Pierwsze kroki z Dartem

Dlaczego więc język Dart? Cóż, Dart świetnie pasuje zarówno do aplikacji mobilnych, jak i aplikacji internetowych. Dart jest darmowy i open source, a repozytorium jest dostępne pod adresem https://github.com/dart-lang. Możesz również zapoznać się z językiem na oficjalnej stronie internetowej: https://www.dartlang.org/. Zaletą Darta jest to, że jest to język programowania zoptymalizowany pod kątem klienta dla aplikacji na wielu platformach, można go używać do tylu celów, co aplikacje komputerowe, mobilne, backendowe i internetowe. Kolejną zaletą jest możliwość transkompilacji do JavaScript, jeśli chcesz. W tym rozdziale wprowadzającym spróbujmy zrozumieć, dlaczego nauka języka Dart jest ważna przy tworzeniu aplikacji mobilnych o znaczeniu krytycznym na iOS i Androida. Jeśli masz już praktyczną wiedzę w języku zorientowanym obiektowo, takim jak Java lub Python, znacznie łatwiej będzie ci zrozumieć podstawowe koncepcje, ponieważ używając składni w stylu C, Dart jest językiem zdefiniowanym w klasach i zbierającym śmieci. Deweloperzy na całym świecie używają Darta do tworzenia wysokiej jakości aplikacji na iOS i Androida oraz do Internetu. Jest bogaty w funkcje, dzięki czemu możliwy jest również rozwój po stronie klienta. W miarę postępów w tej książce zobaczysz, jak poprawne jest to stwierdzenie. Jeśli chcesz dowiedzieć się, jak tworzyć natywne aplikacje mobilne na iOS i Androida oraz aplikacje internetowe za pomocą Darta, ta książka jest dobrym wprowadzeniem, ponieważ została zaprojektowana tak, aby dać Ci pełny obraz działania Darta. Chociaż tworzenie pełnej aplikacji mobilnej wykracza poza zakres tej książki, w rozdziale 9 zbudujesz prostą aplikację internetową.

Podstawowe cechy Darta

W przypadku małych operacji możesz skorzystać z internetowego edytora kodu pod adresem https://

dartpad.dartlang.org. Jednak do budowania pakietów i tworzenia projektów potrzebujesz edytora kodu, takiego jak Android Studio lub IntelliJ IDEA Community Edition. Visual Studio Code obsługuje również testowanie języka Dart. Zalecane jest jednak korzystanie z Android Studio lub IntelliJ IDEA Community Edition. Ułatwiają instalację wymaganych wtyczek; ponadto, jeśli chcesz budować aplikacje mobilne za pomocą Darta i Fluttera, te narzędzia są bardziej przydatne.

Uwaga : te edytory kodu są znane jako zintegrowane środowiska programistyczne (IDE). Posiadają wiele funkcji, które sprawiają, że pisanie kodu jest łatwe i wydajne. Innymi słowy, zostały zaprojektowane, aby ułatwić Ci życie związane z kodowaniem.

Po pierwsze, Dart jest niezwykle produktywny. Jeśli znasz już język programowania obiektowego, taki jak C++, C# lub Java, nauka języka Dart nie zajmie Ci więcej niż kilka dni. Jeśli jesteś absolutnie początkującym, to dobrze, że zaczynasz uczyć się Darta jako pierwszego języka programowania, ponieważ ma jasną i zwięzłą składnię. Posiada również bogate i potężne biblioteki podstawowe i obsługuje tysiące pakietów. Jako absolutny początkujący nie musisz się teraz martwić o biblioteki. Nauczysz się z nich korzystać w dalszej części książki, kiedy nadejdzie czas. Pod względem składni Dart ma podobieństwa z C, C#, Python, Java i JavaScript. Dart jest szybki i wściekły, a jego wydajność jest wysoka na urządzeniach mobilnych i w Internecie. Ponadto jego przenośność jest bardzo dobra. Kompiluje się do kodu ARM i x86, dzięki czemu aplikacje mobilne Dart mogą działać na iOS i Androidzie i nie tylko. Początkujący powinni zauważyć, że istnieje różnica między procesorami ARM i X86; procesory ARM są zgodne z architekturą RISC, podczas gdy procesory x86 są oparte na architekturze złożonego zestawu instrukcji (CISC). Ze względu na te cechy procesory x86 są uważane za szybsze niż procesory ARM. Ponadto w przypadku aplikacji internetowych Dart jest blisko związany z Flutterem, który jest implementowany przy użyciu kodu Dart.

Korzystanie z IDE dla Dart

Możesz użyć dowolnego dobrego IDE; jednak mój wybór to IntelliJ IDEA Community Edition lub Android Studio. Oba są bezpłatne i można je łatwo pobrać w systemach Windows, Linux i Mac. Jeśli chodzi o kod w tej książce, IntelliJ IDEA Community Edition jest najłatwiejszy w użyciu. Jest przeznaczony do ogólnego rozwoju, podczas gdy Android Studio jest przeznaczony do programowania mobilnego. Oznacza to, że łatwo jest uruchomić prostą aplikację Dart w IntelliJ, ale nie w Android Studio. W rzeczywistości Android Studio nawet nie ma takiej opcji. Pozwoli Ci to stworzyć tylko aplikację mobilną Flutter. Jeśli Twoim celem jest rozwój mobilny, polecam użyć IntelliJ, aby nauczyć się Dart z tej książki, a następnie przełączyć się na Android Studio dla swojej pierwszej aplikacji Flutter.

Wskazówka: Jednym z obejść tego podejścia jest utworzenie aplikacji Dart w IntelliJ IDEA Community Edition, a następnie otwarcie jej w Android Studio (z zainstalowaną wtyczką Dart). Android Studio uruchomi aplikację bez problemu; to przede wszystkim stworzenie takiego, które jest trudne. Zobaczysz, że przyjąłem to podejście w książce.

Istnieją dwie opcje dla obu IDE, ponieważ Android Studio jest w zasadzie dostosowana wersja IntelliJ.

• Zainstalowanie IDE w celu przetestowania kodu za pomocą Dart SDK w systemie lokalnym.

• Instalacja Fluttera i wtyczki Dart w dowolnym IDE. W takim przypadku nie potrzebujesz zestawu Dart SDK w swoim systemie operacyjnym.

Instalacja dowolnego środowiska IDE w systemie Windows jest stosunkowo łatwa. Pobierz plik .exe z oficjalnej strony internetowej i kliknij dwukrotnie, aby go uruchomić. To jest zalecany sposób. Możesz także pobrać plik ZIP i rozpakować go do plików programu. Znajdziesz folder bin, w którym możesz uruchomić odpowiednie pliki .exe. Zaleca się jednak pobranie pliku .exe z oficjalnej strony internetowej i uruchomienie go online. Instalowanie IDE na Macu nie jest skomplikowanym procesem. Musisz uruchomić plik DMG, a następnie przeciągnąć i upuścić aplikację do folderu Aplikacje. Po tym proces uruchamiania jest łatwy; kreator konfiguracji poprowadzi Cię przez resztę. Zalecam używanie Linuksa jako głównego systemu operacyjnego; Android jako platforma zawsze będzie działał lepiej na jądrze Linuksa i prawdopodobnie będziesz chciał używać Darta do programowania na Androida. Instalacja zestawu Dart SDK w systemie Linux jest również łatwa. Dlaczego potrzebujesz Dart SDK? Cóż, ma biblioteki i narzędzia wiersza poleceń potrzebne do tworzenia wszelkiego rodzaju aplikacji Dart — internetowych, wiersza poleceń lub aplikacji serwerowych. Aby tworzyć tylko aplikacje mobilne, nie potrzebujesz zestawu Dart SDK. Wtyczki Flutter w IDE będą działać. Aby zainstalować Dart w systemie Linux, najpierw otwórz terminal, a następnie możesz wydać następujące polecenia:

//code 1.1

```
sudo apt-get update

sudo apt-get install apt-transport-https

sudo sh -c 'curl https://dl-ssl.google.com/linux/linux_

signing_key.pub | apt-key add -'

sudo sh -c 'curl https://storage.googleapis.com/download.

dartlang.org/linux/debian/dart_stable.list > /etc/apt/sources.

list.d/dart_stable.list'
```

Następnie zainstaluj stabilną wersję Dart SDK.

//code 1.2

sudo apt-get update

sudo apt-get install dart

Następnie możesz sprawdzić swoją wersję Darta.

//code 1.3

$ dart –version

Dart VM version: 2.4.0 (Unknown timestamp) on "linux_x64"

Dart SDK zawiera katalog lib dla bibliotek Dart, których będziesz używać w IDE. Ponadto zestaw Dart SDK zawiera katalog bin, w którym znajdują się narzędzia wiersza polecenia. Pomaga uruchomić konsolę w twoim IDE, a jeśli chcesz, możesz również mieć wyjście terminala. W tym celu możesz przejść do folderu bin projektu i uruchomić plik main.dart.

Instalowanie IntelliJ IDEA Community Edition

Instalacja IntelliJ IDEA Community Edition jest łatwa. Możesz go zainstalować z Centrum oprogramowania Ubuntu. Otwórz Centrum oprogramowania i wpisz IntelliJ Community Edition. To się pojawi. Kliknij przycisk Install.

Możesz także zainstalować IntelliJ IDEA Community Edition za pośrednictwem wiersz poleceń na terminalu.

//code 1.4

sudo snap install intellij-idea-community –classic

Aplikacje w Centrum oprogramowania Ubuntu to pakiety snap; dlatego, jeśli masz już zainstalowane pakiety snapów na swoim komputerze, możesz zainstalować je za pośrednictwem terminala. Po zakończeniu podstawowej instalacji nie zapomnij zainstalować wtyczek Dart, z opcji Konfiguruj podczas uruchamiania lub z File ➤ Settings ➤ Plugins within the IDE

Wreszcie IntelliJ IDEA Community Edition jest gotowy do działania. Możesz zachować pliki Darta w folderze bin i uruchomić program, naciskając klawisze Shift+F10 lub wybierając polecenie Run ➤ Run z paska menu. Większość przykładowego kodu zostanie wydrukowana w konsoli na dole IDE.

Instalowanie Android Studio

Instalacja Android Studio na Linuksie jest dość prosta i przyjazna dla użytkownika, choć nie tak prosta jak instalacja IntelliJ IDEA Community Edition. Nie musisz wydawać żadnych instrukcji wiersza poleceń. Pobierz plik ZIP i rozpakuj go do /usr/local/ lub /opt/ dla współużytkowników. Teraz przejdź do katalogu /android-studio/bin/ i uruchom plik studio.sh za pomocą tego polecenia:

//code1.5

./studio.sh

Jeśli pojawi się prośba o zainstalowanie wymaganych bibliotek dla 64-bitowych maszyn z systemem Linux, zainstaluj je. Jeśli jesteś użytkownikiem Android Studio po raz pierwszy, możesz zaimportować poprzednie ustawienia Android Studio lub możesz to pominąć, klikając przycisk OK. Kreator Android Studio poprowadzi Cię przez proces konfiguracji; pamiętaj, że ta konfiguracja obejmuje pobranie komponentów Android SDK, które są wymagane do rozwoju; w opcji Konfiguruj po uruchomieniu IDE możesz zainstalować wtyczki Flutter i Dart (lub wybrać File ➤ Settings ➤ Plugins when the IDE is open).

Aby maksymalnie wykorzystać Android Studio na 64-bitowej maszynie z systemem Linux, powiedzmy Ubuntu, musisz zainstalować kilka 32-bitowych bibliotek z następującymi instrukcjami wiersza poleceń. Dostęp do tych bibliotek można uzyskać za pośrednictwem folderu lib projektu. Folder bin składa się z narzędzi wiersza poleceń, jak wspomniałem wcześniej.

//ode 1.6

sudo apt-get install libc6:i386 libncurses5:i386

libstdc++6:i386 lib32z1 libbz2-1.0:i386

Polecenie poprosi o hasło roota. Dla wersji 64-bitowej Fedora, polecenie jest inne.

//code 1.7

sudo yum install zlib.i686 ncurses-libs.i686 bzip2-libs.i686

Teraz jesteś gotowy do pracy w Android Studio.

Pisanie kodu Dart

Spójrzmy na nasz pierwszy kod Darta. Umieść następujący kod w pliku main.dart w swoim IDE. main() jest punktem wejścia, nie tylko dla Darta; jeśli budujesz aplikacje mobilne za pomocą Fluttera, przekonasz się, że we Flutterze jest to również punkt wejścia.

//code 1.8

main() {

print("Hello World!");

}

//output

Hello World!

W Android Studio lub IntelliJ możesz nacisnąć Shift + F10, aby uruchomić kod. Napiszmy trochę więcej kodu opartego na konsoli, aby poczuć Dart. Jednocześnie zobaczysz najbardziej podstawową składnię i sposób współdziałania poleceń.

//code 1.9

main() {

print("Hello World!");

//wywołanie funkcji

doSomething();

}

//zdefiniuj funkcję

doSomething(){

print("Do something!")

//wywołanie funkcji wewnątrz innej funkcji

```
lifeIsShort();

}

//definiowanie innej funkcji

lifeIsShort(){

print("Life is too short to do so many things.");

}
```

Zaczęliśmy nasz kod od funkcji najwyższego poziomu main(); jest to wymagane i ma szczególny charakter, ponieważ tak działa aplikacja. Tak więc wewnątrz funkcji main() wywołaliśmy funkcję doSomething(), która z kolei wywołuje funkcję lifeIsShort(). Każda funkcja wyświetla wyjście z print(); jest to wygodny sposób wyświetlania dowolnych danych wyjściowych. W naszym pierwszym programie omówiliśmy wiele rzeczy. Teraz uruchom kod (Shift+F10). Zobaczysz, że w naszym kodzie jest błąd. Jest to zamierzony błąd, aby zrozumieć, jak w Dart odbywa się debugowanie. Spójrz na wynik:

```
//wyjście z code 1.9

bin/main.dart:12:24: Error: Expected ';' after this.

print("Do something!")
```

Zapomnieliśmy umieścić średnik po wyświetleniu wyniku.

```
//code 1.10

//zdefiniuj funkcję

doSomething(){

print("Do something!");

//wywołanie funkcji wewnątrz innej funkcji

lifeIsShort();

}
```

Poprawmy to i uruchommy program ponownie.

```
// wyjście z  code 1.10

Hello World!

Do something!

Life is too short to do so many things.
```

Teraz jest ok. Nauczyłeś się wielu rzeczy dzięki temu pierwszemu kodowi; najważniejsze jest to, że możemy uczyć się na własnych błędach. Zawsze należy uważać na błędy składni. Brak średnika lub znaku dolara przed zmienną może zmienić grę. Widziałeś, jak komentujemy nasz kod za pomocą // znaków, jak pokazano tutaj:

```
//wywołanie funkcji wewnątrz innej funkcji

lifeIsShort();
```

Wszystko w tym wierszu jest ignorowane podczas działania programu. Postaraj się wnieść jak najwięcej komentarzy, aby wyjaśnić swój punkt widzenia, tak aby gdy inna osoba czytała Twój kod, zrozumiała go i zwizualizowała tak, jak ty wizualizowałeś swój kod podczas pisania. Osoba, która przeczyta twój kod w ciągu sześciu miesięcy, może być tobą, więc bądź miły dla swojego przyszłego ja. Jeśli jesteś kompletnym początkującym, możesz nie być w stanie zrozumieć tych wyjaśnień. Możesz czuć się zakłopotany słowami takimi jak funkcja, komentarz, wyjście itp. Dlatego kilka następnych sekcji jest przeznaczonych dla początkujących.

Zmienne, operatory, warunki warunkowe i przepływ sterowania

W tej sekcji omówimy kilka kluczowych koncepcji Darta, które są absolutnie niezbędne dla początkujących. Po pierwsze, podobnie jak Python, Dart jest obiektowym językiem programowania. Wszystko jest tu przedmiotem. Rozważmy liczbę całkowitą, na przykład 2. W naturze jest to liczba całkowita. W Dart wszystkie liczby całkowite są obiektami. Nawet funkcje i null są obiektami. Wiem, że termin obiekt może wprowadzić początkującego w konsternację. Programowanie obiektowe omówimy we właściwym czasie. Wcześniej dowiesz się, czym są zmienne, stałe i funkcje.

Uwaga: W skrócie null oznacza, że wartość danych nie istnieje. Ponieważ Dart traktuje każdy typ wartości jako obiekt, utworzono również klasę null, która używa tej instancji, w której nie podano żadnej wartości.

Odniesienia do przechowywania zmiennych

Zmienne przechowują odniesienia do obiektów. Innymi słowy, można powiedzieć, że zmienna to miejsce w pamięci lub pojemnik zawierający odniesienia do pewnych wartości. Jak wskazuje zmienna nazwy, odniesienie może ulec zmianie. Podobnie jak inne języki programowania, Dart ma kilka typów, takich jak liczby całkowite, łańcuchy znaków, wartości logiczne itp. Chociaż Dart jest językiem o silnej typografii, pozwala również na użycie typowania kaczego, co oznacza, że Dart może używać typu, o ile ten typ jest odpowiedni do tego celu („Jeśli coś chodzi jak kaczka i kwacze jak kaczka, to jest kaczką”). Typy danych to typy danych, które możemy reprezentować w języku programowania, np. liczba całkowita jest nieułamkową wartością liczbową, np. 1, 2 itd. Później, jeśli zajdzie taka potrzeba, możemy również manipulować tymi wartościami w naszym programie. Na przykład w kalkulatorze wykonujemy wiele operacji numerycznych, takich jak dodawanie, odejmowanie itp. Domyślna wartość większości typów danych to null. Musimy więc wspomnieć, jakiego typu danych będziemy używać. Zmiennych używamy do odwoływania się do typów, które faktycznie są przechowywane w pamięci. Rozważ następujące:

```
int a = 1;
```

Oznacza to, że najpierw przechowujemy w pamięci typ liczby całkowitej 1, a następnie przypisujemy tę wartość do zmiennej a. Znak równości (=) jest operatorem przypisania w Dart, więc przypisuje wartości zmiennym. Później wywołujemy a, aby pobrać 1 dla dowolnego rodzaju operacji matematycznych. W normalnych okolicznościach w Darcie wspominamy jakiego typu będziemy używać. Jeśli używamy liczb całkowitych i ciągów znaków, zapisujemy to w ten sposób:

```
//code 1.11

int myAge = 12;

String myName = "John Smith";
```

W poprzednich przykładach jawnie zadeklarowaliśmy typ, który powinien zostać użyty. W następnym przykładzie robimy to samo, ale w sposób dorozumiany. Dlatego możesz napisać ten sam kod również w ten sposób:

```
//code 1.12

var myAge = 12;

var myName = "John Smith";
```

Pytanie brzmi, czy wraz ze zmianą odniesienia zmienia się także typ? Proszę czytać dalej. W poprzednich fragmentach kodu zmienna myAge przechowuje wartość 12 i odwołuje się do niej jako obiektu typu całkowitego. W ten sam sposób zmienna myName przechowuje wartość John Smith i odwołuje się do niej jako do obiektu String. Zakłada się, że typ zmiennej myName jest specyficzny dla ciągu znaków, ale można go zmienić. Jeśli nie chcesz określonego lub ograniczonego typu, określ typ Obiektowy lub dynamiczny.

```
dynamic myName = "John Smith";
```

Jeśli nie zainicjujesz zmiennej, wartość domyślna zostanie ustawiona na null. Rozważ następujący kod:

```
int myNumber;int mójNumer;
```

Chociaż jest to liczba całkowita, nie jest inicjowana. Dlatego wartością domyślną jest Null. Uruchommy kod i przyjrzyjmy się wynikom.

```
//code 1.13

main() {

print("Hello World!");

int myNumber;

print(myNumber);

}
```

The output is as expected:

```
Hello World!

null
```

Porozmawiajmy o typach wbudowanych w Dart. Do tej pory widziałeś niektóre typy, takie jak liczba i ciąg. Innych nie widziałeś.

Wbudowane typy w Dart

Język Dart obsługuje szczególnie następujące typy i zawsze możesz zastosować wzorce o typie jednoznacznie określonym lub typu kaczego, aby je zainicjować:

- Liczby
- Ciągi
- Wartości logiczne
- Listy (znane również jako tablice)

• Zestawy

• Mapy

• Runy (do wyrażania znaków Unicode w ciągu znaków)

• Symbole

Możesz zainicjować obiekt dowolnego z tych specjalnych typów za pomocą literału. Na przykład Hello John Smith jest literałem łańcuchowym, a false jest literałem logicznym.

Rozważ ten kod:

```
//code 1.14

main() {

String saySomething = "Hello John Smith";

var isFalse = true;

if(saySomething == null){

print("It is ${isFalse}");

}else print("It is not ${isFalse}");

}
```

Ponieważ zmienna łańcuchowa nie ma wartości null, wynik powinien wyglądać następująco:

To nie prawda

Uwaga: używamy ${}, aby uwzględnić wartość wyrażenia w ciągu znaków. W tym przypadku jest to wartość zmiennej konwertowanej na ciąg znaków. Uwzględnianie w ten sposób wartości wyrażenia nazywa się interpolacją ciągu

Najczęściej spotkasz się z pierwszymi czterema wbudowanymi typami. Dowiesz się, jak używać innych typów wbudowanych, zgodnie z wymaganiami sytuacji.

Załóżmy, że nie lubisz zmiennych

Cóż, w niektórych przypadkach wartość musi być stała. Jeśli nie zamierzasz zmieniać wartości zmiennej, możesz zastosować dwie techniki.

• Możesz użyć const zamiast deklaracji typu var, String, int lub bool.

• Możesz także użyć wersji końcowej; pamiętaj jednak, że zmienną końcową można ustawić tylko raz.

Istnieje zatem różnica między tymi dwoma słowami kluczowymi: const i final. Powrócimy do tego tematu przy okazji omawiania programowania obiektowego. Należy pamiętać, że zmienna instancji może być ostateczna, ale nie stała. Rozważ ten kod:

```
//code 1.15

main() {

const firstName = "Sanjib";

final lastName = "Sinha";
```

```
String firstName = "John";

String lastName = "Sinha";

}
```

Spójrz na wynik pełen błędów:

```
//output

bin/main.dart:8:10: Error: 'firstName' is already declared in

this scope.

String firstName = "John";

^^^^^^^^^

bin/main.dart:5:9: Context: Previous declaration of 'firstName'.

const firstName = "Sanjib";

^^^^^^^^^

bin/main.dart:9:10: Error: 'lastName' is already declared in

this scope.

String lastName = "Sinha";

^^^^^^^^

bin/main.dart:6:9: Context: Previous declaration of 'lastName'.

final lastName = "Sinha";
```

Jeśli chcesz, aby zmienna była stałą czasu kompilacji, użyj const; użyj final dla zmiennej instancji, której nigdy nie zmienisz. W ramach krótkiego przeglądu najpierw sprawdzimy liczby. Następnie jeden po drugim dowiesz się o ciągach znaków, wartościach logicznych i innych typach. Liczby darta są dwojakiego rodzaju: liczby całkowite i dziesiętne. Zapisujesz je jako int i double. Liczby całkowite to liczby bez kropek dziesiętnych. Przykładami są 1, 2, 22 itd. W przypadku liczb podwójnych przecinek dziesiętny jest taki jak ten: 1,5, 3,723 itd.

Zabawa z liczbami i double

Zarówno typy int, jak i double są podtypami num. Typ num obejmuje podstawowe operatory, takie jak +, -, / i *; i reprezentują odpowiednio plus, minus, dzielenie i mnożenie. Można je nazwać operatorami arytmetycznymi. Istnieje również modulo, czyli reszta, a znakiem jest %. Zobaczmy kilka interesujących przykładów:

```
//code 1.16

main() {

var one = int.parse('1');

print(one);

if(one.isOdd){
```

```
print("It is an odd number.");
} else print("It is an even number.");
}
```

We have converted a string into an integer, or number.

```
//output
1
```

Jest to liczba nieparzysta.

