHomePage extends StatefulWidget {
@override
_MyHomePageState createState() =>
_MyHomePageState();
}
class _MyHomePageState extends State {
int _counter = 0;
void _incrementCounter() {
setState(() {
_counter++;
});
}
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text("Listing 5-2"),
```

```
),
body: Center(
child: Column(
mainAxisAlignment:
MainAxisAlignment.center,
children: <Widget>[
Text(
'You have pushed the button this
many times:',
),
Text(
’$_counter’,
style:
Theme.of(context).textTheme.display1,
),
],
),
),
floatingActionButton:
FloatingActionButton(
onPressed: _incrementCounter,
tooltip: ’Increment’,
child: Icon(Icons.add),
),
);
}
}
```

Rysunki poniżej przedstawiają przebieg kodu z Listingu 2. Rysunek  pokazuje, co widzisz, gdy aplikacja zaczyna działać, a rysunek drugi to, co widzisz po jednym kliknięciu pływającego przycisku akcji. Przy kolejnych kliknięciach zobaczysz cyfry 2, 3, 4 itd.




Za każdym razem, gdy użytkownik kliknie pływający przycisk akcji, liczba na ekranie zwiększa się o 1. Aby tak się stało, na listingu 5.2 znajdują się trzy odwołania do zmiennej o nazwie _counter. Rysunek 5-10 ilustruje rolę zmiennej _counter w działaniu aplikacji.

Widżet Tekst aplikacji wyświetla wartość zmiennej _counter. Tak więc, gdy aplikacja zaczyna działać, widżet Tekst wyświetla 0. Kiedy użytkownik po raz pierwszy naciska pływający przycisk akcji, a platforma Flutter wywołuje setState, zmienna _counter przyjmuje wartość 1. Tak więc liczba 1 pojawia się na środku ekranu aplikacji. Gdy użytkownik ponownie naciśnie przycisk akcji, _counter zmieni się na 2 i tak dalej.

**Więcej parametrów proszę**

Listing 2 przedstawia kilka sprawdzonych parametrów konstruktora. Na przykład konstruktor MaterialApp ma parametry tytułu i motywu. Tytuł (w tym przykładzie Flutter Demo) pojawia się tylko na telefonach z Androidem i tylko wtedy, gdy użytkownik wyczaruje listę ostatnich aplikacji. Wartością motywu jest instancja ThemeData (stąd użycie konstruktora ThemeData z Listingu 2). W świecie projektowania aplikacji motywy są niezwykle ważne. Motyw to zbiór opcji, które mają zastosowanie do wszystkich części aplikacji. Na przykład „Użyj czcionki Roboto dla wszystkich elementów niezwiązanych z ułatwieniami dostępu” to wybór, który może być częścią motywu. Wybór dokonany na listingu 2 brzmi: „Użyj próbki koloru niebieskiego w całej aplikacji”. Próbka to zbiór podobnych kolorów — wariacji na temat jednego koloru, których można używać w całej aplikacji. Próbka Colors.blue zawiera dziesięć odcieni niebieskiego, od bardzo jasnego do bardzo ciemnego. W ramach eksperymentu uruchom kod z listingu 2, a następnie zmień Colors.blue na Colors.deepOrange lub Colors.blueGrey. Po zapisaniu zmiany wszystkie elementy w aplikacji nagle wyglądają inaczej. To super! Nie musisz określać koloru każdego widżetu.

Motyw zachowuje spójny wygląd wśród wszystkich widżetów na ekranie. W przypadku dużej aplikacji z więcej niż jedną stroną motyw zachowuje spójny wygląd z jednej strony na drugą. Pomaga to użytkownikowi zrozumieć przepływ elementów w aplikacji. Na listingu 2 parametr stylu widżetu Text wykorzystuje okrężną drogę do uzyskania instancji TextStyle. Kod Theme.of(context).textTheme.display1 reprezentuje styl TextStyle o dużym rozmiarze tekstu. Rysunek pokazuje opcje, które są dostępne podczas używania Theme.of(context).textTheme.


Określony styl może być zbyt duży dla ekranów niektórych telefonów. Na przykład, aby utworzyć rysunek , uruchomiłem emulator z urządzeniem wirtualnym Pixel 3 XL. Ale przy zwykłym starym Pixelu 3 w trybie portretowym słowo display4 jest zbyt duże w stosunku do szerokości ekranu. Cyfra 4 pojawia się w osobnym wierszu.

Podobnie jak w przypadku motywu MaterialApp, pojęcie motywu tekstowego jest bardzo przydatne. Gdy polegasz na wartościach Theme.of(context).textTheme Fluttera, zapewniasz jednolity wygląd wszystkich elementów tekstowych w swojej aplikacji. Możesz także pocieszyć się faktem, że używasz wartości standardowych — ładnie wyglądających wartości wybranych przez profesjonalnych projektantów aplikacji.

Nazwy takie jak display1 w API Fluttera nie odpowiadają dokładnie nazwom w specyfikacji Google Material Design i podejrzewam, że opcje dostępne w API Fluttera wkrótce się zmienią.

Wreszcie pływający przycisk akcji na listingu 2 ma podpowiedź i parametry potomne.

* Ciąg podpowiedzi pojawia się, gdy użytkownik długo naciska przycisk. Gdy dotykasz ekranu i trzymasz palec w tym samym miejscu przez sekundę lub dwie, długo naciskasz tę część ekranu. Aplikacja z listingu 2 wyświetla słowo Increment za każdym razem, gdy użytkownik długo naciska pływający przycisk akcji.

*Dla elementu potomnego przycisku konstruujesz instancję Icon. Instancja Icon wyświetla mały obrazek z klasy Icons Fluttera; mianowicie obraz Icons.add. Oczywiście, ten obraz jest znakiem plus

Adnotacja zastąpienia

Wiersz @override, który pojawia się kilka razy na listingu 2, nazywany jest adnotacją. W Dart adnotacja zaczyna się od znaku at (@). Instrukcja, taka jak _pressedOrNot = „Nacisnąłeś przycisk.”, mówi Dartowi, co ma robić podczas wykonywania programu. Ale adnotacja jest inna. Adnotacja mówi Dartowi coś o części programu Dart. Adnotacja @override przypomina Dartowi, że rozszerzana klasa ma pasującą deklarację. Rozważmy na przykład następujący kod z Listingu 2:

```
class App0502 extends StatelessWidget {

@override

Widget build(BuildContext context) {
```

Wiersz @override mówi: „Klasa StatelessWidget, którą rozszerza ta klasa App0502, ma własną deklarację metody build(kontekst BuildContext).” I rzeczywiście, zgodnie z wcześniejszym paskiem bocznym tego rozdziału „Mówię do ciebie, bezstanowy widżecie — musisz mieć metodę kompilacji!” klasa StatelessWidget w kodzie API Flutter ma metodę build(kontekst BuildContext) bez treści. Wszystko ładnie się układa. Listing 2 zawiera adnotacje @override, ale listing 1 ich nie ma. Spójrz na to! Możesz uciec bez adnotacji @override! Więc po co zawracać sobie nimi głowę? Odpowiedź brzmi „bezpieczeństwo”. Im więcej informacji podasz Dart o swoim kodzie, tym mniejsze jest prawdopodobieństwo, że Dart pozwoli ci zrobić coś złego. Jeśli popełnisz błąd i nieprawidłowo zadeklarujesz metodę kompilacji, Dart może cię ostrzec. "Hej! Powiedziałeś, że zamierzasz zastąpić metodę budowania zadeklarowaną w klasie StatelessWidget, ale twoja nowa metoda kompilacji nie robi tego poprawnie. Napraw to, przyjacielu!”

Możesz sprawić, by Dart ostrzegał Cię o metodach które nie pasują do ich adnotacji @override.

Co oznacza <Widżet>?

Na listingu 5.2 lista elementów podrzędnych w kolumnie zaczyna się od kilku dodatkowych elementów:

```
children: <Widget>[

Text(

'You have pushed the button this many times:',

),

Text(

'$_counter',

style:

Theme.of(context).textTheme.display1,

),

]
```

Słowo <Widget> wraz z otaczającymi go nawiasami ostrokątnymi jest nazywane ogólną, a lista rozpoczynająca się od ogólnej nazywana jest listą sparametryzowaną. Na listingu 5.2 ogólny <Widget> mówi Dartowi, że każda z wartości na liście jest w taki czy inny sposób Widżetem. Zgodnie z rozdziałem 3. każda instancja klasy Text jest instancją klasy Widget, więc generyczny <Widget> nie kłamie.

W wielu sytuacjach stosowanie leków generycznych jest kwestią bezpieczeństwa. Rozważ następujące dwa wiersze kodu:

```
var words1 = ["Hello", "Goodbye", 1108];

// No error message

var words2 = <String>["Hello", "Goodbye",

1108]; // Error message!
```

Możesz planować wypełnić swoją listę wartościami łańcuchowymi, ale kiedy deklarujesz words 1 i words 2, przypadkowo dołączasz wartość int 1108. Lista słów 1 nie jest sparametryzowana, więc Dart nie wychwytuje błędu. Ale lista słów2 jest sparametryzowana za pomocą ogólnego <String>, więc Dart wyłapuje błąd i odmawia uruchomienia kodu. Komunikat o błędzie mówi, że typu elementu „int” nie można przypisać do typu listy „String”. Na to powinieneś odpowiedzieć: „Dobry chwyt, Dart. Dziękuję bardzo."

**Funkcje anonimowe**

W języku programowania Dart niektóre funkcje nie mają nazw. Spójrz na następujący kod:

```
void _incrementCounter() {

setState(_addOne);

}

void _addOne() {

_counter++;

}
```

Wyobraź sobie, że Twoja aplikacja nie zawiera innych odniesień do _addOne W takim przypadku wymyśliłeś nazwę _addOne i użyłeś jej tylko raz w swojej aplikacji. Po co zawracać sobie głowę nadawaniem czemuś nazwy, jeśli będziesz jej używać tylko raz? „Nadajmy temu kłosowi nazwę „sinkadillie”. A teraz zjedzmy sinadillie. Aby utworzyć funkcję bez nazwy, usuwasz nazwę. Jeśli nagłówek funkcji ma zwracany typ, również go usuwasz. więc na przykład

```
void _addOne() {

_counter++;

}
```

becomes

```
() {

_counter++;

}
```

Gdy ustawisz to jako parametr wywołania funkcji setState, wygląda to tak:

```
void _incrementCounter() {

setState(() {
```

```
_counter++;
});
}
```

Funkcja bez nazwy nazywana jest funkcją anonimową. Gdy funkcja anonimowa zawiera więcej niż jedną instrukcję, instrukcje te muszą być ujęte w nawiasy klamrowe. Ale jeśli funkcja zawiera tylko jedną instrukcję, możesz użyć notacji grubych strzałek. Na przykład na listingu 5.2 poniższy kod będzie działał poprawnie:

```
void _incrementCounter() {
setState(() => _counter++);
}
```

**SPOTKANIE Z WIELKĄ PUSTKĄ**

Spójrz z nostalgią na fragment kodu z początku tego rozdziału. Znajduje się na listingu 1.

```
void _changeText() {
setState(_getNewText);
}
```

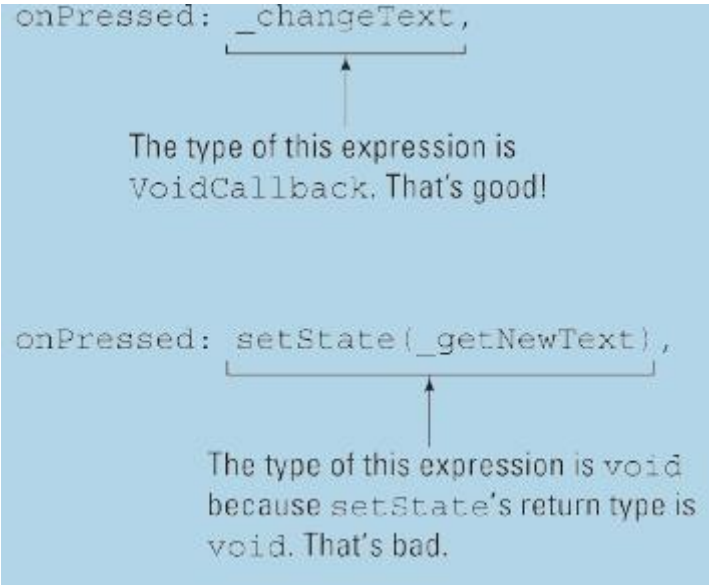
And later, in Listing 5-1:

```
floatingActionButton:
FloatingActionButton(
onPressed: _changeText,
))
```

Naciśnięcie przycisku wyzwala wywołanie _changeText, a funkcja _changeText wywołuje setState(_getNewText). Dlaczego nie wyeliminować pośrednika i skierować onPressed bezpośrednio do setState(_getNewText)? Wynikowy kod wyglądałby mniej więcej tak:

```
floatingActionButton:
FloatingActionButton(
onPressed: setState(_getNewText),
// This doesn't work.
))
```

Podczas pisania tego kodu pojawia się komunikat o błędzie: „Tutaj wyrażenie ma typ „void” i dlatego nie można go użyć”. Flutter chce, aby parametr onPressed był funkcją, ale wyrażenie setState(_getNewText) nie jest funkcją. Jest to wywołanie metody setState, a wywołanie funkcji setState zwraca wartość void.




Funkcja VoidCallback to funkcja, która nie przyjmuje żadnych argumentów i zwraca typ void. Częstym powodem tworzenia funkcji VoidCallback jest… no cóż… wywołanie zwrotne funkcji. Flutter chce, aby parametr onPressed był funkcją VoidCallback, a funkcja _changeText spełnia kryteria bycia funkcją VoidCallback. Tak więc na listingu 1 kod onPressed: _changeText jest w porządku i elegancki. Ale setState(_getNewText) nie jest VoidCallback. Nie, setState(_getNewText) to zwykła pustka. Tak więc kod onPressed: setState(_getNewText) wypada płasko. Jak możesz rozwiązać problem? Możesz powrócić do oryginalnego kodu z Listingu 1 lub uratować sytuację, korzystając z jeszcze jednej anonimowej funkcji. Wystarczy dodać () => przed odwołaniem do setState, jak tak:

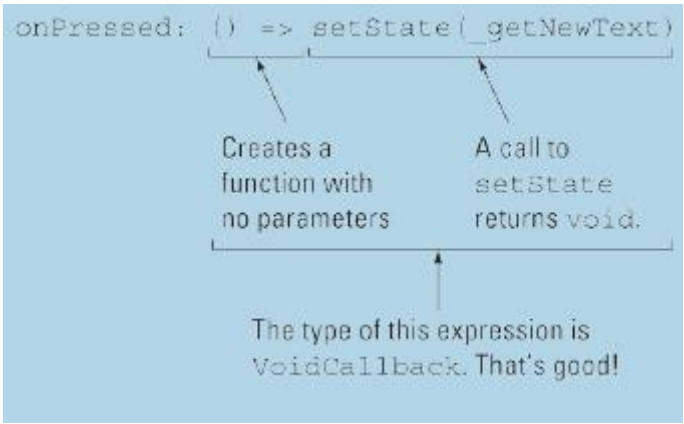
floatingActionButton:

FloatingActionButton(

onPressed: () => setState(_getNewText),

)

Druga cyfra tego paska bocznego opisuje cudowną zmianę, która ma miejsce po dodaniu kilku znaków do kodu. To, co wcześniej było wywołaniem setState, staje się VoidCallback i wszyscy są szczęśliwi. Co najważniejsze, Dart jest szczęśliwy. Twój program działa poprawnie.




**Co należy dokąd**

Na listingu 2 deklaracja zmiennej _counter znajduje się wewnątrz klasy _MyHomePageState, ale poza metodami _incrementCounter i build tej klasy. Zmienna tego rodzaju nazywana jest zmienną instancji lub polem. (To zależy od tego, kogo zapytasz.) Dlaczego zadeklarowałem zmienną _counter w tym konkretnym miejscu? Dlaczego nie umieścić deklaracji w innym miejscu w kodzie? Mógłbym napisać cały rozdział, aby szczegółowo odpowiedzieć na to pytanie, ale nie chcesz tego wszystkiego czytać, a ja na pewno nie chcę tego pisać. Zamiast tego proponuję kilka eksperymentów do wypróbowania:

1. Zaczynając od kodu z listingu 2, dodaj odwołanie do _counter wewnątrz klasy MyHomePage. Android Studio oznacza to nowe odniesienie postrzępionym czerwonym podkreśleniem. Podkreślenie zawstydza cię, przyznając, że to dodatkowe odniesienie było złym pomysłem. Zadeklarowano zmienną _counter w klasie _MyHomePageState, ale próbujesz odwołać się do zmiennej w innej klasie; mianowicie, Klasa Moja strona główna. Ilekroć deklarujesz zmienną wewnątrz klasy, ta zmienna jest lokalna dla tej klasy. Nie możesz odwoływać się do tej zmiennej poza klasą. W szczególności nie możesz odwoływać się do tej zmiennej w innej klasie.

2. Usuń odwołanie do _counter dodane w kroku 1. Następnie przenieś deklarację _counter na koniec klasy _MyHomePageState. Na początku klasy _MyHomePageState robisz _counter++. Ale nie deklarujesz zmiennej _counter do końca klasy _MyHomePageState. Mimo to program działa poprawnie. Morał z tej historii jest taki, że nie musisz deklarować zmiennej przed odwołaniem się do tej zmiennej. Ładny!

3. Przenieś deklarację _counter tak, aby znalazła się wewnątrz ciała funkcji _incrementCounter. Gdy to zrobisz, zobaczysz znacznik błędu przy wystąpieniu _counter w funkcji kompilacji. Zadeklarowałeś zmienną _counter wewnątrz funkcji _incrementCounter, ale próbujesz odwołać się do tej zmiennej w innej funkcji; mianowicie funkcja budowania. Ilekroć deklarujesz zmienną wewnątrz funkcji, ta zmienna jest lokalna dla funkcji. Nie możesz się do tego odnieść zmienną poza funkcją. W szczególności nie możesz odwoływać się do tej zmiennej wewnątrz innej funkcji.

4. Zachowaj deklarację _counter wewnątrz funkcji _incrementCounter i dodaj kolejną deklarację _counter wewnątrz funkcji budującej. Zainicjuj zmienną _counter funkcji kompilacji do 99. Gdy to zrobisz, komunikat o błędzie z kroku 3 zniknie. Więc kod jest poprawny. Prawidłowy? NIE! Kod jest nieprawidłowy. Po uruchomieniu kodu liczba na środku urządzenia to 99 i jej wartość nigdy się nie zmienia. Naciśnięcie pływającego przycisku akcji nie daje żadnego efektu. Co się dzieje? W tym poprawionym kodzie masz dwie różne zmienne _counter — jedną lokalną dla funkcji _incrementCounter, a drugą lokalną dla funkcji budującej. Instrukcja _counter++ dodaje 1 do jednej z tych zmiennych _counter, ale nie dodaje 1 do drugiej zmiennej _counter. To tak, jakby mieć dwóch ludzi o imieniu Barry Burd - jeden mieszka w New Jersey, a drugi w Kalifornii. Jeśli dodasz dolara na jedno z ich kont bankowych, druga osoba nie otrzyma automatycznie dodatkowego dolara.

5. Mieć tylko jedną deklarację _counter. Umieść go tuż przed początkiem klasy _MyHomePageState. Po dokonaniu tej zmiany edytor nie wyświetla żadnych znaczników błędów. Może klikniesz ikonę Uruchom, przewidując złe wieści. Albo aplikacja nie działa, albo działa i źle się zachowuje. Ale oto aplikacja działa poprawnie! Deklaracja, która nie znajduje się wewnątrz klasy ani funkcji, jest nazywana deklaracją najwyższego poziomu, a do nazwy najwyższego poziomu można się odwoływać w dowolnym miejscu programu. (Cóż, prawie wszędzie. Istnieją pewne ograniczenia. W szczególności zobacz późniejszą sekcję „Nazwy zaczynające się od podkreślenia”).

6. Miej dwie deklaracje zmiennych _counter — jedną na najwyższym poziomie, a drugą wewnątrz klasy _MyHomePageState. Zainicjuj _counter najwyższego poziomu na 2873, a drugi _counter na 0.

```
void main() => runApp(App0502());

class App0502 extends StatelessWidget {
  // Blah, blah, blah ...
}

class MyHomePage extends StatefulWidget {
  // Yada, yada ...
}

int _counter = 2873;

class _MyHomePageState extends State {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }

  @override
  Widget build(BuildContext context) {
    // Whatever  ...
  }
}

// What if there's more code after the
//   _MyHomePageState class declaration?
```

2873

0

2873

Przed przetestowaniem tej wersji kodu zakończ uruchamianie dowolnej innej wersji. Uruchom tę wersję kodu od nowa.

Gdy ta zmodyfikowana aplikacja zaczyna działać, liczba na środku ekranu to 0, a nie 2873. Deklaracja najwyższego poziomu _counter nie ma żadnego efektu, ponieważ jest przesłaniana przez deklarację w klasie _MyHomePageState. Deklaracja _counter w klasie _MyHomePageState dotyczy kodu wewnątrz klasy _MyHomePageState. Deklaracja _counter najwyższego poziomu ma zastosowanie wszędzie indziej w kodzie tego pliku.

Ta sekcja dotyczy klas, metod i zmiennych. Sekcja ta opisuje zmienną instancji jako zmienną, której deklaracja znajduje się wewnątrz klasy, ale nie wewnątrz żadnej z jej metod. To prawie poprawny opis zmiennej instancji. Mówiąc ściślej, deklaracja zmiennej instancji to taka, która nie zawiera słowa static @@, słowa, które napotkasz w rozdziałach 7 i 8. Dopóki nie przeczytasz rozdziałów 7 i 8, nie przejmuj się tym.

**Nazwy zaczynające się od podkreślenia**

Któregoś dnia, gdy zostaniesz wielkim programistą Fluttera, utworzysz dużą, skomplikowaną aplikację, która zawiera kilka różnych plików .dart. Instrukcje importu pliku sprawią, że kod z jednego pliku będzie dostępny do użycia w innym pliku. Ale jak to działa? Czy są jakieś ograniczenia? Rysunek 5-18 mówi wszystko.

```
one_file.dart
1  import 'package:flutter/material.dart';
2
3  void main() => runApp(MyApp());
4
5  var _number = 50;
6  var amount = 10;
7
8  class MyApp extends StatelessWidget {
9    @override
10   Widget build(BuildContext context) {
11     return MaterialApp(
```

```
another_file.dart
1  import 'one_file.dart';
2
3
4
5  var num = _number;
6  var amnt = amount;
7
```

Zmienna lub funkcja, której nazwa zaczyna się od znaku podkreślenia (_), jest lokalna dla pliku, w którym została zadeklarowana i nie można do niej odwoływać się w innych plikach .dart. Wszystkie inne nazwy można importować i udostępniać wszystkim plikom w aplikacji. Na rysunku 5.18 zmienna _number może być użyta tylko w pliku one_file.dart. Ale ze względu na instrukcję importu zmienna kwoty jest dostępna zarówno w pliku one_file.dart, jak iw innym_pliku.dart.

Jeśli jesteś przyzwyczajony do pisania kodu w językach takich jak Java, zapomnij o modyfikatorach dostępu, takich jak public i private. Język Dart nie ma tych rzeczy.

**NAZWY NAJWYŻSZEGO POZIOMU NIE ZAWSZE SĄ NAJLEPSZE**

W kroku 5 instrukcji tej sekcji deklarujesz _counter na najwyższym poziomie, a program działa bez żadnych problemów. Jeśli można zadeklarować _counter na najwyższym poziomie, dlaczego nie zrobisz tego na listingu 2? Cóż, od programu należy oczekiwać więcej niż tego, że po prostu działa poprawnie. Oprócz prawidłowego działania dobry program jest solidny.

Program nie psuje się, gdy ktoś zmieni fragment kodu. Na listingu 2 zmienna _counter jest używana tylko wewnątrz klasy _MyHomePageState. Programista pracujący nad kodem klasy _MyHomePageState powinien mieć możliwość manipulowania zmienną _counter. Ale inni programiści, którzy pracują nad innymi częściami aplikacji, nie muszą odwoływać się do zmiennej _counter. Utrzymując dostęp do _counter wewnątrz klasy _MyHomePageState, chronisz zmienną przed przypadkowym niewłaściwym użyciem przez programistów, którzy nie muszą się do niej odwoływać. (Programiści zorientowani obiektowo nazywają to enkapsulacją). Program z listingu 5.2 nie jest dużą aplikacją przemysłową. Tak więc w tym programie każdy, kto pisze kod poza klasą _MyHomePageState, prawdopodobnie wie wszystko o kodzie wewnątrz klasy _MyHomePageState. Jednak w przypadku rzeczywistych aplikacji, w których zespoły programistów pracują nad różnymi częściami kodu, ważna jest ochrona jednej części kodu przed innymi częściami. Nie, to nie jest ważne. To absolutnie niezbędne. Pamiętaj: w każdym programie, który piszesz, ogranicz dostęp do nazw zmiennych i innych nazw tak bardzo, jak to możliwe. Nie deklaruj ich na najwyższym poziomie, jeśli nie musisz. Tak jest bezpieczniej.

**Uff!**

To ciężka część. Jeśli spędziłeś wieczór na czytaniu każdego słowa, prawdopodobnie jesteś trochę zmęczony. Ale to dobrze. Weź oddech. Zrób sobie filiżankę herbaty. Usiądź w fotelu i zrelaksuj się przy wykonaniu The Well Tempered Clavier (Praeludium 1, BWV 846).

**Rozkładanie rzeczy**

Według folkloru rozmiar akwarium określa rozmiary złotych rybek w zbiorniku. Złota rybka w małym zbiorniku może mieć tylko jeden lub dwa cale długości, ale ta sama złota rybka w większym zbiorniku dorasta do dziesięciu cali długości. To tak, jakby komórki ryby wyczuwały granice przestrzeni życiowej ryby i komórki przestawały rosnąć, gdy uznają, że byłoby to niepraktyczne. Kilka zasobów internetowych mówi, że zjawisko rozmiaru czołgu jest mitem, ale to nie powstrzymuje mnie przed porównaniem go z układami Fluttera. (Nic nie powstrzymuje mnie przed porównywaniem układów Flutter.) W układzie Flutter widżety są zagnieżdżone wewnątrz innych widżetów. Widżet zewnętrzny wysyła ograniczenie do widżetu wewnętrznego: „Możesz mieć dowolną szerokość, pod warunkiem, że szerokość mieści się w przedziale od 0 do 400 pikseli niezależnych od gęstości". Później wewnętrzny widżet wysyła swoją dokładną wysokość do zewnętrznego widżetu:

„Mam szerokość 200 niezależnych od gęstości pikseli". Widżet zewnętrzny wykorzystuje te informacje do pozycjonowania widżetu wewnętrznego:

„Ponieważ masz szerokość 200 pikseli niezależnych od gęstości, ustawię twoją lewą krawędź 100 pikseli od mojej lewej krawędzi".

Oczywiście jest to uproszczona wersja prawdziwego scenariusza. Jest to jednak przydatny punkt wyjścia do zrozumienia sposobu działania układów Flutter. Co najważniejsze, ta zewnętrzna/wewnętrzna komunikacja przebiega przez cały łańcuch widżetów aplikacji. Wyobraź sobie, że masz cztery widżety. Zaczynając od najbardziej zewnętrznego widżetu (takiego jak widżet Materiał), nazwij te widżety „prababcią", „babcią", „matką" i „Elsie".

Oto jak Flutter decyduje, jak narysować te widżety:

1. Prababcia opowiada babci jak duża może być (babcia).

2. Babcia mówi mamie, jak duża może być (matka).

3. Matka mówi Elsie, jak duża może być (Elsie).

4. Elsie decyduje, jak duża jest, i mówi o tym mamie.

5. Matka określa pozycję Elsie, decyduje, jak duża ona (matka) jest, a potem mówi babci.

6. Babcia określa stanowisko matki, jak duża jest (babcia), a potem mówi prababci

7. Prababcia określa pozycję matki, a następnie decyduje, ile ona (prababka) ma wzrostu. Tak, szczegóły są niejasne. Warto jednak pamiętać o tym wzorcu, czytając o układach Fluttera.

**Wielkie zdjęcie**

Listingi 1 i 2 przedstawiają kilka koncepcji układu Fluttera, a rysunek 1 pokazuje, co widzisz, gdy uruchomisz te listingi razem.

**LISTING 1 Użyj ponownie tego kodu**

```
// App06Main.dart

import 'package:flutter/material.dart';
```

```
import 'App0602.dart'; // Change this line to
App0605, App0606, and so on.
void main() => runApp(App06Main());
class App06Main extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
home: _MyHomePage(),
);
}
}
class _MyHomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Material(
color: Colors.grey[400],
child: Padding(
padding: const EdgeInsets.symmetric(
horizontal: 20.0,
),
child: buildColumn(context),
),
);
}
}
Widget buildTitleText() {
return Text(
"My Pet Shop",
textScaleFactor: 3.0,
textAlign: TextAlign.center,
);
```

```
}
Widget buildRoundedBox(
String label, {
double height = 88.0,
}) {
return Container(
height: height,
width: 88.0,
alignment: Alignment(0.0, 0.0),
decoration: BoxDecoration(
color: Colors.white,
border: Border.all(color: Colors.black),
borderRadius: BorderRadius.all(
Radius.circular(10.0),
),
),
child: Text(
label,
textAlign: TextAlign.center,
),
);
}
```

**LISTING 2 Bardzo prosty układ**

```
// App0602.dart
import 'package:flutter/material.dart';
import 'App06Main.dart';
Widget buildColumn(BuildContext context) {
return Column(
mainAxisAlignment:
MainAxisAlignment.center,
crossAxisAlignment:
```

```
CrossAxisAlignment.stretch,

children: <Widget>[

buildTitleText(),

SizedBox(height: 20.0),

buildRoundedBox(

"Sale Today",

height: 150.0,

),

],

);

}
```

Kod z Listingu 1 odnosi się do kodu z Listingu 2 i odwrotnie. Dopóki te dwa pliki znajdują się w tym samym projekcie Android Studio, uruchomienie aplikacji z Listingu 1 automatycznie używa kodu z Listingu 2. Działa to ze względu na deklaracje import u góry każdej z list. Listingi 1 i 2 ilustrują niektóre koncepcje kodowania wraz z garścią przydatnych funkcji Fluttera.

**Tworzenie małych fragmentów kodu**

Na listach 1 i 2 tworzę niektóre widżety, wywołując metody.

```
child: buildColumn(context),

// … And elsewhere, …

Column(

// … Blah, blah, …

children: <Widget>[

buildTitleText(),

SizedBox(height: 20.0),

buildRoundedBox(

// … Etc.
```

Każde wywołanie metody zastępuje dłuższy fragment kodu - taki, który szczegółowo opisuje konkretny widżet. Tworzę te metody, ponieważ dzięki temu kod jest łatwiejszy do odczytania i strawienia. Rzut oka na Listing 2 pozwala stwierdzić, że Kolumna składa się z tekstu tytułowego, pola rozmiaru i pola zaokrąglonego. Nie znasz żadnych szczegółów, dopóki nie spojrzysz na deklaracje metod buildTitleText i buildRoundedBox na listingu 1, ale to nic. Dzięki podziałowi kodu na metody w ten sposób nie tracisz z oczu ogólnego zarysu aplikacji. W projektowaniu dobrego oprogramowania planowanie jest niezbędne. Ale czasami twoje plany się zmieniają. Wyobraź sobie następujący scenariusz: zaczynasz pisać kod, który Twoim zdaniem będzie dość prosty. Po kilku minutach (a czasem kilku godzinach) zdajesz sobie sprawę, że kod stał się duży i nieporęczny. Decydujesz się więc podzielić kod na metody.

Aby to zrobić, możesz skorzystać z jednej z przydatnych funkcji refaktoryzacji Android Studio. Oto jak to działa:

1. Zacznij od wywołania konstruktora, które chcesz zastąpić własnym wywołaniem metody.

Na przykład chcesz zastąpić wywołanie konstruktora Text w następującym fragmencie kodu:

```
children: <Widget>[

Text(

"My Pet Shop",

textScaleFactor: 3.0,

textAlign: TextAlign.center,

),

SizedBox(height: 20.0),
```

2. Umieść kursor myszy na nazwie wywołania konstruktora. W przypadku fragmentu w kroku 1 kliknij słowo Tekst.

3. W menu głównym Android Studio wybierz Refactor ⇒ Extract ⇒ Method. W rezultacie Android Studio wyświetla okno dialogowe Metoda wyodrębniania.

4. W oknie dialogowym Extract Method wpisz nazwę nowej metody. Dla konstruktora o nazwie Text Android Studio sugeruje nazwę metody buildText. Ale aby utworzyć Listingi 6-1 i 6-2, wymyśliłem nazwę buildTitleText.

5. W oknie dialogowym Metoda wyodrębniania naciśnij Refaktor. Jak za dotknięciem czarodziejskiej różdżki, Android Studio dodaje do Twojego kodu nową deklarację metody i zastępuje oryginalny konstruktor widgetów wywołaniem metody. Typ zwracany przez nową metodę to rodzaj widżetu, który próbuje skonstruować twój kod. Na przykład, zaczynając od kodu w kroku 1, pierwsze dwa wiersze metody mogą wyglądać tak:

```
Text buildTitleText() {

return Text(
```

6. Zrób sobie przysługę i zmień typ w nagłówku metody na Widget.

```
Widget buildTitleText() {

return Text(
```

Każda instancja klasy Text jest instancją klasy Widget, więc ta zmiana nie wyrządza żadnej szkody. Ponadto zmiana dodaje odrobinę elastyczności, która może ostatecznie zaoszczędzić trochę energii psychicznej. Może później zdecydujesz się otoczyć widżet Tekst metody widżetem Centrum.

```
// Baby, you're no good . . .

