O'REILLY®

Flutter i Dart Receptury

Tworzenie chmurowych aplikacji full stack

Helion

Richard Rose

Tytuł oryginału: Flutter and Dart Cookbook: Developing Full-Stack Applications for the Cloud

Tłumaczenie: Anna Mizerska

ISBN: 978-83-289-0709-6

Helion S.A.
ul. Kościuszki 1c, 44-100 Gliwice
tel. 32 230 98 63
e-mail: *helion@helion.pl*
WWW: *https://helion.pl* (księgarnia internetowa, katalog książek)

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

Printed in Poland.

Spis treści

ROZDZIAŁ 5.

Wprowadzenie do języka Dart zorientowanego obiektowo

W tym rozdziale wprowadzimy techniki programowania zorientowanego obiektowo, by pracować z klasami, i pokażemy, jak można ich używać w języku Dart. Odkryjesz tutaj zarówno deklaracje, jak rozbudowę obiektów. Te techniki są ważne, gdyż Dart jest językiem zorientowanym obiektowo. Nauka podstaw to dobry początek w rozwijaniu własnych umiejętności, a ponadto łatwiej Ci będzie korzystać z cudzego kodu.

Niniejszy rozdział zaczyna się od krótkiego przeglądu kluczowej terminologii związanej z programowaniem zorientowanym obiektowo w kontekście Darta. Następnie powiemy, jak do swojego repertuaru programistycznego wprowadzić klasę. W tym rozdziale omówimy:

- Podstawy programowania zorientowanego obiektowo.
- Dlaczego należy inicjalizować klasę za pomocą konstruktora?
- Obsługa dziedziczenia z użyciem słowa kluczowego `extends`.
- Definicja sygnatury klasy przez użycie interfejsu.
- Zebranie funkcjonalności klas przez domieszkowanie.

Gdy Twoje programowanie stanie się bardziej zaawansowane, będziesz w stanie używać niestandardowych klas, by spełnić wymagania swojej aplikacji. Nauka skutecznego posługiwania się klasami wymaga trochę czasu, więc rób małe kroki. Z czasem klasy staną się dla Ciebie czymś naturalnym i będziesz umiał(a) sprostać bardzo złożonym problemom.

5.1. Dart zorientowany obiektowo — początek

Problem

Chcesz programować obiektowo z użyciem języka Dart, by tworzyć komponenty do ponownego użytku.

Rozwiązanie

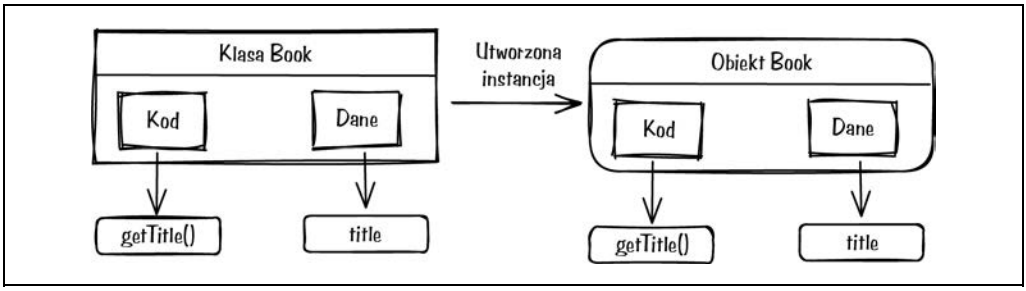
Użyj technik zorientowanych obiektowo, by rozwijać swój kod o obiekty do ponownego użycia. Język Dart obsługuje tego typu techniki, a co więcej, pozwala je stosować podczas tworzenia algorytmów i struktur danych potrzebnych do budowy aplikacji.

Omówienie

Gdy posiądziesz wiedzę o programowaniu obiektowym, Twój zestaw umiejętności w języku Dart i frameworku Flutter znacznie się wzbogaci. Tworzenie obiektów jest podstawową umiejętnością i pomoże Ci zrozumieć, jak łączyć algorytmy ze strukturami danych. Zwykle obiekt musi być inicjalizowany za pomocą specjalnej metody, nazywanej konstruktorem. Konstruktor ma za zadanie ustawienie właściwości w obiekcie na etapie inicjalizacji.

Obiekt grupuje kod i typy danych w reprezentatywną strukturę. Na przykład klasa `Book` (książka) i zainicjalizowany obiekt (innymi słowy, został utworzony egzemplarz klasy) może mieć właściwości `title` (książka), `author` (autor) i `publisher` (wydawca).