Możemy również zamienić ciąg znaków na liczbę podwójną. Zmieńmy trochę poprzedni kod.

```
//code 1.17
main() {
var one = int.parse('1');
var doubleToString = double.parse('23.564');
print(one);
print(doubleToString);
if(one.isOdd && doubleToString.isFinite){
print("The first number is an odd number and the second one
is a double ${doubleToString} and a finite number.");
} else print("It is an even number and the second one is not
a double ${doubleToString} and a non-finite number.");
}
```

Wynik jest całkiem oczekiwany. Obydwa stwierdzenia są prawdziwe, więc operacja relacyjna daje następujący wynik:

```
//wyjście
1
23,564
```

Pierwsza liczba jest liczbą nieparzystą, a druga liczbą podwójną

23,564 i liczba skończona. Możemy też zrobić odwrotnie. Zamierzamy tutaj zamienić liczbę całkowitą na ciąg znaków:

```
//code 1.18
main() {
int myNUmber = 542;
double myDouble = 3.42;
```

```
String numberToString = myNUmber.toString();

String doubleToString = myDouble.toString();

if ((numberToString == '542' && myNUmber.isEven) &&

(doubleToString == '3.42' && myDouble.isFinite)){

print("Both have been converted from an even number

${myNUmber} and a finite double ${myDouble} to string. ");

} else print("Number and double have not been converted to

string.");

}

//output

Obie zostały przekonwertowane z liczby parzystej 542 i skończonej
```

double 3,42 do ciągu znaków. W miarę postępów odkryjemy, że Dart jest niezwykle elastycznym językiem, a składnia jest łatwa do zapamiętania dzięki dużej pomocy podstawowych bibliotek.

Zrozumienie ciągów

Ciąg Dart to sekwencja jednostek kodu UTF-16. Dla absolutnie początkujących opiszę krótko UTF-8, UTF-16 i UTF-32. Wszystkie przechowują Unicode, ale używają różnych bajtów. Spróbujmy najpierw zrozumieć zalety używania kodu UTF-16 w porównaniu z dwoma pozostałymi. Dowiedzmy się o UTF-8. W miejscach, gdzie znaki ASCII stanowią większość tekstu, zaletą jest UTF-8. ASCII jest przeznaczony tylko dla języka angielskiego, ponieważ powstał w Stanach Zjednoczonych. Później rozprzestrzeniło się na cały świat, a inne kraje chciały, aby sznurki pracowały nad ich językami. Podobnie jak ASCII, UTF-8 koduje wszystkie znaki na 8 bitach. Tam, gdzie nie dominuje kod ASCII (w kulturach, w których nie dominuje język angielski), przewagę ma UTF-16. Użycie 2 bajtów (16 bitów) umożliwia zakodowanie 65 536 różnych wartości. Jeśli poważnie myślisz o zrozumieniu kodowania i zestawów znaków, odwiedź ten link:

http://kunststube.net/encoding/

Dla większości znaków UTF-16 pozostaje tylko 2 bajty. Jednak UTF-32 próbuje objąć wszystkie możliwe znaki w 4 bajtach, co oznacza, że procesory mają dodatkowe obciążenie, przez co UTF-32 jest dość rozdęty. Mówiąc najprościej, chodzi o obsługę jak największej liczby języków. Obsługa Unicode sprawia, że Dart jest potężniejszy, a aplikacje mobilne i internetowe możesz tworzyć w dowolnym języku. Zobaczmy tutaj jeden przykład. Wypróbowałem trochę pisma bengalskiego.

```
//code 1.19

main(List<String> arguments) {

String bengaliString = "বাংলা লেখা";

String englishString = "This is some English text.";

print("Here is some Bengali script - ${bengaliString} and

some English script ${englishString}");
```

}

Oto dane wyjściowe:

//output

Oto trochę pisma bengalskiego - বাংলা লেখা i trochę pisma angielskiego. To jest tekst w języku angielskim. Obsługując ciągi znaków, należy pamiętać o kilku rzeczach. Można używać zarówno cudzysłowów pojedynczych („”), jak i cudzysłowów podwójnych („”).

//code 1.20

main(List<String> arguments) {

String stringWithSingleQuote = 'I\'m a single quote';

String stringWithDoubleQuote = "I'm a double quote.";

print("Using delimiter in single quote -

${stringWithSingleQuote} and using delimiter in double

quote - ${stringWithDoubleQuote}");

}

W obu przypadkach możesz użyć ogranicznika, ale w takich przypadkach podwójny cudzysłów jest bardziej pomocny. Spójrz na dane wyjściowe:

//output of code 1.21

Użycie ogranicznika w pojedynczym cudzysłowie – jestem pojedynczym cudzysłowem i użycie ogranicznika w cudzysłowie podwójnym – jestem podwójnym cudzysłowem Umieściliśmy wartość wyrażenia w ciągu znaków, używając naszej zmiennej w ten sposób: ${stringWithSingleQuote} . Jak zauważono wcześniej w tym rozdziale, nazywa się to interpolacją ciągów. Jeśli chcesz po prostu wyrazić wartość zmiennej, nie musisz używać {}. Możesz użyć zmiennej w ten sposób:

print("$stringWithSingleQuote");

print(stringWithSingleQuote);

Łączenie ciągów, a nawet tworzenie wielu linii jest w Dart całkiem łatwe.

Rozważ ten kod:

//code 1.22

main(List<String> arguments) {

String stringInterpolation = 'string ' + 'concatenation';

print(stringInterpolation);

String multiLIneString = """

This is

a multi line

```
string.
""";
print(multiLlneString);
}
```

Do konkatenacji używasz operatora +, co oznacza, że łączysz ze sobą dwa ciągi znaków. Patrząc na wynik, użyliśmy potrójnego cudzysłowu z pojedynczym lub podwójnym cudzysłowem:

```
//output
łączenie ciągów
To jest
linia wieloliniowa
strunowy.
```

Jeśli chcesz przechowywać pewną stałą wartość w stałym ciągu znaków, metoda wartość nie może być zmienną. Rozważ ten kod:

```
//code 1.23
main(List<String> arguments) {
const aConstantInteger = 12;
const aConstantBoolean = true;
const aConstantString = "I am a constant string.";
const aValidConstantString = "this is a constant integer:
${aConstantInteger}, a constant boolean: ${aConstantBoolean},
a constant string: ${aConstantString}";
print("This is a valid constant string and the output is:
$aValidConstantString");
}
```

Utworzyliśmy prawidłowy ciąg stały, przechowując stałą wartość wewnątrz nich. Wyjście jest całkowicie OK.

```
//output
```

Jest to prawidłowy ciąg stały, a wynik jest następujący: jest to stała liczba całkowita: 12, stała wartość logiczna: prawda, ciąg stały: Jestem ciągiem stałym. To nie zadziała, jeśli chcesz przechowywać zmienne dane w stałym ciągu. Zmieniliśmy poprzednią listę kodów na następującą:

```
//code 1.24
main(List<String> arguments) {
var aConstantInteger = 12;
```

```
var aConstantBoolean = true;

var aConstantString = "I am a constant string.";

const aValidConstantString = "this is a constant integer:

${aConstantInteger}, a constant boolean: ${aConstantBoolean},

a constant string: ${aConstantString}";

print("This is a valid constant string and the output is:

$aValidConstantString");

}
```

Oto wynik, który jest pełen błędów:

```
//output

bin/main.dart:9:63: Error: Not a constant expression.

const aValidConstantString = "this is a constant integer:

${aConstantInteger}, a constant boolean: ${aConstantBoolean},

a constant string: ${aConstantString}";

^^^^^^^^^^^^^^^

bin/main.dart:9:104: Error: Not a constant expression.

const aValidConstantString = "this is a constant integer:

${aConstantInteger}, a constant boolean: ${aConstantBoolean},

a constant string: ${aConstantString}";

^^^^^^^^^^^^^^^

bin/main.dart:9:144: Error: Not a constant expression.

const aValidConstantString = "this is a constant integer:

${aConstantInteger}, a constant boolean: ${aConstantBoolean},

a constant string: ${aConstantString}";
```

Nie działało. W miarę postępów dowiesz się więcej o ciągach. Ich zrozumienie jest ważne w kontekście tworzenia aplikacji mobilnych i webowych. W następnej sekcji dowiesz się o wartościach logicznych, które również odgrywają istotną rolę w budowaniu algorytmów.

Być prawdą czy być fałszywym

Widziałeś już, że Dart ma typ zwany bool. Literały logiczne prawda i fałsz mają typ bool. Są to stałe czasu kompilacji. Jest to niezwykle ważna koncepcja w informatyce, ponieważ można używać struktur kontrolnych do zmiany przebiegu programu w zależności od tego, czy stwierdzenie jest prawdziwe, czy fałszywe.

Wprowadzenie do kolekcji: tablice są listami w Dart

Tablica, czyli uporządkowana grupa obiektów, jest najpowszechniejszą kolekcją w każdym języku programowania. W Dart tablice są obiektami typu List. W naszych przyszłych dyskusjach będziemy je nazywać listami. Dart został zaprojektowany tak, aby kompilował się do JavaScript i działał w nowoczesnej sieci; dlatego jeśli masz praktyczną wiedzę na temat JavaScript, znajdziesz pewne podobieństwa w tego typu zbiorach. Oto przykładowy kod do rozważenia, abyś mógł zrozumieć, dlaczego ta koncepcja jest ważna:

```
//code 1.25

main(List<String> arguments) {

List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];

print(fruitCollection[0]);

}
```

Rozważ inny fragment kodu:

```
//code 2.15

main(List<String> arguments) {

List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];

var myIntegers = [1, 2, 3];

print(myIntegers[2]);

print(fruitCollection[0]);

}
```

Jaka jest różnica między tymi dwoma fragmentami kodu? W kodzie 2.14 wyraźnie wspomnieliśmy, że zadeklarujemy kolekcję owoców. I za pomocą klucza możemy wybrać dowolny przedmiot z tej kolekcji. W tablicy klucz nie jest wymieniony w definicji; automatycznie wnioskuje się, że klucz zaczyna się od 0. Dlatego wyjściem kodu 2.14 jest Mango. W drugim przypadku nie mamy jednoznacznej deklaracji co do typu listy myIntegers. Napisaliśmy to:

```
var myIntegers = [1, 2, 3];
```

Jednak Dart wnioskuje, że lista ma typ List<int>. Zobaczmy

wyjście kodu 2.15:

```
//output

3

Mango
```

Co się stanie, jeśli spróbujemy wstrzyknąć obiekty niebędące liczbami całkowitymi do listy myInteger?

```
//code 2.17

main(List<String> arguments) {

List fruitCollection = ['Mango', 'Apple', 'Jack fruit'];
```

```
var myIntegers = [1, 2, 3, 'non-integer object'];

print(myIntegers[3]);

print(fruitCollection[0]);

}
```

Nie spowodowało to żadnego błędu. Zobacz dane wyjściowe pokazane tutaj:

```
//output of code 2.17

non-integer object

Mango
```

Pamiętaj jednak, że listy Dart korzystają z indeksowania od zera, podobnie jak wszystkie inne kolekcje, które mogłeś widzieć w innych językach programowania. Pomyśl o liście jak o parze klucz-wartość, gdzie 0 jest indeksem pierwszej wartości lub elementu. W miarę postępów będziemy omawiać listy, ponieważ istnieją inne przydatne metody, które wykorzystamy podczas tworzenia naszej pierwszej aplikacji mobilnej. Listy dart mają wiele przydatnych metod.

Zdobądź, ustaw, idź

W Dart zestaw to nieuporządkowana kolekcja unikalnych przedmiotów. Istnieją niewielkie różnice w składni pomiędzy listami i zestawami. Przyjrzyjmy się najpierw przykładowi, aby dowiedzieć się więcej o różnicach.

```
//code 1.26

main(List<String> arguments) {

var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};

print(fruitCollection.lookup('Apple'));

}

//output

Apple
```

Zbiór możemy przeszukać za pomocą metody lookup(). Jeśli szukamy czegoś innego, zwraca wartość null.

```
//code 1.27

main(List<String> arguments) {

var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};

print(fruitCollection.lookup('Something Else'));

}

//output

null
```

Kiedy piszemy co następuje, nie tworzy to zbioru, ale mapę:

```
var myInteger = {};
```

Składnia literałów mapy jest podobna do składni literałów zestawu. Dlaczego to? Ponieważ literały mapy były na pierwszym miejscu. Dosłowny {} jest domyślnym typem mapy. Możemy to udowodnić za pomocą prostego testu:

```
//code 1.28

main(List<String> arguments) {

var myInteger = {};

if(myInteger.isEmpty){

print("It is a map that has no key, value pair.");

} else print("It is a set that has no key, value pair.");

}
```

Spójrz na dane wyjściowe:

```
//output of code 2.20
```

To jest mapa, która nie ma pary klucz-wartość. Oznacza to, że mapa jest pusta. Gdyby to był zestaw, otrzymalibyśmy wynik w tym kierunku. Wiele przykładów zestawów zobaczymy w przyszłości podczas tworzenia naszej aplikacji mobilnej. Na razie pamiętaj tylko, że ogólnie mapa to obiekt, który kojarzy klucze z wartościami. W zestawie znajdują się również klucze, ale są one ukryte. W przypadku zbiorów nazywamy je indeksami. Zobaczmy jeden przykład typu Map, mapując literały. Podczas zapisywania kluczy i wartości należy pamiętać, że każdy klucz występuje tylko raz, ale tej samej wartości można używać wiele razy.

```
//code 1.29

main(List<String> arguments) {

var myProducts = {

'first' : 'TV',

'second' : 'Refrigerator',

'third' : 'Mobile',

'fourth' : 'Tablet',

'fifth' : 'Computer'

};

print(myProducts['third']);

}
```

The output is obvious, as shown here:

```
'Mobile'
```

Dart rozumie, że myProducts ma typ Map<String, String>(Map<Key, Value>); moglibyśmy utworzyć kluczowe liczby całkowite lub liczby, zamiast typu ciągu.

```
//code 1.30

main(List<String> arguments) {

var myProducts = {

1 : 'TV',

2 : 'Refrigerator',

3 : 'Mobile',

4 : 'Tablet',

5 : 'Computer'

};

print(myProducts[3]);

}
```

Dane wyjściowe są takie same jak poprzednio: mobilne. Czy możemy dodać kolekcję wartości typu Set wewnątrz mapy? Tak możemy.

Rozważ ten kod:

```
//code 1.31

main(List<String> arguments) {

Set mySet = {1, 2, 3};

var myProducts = {

1 : 'TV',

2 : 'Refrigerator',

3 : mySet.lookup(2),

4 : 'Tablet',

5 : 'Computer'

};

print(myProducts[3]);

}
```

W poprzednim kodzie wstrzyknęliśmy kolekcję typu Set, a także sprawdziliśmy wartość definiującą za pomocą klucza Map. Tutaj, wewnątrz pary klucz-wartość Map, dodaliśmy element zestawu nr 2 w następujący sposób: 3: mójSet.lookup(2). Później mówimy naszemu edytorowi Android Studio, aby wyświetlił plik wartość typu mapy myProducts.

Wynik jest całkiem oczekiwany: 2.

Możesz utworzyć tę samą listę produktów, używając konstruktora Map. Początkującym termin „konstruktor” może wydawać się trudny. Omówimy ten termin szczegółowo w rozdziale 7. Rozważmy następujący kod:

```
//code 1.32

main(List<String> arguments) {

var myProducts = Map();

myProducts['first'] ='TV';

myProducts['second'] ='Mobile';

myProducts['third'] ='Refrigerator';

if(myProducts.containsValue('Mobile')){

print("Our products list has ${myProducts['second']}");

}

}
```

Oto dane wyjściowe:

```
//output
```

Na naszej liście produktów znajdują się urządzenia mobilne ponieważ w kodzie 1.32 mamy instancję klasy Map, doświadczony programista mógł spodziewać się nowej Map() zamiast samej Map(). Począwszy od Dart 2, nowe słowo kluczowe jest opcjonalne. Dowiesz się o tym szczegółowo w Rozdziale 7. O kolekcjach dowiesz się także więcej w Rozdziale 7, gdzie dowiesz się więcej o Liście, Zbiorze i Mapie.

Operatory są przydatne

Mówiąc najprościej, programowanie polega na przetwarzaniu zmiennych. Przetwarzanie to może na przykład polegać na wykonaniu obliczeń matematycznych lub połączeniu dwóch ciągów znaków. W tym celu potrzebujemy operatorów. Prostota Darta polega na tym, że operator + dodaje dwa operandy całkowite (zmienne) i daje wynik. Jednocześnie możemy użyć operatora + do połączenia dwóch ciągów znaków (jak pokazano w kodzie 1.22). W Dart, gdy używasz operatorów, tak naprawdę tworzysz wyrażenia. Oto kilka przykładów wyrażeń: a++, a + b, a ∗ b, a/b, a~/b, a%b i tak dalej. W Dart istnieje wiele typów operatorów. Nawet zupełni nowicjusze prawdopodobnie słyszeli o operatorach arytmetycznych. Operatory relacyjne są niezwykle przydatne w strukturach kontrolnych. Przyjrzymy się im jeden po drugim. Typowe operatory arytmetyczne to - dodawanie (+), odejmowanie (-), mnożenie (∗), dzielenie (/) i modulo lub reszta (%); operator specjalny, dzielenie, zwracanie liczby całkowitej wygląda następująco: ~/. Zobaczmy jeden przykład:

```
//code 1.33

main(List<String> arguments) {

int aNum = 12;

double aDouble = 2.25;

var theResult = aNum ~/ aDouble;
```

```
print(theResult);
}
//output
5
```

Zauważ, że ten operator specjalny wyświetlił liczbę całkowitą, a nie liczbę podwójną. Gdybyśmy jednak podzielili to w prosty sposób, wyglądałoby to tak:

```
//code 1.34
main(List<String> arguments) {
int aNum = 12;
double aDouble = 2.25;
var theResult = aNum / aDouble;
print(theResult);
}
```

Oto dane wyjściowe:

```
//wyjście kodu 2.26
5.33333333333333
```

Jedną z kluczowych cech Darta jest to, że obsługuje zarówno operatory zwiększania i zmniejszania prefiksu, jak i postfiksu. Tutaj przedrostek oznacza ++zmienną lub --zmienną. Dodają one odpowiednio 1 lub odejmuje 1 od wartości zmiennej. Postfiks robi to samo; zmienia się tylko składnia, na przykład: zmienna++ lub zmienna--. Zobaczmy przykład:

```
//code 1.35
main(List<String> arguments) {
int aNum = 12;
aNum++;
++aNum;
int anotherNum = aNum + 1;
print(anotherNum);
}
```

Wynik jest zgodny z oczekiwaniami: 15. Zarówno prefiks, jak i postfiks działają w przypadku -- również.

Operatory relacyjne

Operatory relacyjne nazywane są również operatorami równości, ponieważ == oznacza „równy”, a inne operatory relacyjne zwykle sprawdzają równość w różnych formach. Rozważmy kilka fragmentów kodu, które jednym rzutem oka pokażą nam wiele typów operatorów relacyjnych.

```
//code 1.36

main(List<String> arguments) {

int firstNum = 40;

int secondNum = 41;

if (firstNum != secondNum){

print("$firstNum is not equal to the $secondNum");

} else print("$firstNum is equal to the $secondNum");

}

//output

40 is not equal to the 41
```

W poprzednim kodzie operator != oznacza „nie równy”. Okazuje się, że operandy nie są równe. Mówimy więc: „Jeśli pierwszyNum nie jest równy drugiemuNum, wykonaj kod pomiędzy {}. W przeciwnym razie wykonaj kod po else.” Zmieńmy trochę ten kod:

```
//code 1.37

main(List<String> arguments) {

int firstNum = 40;

int secondNum = 40;

if (firstNum == secondNum){

print("$firstNum is equal to the $secondNum");

} else print("$firstNum is not equal to the $secondNum");

}
```

Tutaj mówimy: „Jeśli pierwszyNum jest równy drugiemuNum, wykonaj kod pomiędzy {}. W przeciwnym razie wykonaj kod po else.” Dodajmy trochę więcej logiki do naszego kodu, jak pokazano tutaj:

```
//code 1.38

main(List<String> arguments) {

int firstNum = 40;

int secondNum = 40;

int thirdNum = 74;

int fourthNum = 56;

if (firstNum == secondNum || thirdNum == fourthNum){

print("If choice between 'true' or 'false', the 'true' gets

the precedence.");
```

```
} else print("If choice between 'true' or 'false', the

'false' gets the precedence.");

}

//output
```

W przypadku wyboru pomiędzy „prawdą” a „fałszem” pierwszeństwo ma „prawda”. Tym razem mówimy: „Jeśli pierwszyNum jest równy drugiemu numerowi LUB jeśli trzeciNum jest równy czwartemu, wykonaj kod pomiędzy {}. W przeciwnym razie wykonaj kod po else.” Aby zaimplementować tę logikę, używamy operatora OR (||). Zatem jeśli jedna strona operatora OR jest prawdziwa, całe stwierdzenie jest prawdziwe. Nie dotyczy to operatora relacyjnego AND (&&). Spójrz na ten kod:

```
//code 1.39

main(List<String> arguments) {

int firstNum = 40;

int secondNum = 40;

int thirdNum = 74;

int fourthNum = 56;

if (firstNum == secondNum && thirdNum == fourthNum){

print("If choice between 'true' or 'false', in this case

the 'true' gets the precedence.");

} else print("If choice between 'true' or 'false', in this

case the 'false' gets the precedence.");

}

//output
```

W przypadku wyboru pomiędzy „prawdą” a „fałszem” w tym przypadku „fałsz” maleje. Użyliśmy operatora relacyjnego && i tutaj wyrażenie jest fałszywe, ponieważ w przypadku operatora AND obie strony muszą być prawdziwe. ! znak ma wiele ról. Rozważ ten fragment kodu:

```
//code 1.40

main(List<String> arguments) {

int aNUmber = 35;

if(!(aNUmber != 150) && aNUmber <= 150){

print("It's true");

} else print("It's false.");

}
```

Czy możesz zgadnąć, jaki byłby wynik? Pierwsze stwierdzenie jest fałszywe, ponieważ zanegowaliśmy stwierdzenie prawdziwe za pomocą ! podpisać.

!(aNUmber != 150)

Drugie stwierdzenie jest prawdziwe; wartość jest mniejsza lub równa 150.

aNUmber <= 150

Ponieważ operatorem logicznym jest tutaj AND (&&), całe wyrażenie będzie fałszywe.

!(!(aNUmber != 150) && aNUmber <= 150

Gdybyśmy użyli operatora logicznego OR (||), wynik zostałby otrzymany

się jako prawdziwe. Dla przypomnienia, operator >= oznacza większy lub równy. To jest > dla większego niż lub < dla mniejszego niż. Poświęć trochę czasu na zabawę z operatorami logicznymi lub relacyjnymi, ponieważ jest to jeden z głównych filarów informatyki.

Wpisz operatory testów

Jak, jest i jest! operatory są przydatne do sprawdzania typów w czasie wykonywania. Rozważ ten kod:

```
//code 1.41

main(List<String> arguments) {

int myNumber = 13;

bool isTrue = true;

print(myNumber is int);

print(myNumber is! int);

print(myNumber is! bool);

print(myNumber is bool);

}
```

Pierwsza jest prawdziwa, druga fałszywa i tak dalej.

```
//output

true

false

true

false
```

Operatory przypisania

Przypisując wartość używamy operatora =. Co się stanie, gdy zmienna przypisana do ma wartość null? Używamy specjalnego typu operatora: - ??=.

Rozważ ten kod:

```
//code 1.42

main(List<String> arguments) {

int firstNum = 10;

int secondNum;

if(firstNum == 10) print("The value of ${firstNum} is set.");

if (secondNum == null) print("It is true.");

secondNum ??= firstNum;

print(secondNum);

}
```

Teraz spójrz na wynik:

```
//output

Ustawiona jest wartość 10.

To True.

10
```

W kodzie 1.42 przypisaliśmy wartość FirstNum do 10, a typem jest liczba całkowita. Można więc powiedzieć, że wartość FirstNum jest ustawiona. Jednocześnie nie przypisaliśmy żadnej wartości do secondNum, więc domyślnie jest to null. Następnie przypisujemy liczbę całkowitą do zmiennej, która zawiera wartość null, za pomocą tego specjalnego operatora: ??=. Prawie to samo dzieje się w przypadku złożonych operatorów przypisania. Teraz napiszemy poprzedni kod w następujący sposób:

```
//code 1.43

main(List<String> arguments) {

int firstNum = 10;

int secondNum;

if(firstNum == 10) print("The value of ${firstNum} is set.");

if (secondNum == null) print("It is true.");

secondNum ??= firstNum;

print(secondNum);

print("After using an assignment operator, the value changes.");

secondNum += secondNum;

print(secondNum);

print("After using an assignment operator, the value changes
```

again.");

secondNum -= secondNum;

print(secondNum);

if (secondNum == null) print("It is true.");

else print("it is false, because the 'secondNUm' has the

value of ${secondNum} now.");

}

Spójrz na ten wynik, gdzie jest oczywiste, że zmieniliśmy wartość secondNum po kolei:

//output

Ustawiona jest wartość 10.