Text buildTitleText() {

return Center(

child: Text(
```

Po wprowadzeniu tej zmiany kod jest błędny, ponieważ zwracany typ nagłówka jest nieprawidłowy. Tak, każda instancja klasy Text jest instancją klasy Widget. Ale nie, instancja klasy Center nie jest instancją klasy Text. Twoja metoda zwraca instancję Center, ale nagłówek metody oczekuje, że metoda zwróci instancję Text. Czy nie chciałbyś zmienić pierwszego słowa w nagłówku na Widget? Zrób to raczej wcześniej niż później. W ten sposób nie będziesz rozpraszany, gdy będziesz koncentrować się na wprowadzaniu zmian w ciele metody.

**Tworzenie listy parametrów**

Na listingu 1 nagłówek deklaracji buildRoundedBox wygląda następująco:

Widget buildRoundedBox(

String label, {

double height = 88.0,

})

Metoda ma dwa parametry: label i height.

* Parametr label jest parametrem pozycyjnym. Jest to parametr pozycyjny, ponieważ nie jest otoczony nawiasami klamrowymi. W nagłówku wszystkie parametry pozycyjne muszą znajdować się przed dowolnym z wymienionych parametrów.

* Parametr height jest parametrem nazwanym. Jest to nazwany parametr, ponieważ jest otoczony nawiasami klamrowymi. W wywołaniu tej metody można pominąć parametr wysokości. Gdy to zrobisz, domyślna wartość parametru to 88,0.

Mając na uwadze te fakty, następujące wywołania funkcji buildRoundedBox są poprawne:

buildRoundedBox( // Flutter style

guidelines recommend having a

"Flutter", // trailing

comma at the end of every list.

height: 1000.0, // It's the

comma after the height parameter.

)

buildRoundedBox("Flutter") // In the method

header, the height parameter

// has the

default value 88.0.

Oto niektóre połączenia, które nie są prawidłowe:

buildRoundedBox( // In a function

call, all positional parameters

height: 1000.0, // must come

before any named parameters.

"Flutter",

)

buildRoundedBox(

label: "Flutter", // The label

parameter is a positional parameter,

height: 1000.0, // not a named

parameter.

)

buildRoundedBox( // The height

parameter is a named parameter,

"Flutter", // not a

positional parameter.

1000.0,

)

buildRoundedBox() // You can't omit

the label parameter, because

// the label

parameter has no default value.

Na listingu 2 deklaracja buildColumn zawiera parametr BuildContext. Możesz zapytać: „Jaki jest pożytek z tego parametru BuildContext? Treść metody buildColumn nie odnosi się do wartości tego parametru”.

**Żywy kolor**

Część 5 przedstawia klasę Colours Fluttera z podstawowymi rzeczami, takimi jak Colors.grey i Colors.black. W rzeczywistości klasa Colors zapewnia 12 różnych odcieni szarości, 7 odcieni czerni, 28 odcieni niebieskiego i podobną różnorodność dla innych kolorów. Na przykład odcienie szarości noszą nazwy Kolory.szary[50] (najjaśniejszy), Kolory.szary[100], Kolory.szary[200], Kolory.szary[300] i tak dalej, aż do Kolory.szary [900] (najciemniejszy). Nie możesz umieszczać dowolnych liczb w nawiasach, więc rzeczy takie jak Colors.grey[101] i Colors.grey[350] po prostu nie istnieją. Ale jeden odcień — Colors.grey[500] — jest wyjątkowy. Możesz skrócić Colors.grey[500], pisząc Colors.grey bez liczby w nawiasach. Jeśli chcesz mieć bardzo precyzyjną kontrolę nad wyglądem swojej aplikacji, możesz użyć konstruktora Color.fromRGBO firmy Flutter. (To kolor w liczbie pojedynczej, w przeciwieństwie do liczby mnogiej kolorów). Litery RGBO oznaczają czerwony, zielony, niebieski i krycie. W konstruktorze wartości Red, Green i Blue mieszczą się w zakresie od 0 do 255, a wartość Opacity mieści się w zakresie

od 0,0 do 1,0. Na przykład Color.fromRGBO(255, 0, 0, 1.0) oznacza całkowicie nieprzezroczystą czerwień. Tabela 1 zawiera kilka innych przykładów:

**Lista parametrów: Co oznacza lista parametrów**

(0, 255, 0, 1,0) : Zielony

(0, 0, 255, 1,0) : Niebieski

(255, 0, 255, 1,0) : Fioletowy (równe ilości czerwieni i błękitu)

(0, 0, 0, 1.0) : Czarny

(255, 255, 255, 1,0) : Biały

(190, 190, 190, 1.0) : szary (około 75% bieli)

(255, 0, 0, 0,5) : 50% przezroczystości Czerwony

(255, 0, 0, 0.0) : Nic (całkowita przezroczystość, niezależnie od wartości czerwonego, zielonego i niebieskiego)

**Dodanie wypełnienia**

Widżet Padding Fluttera umieszcza pustą przestrzeń między jego najbardziej zewnętrzną krawędzią a jego dzieckiem. Na listingu 1 kod

```
Padding(

padding: const EdgeInsets.symmetric(

horizontal: 20.0,

),

child: buildColumn(context),
```

otacza wywołanie buildColumn 20,0 jednostkami pustej przestrzeni po lewej i prawej stronie. Bez wypełnienia kolumna dotykałaby lewej i prawej krawędzi ekranu użytkownika, podobnie jak białe pole Wyprzedaż dzisiaj wewnątrz kolumny. To nie wyglądałoby ładnie. We Flutterze linia taka jak pozioma: 20,0 oznacza 20,0 pikseli niezależnych od gęstości. Piksel niezależny od gęstości (dp) nie ma ustalonego rozmiaru. Zamiast tego rozmiar piksela niezależnego od gęstości zależy od sprzętu użytkownika. W szczególności każdy cal ekranu użytkownika ma około 96 dp długości. Oznacza to, że każdy centymetr ma długość około 38 pikseli. Według oficjalnej dokumentacji Fluttera zasada dotycząca 96 dp na cal „może być niedokładna, czasem ze znacznym marginesem". Uruchom aplikację z tej sekcji na swoim telefonie, a zobaczysz, co one oznaczają. We Flutter opisujesz wszelkiego rodzaju dopełnienie, konstruując obiekt EdgeInsets. Konstruktor EdgeInsets.symmetric z listingu 1 ma jeden parametr — parametr poziomy. Oprócz parametru poziomego konstruktor EdgeInsets.symmetric może mieć parametr pionowy, na przykład:

```
Padding(

padding: const EdgeInsets.symmetric(

horizontal: 20.0,

vertical: 10.0,
```

)

Parametr pionowy dodaje puste miejsce na górze i na dole widżetu podrzędnego. Tabela 2 zawiera listę alternatyw dla konstruktora EdgeInsets.symmetric.

**Wywołanie konstruktora**

**Ile pustej przestrzeni otacza widżet podrzędny**

EdgeInsets.all(20.0)

20.0 dp on all four sides

EdgeInsets.only(

left: 15.0,

top: 10.0,

)

15.0 dp on the left

10.0 dp on top

EdgeInsets.only(

top: 10.0,

right: 15.0,

bottom: 15.0,

)

10.0 dp on top

15.0 dp on the right

15.0 dp on the bottom

EdgeInsets.fromLTRB(

5.0,

10.0,

3.0,

2.0,

)

5.0 dp on the left

10.0 dp on top

3.0 dp on the right

2.0 dp on the bottom

Kiedy zaczynałem pracować nad kodem z Listingu 6-1, nie było w nim widżetu Wypełnienie. Wywołanie buildColumn było bezpośrednim następcą widżetu Material:

```
return Material(

color: Colors.grey[400],

child: buildColumn(context),

);
```

Użyłem sztuczki Alt+Enter z rozdziału 3, aby otoczyć wywołanie buildColumn nowym widżetem Padding. Kiedy to zrobiłem, Android Studio dodało również własny kod const EdgeInsets. Pomajstrowałem trochę przy kodzie Android Studio, ale nie usunąłem słowa kluczowego const kodu. Aby zapoznać się z wewnętrzną historią słowa kluczowego const Darta

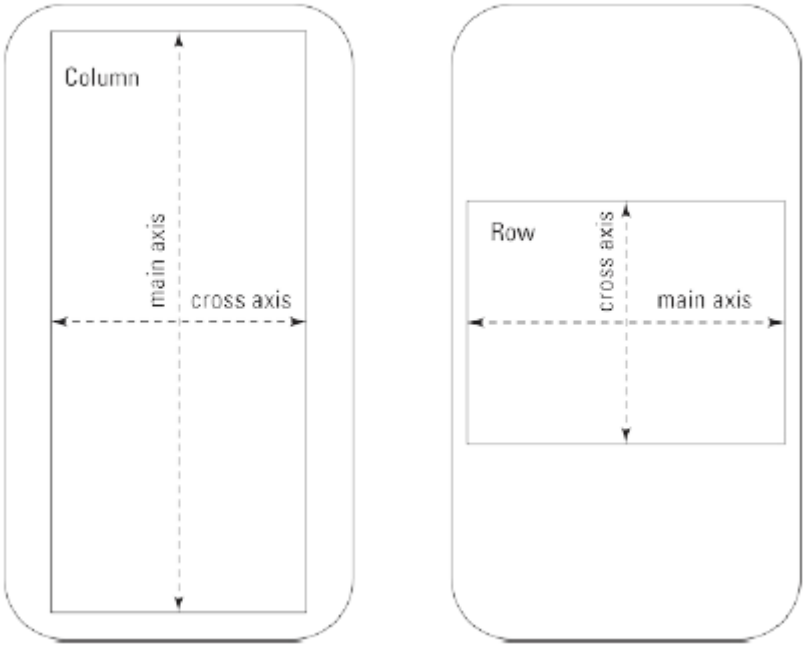
Widżet Paddding dodaje puste miejsce w sobie

**Twój pokorny sługa, widżet Column**

Pomyśl o tym: bez widżetu Flutter Column nie byłbyś w stanie umieścić jednego widżetu nad drugim. Wszystko na ekranie użytkownika zostałoby zgniecione w jednym miejscu. Ekran byłby nieczytelny i nikt nie używałby Fluttera. Nie czytałbyś tej książki. Nie zarabiałbym żadnych tantiem. Cóż to byłby za okropny świat! Widżet Kolumna na listingu 6.2 ma dwie właściwości związane z wyrównaniem:

```
Column(

mainAxisAlignment: MainAxisAlignment.center,

crossAxisAlignment:

CrossAxisAlignment.stretch,

// … And so on.
```

Właściwość mainAxisAlignment została omówiona w Części 3. Opisuje sposób, w jaki elementy podrzędne są umieszczane od góry do dołu kolumny. Dzięki MainAxisAlignment.center dzieci gromadzą się mniej więcej w połowie wysokości od góry ekranu. W przeciwieństwie do tego, crossAxisAlignment opisuje, w jaki sposób dzieci są rozmieszczone z boku na bok w kolumnie.




CrossAxisAlignment kolumny może mieć duże znaczenie w sposobie, w jaki elementy podrzędne kolumny pojawiają się na ekranie. Na przykład, jeśli skomentujesz wiersz crossAxisAlignment na listingu 2, zobaczysz ekran pokazany na rysunku .


Na listingu 2 wartość CrossAxisAlignment.stretch mówi kolumnie, że jej elementy podrzędne powinny wypełnić całą oś poprzeczną. Oznacza to, że niezależnie od jawnych wartości szerokości elementów podrzędnych elementy podrzędne zmniejszają się lub poszerzają, tak że przebiegają przez całą kolumnę. Jeśli mi nie wierzysz, wypróbuj następujący eksperyment:

1. Uruchom kod z Listingu 1. Użyj symulatora iPhone'a, emulatora Androida lub prawdziwego fizycznego telefonu. Zacznij od ustawienia urządzenia w trybie pionowym, jak na rysunku 1.

2. Obróć urządzenie na bok, aby ustawić je w trybie poziomym.

Jeśli korzystasz z urządzenia wirtualnego, naciśnij klawisze Commandstrzałka w prawo (na komputerze Mac) lub Ctrl+strzałka w prawo (w systemie Windows). Jeśli używasz fizycznego urządzenia, obróć to cholerstwo na bok.

3. Obserwuj zmianę rozmiaru pola Dzisiejsza wyprzedaż. Bez względu na to, jak szeroki jest ekran, pole Wyprzedaż dzisiaj rozciąga się prawie na całą szerokość. Ustawienie width: 88.0 z Listingu 1 nie ma żadnego wpływu.

Więcej informacji na temat wyrównania osi można znaleźć w poniższych sekcjach.

Gdy obrócisz urządzenie na bok, urządzenie może nie przełączać się między trybami pionowym i poziomym. Dotyczy to zarówno urządzeń fizycznych (prawdziwe telefony i tablety), jak i urządzeń wirtualnych (emulatory i symulatory). Jeśli orientacja urządzenia nie chce się zmienić, spróbuj tego:

* Na urządzeniu z Androidem w Ustawieniach ⇒ Wyświetlacz włącz Automatyczne obracanie ekranu.

* Na iPhonie lub iPadzie przesuń palcem w górę od dołu ekranu i naciśnij przycisk z kłódką i okrągłą strzałką.

Za pomocą emulatora lub symulatora możesz spróbować obrócić monitor komputera na boki, ale to prawdopodobnie nie zadziała.

**Widżet SizeBox**

Gdybym planował zamieszkać na bezludnej wyspie i mógłbym zabrać ze sobą tylko siedem widżetów, te siedem widżetów to Kolumna, Wiersz, SizedBox, Kontener, Rozszerzony, Odstępnik i Wypełnienie. (Gdybym mógł zabrać ze sobą tylko dwa rodzaje jedzenia, byłyby to cheeseburgery i czekolada). SizedBox to prostokąt używany przez programistów do zajmowania miejsca. SizedBox ma szerokość, wysokość i prawdopodobnie dziecko. Bardzo często liczy się tylko szerokość lub wysokość. Listing 6-2 zawiera SizedBox o wysokości 20,0 umieszczony pomiędzy tekstem tytułowym a zaokrąglonym polem. Bez SizeBox nie byłoby spacji między tekstem tytułu a zaokrąglonym polem.

Spacer jest jak SizedBox, z tą różnicą, że Spacer używa flex zamiast jawnych parametrów wysokości i szerokości.

**Twój przyjaciel, widżet kontenera**

Na listingu 2 pole wyświetlające słowa Sale Today wykorzystuje widżet Kontener. Kontener to widżet zawierający -coś. (Nie jest to zaskakujące.) Chociaż widżet coś zawiera, ma właściwości, takie jak wysokość, szerokość, wyrównanie, dekoracja, dopełnienie i margines.

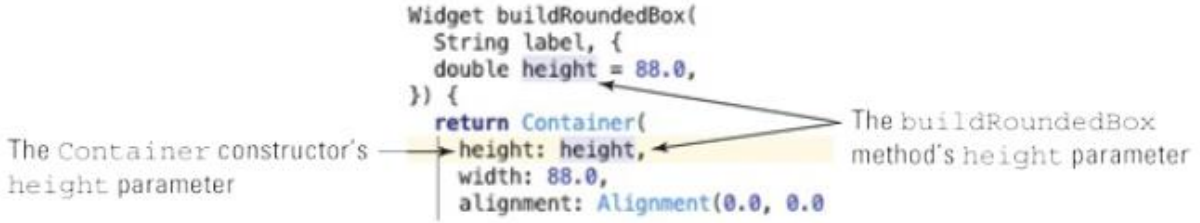
**Parametry wysokości i szerokości**

Być może zainteresuje Cię konkretny wiersz z Listingu 1:

return Container(

height: height,

Co może oznaczać height:height ? Wysokość jest jaka jest? Wysokość to wysokość to wysokość? Aby dowiedzieć się, co się dzieje, umieść kursor na drugim wystąpieniu słowa wysokość — tym po dwukropku. Kiedy to zrobisz, Android Studio podświetli to zdarzenie wraz z jednym innym.



Zauważalnie nieobecne jest jakiekolwiek podświetlenie na wysokości, która znajduje się bezpośrednio przed dwukropkiem. Listing 6-1 zawiera dwie zmienne o nazwie wysokość. Jednym z nich jest parametr buildRoundedBox; drugi jest parametrem konstruktora kontenera. Linia

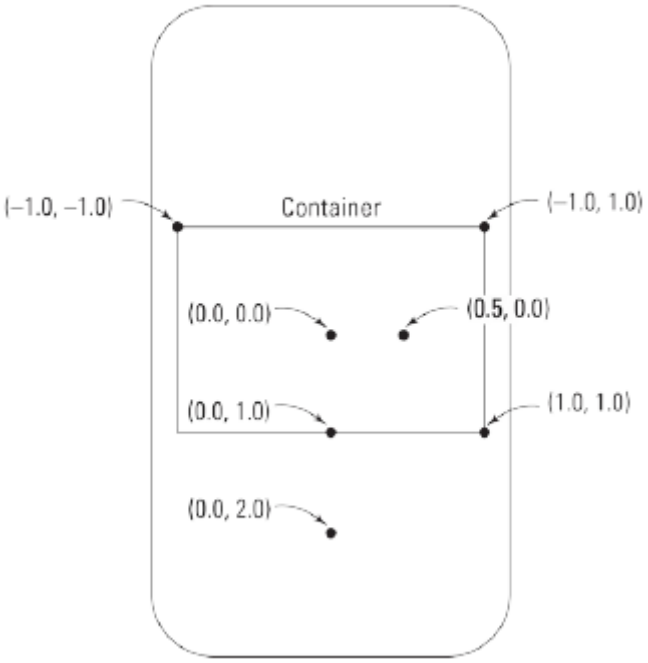
height: height,

sprawia, że parametr Container ma taką samą wartość jak parametr buildRoundedBox. (Parametr buildRoundedBox pobiera swoją wartość z wywołania z listingu 2.)

W wywołaniu konstruktora kontenera parametry wysokości i szerokości są sugestiami, a nie wielkościami bezwzględnymi.

**Parametr wyrównania**

Aby wyrównać element podrzędny w widgecie Kontener, nie używa się funkcji mainAxisAlignment ani crossAxisAlignment. Zamiast tego używasz zwykłego starego parametru wyrównania. Na Listingu 1 wyrównanie linii: Alignment(0.0, 0.0) mówi Flutterowi, aby umieścił element potomny kontenera na środku kontenera. Rysunek 5 ilustruje sekrety klasy Alignment.




**Parametr dekoracji**

Jak sama nazwa wskazuje, dekoracja to coś, co ożywia nudny widżet. Na listingu 1 konstruktor BoxDecoration ma trzy własne parametry:

* color: Kolor wypełnienia widżetu. Ta właściwość wypełnia pole Sprzedaż dzisiaj na rysunku 1 kolorem białym.

Konstruktory Container i BoxDecoration mają parametry koloru. Kiedy umieścisz BoxDecoration wewnątrz kontenera, możesz mieć parametr koloru dla BoxDecoration, a nie dla Container. Jeśli masz oba, program może ulec awarii.

* border: kontur otaczający widżet. Listing 1 używa konstruktora Border.all, który opisuje obramowanie ze wszystkich czterech stron pudełka Sale Today.

Aby utworzyć obramowanie, którego boki nie są takie same, użyj konstruktora Border Fluttera (bez części .all). Oto przykład:

Border(

top: BorderSide(width: 5.0, color:

Colors.black),

bottom: BorderSide(width: 5.0, color:

Colors.black),

left: BorderSide(width: 3.0, color:

Colors.blue),

right: BorderSide(width: 3.0, color:

Colors.blue),

)

* border: kontur otaczający widżet. Listing 1 używa konstruktora Border.all, który opisuje obramowanie ze wszystkich czterech stron pudełka Sale Today.

Aby utworzyć obramowanie, którego boki nie są takie same, użyj konstruktora Border Fluttera (bez części .all). Oto przykład:

Border(

top: BorderSide(width: 5.0, color:

Colors.black),

bottom: BorderSide(width: 5.0, color:

Colors.black),

left: BorderSide(width: 3.0, color:

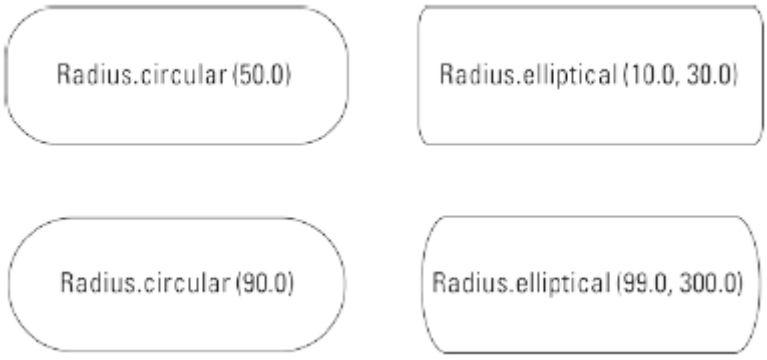
Colors.blue),

right: BorderSide(width: 3.0, color:

Colors.blue),

)

* borderRadius: Wielkość krzywizny obramowania widżetu. Rysunek 6 pokazuje, co się stanie, gdy użyjesz różnych wartości parametru borderRadius.



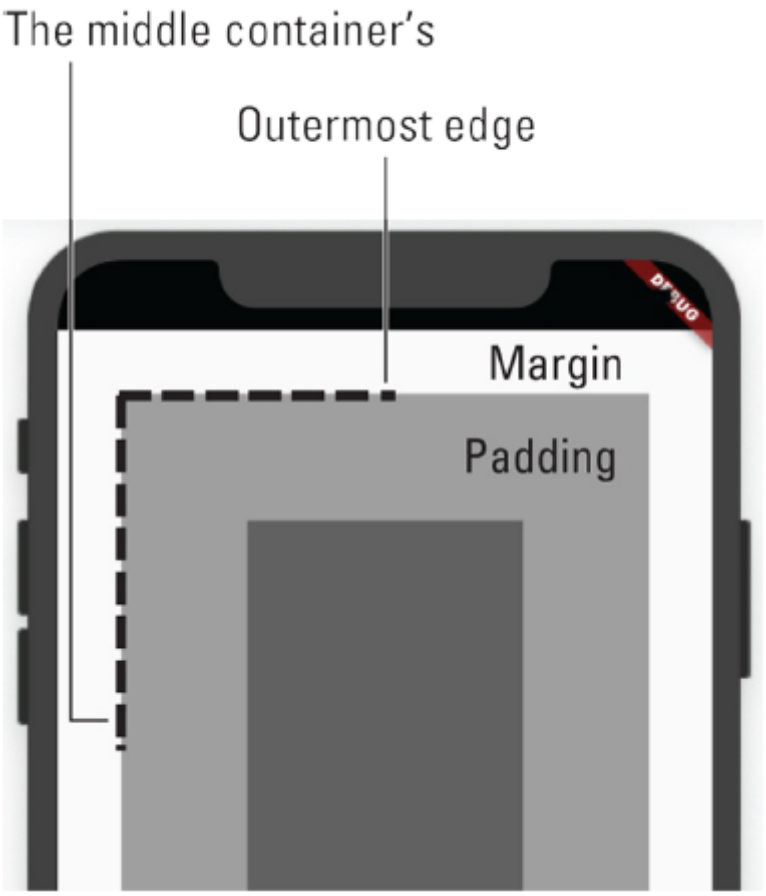

**Parametry dopełnienia i marginesu**

Wywołanie konstruktora kontenera na listingu 1 nie ma parametrów dopełnienia ani marginesu, ale dopełnienie i margines mogą być przydatne w innych ustawieniach. Aby dowiedzieć się, jak działają dopełnienia i marginesy, spójrz najpierw na Listing 3.

**LISTING 3 Bez wypełnienia i marginesu**

```
// App0603.dart

import 'package:flutter/material.dart';

void main() => runApp(App0602());

class App0602 extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Material(

color: Colors.grey[50],

child: Container(

color: Colors.grey[500],

child: Container(

color: Colors.grey[700],

),

),

),

);

}

}
```

Listing 3 zawiera kontener w innym kontenerze, który znajduje się w widgecie Material. Wewnętrzny pojemnik jest szary [700], który jest dość ciemnoszary. Zewnętrzny pojemnik jest jaśniejszy, a tło widżetu Materiał jest szare[50], czyli prawie białe. Powiedziałem mojemu redaktorowi, że chcę wykorzystać miejsce na stronie z rysunkiem poświęconym serii Listingu 3, ale odmówił. Zastanawiam się dlaczego! Kto mógłby sprzeciwić się figurze, która jest niczym innym jak ciemnoszarym prostokątem? Gdy uruchomisz aplikację z listingu 3, wewnętrzny kontener całkowicie zakryje zewnętrzny kontener, który z kolei całkowicie zakryje widżet Material. Każdy z tych widżetów rozszerza się, aby wypełnić swojego rodzica, więc każdy z trzech widżetów zajmuje cały ekran. Jedynym widocznym widżetem jest najbardziej wewnętrzny, ciemnoszary pojemnik. Co za strata! Aby zaradzić tej sytuacji, w listingu 4 zastosowano zarówno dopełnienie, jak i margines. Rysunek 7 pokazuje wynik.




Listing 4 dotyczy środkowego pojemnika - tego, którego kolor to średni odcień szarości. Zaznaczyłem rysunek 7, aby wynik był krystalicznie czysty. Ogólne zasady są następujące:

* Wypełnienie to przestrzeń między najbardziej zewnętrznymi krawędziami widżetu a elementem potomnym widżetu. Na rysunku 7 średnioszary materiał to wyściółka.

* Margines to przestrzeń między najbardziej zewnętrznymi krawędziami widżetu a jego rodzicem. Na rysunku 7 biały (lub prawie biały) materiał to margines. Z tego, co zaobserwowałem, programiści Fluttera często używają dopełnienia, ale oszczędnie używają marginesu.

Możesz dodać wypełnienie do prawie każdego widżetu bez umieszczania tego widżetu w kontenerze. Aby to zrobić, po prostu umieść widżet wewnątrz widżetu Wypełnienie.

Kiedy myślisz o urządzeniu mobilnym, prawdopodobnie wyobrażasz sobie prostokątny ekran. Czy to oznacza, że aplikacja może używać całego prostokąta? Tak nie jest. Górna część prostokąta może mieć wycięcie. Rogi prostokąta mogą być zaokrąglone zamiast kwadratu. System operacyjny (iOS lub Android) może zużywać części ekranu z paskiem akcji lub innymi śmieciami. Aby uniknąć przedmiotów na tym torze przeszkód, Flutter ma widżet SafeArea. SafeArea to część ekranu, z której aplikacja może bezpłatnie korzystać bez żadnych ograniczeń. Na listingu 6.4 SafeArea pomaga mi pokazać dopełnienie i margines w całej okazałości. Bez SafeArea górna część marginesu może być zakryta przez rzeczy, które nie są częścią mojej aplikacji.

**Zagnieżdżanie wierszy i kolumn**

Prawie nigdy nie widzisz aplikacji z tylko jedną kolumną widżetów. W większości przypadków widzisz widżety obok innych widżetów, widżety ułożone w siatki, widżety ustawione pod kątem do innych widżetów i tak dalej. Najprostszym sposobem rozmieszczenia widżetów Flutter jest umieszczenie kolumn wewnątrz wierszy, a wierszy wewnątrz kolumn. Listing 5 zawiera przykład, a rysunek8 przedstawia wyniki.

**LISTING 5 Wiersz w kolumnie**

```
// App0605.dart

import 'package:flutter/material.dart';

import 'App06Main.dart';

Widget buildColumn(BuildContext context) {

return Column(

mainAxisAlignment:

MainAxisAlignment.center,

crossAxisAlignment:

CrossAxisAlignment.stretch,

children: <Widget>[

buildTitleText(),

SizedBox(height: 20.0),

_buildRowOfThree(),

],

);

}

Widget _buildRowOfThree() {

return Row(

mainAxisAlignment:

MainAxisAlignment.spaceBetween,
```

```
children: <Widget>[
buildRoundedBox("Cat"),
buildRoundedBox("Dog"),
buildRoundedBox("Ape"),
],
);
}
```


Na listingu 1 właściwość crossAxisAlignment widżetu Kolumna wymusza, aby pole Sprzedaż dzisiaj było tak szerokie, jak to tylko możliwe. Dzieje się tak, ponieważ pole Wyprzedaż dzisiaj jest jednym z elementów podrzędnych widżetu Kolumna. Jednak na listingu 5 pola Kot, Pies i Małpa nie są elementami podrzędnymi widżetu Kolumna. Zamiast tego są wnukami widżetu Kolumna. Tak więc na listingu 5 głównym czynnikiem ustalającym położenie pól Kot, Pies i Małpa jest właściwość mainAxisAlignment widżetu Wiersz. Aby zobaczyć to w akcji, zmień linie

```
return Row(
mainAxisAlignment:
MainAxisAlignment.spaceBetween,
```

w Listingu 6-5 do następujących wierszy:

```
return Row(
mainAxisAlignment: MainAxisAlignment.center
```

Gdy to zrobisz, zobaczysz układ pokazany na rysunku 9.


Więcej poziomów zagnieżdżania

W każdym worku było siedem kotów,

Każdy kot miał siedem kociąt…

**Z TRADYCYJNEGO JĘZYKA ANGIELSKIEGO RYMOWANKA**

Tak, możesz utworzyć wiersz w kolumnie w wierszu w kolumnie w wierszu. Można tak wymieniać bardzo długo. Ta sekcja zawiera dwa skromne przykłady. Pierwszy przykład (Listing 6) zawiera rząd pól z napisami.

**LISTING 6 (Czy ta lista ma trzy podpisy?)**

```
// App0606.dart

import 'package:flutter/material.dart';

import 'App06Main.dart';

Widget buildColumn(BuildContext context) {

return Column(

mainAxisAlignment:

MainAxisAlignment.center,

crossAxisAlignment:

CrossAxisAlignment.stretch,

children: <Widget>[

buildTitleText(),

SizedBox(height: 20.0),

_buildCaptionedRow(),

],

);

}

Widget _buildCaptionedRow() {
```

```
return Row(
mainAxisAlignment:
MainAxisAlignment.spaceBetween,
children: <Widget>[
_buildCaptionedItem(
"Cat",
caption: "Meow",
),
_buildCaptionedItem(
"Dog",
caption: "Woof",
),
_buildCaptionedItem(
"Ape",
caption: "Chatter",
),
],
);
}
Column _buildCaptionedItem(String label,
{String caption}) {
return Column(
children: <Widget>[
buildRoundedBox(label),
SizedBox(
height: 5.0,
),
Text(
caption,
textScaleFactor: 1.25,
),
```

```
],
);
}
```

Rysunek 10 przedstawia przebieg kodu z Listingu 6


Następny przykład, Listing 7, robi coś nieco innego. Na listingu 7 dwa pudełka współdzielą miejsce, w którym mogłoby znajdować się jedno pudełko.

**LISTING 7 Więcej zagnieżdżania widżetów**

```
// App0607.dart
import 'package:flutter/material.dart';
import 'App06Main.dart';
Widget buildColumn(BuildContext context) {
return Column(
mainAxisAlignment:
MainAxisAlignment.center,
crossAxisAlignment:
CrossAxisAlignment.stretch,
children: <Widget>[
buildTitleText(),
SizedBox(height: 20.0),
_buildColumnWithinRow(),
],
);
}
Widget _buildColumnWithinRow() {
```

```
return Row(
mainAxisAlignment:
MainAxisAlignment.spaceBetween,
children: <Widget>[
buildRoundedBox("Cat"),
SizedBox(width: 20.0),
buildRoundedBox("Dog"),
SizedBox(width: 20.0),
Column(
children: <Widget>[
buildRoundedBox(
"Big ox",
height: 36.0,
),
SizedBox(height: 16.0),
buildRoundedBox(
"Small ox",
height: 36.0,
),
],
),
],
);
}
```

Rysunek 11 przedstawia przebieg kodu z Listingu 7.


**Korzystanie z rozszerzonego widżetu**

Zacznij od kodu z listingu 5 i dodaj do wiersza jeszcze dwa pola:

```
Widget _buildRowOfThree() {
return Row(
mainAxisAlignment:
MainAxisAlignment.spaceBetween,
children: <Widget>[
buildRoundedBox("Cat"),
buildRoundedBox("Dog"),
buildRoundedBox("Ape"),
buildRoundedBox("Ox"),
buildRoundedBox("Gnu"),
],
);
}
```

Tak, nazwa metody to nadal _buildRowOfThree. Jeśli przeszkadza ci nazwa, możesz albo zmienić nazwę, albo Google the Hitchhiker's Guide to the Galaxy trylogia. kiedy uruchomisz ten zmodyfikowany kod na niezbyt dużym telefonie w trybie portretowym, zobaczysz brzydki wyświetlacz na rysunku 6.12. (Jeśli Twój telefon jest zbyt duży, aby zobaczyć brzydotę, dodaj więcej wywołań buildRoundedBox.)


Segment po prawej stronie rysunku 12 (rzecz, która wygląda jak taśma barykadowa) wskazuje na przepełnienie. Mówiąc z grubsza, stworzyłeś blivit. Rząd stara się być szerszy niż ekran telefonu. Spójrz w górę okna narzędzia Uruchom w Android Studio, a zobaczysz następujący komunikat:

A RenderFlex overflowed by 67 pixels on the right.

**Co jeszcze nowego?**

Gdy ustawisz zbyt wiele pól obok siebie, ekran staje się przepełniony. To nie jest zaskakujące. Ale niektóre sytuacje związane z układem nie są tak oczywiste. Możesz natknąć się na problem z przepełnieniem, kiedy najmniej się tego spodziewasz. Co możesz zrobić, gdy Twoja aplikacja jest przepełniona? Oto nieszablonowa sugestia: powiedz każdemu z pudełek, aby się rozszerzyło. (Przeczytałeś to poprawnie: powiedz im, żeby rozwinęli!) Listing 8 zawiera kod, a rysunek 13 przedstawia wyniki.

```
// App0608.dart

import 'package:flutter/material.dart';

import 'App06Main.dart';

Widget buildColumn(BuildContext context) {

return Column(

mainAxisAlignment:

MainAxisAlignment.center,

crossAxisAlignment:

CrossAxisAlignment.stretch,

children: <Widget>[

buildTitleText(),

SizedBox(height: 20.0),

_buildRowOfFive(),

],

);
```

```
}
Widget _buildRowOfFive() {
return Row(
mainAxisAlignment:
MainAxisAlignment.spaceBetween,
children: <Widget>[
_buildExpandedBox("Cat"),
_buildExpandedBox("Dog"),
_buildExpandedBox("Ape"),
_buildExpandedBox("Ox"),
_buildExpandedBox("Gnu"),
],
);
}
Widget _buildExpandedBox(
String label, {
double height = 88.0,
}) {
return Expanded(
child: buildRoundedBox(
label,
height: height,
),
);
}
```


Widżet rozszerzający element podrzędny wiersza, kolumny lub elementu elastycznego w taki sposób, aby element podrzędny wypełniał dostępne miejsce. Korzystanie z widżetu Expanded powoduje, że element podrzędny wiersza, kolumny lub elementu elastycznego rozszerza się, wypełniając dostępne miejsce wzdłuż głównej osi (poziomo dla wiersza lub pionowo dla kolumny). W przypadku rozwinięcia wielu elementów podrzędnych dostępna przestrzeń jest dzielona między nie zgodnie ze współczynnikiem elastyczności.

Pomimo swojej nazwy widżet Expanded niekoniecznie powiększa jego dziecko. Zamiast tego widżet Expanded sprawia, że jego element potomny wypełnia dostępne miejsce wraz z innymi widżetami, które konkurują o to miejsce. Jeśli ta dostępna przestrzeń różni się od jawnej wartości wysokości lub szerokości kodu, niech tak będzie. Listing 8 dziedziczy linię

width: 88.0,

aby opisać szerokość każdego zaokrąglonego pudełka. Jednak na rysunku 13 żaden z prostokątów nie ma szerokości 88,0 dp. Kiedy uruchamiam aplikację na iPhonie 11 Pro Max, każde pudełko ma tylko 74,8 dp szerokości.

**Rozszerzony kontra nierozszerzony**

Kod z poprzedniej sekcji otacza każde pole wiersza widżetem Expanded. W tej sekcji Listing 9 pokazuje, co się stanie, gdy użyjesz Expanded oszczędniej.

**LISTING 9 Rozwijanie jednego z trzech widżetów**

```
// App0609.dart

import 'package:flutter/material.dart';

import 'App06Main.dart';

Widget buildColumn(BuildContext context) {

return Column(

mainAxisAlignment:

MainAxisAlignment.center,

crossAxisAlignment:

CrossAxisAlignment.stretch,

children: <Widget>[
```

```
buildTitleText(),
SizedBox(height: 20.0),
_buildRowOfThree(),
],
);
}
Widget _buildRowOfThree() {
return Row(
mainAxisAlignment:
MainAxisAlignment.spaceBetween,
children: <Widget>[
buildRoundedBox(
"Giraffe",
height: 150.0,
),
SizedBox(width: 10.0),
buildRoundedBox(
"Wombat",
height: 36.0,
),
SizedBox(width: 10.0),
_buildExpandedBox(
"Store Manager",
height: 36.0,
),
],
);
}
Widget _buildExpandedBox(
String label, {
double height = 88.0,
```

```
}) {
return Expanded(
child: buildRoundedBox(
label,
height: height,
),
);
}
```

Kod z listingu 9 otacza tylko jedno pole — okno Store Manager — z rozwiniętym widżetem. Oto, co się dzieje:

* Kod uzyskuje szerokość: 88,0 z metody buildRoundedBox z Listingu 1, więc pola Giraffe i Wombat mają po 88,0 dp szerokości.

* Dwa widżety SizedBox mają szerokość 10,0 dp każdy. Jak dotąd suma wynosi 196,0 dp.

* Ponieważ okno Menedżera sklepu znajduje się wewnątrz rozwiniętego widżetu, pozostała szerokość ekranu trafia do pola Menedżera sklepu.


Użycie widgetu Expanded wpływa na rozmiar widżetu wzdłuż głównej osi jego rodzica, ale nie wzdłuż osi poprzecznej jego rodzica. Tak więc na rysunku 14 pudełko Store Manager rośnie z boku na bok (wzdłuż głównej osi rzędu), ale nie rośnie z góry na dół (wzdłuż osi poprzecznej rzędu). W rzeczywistości tylko liczby 150.0, 36.0 i 36.0 w metodzie _buildRowOfThree (patrz Listing 9) mają jakikolwiek wpływ na wysokość pudełek. Po odrobinie poprawek kod z listingu 9 może dostarczyć więcej dowodów na to, że widżet Expanded niekoniecznie jest dużym widżetem. Wypróbuj te dwa eksperymenty:

1. Uruchom ponownie kod z Listingu 1 i 9. Ale w deklaracji metody buildRoundedBox zmień szerokość: 88,0 na szerokość: 130,0. W moim symulatorze iPhone'a 11 Pro Max szerokości pól żyrafy i wombata wynoszą po 130,0 dp. Ale szerokość pola Expanded Store Manager wynosi tylko 94,0 dp. Pudełka z żyrafą i wombatem są dość duże. Tak więc, gdy pudełko Menedżera sklepu wypełnia pozostałą dostępną przestrzeń, ta przestrzeń ma tylko 94,0 dp szerokości.


2. W deklaracji metody buildRoundedBox zmień szerokość z wartości w kroku 1 (szerokość: 130,0) na szerokość: 180,0. Ponieważ pudełka Giraffe i Wombat oraz widżety SizedBox zajmują 380,0 dp, w moim symulatorze iPhone'a 11 Pro Max nie ma już miejsca na pudełko Store Manager. Niestety! Widzę czarno-żółty pasek, wskazujący na przepełnienie RenderBox. Widżet Expanded nie jest cudotwórcą. Nie pomaga rozwiązać każdego problemu.


**Rozbudowany widżet ratuje sytuację**

Listingi 10 i 11 ilustrują nieprzyjemną sytuację, która może powstać, gdy pomieszasz wiersze i kolumny na różnych poziomach.

**LISTING 10 Listing skazany na niepowodzenie**

```
// App0610.dart -- BAD CODE

import 'package:flutter/material.dart';

import 'App06Main.dart';

import 'constraints_logger.dart';

Widget buildColumn(BuildContext context) {

return Row(

children: [

_buildRowOfThree(),
```

```
],
);
}
Widget _buildRowOfThree() {
return ConstraintsLogger(
comment: 'In _buildRowOfThree',
child: Row(
children: <Widget>[
_buildExpandedBox("Cat"),
_buildExpandedBox("Dog"),
_buildExpandedBox("Ape"),
],
),
);
}
Widget _buildExpandedBox(
String label, {
double height = 88.0,
}) {
return Expanded(
child: buildRoundedBox(
label,
height: height,
),
);
}
```

**LISTING 11 Pomoc w debugowaniu**

```
// constraints_logger.dart
import 'package:flutter/material.dart';
class ConstraintsLogger extends StatelessWidget
{
```

```
final String comment;
final Widget child;
ConstraintsLogger({
this.comment = "",
@required this.child,
}) : assert(comment != null);
Widget build(BuildContext context) {
return LayoutBuilder(
builder: (BuildContext context,
BoxConstraints constraints) {
print('$comment: $constraints to
${child.runtimeType}');
return child;
},
);
}
}
```

Po uruchomieniu kodu z Listingów 10 i 11 dzieją się trzy rzeczy:

* Na ekranie Twojego urządzenia nie pojawia się nic oprócz nudnego, szarego tła.

* W oknie narzędzia Android Studio Run pojawia się następujący komunikat o błędzie:

RenderFlex children have non-zero flex but  incoming width constraints are unbounded.

Twórcy Fluttera zaczynają jęczeć, gdy widzą tę wiadomość.

Później w oknie narzędzia Uruchom…

Jeśli rodzic ma zapakować swoje dziecko w folię termokurczliwą, dziecko nie może jednocześnie rozszerzyć się, aby dopasować się do swojego rodzica.

Ponadto w oknie narzędzia Uruchom wyświetlany jest komunikat podobny do tego:

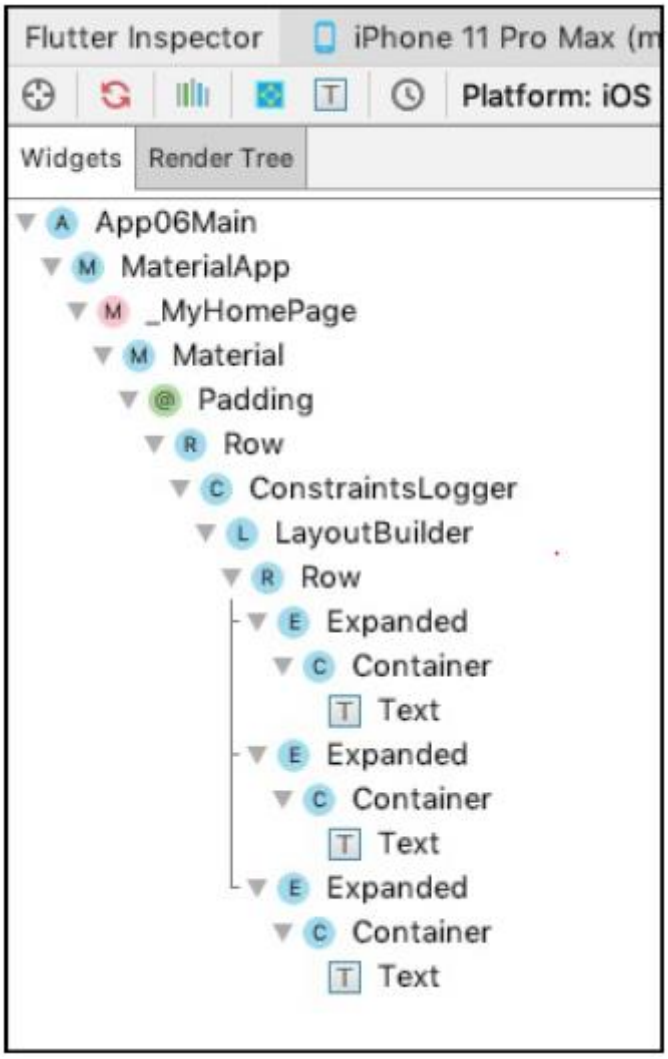
I/flutter ( 5317): In _buildRowOfThree:

BoxConstraints(0.0<=w<=Infinity,

0.0<=h<=683.4) to Row

Ten komunikat I/flutter informuje, że wewnętrzny wiersz układu ma ograniczenie szerokości, które ma coś wspólnego z nieskończonością. Ta pouczająca wiadomość 0.0<=w<=Infinity przychodzi do ciebie dzięki uprzejmości kodu z Listingu 11.

Co oznaczają te wszystkie komunikaty? W aplikacji Flutter widżety tworzą drzewo. Rysunek przedstawia drzewo widżetów przedstawione w narzędziu Flutter Inspector w Android Studio.

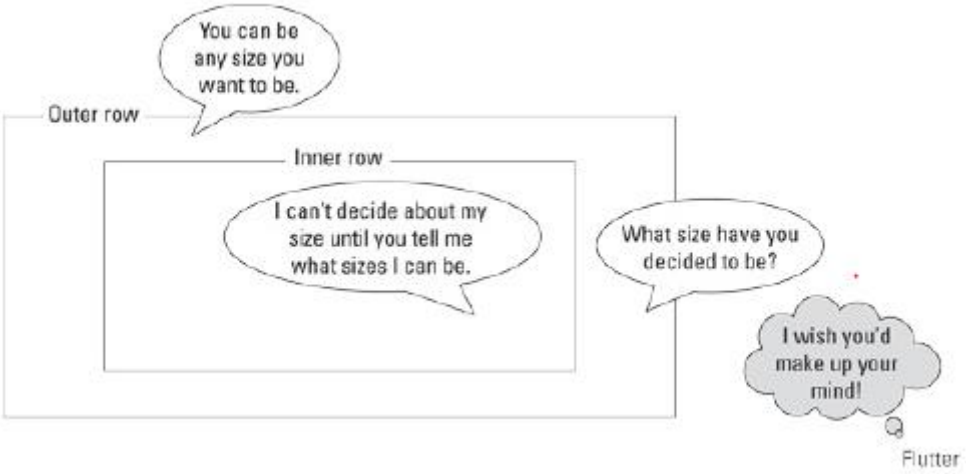



Aby wyświetlić widżety, Flutter podróżuje w dwóch kierunkach:

* Wzdłuż drzewa od góry do dołu Podczas tej podróży każdy widżet mówi swoim dzieciom, jakie mogą mieć rozmiary. W terminologii Fluttera każdy widżet nadrzędny przekazuje ograniczenia swoim dzieciom. Na przykład komunikat okna narzędzia Run mówi, że na listingu 11 wiersz zewnętrzny przekazuje ograniczenie szerokości 0,0<=w<=nieskończoność do wiersza wewnętrznego. Ze względu na słowo nieskończoność, to ograniczenie jest nazywane ograniczeniem nieograniczonym. Jeśli szukasz przykładu ograniczenia ograniczonego, spójrz na ten sam komunikat okna narzędzia Uruchom. Wiersz zewnętrzny przekazuje ograniczenie wysokości 0,0<=h<=683,4 do wiersza wewnętrznego. Ograniczeniem tym jest wartość 683,4 dp. W końcu Flutter dociera do dolnej części drzewa widżetów Twojej aplikacji. W tym momencie …

* Znowu wzdłuż drzewa — tym razem od dołu do góry Podczas tej podróży każdy widżet podrzędny mówi swojemu rodzicowi dokładnie, jaki rozmiar chce mieć. Rodzic zbiera te informacje od każdego ze swoich dzieci i wykorzystuje je do przypisania pozycji dzieciom. Czasami działa to dobrze, ale na listingu 11 kończy się niepowodzeniem.

Na listingu 11, ponieważ każde pudełko ze zwierzętami znajduje się wewnątrz widżetu Expanded, wewnętrzny wiersz nie wie, jak duży powinien być. Wewnętrzny rząd musi mieć określoną szerokość, aby podzielić przestrzeń między boksami dla zwierząt. Ale zewnętrzny rząd dał nieograniczone ograniczenie wewnętrznemu rzędowi. Zamiast informować wewnętrzny rząd o swojej szerokości, zewnętrzny rząd pyta wewnętrzny rząd o swoją szerokość. Nikt nie chce wziąć odpowiedzialności, więc Flutter nie wie, co robić.




Jak rozwiązać ten nieprzyjemny problem? Co dziwne, z pomocą przychodzi kolejny widżet Expanded.