Na rysunku 5.1 klasa `Book` zapewnia definicję, a obiekt `Book` to wersja klasy używana podczas działania programu. Klasa `Book` może mieć również inne właściwości (na przykład `isbn`) lub metody, by uzyskać (`get`) lub zaktualizować (`set`) swoje właściwości. Gdy już będziesz znał lepiej język Dart, docenisz wygodę, jaką daje możliwość połączenia grupy danych z kodem w definicji klasy.



Rysunek 5.1. Przykład klasy Book

By zacząć programować obiektowo w języku Dart, zapoznaj się z przypadkami użycia dziedziczenia (patrz receptura 5.4), implementacji (patrz receptura 5.5) oraz rozbudowy (patrz receptura 5.6), by móc wybrać najodpowiedniejszy wzorzec projektowy dla swoich scenariuszy.

5.2. Tworzenie klasy

Problem

Chcesz utworzyć obiekt klasy, który będzie miał zarówno dane, jak i funkcje.

Rozwiązanie

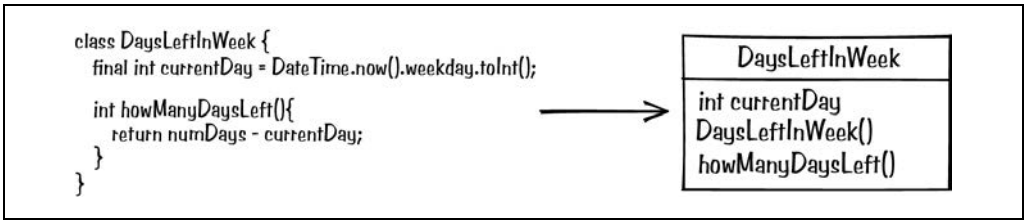
By zebrać informacje w nowym obiekcie, zapewniającym zarówno przechowywanie zmiennych, jak i funkcjonalność przetwarzania danych, użyj klasy. Oto przykład deklaracji klasy w języku Dart:

```
const numDays = 7;

class DaysLeftInWeek {
   int currentDay = 0;

   DaysLeftInWeek(){
      currentDay = DateTime.now().weekday.toInt();
   }

   int howManyDaysLeft(){
      return numDays - currentDay;
   }
}
```

Omówienie

Dart jest językiem zorientowanym obiektowo i ma klasę `Object` (*https://api.dart.dev/stable/2.14.4/dart-core/Object-class.html*, strona w języku angielskim) dla wszystkich obiektów poza `null`. Obiekt niemogący przyjmować wartości `null` jest podklasą klasy `Object`. Gdy zdobędziesz więcej doświadczenia w pisaniu kodu w języku Dart, tworzenie obiektów będzie dla Ciebie zupełnie naturalne. Programowanie zorientowane obiektowo zapewnia środki do modelowania pomysłów i przypisywania zachowań. Klasy udostępniają model, w którym możesz zdefiniować zarówno dane, jak i funkcjonalność, by uzyskać dostęp do danych zawartych w tym modelu.

Przykład z rysunku 5.2 pokazuje zastosowanie klasy w celu określenia, ile dni pozostało to końca tygodnia.



Rysunek 5.2. Deklaracja klasy

W definicji klasy pokazanej na rysunku 5.2 widzimy właściwość `currentDay` i metodę `howManyDaysLeft`.

W deklaracji występuje słowo kluczowe `class`, co zaznacza, że definicja, która po nim następuje, zawiera elementy zarówno dla zmiennych, jak i dla funkcji. Konstruktor klasy, `DaysLeftInWeek`, ma taką samą nazwę jak klasa i jest wywoływany podczas tworzenia instancji obiektu. Używaj konstruktora, by jednym ruchem utworzyć klasę.

W tej klasie zadeklarowaliśmy zmienną `currentDay`, która jest typu `int` i jest stałą `final`, co oznacza, że jej wartość będzie ustalona podczas działania aplikacji i będzie liczbą całkowitą. Ponadto zadeklarowaliśmy metodę `howManyDaysLeft`, służącą do wykonywania obliczeń.

5.3. Inicjalizacja klasy za pomocą konstruktora

Problem

Chcesz wykonać szereg instrukcji za każdym razem, gdy na podstawie klasy tworzony jest obiekt.

Rozwiązanie

By utworzyć instancję obiektu, użyj konstruktora klasy. Inicjalizacja może służyć do ustawienia wartości na domyślne z klasy.