To True.

10

Po użyciu operatora przypisania wartość ulega zmianie.

20

Po użyciu operatora przypisania wartość zmienia się ponownie.

0

jest to fałsz, ponieważ „drugiNUM” ma teraz wartość 0.

W miarę postępów będziesz widzieć więcej przykładów operatorów.

Streszczenie

Liczby, ciągi znaków i wartości logiczne — w Dart wszystkie są literałami. Rozważmy te literały: 1, 2.3, „Niektóre ciągi”, prawda, fałsz. Musimy pamiętać o kilku rzeczach, np.:

var isValid = true;

• var to typ danych.

• isValid to nazwa zmiennej (lub miejsce w pamięci).

• prawda jest dosłownym.

Możesz podać typ danych zmiennej jako int, double, String lub bool. Jeśli tego nie zrobisz, możesz po prostu nazwać je var. W takim przypadku, jeśli nie jest to wspomniane, typ danych jest wywnioskowany. Interpolacja ciągów jest dobrą praktyką. Nie używaj znaku +, aby dodać dwa ciągi. Użyj wyrażenia dla operatorów, takiego jak ${number1 + number2}. Jaki będzie Twój wybór? końcowy czy stały? To trudny wybór. Musisz pamiętać o kilku rzeczach: kiedy wybierzesz final, zostanie on zainicjowany, a kiedy uzyskasz do niego dostęp, przydzielona zostanie mu pamięć. Stała jest domyślnie ostateczna; oznacza to, że podczas kompilacji jest inicjowany, a plikowi przydzielana jest na to pamięć.

Kontrola przepływu i pętla

Kontrolowanie przepływu kodu jest ważne. Programiści chcą kontrolować logikę swojego kodu z wielu powodów; Jednym z głównych powodów jest to, że użytkownik oprogramowania powinien mieć do dyspozycji wiele opcji. Możesz jednak nie znać wcześniej warunków, w jaki sposób powinna się poruszać logika programowania. Możesz się tylko domyślać, więc jako programista powinieneś otworzyć przed użytkownikiem jak najwięcej możliwości. Istnieje kilka technik, które można zastosować w celu kontrolowania przepływu kodu. Popularna jest na przykład logika if-else.

if – else

Spójrzmy na prosty przykład kontrolowania przepływu kodu. Następnie zagłębimy się w logiczne konsekwencje tego podejścia. Po if może nastąpić else, jeśli instrukcja logiczna testowana przez blok if okaże się fałszywa.

```
if to prawda){

Program się wykonuje

}

else {

Blok ten nie zostanie wówczas wykonany

}
```

Zupełnie odwrotnie dzieje się, gdy wyrażenie sprawdzane przez if jest fałszywe.

```
if(to jest fałszywe){

Program nie zostanie wykonany

}

else {

Następnie ten blok zostanie wykonany

}
```

W programowaniu ten mechanizm testowania zależy od różnych relacji. W poprzednim rozdziale widziałeś niektóre z nich. Więcej zobaczysz tutaj.

```
//code 2.1main(List<String> arguments) {

bool firstButtonTouch = true;

bool secondButtonTouch = false;

bool thirdButtonTouch = true;

bool fourthButtonTouch = false;

if(firstButtonTouch) print("The giant starts running.");

else print("To stop the giant please touch the second button.");

if(secondButtonTouch) print("The giant stops.");

else print("You have not touched the second button.");
```

```
print("Touch any button to start the game.");
if(thirdButtonTouch) print("The giant goes to sleep.");
else print("You have not touched any button.");
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}
//wyjście  code 21
The giant starts running.
You have not touched the second button.
Touch any button to start the game.
The giant goes to sleep.
You have not touched any button.
```

Teraz możesz uczynić ten mały fragment kodu bardziej skomplikowanym, jak pokazano tutaj:

```
//kod 2.2
main(List<String> arguments) {
bool firstButtonTouch = true;
var firstButtonUntouch;
bool secondButtonTouch = false;
bool thirdButtonTouch = true;
bool fourthButtonTouch = false;
firstButtonUntouch ??= firstButtonTouch;
firstButtonUntouch = false;
if (firstButtonUntouch == false || firstButtonTouch == true)
print("The giant is sleeping.");
else print("You need to wake up the giant. Touch the first
button.");
if(firstButtonTouch == true && firstButtonUntouch == false)
print("The giant starts running.");
print("To stop the giant please touch the second button.");
if((secondButtonTouch == true && thirdButtonTouch == true)
|| fourthButtonTouch == false) print("The giant stops.");
```

```
else print("You have not touched the second button.");
print("Touch any button to start the game.");
if(thirdButtonTouch) print("The giant goes to sleep.");
else print("You have not touched any button.");
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}
```

Dane wyjściowe będą się różnić, jak pokazano tutaj:

```
//wyjście kodu 2.2
The giant is sleeping.
The giant starts running.
To stop the giant please touch the second button.
The giant stops.
Touch any button to start the game.
The giant goes to sleep.
You have not touched any button.
```

W przypadku logiki if-else zawsze pamiętaj o następujących złotych zasadach. Są to warunki AND:

```
statementOne = TRUE;
statementTwo = TRUE;
if(statementOne and statementTwo){
the statement will execute, as it stands for TRUE
}
```

2. Jeśli oba warunki są fałszywe, wynik jest fałszywy.

```
statementOne = FALSE;
statementTwo = FALSE;
if(statementOne and statementTwo){
the statement will not execute, as it stands for FALSE
}
```

3. Jeśli jeden warunek jest prawdziwy, a drugi fałszywy, wynik jest fałszywy.

```
statementOne = TRUE;
statementTwo = FALSE;
```

```
if(statementOne and statementTwo){
the statement will not execute, as it stands for FALSE
}
```

Oto zasady dotyczące warunku LUB:

1. Jeśli jeden warunek jest prawdziwy lub jeden warunek jest fałszywy, wynik jest prawdziwy.

```
statementOne = TRUE;
statementTwo = FALSE;
if(statementOne or statementTwo){
the statement will execute, as it stands for TRUE
}
```

2. Jeśli oba warunki są fałszywe, wynik jest fałszywy.

```
statementOne = FALSE;
statementTwo = FALSE;
if(statementOne or statementTwo){
the statement will not execute, as it stands for
FALSE
}
```

Powinieneś teraz mieć pojęcie, jak możesz użyć logiki if-else, kiedy jej potrzebujesz. Może się to stać skomplikowane, gdy zaczniesz dodawać operatory relacyjne. Na koniec, zanim opuszczę tę sekcję, pokażę Ci kolejny fragment kodu, w którym zmieniono istniejący zestaw reguł lub zasad. Zmiana kolejności logiki AND i OR da ci wyobrażenie o tym, jak może zmienić się wynik.

```
//code 2.3
main(List<String> arguments) {
bool firstButtonTouch = true;
var firstButtonUntouch;
bool secondButtonTouch = false;
bool thirdButtonTouch = true;
bool fourthButtonTouch = false;
firstButtonUntouch ??= firstButtonTouch;
firstButtonUntouch = false;
if (firstButtonUntouch == false || firstButtonTouch == true)
print("The giant is sleeping.");
```

```
else if (thirdButtonTouch) print("You need to wake up the
giant. Touch the first button.");
else if(firstButtonTouch == true && firstButtonUntouch ==
false) print("The giant starts running.");
else if (secondButtonTouch) print("To stop the giant please
touch the second button.");
else if((secondButtonTouch == true && thirdButtonTouch
== true) || fourthButtonTouch == false) print("The giant
stops.");
else if (thirdButtonTouch) print("You have not touched the
second button.");
else if (secondButtonTouch) print("Touch any button to start
the game.");
else if(thirdButtonTouch) print("The giant goes to sleep.");
else if (firstButtonUntouch) print("You have not touched any
button.");
if(fourthButtonTouch) print("The giant wakes up.");
else print("You have not touched any button.");
}
```

Oto wynik poprzedniego kodu:

Here is the output of the previous code:

The giant is sleeping.

You have not touched any button.

Możesz zmienić wzór i zobaczyć, co się stanie.

Rozważmy pierwszą linijkę kodu pokazaną tutaj:

```
if (firstButtonUntouch == false || firstButtonTouch == true)
print("The giant is sleeping.");
```

FirstButtonUntouch początkowo miał wartość NULL. Potem użyliśmy specjalnego ??= i przypisano jego wartość do operatora FirstButtonTouch, czyli początkowo prawda. Dlatego też parametr

FirstButtonUntouch ma teraz wartość true. Teraz zestaw aksjomatów pomiędzy fałszem a prawdą? To prawda. I mamy wyjście.

Wyrażenia warunkowe

Dart ma dwa wyrażenia warunkowe, które mogą zastąpić klauzulę if-else podczas testowania małych wyrażeń. Rozważ ten kod:

```
//condition? exp1 : exp2;

int num1 = 20;

int num2 = 30;

int smallerNumber = num1 < num2? num1 : num2;

// oczekuje się, że liczba num1 będzie zawsze mniejsza
```

Tutaj porównujemy num1 z num2. Jeśli num1 jest mniejsza (num1 < num2 ma wartość true), przypisujemy zmienną liczbę1. Jeśli num1 < num2 jest fałszywa, przypisujemy liczbę2. Ogólna postać jest następująca: wyrażenie 1 jest zwracane, jeśli warunek jest prawdziwy, a wyrażenie 2, jeśli warunek jest fałszywy:

```
warunek? wyrażenie1 : wyrażenie2
```

Druga forma dotyczy wartości null.

```
int smallNumber = num1 ?? num2";
```

Jeśli liczba1 nie ma wartości null, przypisujemy ją do smallNumber. Jeśli ma wartość null, przypisujemy num2 do smallNumber.

Patrząc na pętlę

W programowaniu komputerowym, gdy musimy powtórzyć daną sekcję kodu określoną liczbę razy, aż spełniony zostanie określony warunek, stosujemy pętlę. Jest to struktura kontrolna, która jest powtarzana do momentu spełnienia określonego warunku. dla pętli

Ogólna składnia pętli for wygląda następująco:

```
for(var x = 0; x <= 10; x++){

//iteracja od 0 do 10 odbywa się pomiędzy

}
```

W poprzednim kodzie wartość x zaczyna się od 0. Następnie sprawdzamy, czy pętla się wykona (x <= 10;). Jeśli to wyrażenie zwróci wartość true, pętla wykona się, a my wykonamy ostatnią instrukcję w klauzuli for (x++), dodając 1 do x. Następnie pętla for sprawdza, czy powinna zostać uruchomiona ponownie; jeśli tak, to x++ również działa ponownie. Trwa to do momentu, gdy x <= 10 zwróci wartość false. Pętla for jest niezbędna do iteracji dowolnych zbiorów danych. Oto typowy przykład pętli for:

```
//code 2.4

main(List<String> arguments) {

var proverb = StringBuffer('As Dark as a Dungeon.');
```

```
for(var x = 0; x <= 10; x++){
proverb.write("!");
print(proverb);
}
}
```

W poprzednim kodzie korzystaliśmy z dwóch funkcji wbudowanych. Są to StringBuffer() i write(). Dostajemy je z bibliotek Dart. Dane wyjściowe są następujące:

```
//output of code 2.4
As Dark as a Dungeon.!
As Dark as a Dungeon.!!
As Dark as a Dungeon.!!!
As Dark as a Dungeon.!!!!
As Dark as a Dungeon.!!!!!
As Dark as a Dungeon.!!!!!!
As Dark as a Dungeon.!!!!!!!
As Dark as a Dungeon.!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!!
As Dark as a Dungeon.!!!!!!!!!!!
```

W naszych przyszłych dyskusjach będziemy dość szeroko używać pętli for, więc na tym się zatrzymamy. Powinieneś zrozumieć koncepcję, dlaczego znak wykrzyknika wzrósł z 0 do 10. Zatrzymuje się, gdy spełniony zostanie określony warunek (tutaj x=10). Teraz omówię interesującą funkcję iteracji kolekcji, a mianowicie użycie Set i Map. Jeśli obiekt, który chcesz iterować, jest iterowalny, możesz użyć metody forEach(). Za chwilę zaprezentujemy dwa zestawy kolekcji; jeden to Set, a drugi to Mapa.

```
//code 2.5
main(List<String> arguments) {
Set mySet = {1, 2, 3};
var myProducts = {
1 : 'TV',
2 : 'Refrigerator',
3 : mySet.lookup(2),
4 : 'Tablet',
5 : 'Computer'
```

```
};

var userCollection = {"name": "John Smith", 'Email':

'john@sanjib.site'};

myProducts.forEach((x, y) => print("${x} : ${y}"));

userCollection.forEach((k,v) => print('${k}: ${v}'));

}
```

Jak widać w poprzednim kodzie, istnieją dwa zestawy: myProducts i userCollection. W obu zestawach zadeklarowana jest para klucz=>wartość. W pierwszym przypadku kluczem jest 1, a wartością jest TV. Teraz Dart ma wbudowaną metodę forEach(key:value), której można użyć do podania wyniku. W pierwszym przypadku x jest kluczem, a y reprezentuje wartość. Następnie używamy interpolacji ciągów, aby uzyskać wynik. Oto dane wyjściowe:

```
//output of code 2.5

1 : TV

2 : Refrigerator

3 : 2

4 : Tablet

5 : Computer

name: John Smith

Email: john@sanjib.site
```

Jeśli nie znasz bieżącego licznika iteracji, funkcja forEach() metoda jest dobrym rozwiązaniem. W zwykłych przypadkach klasy Iterable, takie jak List i Set, obsługują także iterację w formie pętli for(). Rozważ ten kod:

```
//code 2.6

main(List<String> arguments) {

var myCollection = [1, 2, 3, 4];

for(var x in myCollection){

print("${x}");

}

}
```

Oto dane wyjściowe:

```
//wyjście kodu 2.6

1

2
```

3

4

whiile i do-while

W przypadku danego warunku logicznego pętla while steruje przepływem i wielokrotnie wykonuje wartość. Wykonuje pętlę przez blok kodu, o ile określony warunek jest prawdziwy. Rozważ ten prosty przykład, aby zrozumieć strukturę:

```
while (warunek) {

// blok kodu do wykonania

}
```

Oto prosty przykład, który wyświetla liczbę od 0 do 10:

```
int x = 0;

while (x <= 10) {

print("The output: ${x}");

x++;

}
```

Mam nadzieję, że widzisz podobieństwo między pętlami for i while. Struktura syntaktyczna jest po prostu inna. Zachowaj ostrożność podczas obsługi pętli while. Ponieważ pętla while ocenia warunek przed pętlą, musisz wiedzieć, jak zatrzymać pętlę w odpowiednim momencie, zanim wejdzie ona w nieskończoność.

```
//code 2.7

main(List<String> arguments) {

var num = 5;

var factorial = 1;

print("The value of the variable 'num' is decreasing this

way:");

while(num >=1) {

factorial = factorial * num;

num--;

print("'=>' ${num}");

}

print("The factorial is ${factorial}");

}
```

W poprzednim kodzie przed rozpoczęciem pętli pętla while ocenia warunek. Ponieważ wartość zmiennej num wynosi 5 i jest większa lub równa 1, warunek jest prawdziwy. Zatem zaczyna się pętla. Gdy pętla się rozpoczyna, zmniejszamy także wartość zmiennej num; w przeciwnym razie wszedłby w nieskończoną pętlę. Wartość zmiennej zmniejsza się w następujący sposób:

//wyjście kodu 2.8

Wartość zmiennej „num” maleje w następujący sposób:

'=>' 4

'=>' 3

„=>” 2

'=>' 1

'=>' 0

Silnia wynosi 120

W przypadku pętli do-while ocenia warunek po pętli.

//code 2.9

```
main(List<String> arguments) {

var num = 5;

var factorial = 1;

do {

factorial = factorial ∗ num;

num--;

print("The value of the variable 'num' is decreasing to :

${num}");

print("The factorial is ${factorial}");

}

while(num >=1);

}
```

Zmieniliśmy nieco fragment kodu, tak aby pokazywał wartość redukującą zmiennej, a jednocześnie pokazywał, jak rośnie wartość silni.

//wyjście kodu 2.10

Wartość zmiennej „num” maleje do: 4

Silnia wynosi 5

wartość zmiennej 'num' maleje do : 2

Silnia wynosi 60

Wartość zmiennej „num” maleje do: 1

Silnia wynosi 120

Wartość zmiennej „num” maleje do: 0

Silnia wynosi 120

Kiedy już zrozumiesz wzór pętli, możesz łatwo wybierać pomiędzy for, while i do-while. Spójrzmy na to teraz.

Wzory w pętli

Spotkałem wielu uczniów, którzy mieli wątpliwości co do pętli while. W tej sekcji omówię strukturę pętli, abyś mógł ją zrozumieć. Ludzie często nie wiedzą, że pętla for może również przekształcić się w pętlę nieskończoną, jeśli nie jest właściwie obsługiwana. Właściwie niektóre koncepcje pętli są takie same dla każdej pętli, czy to for, while czy do-while. Należy pamiętać o trzech rzeczach. • Zmienna licznika

• Sprawdzanie stanu

• W zależności od warunku, zwiększenie lub zmniejszenie

Rozważmy fragment kodu:

```
void forLoopFunction(){

for(var i = 0; i <= 5; i ++){

print(i);

}

}

void whileLoopFunction (){

var i = 0;

while(i <= 5){

print(i);

i++;

}

}

// in doWhileLoop the execution part comes before the specified

condition. The concept is same.

void doWhileLoopFunction (){

var i = 0;

do{
```

```
print(i);
i++;
} while(i <= 5);
}
main(){
//print(smallerNumber);
//print(smallNumber);
forLoopFunction();
print("");
whileLoopFunction();
print("");
doWhileLoopFunction();
}
```

Oto dane wyjściowe:

```
0
1
2
3
4
5
0
1
2
3
4
5
0
1
2
3
4
```

5

Rozważmy najpierw pętlę for.

```
for(var i = 0; i <= 5; i ++){

print(i);

}
```

Zaczęliśmy od zmiennej licznika, tutaj i = 0. Następnie sprawdziliśmy warunek, jak pokazano tutaj:

```
i <= 5
```

Po drugim kroku zwiększyliśmy wartość: i++. Kroki są dość logiczne. Nie mogliśmy zmniejszyć wartości. Doprowadziłoby to nas do nieskończonej pętli, ponieważ po rozpoczęciu od 0 wartość i zmniejszyłaby się, a określony warunek pozostałby prawdziwy na zawsze. Gdybyśmy zmniejszyli wartość i, pisząc i--, sprawdzanie warunku nigdy by się nie skończyło, dopóki nie pozwoliłaby na to pamięć naszego komputera. Zawieszenie lub zawieszenie następuje, gdy program przestaje odpowiadać na kod. Teraz zrobiliśmy to samo w pętli while. Kroki są tylko trochę inne.

```
var i = 0;

while(i <= 5){

print(i);

i++;

}
```

W poprzednim kodzie zmienna licznika pojawiała się przed rozpoczęciem pętli while. Pętla while rozpoczyna się od sprawdzenia warunku, jak pokazano poniżej:

```
i <= 5
```

To samo widzieliśmy w drugim kroku pętli for. Następnie, zgodnie z warunkiem, zwiększyliśmy wartość i wewnątrz pętli while. Gdy wartość i równa się 6, natychmiast przestaje odpowiadać na sygnały wejściowe. Daje wynik od 0 do 5. Przyjrzyjmy się teraz kodowi pętli do-while. Zaczynamy od zmiennej licznika, a następnie zwiększamy lub zmniejszamy wartość.

```
var i = 0;

do{

print(i);

i++;

} while(i <= 5);
```

W ostatnim etapie sprawdzamy warunek wewnątrz pętli while. Możesz zapytać, która pętla jest lepsza. Właściwie to zależy od kontekstu. W niektórych sytuacjach wystarczy pętla for. Tak naprawdę w większości przypadków możemy sobie z tym poradzić za pomocą pętli for. Jeśli chcemy wiedzieć, ile razy daną liczbę można podzielić przez 2, zanim będzie mniejsza lub równa 1, lepiej skorzystać z pętli while.

Etykieta dla pętli for

W niektórych sytuacjach używamy zagnieżdżonych pętli for. Wewnątrz pętli for możemy uruchomić kolejną pętlę for; a w wielu przypadkach jest to niezbędne. W Dart istnieje koncepcja zwana etykietą, która pozwala nam oddzielnie obsługiwać pętlę zewnętrzną i pętlę wewnętrzną. Za pomocą „kontynuuj” i „przerwaj” możemy przejść do etykiet. Przyjrzyjmy się najpierw kodowi; potem wyjaśnię, co się dzieje:

```
void labelsLoop (){

outerloop: for(var x = 1; x <= 3; x++){

print("One cycle of outerloop with $x starts and the whole

innerloop runs.");

innerloop: for(var y = 1; y <= 3; y++){

if(x == 1 && y == 1){

print("Since outerloop $x and innerloop $y both are 1,

it gives no output.");

break innerloop;

}

print(y);

}

print("One cycle of outerloop ends with $x");

}

}

main(List<String> arguments){

labelsLoop();

}
```

Jeśli spojrzysz na pokazane tutaj dane wyjściowe, możesz zrozumieć, jak to działa:

Rozpoczyna się jeden cykl pętli zewnętrznej z 1 i uruchamiana jest cała pętla wewnętrzna.

Ponieważ pętla zewnętrzna 1 i pętla wewnętrzna 1 mają wartość 1, nie daje to żadnego sygnału wyjściowego. Jeden cykl pętli zewnętrznej kończy się na 1

Jeden cykl pętli zewnętrznej z 2 startami i przebiega cała pętla wewnętrzna.

1

2

3

Jeden cykl pętli zewnętrznej kończy się 2

Jeden cykl pętli zewnętrznej z 3 startami i cała pętla wewnętrzna przebiega.

1

2

3

Jeden cykl pętli zewnętrznej kończy się na 3

Break możemy również zastosować w normalnych przypadkach, bez etykiety. Rozważ ten kod:

```
void main() {

for (var j = 0; j < 5; j++) {

if (j > 3 ) break ;

print(j);

}

}
```

Oto dane wyjściowe:

0

1

2

3

Jak widać w poprzednim kodzie, gdzie użyliśmy etykiet, zmienna licznika, sprawdzanie warunków i części inkrementujące są takie same zarówno w pętli zewnętrznej, jak i wewnętrznej. Zatem kiedy zewnętrzna pętla zaczyna się od 1, wewnętrzna pętla wewnątrz zewnętrznej pętli również zaczyna się od 1 i powinna zakończyć cały cykl. Jednak wstrzyknęliśmy instrukcję if i powiedzieliśmy programowi, że gdy wartość pętli zewnętrznej i wewnętrznej wynosi 1, przerwij pętlę wewnętrzną. Do rozgraniczenia pętli użyliśmy etykiet „Outerloop” i „Innerloop”. Używając instrukcji if, ten konkretny cykl pętli wewnętrznej nie mógł zakończyć całego cyklu. Później jednak toczy się jak zwykle. Etykieta jest charakterystyczną koncepcją Dart.

Kontynuuj z pętlą for

Właśnie widziałeś, jak wyraźnie przerwaliśmy pętlę wewnętrzną i zatrzymaliśmy jeden cykl pętli wewnętrznej. Zatem przerwa jest ważną koncepcją podczas korzystania z pętli for. Jednocześnie słowo kluczowe continue odgrywa również kluczową rolę w pętli for. Rozważmy ten fragment kodu:

```
void loopContinue(){

for(var num = 1; num <= 5; num++){

if(num % 2 == 0 ){

print("These are all even numbers. $num");

continue;

} print("These are all odd numbers. $num");
```

```
}
}
main(List<String> arguments){
loopContinue();
}
```

Przyjrzyj się wynikom, a zrozumiesz, jak działa słowo kluczowe „continue”. Wyprowadza nas z bieżącej pętli na początek następnej.