```
Widget _buildRowOfThree() {
return Expanded(
child: ConstraintsLogger(
comment: 'In _buildRowOfThree',
child: Row(
children: <Widget>[
_buildExpandedBox("Cat"),
_buildExpandedBox("Dog"),
_buildExpandedBox("Ape"),
],
),
),
);
}
```

Ten nowy rozszerzony widżet przekazuje ograniczone ograniczenia w dół drzewa widżetów, jak widać z tego nowego komunikatu w oknie narzędzia Uruchom:

```
I/flutter ( 5317): In _buildRowOfThree:
```

BoxConstraints(w=371.4, 0.0<=h<=683.4) to Row

Nowy widżet Expanded mówi wewnętrznemu wierszowi, że jego szerokość musi wynosić dokładnie 371,4 dp, więc zamieszanie pokazane na rysunku znika. Flutter wie, jak wyświetlić widżety aplikacji, a na ekranie urządzenia widzisz trzy ładnie ułożone pudełka ze zwierzętami. Problem rozwiązany!

Ograniczenie w=371,4 jest nazywane ścisłym ograniczeniem, ponieważ nadaje wierszowi dokładny rozmiar bez jakiejkolwiek swobody. W przeciwieństwie do tego, ograniczenie 0,0<=h<=683,4 jest nazywane ograniczeniem luźnym. Luźne ograniczenie mówi: „Bądź tak niski, jak 0,0 dp wysoki i tak wysoki, jak 683,4 dp wysoki. Zobacz czy mi zależy."

Ten biznes z ograniczeniami i rozmiarami może wydawać się zbyt skomplikowany. Ale proces skanowania w dół drzewa, a następnie w górę drzewa, jest ważną częścią struktury Flutter. Podejście oparte na dwóch skanach umożliwia wydajną przebudowę stanowych widżetów. Odbudowa stanowych widżetów ma fundamentalne znaczenie dla sposobu projektowania aplikacji Flutter. Niektóre schematy układu działają dobrze z małą liczbą komponentów, ale zaczynają zwalniać, gdy liczba komponentów staje się duża. Schemat układu Flutter działa dobrze tylko z kilkoma widżetami i dobrze skaluje się w przypadku skomplikowanych układów z dużą liczbą widżetów.

Widżet ConstraintsLogger służy wyłącznie do debugowania. Przed opublikowaniem aplikacji usuń z kodu wszystkie zastosowania ConstraintsLogger.

**LISTING 12 Jak określić rozmiary względne**

```
// App0612.dart

import 'package:flutter/material.dart';

import 'App06Main.dart';

Widget buildColumn(BuildContext context) {

return Column(

mainAxisAlignment:

MainAxisAlignment.center,

crossAxisAlignment:

CrossAxisAlignment.stretch,

children: <Widget>[

buildTitleText(),

SizedBox(height: 20.0),

_buildRowOfThree(),

],

);

}

Widget _buildRowOfThree() {
```

```
return Row(
mainAxisAlignment:
MainAxisAlignment.spaceBetween,
children: <Widget>[
_buildExpandedBox(
"Moose",
),
_buildExpandedBox(
"Squirrel",
flex: 1,
),
_buildExpandedBox(
"Dinosaur",
flex: 3,
),
],
);
}
Widget _buildExpandedBox(
String label, {
double height = 88.0,
int flex,
}) {
return Expanded(
flex: flex,
child: buildRoundedBox(
label,
height: height,
),
);
}
```

Co stanie się z naszymi bohaterami, Łosiem i Wiewiórką, z Listingu 12? Aby się tego dowiedzieć, zobacz Rysunek


Zwróć uwagę na częste użycie słowa flex w listingu 12. Widżet Expanded może mieć wartość flex, znaną również jako współczynnik flex. Współczynnik elastyczności decyduje o tym, ile miejsca zajmuje widżet w stosunku do innych widżetów w wierszu lub kolumnie. Listing 12 zawiera trzy pola:

* Moose, bez wartości flex (wartość null)

* Squirrel, o wartości flex 1

* Dinosaur o wartości ugięcia 3

Oto podsumowanie wynikowego rozmiaru każdego pudełka:

Ponieważ pole Moose ma zerową wartość flex, pole Moose ma dowolną szerokość, która pochodzi jawnie z metody _buildExpandedBox. Szerokość pudełka Łosia wynosi 88,0. Zarówno pudełka Wiewiórka, jak i Dinozaur mają niezerowe, niezerowe wartości flex. Tak więc te dwa pudełka dzielą przestrzeń, która pozostaje po umieszczeniu pudełka Łosia. Przy wartościach elastyczności Wiewiórki: 1, Dinozaura: 3, pudełko Dinozaurów jest trzykrotnie szersze niż pudełko Wiewiórki. W moim emulatorze Pixel 2 pole Squirrel ma szerokość 70,9 dp, a pole Dinosaur ma szerokość 212,5 dp. Tak działają wartości flex.

Oprócz właściwości flex widżetu Expanded, Flutter ma klasy o nazwach Flex i Flexible. Tę trójkę łatwo pomylić. Każda instancja Flex jest instancją Row lub instancją Column. Każda instancja Expanded jest instancją klasy Flexible. Elastyczna instancja może mieć wartość flex, ale instancja Flexible nie wymusza na swoim potomku wypełniania dostępnego miejsca. Co ty na to!

**Jak duże jest moje urządzenie?**

Tytuł tej sekcji to pytanie, a odpowiedź brzmi „Nie wiesz”. Mogę uruchomić aplikację Flutter na małym iPhonie 6 lub na stronie internetowej na 50-calowym ekranie. Chcesz, aby Twoja aplikacja wyglądała dobrze bez względu na rozmiar mojego urządzenia. Jak możesz to robić? Listing 13 zawiera odpowiedź.

**LISTING 13 Sprawdzanie orientacji urządzenia**

```
// App0613.dart

import 'package:flutter/material.dart';

import 'App06Main.dart';

Widget buildColumn(context) {

if (MediaQuery.of(context).orientation ==
```

```
Orientation.landscape) {
return _buildOneLargeRow();
} else {
return _buildTwoSmallRows();
}
}
Widget _buildOneLargeRow() {
return Column(
mainAxisAlignment:
MainAxisAlignment.center,
children: <Widget>[
Row(
mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
children: <Widget>[
buildRoundedBox("Aardvark"),
buildRoundedBox("Baboon"),
buildRoundedBox("Unicorn"),
buildRoundedBox("Eel"),
buildRoundedBox("Emu"),
buildRoundedBox("Platypus"),
],
),
],
);
}
Widget _buildTwoSmallRows() {
return Column(
mainAxisAlignment:
MainAxisAlignment.center,
children: [
```

```
Row(
mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
children: [
buildRoundedBox("Aardvark"),
buildRoundedBox("Baboon"),
buildRoundedBox("Unicorn"),
],
),
SizedBox(
height: 30.0,
),
Row(
mainAxisAlignment:
MainAxisAlignment.spaceEvenly,
children: [
buildRoundedBox("Eel"),
buildRoundedBox("Emu"),
buildRoundedBox("Platypus"),
],
),
],
);
}
```

Rysunki pokazują, co się stanie, gdy uruchomimy kod z listingu 13. Gdy urządzenie działa w trybie pionowym, widoczne są dwa rzędy z trzema polami w każdym rzędzie. Ale gdy urządzenie jest w trybie poziomym, widzisz tylko jeden rząd z sześcioma polami. Różnica wynika z instrukcji if z listingu 13.

```
if (MediaQuery.of(context).orientation ==
Orientation.landscape) {
return _buildOneLargeRow();
} else {
return _buildTwoSmallRows();
```

}



Tak, język programowania Dart ma instrukcję if. Działa to w ten sam sposób, jak instrukcje if w innych językach programowania.

jeśli (określony warunek jest prawdziwy) {

Zrób to;

} W przeciwnym razie {

Rób te inne rzeczy;

}

W nazwie MediaQuery słowo Media odnosi się do ekranu, na którym działa Twoja aplikacja. Gdy wywołasz MediaQuery.of(context), otrzymasz z powrotem skarbnicę informacji o tym ekranie, takich jak

* orientacja: czy urządzenie jest w trybie pionowym, czy poziomym

* size.height i size.width: Liczba jednostek dp od góry do dołu i na ekranie urządzenia

* size.longestSide i size.shortestSide: większe i mniejsze wartości rozmiaru ekranu, niezależnie od tego, która wysokość, a która szerokość

* size.aspectRatio: szerokość ekranu podzielona przez jego wysokość

* devicePixelRatio: Liczba fizycznych pikseli dla każdej jednostki dp

* padding, viewInsets i viewPadding: Części wyświetlacza, które nie są dostępne dla twórcy aplikacji Flutter, takie jak części zasłonięte wycięciem* telefonu lub (czasami) klawiatura programowa

* alwaysUse24HourFormat: ustawienie wyświetlania czasu urządzenia

platformBrightness: Bieżące ustawienie jasności urządzenia

*… i wiele więcej

Na przykład tablet Pixel C o rozdzielczości 2560 na 1800 dp jest wystarczająco duży, aby wyświetlić rząd sześciu boksów ze zwierzętami w trybie pionowym lub poziomym. Aby przygotować aplikację do działania na takim urządzeniu, możesz nie chcieć polegać na właściwości orientacji urządzenia. W takim przypadku możesz zastąpić warunek z listingu 13 następującym:

```
if (MediaQuery.of(context).size.width >= 500.0)

{

return _buildOneLargeRow();

} else {

return _buildTwoSmallRows();

}
```

Zwróć uwagę na kontekst słowa w kodzie MediaQuery.of(context). Aby wyszukiwać media, Flutter musi znać kontekst, w którym działa aplikacja. Właśnie dlatego, począwszy od pierwszej listy w tym rozdziale, metoda build klasy _MyHomePage ma parametr kontekstu BuildContext. Listing 6-1 zawiera następujące wywołanie metody:

```
buildColumn(context)
```

A inne listy mają deklaracje metod z tym nagłówkiem:

```
Widget buildColumn(BuildContext context)
```

Listingi od 2 do 12 nie wykorzystują tego parametru context. Ale co, jeśli na listingu 1 pominę parametr context metody, tak jak poniżej:

buildColumn()

Potem wszystko idzie jak po maśle, aż do Listingu 13, który nie ma dostępu do kontekstu i nie może wywołać MediaQuery.of(context). Jaka szkoda! Kiedy tworzyłem Listing 1, dodałem parametr context, ponieważ przewidziałem potrzebę zastosowania wartości context w ostatnim listingu tego rozdziału — Listingu 13. Tak, jestem bardzo mądrym gościem. Cóż, to nie do końca prawda. Kiedy zaczynałem pisać ten rozdział, nie przewidziałem potrzeby stosowania wartości kontekstu. Nie widziałem problemu z kontekstem, dopóki nie zacząłem pisać tej ostatniej sekcji. W tym momencie cofnąłem się i zmodyfikowałem każdy pojedynczy listing, tak aby kontekst był dostępny dla Listingu 13. No cóż! Każdy musi dokonać korekty kursu. To część życia i z pewnością część profesjonalnego tworzenia aplikacji.

**Interakcja z Użytkownikiem**

Miłość jest w powietrzu! Słońce świeci. Ptaki śpiewają. Moje serce jest całe-trzepotanie. (Gra słów zamierzona.) Doris D. Developer chce znaleźć partnera i ma dwa ważne kryteria. Po pierwsze, chce kogoś, kto ma 18 lat lub więcej. Po drugie, szuka kogoś, kto uwielbia tworzyć aplikacje Flutter. Czy jest lepszy sposób, aby Doris osiągnęła swój cel, niż napisanie własnej aplikacji randkowej? W tym rozdziale opisano wybitną pracę Doris. Aby stworzyć aplikację, Doris używa kilku rodzajów widżetów: pola tekstowego, suwaka, przycisku rozwijanego i kilku innych. Widżet tego rodzaju — taki, który użytkownik widzi i z którym wchodzi w interakcję — nazywany jest elementem sterującym lub po prostu elementem sterującym. Aplikacja Doris ma również kilka widżetów układu, takich jak Środek, Wiersz i Kolumna, ale te widżety układu nie są nazywane elementami sterującymi. Użytkownik tak naprawdę ich nie widzi i na pewno nie wchodzi z nimi w interakcję. W tym rozdziale nacisk położono na elementy sterujące, a nie na widżety układu lub inne różne części aplikacji. Ostateczna aplikacja randkowa Doris nie jest w pełni funkcjonalna jak na standardy komercyjne, ale kod aplikacji ma kilkaset linii. Dlatego Doris rozwija aplikację w małych fragmentach — najpierw jeden element sterujący, potem drugi, kolejny i tak dalej. Każdy element to mała, wolnostojąca aplikacja do ćwiczeń. Pierwsza aplikacja ćwiczeniowa odpowiada na proste pytanie: czy przyszły partner ma co najmniej 18 lat?

**Prosty przełącznik**

Przełącznik to kontrolka znajdująca się w jednym z dwóch możliwych stanów: włączona lub wyłączona, tak lub nie, prawda lub fałsz, radość lub smutek, ukończone 18 lat lub nie. Listing 1 zawiera kod praktycznej aplikacji Switch.

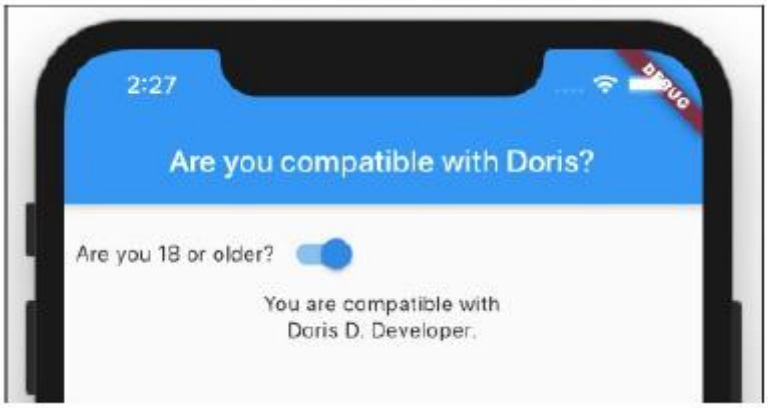
**LISTING 1 Ile masz lat?**

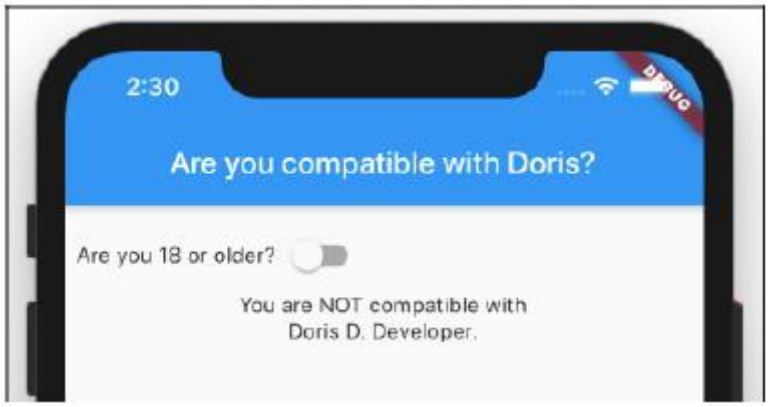
```
import 'package:flutter/material.dart';

void main() => runApp(App0701());

class App0701 extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: MyHomePage(),

);

}

}

class MyHomePage extends StatefulWidget {

@override

_MyHomePageState createState() =>

_MyHomePageState();

}

const _youAre = ’You are’;
```

```
const _compatible = 'compatible with\nDoris D.
Developer.';
class _MyHomePageState extends
State<MyHomePage> {
bool _ageSwitchValue = false;
String _messageToUser = "$_youAre NOT
$_compatible";
/// State
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text("Are you compatible with
Doris?"),
),
body: Padding(
padding: const EdgeInsets.all(8.0),
child: Column(
children: <Widget>[
_buildAgeSwitch(),
_buildResultArea(),
],
),
),
);
}
/// Build
Widget _buildAgeSwitch() {
return Row(
children: <Widget>[
Text("Are you 18 or older?"),
```

```
Switch(
value: _ageSwitchValue,
onChanged: _updateAgeSwitch,
),
],
);
}
Widget _buildResultArea() {
return Text(_messageToUser, textAlign:
TextAlign.center);
}
/// Actions
void _updateAgeSwitch(bool newValue) {
setState(() {
_ageSwitchValue = newValue;
_messageToUser =
_youAre + (_ageSwitchValue ? " " : "
NOT ") + _compatible;
});
}
}
```

Rysunki 1 i 2 przedstawiają aplikację w dwóch możliwych stanach.






Listingi w tej części to aplikacje do ćwiczeń. To małe próbki dużej, pięknej aplikacji randkowej Doris. Ale nawet „krótkie” programy mogą być długie i skomplikowane. Naginam kilka zasad i ignoruję kilka dobrych praktyk programistycznych, aby listingi były ze sobą kompatybilne. Ale nie martw się. Każdy listing działa poprawnie, a każda lista ilustruje przydatne koncepcje programistyczne Fluttera.

Kod z listingu 1 niewiele różni się od kodu z części 5. W części 5 pływający przycisk akcji ma parametr onPressed. Na listingu 1 widżet Switch ma coś podobnego. Listing 1 zawiera parametr onChanged. Wartość parametru onChanged jest funkcją; mianowicie funkcja _updateAgeSwitch. Gdy użytkownik przestawi przełącznik, to przełączenie wyzwala zdarzenie onChanged przełącznika, powodując wywołanie przez platformę Flutter funkcji _updateAgeSwitch. W przeciwieństwie do funkcji obsługi zdarzeń z rozdziału 5. funkcja _updateAgeSwitch z listingu 1 nie jest funkcją VoidCallback. Funkcja VoidCallback nie przyjmuje żadnych parametrów, ale funkcja _updateAgeSwitch ma parametr. Nazwa parametru to newValue:

```
void _updateAgeSwitch(bool newValue)
```

Gdy framework Flutter wywołuje _updateAgeSwitch, przekazuje nową pozycję widżetu Switch (wyłączony lub włączony) do parametru newValue. Ponieważ typ newValue to bool, newValue ma wartość false lub true. Jest fałszem, gdy przełącznik jest wyłączony, a prawdą, gdy przełącznik jest włączony. Jeśli _updateAgeSwitch nie jest VoidCallback, co to jest? (To było pytanie retoryczne, więc odpowiadam na nie… .) Funkcja _updateAgeSwitch jest typu ValueChanged<bool>. Funkcja ValueChanged przyjmuje jeden parametr i zwraca wartość void. Parametr funkcji może być dowolnego typu, ale parametr funkcji ValueChanged<bool> musi być typu bool. W ten sam sposób parametr funkcji ValueChanged<double> musi być typu double. I tak dalej.

Nie popełnij błędu: chociaż termin ValueChanged<bool> nie zawiera słowa Callback, funkcja _updateAgeSwitch jest wywołaniem zwrotnym. Gdy użytkownik odwróci widżet Switch, platforma Flutter odwołuje Twój kod. Tak, funkcja _updateAgeSwitch jest wywołaniem zwrotnym. To po prostu nie jest VoidCallback. W przypadku wielu kontrolek nic się nie stanie, jeśli nie zmienisz wartości kontrolki i nie wywołasz setState. Aby się trochę pośmiać, spróbowałem skomentować wywołanie setState w treści funkcji _updateAgeSwitch na Listingu 1:

```
void _updateAgeSwitch(bool newValue) {

// setState(() {

_ageSwitchValue = newValue;

_messageToUser = _youAre + (_ageSwitchValue ?

" " : " NOT ") + _compatible;
```

```
// });
}
```

Następnie odkomentowałem wywołanie setState i skomentowałem instrukcje przypisania:

```
void _updateAgeSwitch(bool newValue) {
setState(() {
// _ageSwitchValue = newValue;
// _messageToUser =
// _youAre + (_ageSwitchValue ? " " :
" NOT ") + _compatible;
});
}
```

W obu przypadkach ponownie uruchomiłem program, a następnie nacisnąłem przełącznik. Nie tylko _messageToUser odmówił zmiany, ale przełącznik nawet nie drgnął. To ustawia tamto! Wygląd przełącznika jest całkowicie zależny od zmiennej _ageSwitchValue i wywołania setState. Jeśli nie przypiszesz niczego do _ageSwitchValue lub nie wywołasz setState, przełącznik całkowicie przestanie odpowiadać.

**Słowo kluczowe const Darta**

Oto moja główna zasada: kiedy już raz podejmę decyzję, nigdy nie zmieniam zdania. Jedynym wyjątkiem od tej reguły jest zmiana zdania na temat zasady kardynalnej. W tworzeniu aplikacji kwestia zmian jest bardzo ważna. Termin zmienna pochodzi od słowa zmienna, co oznacza „zmianę”. Ale niektórych rzeczy nie powinno się zmieniać. Na listingu 1 odwołuj się więcej niż raz do ciągów „Jesteś” i „kompatybilny z\nDoris D. Developer”, dlatego tworzę dla tych ciągów przydatne nazwy _youAre i _compatiblc. W ten sposób nie muszę więcej niż raz wpisywać rzeczy typu „kompatybilny z\nDoris D. Developer”. Nie ryzykuję wpisania frazy raz poprawnie, a innym razem niepoprawnie. Ale co, jeśli wartość _youAre może się zmienić przez cały czas trwania programu? Deweloper, który pracuje z moim kodem, może omyłkowo napisać

```
_youAre = 'sweet';
```

Nie chcę, żeby tak się stało. Chcę, aby _youAre oznaczało „Jesteś” przez cały czas trwania programu. Android Studio powinno oznaczyć przypisanie _youAre = ’sweet’ jako błąd. Dlatego na listingu 1 deklaruję _youAre słowem const. Słowo kluczowe const Darta jest skrótem od stałej. Jako stała, wartość _youAre nie może się zmienić. To samo dotyczy deklaracji _kompatybilny z Listingu 1. Użycie słowa kluczowego const w Dart jest środkiem bezpieczeństwa i to cholernie dobrym!

Jeśli się zastanawiasz, \n w „kompatybilny z\nDoris D. Developer” każe Dartowi przejść do nowej linii tekstu. W ten sposób Doris D. Developer pojawia się w osobnej linii. Kombinacja znaków \n nazywana jest sekwencją specjalną.

Odnosząc się do kodu z listingu 1, doświadczony programista mógłby powiedzieć „stała _youAre” lub „zmienna _youAre”. Były dokładniejsze, ale to drugie jest do przyjęcia.

Dart ma dwa słowa kluczowe wskazujące, że pewne rzeczy nie powinny się zmieniać: const i final. Słowo kluczowe const mówi: „Nie zmieniaj tej wartości w żadnym momencie działania aplikacji”. Słowo kluczowe final mówi: „Nie zmieniaj tej wartości, chyba że ponownie napotkasz tę deklarację”. Różnica między const a final ma wiele subtelnych konsekwencji, więc nie otwieram puszki robaków w tym rozdziale. Zamiast tego po prostu pamiętaj, że programy z const mogą działać nieco szybciej niż programy z final. Możesz umieścić dowolną starą deklarację const na najwyższym poziomie kodu lub wewnątrz deklaracji funkcji. Ale w przypadku const na początku klasy sytuacja jest inna. Poniższy kod jest nielegalny:

```
// Don’t do this:

class _MyHomePageState extends

State<MyHomePage> {

const _youAre = 'You are';
```

Ale ten kod jest w porządku:

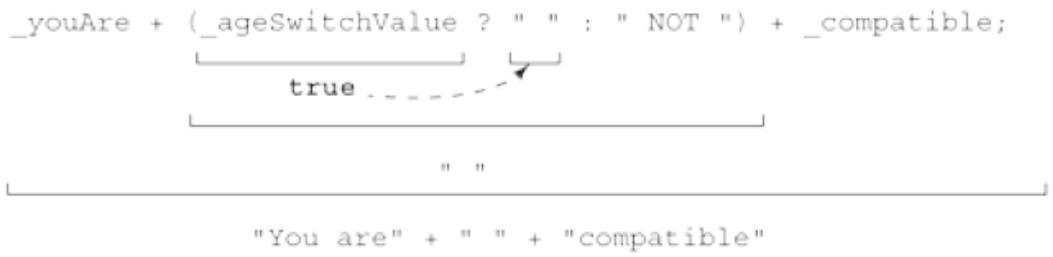
```
// Do this instead:

class _MyHomePageState extends

State<MyHomePage> {

static const _youAre = 'You are';
```

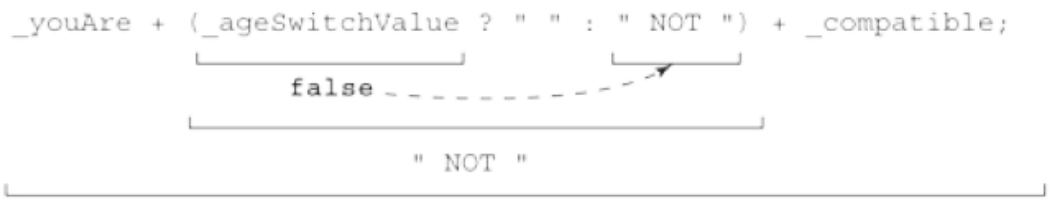
Kompatybilny czy NIE?

W przypadku niektórych użytkowników aplikacja randkowa powinna zawierać komunikat „Jesteś kompatybilny z Doris D. Developer”. W przypadku innych użytkowników aplikacja powinna dodać NOT do swojego komunikatu. Dlatego listing 1 zawiera następujący kod:

```
_messageToUser =

_youAre + (_ageSwitchValue ? " " : " NOT ")

+ _compatible;
```

Wyrażenie _ageSwitchValue? „ ” : „ NOT ” jest wyrażeniem warunkowym, a kombinacja ? a : w tym wyrażeniu jest operatorem warunkowym Darta. Rysunek 7.3 pokazuje, jak Dart ocenia wyrażenie warunkowe.




Wyrażenie warunkowe wygląda następująco:

condition ? expression1 : expression2

Gdy warunek jest spełniony, wartość całego wyrażenia jest taka, jaka znajduje się w części wyrażenia1. Ale gdy warunek jest fałszywy, wartość całego wyrażenia jest taka, jaka znajduje się w części wyrażenia2. Oprócz wyrażeń warunkowych Dart ma instrukcje if. Wyrażenie warunkowe przypomina instrukcję if ale w przeciwieństwie do instrukcji if wyrażenie warunkowe ma wartość. Wartość tę można przypisać do zmiennej. Aby to zilustrować, podam instrukcję if, której efekt jest taki sam, jak wyrażenie warunkowe z listingu 1:

if (_ageSwitchValue) {

_messageToUser = _youAre + " " + _compatible;

} else {

_messageToUser = _youAre + " NOT " +

_compatible;

}

Przetłumaczone na zwykły język, to stwierdzenie if mówi:

Jeśli zmienna bool _ageSwitchValue ma

wartość prawdziwa,

_messageToUser = _jesteś + " " + _kompatybilny;

W przeciwnym razie

_messageToUser = _jesteś + " NIE " + _zgodny;

W niektórych sytuacjach wybór między instrukcją if a wyrażeniem warunkowym jest kwestią gustu. Ale na Listingu 1 wyrażenie warunkowe jest wyraźnym zwycięzcą. W końcu instrukcja if nie ma wartości.

Nie możesz przypisać instrukcji if do czegokolwiek ani dodać instrukcji if do czegokolwiek. Tak więc kod następującego rodzaju jest nielegalny:

```
// THIS CODE IS INVALID.

_messageToUser =

_youAre +

if (_ageSwitchValue) {

" ";

} else {

" NOT ";

} +

_compatible;
```

Inną nazwą operatora warunkowego Darta jest operator trójskładnikowy. Słowo trójskładnikowe oznacza „trzy”, a operator składa się z trzech części: jednej przed znakiem zapytania, drugiej między znakiem zapytania a dwukropkiem i trzeciej po dwukropku.

**Czekaj na to!**

Dzisiejsi użytkownicy są niecierpliwi. Chcą natychmiastowej informacji zwrotnej. Skąd mam to wiedzieć? Przekonało mnie kilku aktorów. Oto historia:

Dawno temu w teatrze daleko od Broadwayu widziałem komedię muzyczną w towarzystwie dziewięciu innych osób. Trzech z nich było moimi przyjaciółmi, dwóch z nich obserwowało z trzeciego rzędu, a pozostała czwórka była wykonawcami sztuki. Zapamiętałem to wydarzenie z dwóch powodów. Po pierwsze, był to początek mojej życiowej filozofii, która polegała na tym, aby wykonawcy czuli się dobrze. Żarty ze spektaklu nie były śmieszne, ale z każdego śmiałem się w głos. Śpiew był rozstrojony, ale energicznie klaskałem po każdej piosence. W końcu wszyscy dobrze się bawiliśmy, a aktorzy nie żałowali, że grali przed sześcioosobową publicznością. Inną niezapomnianą częścią tego wydarzenia była piosenka popisowa spektaklu. To było trochę głupie, ale utkwiło mi w pamięci na lata. Do dziś moja żona i ja chichoczemy, ilekroć wykrzykujemy kilka pierwszych taktów utworu. Tytuł tej melodii brzmiał „Natychmiastowa gratyfikacja”. To była kpina ze współczesnej kultury, w której każda potrzeba jest pilna, a każde pragnienie musi zostać spełnione. Podążając tym tokiem myślenia, przedstawiam skromny widżet znany jako RaisedButton. Guzik to niewiele. Naciskasz i coś się dzieje. Naciskasz go ponownie i coś może się wydarzyć lub nie. Przyciski były kiedyś podstawowym elementem sterującym dla twórców stron internetowych i twórców aplikacji. Ale w dzisiejszych czasach guziki są passé. Gdy użytkownik przestawi przełącznik, aplikacja natychmiast zareaguje. Nie trzeba czekać, aby znaleźć przycisk do naciśnięcia. Stary jasnoszary prostokąt z napisem Prześlij zajął miejsce z tyłu. Aby uczcić stare dobre czasy, przykład z tej sekcji unika szybkiej reakcji aplikacji z Listingu 1. Kiedy użytkownik przesuwa przełącznik, przełącznik po prostu się porusza. Aplikacja nie mówi „Jesteś kompatybilny” lub „Nie jesteś kompatybilny”, dopóki użytkownik nie naciśnie przycisku. Przyjdź, usiądź na werandzie i zrelaksuj się podczas działania tej aplikacji! Kod znajduje się na listingu 2.

LISTING 2 Reagowanie na naciśnięcie przycisku

```
// Copy the code up to and including the
```

```
_buildAgeSwitch
// method from Listing 7-1 here.
Widget _buildResultArea() {
return Row(
children: <Widget>[
RaisedButton(
child: Text("Submit"),
onPressed: _updateResults,
),
SizedBox(
width: 15.0,
),
Text(_messageToUser, textAlign:
TextAlign.center),
],
);
}
/// Actions
void _updateAgeSwitch(bool newValue) {
setState(() {
_ageSwitchValue = newValue;
});
}
void _updateResults() {
setState(() {
_messageToUser = 'You are' +
(_ageSwitchValue ? " " : " NOT ") +
'compatible with \nDoris D.
Developer.';
});
}
```

}

Rysunek 4 przedstawia migawkę przebiegu kodu z Listingu 2.


W połączeniu z jakimś kodem z Listingu 1, aplikacja z Listingu 2 ma zarówno procedury obsługi zdarzeń onPressed, jak i onChanged. W szczególności:

* Funkcja _updateAgeSwitch obsługuje zdarzenia onChanged dla przełącznika. Gdy użytkownik dotknie przełącznika, wygląd

przełączać zmiany z wyłączonego na włączony lub z włączonego na wyłączony.

* Funkcja _updateResults obsługuje zdarzenia onPressed dla przycisku. Gdy użytkownik naciśnie przycisk, komunikat aplikacji nadąża za statusem przełącznika. Jeśli przełącznik jest włączony, komunikat brzmi: „Jesteś zgodny”. Jeśli przełącznik jest wyłączony, komunikat zmieni się na „Nie jesteś kompatybilny”.

Pomiędzy momentem, w którym użytkownik przestawi przełącznik, a momentem, w którym użytkownik naciśnie przycisk, komunikat na ekranie może być niezgodny ze stanem przełącznika. W formularzu online z kilkoma pytaniami to nie problem. Użytkownik nie spodziewa się zobaczyć wyniku, dopóki nie naciśnie przycisku końcowego. Jednak w aplikacjach do ćwiczeń opisanych w tym rozdziale, z których każda ma tylko jedno pytanie do użytkownika, problematyczny jest brak koordynacji między odpowiedzią użytkownika a wyświetlanym komunikatem. Te aplikacje do ćwiczeń nie zdobywają żadnych nagród za wrażenia użytkownika. Na szczęście Doris nie publikuje swoich aplikacji do ćwiczeń. Zamiast tego publikuje aplikację, która łączy w sobie wszystkie elementy sterujące z jej aplikacji do ćwiczeń i nie tylko.

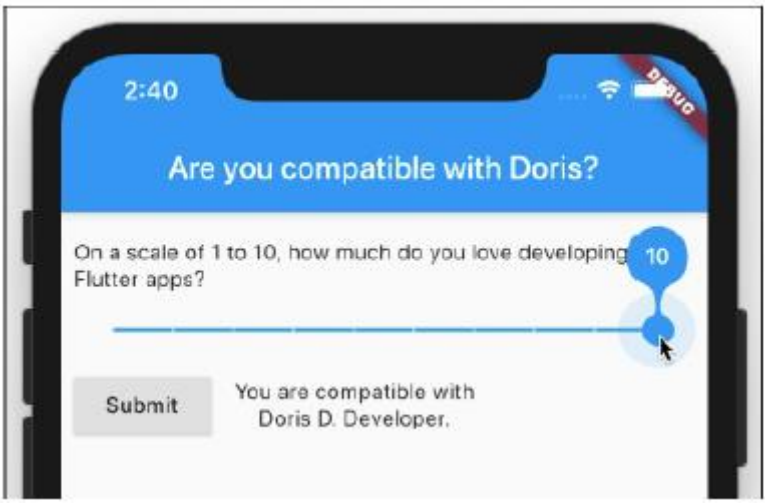
**Jak bardzo kochasz Fluttera?**

Doris the Developer chce poznać kogoś, kto uwielbia tworzyć aplikacje Flutter. Jej domowa aplikacja randkowa zawiera suwak z wartościami od 1 do 10. Akceptowalne są oceny 8 i więcej. Każdy z odpowiedzią 7 lub niższą może wybrać się na wycieczkę. Na listingu 3 przedstawiono najważniejsze elementy aplikacji slider Doris.