Oto przykład deklaracji i użycia konstruktora klasy[1]:

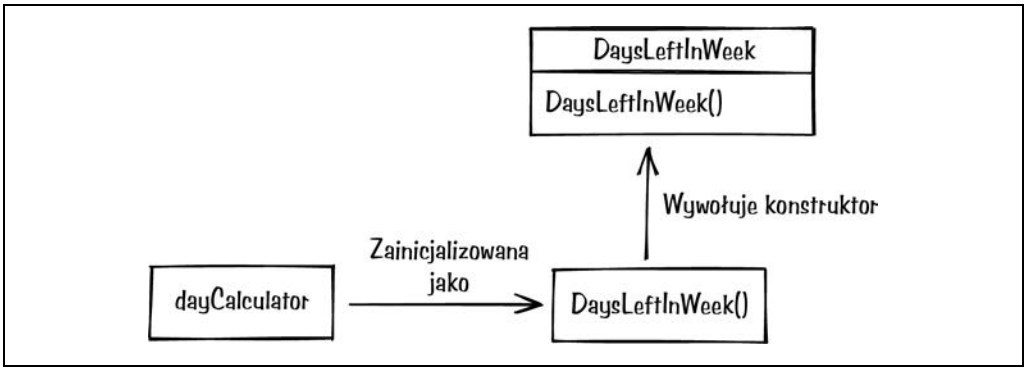
```
const numDays = 7;

class DaysLeftInWeek {
   int currentDay = 0;

   DaysLeftInWeek(){
      currentDay = DateTime.now().weekday.toInt();
   }

   int howManyDaysLeft(){
      return numDays - currentDay;
   }
}

void main() {
   DaysLeftInWeek dayCalculator = DaysLeftInWeek();

   print ('Dzisiaj jest ${dayCalculator.currentDay} dzień');
   print ('Do końca tygodnia zostało ${dayCalculator.howManyDaysLeft()} dni');
}
```

Omówienie

W tym przykładzie klasa służy do określenia, ile dni zostało do końca tygodnia. W deklaracji umieszcza się słowo kluczowe `class`, by zaznaczyć, że definicja, która po nim następuje, zawiera elementy zarówno dla zmiennych, jak i dla funkcji.

Spójrz na rysunek 5.3 i zwróć uwagę, że obiekt jest deklarowany przez wywołanie konstruktora klasy.

[1] W tym przykładzie w przypisaniu wartości do zmiennej `currentDay` pominięto słowo kluczowe `this`. Zalecane przez zespół Dart praktyki mówią, żeby pomijać słowo kluczowe `this`, jeśli nie jest wymagane.



Rysunek 5.3. Inicjalizacja za pomocą konstruktora klasy

Konstruktor klasy przyjmuje taką samą nazwę jak klasa oraz jest wywoływany podczas tworzenia instancji klasy. W tym przykładzie konstruktor w klasie ustawia zmienną `currentDay`, która ma wartość bieżącej daty. Ponadto zwróć uwagę, że w klasie `DaysLeftInWeek` mamy funkcję o takiej samej nazwie, jaką ma klasa.

Aby użyć klasy, zadeklaruj zmienną (na przykład `dayCalculator`) w celu utworzenia instancji klasy `DaysLeftInWeek`. Po zadeklarowaniu zmienna `dayCalculator` ma dostęp zarówno do zmiennych, jak i funkcji powiązanych z tą klasą. Podczas deklarowania zmiennej `dayCalculator` ponownie używamy słowa kluczowego `final`, by wskazać, że wartość tej zmiennej zostanie określona w trakcie działania aplikacji.

W końcu — wyrażenia `print` pokazują, jak dostać się do zmiennej i funkcji, by uzyskać z nich dane. W obu przypadkach wartość klasy pochodzi od zmiennej `dayCalculator`. Zarówno zmienna `currentDay`, jak i metoda `howManyDaysLeft` są w stanie uzyskać dane związane z klasą.

Jeśli znasz inny język zorientowany obiektowo, możesz być zaskoczony(-a) brakiem słowa kluczowego `this`. Dart wymaga użycia słowa kluczowego `this` tylko w celu jawnego wskazania zmiennej do użycia (czyli przesłaniania zmiennej). W dokumentacji języka (*https://dart.dev/guides*, strona w języku angielskim) znajdziesz więcej informacji o przesłanianiu zmiennej, które pokazują w skrócie, jak unikać takich sytuacji.

5.4. Dziedziczenie klasy

Problem

Chcesz wzbogacić istniejącą klasę przez wprowadzenie dodatkowej funkcjonalności, której nie ma w pierwotnej klasie.