To wszystko są liczby nieparzyste. 1

To wszystko są liczby parzyste. 2

To wszystko są liczby nieparzyste. 3

To wszystko są liczby parzyste. 4

To wszystko są liczby nieparzyste. 5

break i continue to dwie ważne koncepcje nie tylko w Dart, ale także w każdym języku programowania. Podejmowanie decyzji za pomocą przełącznika i przypadku W niektórych przypadkach podejmowanie decyzji może być łatwiejsze, jeśli użyjesz przełącznika zamiast logiki if-else. instrukcje switch w Dart porównują liczby całkowite, ciągi znaków lub stałe czasu kompilacji, używając podwójnego znaku równości (==) za kulisami; utrzymuje jednak zasadę, że porównywane obiekty muszą być instancjami tej samej klasy, a nie żadnego z jej podtypów. Rozważmy najpierw prosty przykład:

```
void main() {
var marks = "A";
switch(marks) {
case "A": { print("Very Good"); }
break;
case "B": { print("Good"); }
break;
case "C": { print("Fair"); }
break;
case "D": { print("Poor"); }
break;
default: { print("Fail"); }
break;
}
}
```

Dane wyjściowe są bardzo dobre. Tutaj znaczniki obiektu są tej samej klasy, co w instrukcjach case (String). Spójrzmy na inny przykład:

```
//kod 2.11

main(List<String> arguments) {

//that could be the input value that would take inputs from

users

var startingTime = 5;

switch (startingTime) {

case 5:

print("Printer Ready");

break;

case 6:

print("Start printing");

break;

case 7:

print("Stop for a second");

break;

case 8:

print("Loading a tray and roll the paper.");

break;

case 9:

print("Printer Ready, start printing.");

break;

default:

print("Default ${startingTime}");

}

}
```

Znaczenie przerwy polega na tym, że jeśli warunek jest spełniony, następuje przełączenie

instrukcja kończy się, a program jest kontynuowany. Kiedy ktoś uruchamia drukarkę, otrzymujemy taki wynik, ponieważ czas początkowy wynosi 5:

```
//wyjście kodu 2.11

Drukarka gotowa
```

Użyliśmy klauzuli domyślnej do wykonania kodu, gdy żadna klauzula case nie pasuje.

Streszczenie

Kontrolowanie przepływu kodu jest istotne z wielu powodów. Jest to podstawa wszelkich algorytmów, które instruują komputery, aby zachowywały się w określony sposób. Budowa aplikacji mobilnej lub internetowej wymaga wielu takich instrukcji. Ponieważ algorytmy te mogą być złożone, wymagają zrozumienia kilku innych kluczowych pojęć, takich jak funkcje i programowanie obiektowe. W następnym rozdziale omówimy najpierw funkcje. Następnie dokładnie nauczysz się programowania obiektowego, a my wrócimy do tematu funkcji, aby omówić inne kluczowe ich cechy.

Funkcje i obiekty

Kiedy mówimy, że funkcje są obiektami w Dart, może to wydawać się mylące, jeśli jesteś początkującym. Zasadniczo, ponieważ Dart jest językiem obiektowym, nawet funkcje są obiektami i mają typ zwany Funkcją.

Oznacza to wiele rzeczy. Po pierwsze, możesz przypisać funkcję do zmiennej, a nawet przekazać funkcję jako argument do innych funkcji. Możesz także wywołać instancję klasy Dart tak, jakby była funkcją. W tym rozdziale najpierw dowiesz się, jak działa funkcja Dart. Wtedy nauczysz się o przedmiotach. Aby zrozumieć obiekty, musisz znać programowanie obiektowe, o czym również porozmawiamy w tym rozdziale.

Funkcje

Zobaczmy jak działają funkcje. Ta sekcja stanowi podstawowe wprowadzenie i omówię ten temat bardziej szczegółowo później, omawiając metody w programowaniu obiektowym. Przed napisaniem funkcji należy pamiętać o następujących głównych punktach:

• Dobrą praktyką jest zdefiniowanie typu funkcji. Dlatego zaleca się adnotację typu.

• Chociaż Dart zaleca adnotację typu, funkcja nadal działa bez deklaracji typu. Innymi słowy, możesz pominąć typ i napisać go prosto. Dart wykorzystuje interferencję typu. Oto przykład:

```
Mapa<String, dynamic> arguments = {'John': 'Smith', 'Chicago': 42};
```

Alternatywnie możesz użyć var i pozwolić Dartowi wywnioskować typ:

```
var argumenty = {'John': 'Smith', 'Chicago': 42};

// Mapa<String, Obiekt>
```

• Jednak najważniejszą rzeczą do zapamiętania w Dart jest to, że niezależnie od wartości, którą chcesz zwrócić z funkcji, musisz odpowiednio zmienić typ tej funkcji. Jeśli na przykład chcesz zwrócić wartość całkowitą, powinieneś zmienić typ funkcji na liczbę całkowitą.

• Krótko mówiąc, w przypadku void funkcja nie zwraca nic. Zatem za każdym razem, gdy przed funkcją użyjesz słowa kluczowego void, musisz użyć print(obiekt), aby zobaczyć, co dzieje się wewnątrz tej funkcji. Poniżej znajdują się dwie proste funkcje z adnotacjami typów, które wywołaliśmy wewnątrz funkcji main():

```
main(List<String> arguments) {

isTrue();
```

```
isFalse();
}
void isTrue(){
print("It's true.");
}
void isFalse(){
print("It's false.");
}
//output of code 3.1
It's true.
It's false.
```

Pomińmy typ i zobaczmy, jak działa ten sam kod.

```
//kod 3.2
main(List<String> arguments) {
isTrue();
isFalse();
}
isTrue(){
print("It's true.");
}
isFalse(){
print("It's false.");
}
```

Daje nam to ten sam wynik, ponieważ zgodnie z typem wartości jest ona wywnioskowana automatycznie. Tutaj Dart wie, że tutaj typy są nieważne, ponieważ nie używamy żadnych instrukcji return.

Zatem adnotacje typu nie mają znaczenia w takich przypadkach, ale przy budowaniu interfejsu API, konieczna jest adnotacja typu.

Uwaga: jeśli zmienisz ten kod, dodając typ bool przed nazwą funkcji, będzie on nadal działał bez żadnego błędu. Dart po prostu używa to co napisałeś, ponieważ nie ma powodu wyświetlać błędu (nie staramy się zwrócić czegoś, co nie jest boolem). Jest tak luźno. Możesz wywołać inną funkcję wewnątrz funkcji, jak pokazano tutaj:

```
//kod 3.3
```

```
main(List<String> arguments) {

myName();

}

myName(){

print("My name is John");

myAge(12);

}

myAge(int age){

print("My age is ${age}");

}
```

Później w funkcji myName() przekazaliśmy parametr lub argument wieku i otrzymaliśmy następujący wynik:

```
//wyjście kodu 3.3

My name is John

My age is 12
```

To było krótkie wprowadzenie do funkcji; jest oczywiste, że funkcja odgrywa tę samą rolę, jaką odgrywa czasownik w językach ludzkich. Jest to czynna część programowania. Zrozumiesz to lepiej, kiedy później omówimy metody programowania obiektowego. Zanim opuścimy tę sekcję, spójrzmy na inny fragment kodu, w którym faktycznie zwracamy wartość, więc typ zwracanej wartości jest ważny. Widzimy także inną składnię funkcji zwaną grubą strzałką.

```
void withoutReturningValue(){

print("We cannot return any value from this function.");

}

int anIntegerReturnTypeFunction(){

int num = 10;

return num;

}

//using Fat Arrow

String stringReturnTypeFunction(String name, String address) =>

"This is $name and this is $address and we have used the Fat

Arrow method.";

main(){

withoutReturningValue();
```

```
var returningInteger = anIntegerReturnTypeFunction();

print("We are returning an integer: $returningInteger");

print(stringReturnTypeFunction("John", "Jericho Town"));

}
```

Najpierw używamy int jako typu zwracanego przez funkcję IntegerReturnTypeFunction() i używamy słowa kluczowego return, aby określić zwracaną liczbę int. Wartość ta jest następnie przypisana do zmiennej w funkcji main(). Używając metody grubej strzałki, możemy zwrócić wartość z funkcji w jednym wierszu. Tym razem mówimy, że zwraca typ String.

Ostatnim rodzajem funkcji, któremu powinniśmy się przyjrzeć, jest funkcja rekurencyjna, pokazane tutaj:

```
int getRecurse(int num)

{

if (num > 1)

return num * getRecurse(num - 1);

else return 1;

}

main()

{

print(getRecurse(5));

}
```

Możesz zobaczyć, że w funkcji main() wywołujemy funkcję getRecurse() z parametrem int równym 5. Wewnątrz funkcji getRecurse() mamy klauzulę if kontrolującą liczbę powtórzeń. Jeśli parametr jest większy niż 1, kod mnoży parametr przez wyniki innego wywołania getRecurse, odejmując 1 od parametru. To wywołanie rekurencyjne jest powtarzane, aż parametr osiągnie wartość 1. Na tym etapie łańcuch rekurencji zostaje zakończony i zwracany jest końcowy wynik. Możesz dodać kilka instrukcji print, aby zobaczyć rekurencję w działaniu.

```
int getRecurse(int num)

{

if (num > 1) {

print("In getRecurse and num is $num");

return num * getRecurse(num - 1);

} else return 1;

}

main()
```

```
{
print(getRecurse(5));
}
```

The output is as follows:

```
In getRecurse and num is 5
In getRecurse and num is 4
In getRecurse and num is 3
In getRecurse and num is 2
120
```

Możesz zobaczyć, że 5 × 4 × 3 × 2 to 120, czyli ostatni wiersz wyniku.

Obiekty

Jak wiadomo, Dart jest językiem obiektowym, co oznacza, że posiada klasy i obiekty. Zacznijmy od prostej klasy i obiektu. Jak dotąd widziałeś zmienne i funkcje. Pomyślmy o czymś, co będzie przechowywać zmienne i funkcje. Nazywamy to klasą. Załóżmy, że mamy klasę Samochód, która ma trzy właściwości: nazwę, numer modelu i informację, czy jest włączona. Posiada również dwie metody (poza paradygmatem obiektowym nazywamy je funkcjami) zwane turnOn(bool) i isTurnedOn(). Rozważcie te części „akcji" klasy Car. Kiedy przekazujemy wartość bool true do turnOn(), samochód uruchamia się, a kiedy przekazujemy wartość bool false, samochód się zatrzymuje. Teraz wyobraź sobie, że producent chce zbudować wiele samochodów o różnych nazwach i numerach modeli, ale każdy z nich ma jedną metodę zwaną turnOn(bool). W tym scenariuszu każdy samochód jest obiektem lub instancją klasy Car. Rozważ następujący kod:

```
//kod 3.4
main(List<String> arguments) {
var newCar = new Car();
newCar.carName = "Red Angel";
newCar.carModel = 256;
newCar.turnOn(false);
if(newCar.isTurnedOn()){
print("${newCar.carName} starts. It has model number
${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number
${newCar.carModel}");
}
class Car {
```

```
int carModel = 123;

String carName = "Blue Angel";

bool isOn = true;

bool turnOn(bool turnOn){

isOn = turnOn;

}

bool isTurnedOn() {

return isOn;

}

}
```

It gives us this output:

```
//output of code 3.4

Red Angel stops. It has model number 256
```

Spójrz na klasę Car. Ma trzy właściwości lub atrybuty: carName, carModel i isOn. Traktujemy je jako zmienne, ale ponieważ znajdują się wewnątrz klasy, nazwiemy je właściwościami, elementami lub atrybutami. Wartości te można zmienić podczas tworzenia instancji. W rzeczywistości zrobiliśmy to wewnątrz funkcji main(). Wartości domyślne to 123, Blue Angel i true. Ale mamy pewne dane wyjściowe, w których nazwa zmienia się na Red Angel, model został zmieniony na 256, a samochód się zatrzymuje. Stworzyliśmy instancję lub obiekt klasy Car, po prostu pisząc tę linię:

```
var newCar = new Car();
```

Następnie zdefiniowaliśmy nazwę i numer modelu w następujący sposób:

```
newCar.carName = "Red Angel";

newCar.carModel = 256;
```

Kolejny krok jest istotny, ponieważ sprawdzamy, czy metoda isTurnedOn() zwraca prawdę.

```
if(newCar.isTurnedOn()){

print("${newCar.carName} starts. It has model number

${newCar.carModel}");

} else print("${newCar.carName} stops. It has model number

${newCar.carModel}");
```

Teraz zgodnie z naszą logiką, jeśli to prawda, samochód powinien odpalić. Ale w

wyjście, widzieliśmy, że się zatrzymuje. Dlaczego to się dzieje? Dzieje się tak, ponieważ w naszej klasie Car ustawiliśmy już tę wartość jako FAŁSZ. Zmieńmy to na true, jak pokazano tutaj:

```
//kod 3.5
```

```
main(List<String> arguments) {
var newCar = new Car();
newCar.carName = "Red Angel";
newCar.carModel = 256;
newCar.turnOn(true);
if(newCar.isTurnedOn()){
print("${newCar.carName} starts. It has model number
${newCar.carModel}");
} else print("${newCar.carName} stops. It has model number
${newCar.carModel}");
}
class Car {
int carModel = 123;
String carName = "Blue Angel";
bool isOn = true;
bool turnOn(bool turnOn){
isOn = turnOn;
}
bool isTurnedOn() {
return isOn;
}
}
```

Spójrz jeszcze raz na wynik:

//wyjście kodu 3.5

Zaczyna się Czerwony Anioł. Ma numer modelu 256

Z tego przykładu możemy wyciągnąć jeden wniosek: klasa jest planem obiektu. Obiekt lub instancja klasy ma niezwykłą moc; nie przypomina to prostych zmiennych, które przechowują jedno odniesienie do miejsca w pamięci, w którym możemy przechowywać tylko wartość. Poprzez obiekt aplikacji możemy uruchomić dużą, skomplikowaną aplikację; co więcej, za obiektem możemy stworzyć złożoną warstwę logiki.

Zagłębianie się w programowanie obiektowe

W poprzednim kodzie widziałeś następujące wiersze kodu, w których użyliśmy notacji (.), aby uzyskać wartość członków klasy Car:

```
if(newCar.isTurnedOn()){

print("${newCar.carName} starts. It has model number

${newCar.carModel}");

} else print("${newCar.carName} stops. It has model number

${newCar.carModel}");
```

Właściwie korzystaliśmy z członków klasy. Kiedy używamy notacji (.), zwykle mamy na myśli właściwości lub metody obiektu. Klasa może posiadać właściwości i metody. W końcu jest to plan zachowania obiektu. To, jak obiekt będzie się zachowywał w przyszłości, zależy od klasy, która została już napisana. To, czy obiekt samochodowy uruchomi się, czy zatrzyma, zależy od tego planu. Można więc powiedzieć, że obiekty mają elementy składające się z funkcji i danych; wywołując metodę, faktycznie wywołujesz ją na obiekcie. Zobaczmy więcej przykładów, aby zapoznać się z ideą klas i obiektów. Na początek załóżmy, że niedźwiedź zjada sześć ryb. Aby utworzyć obiekt ojca niedźwiedzia, musimy najpierw mieć klasę niedźwiedzia, w której mamy jedną właściwość składową dla „liczby ryb” i jedną metodę składową dla „zjedzenia takiej liczby ryb”. W idealnym przypadku zarówno właściwość, jak i metoda powinny być oznaczone adnotacją typu int.

```
//kod 3.6

class Bear {

int eatFish(int numberOfFish){

return numberOfFish;

}

}

main(List<String> arguments){

var fatherBear = new Bear();

print("Father bear eats ${fatherBear.eatFish(6)} number of

fish.");

}
```

To bardzo prosty program. Otrzymujemy następujące dane wyjściowe:

```
//wyjście kodu 3.6

Ojciec niedźwiedź zjada 6 ryb.
```

Czy możemy przenieść ten kod na wyższy poziom? Kiedy ojciec niedźwiedź je ryby i śpi przez kilka godzin, przybiera na wadze. Rozważ ten kod:

```
//kod 3.7

class Bear {

int numberOfFish;

int hourOfSleep;
```

```
int weightGain;
int eatFish(int numberOfFish){
return numberOfFish;
}
int sleepAfterEatingFish(int hourOfSleep){
return hourOfSleep;
}
int weightGaining(int numberOfFish, int hourOfSleep){
weightGain = numberOfFish * hourOfSleep;
return weightGain;
}
}
main(List<String> arguments){
var fatherBear = new Bear();
fatherBear.numberOfFish = 6;
fatherBear.hourOfSleep = 10;
print("Father bear eats ${fatherBear.eatFish(fatherBear.
numberOfFish)} number of fish. And he sleeps for ${fatherBear.
sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.
weightGaining(fatherBear.numberOfFish, fatherBear.
hourOfSleep)} pounds of weight.");
}
```

W poprzednim kodzie dodaliśmy kilka rzeczy, takich jak hourOfSleep i weightGain ; ponadto dodaliśmy dwie powiązane metody:

SleepAfterEatingFish() i WeightGaining(). Jak widzisz, przekazał za pomocą tych metod dwa powiązane parametry. Ojciec niedźwiedź śpi po zjedzeniu ryby i przybiera na wadze. Wartość przybieranej przez niego wagi wynika z pomnożenia godzin i liczby ryb. Otrzymujemy więc ten wynik podczas uruchamiania tego małego programu:

//wyjście kodu 3.7

Ojciec niedźwiedź zjada 6 ryb. I śpi 10 godzin.

Ojciec Niedźwiedź przybrał na wadze 60 funtów.

Dart to niezwykle elastyczny język. Możesz napisać ten sam kod w mniejszej liczbie wierszy niż w innych językach. Nie musisz używać typowych nawiasów klamrowych, możesz nawet pominąć słowo kluczowe return, aby automatycznie zwrócić wartość. Możesz także pominąć nowe słowo, aby utworzyć instancję. Teraz napiszemy ten sam kod w następujący sposób:

```
//kod 3.8

class Bear {

int numberOfFish;

int hourOfSleep;

int weightGain;

//changing the styles of the methods completely

int eatFish(int numberOfFish) => numberOfFish;

int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;

int weightGaining(int numberOfFish, int hourOfSleep) =>

weightGain = numberOfFish * hourOfSleep;

}

main(List<String> arguments){

var fatherBear = Bear(); //omitted the 'new' word

fatherBear.numberOfFish = 7;

fatherBear.hourOfSleep = 20;

print("Father bear eats ${fatherBear.eatFish(fatherBear.

numberOfFish)} fishes. And he sleeps for ${fatherBear.

sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");

print("Father bear has gained ${fatherBear.

weightGaining(fatherBear.numberOfFish, fatherBear.

hourOfSleep)} pounds of weight.");

}
```

Nieznacznie zmieniliśmy wartość godzin i liczbę ryb. Dane wyjściowe również się zmieniają, jak pokazano tutaj:

```
//wyjście kodu 3.8

Ojciec niedźwiedź zjada 7 ryb. I śpi 20 godzin.

Ojciec Niedźwiedź przybrał na wadze 140 funtów.
```

Aby ułatwić nam życie, w programowaniu obiektowym istnieje pojęcie zwane konstruktorem. Za każdym razem, gdy tworzysz instancję lub obiekt ze słowem kluczowym new lub bez niego, wewnątrz

klasy automatycznie wywoływana jest metoda. Metoda ta nazywana jest metodą konstruktora. W następnej sekcji omówimy koncepcję konstruktorów.

Badanie konstruktorów

Najważniejszym zadaniem konstruktorów jest tworzenie obiektów. Ilekroć próbujemy utworzyć obiekt i napisać tę linię:

```
var fatherBear = Bear();
```

tak naprawdę staramy się zorganizować miejsce w pamięci dla tego obiektu. Prawdziwa praca zaczyna się, gdy połączymy to miejsce z właściwościami i metodami klasy. Używając konstruktora, możemy wykonać to zadanie wydajniej, ponieważ konstruktorzy są na pierwszym miejscu podczas tworzenia instancji. Co więcej, Dart pozwala nam stworzyć więcej niż jednego konstruktora, co jest ogromną zaletą. Napiszmy naszą klasę Bear w nowy sposób, korzystając z konstruktora:

```
//kod 3.9

class Bear {

int numberOfFish;

int hourOfSleep;

int weightGain;

Bear(this.numberOfFish, this.hourOfSleep );// Constructor

int eatFish(int numberOfFish) => numberOfFish;

int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;

int weightGaining(int numberOfFish, int hourOfSleep) =>

weightGain = numberOfFish * hourOfSleep;

}

main(List<String> arguments){

var fatherBear = Bear(6, 10);

print("Father bear eats ${fatherBear.eatFish(fatherBear.

numberOfFish)} fishes. And he sleeps for ${fatherBear.

sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");

print("Father bear has gained ${fatherBear.weightGaining

(fatherBear.numberOfFish, fatherBear.hourOfSleep)} pounds of

weight.");

}
```

Utworzenie konstruktora jest niezwykle proste. Spójrz na tę linię:

```
Bear(this.numberOfFish, this.hourOfSleep);
```

Ta sama nazwa klasy działa jako metoda i za pośrednictwem tej metody przekazaliśmy dwa argumenty. Kiedy już otrzymamy te wartości, obliczamy trzecią zmienną dotyczącą przyrostu masy ciała. W dalszej części tego rozdziału książki porozmawiamy więcej o konstruktorach. Teraz łatwiej będzie przekazać te dwie wartości podczas tworzenia obiektu. Użyliśmy słowa kluczowego this. Słowo kluczowe this reprezentuje ukryty obiekt wskazujący na bieżący obiekt klasy. Moglibyśmy zrobić to samo, tworząc konstruktor w ten sposób, który jest bardziej tradycyjny:

```
//kod 3.10

class Bear {

int numberOfFish;

int hourOfSleep;

int weightGain;

Bear(int numOfFish, int hourOfSleep ){// constructor

this .numberOfFish = numOfFish ;// using this keyword to

point out the current

class object

this .hourOfSleep = hourOfSleep;

}

//Bear(this.numberOfFish, this.hourOfSleep);

int eatFish(int numberOfFish) => numberOfFish;

int sleepAfterEatingFish(int hourOfSleep) => hourOfSleep;

int weightGaining(int numberOfFish, int hourOfSleep) =>

weightGain = numberOfFish * hourOfSleep;

}

main(List<String> arguments){

var fatherBear = Bear(6, 10);

print("Father bear eats ${fatherBear.eatFish(fatherBear.

numberOfFish)} fishes. And he sleeps for ${fatherBear.

sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");

print("Father bear has gained ${fatherBear.

weightGaining(fatherBear.numberOfFish, fatherBear.

hourOfSleep)} pounds of weight.");

}
```

W obu przypadkach wynik jest taki sam jak poprzednio:

//wyjście kodu 3.10

Ojciec niedźwiedź zjada 6 ryb. I śpi 10 godzin.

Ojciec Niedźwiedź przybrał na wadze 60 funtów.

W poprzednim kodzie można było nawet bardzo łatwo określić typ obiektu. Typ wartości możemy dość łatwo zmienić. Obejrzyj jeszcze raz funkcję main():

//kod 3.11

```
main(List<String> arguments){
var fatherBear = Bear(6, 10);
fatherBear.weightGain = fatherBear.numberOfFish * fatherBear.
hourOfSleep;
print("Father bear eats ${fatherBear.eatFish(fatherBear.
numberOfFish)} fishes. And he sleeps for ${fatherBear.
sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.
weightGaining(fatherBear.weightGain)} pounds of weight.");
print("The type of the object : ${fatherBear.weightGain.
runtimeType}");
String weightGained = fatherBear.weightGain.toString();
print("The type of the same object has changed to :
${weightGained.runtimeType}");
}
```

Oto dane wyjściowe:

//kod 3.12

```
main(List<String> arguments){
var fatherBear = Bear(6, 10);
print("Father bear eats ${fatherBear.eatFish(fatherBear.
numberOfFish)} fishes. And he sleeps for ${fatherBear.
sleepAfterEatingFish(fatherBear.hourOfSleep)} hours.");
print("Father bear has gained ${fatherBear.
weightGaining(fatherBear.numberOfFish, fatherBear.
hourOfSleep)} pounds of weight.");
```

```
print("The type of the object : ${fatherBear.weightGain.

runtimeType}");

String weightGained = fatherBear.weightGain.toString();

print("The type of the same object has changed to :

${weightGained.runtimeType}");

}
```

Jak implementować klasy

Teraz masz pojęcie o tym, jak klasy i obiekty współdziałają ze sobą. Klasa to plan zawierający pewne zmienne i metody instancji. Klasa może mieć wiele zadań, ale dobrą praktyką i jednym z głównych paradygmatów programowania obiektowego jest to, że pojedyncza klasa powinna mieć jedno zadanie. Gdy wiele klas współpracuje ze sobą, nie powinny być one ściśle powiązane. Powinny być luźno powiązane. Luźno powiązane oznacza, że w przypadku użycia wielu obiektów z różnych klas nie należy ich sklejać. Nie powinny oddziaływać na inne obiekty, gdy na nie wpływają. Jest to zasada znana jako zasada projektowania SOLID. Krótko mówiąc, oznacza to, że jeden obiekt nie powinien kolidować z innym obiektem. Rozważmy klasę Car, w której klasa Wheel nie powinna być sklejana z klasą Steering. Dlatego też, gdy złapiemy gumę, nadal możemy skierować samochód w bezpieczne miejsce. Tworząc aplikacje, należy zawsze próbować oddzielić wszystkie klasy. W Darcie moglibyśmy zastosować tę samą zasadę podczas tworzenia klas. Stwórzmy jedną klasę z jednym zadaniem. Stworzymy klasę, która sprawdzi, czy adres URL jest bezpieczny, czy nie.

```
//kod 3.13

class CheckHTTPS {

String urlCheck;

CheckHTTPS(this.urlCheck);

bool checkURL(String urlCheck){

if(this.urlCheck.contains("https")){

return true;

} else return false;

}

}

main(List<String> arguments){

var newURL = CheckHTTPS('http://sanjib.site');

if(!newURL.checkURL(newURL.urlCheck)) {

print("The URL ${newURL.urlCheck} is not secured");

}

}
```

Otrzymujemy ten wynik po sprawdzeniu adresu URL:

//wyjście kodu 3.13

Adres URL http://sanjib.site nie jest zabezpieczony

Mamy więc kilka podstawowych kroków do wykonania. Ilekroć chcemy stworzyć klasę, powinniśmy mieć jasną wizję tego, co ta klasa będzie robić. Jakie będzie jego zadanie? Po pierwsze potrzebujemy kilku zmiennych. Następnie potrzebujemy jednej lub więcej metod, w których możemy bawić się tymi zmiennymi.