```
// This is not a complete program. (No way!)

class _MyHomePageState extends

State<MyHomePage> {
```

```
double _loveFlutterSliderValue = 1.0;
Widget _buildLoveFlutterSlider() {
return // …
Text("On a scale of 1 to 10, "
"how much do you love developing
Flutter apps?"),
Slider(
min: 1.0,
max: 10.0,
divisions: 9,
value: _loveFlutterSliderValue,
onChanged: _updateLoveFlutterSlider,
label:
'${_loveFlutterSliderValue.toInt()}',
),
}
void _updateLoveFlutterSlider(double
newValue) {
setState(() {
_loveFlutterSliderValue = newValue;
});
}
void _updateResults() {
setState(() {
_messageToUser = _youAre +
(_loveFlutterSliderValue >= 8 ? " " :
" NOT ") +
_compatible;
});
}
}
```

Rysunek 5 przedstawia uruchomienie aplikacji suwaka z suwakiem ustawionym na 10. (Jak inaczej można oczekiwać, że ustawię suwak „miłosne trzepotanie”?)




Wywołanie konstruktora Slider na listingu 3 ma sześć następujących parametrów:

min: najmniejsza wartość suwaka. Mały gadżet, który porusza się od lewej do prawej wzdłuż suwaka, nazywa się kciukiem. Pozycja kciuka określa wartość suwaka. Tak więc min jest wartością suwaka, gdy kciuk suwaka znajduje się w skrajnym lewym punkcie. Parametr min ma typ double.

max: największa wartość suwaka. Jest to wartość suwaka (ponownie podwójna), gdy kciuk znajduje się w skrajnym prawym punkcie.

Wartości suwaka mogą wzrosnąć, przechodząc od lewej do prawej lub od prawej do lewej. Przed wyświetleniem suwaka Flutter sprawdza właściwość textDirection. Jeśli wartością jest TextDirection.ltr, minimalna wartość suwaka znajduje się po lewej stronie. Ale jeśli wartość właściwości textDirection to TextDirection.rtl, minimalna wartość suwaka znajduje się po prawej stronie. Aplikacje napisane dla osób mówiących po arabsku, persku, hebrajsku, paszto i urdu używają TextDirection.rtl. Inne aplikacje używają TextDirection.ltr. Jeśli się zastanawiasz, Flutter nie obsługuje pisma bustrofedonowego — starożytnego stylu, w którym naprzemienne linie płyną od lewej do prawej, a następnie od prawej do lewej.

division: Liczba odstępów między punktami, w których można umieścić kciuk. Suwak na listingu 3 można ustawić na wartości 1,0, 2,0, 3,0 itd., aż do 10,0. Jeśli pominiesz parametr dzielenia lub ustawisz ten parametr na wartość null, kciuk można umieścić w dowolnym miejscu wzdłuż suwaka. Na przykład z poniższym

konstruktora wartość suwaka może wynosić 0,0, 0,20571428571428554, 0,917142857142857, 1,0 lub prawie dowolną inną liczbę z przedziału od 0 do 1.

```
Slider(

min: 0.0,

max: 1.0,

value: _loveFlutterSliderValue,

onChanged: _updateLoveFlutterSlider,
```

)

value: liczba z zakresu od min do max. Ten parametr określa pozycję kciuka.

onChanged: Funkcja obsługi zdarzeń dla zmian w suwaku. Kiedy użytkownik porusza kciukiem suwaka, framework Flutter wywołuje tę funkcję.

label: Widżet wyświetlany na wskaźniku wartości suwaka.

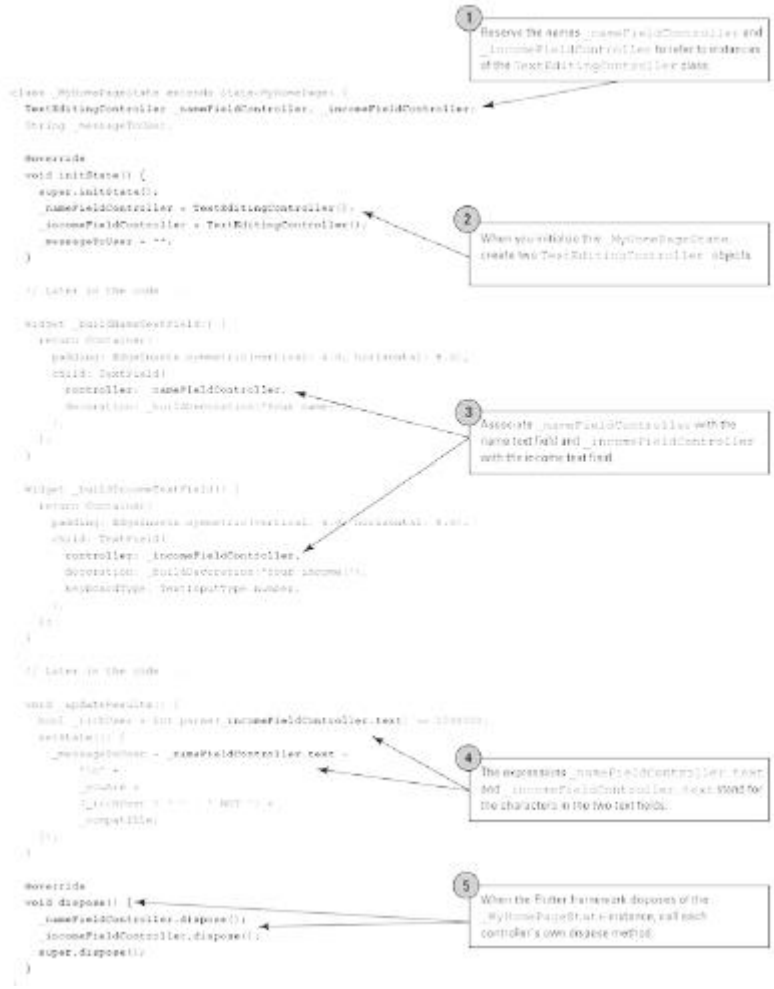
Gdy użytkownik porusza kciukiem, pojawia się dodatkowy kształt. Ten kształt jest wskaźnikiem wartości suwaka. Na rysunku  dymek z liczbą 10 to wskaźnik wartości suwaka. Pomimo swojej nazwy wskaźnik wartości niekoniecznie wyświetla widżet Tekst pokazujący wartość suwaka. W rzeczywistości wskaźnik wartości może wyświetlać wszystko, co chcesz. (Cóż, prawie wszystko). Na szczęście dla nas widżet na suwaku z listingu 3 wyświetla wartość _loveFlutterSliderValue — własną wartość suwaka. Ale pamiętaj: jeśli nie chcesz, aby liczby takie jak 0,20571428571428554 pojawiały się we wskaźniku wartości, musisz przekonwertować wartości double suwaka na wartości typu int. Dlatego na listingu 3 widżet na wskaźniku wartości suwaka wyświetla _loveFlutterSliderValue.toInt(), a nie zwykły stary

_loveFlutterSliderValue.

Jeśli nie określisz parametru etykiety lub określisz etykietę, ale ustawisz ją na wartość null, wskaźnik wartości nigdy się nie pojawi.

**Radzenie sobie z polami tekstowymi**

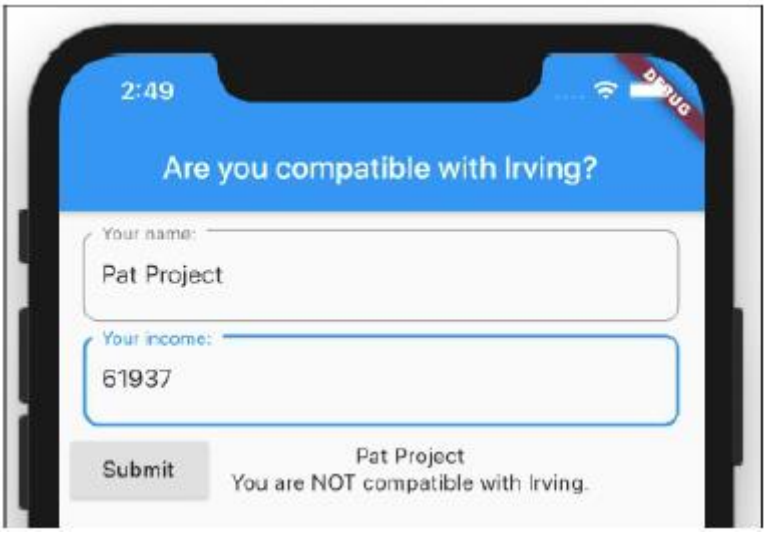
W tej części przedstawiam przyjaciela Doris, Irvinga. W przeciwieństwie do Doris, Irving chce towarzysza z dużą ilością pieniędzy. W tym celu Irving prosi Doris o stworzenie wariacji na temat jej aplikacji randkowej. Niestandardowa aplikacja Irvinga ma dwa pola tekstowe — jedno na imię i nazwisko użytkownika, a drugie na dochód użytkownika. Jeśli dochód użytkownika wynosi 100 000 USD lub więcej, aplikacja zgłasza „kompatybilność”. W przeciwnym razie aplikacja zgłasza „niezgodność”.




Aby zachować rozmiar pokazany na rysunku 7, pominąłem deklarację _buildDecoration. Jeśli się zastanawiasz, oto kod tej metody:

```
InputDecoration _buildDecoration(String label)

{

return InputDecoration(

labelText: label,

border: OutlineInputBorder(

borderRadius:

BorderRadius.all(Radius.circular(10.0)),

),

);

}
```

Rysunek 8 pokazuje żałosne próby Pata, by być zgodnym z Irvingiem. Z dochodem w wysokości 61 937 dolarów Pat nie ma szans. (W 2018 r. średni dochód gospodarstw domowych w Stanach Zjednoczonych wyniósł 61 937 USD. Irving ma zbyt wysokie cele).




Pola tekstowe mają te same typy procedur obsługi zdarzeń, co przełączniki i suwaki. W szczególności konstruktor TextField może mieć procedurę obsługi zdarzenia onChanged — funkcję, która wygląda tak:

void _updateStuff(String newValue) {

// When the user types a character, do

something with

// the characters inside the text field

(the newValue).

}

Ale co z naciśnięciem przycisku? Czy istnieje dobry sposób, aby dowiedzieć się, co znajduje się w polu tekstowym, gdy znaki pola

Ale co z naciśnięciem przycisku? Czy istnieje dobry sposób, aby dowiedzieć się, co znajduje się w polu tekstowym, gdy znaki pola się nie zmieniają? Tak jest. To TextEditingController — wyróżniająca się funkcja na rysunku 7. W rzeczywistości na rysunku 7 znajdują się dwa obiekty TextEditingController — jeden dla pola Twoje imię i drugi dla pola Twój dochód. Kilka następnych akapitów dodaje szczegóły do ponumerowanych objaśnień na rysunku 7.

Objaśnienia 1 i 2

W programie Flutter wywołania konstruktora rządzą grzędą. Otrzymujesz widżet Text z wywołaniem konstruktora, takim jak Text("Hello"). Otrzymujesz Kolumnę i dwa widżety Tekst z kodem takim jak Kolumna(dzieci: [Tekst(’a’), Tekst(’b’)]). Kiedy wydajesz wywołanie konstruktora, samo wywołanie oznacza obiekt. Na przykład wywołanie Text("Hello") oznacza konkretny widżet Text — instancja klasy Text. Możesz przypisać wywołanie do zmiennej i użyć tej zmiennej w innym miejscu kodu:

@override

Widget build(BuildContext context) {

Text myTextInstance = Text("I’m reusable");

return Scaffold(

```
appBar: AppBar(
title: myTextInstance,
),
body: Column(
children: <Widget>[
myTextInstance,
],
),
);
}
```

W wielu przypadkach można oddzielić deklarację zmiennej od wywołania:

```
Text myTextInstance;
// More code here, and elsewhere …
myTextInstance = Text("I'm reusable");
```

Na rysunku 7 deklaracja dwóch zmiennych kontrolera _nameFieldController i _incomeFieldController) jest oddzielona od odpowiednich wywołań konstruktora TextEditingController. Robię to, aby wprowadzić metody initState idisplay Fluttera. Obiekt państwowy jest jak wszystko inne na świecie — powstaje iw końcu znika. Flutter wywołuje funkcję initState, gdy pojawia się instancja State, oraz funkcję usuwania, gdy instancja State znika.

**NULL POUR LES NULS**

Możesz zadeklarować nazwę zmiennej bez przypisywania czegokolwiek do tej zmiennej. Jeśli to zrobisz, wartość początkowa zmiennej będzie równa null, co oznacza „absolutnie nic”. W wielu przypadkach właśnie to chcesz zrobić. Ale musisz być ostrożny. Niepożądana wartość null może być niebezpieczna. Na przykład następujący kod ulega awarii jak lekkomyślny samochód na New Jersey Turnpike:

```
main() {
int quantity;
print(quantity.isEven); // null.isEven
-- You can’t do this
}
```

Z drugiej strony, jeśli przypiszesz coś do zmiennej ilości, kod działa bez żadnych problemów:

```
main() {
int quantity;
quantity = 22;
print(quantity.isEven); // Outputs the
```

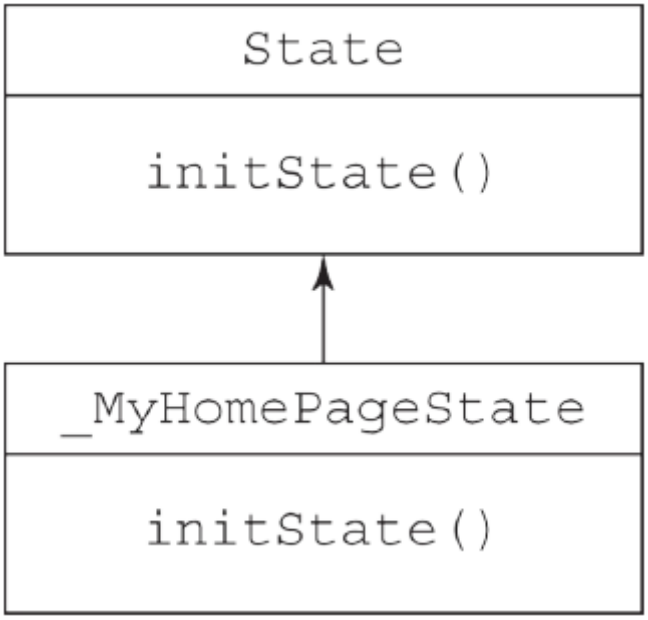
word "true" (without quotes)

}

Oto błąd, który czasami popełniam: tworzę deklarację zmiennej, która nie przypisuje wartości swojej zmiennej. Potem zapominam przypisać wartość do tej zmiennej w innym miejscu kodu. Ups! Mój kod ulega awarii. Moja rada: staraj się nie popełniać tego błędu.

Może to nie być oczywiste, ale kod na rysunku 7 odnosi się do dwóch różnych metod initState. Deklaracja rozpoczynająca się od void initState() opisuje metodę należącą do klasy _MyHomePageState. Ale klasa _MyHomePageState

rozszerza własną klasę State Fluttera, a ta klasa State ma własną deklarację initState.




Kiedy masz dwie metody o nazwie initState, jak odróżnić jedną od drugiej? A co jeśli spotkasz kobietę o imieniu Mary, której dziecko również ma na imię Mary? Możliwe, że dziecko nie mówi do swojej matki „Mary”. Zamiast tego dziecko nazywa swoją matkę „mamo” lub coś w tym stylu. Na urodziny swojej matki kupuje pamiątkowy kubek z napisem Super Mama, a mama uśmiecha się uprzejmie, otrzymując kolejny bezużyteczny prezent. To samo dzieje się, gdy dwie klasy — rodzic i jego dziecko — mają metody o nazwie initState. Klasa potomna (_MyHomePageState) musi wywołać metodę initState należącą do jej klasy nadrzędnej (klasa State Fluttera). W tym celu klasa potomna wywołuje funkcję super.initState(). Inaczej niż w przypadku Marii, użycie słowa kluczowego „super” nie ma na celu schlebiać. Jest to po prostu odwołanie do metody initState zdefiniowanej w klasie nadrzędnej. (Nie mogę się oprzeć: słowo kluczowe „super” może nie jest pochlebne, ale z pewnością jest trzepotanie.) Aby nieco rozszerzyć metaforę matka/córka, wyobraź sobie, że Super Mama Mary jest agentką nieruchomości. W takim przypadku dziecko nie może kupić domu bez uprzedniej konsultacji z matką. Metoda DecydujWhichHouse dziecka musi zawierać wywołanie metody DecydujWhichHouse matki, na przykład:

// The child’s method declaration:

@override

void decideWhichHouse() {

```
super.decideWhichHouse();

// Etc.

}
```

Może tak być w sytuacji, gdy Twój kod nadpisuje metodę initState Fluttera. W niektórych wersjach Fluttera, jeśli nie wywołasz super.initState(), Twój kod nie zadziała.

**Objaśnienie 3**

Każdy konstruktor TextField może mieć własny parametr kontrolera. Kontroler pola tekstowego pośredniczy w przepływie informacji między polem tekstowym a innymi częściami aplikacji. W innym miejscu wywołania konstruktora TextField parametr TextInputType.number w konstruktorze pola tekstowego dochodu informuje urządzenie, aby wyświetlało klawiaturę programową zawierającą tylko cyfry na klawiszach. Alternatywy obejmują TextInputType.phone, TextInputType.emailAdress, TextInputType.datetime i inne.

Ta wskazówka ma zastosowanie podczas opracowywania i testowania aplikacji. Emulator Androida i symulator iPhone'a mają opcje tłumienia wyglądu klawiatury programowej, umożliwiając wprowadzanie danych tylko za pomocą twojej klawiatury komputera deweloperskiego. Jeśli ta opcja jest włączona, nie widzisz efektu parametru TextInputType.number. Jeśli wpiszesz literę na klawiaturze komputera, pojawi się ona w polu tekstowym aplikacji. Jeśli planujesz uruchomić aplikację na prawdziwym, fizycznym telefonie, powinieneś przetestować aplikację z włączoną klawiaturą programową urządzenia wirtualnego. Kiedy to zrobisz, możesz zobaczyć kłopotliwe efekty, których się nie spodziewałeś. Na przykład po przejściu z pola tekstowego do innego rodzaju kontrolki klawiatura programowa nie znika. Aby klawiatura programowa zniknęła automatycznie, zamknij rusztowanie w czujniku gestów. Oto jak to zrobić:

```
Widget build(BuildContext context) {

return GestureDetector(

onTap: () {

final currentFocus =

FocusScope.of(context);

if (!currentFocus.hasPrimaryFocus) {

currentFocus.unfocus();

}

},

child: Scaffold(

// … Etc.
```

**Objaśnienie 4**

Na rysunku 7 wyrażenie _nameFieldController.text oznacza znaki pojawiające się w polu tekstowym Nazwa, a _incomeFieldController.text oznacza znaki w polu tekstowym Dochód. Jeśli kod zawierał instrukcję

_nameFieldController.text = "May I.

Havanother";

wykonanie tego oświadczenia zmieniłoby to, co już znajdowało się w polu tekstowym Imię, na May I. Havanother. Na rysunku 7 wyrażenie _nameFieldController.text dodaje nazwę użytkownika do wiadomości wychodzącej. Wyrażenie _incomeFieldController.text oznacza dowolne znaki wprowadzone przez użytkownika w polu Dochód aplikacji, ale te znaki mają mały haczyk. Zawartość pola tekstowego jest zawsze wartością typu String, nigdy wartością liczbową. Na rysunku 8 Pat wprowadza 61937 w polu tekstowym Dochód, więc wartość _incomeFieldController.text to „61937" (łańcuch), a nie 61937 (liczba). Na szczęście klasa int Darta ma metodę parsowania. Jeśli wartość _incomeFieldController.text to „61937" (ciąg), wartość int.parse(_incomeFieldController.text) wynosi 61937 (wartość liczby int). Na rysunku 7 kod

int.parse(_incomeFieldController.text) >=

1000000

porównuje liczbę taką jak 61937 z żądaną przez Irvinga liczbą 1000000. Wynikiem porównania jest prawda lub fałsz, więc wartość _richUser przyjmuje wartość prawda lub fałsz.

**CO ROBI DARN DOT?**

W programowaniu obiektowym obiekt może mieć pewne rzeczy zwane właściwościami. Używając notacji kropkowej, możesz odwoływać się do każdej z tych właściwości. Oto kilka przykładów:

*Każda instancja String ma właściwości length i isEmpty. Wartość „Dart".length to 4, a wartość „".isEmpty to prawda.

*Każda wartość int ma właściwości isEven, isNegative i bitLength. Wartość 44.isEven to prawda, a wartość 99.isNegative to fałsz. Wartość 99.bitLength wynosi 7, ponieważ binarna reprezentacja liczby 99 to 1100011, która ma 7 bitów.

*Każda instancja TextEditingController ma właściwość text.

Na rysunku 7 wartością _nameFieldController.text jest dowolny ciąg znaków, który pojawia się w polu tekstowym Name.

Możesz zastosować notację kropkową do wszelkiego rodzaju wyrażeń. Na przykład wartość (29 + 10).isEven jest fałszem. W przypadku wyrażenia „Lubię Darta" wartość wyrażenia „długość" wynosi 11. Właściwości to przykłady rzeczy zwanych członkami. Członkowie klasy obejmują również zmienne i metody klasy. Rozważ następujące:

*Każda instancja String ma metody o nazwach toUpperCase, endsWith, split, trim i wiele innych.

Wartość „Uwaga!".toUpperCase() to „UWAGA!".

Wartość „Holy moly! .trim() to „Holy moly!".

*Każda wartość int ma metody o nazwach abs, toRadixString i kilka innych. Wartość (-182).abs() wynosi 182, ponieważ 182 to

wartość bezwzględna –182. Wartość 99.toRadixString(2) wynosi 1100011, ponieważ binarna (o podstawie 2) reprezentacja 99 to 1100011.

Nie ma nic tajemniczego w członkach klasy. Oto klasa o nazwie Account i główna funkcja wywołująca konstruktor klasy Account

```
class Account {

// Two member variables:

String customerName;

int balance;

// A member method:

void deposit({int amount}) {

balance += amount;

}

}

void main() {

// A call to the Account class's

constructor:

Account myAccount = Account();

// References to the Account class's

members:

myAccount.customerName = "Barry Burd";

myAccount.balance = 100;

myAccount.deposit(amount: 20);

print(myAccount.customerName);

print(myAccount.balance);

}

/*

* Output:

* Barry Burd

* 120

*/
```

Klasy na liście Flutter w tej książce również mają członków. Na przykład klasa na rysunku 7 ma kilka członków, w tym _nameFieldController, _incomeFieldController, _messageToUser, initState i build. Niektóre klasy mają tak zwane składowe statyczne. Statyczny element członkowski należy do całej klasy, a nie do żadnej instancji klasy. Na przykład klasa int ma metodę statyczną o nazwie parse.

Ponieważ metoda analizy jest statyczna, przed kropką umieszcza się nazwę klasy (słowo int). Nie umieszczasz żadnej konkretnej wartości int przed kropką. Oto kilka przykładów:

* Wartość int.parse("1951") to liczba 1951.

* Wyrażenia takie jak 1951.parse("1951"), 1951.parse() i 1951.parse są nieprawidłowe. Żadne z nich nie działa, ponieważ w każdym przypadku wartość przed kropką nie jest nazwą klasy int. Zamiast tego wartość przed kropką jest obiektem — instancją klasy int.

* Umieszczenie dowolnego wyrażenia z wartością int przed .parse jest nieprawidłowe.

Na przykład następujący kod psuje twój program:

```
int numberOfClowns;

int otherNumber =

numberOfClowns.parse("2020");
```

Tworzenie statycznego członka nie jest wielkim problemem. Po prostu dodaj słowo static do swojej deklaracji członkowskiej, na przykład:

```
class Automobile {

static int numberOfWheels = 4;

}

void main() {

Automobile jalopy = Automobile();

// print(jalopy.numberOfWheels); This

is incorrect.

print(Automobile.numberOfWheels);

}

/*

* Output:

* 4

*/
```

Rysunek 7 wywołuje metodę int.parse Darta - naprawdę przydatna metoda! Ale Dart ma jeszcze lepszą metodę. Nazywa się int.tryParse. Jest bardzo podobny do int.parse, ale jest bezpieczniejszy w użyciu. Kiedy wywołasz int.tryParse('To nie jest liczba'), aplikacja nie wybuchnie ci w twarz.

**Objaśnienie 5**

Wiele zamieszania we wcześniejszych akapitach dotyczących metody initState odnosi się w równym stopniu do metody utylizacji Fluttera. Zanim klasa State zacznie działać, framework Flutter wywołuje metodę usuwania kodu. Na rysunku 7 metoda usuwania wykonuje trzy czynności:

* wywołuje metodę usuwania należącą do _nameFieldController.

Metodadispose dla _nameFieldController kasuje ten kontroler, zwalniając wszelkie zasoby, które kontroler akurat blokuje.

* Wywołuje metodę usuwania należącą do _incomeFieldController. GoodbayŻegnaj, _incomeFieldController. Cieszę się, że twoje zasoby są uwalniane.

* Wywołuje metodę usuwania klasy State. Metoda usuwania klasy State, solidnie wbudowana we framework Flutter, czyści wszelkie inne rzeczy związane z _MyHomePageState. Podobnie jak w przypadku initState, metoda usuwania twojego własnego kodu musi wywoływać funkcję super.dispose().

**Tworzenie przycisków radiowych**

Każda aplikacja randkowa zawiera pytanie o płeć użytkownika. W przypadku tego pytania Doris wybiera grupę przycisków radiowych. Listing 5 zawiera większość kodu przycisku radiowego Doris.

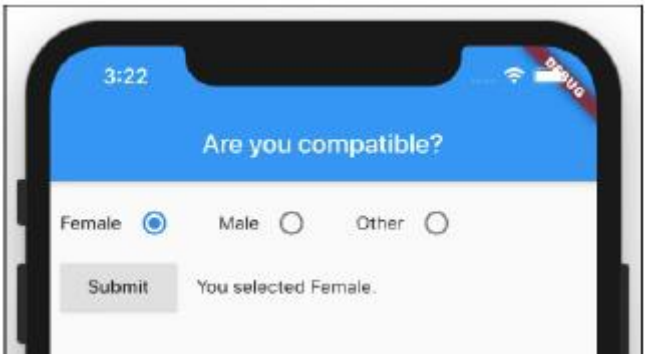
**LISTING 5 Jak rozpoznać?**

```
// This is not a complete program.

// Some trees have been saved.

// The trees are happy about that.

enum Gender { Female, Male, Other }

String shorten(Gender gender) =>

gender.toString().replaceAll("Gender.", "");

class _MyHomePageState extends

State<MyHomePage> {

String _messageToUser = "";

Gender _genderRadioValue;

// And later …

Widget _buildGenderRadio() {

return Row(

children: <Widget>[

Text(shorten(Gender.Female)),

Radio(

value: Gender.Female,

groupValue: _genderRadioValue,

onChanged: _updateGenderRadio,

),

SizedBox(width: 25.0),
```

```
Text(shorten(Gender.Male)),
Radio(
value: Gender.Male,
groupValue: _genderRadioValue,
onChanged: _updateGenderRadio,
),
SizedBox(width: 25.0),
Text(shorten(Gender.Other)),
Radio(
value: Gender.Other,
groupValue: _genderRadioValue,
onChanged: _updateGenderRadio,
),
],
);
}
Widget _buildResultArea() {
return Row(
children: <Widget>[
RaisedButton(
child: Text("Submit"),
onPressed: _genderRadioValue != null
? _updateResults : null,
),
SizedBox(
width: 15.0,
),
Text(
_messageToUser,
textAlign: TextAlign.center,
),
```

```
],
);
}
/// Actions
void _updateGenderRadio(Gender newValue) {
setState(() {
_genderRadioValue = newValue;
});
}
void _updateResults() {
setState(() {
_messageToUser =
"You selected
${shorten(_genderRadioValue)}.";
});
}
}
```

Rysunki poniższe  przedstawiają migawki z wykonania kodu na Listingu 5.

**Tworzenie wyliczenia**

Część 3 przedstawia wbudowane wyliczenie Brightness Fluttera wraz z jego wartościami Brightness.light i Brightness.dark. To miłe, ale po co pozwalać twórcom Fluttera na zabawę? Możesz zdefiniować własne wyliczenie, robiąc to, co widzisz na Listingu 5.

enum Gender { Female, Male, Other }

Dzięki tej deklaracji Twój kod ma trzy nowe wartości; mianowicie Płeć.Kobieta, Płeć.Mężczyzna i Płeć.Inne. Możesz użyć tych wartości w pozostałej części kodu aplikacji. Budowanie grupy opcji Kod z listingu 5 zawiera trzy przyciski opcji. Każdy przycisk opcji ma swoją własną wartość, ale razem wszystkie trzy przyciski mają tylko jedną wartość grupy. W rzeczywistości wspólna wartość grupy jest tym, co łączy ze sobą trzy przyciski. Gdy użytkownik wybierze przycisk z wartością Gender.Female, groupValue wszystkich trzech staje się Gender.Female. To tak, jakby część kodu nagle wyglądała tak:

// Don't try this at home. This is fake code.

Radio(

value: Gender.Female,

groupValue: Gender.Female,

),

Radio(

value: Gender.Male,

groupValue: Gender.Female,

),

Radio(

value: Gender.Other,

groupValue: Gender.Female,

),

Każdy przycisk opcji ma swój własny parametr onChanged. Na listingu 5 funkcja obsługująca zdarzenia onChanged (funkcja _updateGenderRadio) robi dokładnie to, czego można się spodziewać — zmienia wartość groupValue przycisków opcji na wartość wybraną przez użytkownika.

**PO CO SIĘ MĘCZYĆ?**

Czytelnik z Minnesoty pyta: „Do czego służy deklaracja enum z Listingu 5? Dlaczego nie mogę przypisać wartości String „Female",

„Mężczyzna" i „Inne" bezpośrednio do trzech przycisków opcji?" Dobre pytanie, czytelniku! Dzięki, że pytasz. Odpowiedź brzmi: „Masz rację. Możesz przypisać wartości String do przycisków opcji". Tak naprawdę nie potrzebujesz wyliczenia, aby utworzyć grupę przycisków radiowych. Poniższy kod bez wyliczenia jest prawidłowy:

String _genderRadioValue;

// And later …

```
Radio(
value: "Female",
groupValue: _genderRadioValue,
onChanged: _updateGenderRadio,
),
Radio(
value: "Male",
groupValue: _genderRadioValue,
onChanged: _updateGenderRadio,
),
Radio(
value: "Other",
groupValue: _genderRadioValue,
onChanged: _updateGenderRadio,
),
// And later …
void _updateGenderRadio(String newValue)
{
setState(() {
_genderRadioValue = newValue;
});
}
void _updateResults() {
setState(() {
_messageToUser = "You selected
$_genderRadioValue.";
});
}
```

Dlaczego więc na listingu 5 zawracam sobie głowę tworzeniem wyliczenia Gender? A odpowiedź brzmi: płcie nie są łańcuchami. Bycie mężczyzną nie oznacza, że dana osoba nosi ze sobą cztery litery m, potem a, potem l, a potem e. Zamiast tego męskość jest jedną z dwóch lub więcej możliwości, inną możliwością jest kobiecość. Najlepszym sposobem przedstawienia płci w kodzie jest wyliczenie

alternatyw, a nie użycie kilku ciągów znaków i nadzieja, że nikt ich nie przeliteruje. Rozważ ten kod, który używa typu String:

```
String _genderRadioValue = "Femail";
```

Kod jest niepoprawny, ale jeśli chodzi o język Dart, kod jest brzoskwiniowy. Rozważmy teraz ten kod, który używa typu enum:

```
enum Gender { Female, Male, Other }

Gender _genderRadioValue = Gender.Femail;
```

Kod jest nieprawidłowy, a Dart odmawia jego przyjęcia. Dzięki deklaracji wyliczenia płci programista gwarantuje, że jedynymi możliwymi wartościami _genderRadioValue są Gender.Female, Gender.Male i Gender.Other. To dobra praktyka programistyczna. Bezpieczeństwo przede wszystkim!

**Wyświetlanie wyboru użytkownika**

Metoda short z listingu 5 jest obejściem nieco irytującej funkcji języka Dart. W Dart każda wartość wyliczeniowa ma metodę toString, która teoretycznie zapewnia użyteczny sposób wyświetlania wartości. Problem polega na tym, że po zastosowaniu metody toString wynikiem jest zawsze pełna nazwa. Na przykład Gender.Female.toString() to „Płeć.Kobieta”, a to nie jest dokładnie to, co chcesz wyświetlić. Na rysunku 7.10 użytkownik widzi zdanie Wybrałeś kobietę zamiast zbyt technicznego zdania Wybrałeś płeć. Kobieta. Zastosowanie wywołania metody replaceAll("Gender.", "") zmienia wartość na "Płeć". do pustego ciągu, więc „Płeć.Kobieta” staje się zwykłym starym „Kobietą”. Problem rozwiązany! - albo może nie. Spójrz na deklarację _genderRadioValue na Listingu 5:

```
Gender _genderRadioValue;
```

Ta deklaracja nie przypisuje niczego do _genderRadioValue, więc _genderRadioValue zaczyna być puste. To dobrze, ponieważ posiadanie wartości null dla _genderRadioValue oznacza, że żaden z przycisków grupy radiowej nie jest zaznaczony. Właśnie tego chcesz, gdy aplikacja zaczyna działać. Ale co, jeśli użytkownik naciśnie przycisk Prześlij bez wybierania jednego z przycisków opcji? Wtedy _genderRadioValue nadal ma wartość null, więc parametr płci metody short ma wartość null. W środku

short metoda, masz następującą nieprzyjemną sytuację:

```
String shorten(Gender null) =>

null.toString().replaceAll("Gender.", "");
```

Ups! Gdy zastosujesz metodę toString() do wartości null, otrzymasz „null” (ciąg składający się z czterech znaków). Jeśli nic z tym nie zrobisz, komunikat na ekranie użytkownika stanie się Wybrałeś null. To nie jest przyjazne dla użytkownika!

„null” to czteroliterowe słowo.

Na listingu 5 procedura obsługi onPressed przycisku Submit reaguje odpowiednio, gdy _genderRadioValue ma wartość null.

Oto kod:

```
RaisedButton(

child: Text("Submit"),
```

```
onPressed: _genderRadioValue != null ?