Rozwiązanie

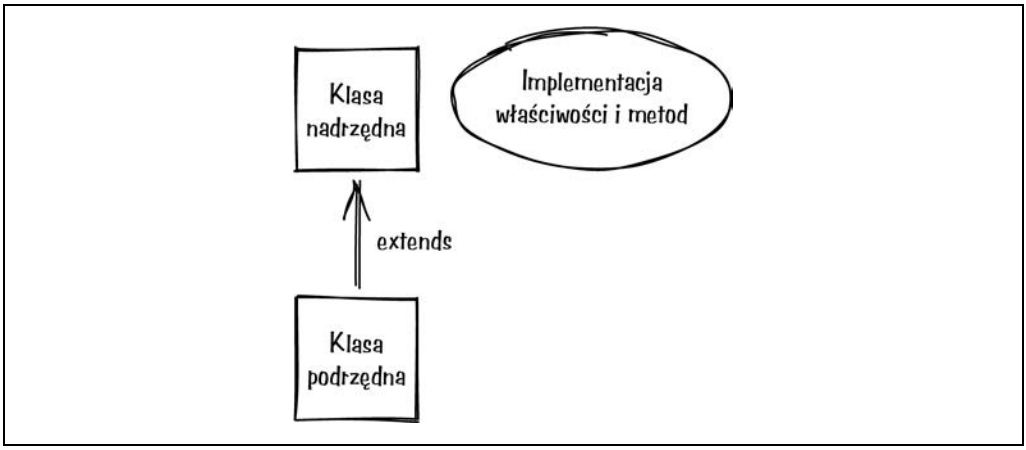
By wprowadzić dziedziczenie z klasy nadrzędnej, użyj klasy ze słowem kluczowym `extends`. Dzięki słowu kluczowemu `extends` podklasa uzyska funkcjonalność klasy nadrzędnej. Jako że język Dart

jest językiem zorientowanym obiektowo, z każdym nowym wydaniem zapewnia coraz bardziej rozbudowaną obsługę klas. Oto przykład, jak używać słowa kluczowego `extends`, by dodać dziedziczenie klasy:

```
class Media {
   String title = "";
   String type = "";

   Media(){ type = "Klasa"; }

   void setMediaTitle(String mediaTitle){ title = mediaTitle; }

   String getMediaTitle(){ return title; }

   String getMediaType(){ return type; }
}

class Book extends Media {
   String author = "";
   String isbn = "";

   Book(){ type = "Podklasa"; }

   void setBookAuthor(String bookAuthor){ author = bookAuthor; }

   void setBookISBN(String bookISBN){ isbn = bookISBN; }

   String getBookTitle(){ return title; }

   String getBookAuthor(){ return author; }

   String getBookISBN(){ return isbn; }
}
void main() {
   var myMedia = Media();

   myMedia.setMediaTitle('Gra o Tron');
   print ('Tytuł: ${myMedia.getMediaTitle()}');
   print ('Typ: ${myMedia.getMediaType()}');

   var myBook = Book();
   myBook.setMediaTitle("Księga Dżungli");
   myBook.setBookAuthor("R Kipling");
   print ('Tytuł: ${myBook.getMediaTitle()}');
   print ('Autot: ${myBook.getBookAuthor()}');
   print ('Typ: ${myBook.getMediaType()}');
}
```

Omówienie

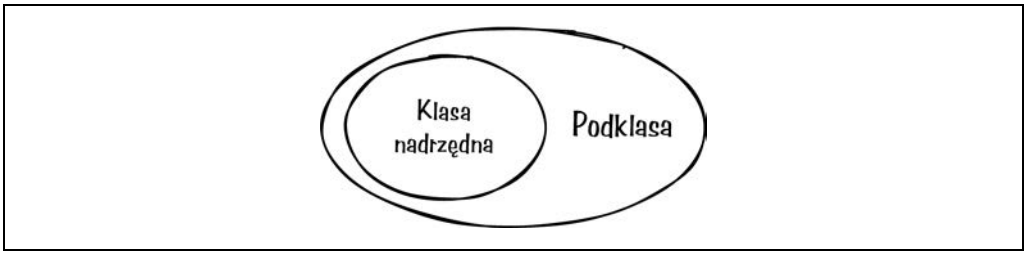
W przykładowym kodzie klasa `Media` jest poszerzona o podklasę `Book`. Funkcjonalność klasy `Media`, jako klasy nadrzędnej, będzie dostępna w każdej klasie podrzędnej, tak jak pokazano na rysunku 5.4.