//kod 3.14

```
class MyClass {

String myVariable; // property or instance variable, initially

null

MyClass(this.myVariable); //constructor

String myMethod(){ //method declaration

return "This is my method and this is ${myVariable}";

//returning value

}

}

main(List<String> arguments){

var myObject = MyClass("My String"); // creating new instance

of class MyClass

print("${myObject.myMethod()}"); //printing the value

}
```

Spójrz na kod: najpierw zadeklarowaliśmy zmienną instancji. Jest typu String. Ponieważ nie zainicjowaliśmy zmiennej, początkowo ma ona wartość null. W kolejnym kroku skonstruowaliśmy obiekt, deklarując konstruktor, do którego przekazaliśmy zmienną instancji. Typ naszej metody to również String. W metodzie zwróciliśmy obiekt String. W funkcji main() utworzyliśmy obiekt i zadeklarowaliśmy typ jako MyClass; i jednocześnie przekazaliśmy konstruktorowi wartość ciągu. Na koniec wywołaliśmy metodę klasy i wyświetliliśmy wynik. W następnej sekcji napiszemy kilka metod i spróbujemy zrozumieć, jak działają. Metody są istotną częścią każdej klasy, ponieważ są częścią akcji.

Zakres leksykalny w funkcjach

Koncepcja ta jest niezwykle istotna w kontekście funkcji Darta.

Uwaga: Później, gdy zagłębimy się w programowanie obiektowe, zobaczymy, jak koncepcje dostępu odgrywają istotną rolę w Dart.

Wróćmy do funkcji. Najpierw spójrz na poniższy kod i przeczytaj komentarze:

//kod 3.15

```
var outsideVariable = "I am an outsider.";

main(List<String> arguments){

//we can access the outside variable

print(outsideVariable);

// we cannot access the insider variable, it gives us error

//print(insiderVariable);

// it is an insider function

String insiderFunction(){

// I can access the outside variable, no problem

print("This is from the insider function.");

print(outsideVariable);

String insiderVariable = "I am an insider";

print(insiderVariable); // it's okay to access this insider

}

insiderFunction();

}
```

Najpierw zadeklarowaliśmy zmienną poza naszą funkcją main(). Nazywa się to zmienną zewnętrzną. Możemy uzyskać dostęp do tej zmiennej wewnątrz funkcji main() jako obiekt. Pamiętaj, że wszystko w Darcie jest obiektem. Po drugie, zadeklarowaliśmy funkcję insider o nazwie funkcja insider(). Teraz wewnątrz tej funkcji insider możemy bezpiecznie wywołać zmienną outsider. Ponadto, jeśli utworzymy kolejną zmienną poufną, możemy ją również nazwać. Otrzymujemy więc ten wynik:

```
//wyjście kodu 3.15

I am an outsider.

This is from the insider function.

I am an outsider.

I am an insider
```

W związku z tym nie ma problemu z wydajnością. Jednak nie będzie to takie samo doświadczenie, jeśli spróbujemy wywołać zmienną insider spoza zakresu naszej funkcji insider.

```
//kod 3.16

var outsideVariable = "I am an outsider.";

main(List<String> arguments){

//we can access the outside variable

print(outsideVariable);
```

```
// we cannot access the insider variable, it gives us error

print(insiderVariable);

// it is an insider function

String insiderFunction(){

// I can access the outisde variable, no problem

print("This is from the insider function.");

print(outsideVariable);

String insiderVariable = "I am an insider";

print(insiderVariable); // it's okay to access this insider

}

insiderFunction();

}
```

Teraz spójrz na wynik:

```
//wyjście kodu 3.16

bin/main.dart:11:9: Błąd: Nie znaleziono programu pobierającego: „insiderVariable”.

print(zmienna poufna);

^^^^^^^^^^^^^^^^^
```

Powinniśmy zrozumieć tę sprawę „wewnątrz i na zewnątrz”. Nazywa się to zakresem leksykalnym. Możesz wywołać zmienną zewnętrzną wewnątrz funkcji main(). Jeśli jednak zdefiniujesz obiekt wewnątrz funkcji, nie możesz wywołać go spoza tej funkcji.

Kilka słów o getterze i seterze

Wróćmy ponownie do tematu programowania obiektowego, aby poznać kluczowe pojęcia zwane getterem i setterem. Możemy jawnie ustawić wartość i uzyskać ją w ten sposób, używając metody . notacja:

```
//kod 3.17

class myClass {

String name;

String get getName => name;

set setName(String aValue) => name = aValue;

}

main(List<String> arguments){

var myObject = myClass();
```

```
myObject.setName = "Sanjib";

print(myObject.getName);

}
```

To daje nam wynik Sanjib. Ale jak to się dzieje? W myClass zdefiniowaliśmy metodę setName(), która akceptuje parametr o nazwie aValue. Później wywołaliśmy tę metodę poprzez instancję (myObject. setName) klasy myClass. Ciekawostką jest to, że metoda setName(String aValue) zdefiniowana w myClass działa teraz jako atrybut. Możesz zapytać, dlaczego powinniśmy używać modułu pobierającego i ustawiającego, skoro każda klasa została powiązana z domyślnym modułem pobierającym i ustawiającym? Właściwie nadpisujemy wartość domyślną, jawnie definiując moduł pobierający i ustawiający.

Różne typy parametrów

Niezależnie od tego, czy jest to metoda klasowa, czy funkcja, czasami trzeba przekazać wartości. Możesz nazwać je argumentami lub parametrami, jakkolwiek chcesz. Dart jest elastyczny; daje programistom szerokie możliwości manipulowania parametrami. Możesz użyć parametrów domyślnych; w takich przypadkach należy podać wartość domyślną. Jest to obowiązkowe. Ale w Dart dostępne są trzy inne opcje. Można używać parametrów pozycyjnych, parametrów nazwanych i parametrów opcjonalnych.

Poniższy kod wykorzystuje parametry domyślne i pozycyjne:

```
//kod 3.18

/default parameters

String defaultParameters(String name, String address, {int

age = 10}){

return "$name and $address and age $age";

}

//optional parameters

String optionalParameters(String name, String address,

[int age] ){

return "$name and $address and $age";

}

main(){

print(defaultParameters("John", "Jericho"));

print(optionalParameters("John", "Form Chikago"));

// overriding the default age

print(defaultParameters("JOhn", "Jericho", age : 20));

}
```

Wewnątrz funkcji main() w naszej domyślnej funkcji parametru przekazaliśmy tylko dwie wartości: nazwę i adres. Nie przekroczyliśmy wieku. Nie musieliśmy, bo zostało to już zdefiniowane w naszej funkcji: {int age = 10}. Pamiętaj o użyciu nawiasów klamrowych do zdefiniowania parametru domyślnego. Czy możemy zastąpić parametr domyślny? Tak możemy. Spójrz na tę część funkcji main():

```
// overriding the default age

print(defaultParameters("JOhn", "Jericho", age : 20));
```

Nadpisaliśmy domyślny wiek i zmieniliśmy go z 10 na 20. Następnie w opcjonalnej funkcji parametru uczyniliśmy wiek opcjonalnym, utrzymując wartość w nawiasach kwadratowych.

```
//optional parameters

String optionalParameters(String name, String address, [int age] ){

return "$name and $address and $age";

}
```

Ponieważ parametr wiek jest opcjonalny, możemy go przekazać lub zignorować. Jednak zignorowanie opcjonalnego parametru spowoduje ustawienie go na wartość null. Zatem wynik poprzedniego kodu będzie wyglądał następująco:

```
//wyjście kodu 3.19

John and Jericho and age 10

John and Form Chikago and null

JOhn and Jericho and age 20
```

W przypadku nazwanego parametru możemy zamienić wartości, ponieważ użycie nazwanego parametru jest bardzo elastyczne. Tutaj kolejność nie ma znaczenia. Rozważmy ten kod:

```
//kod 3.20

//named parameter

int findTheVolume(int length, {int height, int breadth}){

return length * height * breadth;

}

void main(){

//sequence does not matter

var result1 = findTheVolume(10, height: 20, breadth: 30);

var result2 = findTheVolume(10, breadth: 30, height: 10);

print(result1);

print(result2);

}
```

W poprzednim kodzie wysokość i szerokość umieściliśmy w nawiasach klamrowych. Są to więc nazwane parametry, którymi możemy się wymieniać podczas przekazywania wartości. Zamiana wartości nie będzie miała wpływu na nasz kod. To jest zaleta nazwanych parametrów.

Więcej o konstruktorach

W każdej klasie istnieje wiele typów konstruktorów, których można używać w dowolnej aplikacji. Jak zwykle mamy konstruktora domyślnego. Możemy przez niego przekazywać parametry. Mamy również nazwane parametry. Przyjrzyjmy się poniższemu fragmentowi kodu i spróbujmy zrozumieć, jak one działają:

```
//kod 3.21

class Bear {

//reference variable

int collarID;

//default and parameterized constructor

Bear(this.collarID);

//first named constructor

Bear.firstNamedConstructor(this.collarID);

//second named constructor

Bear.secondNamedConstructor(this.collarID);

void trackingBear() {

String color; // local varia print("Tracking the bear

with collar ID ${collarID}");

}

}

main(List<String> arguments){

// bear1 is reference variable

// Bear() is object// It should be class no object I suppose

var bear1 = Bear(1);

bear1.trackingBear();

var bear2 = Bear.firstNamedConstructor(2);

bear2.trackingBear();

var bear3 = Bear.secondNamedConstructor(3);

bear3.trackingBear();

}
```

W poprzednim kodzie, zgodnie z konwencją Dart, pisząc klasę, mogliśmy mieć wiele rzeczy na miejscu. Po pierwsze, mamy tutaj zmienną referencyjną:

int collarID;. Zmienna o nazwie kołnierzID zawiera odniesienie do int obiekt z wartością obiektu Bear.

Kiedy utworzymy instancję, wewnątrz funkcji main() zrobimy to ponownie aby mieć zmienną referencyjną.

// bear1 is reference variable

// Bear() is object

var bear1 = Bear(1);

Przekazaliśmy zmienną referencyjną na poziomie klasy collarID przez domyślny konstruktor.

Tak więc, definiując klasę, a następnie tworząc instancję, mamy dwa typy zmiennych referencyjnych: pierwszy to zmienna referencyjna na poziomie klasy, a drugi to zmienna referencyjna na poziomie obiektu lub instancji. Jeśli nie ma to żadnego sensu, nie martw się. Omówimy to w rozdziale 7. W części konstruktora mamy jeden domyślny i sparametryzowany konstruktor, pokazany tutaj:

//domyślny i sparametryzowany konstruktor

Bear(this.collarID);

Poza tym mamy dwóch nazwanych konstruktorów.

//pierwszy nazwany konstruktor

Bear.firstNamedConstructor(this.collarID);

//drugi nazwany konstruktor

Bear.drugiNamedConstructor(this.collarID);

Za pomocą nazwanych konstruktorów utworzyliśmy trzy instancje niedźwiedzia; ponadto każda instancja ma tę samą funkcjonalność. Wreszcie, po uruchomieniu kodu nie można rozróżnić zachowania kodu korzystającego z konstruktora domyślnego od kodu korzystającego z nazwanych konstruktorów.

Śledzenie niedźwiedzia za pomocą collarr ID 1

Śledzenie niedźwiedzia za pomocą collar ID 2

Śledzenie niedźwiedzia za pomocą collar ID 3

Jedną z kluczowych cech programowania obiektowego jest możliwość rozszerzania zajęć. Rozszerzasz klasę, aby utworzyć inną klasę, a rozszerzona klasa jest nazywana podklasą. Podklasa dziedziczy zmienne referencyjne i metody klas z klasy nadrzędnej, zwanej nadklasą. Właściwości klasy nadrzędnej są dziedziczone przez klasę podrzędną; ponieważ właściwości klasy nadrzędnej są rozszerzane na klasę podrzędną, klasa nadrzędna nazywana jest także klasą bazową. Z tego samego powodu klasa potomna nazywana jest klasą pochodną, ponieważ dziedziczy właściwości klasy bazowej. Możliwość ta, znana jako dziedziczenie, działa na dwa sposoby. Po pierwsze, możesz utworzyć nowe klasy na podstawie istniejących. Nazywa się to pojedynczym dziedziczeniem. Dart nie obsługuje dziedziczenia wielokrotnego (dziedziczenia z więcej niż jednej klasy). Obsługuje jednak dziedziczenie wielopoziomowe. Możemy zatem stwierdzić, że Dart obsługuje dwa rodzaje dziedziczenia.

• Pojedyncze dziedzictwo

• Dziedziczenie wielopoziomowe

Pierwsze spojrzenie na dziedziczenie

Rozważmy prosty przykład, w którym rozszerzyliśmy klasę Animal do klasy Cat. To jest przykład pojedynczego dziedziczenia.

```
//kod 4.1

class Animal {

String name = "Animal";

Animal(){

print("I am Animal class constructor.");

}

Animal.namedConstructor(){

print("This is parent animal named constructor.");

}

void showName(){

print(this.name);

}

void eat(){

print("Animals eat everything depending on what type it

is.");

}

}

class Cat extends Animal {

//overriding parent constructor

//although constructors are not inherited

Cat() : super(){

print("I am child cat class overriding super Animal

class.");

}

Cat.namedCatConstructor() : super.namedConstructor(){

print("The child cat named constructor overrides the parent
```

```
animal named constructor.");
}
@override // method overriding
void showName(){
print(this.name);
}
@override
void eat(){
super.eat();
print("Cat doesn't eat vegetables..");
}
}
main(List<String> arguments){
var cat = Cat();
cat.name = "Meaow";
cat.showName();
cat.eat();
var anotherCat = Cat.namedCatConstructor();
}
```

Przyjrzyjmy się najpierw wynikom; następnie omówimy cechy podklas i nadklas.

```
//wyjście kodu 4.1
I am Animal class constructor.
I am child cat class overriding super Animal class.
Hi from cat.
Animals eat everything depending on what type it is.
Cat doesn't eat vegetables..
This is parent animal named constructor. The child cat named constructor overrides the parent animal named constructor.
```

Kod jest dość prosty do naśladowania; nadklasa lub klasa bazowa Animal ma dwa konstruktory: konstruktor domyślny i nazwany. Podklasy nie dziedziczą konstruktorów ze swojej nadklasy. Podklasa lub klasa pochodna Cat zastępuje oba konstruktory. Musisz określić, który konstruktor zastępujesz w definicji konstruktora podklasy. Jeśli tego nie zrobisz, nazwany konstruktor podklasy zastąpi domyślny konstruktor klasy nadrzędnej.

```
Cat.namedCatConstructor() : super.namedConstructor(){

print("The child cat named constructor overrides the parent

animal named constructor.");

}
```

Zmieńmy teraz trochę kod i postępujmy zgodnie z wynikami. Koncepcję pojedynczego dziedziczenia lepiej zrozumiesz w drugim przykładzie.

```
//kod 4.2

class Animal {

String name = "Animal";

Animal(){

print("I am Animal class constructor.");

}

Animal.namedConstructor(){

print("This is parent animal named constructor.");

}

void showName(){

print(this.name);

print("Hi from ${this.name}");

}

void eat(){

print("Animals eat everything depending on what type it

is.");

}

}

class Cat extends Animal {

//overriding parent constructor

//although constructors are not inherited

Cat() : super(){

print("I am child cat class overriding super Animal class.");

}

Cat.namedCatConstructor() : super.namedConstructor(){
```

```
print("The child cat named constructor overrides the parent
animal named constructor.");
}
@override
void showName(){
print("Hi from cat.");
print(this.name);
}
@override
void eat(){
super.eat();
print("Cat doesn't eat vegetables..");
}
}
class Cow extends Animal {
//overriding parent constructor
//although constructors are not inherited
Cow() : super(){
print("I am child cow class overriding super Animal
class.");
}
Cow.namedCatConstructor() : super.namedConstructor(){
print("The child cow named constructor overrides the parent
animal named constructor.");
}
@override
void showName(){
print("Hi from cow.");
print(this.name);
}
@override
```

```
void eat(){
super.eat();
print("Cow does eat grass..");
}
}
main(List<String> arguments){
var cow = Cow();
cow.name = "Daisy";
cow.showName();
var cat = Cat();
cat.name = "Meaow";
cat.showName();
cat.eat();
var anotherCat = Cat.namedCatConstructor();
}
```

Dodaliśmy więcej linii w klasie nadrzędnej, utworzyliśmy nową klasę Cow i dodaliśmy kilka linii do obu klas podrzędnych; jednocześnie dodaliśmy kilka linii w naszej funkcji main(), aby uzyskać wynik. Oto nowe dane wyjściowe:

```
//wyjście kodu 4.2
/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:
33101 /home/ss/IdeaProjects/bin/main.dart
Observatory listening on http://127.0.0.1:33101/
I am Animal class constructor.
I am child cow class overriding super Animal class.
Hi from cow.
Daisy
I am Animal class constructor.
I am child cat class overriding super Animal class.
Hi from cat.
Meaow
Animals eat everything depending on what type it is.
Cat doesn't eat vegetables..
```

This is parent animal named constructor.

The child cat named constructor overrides the parent animal

named constructor.

Process finished with exit code 0

Jak widać, na nadklasie może opierać się więcej niż jedna klasa.

Dziedziczenie wielopoziomowe

Rozważmy najpierw kod, a po przejrzeniu wyników omówimy, jak działa dziedziczenie wielopoziomowe.

```
//kod 4.3
class Animal {
String name = "Animal";
Animal(){
print("I am Animal class constructor.");
}
Animal.namedConstructor(){
print("This is parent animal named constructor.");
}
void showName(){
print(this.name);
print("Hi from ${this.name}");
}
void eat(){
print("Animals eat everything depending on what type it
is.");
}
}
class Dog extends Animal {
//overriding parent constructor
//although constructors are not inherited
Dog() : super(){
```

```
print("I am child class dog overriding super Animal
class.");
}
Dog.namedDogConstructor() : super.namedConstructor(){
print("The child dog named constructor overrides the parent
animal named constructor.");
}
Dog.anotherNamedConstructor(){
print("This is parent Dog named constructor.");
}
@override
void showName(){
print("Hi from parent dog.");
print(this.name);
}
@override
void eat(){
super.eat();
print("Dog doesn't eat vegetables..");
}
}
class PuppyDog extends Dog {
//overriding parent constructor
//although constructors are not inherited
PuppyDog() : super(){
print("I am child class puppy dog overriding my immediate
parent Dog class.");
}
PuppyDog.namedDogConstructor() : super.anotherNamedConstructor(){
print("The child puppy dog named constructor overrides the
parent Dog another named constructor.");
```

```
}
@override
void showName(){
print("Hi from puppy dog.");
print(this.name);
}
@override
void eat(){
super.eat();
print("Puppy Dog eats milk only ...");
}
}
main(List<String> arguments){
var animal = Animal();
animal.name = "Cow";
animal.showName();
var dog = Dog();
dog.name = "Lucky";
dog.showName();
dog.eat();
var anotherDog = Dog.namedDogConstructor();
var puppy = PuppyDog();
puppy.name = "I am offspring of Lucky";
puppy.showName();
puppy.eat();
var anotherPuppy = PuppyDog.namedDogConstructor();
}
```

Oto dane wyjściowe:

```
//wyjście kodu 4.3
/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:
40767 /home/ss/IdeaProjects/bin/main.dart
```

Observatory listening on http://127.0.0.1:40767/

I am Animal class constructor.

Cow

Hi from Cow

I am Animal class constructor.

I am child class dog overriding super Animal class.

Hi from parent dog.

Lucky

Animals eat everything depending on what type it is.

Dog doesn't eat vegetables..

This is parent animal named constructor.

The child dog named constructor overrides the parent animal

named constructor.

I am Animal class constructor.

I am child class dog overriding super Animal class.

I am child class puppy dog overriding my immediate parent Dog class.

Hi from puppy dog.

I am offspring of Lucky

Animals eat everything depending on what type it is.

Dog doesn't eat vegetables..

Puppy Dog eats milk only ...

I am Animal class constructor.

This is parent Dog named constructor.

The child puppy dog named constructor overrides the parent Dog

another named constructor.

Process finished with exit code 0

W poprzednim kodzie klasą nadrzędną jest Animal. Klasa Dog dziedziczy wszystkie jej właściwości. Następnie klasa Dog ma swoje potomstwo, klasę PuppyDog. Teraz klasa PuppyDog dziedziczy po klasie Dog. Tutaj mamy właściwie dwie klasy podrzędne lub podstawowe: Dog i PuppyDog. Różni się to jednak od dziedziczenia pojedynczego, ponieważ w przypadku dziedziczenia wielopoziomowego jedna klasa podrzędna jest dziedziczona z innej klasy podrzędnej. W tym przykładzie klasa podrzędna PuppyDog dziedziczy z innej klasy podrzędnej Dog. Możesz porównać to drzewo rodowodu rodzinnego do ludzkiego. Mam ojca, ale mój ojciec ma ojca, który jest moim dziadkiem i tak dalej.Mieszanki: dodawanie większej liczby funkcji do klasy. Dart ma wiele do zaoferowania, gdy klasy wymagają

ponownego wykorzystania; istnieje ważna koncepcja zwana miksowaniem. Jest to sposób na ponowne wykorzystanie kodu dowolnej klasy w wielu hierarchiach klas. Możemy przepisać poprzedni kod za pomocą miksów. Wszystko, co musimy zrobić, to użyć słowa kluczowego with. Załóżmy, że mamy klasę Dog z metodą canRun(). Obiekt Kot może również działać, prawda? Wypróbujmy ten sam kod w nieco inny sposób.

```
//kod 4.4

class Animal {

String name = "Animal";

Animal(){

print("I am Animal class constructor.");

}

Animal.namedConstructor(){

print("This is parent animal named constructor.");

}

void showName(){

print(this.name);

}

void eat(){

print("Animals eat everything depending on what type it

is.");

}

}

class Dog {

void canRun(){

print("I can run.");

}

}

class Cat extends Animal with Dog {//reusing another class

//overriding parent constructor

//although constructors are not inherited

Cat() : super(){

print("I am child cat class overriding super Animal class.");
```

```
}
Cat.namedCatConstructor() : super.namedConstructor(){
print("The child cat named constructor overrides the parent
animal named constructor.");
}
@override
void showName(){
print("Hi from cat.");
}
@override
void eat(){
super.eat();
print("Cat doesn't eat vegetables..");
}
}
main(List<String> arguments){
var cat = Cat();
cat.name = "Meaow";
cat.showName();
cat.eat();
var anotherCat = Cat.namedCatConstructor();
anotherCat.canRun();
}
```

Podklasa Cat została rozbudowana, a jednocześnie zastosowano w niej miksy, wykorzystując ponownie kod klasy Dog. Spójrz na tę linię:

```
class Cat extends Animal with Dog {...}
```

W funkcji main() obiekt Cat wykorzystuje metodę klasy Dog w 55tes sposób:

```
anotherCat.canRun();
```

Dane wyjściowe nie zostały zmienione z wyjątkiem ostatniej linii, jak pokazano tutaj:

```
//wyjście kodu 4.4
I am Animal class constructor.
```

I am child cat class overriding super Animal class.