_updateResults : null,

),
```

Jeśli _genderRadioValue nie ma wartości NULL, procedurą obsługi jest ładna, konwencjonalna metoda _updateResults — metoda, która tworzy obiekt _messageToUser i wyświetla tę wiadomość w widgecie Tekst. Nic specjalnego. Ale gdy _genderRadioValue ma wartość null, przycisk Prześlij jest wyświetlany

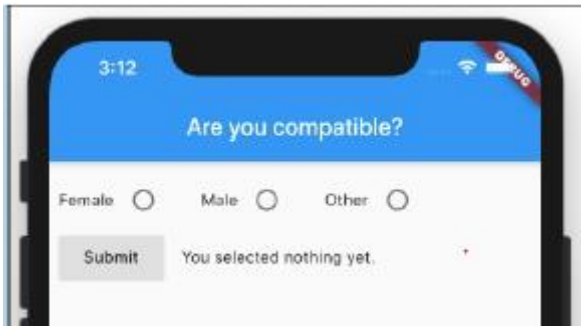
Program obsługi onPressed jest również pusty. Dobrą wiadomością jest to, że RaisedButton z pustym modułem obsługi jest całkowicie wyłączony. Użytkownik widzi przycisk, ale jego powierzchnia jest wyszarzona, a naciśnięcie przycisku nie daje żadnego efektu. To wspaniale! Jeśli nie zostanie wybrana żadna płeć, a użytkownik spróbuje nacisnąć przycisk Prześlij, nic się nie stanie i nie pojawi się żaden komunikat.

**¿CO ROBIĆ? I ?? DO?**

Na listingu 5 przycisk Prześlij jest martwy, dopóki użytkownik nie wybierze płci. Czy istnieją inne sposoby podejścia do problemu braku wyboru? Ten pasek boczny przedstawia dwie alternatywy. Po pierwsze, oto nudna alternatywa:

```
RaisedButton(

child: Text("Submit"),

onPressed: _updateResults,

),

// And later in the code …

void _updateResults() {

setState(() {

if (_genderRadioValue != null) {

_messageToUser = "You selected

${shorten(_genderRadioValue)}.";

} else {

_messageToUser = "You selected

nothing yet.";

}

});

}
```

W tej wersji kodu przycisk Prześlij jest zawsze włączony i metoda obsługująca kliknięcie przycisku (metoda _updateResults) traktuje wartość null _genderRadioValue jako przypadek specjalny. (Zobacz pierwszy rysunek na pasku bocznym.) Umieszczenie instrukcji if w metodzie _updateResults z pewnością działa, ale, jak mówię, jest nudne.




To, co nie jest nudne, to operatorzy Darta świadomi wartości null. Oto kod:

```
String shorten(Gender gender) =>

gender?.toString()?.replaceAll("Gender.",

"");

_messageToUser =

"You selected

${shorten(_genderRadioValue) ?? 'nothing

yet'}.";
```

Operator świadomy wartości null to thingamajig, który robi coś specjalnego, gdy zastosuje się go do wartości null. Weźmy na przykład Darta ?. operator.

* Jeśli płeć ma wartość null, to gender.toString() ma wartość „null”. Tak się dzieje, gdy nie używasz Darta?. operator. W Dart wszystko ma swoją własną metodę toString. Jaka jest lepsza reprezentacja ciągu dla wartości pustej niż ciąg „null”?

* Jeśli płeć ma wartość null, to gender?.toString() ma wartość null. Tak się dzieje, gdy używasz Darta?. operator. Zawsze, gdy jakaśWartość ma wartość null, jakaśWartość?.coś_else ma wartość zero. Taka jest zasada.

Kolejny operator Darta świadomy wartości null — ?? operator - może zająć się wartością pustą. Wyrażenie short(_genderRadioValue) ?? „nothing yet” oznacza skrócenie (_genderRadioValue) lub „nothing yet”. Aby być trochę bardziej precyzyjnym,

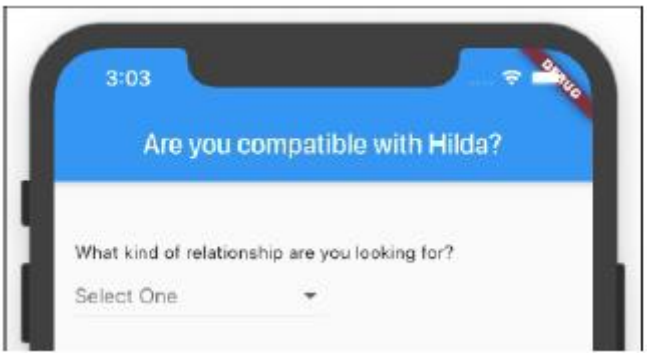
* ?? wyrażenie oznacza short(_genderRadioValue), chyba że short(_genderRadioValue) ma wartość null.

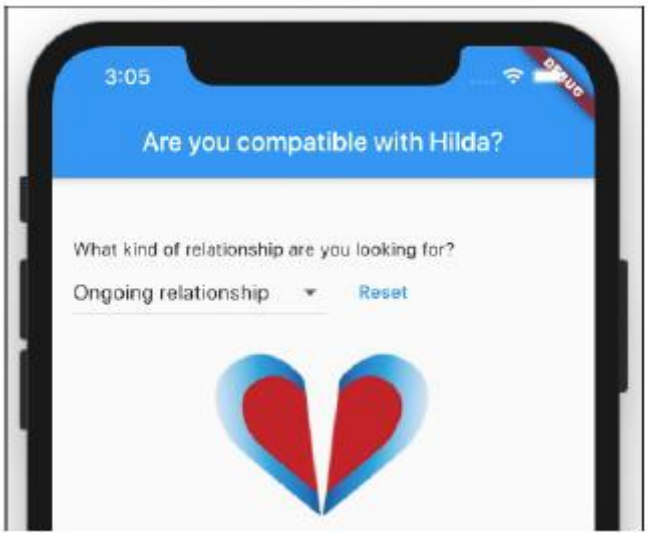
* Jeśli short(_genderRadioValue) ma wartość NULL, ?? wyrażenie oznacza „nothing yet”.

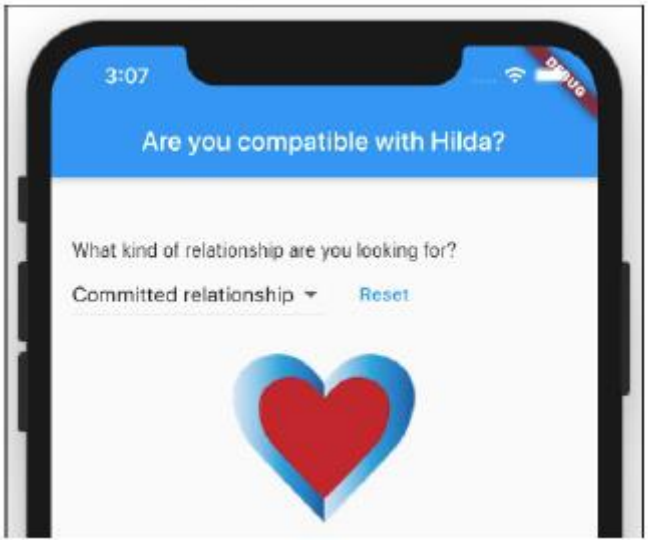
?? operator ma nazwę. Nazywa się to operatorem if-null. Kiedy użyjesz operatora if-null w połączeniu z ?. operatorów, otrzymasz wynik podobny do tego na pierwszym rysunk.

**Tworzenie przycisku rozwijanego**

Gdy tylko rozejdą się wieści o aplikacji randkowej Doris, wszyscy chcą wziąć udział w akcji. Przyjaciółka Doris, Hilda, potrzebuje przycisku rozwijanego, który pozwoli ocenić poziom zaangażowania potencjalnego partnera. Hilda chce zaangażowanego związku i być może małżeństwa. Listing 6 pokazuje część kodu, który Doris pisze dla Hildy. Rysunki przedstawiają kod w akcji.




**LISTING 6 Czego szukasz?**

// This listing is missing some parts. See

App0705 in the files that

// you download from this book's website

(www.allmycode.com/Flutter).

```
enum Relationship {
Friend,
OneDate,
Ongoing,
Committed,
Marriage,
}
Map<Relationship, String> show = {
Relationship.Friend: "Friend",
Relationship.OneDate: "One date",
Relationship.Ongoing: "Ongoing relationship",
Relationship.Committed: "Committed
relationship",
Relationship.Marriage: "Marriage",
};
List<DropdownMenuItem<Relationship>>
_relationshipsList = [
DropdownMenuItem(
value: Relationship.Friend,
child: Text(show[Relationship.Friend]),
),
DropdownMenuItem(
value: Relationship.OneDate,
child: Text(show[Relationship.OneDate]),
),
DropdownMenuItem(
value: Relationship.Ongoing,
child: Text(show[Relationship.Ongoing]),
),
DropdownMenuItem(
value: Relationship.Committed,
```

```
child: Text(show[Relationship.Committed]),
),
DropdownMenuItem(
value: Relationship.Marriage,
child: Text(show[Relationship.Marriage]),
),
];
class _MyHomePageState extends
State<MyHomePage> {
Relationship _relationshipDropdownValue;
// And later in the program …
/// Build
Widget _buildDropdownButtonRow() {
return Row(
mainAxisAlignment:
MainAxisAlignment.start,
children: <Widget>[
DropdownButton<Relationship>(
items: _relationshipsList,
onChanged:
_updateRelationshipDropdown,
value: _relationshipDropdownValue,
hint: Text("Select One"),
),
if (_relationshipDropdownValue != null)
FlatButton(
child: Text(
"Reset",
style: TextStyle(color:
Colors.blue),
),
```

```
onPressed: _reset,
),
],
);
}
Widget _buildResultsImage() {
if (_relationshipDropdownValue != null) {
return
Image.asset((_relationshipDropdownValue.index
>= 3)
? "Heart.png"
: "BrokenHeart.png");
} else {
return SizedBox();
}
}
/// Actions
void _reset() {
setState(() {
_relationshipDropdownValue = null;
});
}
void _updateRelationshipDropdown(Relationship
newValue) {
setState(() {
_relationshipDropdownValue = newValue;
});
}
}
```

**Tworzenie przycisku rozwijanego**

Konstruktor DropdownButton ma kilka parametrów, z których jeden jest listą elementów. Każdy element jest instancją klasy DropdownMenuItem. Każda taka instancja ma wartość i dziecko.

* Wartość przedmiotu to coś, co identyfikuje ten konkretny przedmiot. Na listingu 7.6 wartości pozycji to Relationship.Friend, Relationship.OneDate i tak dalej. Wszyscy są członkami wyliczenia Relationship. Nie chcesz takich rzeczy jak Relationship.OneDate pojawiające się na powierzchni elementu menu, więc…

* Element potomny elementu to element wyświetlany na tym elemencie.

Na listingu 6 wszystkie elementy podrzędne elementów to widżety tekstowe, ale elementy rozwijane umożliwiają wyświetlanie wszelkiego rodzaju elementów. Na przykład element potomny elementu może być wierszem zawierającym widżet Tekst i widżet Ikona. Oprócz listy elementów konstruktor DropdownButton ma parametry onChanged, value i podpowiedzi.

* Parametr onChanged robi to samo, co takie parametry w wielu innych konstruktorach. Parametr odnosi się do funkcji, która obsługuje stuknięcia, naciśnięcia, poprawki i szturchnięcia użytkownika.

* W dowolnym momencie parametr wartości odnosi się do wybranego elementu przycisku rozwijanego.

*Parametr podpowiedzi mówi Flutterowi, co wyświetlić, gdy żaden z elementów przycisku rozwijanego nie został wybrany.

W przykładzie z tej sekcji Flutter wyświetla słowa Wybierz jeden.

Podpowiedź przycisku rozwijanego jest zwykle wyświetlana, zanim użytkownik wybierze którykolwiek z elementów przycisku. Ale na listingu 7.6 znajduje się przycisk Reset. Gdy użytkownik naciśnie przycisk Reset, procedura obsługi przycisku onPressed ustawia _relationshipDropdownValue z powrotem na wartość null, więc

wskazówka przycisku rozwijanego pojawi się ponownie.

**Mały przycisk resetowania**

Przycisk Reset na listingu 7.6 jest interesujący z więcej niż jednego powodu. Po pierwsze, nie jest to RaisedButton. Zamiast tego jest to FlatButton. FlatButton jest podobny do RaisedButton, z tą różnicą, że… cóż, FlatButton jest płaski. (Patrz rysunki 7-13 i 7-14.) Kolejnym powodem, dla którego warto zagłębić się w kod przycisku Reset, jest specyficzna funkcja języka Dart — dostępna dopiero od wersji Dart 2.3. Oto skrócona wersja kodu metody _buildDropdownButtonRow z Listingu 7-6:

```
Widget _buildDropdownButtonRow() {

return Row(

children: <Widget>[

DropdownButton<Relationship>(

),

if (_relationshipDropdownValue != null)

FlatButton(
```

),

],

);

}

W tym kodzie parametr children widżetu Row jest listą, a lista składa się z dwóch elementów: przycisku DropdownButton i czegoś, co wygląda jak instrukcja if. Ale pozory mogą mylić. Rzecz na listingu 6 nie jest instrukcją if. Rzecz na listingu 6 to kolekcja if. W Części 4 bezceremonialnie wkradnę się do kolekcji słów, aby opisać typy Listy, Zestawu i Mapy Darta. Kolekcja if pomaga zdefiniować instancję jednego z tych typów. Na listingu 6 znaczenie kolekcji if jest dokładnie takie, jak można się domyślić. Jeśli _relationshipDropdownValue nie jest puste, lista zawiera element FlatButton. W przeciwnym razie lista nie zawiera elementu FlatButton. Ma to sens, ponieważ gdy _relationshipDropdownValue ma wartość null, nie ma sensu oferować użytkownikowi opcji zmiany wartości na zero. Oprócz kolekcji if, język programowania Dart ma kolekcję for.

**Tworzenie mapy**

Część 4 wprowadza typy Darta, z których jednym jest typ mapy. Mapa jest bardzo podobna do słownika. Aby znaleźć definicję słowa, wyszukaj słowo w słowniku. Aby znaleźć przyjazną dla użytkownika reprezentację wartości wyliczeniowej Relationship.OneDate, wyszukaj Relationship.OneDate na mapie pokazu.

Relationship.Friend /ri-lAY-shuhn-ship frEnd/ n.

Friend.

Relationship.OneDate /ri-lAY-shuhn-ship wUHn

dAYt/ n. One date.

Relationship.Ongoing /ri-lAY-shuhn-ship AWn-gohing/

adj. Ongoing relationship.

Relationship.Committed /ri-lAY-shuhn-ship kuh-mItuhd

/ adj. Committed relationship.

Relationship.Marriage /ri-lAY-shuhn-ship mAIR-ij /

n. Marriage.

Mówiąc dokładniej, mapa to zbiór par, z których każda składa się z klucza i wartości. Na listingu 7.6 zmienna show odnosi się do mapy, której kluczami są Relationship.Friend, Relationship.OneDate i tak dalej. Wartości mapy to „Przyjaciel”, „Jedna randka”, „Trwający związek” i tak dalej.

Patrz Tabela 7-1.

Key : Value : Index

Relationship.Friend : "Friend" : 0

Relationship.OneDate : "One date" : 1

Relationship.Ongoing : "Ongoing relationship" : 2

Relationship.Committed : "Committed relationship" : 3

Relationship.Marriage : "Marriage" : 4

W programie Dart używasz nawiasów, aby wyszukać wartość na mapie. Na przykład na listingu 7.6 wyszukiwanie show[Relationship.OneDate] daje ciąg „Jedna data”. Oprócz kluczy i wartości każdy wpis mapy ma indeks. Indeksem wpisu jest jego numer pozycji w deklaracji mapy, zaczynając od pozycji 0. Koleżanka Doris, Hilda, chce zaangażowanego związku i ewentualnie małżeństwa. Zatem kod z listingu 7.6 sprawdza ten warunek: gdy ten warunek jest spełniony, aplikacja wyświetla serce, wskazując dobre dopasowanie. W przeciwnym razie aplikacja wyświetla złamane serce. (Przepraszam, Hilda.)

**Naprzód i w górę**

Praca Doris nad aplikacją randkową bardzo się opłaciła. Doris jest teraz w zaangażowanym związku z równie geekowskim programistą Fluttera — który ma dobrze ponad 18 lat i zarabia wystarczająco dużo pieniędzy, by żyć wygodnie. Doris i jej partner będą żyli długo i szczęśliwie, a przynajmniej do czasu, gdy Google zmieni specyfikację języka Dart i złamie część kodu Doris. Następny rozdział dotyczy nawigacji. Jak Twoja aplikacja może przechodzić z jednej strony na drugą? Kiedy użytkownik zakończy korzystanie z nowej strony, w jaki sposób Twoja aplikacja może wrócić? Jeśli w Twojej aplikacji jest więcej niż jedna strona, w jaki sposób strony mogą udostępniać informacje?

**Nawigacja, listy i inne korzyści**

Badając tą część, dowiedziałem się kilku interesujących rzeczy o sztuce przewracania stron:

* W sieci jest wiele witryn, które pomagają muzykom rozwiązać problemy związane z przewracaniem stron. Niektóre witryny oferują porady dotyczące najlepszych sposobów ręcznego przewracania stron. Inne oferują rozwiązania mechaniczne z pedałami nożnymi do sterowania urządzeniami przerzucającymi. Artykuły naukowe zawierają przegląd alternatyw i wyciągają wnioski na podstawie badań.

* Dla niemuzyków, kilka witryn opisuje urządzenia do przewracania stron. Żadne z tych urządzeń nie poprawia domowego wyglądu salonu lub gabinetu.

* Jedna witryna opisuje w dziesięciu krokach, jak człowiek może przewracać strony książki za pomocą rąk. Witryna zawiera ilustracje i szczegółowe instrukcje dotyczące każdego z dziesięciu kroków. (Myślałem, że już wiem, jak przewracać strony, ale może się myliłem!)

* Na jednym forum widziałem link do strony, która sprzedaje najbardziej profesjonalne urządzenie do przewracania stron. Kiedy kliknąłem łącze, powtarzające się zdanie oznajmiło: „Przepraszamy, żądana strona nie została znaleziona”. Być może mieli na myśli: „Nie można znaleźć żądanego urządzenia do przewracania stron”.

* Klasa Navigator Fluttera może przechodzić w aplikacji z jednej strony na drugą. W rzeczywistości funkcje nawigacyjne Fluttera są tak ważne, że jedna książka ma cały rozdział poświęcony temu tematowi.

**Rozszerzanie klasy Dart**

Jeśli nie będę ostrożny, wykazy kodów w tej książce mogą stać się nieznośnie długie. Prosty przykład ilustrujący jedną nową koncepcję może zająć kilka stron. Potrzebujesz magicznych mocy, aby znaleźć nowy i interesujący kod każdej aukcji. Aby zwalczyć tę trudność, dzielę przykłady niektórych sekcji na dwa pliki — jeden plik zawiera kod wzorcowy, a drugi zawiera nowe funkcje sekcji. Kiedy przechodzę z jednej sekcji do drugiej, ponownie wykorzystuję plik zawierający kod wzorcowy i wprowadzam osobny plik zawierający tylko nowe funkcje. Wszystko jest dobrze, dopóki nie spróbuję podzielić kodu określonej klasy między dwa pliki. Wyobraź sobie, że mam dwa pliki. Jeden plik nazywa się ReuseMe.dart:

```
// This is ReuseMe.dart

import 'MoreCode.dart';

class ReuseMe {

int x = 229;

}

main() => ReuseMe().displayNicely();
```

Nazwa innego pliku to MoreCode.dart.

```
// This is a bad version of MoreCode.dart

import 'ReuseMe.dart';

void displayNicely() {
```

```
print('The value of x is $x.');

}
```

Co może pójść nie tak?

Oto, co poszło nie tak: Deklaracja displayNicely nie znajduje się w klasie ReuseMe. W tej parze plików displayNicely to samotna funkcja, która znajduje się poza jakąkolwiek konkretną klasą. Powoduje to dwa problemy:

* Linia ReuseMe().displayNicely() nie ma sensu.

* Funkcja displayNicely nie może przypadkowo odwoływać się do zmiennej x klasy ReuseMe.

Ten kod jest fałszywy. Wyrzuć to!

Ale poczekaj! Podstępna sztuczka może uratować ten przykład. Od wersji Dart 2.7 mogę dodawać metody do klasy bez umieszczania ich w kodzie klasy. Wskazuję tę firmę za pomocą słowa kluczowego rozszerzenia Darta. Oto jak to zrobić:

```
// This is a good version of MoreCode.dart

import 'ReuseMe.dart';

extension MyExtension on ReuseMe {

void displayNicely() {

print('The value of x is $x.');

}

}
```

Po dokonaniu tej zmiany funkcja displayNicely staje się metodą należącą do klasy ReuseMe. Dart zachowuje się tak, jakbym napisał następujący kod:

```
// The extension keyword makes Dart pretend

that I wrote this code:

class ReuseMe {

int x = 229;

void displayNicely() {

print('The value of x is $x.');

}

}
```

Wewnątrz ciała metody displayNicely nazwa x odnosi się do zmiennej x klasy ReuseMe. A każda instancja klasy ReuseMe ma metodę displayNicely. Tak więc wywołanie ReuseMe().displayNicely() ma sens. Wszystko działa poprawnie. A co najważniejsze, mogę zamienić plik MoreCode.dart na inną wersję pliku, kiedy tylko zechcę.

```
// Another good version of MoreCode.dart
```

```
import 'ReuseMe.dart';

extension MyExtension on ReuseMe {

void displayNicely() {

print(' * $x * ');

print(' ** $x ** ');

print(' *** $x ***');

print('**** $x ****');

}

}
```

Mogę zmienić funkcję displayNicely bez dotykania pliku zawierającego oryginalną deklarację klasy ReuseMe. To przydatne!

Rozszerzenia nie są dostępne we wszystkich wersjach Dart. Jeśli Android Studio narzeka na używanie rozszerzeń, poszukaj sekcji środowiska w pliku pubspec.yaml swojego projektu. Ta sekcja środowiska może wyglądać mniej więcej tak:

```
environment:

sdk: ">=2.1.0 <3.0.0"
```

Zmień niższy numer wersji Darta w następujący sposób:

```
environment:

sdk: ">=2.6.0 <3.0.0"
```

Nazwa rozszerzenia odróżnia to rozszerzenie od innych rozszerzeń w tej samej klasie. Na przykład wyobraź sobie, że ja zdefiniowałem MyExtension, a ty zdefiniowałeś YourExtension, oba w klasie ReuseMe:

```
extension YourExtension on ReuseMe {

void displayNicely() {

print('!!! $x !!!');

}

}
```

W przypadku dwóch rozszerzeń deklarujących metody displayNicely wyrażenie ReuseMe().displayNicely() jest niejednoznaczne. Aby wyjaśnić zamieszanie, wyraźnie nazwij jedno z rozszerzeń:

```
YourExtension(ReuseMe()).displayNicely()
```

**Z jednej strony na drugą**

Prawdopodobnie korzystałeś z aplikacji z interfejsem główny-szczegółowy. Interfejs główny-szczegółowy ma dwie strony. Na pierwszej stronie wyświetlana jest lista elementów. Gdy użytkownik

wybierze element z listy, na drugiej stronie zostaną wyświetlone szczegółowe informacje o tym elemencie. Pierwszy przykład z tego rozdziału (na Listingach 1 i 2) ma uproszczony interfejs masterdetail. I dlaczego mówię „rozebrany”? Lista strony wzorcowej składa się tylko z jednego elementu — nazwy konkretnego filmu.

**LISTING 1 Użyj ponownie tego kodu**

```
// App08Main.dart

import 'package:flutter/material.dart';

import 'App0802.dart'; // Change this line to

App0803, App0804, and so on.

void main() => runApp(App08Main());

class App08Main extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: MovieTitlePage(),

);

}

}

class MovieTitlePage extends StatefulWidget {

@override

MovieTitlePageState createState() =>

MovieTitlePageState();

}

class MovieTitlePageState extends

State<MovieTitlePage> {

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(

title: Text(

'Movie Title',

),
```

```
),
body: Padding(
padding: const EdgeInsets.all(16.0),
child: Center(
child: buildTitlePageCore(),
),
),
);
}
}
class DetailPage extends StatelessWidget {
final overview = '(From themoviedb.com) One
day at work, unsuccessful '
'puppeteer Craig finds a portal into the
head of actor John '
'Malkovich. The portal soon becomes a
passion for anybody who '
'enters its mad and controlling world of
overtaking another human '
'body.';
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text(
'Details',
),
),
body: Padding(
padding: const EdgeInsets.all(16.0),
child: Center(
```

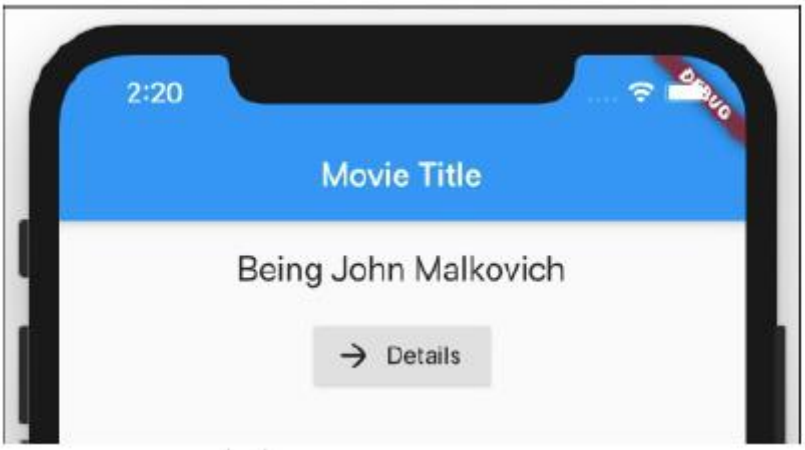
```
child: buildDetailPageCore(context),
),
),
);
}
}
```

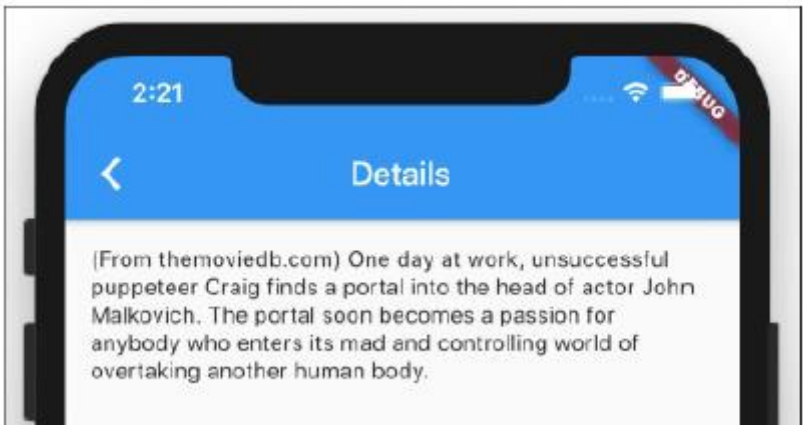
**LISTING 2 Podstawowa nawigacja**

```
// App0802.dart
import 'package:flutter/material.dart';
import 'App08Main.dart';
extension MoreMovieTitlePage on
MovieTitlePageState {
goToDetailPage() {
Navigator.push(
context,
MaterialPageRoute(
builder: (context) => DetailPage(),
),
);
}
Widget buildTitlePageCore() {
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(
'Being John Malkovich',
textScaleFactor: 1.5,
),
SizedBox(height: 16.0),
RaisedButton.icon(
```

```
icon: Icon(Icons.arrow_forward),
label: Text('Details'),
onPressed: goToDetailPage,
),
],
);
}
}
extension MoreDetailPage on DetailPage {
Widget buildDetailPageCore(context) {
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(
overview,
),
],
);
}
}
```

Aby uruchomić pierwszą aplikację z tego rozdziału, Twój projekt musi zawierać zarówno Listing 1, jak i Listing 2. Każda z tych aukcji jest zależna od kodu z innej aukcji. W rzeczywistości wiele list z tego rozdziału jest zależnych od kodu z listingu 1. Listy 1 i 2 muszą znajdować się w osobnych plikach .dart, ponieważ obie listy zawierają deklaracje importowe. Listing 2 nie zawiera metody main. Tak więc, aby uruchomić aplikację na Listingach 1 i 2, poszukaj zakładki App08Main.dart nad edytorem Android Studio. Kliknij tę kartę prawym przyciskiem myszy, a następnie wybierz Uruchom „App08Main.dart" z wyświetlonego menu. Rysunki 1 i 2 przedstawiają strony wygenerowane przez kod z Listingów 1 i 2.




Rysunek 1 przedstawia stronę startową aplikacji — stronę z RaisedButton. Kiedy użytkownik naciśnie ten przycisk, Flutter wywołuje metodę goToDetailPage z Listingu 2. Metoda goToDetailPage wywołuje metodę push klasy Navigator. Parametry metody push wskazują bezpośrednio na klasę DetailPage. Aplikacja przechodzi więc do swojej drugiej strony — strony DetailClass na rysunku 2. W lewym górnym rogu rysunku 2 znajduje się mała strzałka skierowana do tyłu. Flutter tworzy tę strzałkę automatycznie za każdym razem, gdy przechodzi do strony z paskiem aplikacji. Gdy użytkownik naciśnie tę strzałkę, aplikacja powraca do pierwszej strony — strony MovieTitlePage.

**Ikona na przycisku**

Dla odrobiny uroku dodaję ikonę (mała strzałka skierowana do przodu) do przycisku RaisedButton na rysunku 1. Aby tak się stało, wielokrotnie używam słowa ikona na Listingu 2. Zamiast wywoływać zwykły konstruktor RaisedButton, wywołuję konstruktor RaisedButton.icon Fluttera. Następnie dla parametru icon konstruktora zapisuję Icon(Icons.arrow_forward), co oznacza: „Zbuduj rzeczywisty widżet Icon, którego wygląd odpowiada wbudowanej wartości Icons.arrow_forward Fluttera”. Flutter ma całą masę wbudowanych ikon. Większość z nich to znane ikony interfejsu użytkownika, takie jak zwiększanie głośności, ostrzeżenie i sygnał_komórkowy_4_bar. Ale innych nie spodziewasz się znaleźć. Na przykład Flutter ma ikonę zwierzaka (zdjęcie łapy), ikonę kasyna (twarz kostki) oraz ikonę linii lotniczych_seat_legroom:reduced (osoba wciśnięta w małą przestrzeń).

**Pchanie i wyskakiwanie**

Oto kilka przydatnych terminów:

* Strona wywołująca Navigator.push jest stroną źródłową. Na listach 1 i 2 MovieTitlePage jest stroną źródłową.

* Strona, którą widzi użytkownik w wyniku wywołania Navigator.push jest stroną docelową.

Na listach 1 i 2 strona DetailPage jest stroną docelową.

Niektóre przejścia prowadzą ze strony źródłowej do strony docelowej; inne przechodzą ze strony docelowej z powrotem do strony źródłowej. W przykładzie z tej sekcji

* Użytkownik naciska przycisk RaisedButton na rysunku 1, aby przejść od źródła do miejsca docelowego.

* Użytkownik naciska przycisk Wstecz na pasku aplikacji na rysunku 2, aby przejść z miejsca docelowego do źródła.

W większości przejścia aplikacji mobilnej tworzą strukturę znaną jako stos. Aby utworzyć stos, każdą nową stronę układasz na wierzchu wszystkich istniejących stron. Następnie, gdy będziesz gotowy do usunięcia strony, usuń stronę znajdującą się na szczycie stosu. To jak system starszeństwa dla stron. Najmłodsza strona jest usuwana jako pierwsza. Dzięki tej zasadzie Last-In-First-Out (LIFO) użytkownik tworzy jasny mentalny obraz swojego miejsca wśród stron aplikacji.

Oto trochę więcej terminologii:

* Kiedy dodajesz coś na wierzch stosu, wpychasz to na stos.

* Kiedy usuwasz coś ze szczytu stosu, zdejmujesz to ze stosu.

Na listingu 2 nazwa Navigator.push sugeruje umieszczenie strony na stosie stron. W rzeczywistości, kiedy myślę o przejściach stron, zawsze wyobrażam sobie strony na górze stron. Najnowsza strona zasłania starsze strony, które znajdują się pod nią. Podczas uruchamiania pierwszej aplikacji tego rozdziału strona DetailPage jest wygodnie umieszczona na stronie MovieTitlePage, całkowicie zasłaniając stronę MovieTitlePage przed wzrokiem użytkownika. W niektórych sytuacjach koncepcja układania jednej strony na drugiej nie jest odpowiednia. Może nie chcesz umieszczać strony docelowej na górze strony źródłowej. Zamiast tego chcesz zastąpić stronę źródłową stroną docelową. Aby to zrobić na listingu 8.2, dokonujesz jednej drobnej zmiany: zamieniasz słowa Navigator.push na słowa Navigator.pushReplacement. Gdy to zrobisz, MovieTitlePage będzie wyglądać tak, jak na rysunku 1, ale strona DetailPage różni się nieco od obrazu na rysunku 2. W nowym DetailPage pasek aplikacji nie ma przycisku Wstecz.We Flutterze ekrany i strony nazywane są routes. Dlatego listing 2 zawiera wywołanie konstruktora MaterialPageRoute. Aby Twoja aplikacja wyglądała jak aplikacja na iPhone'a, użyj widżetów Flutter Cupertino zamiast widżetów Material Design i utwórz CupertinoPageRoute zamiast MaterialPageRoute. CupertinoPageRoute sprawia, że przejścia między stronami wyglądają jak Apple.

**Przekazywanie danych ze źródła do miejsca docelowego**

Czasami chcesz przekazać informacje z jednej strony na drugą. Następny przykład pokazuje, w jaki sposób źródło wysyła informacje do miejsca docelowego. Zanim spróbujesz uruchomić aplikację z tej sekcji, zmień jedną z linii importu na listingu 8.1. Zmień „App0802.dart” na „App0803.dart”. Wprowadź podobne zmiany, aby uruchomić Listingi 3, 4, 5, 7, 8 i 10.

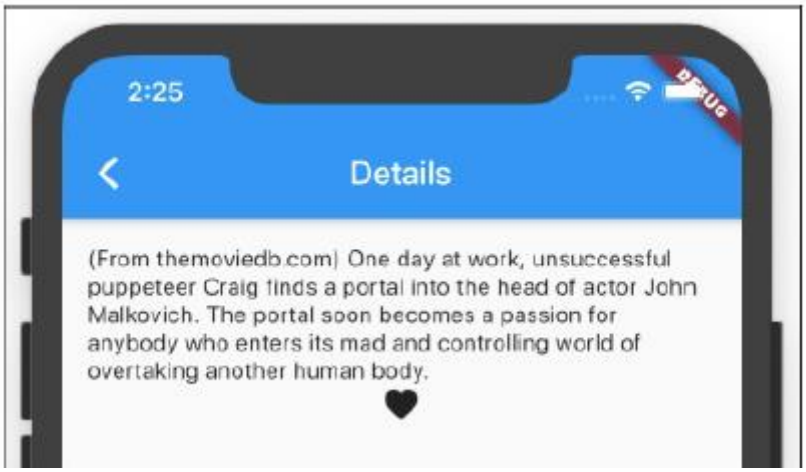
**LISTING 3 Od strony tytułowej filmu do strony szczegółów**

```
// App0803.dart

import 'package:flutter/material.dart';

import 'App08Main.dart';
```

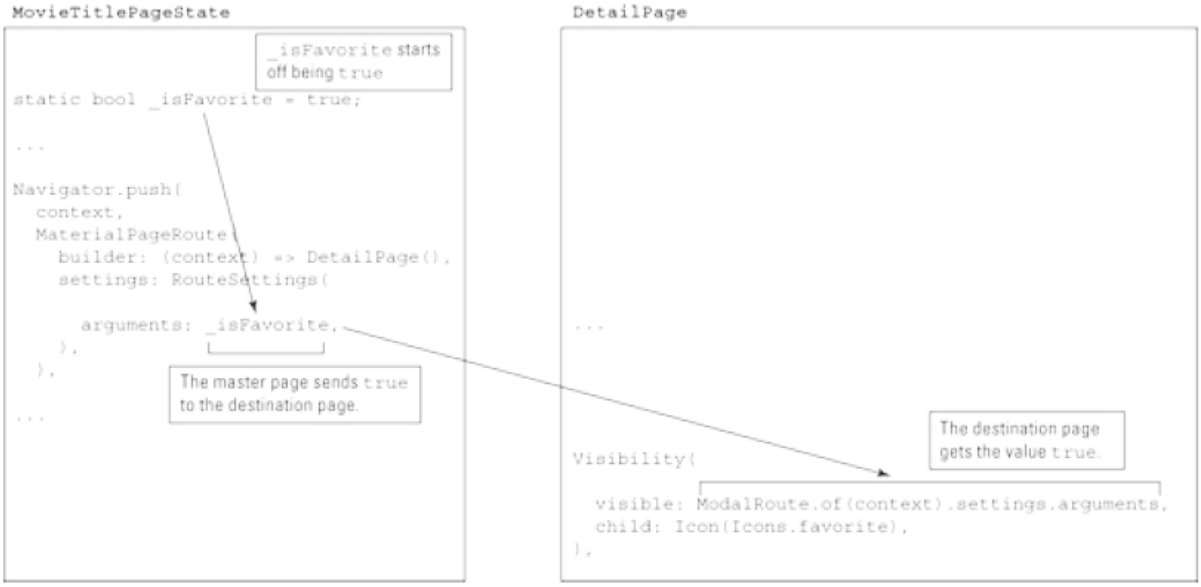
```
extension MoreMovieTitlePage on
MovieTitlePageState {
static bool _isFavorite = true; // You can
change this to false.
goToDetailPage() {
Navigator.push(
context,
MaterialPageRoute(
builder: (context) => DetailPage(),
settings: RouteSettings(
arguments: _isFavorite,
),
),
);
}
Widget buildTitlePageCore() {
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(
'Being John Malkovich',
textScaleFactor: 1.5,
),
SizedBox(height: 16.0),
RaisedButton.icon(
icon: Icon(Icons.arrow_forward),
label: Text('Details'),
onPressed: goToDetailPage,
),
],
```

```
);
}
}
extension MoreDetailPage on DetailPage {
Widget buildDetailPageCore(context) {
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(
overview,
),
Visibility(
visible:
ModalRoute.of(context).settings.arguments ??
false,
child: Icon(Icons.favorite),
),
],
);
}
}
```

Rysunek 3 przedstawia stronę DetailPage wygenerowaną przez kod z Listingów 1 i 3. Małe serduszko wskazuje, że Być jak John Malkovich to ulubiony film.




Rysunek 4 ilustruje błąd wywołany wartością zmiennej _isFavorite w przebiegu przykładu z tej sekcji.




Gdy Flutter wyświetla stronę szczegółów, wartość ModalRoute.of(context).settings.arguments jest równa true. To tak, jakby kod w dolnej części Listingu 3 wyglądał tak:

// Remember, I said "as if" the code looked

like this…

Visibility(

visible: true,

child: Icon(Icons.favorite),

),

Widżet Widoczność pokazuje lub ukrywa swoje dziecko w zależności od wartości jego widocznego parametru. Tak więc w tym przykładzie wbudowana ulubiona ikona Fluttera pojawia się na ekranie użytkownika. Na listingu 8.3 możesz zmienić deklarację _isFavorite w następujący sposób:

static bool _isFavorite = false;

Gdy to zrobisz, strona tytułowa filmu przekaże wartość false na stronę szczegółów. Właściwość widżetu visible staje się więc fałszywa, a mała ulubiona ikona nie pojawia się. Na listingu 3 zmienna _isFavorite jest statyczna. Jedną z konsekwencji tego jest to, że ponowne uruchomienie aplikacji na gorąco nie działa. Jeśli zmienisz _isFavorite z true na false, a następnie zapiszesz swój kod, ikona małego serca nie zniknie. Aby ta zmiana wartości _isFavorite zaczęła obowiązywać, zatrzymaj działanie aplikacji, a następnie uruchom ją ponownie.

**RUBE GOLDBERG BYŁBY ZADOWOLONY**

Przyznaję: nigdy nie widziałem bardziej skomplikowanego sposobu na wyświetlenie małej ikony niż ten z listingu 3. Pamiętaj jednak, że przekazywanie informacji z jednej strony na drugą jest ważne, niezależnie od tego, czy przekazujesz prostą wartość _isFavorite, czy dużą porcję danych medycznych. Podział aplikacji na strony zapewnia porządek na stronach. Zapewnia również ciągłość przepływu aplikacji. W części 5 dowiesz się, że Dart ma zmienne najwyższego poziomu — zmienne, które nie są zadeklarowane wewnątrz klasy. Jeśli umieścisz cały kod swojej aplikacji w jednym pliku, cały kod w Twojej aplikacji będzie mógł odnosić się bezpośrednio do tych zmiennych najwyższego poziomu. Dlaczego więc potrzebujesz funkcji argumentów w tej sekcji?

Dlaczego nie pozwolić, aby strony wzorcowe i strony szczegółowe współdzieliły wartości zmiennych najwyższego poziomu? Odpowiedź brzmi: zmienne najwyższego poziomu mogą być niebezpieczne. Podczas gdy Mary wypłaca środki na jednej stronie, inna strona przetwarza automatyczną płatność i prawie opróżnia konto Mary. W rezultacie Mary przepełnia swoje konto i jest winna bankowi wysoką opłatę. To nie jest dobrze. Oszczędnie używaj zmiennych najwyższego poziomu. Nie używaj zmiennych najwyższego poziomu do przekazywania informacji między stronami. Zamiast tego użyj funkcji argumentów Fluttera.

W tej sekcji popełniam poważny grzech. Chcę, aby przykłady z tego rozdziału były jak najprostsze, więc Listingi 1 i 3 nie zapewniają użytkownikowi możliwości zmiany wartości _isFavorite. Zamiast tego zapraszam czytelnika do przejrzenia Listingu 3, zmiany _isFavorite na false, a następnie ponownego uruchomienia kodu. To okropny sposób na usunięcie filmu z ulubionych — na przykład nakazanie klientom konta bankowego edytowania kodu źródłowego stron konta — ale spełnia swoje zadanie.

**ZMIENNA STATYCZNA**

Na listingu 3 deklaracja _isFavorite zaczyna się od słowa static. Każda zmienna zadeklarowana w rozszerzeniu, a nie w którejkolwiek z metod rozszerzenia, musi być statyczna. Jeśli zastosujesz się do tej zasady na ślepo, możesz zrozumieć Listing 3, nie wiedząc, co oznacza statyczność. Ale jeśli chcesz wiedzieć, co oznacza statyczność, rozważ ten mały fragment kodu z Części 7:

```
… => _MyHomePageState();

// … and later …

class _MyHomePageState extends

State<MyHomePage> {

bool _ageSwitchValue = false;
```

W pierwszym wierszu wywołanie konstruktora tworzy instancję programu Klasa _MyHomePageState. Nieco później kod daje _MyHomePageState zmienna instancji o nazwie _ageSwitchValue. Kod nie ma innego _MyHomePageState wywołania konstruktora, więc masz tylko jeden _MyHomePageState i tylko jedną zmienną _ageSwitchValue. W niektórych programach możesz mieć okazję zadzwonić

wywołać konstruktor _MyHomePageState dwukrotnie. Jeśli to zrobisz, będziesz miał dwie instancje _MyHomePageState, każda z własną zmienną _ageSwitchValue. Jeśli wykonasz zadanie np. _ageSwitchValue = true w jednym z przypadków, nie ma to wpływu na zmienną _ageSwitchValue w drugiej instancji. To jest sposób działania zmiennych instancji, ale… … to nie jest sposób, w jaki działają zmienne statyczne. Na listingu 8.3 zmienna _isFavorite jest statyczna. Jeśli zdarzy ci się zadeklarować dwa wystąpienia MovieTitlePageState, obie instancje współdzielą jedną zmienną _isUlubiona .Jeśli wykonasz zadanie np. _isFavorite = true w jednym z przypadków ustawia _isFavorite wartość dla obu instancji. W wywołaniu konstruktora RouteSettings argumenty nazwy parametru są nieco mylące. Ten parametr może mieć tylko jedną wartość naraz — wartość taką jak _isFavorite. Dlaczego więc nazwa parametru ma liczbę mnogą (argumenty) zamiast liczby pojedynczej (argument)? Jest to liczba mnoga, ponieważ pojedyncza rzecz, którą przekazujesz na inną stronę, może składać się z kilku części. Na przykład możesz przekazać wiele wartości, ustawiając wartość jednego i jedynego argumentu jako listę:

```
settings: RouteSettings(

arguments: [_isFavorite, _isInTheaters,

_isAComedy,],

),
```

**Przekazywanie danych z powrotem do źródła**

W poprzedniej sekcji kod używa Navigator.push do wysyłania wartości ze źródła do miejsca docelowego. To fajnie, ale w jaki sposób miejsce docelowe może wysyłać wartości z powrotem do źródła? Listing 4 zawiera odpowiedź.

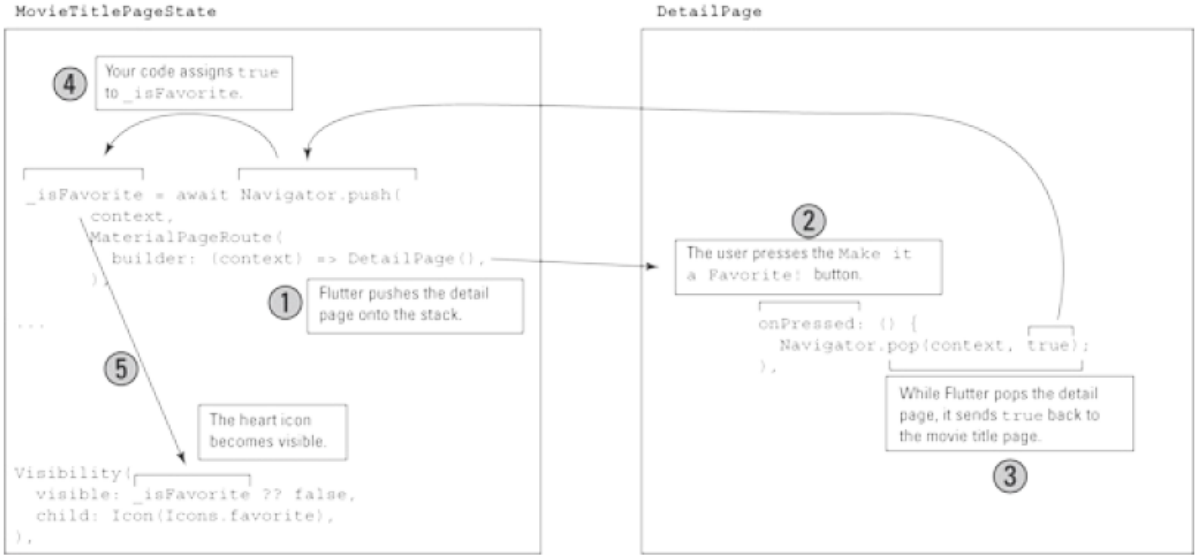
**LISTING 4 Od strony szczegółów do strony tytułowej filmu**

```
// App0804.dart

import 'package:flutter/material.dart';

import 'App08Main.dart';

extension MoreMovieTitlePage on

MovieTitlePageState {

static bool _isFavorite;

goToDetailPage() async {

_isFavorite = await Navigator.push(

context,

MaterialPageRoute(

builder: (context) => DetailPage(),

),

) ??