Rysunek 5.4. Dziedziczenie klasy

Dziedziczenie umożliwia klasom przyjmowanie metod i właściwości klas nadrzędnych, tak jak pokazano na rysunku 5.4. W rezultacie klasa podrzędna i nadrzędna obsługują te same właściwości i metody.

I tak, zgodnie z tym, co widać na rysunku 5.5, klasa `Book` uwzględnia właściwości i metody powiązane z klasą `Media`, poza tym, co zostało bezpośrednio zdefiniowane w klasie `Book`.



Rysunek 5.5. Związek dziedziczenia klasy

Tworzymy podklasę `Book`, która poszerza klasę `Media`, co oznacza, że można jej użyć, by dostać się do metod i zmiennych klasy `Media`. Użycie podklasy umożliwia przesłanianie funkcjonalności, na przykład metod itd.

Słowo kluczowe `extends` zapewnia dziedziczenie klas, gdzie funkcjonalność nadrzędna jest dostępna dla klasy podrzędnej. Definiowana przez słowo kluczowe `extends` relacja między klasą nadrzędną a podrzędną to jeden-do-jednego, czyli kilka dziedziczeń nie jest obsługiwanych. Gdy korzystasz z tej relacji, pamiętaj, że prawdopodobnie będziesz musiał(a) wywoływać metodę `super.method()`, by się upewnić, że klasa nadrzędna wie o zmianach poczynionych w klasie podrzędnej.

Uważaj, żeby nie zadeklarować jeszcze raz metody `setMediaTitle` dla klasy `Book`, gdyż możemy ją wywołać tak, jakby była jawnie zadeklarowana w klasie `Book`.

Używanie słowa kluczowego `extends` to przydatne podejście tam, gdzie są podobne struktury danych potrzebujące nieco innych metod. W tym przykładzie klasa `Media` jest ogólną abstrakcją,

utworzoną, by przechowywać podstawowe informacje. Klasa `Book` jest specjalizacją klasy `Media` oferującą informacje szczególne dla książek.

5.5. Dodawanie interfejsu klasy

Problem

Chcesz użyć specyfikacji, by przedstawić zarys właściwości i metod do zadeklarowania podczas definiowania nowego obiektu.

Rozwiązanie

By zdefiniować specyfikacje dla obiektu, do których musisz się stosować, użyj interfejsu klasy. Oto przykład, jak definiować interfejs klasy w języku Dart:

```
abstract class Media {
   late String myId;
   late String myTitle;
   late String myType;

   void setMediaTitle(String mediaTitle);
   String getMediaTitle();

   void setMediaType(String mediaType);
   String getMediaType();

   void setMediaId(String mediaId);
   String getMediaId();
}

class Book implements Media {
   @override
   late String myId;
   @override
   late String myTitle;
   @override
   late String myType;

   @override
   void setMediaTitle(String mediaTitle) {
      myTitle = mediaTitle;
   }

   @override
   String getMediaTitle() {
      return myTitle;
   }

   @override
   void setMediaType(String mediaType) {
      myType = mediaType;
   }
```

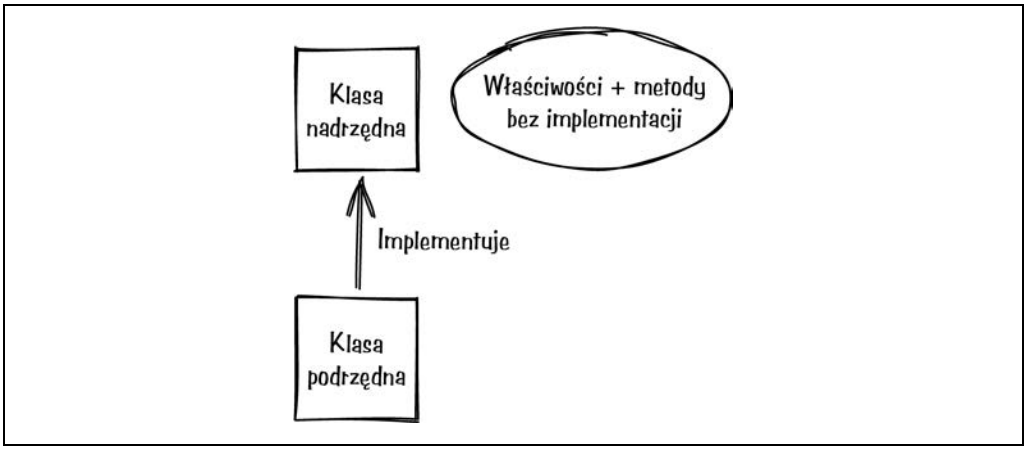
```
    @override
    String getMediaType() {
       return myType;
    }

    @override
    void setMediaId(String mediaId) {
       myType = mediaId;
    }

    @override
    String getMediaId() {
       return myId;
    }

    Book(String mediaTitle, String mediaType, String mediaId) {
       myTitle = mediaTitle;
       myType = mediaType;
       myId = mediaId;
    }
}

void main() {
    final Book myBook =
       Book("Bezserwerowe przetwarzanie danych z Google Cloud", "Książka", "ISBN-1111");

    print(myBook.getMediaTitle());
    print(myBook.getMediaType());
    print(myBook.getMediaId());
}
```