Hi from cat.

Animals eat everything depending on what type it is.

Cat doesn't eat vegetables..

This is parent animal named constructor.

The child cat named constructor overrides the parent animal

named constructor.

I can run.

Pamiętaj, że w przypadku miksów musisz użyć słowa kluczowego with, po którym następuje jedna lub więcej nazw miksów.

Uwaga: obsługa miksów została wprowadzona w Dart 2.1. Wcześniej w takich przypadkach używano klasy abstrakcyjnej. W następnym rozdziale poznasz klasy i metody abstrakcyjne.

Mieszanki są rodzajem ograniczonego dziedziczenia wielokrotnego; w poprzednim kodzie rozszerzamy jedną klasę (Animal), a następnie używamy miksu, aby wprowadzić funkcje z innej (Pies). Powinieneś zauważyć tutaj jedną cechę; na każdym etapie korzystamy wyłącznie z zajęć. Możemy dziedziczyć po klasie, ale możemy również użyć klasy jako miksu, używając słowa kluczowego with. W następnym rozdziale dodamy kolejną funkcję: interfejsy. Budują one kontrakt pomiędzy dwiema klasami, dzięki czemu nie musimy na stałe kodować funkcjonalności jednej klasy w innej klasie. Dopóki klasa jest zgodna z umową, możemy ją zmienić bez wpływu na klasę wywołującą.

Relacje między jednostkami: klasy abstrakcyjne, interfejsy i obsługa wyjątków

W poprzednim rozdziale dowiedziałeś się, że byty nie istnieją w izolacji. Widziałeś kilka przykładów dziedziczenia. Więcej zobaczycie za chwilę, chociaż w różnych formach. Istnieje jeszcze kilka typów relacji pomiędzy klasami. Relacja pomiędzy każdą klasą jest zawsze definiowana wcześniej, dzięki czemu nie musimy wielokrotnie używać tego samego kodu. Podobnie jak C#, PHP, Python i Java, w Dart klasy w programie mogą być ze sobą powiązane. Identyfikacja i ustalenie relacji między nimi jest ważnym aspektem programowania obiektowego (OOP). Dlatego też głównym celem tego rozdziału jest nauczenie się, jak identyfikować relacje między klasami, jak definiować abstrakcyjne klasy i metody oraz jak korzystać z interfejsów. Przyjrzymy się także obsłudze wyjątków, ponieważ mają na nią wpływ relacje między encjami, a także używamy dziedziczenia i interfejsów w ramach wydajnej obsługi błędów

Identyfikowanie relacji między podmiotami

Generalnie naszym wyzwaniem jest stworzenie aplikacji jak najbardziej zbliżonej do świata rzeczywistego. W tym celu w aplikacji łączymy ze sobą klasy i obiekty w taki sposób, że pozostają one luźno powiązane. Działają i reagują z innymi klasami i obiektami. Ta dynamika sprawia, że są one jak najbardziej zbliżone do realnego świata. W OOP obiekty wykonują akcje w odpowiedzi na komunikaty od innych obiektów, określając zachowanie obiektu odbierającego. Istnieją podobieństwa i różnice pomiędzy bytami, obiektami i klasami jako całością. Przyjrzyjmy się następującym obserwacjom:

• Autobus jest rodzajem samochodu.

• Samochód jest rodzajem samochodu.

• Silnik jest częścią samochodu.

• Koło jest częścią samochodu.

• Kierowca prowadzi samochód.

Powyższe encje reprezentują różne obiekty i klasy; mimo to są ze sobą powiązane. Ponadto powinny być luźno połączone; jeśli dotyczy to jednego, nie ma to żadnego wpływu na drugiego (na przykład, jeśli zmienia się konstrukcja kierownicy w samochodzie, relacja z kierowcą nie ulega zmianie). Teraz, w oparciu o poprzednie obserwacje, możemy podsumować nasze relacje między podmiotami w następujący sposób:

• Stosunek dziedziczenia

• Związek kompozycji

• Stosunek wykorzystania

• Relacja instancji

W poprzednim rozdziale widziałeś przykłady dziedziczenia. Można powiedzieć, że samochód to nadklasa samochodu osobowego i autobusu. Na inne ręka, samochód i autobus to podklasy. Wywodzą cechy określone w klasie samochód klasy podstawowej lub superklasy. Mają relację, w której jeden obiekt jest rodzajem innego obiektu, ale sytuacja odwrotna nie jest prawdą. Każdy samochód jest samochodem, ale każdy samochód nie jest samochodem. Przypomnijmy, że Dart umożliwia dziedziczenie pojedyncze i wielopoziomowe. Wielokrotne dziedziczenie w Dart nie jest dozwolone, chociaż można to skompensować za pomocą miksów. Rozważmy inny zestaw relacji.

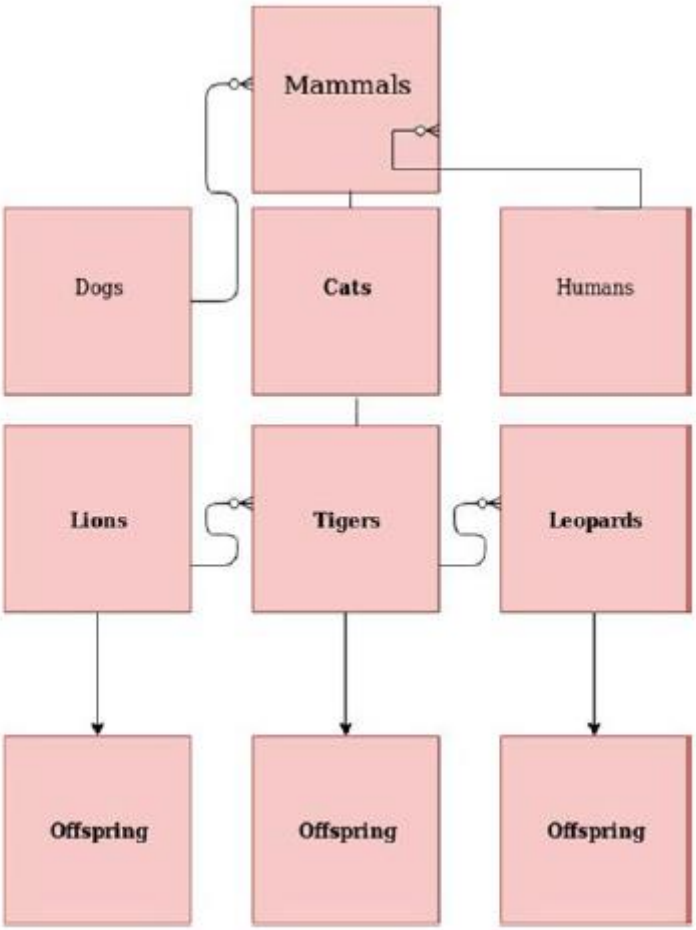
• Człowiek jest rodzajem ssaka.

• Kot jest rodzajem ssaka.

• Tygrys jest rodzajem kota.

• Lew jest rodzajem kota.

Teraz możemy przedstawić tę zależność na rysunku



Na rysunku przedstawiono zbiór klas: ssaki, psy, koty, ludzie, lwy, tygrysy, lamparty i klasy ich potomstwa. Ssaki nadklasowe mają następujący zestaw cech:

• Są stałocieplne.

• Są kręgowcami.

• Wszystkie mają uszy zewnętrzne.

• Wszyscy mają wewnętrzne mózgi pokryte czaszkami.

Można powiedzieć, że pies, kot i ludzie mają podobieństwa; mają podobne cechy, ponieważ odziedziczyli je od ssaków z nadklasy. Jednak podklasa kotów jest nadklasą lwów, tygrysów i lampartów; dlatego będą mieli podobieństwa, których nie mają z psami i ludźmi. Jedną z kluczowych cech programowania obiektowego w Dart jest to, że pozwala nam stworzyć obiekt, którego częścią jest inny obiekt. Ten mechanizm tworzenia obiektu nazywa się relacją kompozycyjną. Mając na uwadze rysunek, możemy stwierdzić, że ludzi i tygrysy nie łączy związek kompozycyjny, podczas gdy koty i tygrysy mają związek kompozycyjny. Nazywa się to kompozycją, ponieważ jedna klasa ma pewne cechy w innych klasach, które są bezpośrednio związane z poprzednią klasą. Przykładami są kot, tygrys, lew itp. Stosunek wykorzystania jest inny. Rozważ rysunek . Człowiek może na przykład używać psa do polowania. Dart pozwala klasie korzystać z innej klasy. Relacja instancji to nic innego jak relacja pomiędzy klasą a jej obiektem lub instancją. Jan jest obiektem klasy ludzkiej. Kiedy tworzymy obiekt John, używamy klasy ssaków jako abstrakcyjnej nadklasy. W następnej sekcji zobaczysz, jak możemy wykorzystać klasy abstrakcyjne.

Korzystanie z klas abstrakcyjnych

Klasa abstrakcyjna służy do zapewnienia częściowej implementacji klasy, pozostawiając niezaimplementowany element podklasie. Metody abstrakcyjne mogą istnieć tylko w klasach abstrakcyjnych. W metodach abstrakcyjnych po prostu zostawiamy średnik (;) na końcu nazwy metody. Nie definiujemy treści metody. W klasie abstrakcyjnej możemy również zdefiniować interfejs, ale jego implementację pozostawiamy innym klasom. Jak powiedziałem na końcu rozdziału 4, interfejs jest kontraktem pomiędzy dwiema klasami. Interfejsem może być dowolna klasa, abstrakcyjna lub konkretna, w Dart. Po prostu znacznie częściej używa się klasy abstrakcyjnej i pozostawia się szczegóły klasie podrzędnej. Oto dwa kluczowe punkty, o których należy pamiętać, pisząc klasę abstrakcyjną:

• Nie można utworzyć instancji klasy abstrakcyjnej.

• Nie można zadeklarować metody abstrakcyjnej poza klasą abstrakcyjną.

```
//code 5.1

//we cannot instantiate any abstract class

abstract class volume{

//we can declare instance variable

int age;

void increase();

void decrease();

// a normal function

void anyNormalFunction(int age){

print("This is a normal function to know the $age.");

}

}

class soundSystem extends volume{

void increase(){

print("Sound is up.");

}

void decrease(){

print("Sound is down.");

}

//it is optional to override the normal function

void anyNormalFunction(int age){

print("This is a normal function to know how old the sound

system is: $age.");

}
```

```
}

main(List<String> arguments){

var newSystem = soundSystem();

newSystem.increase();

newSystem.decrease();

newSystem.anyNormalFunction(10);

}
```

Oto wynik poprzedniego kodu:

Dźwięk jest gotowy.

Dźwięk jest wyłączony.

Jest to normalna funkcja pozwalająca sprawdzić, ile lat ma system dźwiękowy: 10. Użyliśmy modyfikatora abstrakcyjnego, aby zdefiniować klasę abstrakcyjną, której instancji nie można utworzyć. Można więc powiedzieć, że klasa abstrakcyjna i metody podsumowują główne idee i możemy rozszerzyć tę ideę. Jest jeszcze kilka rzeczy, o których należy pamiętać na temat klasy abstrakcyjnej Strzałka.

• W klasie abstrakcyjnej możemy także używać normalnych właściwości i metod.

• Opcjonalne jest zastąpienie metody.

• Możemy także zdefiniować zmienne instancji w klasie abstrakcyjnej.

Rozważ poniższy kod, aby zrozumieć, czym klasy abstrakcyjne w Dart różnią się od innych obiektowych języków programowania:

```
//code 5.2

abstract class Mammal {

void run();

void walk();

void sound(){

print("Mammals make sound");

}

}

class Human implements Mammal {

void run(){

print("I am running.");

}

void walk(){
```

```
print("I am walking");
}
void sound(){
print("Humans make sound");
}
}
main(List<String> arguments){
var John = Human();
print("John says: ");
John.run();
print("John says: ");
John.walk();
print("John makes no sound.");
John.sound();
}
```

Oto wynik, w którym wyraźnie widać, jak zastąpiliśmy metodę abstrakcyjną:

```
//output of code 5.2
/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:
35727 /home/ss/IdeaProjects/bin/main.dart
Observatory listening on http://127.0.0.1:35727/
John says:
I am running.
John says:
I am walking
John makes sound.
Humans make sound
Process finished with exit code 0
```

Zalety interfejsów

W niektórych przypadkach musimy użyć zmiennych referencyjnych i metod wielu klas jednocześnie. Mixiny mogą pomóc. Ale Dart ma jeszcze jedną dobrą funkcję: możemy również użyć interfejsu. Interfejs definiuje umowę syntaktyczną, której powinny przestrzegać wszystkie klasy pochodne. Za

chwilę zobaczysz, jak to działa. najpierw zobaczmy kod, a potem omówimy go szczegółowo. Pamiętaj, że interfejs w Dart jest napisany jako klasa, ale nie rozszerzamy go; wdrażamy to.

```
//code 5.3

// interface in Dart is a class, but we don't extend,

// we implement it

class Vehicle {

void steerTheVehicle() {

print("The vehicle is moving.");

}

}

class Engine {

//in the interface

final _name; // final means single assignment and it must

have an initializer as I use here

//not in the interface, since it is a constructor

Engine(this._name);

String lessOilConsumption(){

return "It consumes less oil.";

}

}

class Car implements Vehicle, Engine{

var _name;

void steerTheVehicle() {

print("The car is moving.");

}

String lessOilConsumption(){

print("This model of car consumes less oil.");

}

void ridingExperience() => print("This car gives good ride,

because it is an ${this._name}");

}
```

```
main(List<String> arguments){

var car = Car();

car._name = "Opel";

print("Car name: ${car._name}");

car.steerTheVehicle();

car.lessOilConsumption();

car.ridingExperience();

}
```

Here is the output of the previous code:

```
Car name: Opel

The car is moving.

This model of car consumes less oil.

This car gives good ride, because it is an Opel
```

Kiedy klasa implementuje interfejs, niejawnie definiuje wszystkie elementy członkowskie instancji zaimplementowanego interfejsu. Klasa implementuje jeden lub więcej interfejsów jednocześnie, deklarując słowo kluczowe implements. Biorąc pod uwagę poprzedni kod, widzimy, że klasa Car obsługuje API klasy Vehicle i klasy Engine, a dla tego wymagania klasa Car implementuje interfejsy klasy Vehicle i klasy Engine. Widać, że obiekt Car może wywoływać metody określone w pojazdach i silnikach, a także własne metody. Interfejsu używamy, gdy potrzebujemy standardowej struktury metod; nie jest konieczne implementowanie elementów interfejsu w dowolnym interfejsie. Rozważ ten kod:

```
//code 5.4

class OrderDetails {

void UpdateCustomers(){

}

void TakeOrder(){

}

}

class ItemDetails implements OrderDetails{

void UpdateCustomers(){

//implementing interface members

print("Updating customers.");

}

void TakeOrder(){
```

```
//implementing interface members
print("Taking orders from customers.");
}
}
main(List<String> arguments){
var book = ItemDetails();
book.TakeOrder();
book.UpdateCustomers();
}
```

Now, look at the output, shown here:

```
//output of code 5.4
/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:
40359 /home/ss/IdeaProjects/bin/main.dart
Observatory listening on http://127.0.0.1:40359/
Taking orders from customers.
Updating customers.
Process finished with exit code 0
```

Co się stanie, jeśli nie będziemy przestrzegać tej standardowej struktury? Kiedy implementujemy interfejs, powinniśmy zaimplementować elementy interfejsu. Następny fragment kodu i wynik wyjaśnią to:

```
//code 5.5
class OrderDetails {
void UpdateCustomers(){
}
void TakeOrder(){
}
}
class ItemDetails implements OrderDetails{
void UpdateCustomers(){
//implementing interface members
print("Updating customers.");
}
```

```
/*
void TakeOrder(){
//implementing interface members
print("Taking orders from customers.");
}
*/
}
main(List<String> arguments){
var book = ItemDetails();
//book.TakeOrder();
book.UpdateCustomers();
}
```

Nie zaimplementowaliśmy elementu interfejsu TakeOrder(). Skomentowaliśmy tę część poprzedniego kodu. W tym przypadku wyjątki zgłoszone w Android Studio i błędy podane jako dane wyjściowe mówią nam, co powinniśmy byli zrobić. Spójrz na dane wyjściowe:

```
//output of code 5.5
/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:
34271 /home/ss/IdeaProjects/bin/main.dart
Observatory listening on http://127.0.0.1:34271/
bin/main.dart:40:7: Error: The non-abstract class 'ItemDetails'
is missing implementations for these members:
- OrderDetails.TakeOrder
Try to either
- provide an implementation,
- inherit an implementation from a superclass or mixin,
- mark the class as abstract, or
- provide a 'noSuchMethod' implementation.
class ItemDetails implements OrderDetails{
^^^^^^^^^^
bin/main.dart:36:8: Context: 'OrderDetails.TakeOrder' is
defined here.
void TakeOrder(){
```

^^^^^^^^^

Process finished with exit code 254

Z poprzedniego wyniku jasno wynika, że Dart wyraźnie zauważa, że nie zaimplementowaliśmy metody, choć powinniśmy. Jeśli istnieje implementacja w klasie abstrakcyjnej, możemy jej użyć podczas rozszerzania tej klasy. Rozważ ten kod:

```
//code 5.6

class OrderDetails {

//int age;

/*

void anyNormalFunction(int age){

print("This is a normal function to know the $age.");

}

*/

void UpdateCustomers(){

}

void TakeOrder(){

}

}

abstract class CustomerDetails {

void Customers(){

print("A list of customers.");

}

}

class ItemDetails extends CustomerDetails implements

OrderDetails {

void anyNormalFunction(int age){

print("This is a normal function to know the age: $age.");

}

void UpdateCustomers(){

//implementing interface members

print("Updating customers.");

}
```

```
void TakeOrder(){
}
}
main(List<String> arguments){
var book = ItemDetails();
//book.TakeOrder();
book.UpdateCustomers();
book.anyNormalFunction(12);
book.Customers();
}
```

W poprzednim kodzie rozszerzyliśmy klasę abstrakcyjną i jednocześnie zaimplementowaliśmy interfejs. Dane wyjściowe są tutaj:

```
//output of code 5.6
/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:
39205 /home/ss/IdeaProjects/bin/main.dart
Observatory listening on http://127.0.0.1:39205/
Updating customers.
This is a normal function to know the age: 12.
A list of customers.
Process finished with exit code 0
```

Jak widać, metoda Customers() klasy abstrakcyjnej jest wywoływana, gdy sami jej nie implementujemy. Istnieje jeszcze jedna istotna różnica pomiędzy klasą abstrakcyjną a interfejsem. Klasa abstrakcyjna może używać normalnych właściwości i metod. Jednakże, jeśli nie zaimplementujemy żadnej części interfejsu, innymi słowy, jeśli opuścimy interfejs, aby zachować własną implementację właściwości lub metody, wystąpią błędy. Spójrz na ten kod i jego wynik:

```
//code 5.7
class OrderDetails {
int age;
void anyNormalFunction(int age){
print("This is a normal function to know the $age.");
}
void UpdateCustomers(){
}
```

```
void TakeOrder(){
}
}
abstract class CustomerDetails {
void Customers(){
}
}
class ItemDetails extends CustomerDetails implements
OrderDetails {
//trying to implement interface normal functions
void anyNormalFunction(int age){
print("This is a normal function to know the age: $age.");
}
void UpdateCustomers(){
//implementing interface members
print("Updating customers.");
}
void TakeOrder(){
}
void Customers(){
}
}
main(List<String> arguments){
var book = ItemDetails();
//book.TakeOrder();
book.UpdateCustomers();
book.anyNormalFunction(12);
}
```

Oto raport o błędach:

```
//output of code 5.7
/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:
```

38747 /home/ss/IdeaProjects/bin/main.dart

Observatory listening on http://127.0.0.1:38747/

bin/main.dart:50:7: Error: The non-abstract class 'ItemDetails'

is missing implementations for these members:

- OrderDetails.age

Try to either

- provide an implementation,

- inherit an implementation from a superclass or mixin,

- mark the class as abstract, or

- provide a 'noSuchMethod' implementation

class ItemDetails extends CustomerDetails implements

OrderDetails {

^^^^^^^^^^

bin/main.dart:34:7: Context: 'OrderDetails.age' is defined here.

int age;

^^^

Process finished with exit code 254

Dlatego oto kilka rzeczy do zapamiętania na temat interfejsów w Dart:

• Największą zaletą interfejsów jest to, że możemy wdrożyć wiele interfejsów. Ponieważ w Dart wielokrotne dziedziczenie nie jest dozwolone, możemy zaprojektować naszą aplikację w taki sposób, aby naśladować dziedziczenie wielu klas za pomocą interfejsów. Nie możemy jednak używać żadnych normalnych właściwości i zachowań w interfejsach.

• Chociaż nie możemy dziedziczyć wielu klas poprzez dziedziczenie, możemy pokonać to ograniczenie, łącząc klasy abstrakcyjne, interfejsy i miksy.

Zmienne statyczne i metody

Aby zaimplementować zmienne i metody dla całej klasy, używamy słowa kluczowego static. Zmienne statyczne nazywane są także zmiennymi klasowymi. Najpierw zobaczmy fragment kodu, a następnie omówimy zalety i wady zmiennych i metod statycznych.

```
//code 5.8

// static variables and methods consume less memory

// they are lazily initialized

class Circle{
```

```
static const pi = 3.14;
static void drawACircle(){
//from static method you cannot call a normal function
print(pi);
}
void aNonStaticFunction(){
//from a normal function or method you can call a static meethod
Circle.drawACircle();
print("This is normal function.");
}
}
main(List<String> arguments){
var circle = Circle();
circle.aNonStaticFunction();
Circle.drawACircle();
}
```

Here is the output:

3.14

This is normal function.

3.14

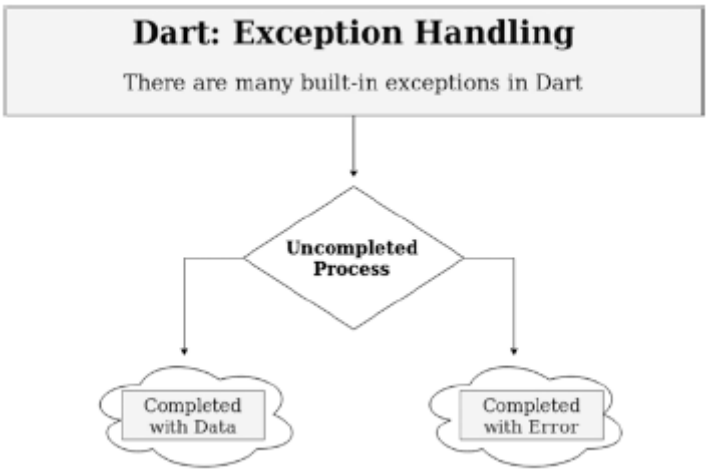
Jak widać, zmienne statyczne są przydatne w przypadku stanów i stałych obejmujących całą klasę. Zatem w metodzie main() możemy na końcu dodać następującą linię:

```
main(List<String> arguments){
var circle = Circle();
circle.aNonStaticFunction();
Circle.drawACircle();
print(Circle.pi);
}
```

Obsługa wyjątków

Podczas wykonywania dowolnego programu mogą wystąpić błędy, które automatycznie zakłócają działanie programu. Błędy te nazywane są wyjątkami. W przypadkach obsługi wyjątków klasa Wyjątek

jest nadklasą wszystkich wyjątków, aby zapobiec nagłemu zakończeniu działania aplikacji. Rysunek ilustruje tę koncepcję, w której proces obliczeniowy ma dwa możliwe wyniki.