_isFavorite;
```

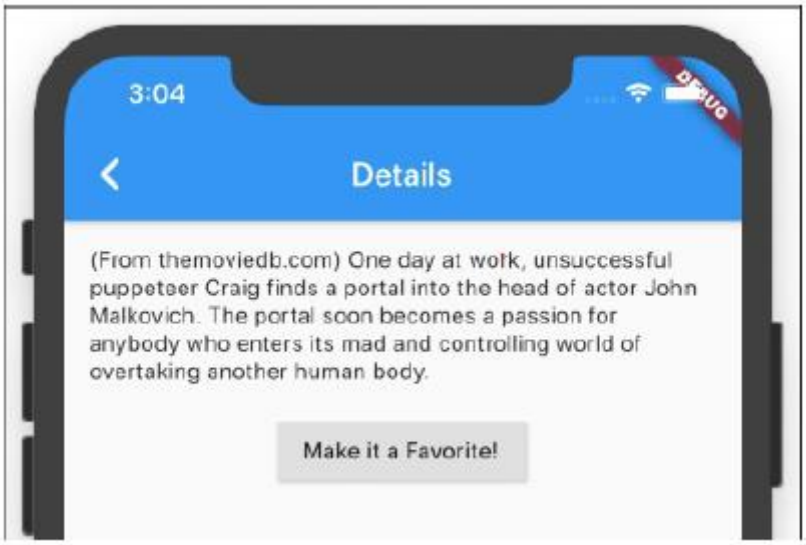
```
}
Widget buildTitlePageCore() {
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Row(
mainAxisAlignment:
MainAxisAlignment.center,
children: <Widget>[
Text(
'Being John Malkovich',
textScaleFactor: 1.5,
),
Visibility(
visible: _isFavorite ?? false,
child: Icon(Icons.favorite),
),
],
),
SizedBox(height: 16.0),
RaisedButton.icon(
icon: Icon(Icons.arrow_forward),
label: Text('Details'),
onPressed: goToDetailPage,
),
],
);
}
}
extension MoreDetailPage on DetailPage {
```

```
Widget buildDetailPageCore(context) {
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(
overview,
),
SizedBox(height: 16.0),
RaisedButton(
child: Text(
'Make it a Favorite!',
),
onPressed: () {
Navigator.pop(context, true);
},
),
],
);
}
}
```

Rysunek 5 ilustruje akcję, która ma miejsce podczas uruchamiania  przykład z tej sekcji.




Kod z listingu 4 tworzy stronę DetailPage, którą widzisz na rysunku.




Strona szczegółów ma dwa przyciski — jeden na pasku aplikacji (przycisk Wstecz) i jeden pod przeglądem filmu (przycisk Dodaj do ulubionych!). Jeśli użytkownik naciśnie przycisk Wstecz na pasku aplikacji, nic ciekawego się nie wydarzy. Aplikacja powraca do strony MovieTitlePage, takiej jak na rysunku 1. Ale jeśli użytkownik naciśnie przycisk Dodaj go do ulubionych! przycisk, Flutter wykonuje następującą instrukcję:

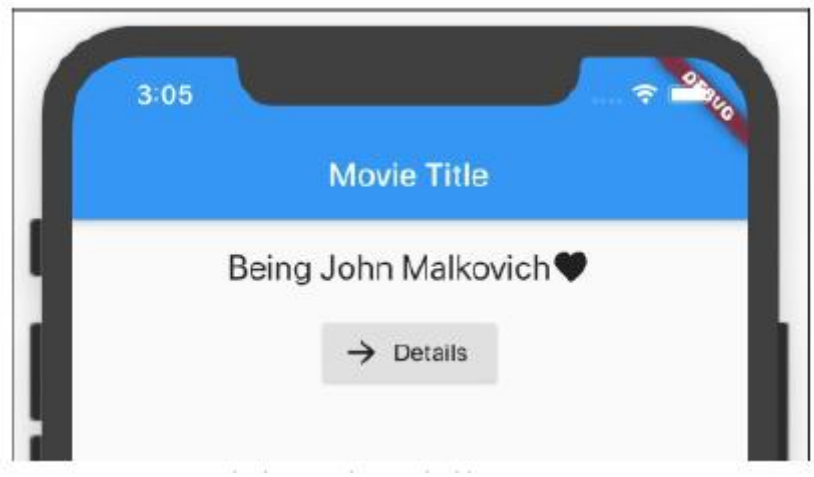
Navigator.pop(context, true);

Flutter zdejmuje DetailPage ze swojego stosu i wysyła wartość true z powrotem do MovieTitlePage. Na MovieTitlePage zadanie z tajemniczo wyglądającym słowem await ustawia _isFavorite na true:

_isFavorite = await Navigator.push(

// … Etc.

Wreszcie, gdy parametr _isFavorite ma wartość true, strona MovieTitlePage wyświetla małą ikonę serca, jak widać na rysunku 7.




**Słowa kluczowe Darta async i await**

Użytkownik uruchamia aplikację na listingu 4, przechodzi ze strony MovieTitlePage do strony szczegółów, a następnie zatrzymuje się, aby napić się kawy. Ten użytkownik nalega na posiadanie tylko najlepszej kawy. Ze smartfonem wyświetlającym stronę DetailPage, ten użytkownik leci samolotem do Wietnamu, kupuje świeżą filiżankę Kopi Luwak (pochodzącej z przewodu pokarmowego cywety palmowej), a następnie leci do domu. Wreszcie, trzy dni po uruchomieniu aplikacji z tej sekcji, użytkownik naciska przycisk Dodaj go do ulubionych! przycisk, który zwraca wartość True do strony MovieTitlePage aplikacji. Nigdy nie wiadomo, jak długo użytkownik pozostanie na stronie szczegółów aplikacji. Dlatego metoda Navigator.push Fluttera tak naprawdę nie zwraca wartości ze strony DetailPage. Zamiast tego wywołanie Navigator.push zwraca obiekt typu Future. Obiekt Future jest rodzajem wywołania zwrotnego. Jest to pole, które może zawierać wartość taką jak prawda lub nie. Podczas gdy nasz kochający kawę użytkownik odwiedza Wietnam, w pudełku Future nie ma nic. Ale później, gdy użytkownik wróci do domu i kliknie opcję Dodaj do ulubionych! przycisk, pole Przyszłość zawiera wartość true. W ten sposób Flutter radzi sobie z problemem nawigacji „nie wiadomo kiedy". Co by się stało z poniższym kodem?

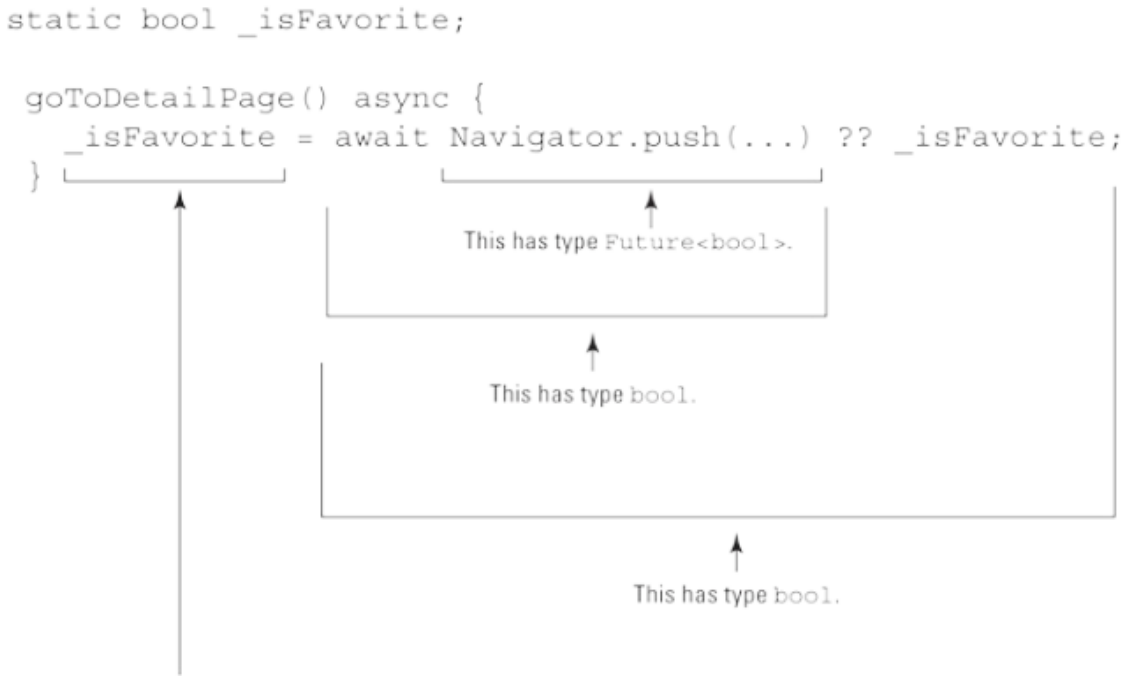
```
// Bad code because await is missing:

static bool _isFavorite;

// And elsewhere, …

_isFavorite = Navigator.push(

// … Etc.
```

W tym błędnym kodzie wywołanie Navigator.push próbuje przekazać obiekt Future do zmiennej _isFavorite. Ale zmienna _isFavorite nie będzie miała żadnego z nich, ponieważ typ zmiennej _isFavorite to bool, a nie Future. Co może zrobić programista? Listing 4 rozwiązuje ten problem za pomocą słowa kluczowego await programu Dart. Słowo kluczowe await robi dwie rzeczy:

*Słowo kluczowe await mówi Dartowi, aby nie kontynuował wykonywania bieżącej linii, dopóki pole Future nie będzie zawierało czegoś przydatnego. Na listingu 4 Dart nie przypisuje niczego do _isFavorite, dopóki strona szczegółów nie zostanie wyświetlona.

* Po otwarciu DetailPage słowo kluczowe await pobiera użyteczną wartość z pola Future.

Na listingu 4 wywołanie Navigator.push jest wartością Future, ale wyrażenie await Navigator.push( // … etc jest wartością bool. (Patrz rysunek 8.8). Twój kod przypisuje tę wartość bool do _isFavorite , która odpowiednio jest zmienną typu bool.




Wykonanie funkcji zawierającej słowo kluczowe await może zająć dużo czasu. Jeśli nie będziesz ostrożny, cała aplikacja może się zatrzymać z piskiem, podczas gdy oczekiwanie czeka. Tak więc, oprócz słowa kluczowego await, Dart ma słowo kluczowe async i regułę towarzyszącą temu słowu kluczowemu: jeśli deklaracja funkcji zawiera słowo kluczowe await, ta deklaracja musi również zawierać słowo kluczowe async. Słowo kluczowe async mówi Dartowi, że można wykonać jakiś inny kod, podczas gdy ta funkcja siedzi tam, nic nie robiąc, wykonując swoje słowo kluczowe await. W ten sposób aplikacja może kontynuować to, co robi, podczas gdy nasz przyjaciel, miłośnik kawy Kopi Luwak, odwiedza Wietnam.

**Przejmowanie kontroli nad przyciskiem Wstecz paska aplikacji**

Przycisk paska aplikacji na rysunku 6 to strzałka skierowana do tyłu. Gdy użytkownik kliknie ten przycisk, aplikacja powróci do strony źródłowej. Te dwa fakty są domyślnie prawdziwe. Ale co, jeśli nie lubisz ustawień domyślnych? Czy możesz je zmienić? Oczywiście, że możesz. Na przykład możesz zmienić wygląd przycisku ze strzałki wstecz na czerwony przycisk cofania. Aby to zrobić, dodaj parametr wiodący do wywołania konstruktora AppBar na listingu 1.

appBar: AppBar(

title: Text(

'Details',

),

leading: IconButton(

icon: new Icon(Icons.keyboard_backspace,

```
color: Colors.red),
onPressed: () => Navigator.pop(context),
),
)
```

Jeśli nie chcesz, aby przycisk Wstecz pojawiał się na pasku aplikacji, dodaj parametr automaticImplyLeading do wywołania konstruktora AppBar.

```
appBar: AppBar(
automaticallyImplyLeading: false,
```

Zmiana zachowania przycisku paska aplikacji jest trudniejsza. Na listingu 8.1 wywołanie konstruktora Scaffold otaczamy wywołaniem WillPopScope:

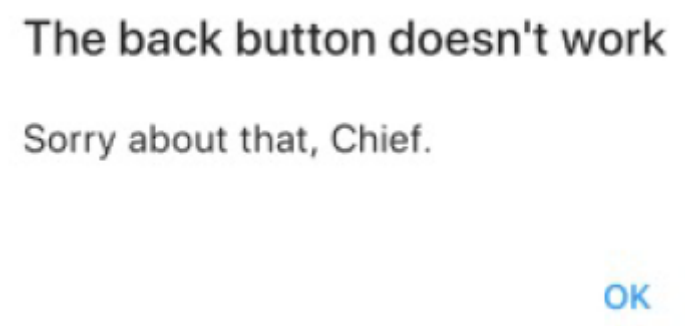
```
@override
Widget build(BuildContext context) {
return WillPopScope(
onWillPop: () => _onPop(context),
child: Scaffold(
```

W wywołaniu konstruktora WillPopScope parametr onWillPop jest funkcją i zgodnie ze słowem Will w onWillPop funkcja ta zwraca wartość Future. Oto mały przykład:

```
Future<bool> _onPop(BuildContext context) async
{
return await showDialog(
context: context,
child: AlertDialog(
title: Text("The back button doesn't
work"),
content: Text('Sorry about that,
Chief.'),
actions: <Widget>[
new FlatButton(
onPressed: () =>
Navigator.pop(context, false),
child: Text('OK'),
),
```

```
],
),
) ??
false;
}
```

Gdy użytkownik kliknie przycisk Wstecz na pasku aplikacji DetailPage, Flutter wyświetli okno dialogowe zawierające przycisk FlatButton oznaczony jako OK. Gdy użytkownik kliknie przycisk FlatButton, Flutter zamknie okno dialogowe.




**Przekazywanie danych w obu kierunkach**

Przykład w tej sekcji jest nieco bardziej realistyczny niż przykłady w poprzednich sekcjach. W tej sekcji strony źródłowa i docelowa przekazują informacje tam i z powrotem. Kod znajduje się na listingu 5.

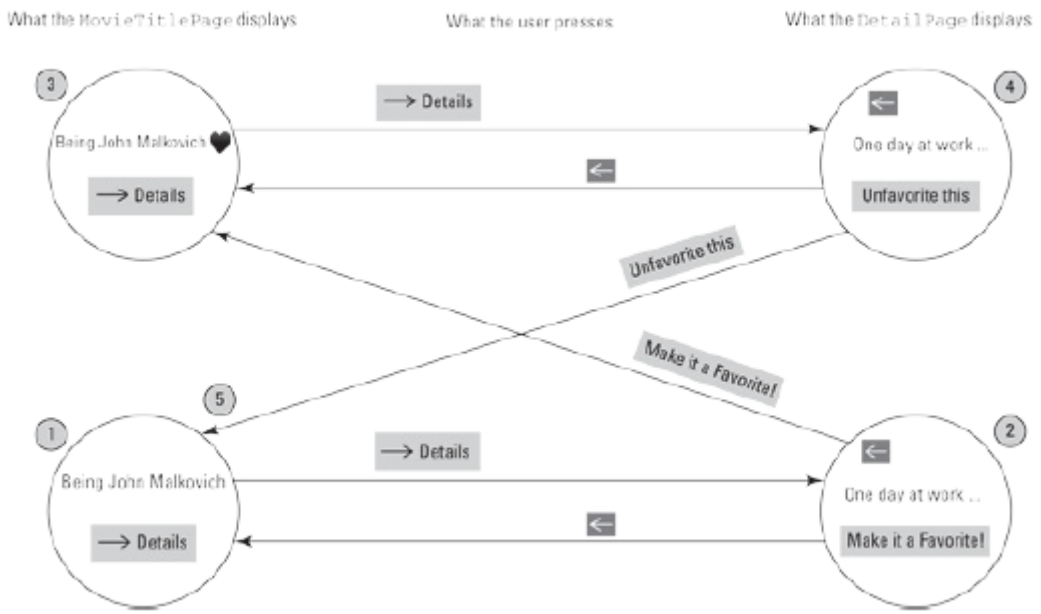
**LISTING 5 Od strony tytułowej do strony szczegółowej i z powrotem**

```
// App0805.dart
import 'package:flutter/material.dart';
import 'App08Main.dart';
extension MoreMovieTitlePage on
MovieTitlePageState {
static bool _isFavorite;
goToDetailPage() async {
_isFavorite = await Navigator.push(
context,
MaterialPageRoute(
builder: (context) => DetailPage(),
settings: RouteSettings(
arguments: _isFavorite,
```

```
),
),
) ??
_isFavorite;
}
Widget buildTitlePageCore() {
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Row(
mainAxisAlignment:
MainAxisAlignment.center,
children: <Widget>[
Text(
'Being John Malkovich',
textScaleFactor: 1.5,
),
Visibility(
visible: _isFavorite ?? false,
child: Icon(Icons.favorite),
),
],
),
SizedBox(height: 16.0),
RaisedButton.icon(
icon: Icon(Icons.arrow_forward),
label: Text('Details'),
onPressed: goToDetailPage,
),
],
```

```
);
}
}
extension MoreDetailPage on DetailPage {
Widget buildDetailPageCore(context) {
final _isFavoriteArgument =
ModalRoute.of(context).settings.arguments ??
false;
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(
overview,
),
SizedBox(height: 16.0),
RaisedButton(
child: Text(
_isFavoriteArgument ? 'Unfavorite
this' : 'Make it a Favorite!',
),
onPressed: () {
Navigator.pop(context,
!_isFavoriteArgument);
},
),
],
);
}
}
```

W aplikacji tej sekcji strony MainTitlePage i DetailPage wspólnie odpowiadają za status „ulubionego” filmu. Kiedy pojawia się ikona Ulubione, pojawia się ona na MainTitlePage, ale strona DetailPage ma przycisk, który przełącza między „ulubionym” a „nieulubionym”. Rysunek 10 przedstawia działanie aplikacji z tej sekcji. W następnych kilku akapitach przeprowadzę cię przez ponumerowane wypunktowania na tym rysunku.




1. Po uruchomieniu aplikacji z tej sekcji wartość _isFavorite staje się fałszywa. Widzisz stronę z tytułem filmu i przyciskiem Szczegóły, ale bez ikony serca. Aby wyświetlić tę stronę w telefonie, zobacz Rysunek 8-1. Po naciśnięciu przycisku Szczegóły metoda goToDetailPage wysyła wartość _isFavorite do strony DetailPage:

MaterialPageRoute(

builder: (context) => DetailPage(),

settings: RouteSettings(

arguments: _isFavorite,

),

),

2. DetailPage otrzymuje wartość pochodzącą z MovieTitlePage. DetailPage przechowuje tę wartość we własnej zmiennej _isFavoriteArgument:

_isFavoriteArgument variable:

final _isFavoriteArgument =

ModalRoute.of(context).settings.arguments

?? false;

Używając wartości tej zmiennej, DetailPage decyduje, co ma być wyświetlane na froncie przycisku:

RaisedButton(

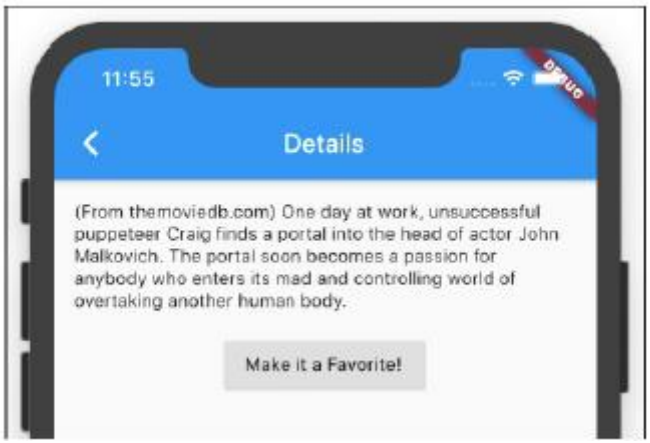
child: Text(

_isFavoriteArgument ? 'Unfavorite this' :

'Make it a Favorite!',

),

W tym momencie działania aplikacji _isFavoriteArgument ma wartość false. Tak więc wypukły przycisk wyświetla zdanie Dodaj go do ulubionych! Rysunek 11 przedstawia stronę DetailPage wyświetlaną w telefonie.




Jeśli naciśniesz przycisk Dodaj go do ulubionych! przycisk, operator wykrzyknika Darta (!) przygotowuje przeciwieństwo _isFavoriteArgument do wysłania z powrotem do MovieTitlePage:

onPressed: () {

Navigator.pop(context,

!_isFavoriteArgument);

},

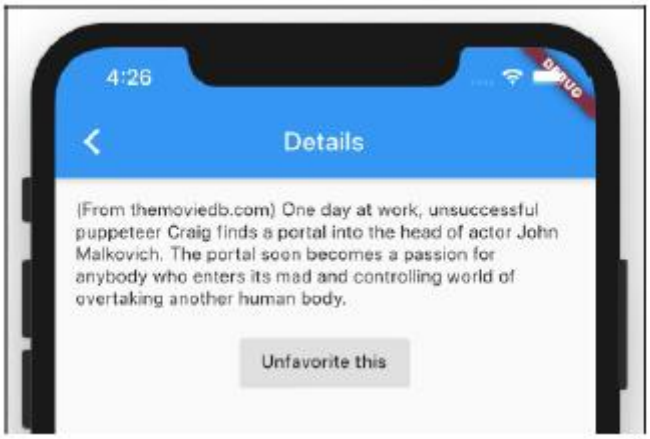
Ponieważ _isFavoriteArgument ma wartość false, DetailPage wysyła swoje przeciwieństwo (true) z powrotem do MovieTitlePage.

3. Po otrzymaniu wartości true strona MovieTitlePage wyświetla ikonę serca. Rysunek 12 przedstawia stronę MovieTitlePage wyświetlaną w telefonie.


Po naciśnięciu przycisku Szczegóły MovieTitlePage wysyła true do DetailPage.

4. Tym razem linia _isFavoriteArgument ? „Usuń to z ulubionych”:

„Uczyń to ulubionym!” mówi DetailPage, aby wyświetlił przycisk Usuń to z ulubionych. Rysunek 13 przedstawia stronę DetailPage wyświetlaną w telefonie.




Jeśli naciśniesz przycisk Unfavorite This, operator wykrzyknika Darta przygotuje przeciwieństwo _isFavoriteArgument do przesłania z powrotem do MovieTitlePage. Ponieważ _isFavoriteArgument ma wartość true, DetailPage wysyła swoje przeciwieństwo (false) z powrotem do MovieTitlePage.

5. Po otrzymaniu wartości false strona MovieTitlePage nie wyświetla ikony serca.

Rysunek na rysunku 10 jest tak zwanym diagramem maszyny skończonej. Diagramy tego rodzaju bardzo pomagają, gdy chcesz uporządkować swoje przemyślenia na temat przejść między stronami aplikacji.

**Tworzenie nazwanych tras**

Nawigacja może być skomplikowana. Oto przykład: „Idź tam, gdzie użytkownik chce się udać, chyba że użytkownik nie jest zalogowany, w takim przypadku przejdź do strony logowania (ale pamiętaj, gdzie użytkownik chciał się udać). Jeśli użytkownik zaloguje się poprawnie, przejdź tam, gdzie chciał. W przeciwnym razie przejdź do strony „nieprawidłowy login”, gdzie użytkownik ma możliwość przejścia do strony „zapomniałem hasła”. Ze strony „zapomnij hasło”…” I tak dalej. We Flutter ekrany i strony nazywane są trasami, a Flutter umożliwia przypisanie nazwy do każdej z tras. Ta funkcja nazwanych tras sprawia, że kod jest nieco bardziej zwięzły. Co ważniejsze, ta funkcja chroni Cię przed szaleństwem, śledząc ścieżki użytkownika i objazdy. Kod z listingu 6 nie wyświetla żadnych danych filmu — tylko paski i przyciski aplikacji. Mimo to lista pokazuje, jak działają nazwane trasy.

**LISTING 6 Zagrajmy wszyscy w „Nazwij tę trasę”**

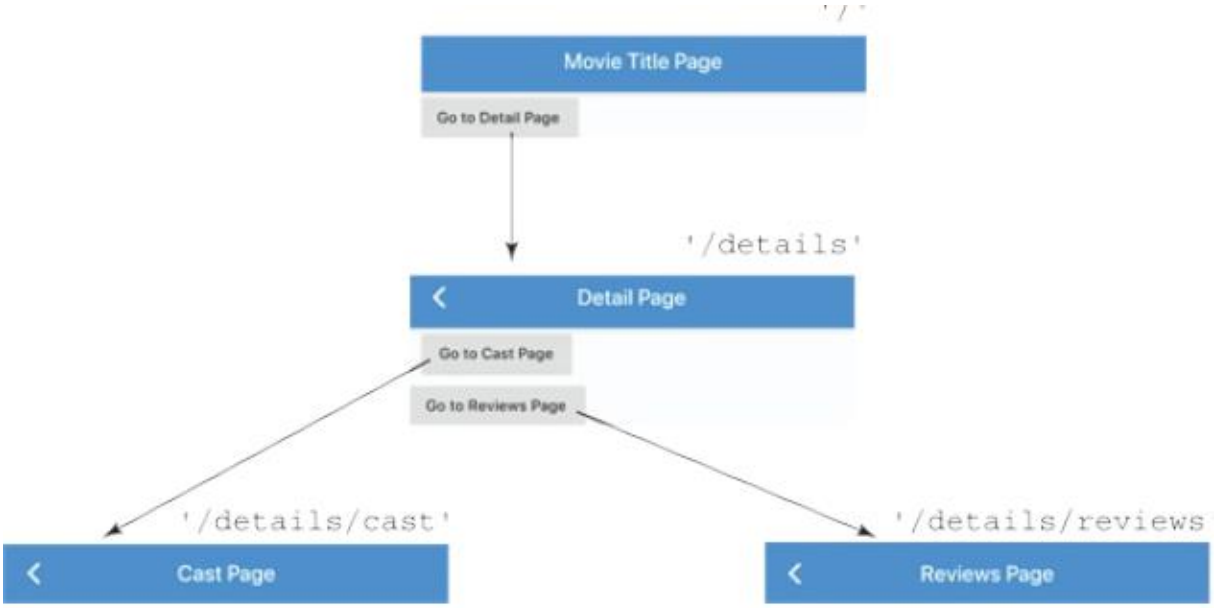
```
// App0806.dart

import 'package:flutter/material.dart';

void main() => runApp(App0806());

class App0806 extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(
```

```
routes: {
’/’: (context) => MovieTitlePage(),
’/details’: (context) => DetailPage(),
’/details/cast’: (context) =>
CastPage(),
’/details/reviews’: (context) =>
ReviewsPage(),
},
);
}
}
class MovieTitlePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return _buildEasyScaffold(
appBarTitle: 'Movie Title Page',
body: _buildEasyButton(
context,
label: 'Go to Detail Page',
whichRoute: ’/details’,
),
);
}
}
class DetailPage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return _buildEasyScaffold(
appBarTitle: 'Detail Page',
body: Column(
children: <Widget>[
```

```
_buildEasyButton(
context,
label: 'Go to Cast Page',
whichRoute: ’/details/cast’,
),
_buildEasyButton(
context,
label: 'Go to Reviews Page',
whichRoute: ’/details/reviews’,
),
],
),
);
}
}
class CastPage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return _buildEasyScaffold(
appBarTitle: 'Cast Page',
body: Container(),
);
}
}
class ReviewsPage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return _buildEasyScaffold(
appBarTitle: 'Reviews Page',
body: Container(),
);
```

```
}
}
Widget _buildEasyScaffold({String appBarTitle,
Widget body}) {
return Scaffold(
appBar: AppBar(
title: Text(appBarTitle),
),
body: body,
);
}
Widget _buildEasyButton(
BuildContext context, {
String label,
String whichRoute,
}) {
return RaisedButton(
child: Text(label),
onPressed: () {
Navigator.pushNamed(
context,
whichRoute,
);
},
);
}
```

Kod z listingu 6 nie zależy od żadnego innego kodu z listingu. Po prostu umieść kod tej sekcji w pliku .dart, a następnie uruchom go. Rysunek 14 przedstawia górne strony stron aplikacji z listingu 6.




Inne listingi w tym rozdziale rozrzucają informacje o routingu w całym kodzie, chcąc nie chcąc. Ale Listing 6 podsumowuje informacje o routingu w parametrze tras konstruktora MaterialApp. Zwróć uwagę na hierarchiczne nazewnictwo tras na rysunku 14. Im bardziej podrzędna trasa, tym więcej ukośników (/) w nazwie trasy. Jako dodatkowy bonus, metoda pushNamed klasy Navigator jest nieco prostsza niż zwykła, stara metoda push tej klasy. Prostszy kod oznacza mniej udręki dla ciebie, programisty i większą szansę, że kod jest poprawny. Na listingu 6 wywołanie konstruktora MaterialApp nie ma parametru home. To jest w porządku, ponieważ parametr tras konstruktora zajmuje luz. Domyślnie trasa o nazwie „/” jest punktem początkowym Twojej aplikacji. Jeśli zdecydujesz się nie mieć trasy o nazwie „/” lub jeśli chcesz zastąpić domyślną, możesz dodać parametr initialRoute. Na przykład możesz dodać jedną linię do kodu z listingu 6, tak jak poniżej:

```
Widget build(BuildContext context) {

return MaterialApp(

routes: {

'/': (context) => MovieTitlePage(),

'/details': (context) => DetailPage(),

'/details/cast': (context) => CastPage(),

'/details/reviews': (context) =>

ReviewsPage(),

},

initialRoute: ’/details/cast’,

);

}
```

Kiedy aplikacja z tym zmodyfikowanym kodem zaczyna działać, użytkownik widzi CastPage aplikacji, a to, co dzieje się dalej, może Cię zaskoczyć lub nie. Gdy użytkownik naciśnie przycisk Wstecz na pasku

aplikacji, Flutter przechodzi do strony DetailPage. Dzieje się tak, ponieważ Flutter patrzy na ukośniki w nazwach tras. Gdy wycofasz się z trasy o nazwie „/details/cast”, „/details/reviews” lub „/details/cokolwiek”, Flutter przeniesie Cię do trasy o nazwie „/details”.

**Tworzenie listy**

Wyobraź to sobie. Siedzisz z przyjaciółmi w lokalnej jadłodajni. Ktoś mówi: „Pamiętasz ten film Być jak John Malkovich? Zastanawiam się, o co chodziło. Więc wyciągasz telefon i mówisz: „Co za zbieg okoliczności! Mam aplikację, której jedynym celem jest pokazanie mi przeglądu fabuły tego filmu”. Po przeczytaniu przeglądu znajomym jeden z nich mówi: „To świetnie! A co z kolejnym filmem Charliego Kaufmana? Myślę, że to się nazywa Adaptacja”. A ty mówisz: „Nie mamy szczęścia. Moja aplikacja zawiera informacje tylko o jednym filmie”. W tym momencie jeden z Twoich znajomych mówi: „Ile zapłaciłeś za tę aplikację?” A ty odpowiadasz: „Nie zapłaciłem za to. Barry Burd zapłacił mi za zainstalowanie tego. Na początku tego rozdziału opisano interfejs główny-szczegółowy. Mówi: „Pierwsza strona [w interfejsie główny-szczegółowy] wyświetla listę elementów”. Nie możesz zmieścić informacji o każdym elemencie z listy na jednej stronie. Aby uzyskać szczegółowe informacje na temat konkretnego elementu, użytkownik klika ten element i przechodzi do osobnej strony. W tej sekcji pokazano, jak poruszać się między listą elementów a stroną szczegółów. Nowy przykład jest znacznie bardziej użyteczny niż przykłady z rozdziału Być jak John Malkovich. Aplikacja na listingu 7 zawiera listę wszystkich 25 filmów z serii Rocky Sylvestra Stallone.

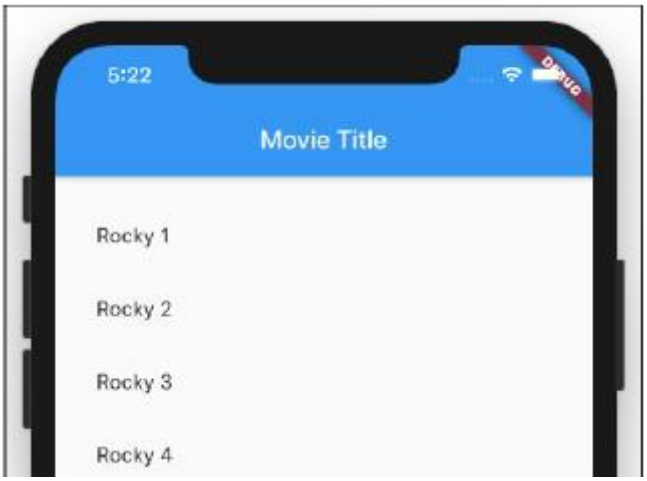
**LISTING 7 Dość długa lista**

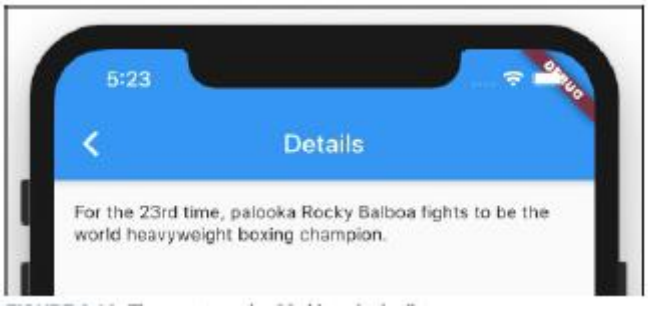
```
// App0807.dart

import 'package:flutter/material.dart';

import 'App08Main.dart';

extension MoreMovieTitlePage on

MovieTitlePageState {

goToDetailPage(int index) {

Navigator.push(

context,

MaterialPageRoute(

builder: (context) => DetailPage(),

settings: RouteSettings(

arguments: index,

),

),

);

}

Widget buildTitlePageCore() {
```

```
return ListView.builder(
itemCount: 25,
itemBuilder: (context, index) =>
ListTile(
title: Text('Rocky ${index + 1}'),
onTap: () => goToDetailPage(index + 1),
),
);
}
}
extension MoreDetailPage on DetailPage {
Widget buildDetailPageCore(context) {
final sequelNumber =
ModalRoute.of(context).settings.arguments;
final overview =
'For the
$sequelNumber${getSuffix(sequelNumber)} time,
palooka '
'Rocky Balboa fights to be the world
heavyweight boxing champion.';
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(overview),
],
);
}
String getSuffix(int sequelNumber) {
String suffix;
switch (sequelNumber) {
```

```
case 1:
case 21:
suffix = 'st';
break;
case 2:
case 22:
suffix = 'nd';
break;
case 3:
case 23:
suffix = 'rd';
break;
default:
suffix = 'th';
}
return suffix;
}
}
```

Aby uruchomić aplikację z Listingu 7, Twój projekt musi mieć co najmniej dwa pliki .dart - jeden zawierający kod z Listingu 7 i drugi z kodem z Listingu 1. Gdy uruchomisz kod z listingu 7, otrzymasz dwie strony — stronę główną z listą tytułów filmów i, jak zwykle, stronę szczegółów. Na rysunku 15 pokazano stronę z listą tytułów filmów, a na rysunku 16 stronę ze szczegółami.






**Widżet ListView**

Istotą Listingu 7 jest wywołanie konstruktora ListView.builder Fluttera. Konstruktor przyjmuje dwa parametry: itemCount i itemBuilder.

**Parametr itemCount**

Nikogo nie dziwi, że itemCount mówi Flutterowi, ile elementów ma wyświetlić na liście. W kodzie z listingu 8.7 ostatnim elementem listy jest Rocky 25. Jeśli jednak pominiemy parametr itemCount, lista nigdy się nie skończy. Użytkownik może przewijać godzinami, aby zobaczyć elementy listy o nazwach Rocky 1000 i Rocky 10000. (Przynajmniej tak myślę. Szczerze mówiąc, nigdy nie próbowałem przewijać dalej niż Rocky 1200. Kiedy próbuję, moja ręka się męczy .) Sekretem ListView z jego itemCount jest możliwość przewijania. Teoretycznie lista zawiera więcej pozycji niż użytkownik widzi na ekranie urządzenia. W rzeczywistości Flutter żongluje elementami listy i przechowuje tylko tyle, aby wypełnić ekran użytkownika. Kiedy przedmiot znika poza krawędzią ekranu, Flutter przetwarza ten przedmiot, nadając mu nowy numer Rocky'ego i wyświetlając go na drugim końcu ekranu. Recykling elementów listy, Flutter oszczędza miejsce w pamięci i czas przetwarzania. Przewijanie listy przebiega więc płynnie.

**Parametr itemBuilder**

Wartość parametru itemBuilder jest funkcją. Na listingu 7, aby utworzyć 25 elementów, Flutter zaczyna od utworzenia 25 indeksów o wartościach 0, 1, 2 itd., aż do 24 włącznie. Flutter wstawia te wartości do funkcji itemBuilder w następujący sposób:

```
// This isn’t real code. It’s the way

itemBuilder behaves.