Omówienie

W tym przykładowym kodzie interfejs klasy `Media` jest używany przez podklasę `Book`. Klasa `Media` jest nadrzędna, więc jej definicje będą dostępne dla każdej klasy podrzędnej. Zwróć uwagę na brak implementacji i inicjalizacji powiązanej z klasą nadrzędną. Zamiast tego implementacja będzie zadaniem użytkownika interfejsu.

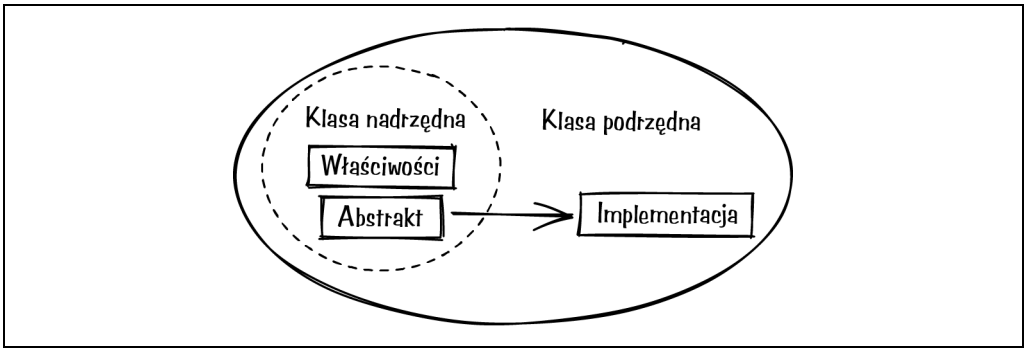
By użyć interfejsu klasy, zastosuj słowo kluczowe `implements`. Jeśli znasz jakiś inny język programowania, możliwe, że znasz termin **klasa abstrakcyjna**. Klasa abstrakcyjna zapewnia definicję klasy, ale nie może być użyta do utworzenia instancji obiektu.

W tym przykładzie klasa abstrakcyjna o nazwie `Media` tworzy ogólny interfejs dla informacji związanych z mediami, tak jak pokazano na rysunku 5.6. Klasa `Book` implementuje interfejs klasy `Media`, co oznacza, że jest odpowiedzialna za pobranie i ustawienie wartości nazwanych w klasie abstrakcyjnej. Zarówno właściwości, jak i metody muszą być zdefiniowane tam, gdzie przesłaniają wartości ustalone w interfejsie. Każda wartość zdefiniowana w klasie `Book`, która jest zdefiniowana w kasie `Media`, jest poprzedzona słowem `@override`, co znaczy, że interfejs został już wcześniej zdefiniowany.



Rysunek 5.6. Interfejs klasy

Zwykle interfejs klasy abstrakcyjnej służy do zdefiniowania generycznych typów, a przy tworzeniu podklasy musimy je zdefiniować. Na rysunku 5.7 pokazano relację między klasą nadrzędną a podrzędną.



Rysunek 5.7. Implementacja w klasie podrzędnej

Klasa podrzędna może implementować wiele interfejsów. Jednak podczas tworzenia podklasy musisz uważać, by hierarchia klas nie była zbyt skomplikowana. Aby stosować interfejs klasy, musisz go zaimplementować i trzymać się sygnatury używanej przez klasę abstrakcyjną.

Zwykle definicja interfejsu jest używana wtedy, gdy implementacja będzie obsługiwana przez osobne obiekty.

5.6. Domieszkowanie

Problem

Chcesz, by istniejąca klasa zawierała funkcjonalność z kilku hierarchii klas.