Jednym z nich mogą być przetworzone dane, które chcieliśmy, a innym może być błąd. Powinniśmy pozwolić sobie na wyłapanie tego błędu, zanim spowoduje on brzydki wyjątek w interfejsie użytkownika. Załóżmy, że chcesz podzielić liczbę przez zero. Jest to zadanie niemożliwe i zakłóci przepływ, powodując pewne błędy. Nie możesz jednak kontrolować zachowania użytkownika, dlatego musisz podjąć wszelkie środki ostrożności, aby sprawnie obsługiwać błędy. Programiści Darta pomyśleli o tym i umieścili wiele wbudowanych klas wyjątków. Jednym z nich jest wyjątek IntegerDivisionByZeroException; jest wyrzucany, gdy liczba jest dzielona przez zero. Podobnie, gdy podczas oczekiwania na wynik asynchroniczny nastąpi zaplanowane przekroczenie limitu czasu, wystąpi wyjątek przekroczenia limitu czasu. Jeśli załadowanie odroczonych bibliotek nie powiedzie się, wystąpi wyjątek DeferredLoadException. Załóżmy, że nie można przeanalizować ciągu znaków, ponieważ nie ma on odpowiedniego formatu. W takim przypadku wystąpi wyjątek FormatException. Wszelkie wyjątki związane z danymi wejściowymi i wyjściowymi są przechwytywane za pośrednictwem klasy IOException. Zobaczmy kilka fragmentów kodu, abyśmy mogli łatwo zrozumieć, w jaki sposób możemy wychwycić wyjątki.

```
//code 5.9

main(List<String> arguments){

try{

int result = 10 ~/ 0;

print("The result is $result");

} on IntegerDivisionByZeroException{

print("We cannot divide by zero");

}

try{
```

```
int result = 10 ~/ 0;
print("The result is $result");
} catch(e){
print(e);
}
try{
int result = 10 ~/ 0;
print("The result is $result");
} catch(e){
print("The exception is : $e");
} finally{
print("This is finally and it always is executed.");
}
}
```

Wyłapaliśmy te błędy, zanim dały użytkownikowi brzydkie wyniki.

```
//the output of code 5.9
We cannot divide by zero
IntegerDivisionByZeroException
The exception is : IntegerDivisionByZeroException
This is finally and it always is executed.
```

Jak widać w wynikach, istnieje kilka metod, dzięki którym możemy wyłapać wyjątki. Jeśli znamy typ wyjątku, możemy użyć funkcji try/on, tak jak zastosowaliśmy to w poprzednim kodzie:

```
try{
int result = 10 ~/ 0;
print("The result is $result");
} on IntegerDivisionByZeroException{
print("We cannot divide by zero");
}
```

W tym przypadku wiedzieliśmy, jaki typ wyjątku można wygenerować. Użyliśmy więc try/on. Ale co się stanie, gdy nie znamy wyjątku? W większości przypadków prawdopodobnie początkujący nie będzie znał wszystkich klas wyjątków, które są predefiniowane w bibliotekach Dart. Warto jednak poznać kilka, o których wspomniałem wcześniej. Poza tym głównym powodem zawinięcia naszego kodu w blok

try/catch jest to, że w naszym kodzie mogą występować błędy. Nasz kod może zawierać problemy. Jako programista nie powinniśmy podejmować żadnego ryzyka. Składnia obsługi wyjątku jest następująca:

```
try{

int result = 10 ~/ 0;

print("The result is $result");

} catch(e){

print(e);

}
```

Blok catch jest używany, gdy procedura obsługi potrzebuje obiektu wyjątku.

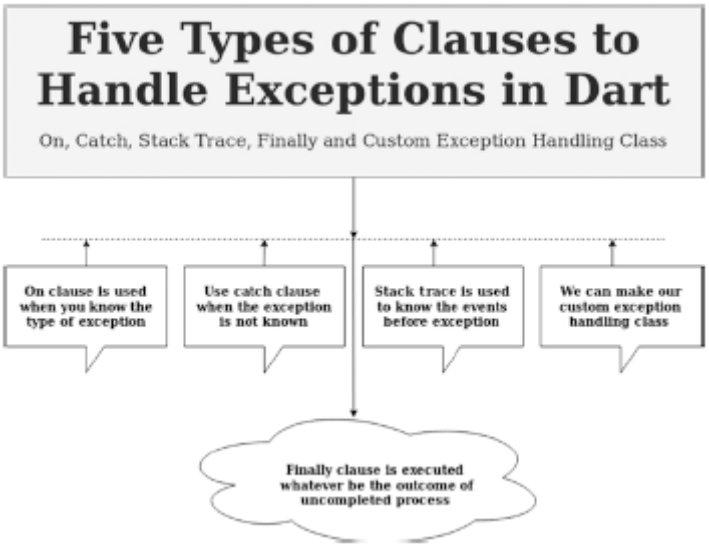
Po bloku try może nastąpić blok last po bloku catch. Użyliśmy tego samego w następującym poprzednim kodzie:

```
try{

int result = 10 ~/ 0;

print("The result is $result");

} catch(e){

print("The exception is : $e");

} finally{

print("This is finally and it always is executed.");

}
```

Blok Final zostanie wykonany na końcu, niezależnie od wyniku:

Wyjątkiem jest: IntegerDivisionByZeroException. Jest to ostatecznie i zawsze wykonywane

Jeśli w bloku try wystąpi wyjątek, sterowanie przechodzi do bloku catch; i na koniec blok last daje wynik. Możemy teraz zakończyć tę sekcję rysunkiem 5.3, który przedstawia, ile typów obsługi wyjątków jest używanych w Dart za pomocą klas wyjątków. Na rysunku znajdziesz termin „śledzenie stosu”.



Kiedy program jest uruchamiany, pamięć jest przydzielana w dwóch miejscach: na stosie i na stercie. Jeśli w naszym kodzie wystąpi problem, przed alokacją pamięci uruchamiane są pewne zdarzenia, które można prześledzić na stosie. Mówiąc najprościej, ślad stosu to lista wywołań metod, w trakcie których aplikacja znajdowała się, gdy został zgłoszony wyjątek. Poszukamy najlepszej metody i będziemy wiedzieć, gdzie występują błędy. W Dart będziesz musiał przeczytać raport śledzenia stosu; Jestem pewien, że dowiesz się wielu rzeczy na temat sposobu uruchamiania programu. Dodatkowo możemy stworzyć własną klasę obsługi wyjątków, która wyłapie błąd. Dlaczego tego potrzebujemy? Możesz zwiększyć elastyczność swojego kodu, budując niestandardową obsługę wyjątków, aby nadać wyjątkom bardziej przydatne nazwy lub przykład. Jednak jako początkujący nie będę sugerował, abyś od razu zawracał sobie głowę tworzeniem niestandardowych klas wyjątków. Możemy też w końcu całkowicie zakończyć zadanie z jedną bazą kodu, w której przejdziemy przez wszystkie klauzule obsługi wyjątków.

```
//code 5.10

class InputException implements Exception {

String customException() {

return "The input of negative number is not valid.";

}

}

void main() {

// ON Clause is used when the exception is known

try {

var res = 4 ~/ 0;

print("The result: $res");

} on IntegerDivisionByZeroException {

print("You cannot divide by zero, the value is undefined");
```

```
}
// CATCH Clause is used when exception is unknown
try {
var res = 3 ~/ 0;
print("The result is $res");
} catch (e) {
print("The exception thrown is $e");
}
// STACK TRACE is used to know the steps of the events
// these events took place before the actual Exception was
thrown
try {
int res = 10 ~/ 0;
print("The result is $res");
} catch (e, s) {
print("The exception: $e");
print("Stack trace is \n $s");
}
// FINALLY Clause is always Executed
// whether exception is thrown or not
try {
int res = 9 ~/ 0;
print("The result: $res");
} catch (e) {
print("The exception: $e");
} finally {
print("The finally clause is always executed.");
}
// we can make our Custom Exception by creating a class
try {
inputValue(-14);
```

```
} catch (e) {
print(e.customException());
} finally {
print("The finally clause is always executed");
}
}
void inputValue(int inputNumber) {
if (inputNumber < 0) {
var inputException = InputException();
throw inputException;
}
}
```

Zwróć uwagę na użycie słowa kluczowego rzut w funkcji inputValue(). Spowoduje to zgłoszenie określonego wyjątku i przekazanie kontroli z powrotem do kodu wywołującego. Blok try/catch może następnie obsłużyć ten zgłoszony wyjątek. Teraz możemy przyjrzeć się wynikom, aby zobaczyć ślad stosu:

```
//output of code 5.10
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
--enable-asserts --enable-vm-service:42201 /home/ss/
IdeaProjects/my_app/main.dart
Observatory listening on http://127.0.0.1:42201/eUtY0DGP6ro=/
You cannot divide by zero, the value is undefined
The exception thrown is IntegerDivisionByZeroException
The exception: IntegerDivisionByZeroException
Stack trace is
#0 int.~/ (dart:core-patch/integers.dart:18:7)
#1 main (
file:///home/ss/IdeaProjects/my_app/main.
dart:24:18)
#2 _startIsolate.<anonymous closure> (dart:isolate-patch/
isolate_patch.dart:301:19)
#3 _RawReceivePortImpl._handleMessage (dart:isolate-patch/
```

isolate_patch.dart:172:12)

The exception: IntegerDivisionByZeroException

The finally clause is always executed.

The input of negative number is not valid.

The finally clause is always executed

Process finished with exit code 0

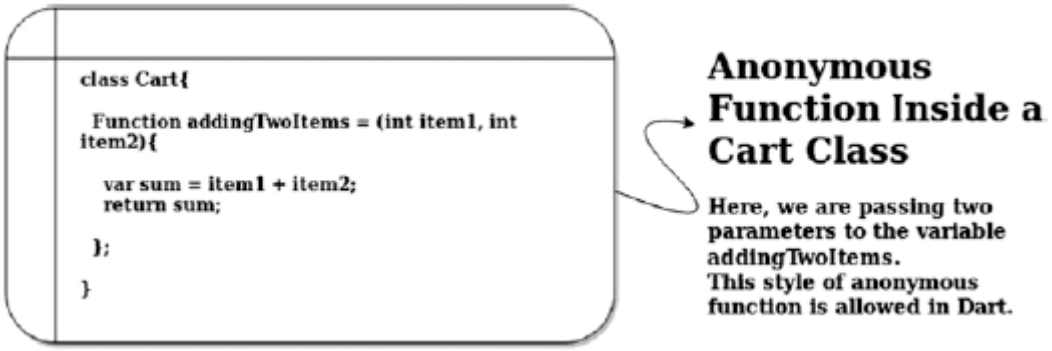
Teraz całkowicie zależy od programisty, jak obsłuży wyjątek. Jedyne, o czym powinniśmy pamiętać, to to, że użytkownikowi nie będzie się podobać, jeśli kod zgłosi wyjątek. Dlatego przed publikacją obowiązkowe jest przejście testu, a w razie potrzeby zawsze lepiej jest zastosować mechanizm obsługi wyjątków.

Funkcje anonimowe

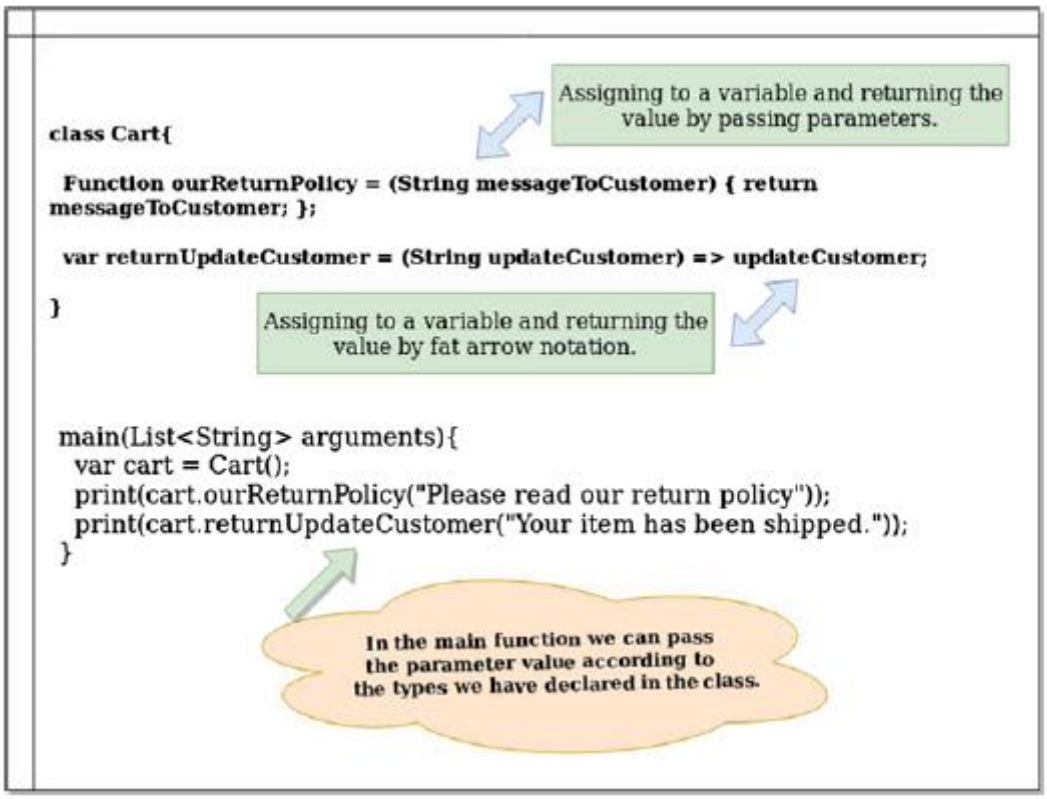
W Dart większość funkcji, które do tej pory widzieliśmy, to funkcje nazwane, które są podobne do funkcji w językach takich jak C# i Java. Mimo to składnia funkcji Dart ma więcej podobieństw do JavaScriptu niż w wielu językach o silnie typowanych typach, takich jak C# czy Java. Ponieważ w Darcie wszystko jest obiektem, obiektem jest także funkcja; oznacza to, że możemy przechowywać go w zmiennej i używać go w dowolnym miejscu naszej aplikacji. Zaletą Darta jest to, że możemy przekazać funkcję jak każdy inny typ, taki jak ciąg znaków, liczba całkowita itp. Funkcje te są znacznie ulepszone, gdy funkcje nie mają w ogóle nazw. Te bezimienne funkcje działają tak samo jak funkcje nazwane; mogą mieć dowolną liczbę parametrów, w tym parametry zerowe. Adnotacje typu są opcjonalne. Funkcje te nazywane są funkcjami anonimowymi. Podobnie jak funkcje nazwane, możemy przypisać dowolną funkcję anonimową do zmiennej obiektu funkcji. Możemy również przekazać go do innej funkcji. Lambdy, funkcje wyższego rzędu i domknięcia leksykalne są funkcjami anonimowymi i mają pewne podobieństwa. Te cechy Darta są bardzo interesujące w swojej bezimiennej i anonimowości. Zacznijmy od lambd. Następnie omówimy funkcje i domknięcia wyższego rzędu. W rzeczywistości przekonasz się, że lambdy faktycznie implementują funkcje wyższego rzędu.

Pierwsze spojrzenie na Lambdy

Rysunek pokazuje, jak możemy używać lambd, jednego z typów funkcji anonimowych.



Na rysunku dłuższa wersja funkcji anonimowej wymaga średnika kończącego; dzieje się tak, ponieważ przypisujemy wartość do zmiennej o nazwie dodanieTwoItems. Dodatkowo w wersji długiej możemy używać notacji grubej strzałk. Rysunek pokazuje dwa typy funkcji anonimowych i sposób, w jaki możemy ich używać w naszej aplikacji. Kod zobaczymy również za minutę.



Na rysunku wspomnieliśmy o typie parametrów. Na wypadek, gdybyśmy o tym nie wspomnieli, Dart przydziela je dynamicznie. Przyjrzyjmy się kodowi z rysunku wcześniejszego i zobaczmy wyniki, aby zrozumieć tę koncepcję.

```
//code 6.1

class Cart{

Function addingTwoItems = (int item1, int item2){

var sum = item1 + item2;

return sum;

};

}

main(List<String> arguments){

var cart = Cart();

print("Your total price is:");

print(cart.addingTwoItems(120, 458));

}
```

Here is the output of this code:

```
//output of code 6.1

Your total price is:
```

Kod użyty na rysunku 6-2 pokazano poniżej. Zastosowaliśmy dwie metody deklarowania funkcji anonimowych: długą i skróconą.

```
//code 6.2

class Cart{

Function ourReturnPolicy = (String messageToCustomer) {

return messageToCustomer;

};

var returnUpdateCustomer = (String updateCustomer) =>

updateCustomer;

}

main(List<String> arguments){

var cart = Cart();

print(cart.ourReturnPolicy("Please read our return policy"));

print(cart.returnUpdateCustomer("Your item has been

shipped."));

}
```

Dane wyjściowe są dość proste. Do każdej funkcji przekazaliśmy jeden parametr i otrzymaliśmy wynik w postaci ciągu znaków:

```
//wyjście kodu 6.2

Prosimy o zapoznanie się z naszą polityką zwrotów

Twój przedmiot został wysłany.
```

Tutaj podsumowujemy kluczowe cechy funkcji anonimowych:

• Możemy zadeklarować dowolną funkcję anonimową bez nazwy funkcji.

• Możemy przypisać go do zmiennej.

• Funkcję anonimową można przekazać do innej funkcji, jak zobaczymy później.

• W wersji długiej musimy zakończyć instrukcję średnikiem, ponieważ przypisujemy ją do zmiennej.

• Jedyną wadą funkcji anonimowej jest to, że nie można jej używać rekurencyjnie, ponieważ nie ma ona nazwy.

Odkrywanie funkcji wyższego rzędu

Specjalnością funkcji wyższego rzędu jest to, że mogą one przyjmować funkcję jako parametr. Dlatego nazywa się je funkcjami wyższego rzędu. Nie tylko mogą zaakceptować funkcję jako parametr; mogą

go również zwrócić. Aby przyzwyczaić się do tego pomysłu, spójrzmy na następujący prosty fragment kodu:

```
//code 6.3

//returning a function

Function DividingByFour(){

Function LetUsDivide = (int x) => x ~/ 4;

return LetUsDivide;

}

main(List<String> arguments){

var result = DividingByFour();

print(result(56));

}
```

The output is 14.

Zatem dość łatwo zwracamy funkcję LetUsDivide() poprzez funkcję wyższego rzędu o nazwie Function DividingByFour(). Zobaczmy więcej przykładów, aby zrozumieć, jak działają funkcje anonimowe. Spójrz na tę linię w poprzednim kodzie:

```
Function LetUsDivide = (int x) => x ~/ 4;
```

Funkcja LetUsDivide() jest przypisana do anonimowej funkcji o nazwie (int x). Następnie użyliśmy notacji grubej strzałki, aby zwrócić wartość. Główną zaletą jest to, że możemy przechowywać funkcje w zmiennej lub odwoływać się do nich nazwą zmiennej. Posiadanie zmiennej zawierającej obiekt funkcyjny daje nam swobodę przekazywania jej po aplikacji, jak każdą inną zmienną. Co więcej, możemy zwrócić obiekt funkcji przechowywany w zmiennej lub przekazać go do innej funkcji, gdzie będziemy mogli go wywołać jako dowolną zadeklarowaną funkcję. W następnej sekcji zobaczymy, jak koncepcja zamknięć zmienia się w zależności od sytuacji.

Zamknięcie jest funkcją specjalną

Zamknięcie możemy zdefiniować na dwa sposoby.

• Można powiedzieć, że zamknięcie jest jedyną funkcją, która ma dostęp do zakresu nadrzędnego, nawet po zamknięciu zakresu.

• Termin zamknięcie pochodzi od terminu zamknięcie. Ponieważ otacza dowolną nielokalną zmienną, która była ważna w momencie deklaracji, w rzeczywistości zamyka tę zmienną.

Można to powiedzieć, gdy jedna funkcja zwraca inną funkcję

powstają zamknięcia; to samo dzieje się w funkcjach wyższego rzędu. Następny przykład wyjaśni tę koncepcję. Aby zrozumieć tę definicję, spójrzmy na następujący krótki fragment kodu, w którym zamknięcie funkcji anonimowej zastępuje zakres nadrzędny:

```
//code 6.4

//a closure can modify the parent scope
```

```
String message = "Any Parent String";

Function overridingParentScope = (){

String message = "Overriding the parent scope";

print(message);

};

main(List<String> arguments){

print(message);

overridingParentScope();

}
```

Dane wyjściowe są następujące:

```
Any Parent String

Overriding the parent scope
```

W drugiej definicji możemy powiedzieć, że domknięcie to obiekt funkcyjny, który ma dostęp do zmiennych w swoim zakresie leksykalnym, nawet jeśli funkcja jest używana poza jej pierwotnym zakresem.

```
//code 6.5

//declaring an anonymous function without any parameter

Function show = (){

Function gettingImage(){

String path = "This is a new path to image.";

print(path);

}

return gettingImage;

};

main(List<String> arguments){

String path = "This is an old path.";

var showing = show();

showing();

}
```

Oto dane wyjściowe:

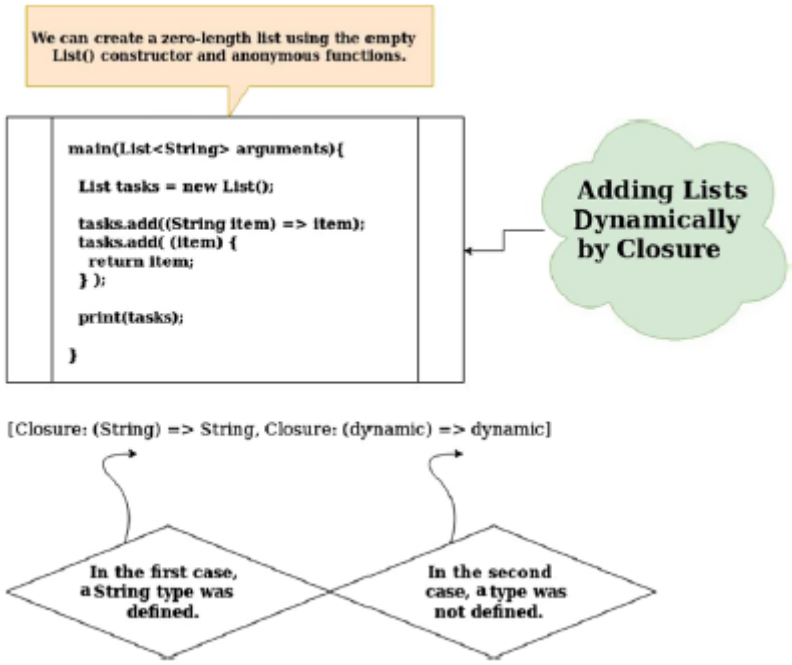
```
This is a new path to image.
```

Ten kod faktycznie zwraca obiekt funkcji o nazwie getImage, który uzyskał dostęp do zmiennej w jej zakresie leksykalnym. Dart jest językiem o ograniczonym zakresie leksykalnym, co oznacza, że najpierw przeszukiwany jest najbardziej wewnętrzny zakres. Na koniec tej sekcji możemy podsumować kilka punktów dotyczących zamknięć.

• W kilku innych językach, w tym w Pythonie i PHP, nie wolno modyfikować zmiennej nadrzędnej.

• Jednakże w obrębie zamknięcia można mutować lub modyfikować wartości zmiennych występujących w zakresie nadrzędnym.

Rysunek pokazuje nieco więcej szczegółów na temat zamknięć (w szczególności ich typów w momencie ich tworzenia).



Połączenie tego wszystkiego

Teraz zakończymy naszą podróż polegającą na badaniu funkcji bezimiennych lub anonimowych w jednej bazie kodu i przyjrzymy się również wynikom. W poniższym fragmencie kodu próbowaliśmy dać Ci pojęcie o wszystkich typach funkcji anonimowych. Przeczytaj komentarze w kodzie, aby zobaczyć, jakich typów anonimowych funkcji tutaj używamy.

```
//code 6.6

//Lambda is an anonymous function

class AboutLambdas{

//first way of expressing Lambda or anonymous function

Function addingNumbers = (int a, int b){

var sum = a + b;

return sum;

};
```

```
Function multiplyWithEight = (int num){
return num * 8;
};
//second way of expressing Lambda by Fat Arrow
Function showName = (String name) => name;
//higher order functions pass function as parameter
int higherOrderFunction(Function myFunction){
int a = 10;
int b = 20;
print(myFunction(a, b));
}
//returning a function
Function returningAFunction(){
Function showAge = (int age) => age;
return showAge;
}
//a closure can modify the parent scope
String anyString = "Any Parent String";
Function overridingParentScope = (){
String message = "Overriding the parent scope";
print(message);
};
Function show = (){
// the anonymous function will return this originally
Function gettingImage(){ // anonymous function returns a
function
String path = "This is a new path to image.";
print(path);
}
return gettingImage;
};
```

```
}
main(List<String> arguments){
var add = AboutLambdas();
var addition = add.addingNumbers(5, 10);
print(addition);
var mul = AboutLambdas();
var result = mul.multiplyWithEight(4);
print(result);
var name = AboutLambdas();
var myName = name.showName("Sanjib");
print(myName);
var higher = AboutLambdas();
var higherOrder = higher.higherOrderFunction(add.addingNumbers);
higherOrder;
var showAge = AboutLambdas();
var showingAge = showAge.returningAFunction();
print(showingAge(25));
var sayMessage = AboutLambdas();
sayMessage.overridingParentScope();
var image = AboutLambdas();
String path = "This is an old path.";
var imagePath = image.show();
imagePath();
}
```

The output shows how the nameless functions work.

```
//output of code 8.6
15
32
Sanjib
30
25
```

Overriding the parent scope

This is a new path to image.