(context, 0) => ListTile(

title: Text('Rocky ${0 + 1}'),

onTap: () => goToDetailPage(0 + 1),

)

(context, 1) => ListTile(

title: Text('Rocky ${1 + 1}'),

onTap: () => goToDetailPage(1 + 1),

)

(context, 2) => ListTile(

title: Text('Rocky ${2 + 1}'),
```

```
onTap: () => goToDetailPage(2 + 1),

)

// … and so on.
```

Wynikiem jest lista zawierająca 25 pozycji. Liczba Rocky'ego na widżecie Tekst każdego elementu jest o jeden większa niż wartość indeksu. W ten sposób lista nie zaczyna się od filmu o nazwie Rocky 0. (Rocky: The Prequel?)

W Dart wszystko, co się liczy, zaczyna się automatycznie od 0, a nie od 1. Obejmuje to takie rzeczy, jak indeks elementu itemBuilder, pozycja znaku w łańcuchu i domyślna minimalna wartość elementu Slider.

Oprócz widżetu tekstowego każdy element ma funkcję onTap. Każda funkcja onTap wysyła własną wartość (liczbę od 1 do 25) do funkcji goToDetailPage. Jeśli będziesz podążać dalej śladem, zauważysz, że funkcja goToDetailPage wysyła wartość liczbową dalej jako argument do strony DetailPage aplikacji. Z kolei DetailPage używa tej wartości do decydowania, jakie informacje mają być wyświetlane. W rzeczywistej aplikacji strona szczegółów może użyć tej wartości do wyszukania przeglądu filmu — być może jednego z kilku tysięcy filmów. Ale na listingu 7 strona DetailPage po prostu tworzy fałszywy przegląd.

Nawiasem mówiąc, możesz zauważyć, że Listingi 1 i 7 mają zmienne przeglądowe i obie te zmienne znajdują się w tej samej klasie DetailPage. (Przegląd na listingu 1 znajduje się w oryginalnej deklaracji DetailPage. Przegląd na listingu 7 jest rozszerzeniem klasy DetailPage.) To podwójne użycie nazwy zmiennej jest w porządku. Przegląd na listingu 1 jest zmienną instancji, a przegląd na listingu 7 jest lokalny dla metody buildDetailPageCore. Tak więc po uruchomieniu kodu z listingu 7 przegląd nazw oznacza zdanie o Rocky'm Balboa. Wszystko jest dobrze.

**JAK UMIEŚCIĆ ListView WEWNĄTRZ KOLUMNY**

Niektóre układy — takie, które możesz uważać za w porządku — wysyłają Fluttera w niekończącą się grę polegającą na pogoni za ogonem. Część 6 zawiera sekcję na ten temat. Gra jest szczególnie frustrująca, gdy próbujesz umieścić widok listy w kolumnie. Oto zły kod:

```
// Don’t do this:

Widget buildTitlePageCore() {

return Column(

children: <Widget>[

Text('Rocky Movies'),

ListView.builder(

itemCount: 25,

itemBuilder: (context, index) =>

ListTile(

title: Text('Rocky ${index +

1}'),
```

```
onTap: () =>

goToDetailPage(index + 1),

),

),

],

);

}
```

Po uruchomieniu tego kodu nie pojawia się żaden widok listy. Wśród dziesiątek linii diagnostycznych raportuje o tym okno narzędzia Uruchom w Android Studio. Widokowi pionowemu nadano nieograniczoną wysokość. Tak jak w części 6, jeden widżet (widżet Kolumna) wysyła do swoich dzieci ograniczenie nieograniczonej wysokości, a jedno z nich (widżet ListView) nie może obsłużyć całej tej swobody. Rezultatem jest impas, w którym nie można wyświetlić ListView. Aby rozwiązać problem, zrób to samo, co w części 6 — dodaj widżet Expanded:

```
// Do this instead

return Column(

children: <Widget>[

Text('Rocky Movies'),

Expanded(

child: ListView.builder(

// … etc.
```

Widżet Expanded mówi: „Hej, kolumna. Dowiedz się, jak wysoki jest widżet Tekst i powiedz ListView, ile pozostało miejsca w pionie”. Kiedy kolumna przekazuje te informacje do ListView, ListView mówi: „Dzięki. Wykorzystam całą pozostałą przestrzeń”. Aplikacja wyświetla się poprawnie i wszyscy są zadowoleni.

**Instrukcja switch Darta**

W moim pierwszym szkicu Listingu 7 przegląd Rocky 3 brzmi: Po raz trzeci palooka Rocky Balboa walczy o tytuł mistrza świata wagi ciężkiej w boksie. Nie mogłem z tym żyć, więc poprosiłem o pomoc mojego przyjaciela — oświadczenie o zmianie. Instrukcja switch jest podobna do instrukcji if, z tą różnicą, że instrukcje switch nadają się do rozgałęzień wielokierunkowych. Instrukcja switch na listingu 8.7 mówi:

```
Look at the value of sequelNumber.

If that value is 1 or 21,

assign 'st' to suffix,

and then break out of the entire switch

statement.

If you’ve reached this point and that value
```

is 2 or 22,

assign 'nd' to suffix,

and then break out of the entire switch

statement.

If you’ve reached this point and that value

is 3 or 23,

assign 'rd' to suffix,

and then break out of the entire switch

statement.

If you’ve reached this point,

assign 'th' to suffix.

Każda instrukcja break wysyła cię z instrukcji switch i dalej do dowolnego kodu, który następuje po instrukcji switch. Co się stanie, jeśli spróbujesz pominąć instrukcje break?

// Dart doesn’t tolerate this …

switch (sequelNumber) {

case 1:

case 21:

suffix = 'st';

case 2:

// … and so on.

W Dart jest to nie-nie. Jeśli wpiszesz ten kod w edytorze Dart w Android Studio, Android Studio natychmiast się zgłosi. Android Studio odmawia uruchomienia twojego programu. Jeśli nie lubisz instrukcji break, możesz przepisać funkcję getSuffix, używając instrukcji return:

String getSuffix(int sequelNumber) {

switch (sequelNumber) {

case 1:

case 21:

return 'st';

case 2:

case 22:

return 'nd';

case 3:

```
case 23:

return 'rd';

}

return 'th';

}
```

Ta nowa wersja getSuffix jest znacznie bardziej zwięzła niż ta z listingu 7. W tej wersji getSuffix każda instrukcja return całkowicie wyskakuje z funkcji getSuffix. Nie potrzebujesz nawet klauzuli default, ponieważ docierasz do instrukcji return „th”, gdy żadna z klauzul case nie ma zastosowania. Nawet ta nowa i ulepszona funkcja getSuffix zawodzi, jeśli Sylvester Stallone zrobi Rocky 31. Przegląd filmu będzie brzmiał: „Po raz 31, palooka Rocky Balboa…” To nie brzmi dobrze. Istnieją dziesiątki sposobów na tworzenie bardziej wszechstronnych wersji getSuffix i fajnie jest spróbować stworzyć własną. Jeden z moich osobistych faworytów wygląda tak:

```
String getSuffix(int sequelNumber) {

int onesDigit = sequelNumber % 10;

int tensDigit = sequelNumber ~/ 10 % 10;

Map<int, String> suffixes = {1: 'st', 2:

'nd', 3: 'rd'};

String suffix = suffixes[onesDigit] ?? 'th';

if (tensDigit == 1) suffix = 'th';

return suffix;

}
```

**KILKA WIADOMOŚCI O PRZEWIJANIU**

Nie potrzebujesz ListView, aby utworzyć przewijany ekran. Możesz umieścić wszystkie rodzaje rzeczy w SingleChildScrollView. Oto kod:

```
return MaterialApp(

home: Material(

child: Column(

children: <Widget>[

SizedBox(height: 200, child:

Text("You've")),

SizedBox(height: 200, child:

Text("read")),

SizedBox(height: 200, child:

Text("many")),
```

```
SizedBox(height: 200, child:

Text("chapters")),

SizedBox(height: 200, child:

Text("of")),

Icon(Icons.book),

SizedBox(height: 100, child:

Text("Flutter For Dummies")),

Icon(Icons.thumb_up),

],

),

),

);
```

Mój telefon nie ma wystarczająco dużo miejsca na te wszystkie rzeczy. Więc jeśli nie dodam jakiegoś przewijania, widzę przerażające czarno-żółte paski wzdłuż dolnej części ekranu. Aby uniknąć wyświetlania tych pasków, umieszczam widżety w SingleChildScrollView:

```
return MaterialApp(

home: Material(

child: SingleChildScrollView(

child: Column(

children: <Widget>[

SizedBox(height: 200, child:

Text("You've")),

// . . . etc.

],

),

),

),

);
```

Kiedy ponownie uruchamiam kod, widzę kilka najwyżej położonych widżetów z opcją przewijania i przeglądania innych.

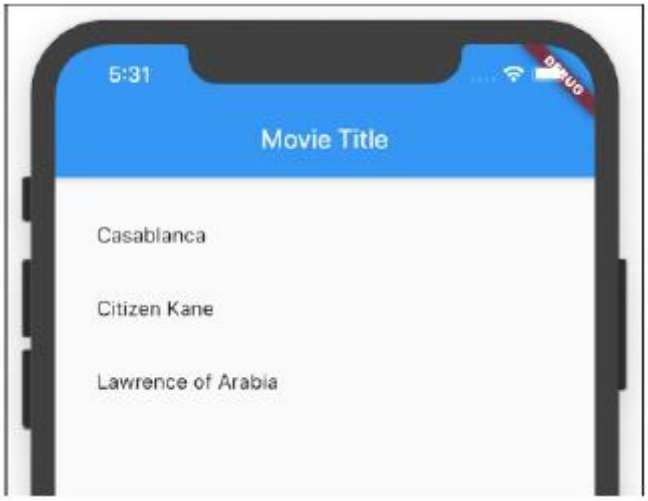
**Tworzenie elementów listy jeden po drugim**

Od jednego rzędu do drugiego pozycje na rysunku 15 nie mają żadnych niespodzianek. Każdy element wyświetla nazwę Rocky i numer. Każdy element zachowuje się tak samo po dotknięciu. Dzięki tej jednolitości mogę utworzyć jeden element budujący elementy, który opisuje wszystkie 25 elementów z listy. Co zrobić, jeśli jest mało jednolitości lub nie ma jej wcale? Co jeśli elementy są w pewnym stopniu jednorodne, ale jest ich tak mało, że tworzenie itemBuilder nie jest warte wysiłku? W takich przypadkach opisujesz elementy jeden po drugim, używając konstruktora ListView Fluttera. Listing 8 zawiera kod; Rysunki 17 i 18 przedstawiają niektóre wyniki.

```
// App0808.dart

import 'package:flutter/material.dart';

import 'App08Main.dart';

const Map<String, String> synopses = {

'Casablanca':

'In Casablanca, Morocco in December 1941,

a cynical American expatriate '

'meets a former lover, with

unforeseen complications.',

'Citizen Kane':

’… Charles Foster Kane is taken from his

mother as a boy … ’

'As a result, every well-meaning,

tyrannical or '

'self-destructive move he makes for

the rest of his life appears '

'in some way to be a reaction to that

deeply wounding event.',

'Lawrence of Arabia':

"The story of British officer T.E.

Lawrence's mission to aid the Arab "

"tribes in their revolt against the

Ottoman Empire during the "

"First World War.",

};

extension MoreMovieTitlePage on
```

```
MovieTitlePageState {
goToDetailPage(String movieName) {
Navigator.push(
context,
MaterialPageRoute(
builder: (context) => DetailPage(),
settings: RouteSettings(
arguments: movieName,
),
),
);
}
Widget buildTitlePageCore() {
return ListView(
children: [
ListTile(
title: Text('Casablanca'),
onTap: () =>
goToDetailPage('Casablanca'),
),
ListTile(
title: Text('Citizen Kane'),
onTap: () => goToDetailPage('Citizen
Kane'),
),
ListTile(
title: Text('Lawrence of Arabia'),
onTap: () => goToDetailPage('Lawrence
of Arabia'),
),
],
```

```
);
}
}
extension MoreDetailPage on DetailPage {
Widget buildDetailPageCore(context) {
final movieName =
ModalRoute.of(context).settings.arguments;
final overview = '(From themoviedb.com)
${synopses[movieName]}';
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
Text(overview),
],
);
}
}
```




Nienazwany konstruktor ListView Fluttera ma parametr child, a wartość tego parametru children to… poczekaj… lista języka Dart. Lista języków Dart to zbiór obiektów ujętych w parę nawiasów kwadratowych, tak jak poniżej:

// A Dart language List:

[

ListTile(…),

ListTile(…),

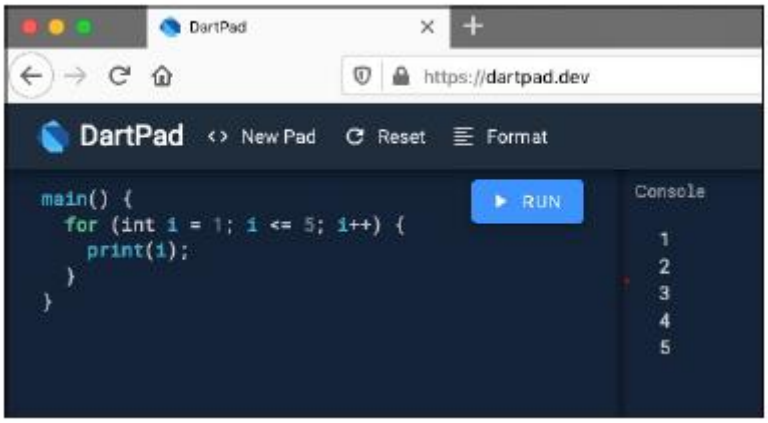
ListTile(…),

]

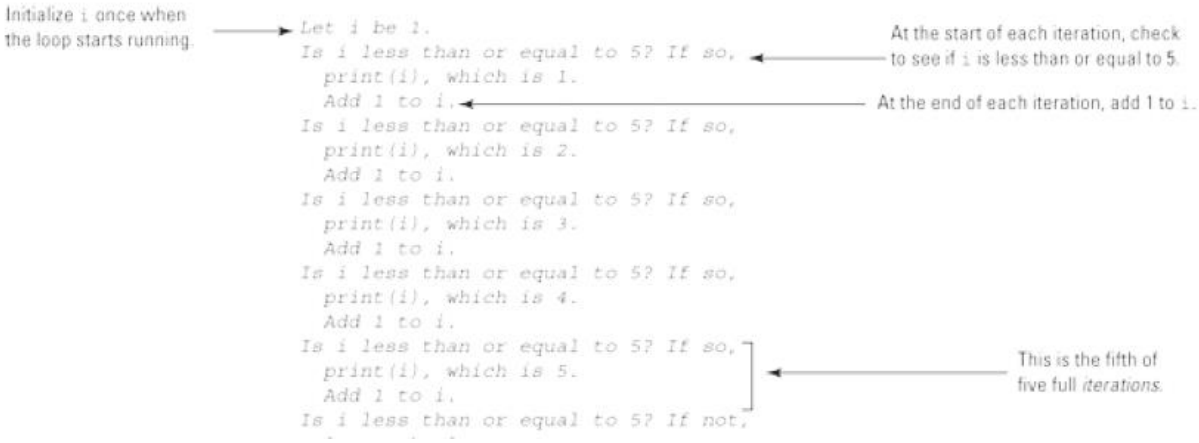
Na listingu 8 lista rzutek jest w rzeczywistości zbiorem widżetów ListTile. (Są jak płytki łazienkowe bez fugi między nimi). Ale ListView Fluttera jest wszechstronny. Dzieci nie muszą być widżetami ListTile. Elementy podrzędne ListView mogą być mieszanką widżetów tekstowych, widżetów SizedBox, widżetów Image.asset i wszelkich innych rodzajów widżetów. Może to być duża torba na zakupy. (Jeśli prowadzisz punktację, Listing 8.8 zawiera ListView, który zawiera listę widżetów ListTile w języku Dart).

**Kolejna nowa funkcja języka Dart**

Pewnego wieczoru, po 14 godzinach pracy nad tym rozdziałem, byłem wyczerpany i wpadłem w delirium. Pomyślałem o aplikacji z Listingu 7 z 25 elementami ListView. Wyobraziłem sobie małą osobę w telefonie użytkownika, która tworzy element Rocky 1, a następnie element Rocky 2, a następnie Rocky 3 i tak dalej. Może uda mi się wykorzystać tę głupią wizję i utworzyć widok listy z instrukcjami takimi jak „Zbuduj pierwszy element, zbuduj drugi element, zbuduj trzeci element…" i więcej. Większość języków programowania ma instrukcje, które wykonują powtarzalne zadania. Na przykład języki takie jak Java, C/C++ i Dart mają rzecz zwaną instrukcją for, znaną również jako pętla for. Rysunek 19 przedstawia mały przykład.




Przykład na rysunku 19 to program Dart, ale nie jest to program Flutter. Aby uruchomić ten program, nie zawracałem sobie głowy tworzeniem projektu Flutter. Zamiast tego odwiedziłem https://dartpad.dev, wpisałem kod w dużym oknie edytora strony, a następnie nacisnąłem Uruchom. Na rysunku 19 wyjściem programu jest kolumna zawierająca liczby od 1 do 5. Dzieje się tak, ponieważ instrukcja for nakazuje urządzeniu powtarzanie czynności w kółko. Rysunek 20 przedstawia anglojęzyczną parafrazę instrukcji for z rysunku 19. Fakt, że Dart ma oświadczenie for, nie jest warty opublikowania. Dart's for statement jest prawie dokładnie taki sam, jak język C for statement, który został stworzony na początku lat 70. przez Dennisa Ritchiego z Bell Labs. A język C dla instrukcji jest bezpośrednim potomkiem instrukcji DO w FORTRANIE z wczesnych lat 60-tych. Nowością i ekscytacją w Dart jest pomysł, że możesz umieścić konstrukcję for na liście języków Dart. Listing 9 zawiera pouczający fragment kodu.




**LISTING 9 Ciekawy kod!**

```
Widget buildTitlePageCore() {

return ListView(

children: <Widget>[

for (int index = 0; index < 25; index++)

ListTile(

title: Text('Rocky ${index + 1}'),

onTap: () => goToDetailPage(index +
```

```
1),
),
],
);
}
```

Jeśli zastąpisz metodę buildTitlePageCore z listingu 7 kodem z listingu 9, Twoja aplikacja będzie zachowywać się dokładnie tak samo. Kiedy Flutter natrafi na kod z listingu 9, zaczyna tworzyć 25 widżetów ListTile. Listingi 7 i 9 przedstawiają dwa sposoby tworzenia 25-elementowego ListView. Który sposób jest lepszy? Możesz zdecydować, pytając: „W jaki sposób kod jest łatwiejszy do odczytania i zrozumienia?” Moim zdaniem nowy kod jest dużo bardziej przejrzysty. Elementy z listingu 9 wyglądają jak instrukcja for, ale tak naprawdę nie są instrukcją for. To kolekcja dla Nazwa kolekcji dla pochodzi z faktu, że lista jest jednym z typów kolekcji Darta. Możesz umieścić kolekcję wewnątrz dowolnego rodzaju kolekcji — Listy, Zestawu lub Mapy. Poniższy kod wykonuje wszystkie trzy czynności:

```
main() {
List<int> myList = [for (int i = 1; i <= 5;
i++) i];
Set<int> mySet = {for (int i = 1; i <= 5;
i++) i};
Map<int, int> myMap = {for (int i = 1; i <=
5; i++) i: i + 100};
print(myList);
print(mySet);
print(myMap);
}
```

Kolekcja dla i kolekcja, jeśli funkcje działają z Dart w wersji 2.3 i nowszych. W przypadku wcześniejszych wersji Dart nie masz szczęścia.

Kolekcja Darta dla jest interesująca, ponieważ jest to nowy rodzaj konstrukcji języka programowania. Dwa filary języków programowania to instrukcje i wyrażenia, ale kolekcja for nie jest ani instrukcją, ani wyrażeniem

**Pobieranie danych z Internetu**

Czy przykłady z tego rozdziału przypominają ci filmy, które ci się podobały? Czy chcesz aplikację, która wyświetla fakty na ich temat? Jeśli tak, nie szukaj dalej niż na Listingu 10.

LISTING 10 Dostęp do danych online

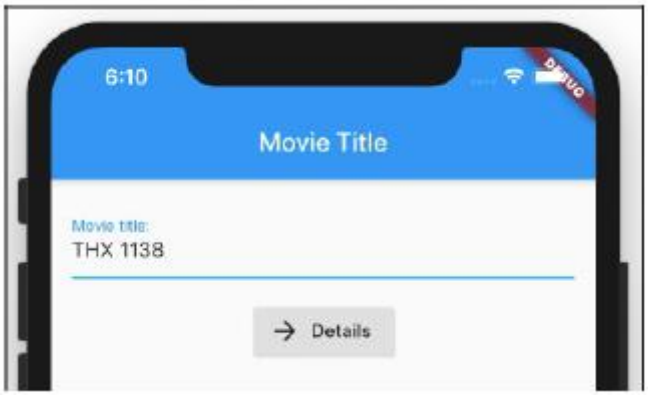
```
// App0810.dart
import ’dart:convert’;
```

```
import 'package:flutter/material.dart';
import ’package:http/http.dart’;
import 'App08Main.dart';
extension MoreMovieTitlePage on
MovieTitlePageState {
goToDetailPage(String movieTitle) {
Navigator.push(
context,
MaterialPageRoute(
builder: (context) => DetailPage(),
settings: RouteSettings(
arguments: movieTitle,
),
),
);
}
Widget buildTitlePageCore() {
TextEditingController _controller =
TextEditingController();
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
TextField(
decoration:
InputDecoration(labelText: 'Movie title:'),
controller: _controller,
),
SizedBox(height: 16.0),
RaisedButton.icon(
icon: Icon(Icons.arrow_forward),
```

```
label: Text('Details'),
onPressed: () =>
goToDetailPage(_controller.text),
),
],
);
}
}
extension MoreDetailPage on DetailPage {
Future<String> _getMovieData(String
movieTitle) {
return updateOverview(
movieTitle: movieTitle,
api_key: "Parents: Don’t let your sons
and "
"daughters put api keys in their
code.",
);
}
Widget buildDetailPageCore(context) {
final _movieTitle =
ModalRoute.of(context).settings.arguments ??
'';
return Column(
crossAxisAlignment:
CrossAxisAlignment.center,
children: <Widget>[
FutureBuilder<String>(
future: _getMovieData(_movieTitle),
builder: (context, snapshot) {
if (snapshot.hasData) {
```

```
return Text(snapshot.data);
}
return CircularProgressIndicator();
},
),
],
);
}
Future<String> updateOverview({String
api_key, String movieTitle}) async {
final response = await get(
’https://api.themoviedb.org/3/search/movie?
api_key=’ +
’$api_key&query="$movieTitle"’);
return json.decode(response.body)
[’results’][0][’overview’];
}
}
```

Rysunki 21 i 22 przedstawiają przebieg kodu z Listingu 10.




Na listingu 10 jest wiele rzeczy do rozpakowania, więc podzielę go na części.

**Korzystanie z publicznego interfejsu API**

Przed utworzeniem Listingu 10 przeszukałem Internet w poszukiwaniu strony, która zapewnia bezpłatny dostęp do informacji o filmach. Wśród witryn, które znalazłem, najbardziej podobała mi się The Movie Database (https://www.themoviedb.org). Podobnie jak wiele takich witryn, The Movie Database zapewnia dostęp za pośrednictwem własnego interfejsu programowania aplikacji (API). Gdy użyjesz zalecanego kodu API do wysłania zapytania do themoviedb.org, strona wypluwa informacje o jednym lub kilku filmach. Na przykład, aby uzyskać informacje o filmie THX 1138, możesz spróbować wpisać następujący adres URL w pasku adresu przeglądarki:

https://api.themoviedb.org/3/search/movie? api_key=XYZ&query="THX 1138"

Gdy to zrobisz, w oknie przeglądarki pojawi się następujący komunikat:

Nieprawidłowy klucz API: Musisz otrzymać prawidłowy klucz.

Ups! Zamiast wpisywać XYZ, powinieneś wpisać prawidłowy klucz API — ciąg znaków, który otrzymujesz podczas rejestracji w witrynie The Movie Database. Każdy, kto się zarejestruje, otrzymuje swój własny klucz API. Zdobądź własny klucz API, zastąp XYZ kluczem API i wpisz nowy adres URL w pasku adresu przeglądarki internetowej:

// To nie jest prawdziwy klucz API…

https://api.themoviedb.org/3/search/movie?

api_key=4c23b2f8f&query="THX 1138"

Kiedy to zrobisz, nie zobaczysz komunikatu o błędzie ani nie otrzymasz fantazyjnie wyglądającej strony internetowej. Zamiast tego otrzymujesz kod, który wygląda mniej więcej tak, jak na listingu 11.

**LISTING 8-11 Kod JSON**

{

"page": 1,

"total_results": 4,

"total_pages": 1,

"results": [

{

"popularity": 8.126,

.

.

.

"title": "THX 1138",

"vote_average": 6.6,

"overview": "People in the future live in

a totalitarian …",

"release_date": "1971-03-11"

},

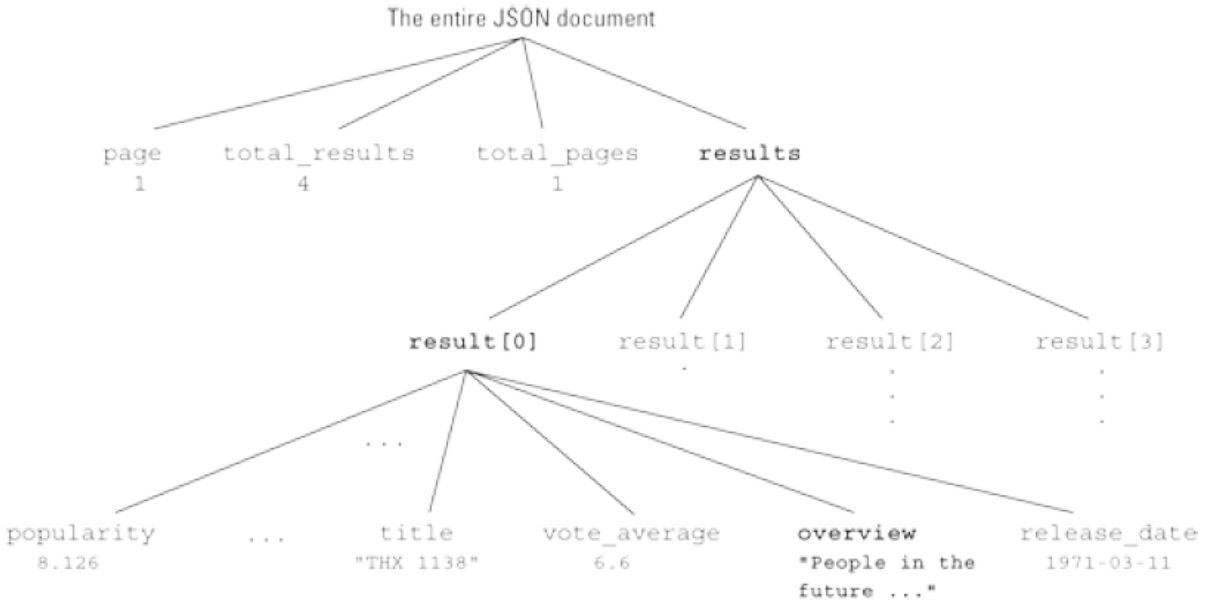
{

"popularity": 3.11,

"id": 140979,

. . . and more . . .

Tekst na listingu 11 nie jest kodem Dart. To kod JSON. Akronim JSON oznacza JavaScript Object Notation. Najlepszym sposobem na zrozumienie kodu JSON jest uświadomienie sobie, że opisuje on drzewo. Porównaj kod z listingu 11 z odwróconym drzewem na rysunku 23.




Przykładem RepresentationalState Transfer, znanego również jako REST, jest wysłanie adresu URL do serwera i otrzymanie w zamian kodu JSON. Twoim zadaniem jako programisty aplikacji jest sprawienie, by aplikacja spełniała dwie funkcje:

* Wyślij adres URL do bazy danych filmów.

*Zrozum kod JSON, który pochodzi z Movie Database.

**Wysyłanie adresu URL do serwera**

Jednym ze sposobów umożliwienia komunikacji z serwerem WWW jest zaimportowanie pakietu http Darta. Linia importu u góry Listingu 10 załatwia sprawę. Jedynym problemem jest to, że jeśli nie dodasz linii do pliku pubspec.yaml swojego projektu, Flutter nie będzie mógł zaimportować:

dependencies:

flutter:

sdk: flutter

http: ^0.12.0+4

Oczywiście, dziwnie wyglądająca wersja o numerze ^0.12.0+4 z pewnością będzie przestarzała.

W pliku .yaml wcięcia mają znaczenie. Dlatego w pliku pubspec.yaml swojego projektu pamiętaj o wcięciu wiersza http tak, jak widzisz go tutaj.

W kodzie Dart Twojej aplikacji używasz funkcji get pakietu wyślij adres URL do sieci:

final response = await get(

'https://api.themoviedb.org/3/search/movie?

api_key=' +

'$api_key&query="$movieTitle"');

Prosta nazwa funkcji, taka jak get, nie krzyczy: „Jestem częścią pakietu http”. Aby kod był bardziej czytelny, wykonaj dwie czynności: Dodaj kilka dodatkowych słów do deklaracji importu pakietu http

import 'package:http/http.dart' as http;

i dodaj prefiks do wywołania funkcji get:

final response = await http.get( // … etc.

Potrzeba oczekiwania, asynchronizacji i przyszłości na listingu 8.10 wynika z jednego niezaprzeczalnego faktu: jeśli wysyłasz żądanie do serwera WWW, nie wiesz, kiedy otrzymasz odpowiedź. Nie chcesz, aby Twoja aplikacja Flutter zawieszała się, czekając na odpowiedź od nie wiadomo skąd. Chcesz zabawiać użytkownika, podczas gdy odpowiedź trafia do Internetu. Dlatego na listingu 8.10 wyświetlasz widżet CircularProgressIndicator do czasu nadejścia odpowiedzi

**Zrozumienie odpowiedzi JSON**

Na listingu 10 metoda updateOverview oczekuje na odpowiedź z Movie Database. Gdy nadejdzie odpowiedź, metoda przypisuje tę odpowiedź do własnej zmiennej o nazwie response. (Jak sprytnie!) Zmienna odpowiedzi zawiera wszelkiego rodzaju informacje o nagłówkach HTTP i kodach stanu, ale zawiera także treść, która wygląda jak kod JSON z listingu 11. Ale poczekaj! Jak odsiać informacje z całego tego kodu JSON? Powiem ci jak. Wywołujesz funkcję json.decode — jedną z wielu funkcji w pakiecie konwersji Darta. Funkcja json.decode przekształca kod z Listingu 11 w dużą strukturę Dart Map. Podobnie jak wszystkie mapy Darta, ta mapa ma klucze i wartości, a niektóre wartości mogą być listami. Używasz nawiasów kwadratowych, aby uzyskać wartości z map i list. Aby wyciągnąć przegląd filmu z kodu z listingu 11, napisz następujący wiersz:

return json.decode(response.body)['results'][0]

['overview'];

Każda para nawiasów kwadratowych przybliża cię do dolnej części drzewa na rysunku 8.23..

Co dalej?

Navigator.push(

context,

MaterialPageRoute(

builder: (context) => Chapter_9(),

),

);

**Poruszanie się w prawo…**

Jeśli czytałeś inne materiały , prawdopodobnie masz dość moich długich wstępów , z wszystkimi ich osobistymi historiami i kiepskimi dowcipami. Dlatego tu pomijam wstępy i przechodzę od razu do sedna. Ta część dotyczy animacji - zmieniania rzeczy na oczach użytkownika. Kiedy myślę o animacji, od razu myślę o ruchu, ale Flutter zapewnia znacznie szerszą definicję animacji. Dzięki Flutter możesz zmienić prawie każdą właściwość widżetu w niemal dowolnej skali czasowej.

**Przygotowanie sceny dla animacji Flutter**

Pierwszy listing w tym rozdziale zawiera kilka kodów wielokrotnego użytku. Kolejne wpisy zawierają kod, który współpracuje z kodem z pierwszego wpisu. Dzięki funkcji rozszerzeń Darta, każdy nowy listing może tworzyć metody należące do klas pierwszego listingu. Możesz przeczytać wszystko o rozszerzeniach Dart w Części 8. Kod z Listingu 1 sam nie może nic zrobić. Zamiast tego ten kod opiera się na deklaracjach w innych listingach tego rozdziału.

**LISTING 1 Ponownie wykorzystaj ten kod**

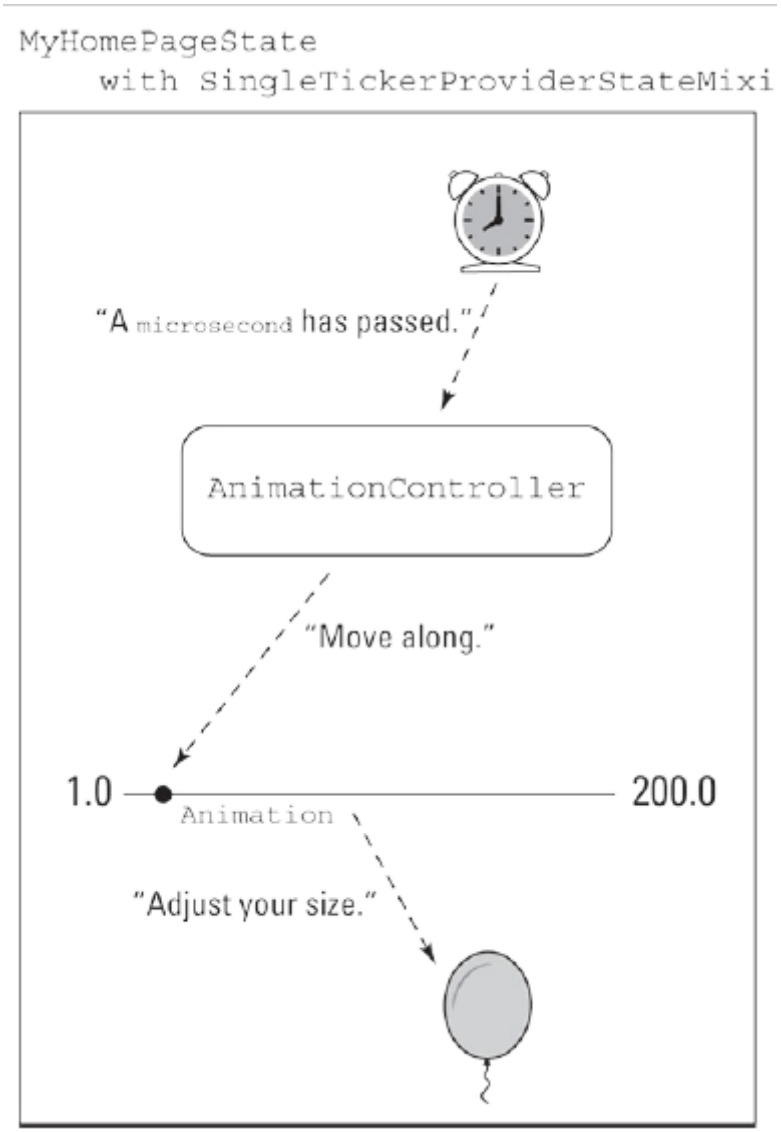
```
// App09Main.dart

import 'package:flutter/material.dart';

import 'App0902.dart'; // Change to App0903,

App0904, and so on.

void main() => runApp(App09Main());

class App09Main extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: MyHomePage(),