Rozwiązanie

Gdy potrzebujesz funkcjonalności z kilku klas, używaj obiektów `mixin`. Podczas pracy z klasami obiekty `mixin` zapewniają wiele możliwości i dzięki nim możemy wcielać informacje z wielu klas.

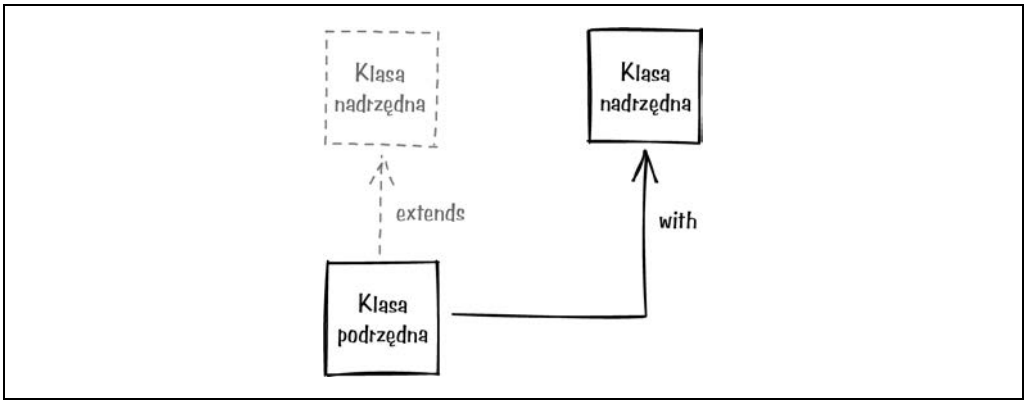
Oto przykład, jak używać obiektów `mixin`:

```
mixin SnickersOriginal {
   bool hasHazelnut = true;
   bool hasRice = false;
   bool hasAlmond = false;
}

mixin SnickersCrisp {
   bool hasHazelnut = true;
   bool hasRice = true;
   bool hasAlmond = false;
}

abstract class ChocolateBar {
   bool hasChocolate = true;
}

class CandyBar extends ChocolateBar with SnickersOriginal {
   List<String> ingredients = [];

   CandyBar(){
      if (hasChocolate){
         ingredients.add('Czekolada');
      }
      if (hasHazelnut){
         ingredients.add('Orzechy laskowe');
      }
      if (hasRice){
         ingredients.add('Ryż');
      }
      if (hasAlmond){
         ingredients.add('Migdały');
      }
    }

    List<String> getIngredients(){
       return ingredients;
   }
}

void main() {
   var snickersOriginal = CandyBar();
   print ('Skład:');
   snickersOriginal.getIngredients().forEach((ingredient) => print(ingredient));
}
```

Omówienie

W tym przykładzie mamy zdefiniowane dwa obiekty `mixin` do przechowywania informacji związanych z rodzajami batonów Snickers. Klasa bazowego batona nie zawiera wymaganej funkcjonalności,

dlatego by zwiększyć możliwości programu, włączamy nową klasę. Rysunek 5.8 pokazuje, jak można rozszerzać klasy za pomocą domieszek (ang. *mixin*).

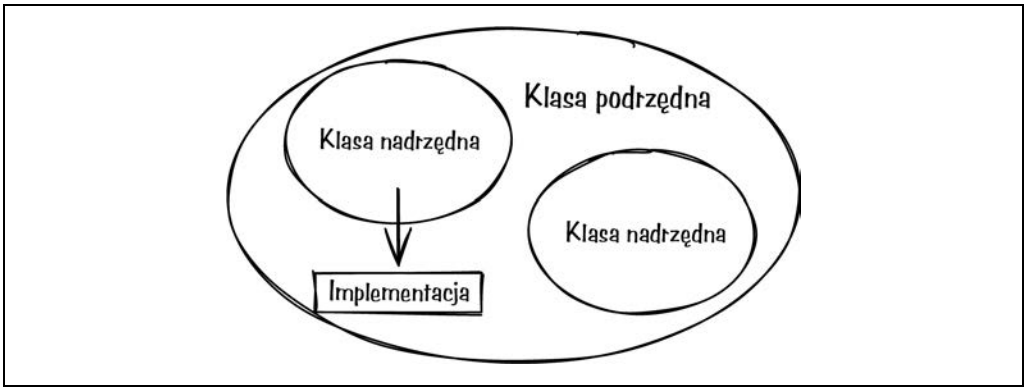


Rysunek 5.8. Rozszerzenie klasy przez domieszkowanie

Słowo kluczowe `with`, o które prosili sami programiści, wprowadzono do języka Dart niedawno. Podczas pracy z silnikiem gier Flame tego słowa kluczowego używa się bardzo często. Jeśli masz już doświadczenie w innych językach, możliwe, że wiesz, czym jest domieszkowanie, i znasz już słowo kluczowe `mixin`, które pozwala połączyć kilka klas, by zapewnić dodatkową funkcjonalność.

Słowa kluczowego `mixin` można używać zarówno z definicją dziedziczenia, jak i interfejsu klasy. Aby użyć klasy abstrakcyjnej z klasą `CandyBar`, stosujemy właśnie słowo kluczowe `mixin`. Z kolei obiekt `mixin`, gdy jest łączony z obiektem klasy, wymaga słowa kluczowego `with`.

Na rysunku 5.9 klasa nadrzędna, do której się odwołujemy, powinna być cały czas odizolowana, co oznacza, że użyte klasy nie mogą się nakładać. Klasa bazowa `CandyBar` nie może przesłaniać domyślnego konstruktora używanego w klasie abstrakcyjnej lub klasie nadrzędnej.



Rysunek 5.9. Klasa rozszerza relację

Zwykle klasa abstrakcyjna służy do definiowania wzoru obiektu do utworzenia. W tym przykładzie klasa abstrakcyjna to kluczowe składniki batona. Co więcej, utworzyliśmy klasę dla batona czekoladowego, która może przechowywać wspólne informacje.

By umożliwić podklasie wzbogacenie funkcjonalności bez potrzeby pisania kodu, użyj słowa kluczowego `mixin`. W tym przykładzie połączenie klasy nadrzędnej z obiektem `mixin` pozwala na połączenie odrębnych funkcjonalności. Dzięki temu podklasa `CandyBar` jest obdarzona zachowaniem batona czekoladowego i odmianami Snickersa (`SnickersOriginal` i `SnickersCrisp`). Po utworzeniu podklasy możemy jej użyć w celu uzyskania dostępu do ogólnych składników batona.

Znajdziesz tu przykłady zastosowania języka Dart i frameworka Flutter w rozwiązywaniu problemów. Wisienką na torcie jest omówienie Firebase i Google Cloud!

Rob Edwards, Google Cloud UK&I

Tworzenie oprogramowania wymaga zarówno umiejętności, jak i wysiłku. Jeśli jednak zależy Ci na szybkich i satysfakcjonujących efektach, wypróbuj wieloplatformowy framework Flutter i język Dart. Obydwie technologie zapewniają bogaty zestaw narzędzi dla programistów i są świetnym punktem startowym do tworzenia pięknych aplikacji niewielkim nakładem pracy.

Ta książka będzie doskonałym uzupełnieniem wiedzy o Flutterze i Darcie, sprawdzi się również jako wsparcie podczas rozwiązywania konkretnych problemów. Znalazło się tu ponad sto receptur, dzięki którym poznasz tajniki pisania efektywnego kodu, korzystania z narzędzi udostępnianych przez framework Flutter czy posługiwania się rozwiązaniami dostawców usług chmurowych. Dowiesz się, jak należy pracować z bazami Firebase i platformą Google Cloud. Przy czym poszczególne receptury, poza rozwiązaniami problemów, zawierają również nieco szersze omówienia, co pozwoli Ci lepiej wykorzystać zalety Fluttera i Darta — spójnego rozwiązania do wydajnego budowania aplikacji!

Świetna książka dla każdego, kto chce się nauczyć języka Dart i frameworka Flutter!

Alex Moore, Google Cloud UK&I

Dzięki recepturom:

- poznasz zasady efektywnej pracy z Dartem
- nauczysz się korzystać z narzędzi Fluttera
- dowiesz się, jak integrować rozwiązania chmurowe z aplikacjami Fluttera
- rozwiążesz problemy związane z zarządzaniem danymi przez API
- rozpoczniesz pracę z bazami danych Firebase
- zaczniesz budować wieloplatformowe, efektywne i efektowne aplikacje

Richard Rose jest architektem w zespole Google Cloud. Słynie z zamiłowania do najnowszych technologii i rozwiązań chmurowych. Stale pracuje nad doskonaleniem swoich umiejętności programistycznych. Często bierze udział w konferencjach technicznych. Mieszka w Wielkiej Brytanii.

Helion
helion.pl
HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl
KOD KORZYŚCI
Sięgnij po więcej!
ISBN 978-83-289-0709-6
9 788328 907096
Cena: 69,00 zł