);

}

}

class MyHomePage extends StatefulWidget {

@override

MyHomePageState createState() =>

MyHomePageState();

}

class MyHomePageState extends State<MyHomePage>

with SingleTickerProviderStateMixin {

Animation<double> animation;
```

```
AnimationController controller;
@override
void initState() {
super.initState();
controller =
AnimationController(duration: const
Duration(seconds: 3), vsync: this);
animation = getAnimation(controller);
}
@override
Widget build(BuildContext context) {
return Material(
child: SafeArea(
child: Padding(
padding: const EdgeInsets.all(8.0),
child: Column(
children: <Widget>[
Expanded(
child: Stack(
children: <Widget>[
buildPositionedWidget(),
],
),
),
buildRowOfButtons(),
],
),
),
),
);
}
```

```
Widget buildRowOfButtons() {
return Row(
mainAxisAlignment:
MainAxisAlignment.center,
children: <Widget>[
RaisedButton(
onPressed: () =>
controller.forward(),
child: Text('Forward'),
),
SizedBox(
width: 8.0,
),
RaisedButton(
onPressed: () =>
controller.animateBack(0.0),
child: Text('Backward'),
),
SizedBox(
width: 8.0,
),
RaisedButton(
onPressed: () => controller.reset(),
child: Text('Reset'),
),
],
);
}
@override
void dispose() {
controller.dispose();
```

```
super.dispose();

}

}
```

Rysunek 1 ilustruje koncepcje, które składają się na animację Flutter.




Chcesz, aby coś się zmieniło, gdy użytkownik patrzy. Aby to zrobić, potrzebujesz czterech rzeczy: animacji, kontrolera animacji, paska i funkcji aplikacji, która się zmienia. Oto jak to wszystko działa:

* Animacja to plan zmiany wartości.

Na listingu 1 słowa Animation<double> wskazują, że zmieniającą się wartością jest liczba z cyframi poza przecinkiem — liczba taka jak 0,0, 0,5 lub 0,75. Plan przedstawiony na rysunku 1 polega na zmianie wartości w zakresie od 1,0 do 200,0. Sama animacja nie dotyczy żadnego ruchu. Wartość od 1,0 do 200,0 może być pozycją, ale może to być również rozmiar, stopień przezroczystości, stopień obrotu lub cokolwiek innego. W przypadku zmiennej animacji z listingu 9.1 wartości takie jak 1,0 i 200,0 są tylko liczbami. Nic więcej. Nawiasem mówiąc, jeśli szukasz w listingu 9.1 odniesienia do podwójnej wartości animacji, przestań szukać. Kod z listingu 9.1 nie odnosi się do takiej wartości. Jeśli spojrzysz do następnej sekcji, zobaczysz wartość animacji. To namacalny dowód na to, że instancja Animation ma

jakąś wartość. Klasa animacji Fluttera jest niezła, ale animacja nie może wiele zrobić bez kontrolera AnimationController. Dlatego:

* AnimationController sprawia, że animacja zaczyna się, zatrzymuje, idzie do przodu, cofa się, powtarza i tak dalej. Wywołania takie jak controller.forward(), controller.animateBack(0.0) i controller.reset() popychają animację w jednym lub drugim kierunku.

Na listingu 1 wywołanie konstruktora AnimationController mówi, że animacja trwa 3 sekundy. Jeśli sekundy to za mało, możesz użyć innych parametrów, takich jak mikrosekundy, milisekundy, minuty, godziny i dni. Każdy z poniższych konstruktorów opisuje 51 godzin:

Czas trwania (godziny: 51)

Czas trwania (dni: 1, godziny: 27)

Czas trwania (dni: 2, godziny: 3)

Czas trwania (minuty: 3060)

*Oprócz czasu trwania AnimationController z listingu 1 ma właściwość vsync. Jeśli zastanawiasz się, co to jest, czytaj dalej. * Pasek powiadomi kontroler AnimationController, kiedy minie każdy przedział czasu. Słowa z SingleTickerProviderStateMixin na listingu 1 zamieniają MyHomePageState w ticker. Ticker budzi się wielokrotnie i mówi: „Czas zmienić wartość”. Ale która wartość zostanie zmieniona? Która część kodu słyszy zapowiedź paska? Ustawianie MyHomePageState jako ticker nie łączy MyHomePageState z konkretnym kontrolerem AnimationController. Aby nawiązać to połączenie, AnimationController na listingu 1 ma parametr vsync: this. Ten parametr mówi Flutterowi, że „ta instancja MyHomePageState jest tickerem dla nowo skonstruowanego AnimationController”.

Starannie sformułowałem moje wyjaśnienie pasków i vsync: to wyjaśnienie nie urazi nikogo, kto zna wszystkie szczegóły. Problem w tym, że precyzyjne wyjaśnienia mogą być trudne aby je zrozumieć. Jeśli nie rozumiesz wszystkich rzeczy na temat vsync: to, po prostu dodaj te słowa do własnego kodu, a następnie przejdź dalej. Żaden z przykładów w tej książce nie wymaga dogłębnego zrozumienia pasków i vsync.

Na listingu 1 nazwa SingleTickerProviderStateMixin sugeruje, że język programowania Dart ma coś, co nazywa się Mixin. Mixin to coś w rodzaju rozszerzenia, z tą różnicą, że to nie to samo, co rozszerzenie. Oto końcowy składnik animacji Flutter:

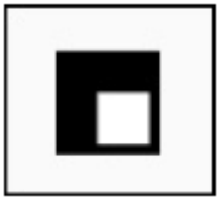
*Niektóre funkcje zmieniają się w wyniku zmiany wartości animacji. Na rysunku 1 rozmiar dymku zmienia się wraz z podwójną wartością instancji Animation. Ale kod z listingu 1 nie odnosi się do rozmiaru balonu ani do żadnego innego wykorzystania wartości zmiennej animacji. Pod tym względem Listing 1 jest nieco chybiony. Kod służący do wprowadzania zmian znajduje się w funkcji buildPositionedWidget, a treść tej funkcji znajduje się na listingach od 2 do 6. Każda z tych list robi coś innego z podwójnymi wartościami obiektu Animation.

Listing 1 ma jeszcze jedną interesującą cechę: posiada miejsce, w którym widżety mogą się swobodnie przemieszczać. Wyobraź sobie, że ikona jest dzieckiem widżetu Centrum. Widżet Centrum określa pozycję ikony i to koniec historii. Konstruktor widżetu Center nie ma parametrów, które pozwalają na poruszanie jego elementem potomnym w jednym lub drugim kierunku. Nie przejmuj się próbami przesunięcia elementu podrzędnego widżetu Centrum. Nie masz słownictwa, aby to przenieść. Potrzebujesz widżetu, który pozwoli ci zaznaczyć dokładne współrzędne jego dzieci w dostępnej przestrzeni. W tym celu Flutter ma stos. Stos jest jak wiersz lub kolumna, ale stos nie umieszcza swoich elementów podrzędnych w linii prostej. Zamiast tego stos ma dwa rodzaje elementów podrzędnych —

pozycjonowane widżety i wszystkie inne rodzaje widżetów. Każdy pozycjonowany widżet może mieć właściwości top, bottom, left i right, które określają dokładną lokalizację elementu podrzędnego pozycjonowanego widżetu. Inne widżety (te, które nie są pozycjonowane) są umieszczane w jakiejś domyślnej lokalizacji. Spójrz na następujący kod:

```
Stack(
children: <Widget>[
Positioned(
top: 100.0,
left: 100.0,
child: Container(
width: 50.0,
height: 50.0,
color: Colors.black,
),
),
Positioned(
top: 120.0,
left: 120.0,
child: Container(
width: 25.0,
height: 25.0,
color: Colors.white,
),
),
],
)
```

Ten kod tworzy rysunek pokazany na rysunku 2.



Rysunek składa się z dwóch prostokątów pojemnika — jednego czarnego, a drugiego białego. Szerokość i wysokość białego prostokąta są o połowę mniejsze od czarnego prostokąta. Ale zauważ to: dwa prostokąty zachodzą na siebie, ponieważ ich górna i lewa krawędź są prawie takie same.

Konstruktor stosu ma parametr potomny, a wartością tego parametru jest lista. Kolejność widżetów na liście ma znaczenie. Jeśli dwa widżety nakładają się na siebie, widżet znajdujący się później na liście wydaje się znajdować na górze. W kodzie z rysunku 2 nie chcesz zmieniać kolejności dwóch widżetów Positioned na liście. Jeśli to zrobisz, biały prostokąt zostanie całkowicie ukryty za większym czarnym prostokątem.

**Poruszanie się po linii prostej**

Listing 2 zawiera rozszerzenie kodu z listingu 1

LISTING 2 Idąc w dół

```
// App0902.dart

import 'package:flutter/material.dart';

import 'App09Main.dart';

extension MyHomePageStateExtension on

MyHomePageState {

Animation getAnimation(AnimationController

controller) {

Tween tween = Tween<double>(begin: 100.0,

end: 500.0);

Animation animation =

tween.animate(controller);

animation.addListener(() {

setState(() {});

});

return animation;

}

Widget buildPositionedWidget() {

return Positioned(

left: 150.0,

top: animation.value,

child: Icon(

Icons.music_note,
```

size: 70.0,

),

);

}

}

Listingi 1 i  2 razem tworzą kompletną aplikację Flutter. Rysunek 3 pokazuje, jak wygląda aplikacja po uruchomieniu. Linia przerywana to mój sposób na zilustrowanie ruchu ikony Musical Note w aplikacji. (Linia przerywana w rzeczywistości nie jest częścią aplikacji). Listing 2 zawiera metodę buildPositionedWidget , deklaracja, której brakuje na listingu 1. W ciele metody umieszczony widżet informuje Fluttera, gdzie powinien pojawić się jego element potomny (ikona nuty). Gdy aplikacja zacznie działać, liczby

left: 150,

top: animation.value,

umieszczą ikonę 150,0 dps od lewej krawędzi stosu i 100,0 dps od góry stosu. Liczba 100,0 pochodzi od wartości początkowej animacji, która jest zadeklarowana na początku Listingu 9-2. Wraz ze wzrostem wartości animacji ikona nuty przesuwa się w dół.

Listing 2 zawiera również metodę getAnimation — metodę, która została wywołana na listingu 1, ale nie została zadeklarowana na listingu 1. Metoda getAnimation z listingu 2 tworzy animację — rzecz wywodzącą się ze świata animowanych kreskówek. Wyobraź sobie postać z kreskówki poruszającą ręką od lewej do prawej. Rysownik rysuje pozycję początkową i końcową ramienia, a komputer tworzy obrazy „pomiędzy” ramienia. W ten sam sposób instancja klasy Tween Fluttera ma wartości początkowe i końcowe. Gdy animacja porusza się do przodu, Flutter stopniowo zmienia te wartości od wartości początkowej do wartości końcowej. Pozostała część kodu metody getAnimation łączy animację ze wszystkimi pozostałymi elementami układanki:

*Wywołanie funkcji tween.animate(controller) tworzy rzeczywistą instancję animacji. Sposób, w jaki opisuję animację, może ci się wydawać, że animacja to to samo, co animacja. Ale nie jest. Na szczęście, jeśli stworzyłeś animację, możesz zrobić z niej animację. Na listingu 2 wywołanie tween.animate(controller) tworzy obiekt Animation. To krok we właściwym kierunku.

*Wywołanie addListener mówi MyHomePageState, aby odbudował się za każdym razem, gdy zmieni się wartość animacji. W tworzeniu aplikacji detektor to ogólna nazwa czegoś, co nasłuchuje zdarzeń. Kod z listingu 2 mówi:

Utwórz funkcję, która przerysowuje ekran, wywołując setState. Spraw, aby ta funkcja nasłuchiwała zmian wartości animacji. W ten sposób Flutter przerysowuje ekran za każdym razem, gdy zmienia się wartość animacji.

Każde wywołanie setState powoduje, że Flutter aktualizuje lewe i górne wartości widżetu Positioned na Listingu 2. Ponieważ left ma zawsze wartość 150,0, ikona nie porusza się na boki. Ale właściwość value obiektu animacji zmienia się z chwili na chwilę, więc ikona porusza się w górę iw dół wzdłuż ekranu.

AnimationController na listingu 1 określa ruch ikony:

* Gdy użytkownik naciśnie przycisk Forward w aplikacji, listing 1 wywołuje metodę controller.forward. Ikona przesuwa się w dół, jeśli nie znajduje się już na dole swojej trajektorii.

* Gdy użytkownik naciśnie przycisk Wstecz w aplikacji, Listing 1 wywoła funkcję controller.animateBack(0.0). Ikona przesuwa się w górę, jeśli nie jest już u góry. W świecie animacji bardzo przydatne są liczby od 0,0 do 1,0. W wywołaniu animateBack liczba 0.0 oznacza „przewiń animację do tyłu, aż osiągnie wartość początkową". Aby animacja osiągnęła punkt środkowy, wywołaj funkcję controller.animateBack(0.5).

* Gdy użytkownik naciśnie przycisk Reset w aplikacji, listing 1 wywołuje funkcję controller.reset(). Ikona przeskoczy do pozycji początkowej. (Jeśli jest już w pozycji początkowej, pozostaje tam.)

Możesz nigdy nie zobaczyć kodu z Listingu 2 w żadnej innej książce. Wersja metody getAnimation opisana w tej książce pozwala uniknąć sztuczki często stosowanej przez programistów Fluttera. Podsumowują całą treść metody w jednym oświadczeniu:

```
return Tween<double>(begin: 100.0, end:

500.0).animate(controller)

..addListener(() {

setState(() {});

});
```

W tym kodzie para kropek przed addListener to kaskadowy operator Darta. Operator wywołuje metodę addListener w instancji Animation, która ma zostać zwrócona. Użycie tego operatora znacznie upraszcza kod.

**INNY SPOSÓB NA PONOWNE WYKORZYSTANIE KODU**

Listing 2 ma rozszerzenie, a listing 1 zawiera domieszkę. Zarówno rozszerzenia, jak i mixiny to sposoby na wykorzystanie kodu z zewnętrznych źródeł. Czym różnią się mixiny od rozszerzeń? Podczas tworzenia rozszerzenia nadajesz nazwę klasie, którą chcesz rozszerzyć.

rozszerzenie MyHomePageStateExtension na MyHomePageState Ten kod z listingu 2 dodaje funkcjonalność tylko do jednej klasy — klasy MyHomePageState z listingu 1. Nie możesz użyć tego rozszerzenia w każdym innym kontekście. Z drugiej strony możesz dodać mixin do prawie każdej klasy. Oto deklaracja SingleTickerProviderStateMixin z API Fluttera:

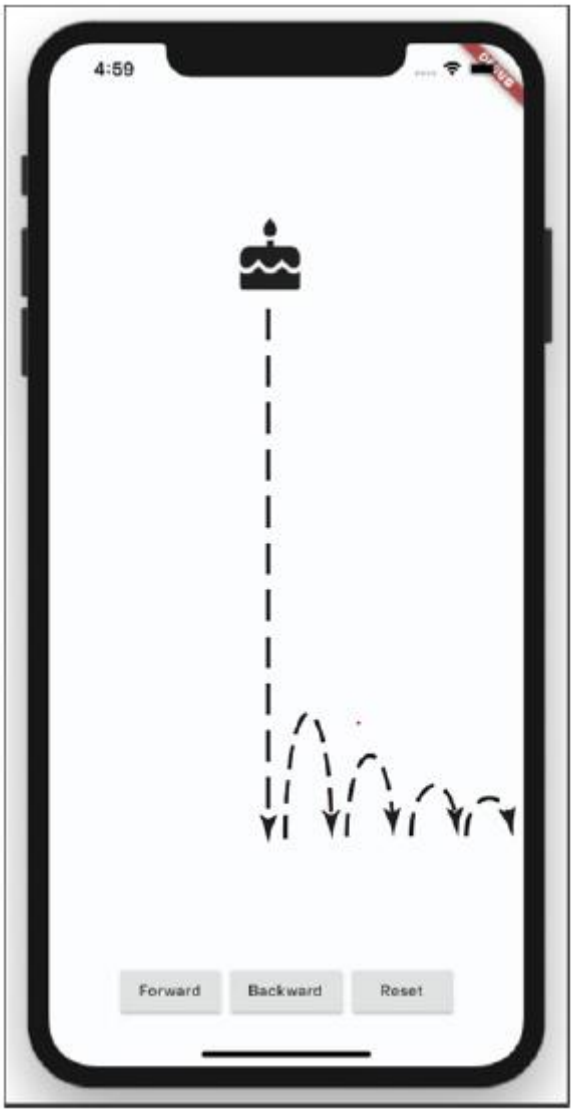
```
mixin SingleTickerProviderStateMixin<T

extends StatefulWidget> on State<T>

implements TickerProvider
```

Deklaracja nie mówi nic o klasie MyHomePageState lub

o każdej innej takiej klasie, więc każda klasa może używać tego miksu. (Cóż, każda klasa, która jest już StatefulWidget, może używać tego miksu). Dobrą rzeczą w mixinach jest to, że rozprzestrzeniają bogactwo. Stewardzi Fluttera piszą 75 linii kodu SingleTickerProviderStateMixin, w wyniku czego każdy StatefulWidget może stać się tickerem. Jak wygodnie!

**Odbijanie się**

Moim wielkim rozczarowaniem podczas pisania tego rozdziału jest to, że liczby nie oddają sprawiedliwości aplikacjom, które mają ilustrować. Rysunek 3 przedstawia przerywaną linię zamiast rzeczywistego ruchu. Rysunek 4 jest jeszcze gorszy, ponieważ linia przerywana nie jest tak naprawdę dokładna. W aplikacji tej sekcji ikona Cake nie porusza się na boki. Kropkowana linia na rysunku 4 przesuwa się w prawo tylko po to, by pokazać ruch w górę iw dół pod koniec animacji. Mimo to API Fluttera nazywa ten ruch krzywą. Kod z rysunku 4 znajduje się na listingu 3.




**LISTING 3 Zmiana prędkości animacji**

```
// App0903.dart

import 'package:flutter/material.dart';

import 'App09Main.dart';

extension MyHomePageStateExtension on

MyHomePageState {

Animation getAnimation(AnimationController
```

```
controller) {
return Tween<double>(begin: 100.0, end:
500.0).animate(
CurvedAnimation(
parent: controller,
curve: Curves.bounceOut,
),
)..addListener(() {
setState(() {});
});
}
Widget buildPositionedWidget() {
return Positioned(
left: 150.0,
top: animation.value,
child: Icon(
Icons.cake,
size: 70.0,
),
);
}
}
```

Ponownie, aby zmienić właściwości obiektu, umieszczasz ten obiekt wewnątrz innego obiektu. Jest to wzorzec, który pojawia się w kółko podczas tworzenia aplikacji Flutter. Zamiast wywoływać funkcję animate(controller) w sposób pokazany na listingu 2, wywołasz

```
animate(
CurvedAnimation(
parent: controller,
curve: Curves.bounceOut,
)
```

Owijasz kontroler wewnątrz obiektu CurvedAnimation. Na listingu 2 właściwość curve obiektu to Curves.bounce.Out, co oznacza „odbijanie się po zakończeniu animacji”. Tabela 1 zawiera listę

niektórych alternatywnych wartości krzywych. Flutter API ma o wiele więcej wartości krzywych. Każda wartość pochodzi z precyzyjnego równania i opisuje swój własny, wyjątkowymwzorzec synchronizacji animacji.

**Niektóre stałe klasy Curves**

**Wartość: co robi**

Curves.bounceIn : Odbija się na początku animacji

Curves.decelerate: Zwalnia w miarę postępu animacji

Curves.slowMiddle : Porusza się normalnie, potem powoli, a potem normalnie

Curves.fastOutSlowIn : (Możesz zgadnąć?)

Curve.ease : Przyspiesza szybko, ale kończy się powoli

Curve.elasticOut : Pędzi wystarczająco szybko, aby przekroczyć wartość końcową, a następnie ustala wartość końcową

Curve.linear : niczego nie zmienia (używane, gdy z jakiegoś powodu musisz użyć CurvedAnimation, ale nie chcesz zastosować krzywej)

**Animowanie zmian rozmiaru i koloru**

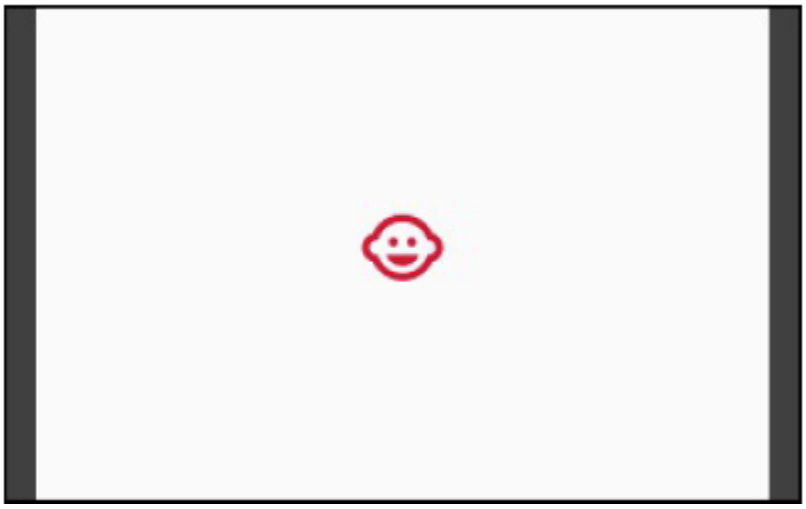
Dzięki klasie animacji Flutter nie jesteś ograniczony do przenoszenia rzeczy. Możesz kontrolować zmianę dowolnej wartości, która Twoim zdaniem wymaga zmiany. Przykład w tej sekcji zmienia rozmiar i kolor ikony. Kod znajduje się na listingu 4.

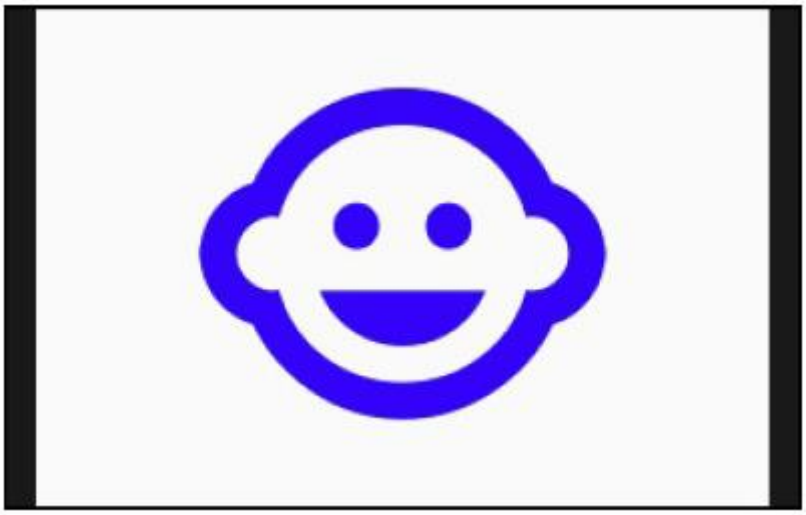
**LISTING 4 Zmiana kilku wartości**

```
// App0904.dart

import 'package:flutter/material.dart';

import 'App09Main.dart';

extension MyHomePageStateExtension on

MyHomePageState {

Animation getAnimation(AnimationController

controller) {

return Tween<double>(begin: 50.0, end:

250.0).animate(controller)

..addListener(() {

setState(() {});

});

}
```

```
Widget buildPositionedWidget() {
int intValue = animation.value.toInt();
return Center(
child: Icon(
Icons.child_care,
size: animation.value,
color: Color.fromRGBO(
intValue,
0,
255 - intValue,
1.0,
),
),
);
}
}
```

Kiedy aplikacja z listingu 4 zaczyna działać, na ekranie pojawia się mała, niebieska twarz dziecka. Kiedy użytkownik naciska przycisk Dalej, twarz dziecka rośnie i zmienia kolor z niebieskiego na czerwony.





Ikona na listingu 4 ma dwie właściwości, których wartości mogą się zmieniać.

* Właściwość size zmienia się wraz z wartością animacji. Ikona rośnie z 50,0 dps do 250,0 dps.

* W miarę postępu animacji czerwień właściwości koloru zmniejsza się, a jej błękit rośnie.

Część 6 przedstawia konstruktora Color.fromRGBO Fluttera. Parametrami konstruktora są wartości int reprezentujące ilości koloru czerwonego, zielonego i niebieskiego oraz wartość double reprezentująca krycie. Na listingu 4 ilość czerwieni wzrasta z 50 do 250, a ilość niebieskiego maleje z 205 do 5. Ta sekcja jest już prawie na ukończeniu. Morał płynący z tej sekcji jest taki, że wartość instancji animacji może oznaczać wszystko, co chcesz. Na listingach 2 i 3 wartość animacji steruje pozycją ikony. Jednak na listingu 9.4 wartość animacji steruje rozmiarem i kolorem ikony. Jaką wartość chciałbyś animować? Obrót? Głośność dźwięku? Prędkość? Krzywizna? Cień? Kolor tła? Kształt granicy? Nastrój? Cena książki Dla bystrzaków? Bądź kreatywny.

**Poruszanie się po krzywej**

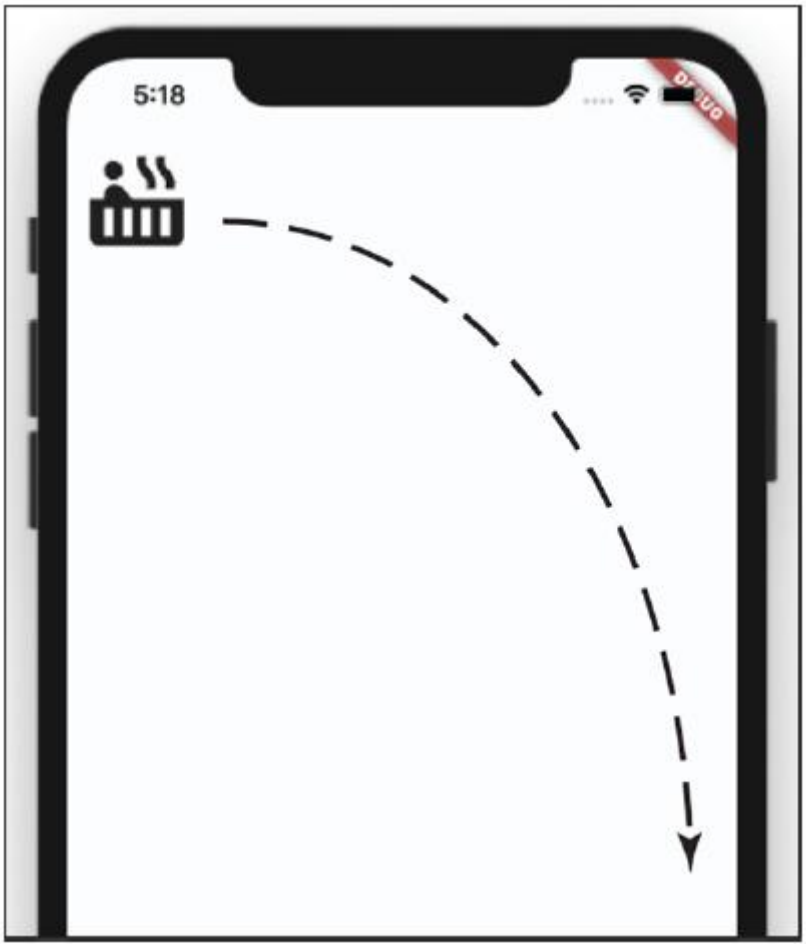
Życie nie zawsze toczy się po prostej linii. Czasami los zatacza zakręty i zakręty. Aby tak się stało we Flutterze, nie musisz nic zmieniać w animacji. Zamiast tego zmieniasz sposób korzystania z wartości animacji. Wywołanie konstruktora Tween w przykładzie z tej sekcji jest prawie identyczne z wywołaniami w innych wykazach tego rozdziału. Tym, co wyróżnia przykład z tej sekcji, są parametry pozycjonowanego widżetu. Wszystko to znajduje się na listingu 5.

**LISTING 5 Fantazyjny ruch paraboliczny**

```
// App0905.dart

import 'dart:math';

import 'package:flutter/material.dart';

import 'App09Main.dart';

extension MyHomePageStateExtension on

MyHomePageState {

Animation getAnimation(AnimationController

controller) {
```

```
return Tween<double>(begin: 0.0, end:
400.0).animate(controller)
..addListener(() {
setState(() {});
});
}
Widget buildPositionedWidget() {
double newValue = animation.value;
return Positioned(
left: 15 * sqrt(newValue),
top: newValue,
child: Icon(
Icons.hot_tub,
size: 70,
),
);
}
}
```

Na rysunku 7 kropkowana linia pokazuje ścieżkę, którą podąża ikona Hot Tub, gdy animacja przesuwa się do przodu.




Spójrz na kod z listingu 5. Wraz ze wzrostem wartości animacji zmieniają się zarówno lewe, jak i górne wartości parametrów ikony. Górny parametr jest taki sam jak wartość animacji, ale lewy parametr jest 15-krotnością pierwiastka kwadratowego wartości animacji. Jak wpadłem na pomysł, aby wziąć 15-krotność pierwiastka kwadratowego z wartości animacji? To częściowo znajomość matematyki, a częściowo metoda prób i błędów. Możesz użyć funkcji sqrt Darta tylko wtedy, gdy importujesz plik dart.math. Gdy zapomnisz zaimportować dart.math, Android Studio powie: „Metoda „sqrt" nie jest zdefiniowana". Przygotowując przykład w tej sekcji, dodałem trochę kodu, aby aplikacja wyświetlała wartości left i top. Oto, co mam

```
left:  top:
  0.0   0.0
  7.4  40.7
 22.1  70.5
 29.4  81.4
 41.2  96.2
 65.0 120.9
 71.8 127.1
 86.5 139.5
101.5 151.1
119.7 164.1
147.4 182.1
165.4 192.9
174.3 198.0
197.9 211.0
206.8 215.7
```

```
222.7 223.9
238.3 231.6
266.8 245.0
290.0 255.5
312.6 265.2
335.1 274.6
352.3 281.5
367.2 287.4
384.6 294.2
399.0 299.6
400.0 300.0
```

Lewa i górna wartość widżetu Pozycjonowane zmieniają się. Ale ze względu na formułę pierwiastka kwadratowego lewa i górna wartość zmieniają się w różnym tempie. Dlatego ruch ikony tworzy krzywą.

**Przeciąganie rzeczy dookoła**

W aplikacji tej sekcji użytkownik przeciąga widżet po całym ekranie. Chciałbym stworzyć figurę, aby pokazać, co się dzieje, ale po prostu nie mogę tego zrobić. Może moja następna książka Fluttera będzie wyskakującą książką z kartonowymi kawałkami, które można przesuwać z miejsca na miejsce. Do tego czasu musisz użyć wyobraźni. Wyobraź sobie ikonę, która wygląda jak symbol nieskończoności (∞). Gdy użytkownik porusza palcem, ikona zmienia położenie. Ale poczekaj! Zamiast wyobrażać sobie użytkownika przeciągającego ikonę, możesz uruchomić kod z listingu 6 i zobaczyć go w akcji.

**LISTING 6 Ćwiczenie na palec wskazujący użytkownika**

```
// App0906.dart

import 'package:flutter/material.dart';

import 'App09Main.dart';

double distanceFromLeft = 100;

double distanceFromTop = 100;

extension MyHomePageStateExtension on

MyHomePageState {

Animation getAnimation(AnimationController

controller) {

return null;

}

Widget buildPositionedWidget() {

return Positioned(

top: distanceFromTop,

left: distanceFromLeft,

child: GestureDetector(

onPanUpdate: (details) {

setState(() {
```

```
distanceFromLeft +=

details.delta.dx;

distanceFromTop +=

details.delta.dy;

});

},

child: Icon(

Icons.all_inclusive,

size: 70,

),

),

);

}
```

Podobnie jak inne listingi w tym rozdziale, listing 6 opiera się na kodzie z listingu 1. Z tego powodu aplikacja wygenerowana na listingu 6 ma przyciski Do przodu, Do tyłu i Resetuj. Mimo to naciskanie tych przycisków nie daje żadnego efektu. W ten sam sposób Listing 9.6 zawiera metodę getAnimation. Jest to konieczne, ponieważ kod z listingu 1 wywołuje metodę getAnimation. Ale aby widżet poruszał się wraz z palcem użytkownika, nie potrzebujesz instancji Animation. W pewnym sensie użytkownik jest kontrolerem AnimationController aplikacji, a instancja Animation znajduje się gdzieś w umyśle użytkownika. Tak więc na listingu 6 metoda getAnimation zwraca wartość null. W Dart null oznacza „nic", „nada", „zip", „gęsie jajo", „zilch", „diddly", „bupkis". Listing 9.6 nie zawiera instancji Animation, więc jaka część kodu powoduje przesunięcie ikony all_inclusive? Ikona znajduje się wewnątrz GestureDetector — widżetu, który wykrywa dotyk na ekranie. GestureDetector ma mnóstwo właściwości, takich jak onTap, onDoubleTap, onTapUp, onTapDown, onLongPress, onLongPressStart i onLongPressEnd. Inne metody należące do klasy GestureDetector mają nazwy o mniej niż oczywistym znaczeniu. Poniższa lista zawiera kilka (nieco uproszczonych) przykładów:

* onSecondaryTapDown: trzymając jeden palec na ekranie, użytkownik kładzie drugi palec na ekranie.

* onScaleUpdate: Za pomocą dwóch palców użytkownik szczypie lub wysuwa.

* onHorizontalDragUpdate: Użytkownik przesuwa coś w bok — powszechny gest służący do odrzucania elementu.

* onPanUpdate: Użytkownik przesuwa palec w jednym lub drugim kierunku.

Wartością parametru onPanUpdate jest metoda, a parametrem tej metody jest obiekt DragUpdateDetails. Na listingu 9.6 obiekt DragUpdateDetails występuje pod nazwą szczegóły:

```
onPanUpdate: (details) {

setState(() {

distanceFromLeft += details.delta.dx;
```

distanceFromTop += details.delta.dy;

});

Kiedy użytkownik przesuwa palcem po ekranie, Flutter wypełnia szczegóły informacjami o ruchu i wywołuje metodę parametru onPanUpdate. Zmienna details zawiera kilka przydatnych informacji:

* details.globalPosition: Odległość od lewego górnego rogu ekranu aplikacji do aktualnej pozycji palca użytkownika

* details.localPosition: Odległość od miejsca, w którym palec użytkownika po raz pierwszy wylądował na ekranie, do aktualnej pozycji palca użytkownika

* details.delta: Odległość od poprzedniej pozycji palca do jego bieżącej pozycji

Każda informacja składa się z dwóch części: dx (odległość pozioma) i dy (odległość pionowa). Widżet Positioned na listingu 6 umieszcza ikonę all_inclusive aplikacji w punktach distanceFromLeft i distanceFromTop. Gdy Flutter wykryje ruch palca, kod zmienia wartości distanceFromLeft i distanceFromTop, dodając wartości dx i dy parametru details.delta. To właśnie sprawia, że ikona się porusza. To całkiem sprytne!

GestureDectector z listingu 6 ma dziecko. Ale dla każdego starego wywołania konstruktora GestureDectector parametr child jest opcjonalny. GestureDetector bez elementu podrzędnego staje się tak duży, jak jego widżet nadrzędny. Natomiast GestureDetector z dzieckiem kurczy się, aby ściśle przylegać do dziecka. W aplikacji z listingu 6 GestureDetector ma mniej więcej taki sam rozmiar jak jego dziecko - ikona all_inclusive. Aby ikona się poruszyła, palec użytkownika musi zaczynać się dokładnie na ikonie. W przeciwnym razie nic się nie dzieje.

Zbliżasz się do końca, więc może czas się zrelaksować i hałaśliwą, beztroską zabawę. Czy niszczenie czegoś może być zabawne? Oto kilka sposobów na złamanie Listingu 6:

* Usuń wywołanie setState.

// Bad code:

onPanUpdate: (details) {

distanceFromLeft += details.delta.dx;

distanceFromTop += details.delta.dy;

}

Usunięcie wywołania setState prawie nigdy nie jest dobrym pomysłem. Jeśli usuniesz wywołanie z listingu 6, wartości distanceFromLeft i distanceFromTop zmienią się, ale Flutter nie przerysuje ekranu. W rezultacie ikona nie drgnie.

* Przenieś deklaracje distanceFromLeft i distanceFromTop tak, aby znajdowały się bezpośrednio przed metodą buildPositionedWidget.

// More bad code:

Animation getAnimation(AnimationController

controller) {

return null;

```
}

double distanceFromLeft = 100;

double distanceFromTop = 100;

Widget buildPositionedWidget() {

// … etc.
```

Jeśli to zrobisz, nie możesz nawet uruchomić aplikacji. Reguły Darta obejmują jedną dotyczącą deklarowania zmiennych najwyższego poziomu wewnątrz rozszerzeń. Po prostu nie wolno ci tego robić.

* Przenieś deklaracje distanceFromLeft i distanceFromTop tak, aby znalazły się wewnątrz metody buildPositionedWidget.

```
// Even more bad code:

Widget buildPositionedWidget() {

double distanceFromLeft = 100;

double distanceFromTop = 100;

return Positioned(

// … etc.
```

Program działa, ale ikona nigdy się nie porusza. Dzieje się tak, ponieważ kod ustawia distanceFromLeft i distanceFromTop na 100 za każdym razem, gdy Flutter przerysowuje ekran. (Właściwie ikona porusza się odrobinę, ale nie na tyle, abyś mógł to zauważyć. Otrzymujesz niewielką ilość ruchu z wartości details.delta, ale nie takiego ruchu, jaki chcesz).

* Zamiast dodawać wartości distanceFromLeft i distanceFromTop, ustaw je tak, aby odpowiadały pozycji palca użytkownika:

```
// You guessed it! Bad code!

onPanUpdate: (details) {

setState(() {

distanceFromLeft =

details.globalPosition.dx;

distanceFromTop =

details.globalPosition.dy;

});

}
```

Aplikacja działa, ale ikona przeskakuje, gdy palec użytkownika zaczyna się poruszać. Podczas gestu przeciągania ikona pozostaje pół cala od palca użytkownika. Dzieje się tak, ponieważ Flutter nie używa środka ikony jako górnego i lewego punktu pozycjonowanego widżetu. Podobne rzeczy dzieją się, jeśli spróbujesz użyć details.localPosition.

Funkcje animacji Fluttera nie kończą się na prostych ruchach i podstawowych zmianach rozmiaru. Jeśli interesuje Cię wprawianie obiektów w ruch, koniecznie sprawdź pakiet fizyki.dart Fluttera. Dzięki temu pakietowi możesz symulować sprężyny, grawitację, tarcie i wiele więcej.