Struktury danych i kolekcje

Zrozumienie koncepcji struktur danych i kolekcji jako całości odgrywa kluczową rolę w przyszłym programowaniu Dart. Za chwilę dowiesz się, że w Dart istnieją cztery typy struktur danych.

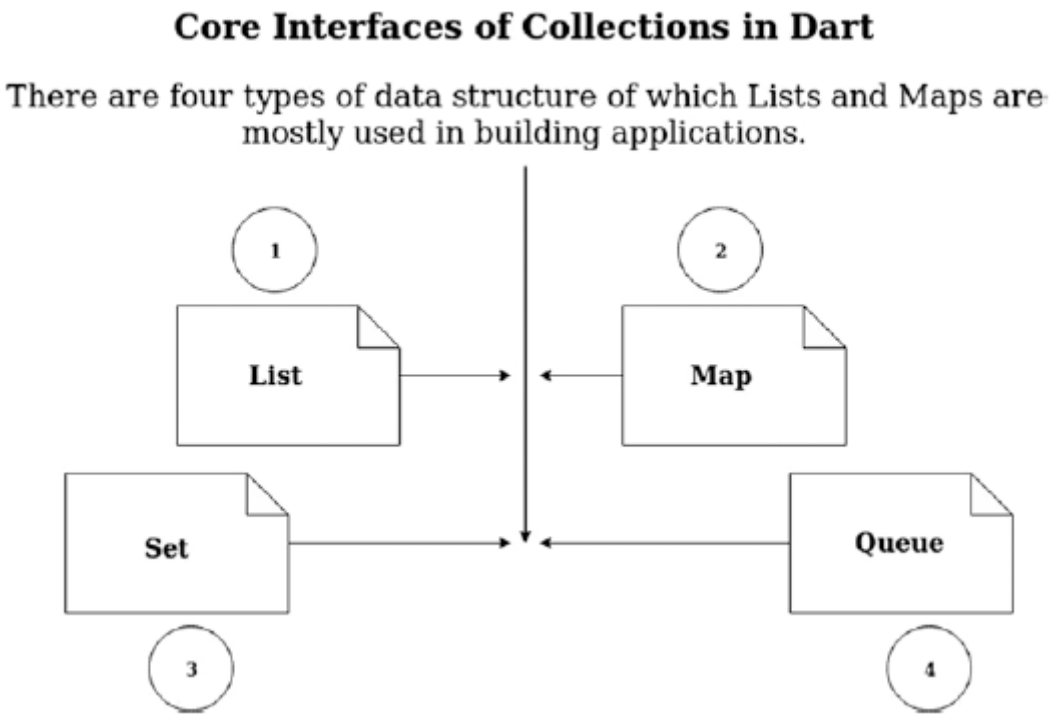
• Lista

• Ustawić

• Mapa

• Kolejka

Moim zdaniem Listy i Mapy obejmują prawie wszystko, więc pozostałe dwa typy prawie nie są potrzebne w życiu programistycznym, z wyjątkiem kilku okazji. Sugeruję jednak, aby nie ignorować nauki o ustawianiu i kolejce; w kilku przypadkach mają nieobliczalną wartość. Rysunek pokazuje, jakiego rodzaju kolekcji będziemy używać.



O tych strukturach danych dowiesz się w tym rozdziale. Omówimy szczegółowo wszystkie koncepcje kolekcji Dart. Przenoszenie/przetwarzanie zbiorów danych w sposób bezpieczny dla typu jest zawsze naszym priorytetem przy tworzeniu oprogramowania. Aby to zrobić, najpierw musimy mieć podstawową wiedzę na temat organizowania dużej porcji danych w celu późniejszego pobrania. Krótko mówiąc, struktury danych pomagają organizować informacje do przechowywania i późniejszego wyszukiwania. Korzystanie z klas kolekcji wbudowanych w Dart jest dla nas dużą zaletą. Zarówno Lista, jak i Mapa należą do tej kategorii. Pozwalają nam manipulować listami danych i umożliwiają dostęp do zbiorów danych w sposób bezpieczny dla typu; ponadto możemy skorzystać z dodatkowych walidacji

przeprowadzanych przez moduł sprawdzania typów dostarczony przez Dart. Nie tylko to, wbudowane narzędzia dostarczane przez Dart pomagają nam uzyskać dostęp do elementów bezpośrednio na listach i mapach; dowiesz się także, jak tworzyć mapy i listy na podstawie istniejących wartości. Zacznijmy więc od list.

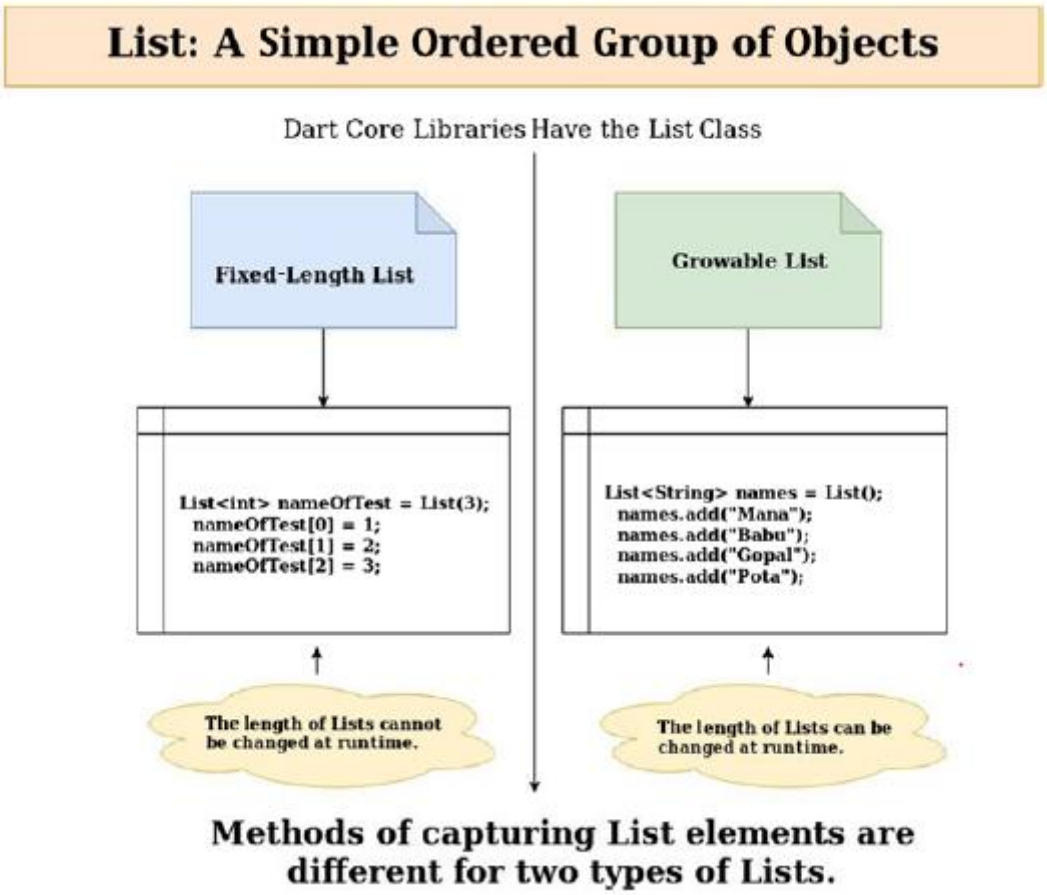
Listy: uporządkowana kolekcja

Lista to prosta uporządkowana grupa obiektów. Tworzenie listy wydaje się łatwe, ponieważ podstawowe biblioteki Dart mają niezbędne wsparcie i klasę List. Istnieją dwa typy list.

• Lista o stałej długości

• Lista upraw

Na liście o stałej długości długość listy nie może się zmienić w czasie wykonywania; jednakże w drugim typie, liście rozwijalnej, długość może zmieniać się w czasie wykonywania. Rysunekopisuje działanie tych list.



W następnych przykładach przyjrzymy się oddzielnie dwóm typom list. Zobaczymy również, jak działają. Najpierw po prostu utwórz listę, jak pokazano tutaj

```
void main() {

var lst = new List();

lst.add(3);
```

```
lst.add(4);
print(lst);
}
//output
[3, 4]
```

Now here’s our first example:

```
//code 7.1
int listFunction(){
List<int> nameOfTest = List(3);
nameOfTest[0] = 1;
nameOfTest[1] = 2;
nameOfTest[2] = 3;
//there are three methods to capture the list
//1. method
for(int element in nameOfTest){
print(element);
}
print("-----------");
//2. method
nameOfTest.forEach((v) => print('${v}'));
print("-----------");
//3. method
for(int i = 0; i < nameOfTest.length; i++){
print(nameOfTest[i]);
}
}
main(List<String> arguments){
listFunction();
}
```

Jak widać, jest to uporządkowana lista trzech liczb. Dane wyjściowe uzyskujemy za pomocą trzech metod, z których każda jest prosta.

```
//output of code 7.1
1
2
3
-----------
1
2
3
-----------
1
2
3
```

Następny przykład to lista rozwijalna, pokazana tutaj:

```
//code 7.2
Function growableList(){
//1. method
List<String> names = List();
names.add("Mana");
names.add("Babu");
names.add("Gopal");
names.add("Pota");
//there are two methods to capture the list
print("-----------");
//1. method
names.forEach((v) => print('${v}'));
print("-----------");
//2. method
for(int i = 0; i < names.length; i++){
print(names[i]);
}
}
```

```
main(List<String> arguments){

growableList();

}
```

To również jest proste; nie przekazaliśmy żadnej liczby przez List(), co pozwala nam dodać do niej dowolną liczbę elementów. Tutaj dodaliśmy kilka nazwisk. Elementy List możemy przechwycić dwiema metodami, a nie trzema. Wynik jest całkiem oczekiwany, jak pokazano tutaj:

```
//output of code 7.2

-----------

Mana

Babu

Gopal

Pota

-----------

Mana

Babu

Gopal

Pota
```

Z danych wyjściowych i kodu jasno wynika, że listy rozwijalne mają charakter dynamiczny. Możemy dynamicznie dodawać dowolną liczbę elementów, ale możemy też je usuwać prostą names.remove("any name"). Możemy także użyć klucza; zwróć uwagę, że ta uporządkowana lista zaczyna się od 0. Możemy więc usunąć imię po prostu przekazując tę wartość klucza: nazwy. usuńAt(0). Używamy metody usuwaniaAt(klucz) dla tej operacji.Możemy także wyczyścić Listy po prostu wpisującnames.clear().Rozważmy inną listę kodu, w której użyliśmy wielu domyślnych metod klasy List.

```
//code 7.3

main(){

var number1 = 1;

var number2 = 1;

while(number2 < 50){

print(number2);

number2 += number1;

number1 = number2 - number1;

}

print("Separator line: =============");

var fibonacciNumbers = [1, 2, 3, 5, 8, 13, 21, 34];
```

```
print(fibonacciNumbers.take(3).toList());
print("Separator line: =============");
print(fibonacciNumbers.skip(5).toList());
print("Separator line: =============");
print(fibonacciNumbers.skip(2).contains(5));
print("Separator line: =============");
print(fibonacciNumbers.take(3).skip(2).take(1).toList());
print("Separator line: =============");
var clonedFibonacciNumbers = List.from(fibonacciNumbers);
print("Cloned list: $clonedFibonacciNumbers");
}
```

Najpierw sprawdźmy dane wyjściowe.

First, let’s check the output.//output of code 7.3

```
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
--enable-asserts --enable-vm-service:33845 /home/ss/
IdeaProjects/my_app/main.dart
Observatory listening on http://127.0.0.1:33845/6uljPm-VaFM=/
1
2
3
5
8
13
21
34
Separator line: =============
[1, 2, 3]
Separator line: =============
[13, 21, 34]
Separator line: =============
true
```

Separator line: ==========

[3]

Separator line: ==========

Cloned list: [1, 2, 3, 5, 8, 13, 21, 34]

Process finished with exit code 0

Linia oddzielająca użyta pomiędzy wynikami pomaga pokazać, jak działają inne domyślne metody klasy List. Użyliśmy kilku metod domyślnych, takich jak toList(), zawiera(), skip() itp. Z klasy List możemy pobrać dowolną liczbę elementów. Możemy wyeliminować dwie pierwsze liczby i wydrukować inne wartości. Istnieją również inne metody List, które są niezwykle elastyczne i pozwalają na dodanie specjalnych słów kluczowych do Listy. Rozważmy najpierw ten mały przykład:

```
//code 7.4

main(){

var names = ["John", "Robert", "Smith", "Peter"];

names.forEach((name) => print(name));

}
```

It will give us some nice output of names.

```
//output of code 7.4

John

Robert

Smith

Peter
```

Czy możemy dodać dodatkową funkcjonalność do tej listy nazw, aby miały ze sobą coś wspólnego? Załóżmy, że każde nazwisko, które wymieniliśmy, ucieka. Każde imię musimy poprzedzić słowem Absconding. Funkcja map() daje nam możliwość utworzenia nowej listy poprzez jednoczesną transformację każdego elementu. Teraz poprzedni kod zmienia się na następujący:

```
//code 7.5

main(){

var names = ["John", "Robert", "Smith", "Peter"];

names.forEach((name) => print(name));

var mappedNames = names.map((name) => "Absconding $name").

toList();

print(mappedNames);

}
```

The output changes to this:

//output of code 7.5

John

Robert

Smith

Peter

[Absconding John, Absconding Robert, Absconding Smith,

Absconding Peter]

Pomyślnie zmapowaliśmy każdy element i przekształciliśmy go, umieszczając wyniki na nowej liście. Aby uzyskać więcej informacji na temat klasy List, możesz przejść do repozytoriów języka Dart na Listach:

https://api.dartlang.org/dev/2.0.0-dev.65.0/dart-core/List-class.html

Zestaw: Nieuporządkowane kolekcje unikalnych przedmiotów

Nagłówek mówi wszystko. Zestaw reprezentuje zbiór obiektów, w którym każdy obiekt może wystąpić tylko raz. W podstawowej bibliotece Dart znajduje się klasa Set z tą funkcjonalnością. Ponieważ Set jest nieuporządkowaną kolekcją unikalnych elementów, nie można uzyskać elementów według indeksu. Istnieje koncepcja zwana HashSet, która faktycznie implementuje nieuporządkowany zestaw i opiera się na implementacji zestawu opartej na tablicy mieszającej. Za chwilę przyjrzymy się tym funkcjom. Zestawy możemy tworzyć na dwa sposoby.

Set <type> set name = {};

var setname = <Type> {};

//code 7.6

```
void setFunction(){

//set is an unordered collections of unique items

//cannot get elements by INDEX since the items are unordered

//1. method of creating Set

Set<String> countries = Set.from(['India', 'England', 'US']);

Set<int> numbers = Set.from([1, 45, 58]);

Set<int> moreNumbers = Set();

moreNumbers.add(178);

moreNumbers.add(568);

moreNumbers.add(569);

//1. method

for(int element in numbers){

print(element);
```

```
}
print("-----------");
//2. method
countries.forEach((v) => print('${v}'));
print("-----------");
for(int element in moreNumbers){
if(moreNumbers.lookup(element) == 178){
print(element);
break;
}
}
//set
var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};
print(fruitCollection.lookup('Something Else'));
//it gives null
//lists
List fruitCollections = ['Mango', 'Apple', 'Jack fruit'];
var myIntegers = [1, 2, 3, 'non-integer object'];
print(myIntegers[3]);
print(fruitCollections[0]);
}
main(List<String> arguments){
setFunction();
}
```

Przyjrzyjmy się najpierw wynikom; wtedy będziemy w stanie zrozumieć, co się dzieje.

```
//output of code 7.6
1
45
58
-----------
India
```

England

US

178

null

non-integer object

Mango

Stworzyliśmy zestaw krajów, liczb i nie tylko; na koniec stworzyliśmy Listę, aby rozróżnić znaki List i Zestawów. Te trzy metody utworzyły zestawy:

```
Set<String> countries = Set.from(['India', 'England', 'US']);

Set<int> numbers = Set.from([1, 45, 58]);

Set<int> moreNumbers = Set();
```

Za pomocą tej metody otrzymujemy wynik pierwszego, który mamy:

```
countries.forEach((v) => print('${v}'));
```

Druga lista została pobrana tą metodą:

```
for(int element in numbers){

print(element);

}
```

We have captured the values of the third Set using this method:

```
for(int element in moreNumbers){

if(moreNumbers.lookup(element) == 178){

print(moreNumbers);

break;

}

}
```

Użyliśmy metody lookup() z każdym elementem listy argumentów. Kiedy dopasujemy 178, drukujemy całą listę. Aby manipulować zestawem, w Dart dostępnych jest wiele metod podstawowej biblioteki. Możesz użyć moreNumbers.contains(value), moreNumbers. Remove(value), moreNumbers.isEmpty() itp. W poniższym fragmencie kodu wartość zwracana przez funkcję lookup() ma wartość null, ponieważ w zestawie nie ma takiej wartości:

```
//set

var fruitCollection = {'Mango', 'Apple', 'Jack fruit'};

print(fruitCollection.lookup('Something Else'));
```

Musimy pamiętać o jednej rzeczy. Gdy typ Set jest liczbą całkowitą, łatwiej jest użyć pętli for do zapętlenia elementów. W przeciwnym razie mądrze jest użyć foreach, tak jak użyliśmy w poprzednim kodzie.

```
countries.forEach((v) => print('${v}'));
```

W następnej sekcji zobaczymy jak działa Mapa w Darcie.

Mapy: para klucz-wartość

Nieuporządkowany zbiór par klucz-wartość jest w Dart nazywany mapą. Główną zaletą mapy jest to, że para klucz-wartość może być dowolnego typu. Elastyczność rozszerzania i zmniejszania tych nieuporządkowanych kolekcji to kolejna wielka zaleta przy zarządzaniu dużymi porcjami danych. Rysunek podsumowuje działanie Map w Dart.

Na początek zacznijmy od kilku kluczowych funkcji Mapy, o których powinniśmy pamiętać pracując z Mapą.

• Każdy klucz na mapie powinien być unikalny.

• Wartość można powtórzyć.

• Mapę można powszechnie nazwać skrótem lub słownikiem.

• Rozmiar mapy nie jest stały; może zwiększać się lub zmniejszać w zależności od liczby elementów. Innymi słowy, Mapy mogą się powiększać lub zmniejszać w czasie wykonywania.

• HashMap jest implementacją mapy opartą na tablicy mieszającej.

Przyjrzyjmy się następującemu fragmentowi kodu, aby zrozumieć, jak działa mapa w Dart:

```
//code 7.7

void mapFunction(){

//unordered collection of key=>value pair

Map<String, String> countries = Map();

countries['India'] = "Asia";

countries["Germany"] = "Europe";

countries["France"] = "Europe";
```

```
countries["Brazil"] = "South America";
//1. method we can obtain key or value
for(var key in countries.keys){
print("Country's name: $key");
}
print("-----------");
for(String value in countries.values){
print("Continent's name: $value");
}
//2. method
countries.forEach((key, value) => print("Country: $key and
Continent: $value"));
//we can update any map very easily
if(countries.containsKey("Germany")){
countries.update("Germany", (value) => "European Union");
print("Updated country Germany.");
countries.forEach((key, value) => print("Country: $key and
Continent: $value"));
}
//we can remove any country
countries.remove("Brazil");
countries.forEach((key, value) => print("Country: $key and
Continent: $value"));
print("Barzil has been removed successfully.");
print("-----------");
//3. method of creating a map
Map<String, int> telephoneNumbersOfCustomers = {
"John" : 1234,
"Mac" : 7534,
"Molly" : 8934,
"Plywod" : 1275,
```

```
"Hagudu" : 2534
};
telephoneNumbersOfCustomers.forEach((key, value) =>
print("Customer: $key and Contact Number: $value"));
}
main(List<String> arguments){
mapFunction();
}
```

Here is the output of the previous code:

```
Country's name: India
Country's name: Germany
Country's name: France
Country's name: Brazil
-----------
Continent's name: Asia
Continent's name: Europe
Continent's name: Europe
Continent's name: South America
Country: India and Continent: Asia
Country: Germany and Continent: Europe
Country: France and Continent: Europe
Country: Brazil and Continent: South America
Updated country Germany.
Country: India and Continent: Asia
Country: Germany and Continent: European Union
Country: France and Continent: Europe
Country: Brazil and Continent: South America
Country: India and Continent: Asia
Country: Germany and Continent: European Union
Country: France and Continent: Europe
Barzil has been removed successfully.
```

```
-----------
Customer: John and Contact NUmber: 1234
Customer: Mac and Contact NUmber: 7534
Customer: Molly and Contact NUmber: 8934
Customer: Plywod and Contact NUmber: 1275
Customer: Hagudu and Contact NUmber: 2534
```

There are three methods that we can use to retrieve the values of a Map.

```
//1. method we can obtain key or value
for(var key in countries.keys){
print("Country's name: $key");
}
print("-----------");
//2. Method
for(String value in countries.values){
print("Continent's name: $value");
}
//3. method
countries.forEach((key, value) => print("Country: $key and
Continent: $value"));
```

Ponadto istnieje kilka metod dodawania, aktualizowania lub usuwania elementów mapy.

Wspólne korzystanie z kolekcji

Możemy łączyć listy i mapy oraz testować walidacje. Dart oferuje dużą elastyczność, gdy chcemy potwierdzić, że każdy element przechodzi określony test. Załóżmy, że chcemy sprawdzić, czy każdy użytkownik ma ukończone 18 lat.

```
//code 7.8
main(){
var name;
var age;
List<Map<String, dynamic>> users = [
{ name: "Peter", age: 18 },
{ name: "Mira", age: 20 },
{ name: "Jason", age: 22 },
```

```
];
var is18AndOver = users.every((user) => user[age] >= 18);
print(is18AndOver);
}
```

Dane wyjściowe będą prawdziwe. Wszyscy użytkownicy naszych połączonych List i Map mają ukończone 18 lat. Możemy także sprawdzić, czy nazwa każdego użytkownika zaczyna się na literę A, czy nie. Używamy tej samej metody Every() w inny sposób. Poniższy fragment kodu jest interesujący, ponieważ użyliśmy tutaj anonimowej funkcji:

```
//code 7.9
main(){
var name;
var age;
List<Map<String, dynamic>> users = [
{ name: "Peter", age: 18 },
{ name: "Mira", age: 20 },
{ name: "Jason", age: 22 },
];
var isEighteenAndOver = users.every((user) => user[age] >= 18);
print(isEighteenAndOver);
var hasNamesWithLetterA = users.every((user) => user.
toString().startsWith("A"));
print(hasNamesWithLetterA);
}
```

Spójrzmy tym razem na dane wyjściowe w konsoli edytora:

```
//output of code 7.9
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
--enable-asserts --enable-vm-service:45239 /home/ss/
IdeaProjects/my_app/main.dart
Observatory listening on http://127.0.0.1:45239/o1AwiUzW7MQ=/
true
false
Process finished with exit code 0
```

Pierwszy test przechodzi pomyślnie; jednak drugi test kończy się niepowodzeniem, ponieważ nazwa każdego użytkownika nie zaczyna się na literę A. Uczyńmy ten kod bardziej interesującym, dodając więcej opcji testowania. Tym razem dodaliśmy do tej listy więcej nazwisk, mapując ich rekordy wiekowe, aby dowiedzieć się, ilu użytkowników ma więcej niż 21 lat.

```
//code 7.10

main(){

var name;

var age;

List<Map<String, dynamic>> users = [

{ name: "Peter", age: 18 },

{ name: "Mira", age: 20 },

{ name: "Jason", age: 22 },

{ name: "Morgan", age: 32 },

{ name: "Mary", age: 50 },

{ name: "Will", age: 86 },

{ name: "Bruce", age: 96 },

];

var isEighteenAndOver = users.every((user) => user[age] >= 18);

print(isEighteenAndOver);

var hasNamesWithLetterA = users.every((user) => user.

toString().startsWith("A"));

print(hasNamesWithLetterA);

var overTwentyOne = users.where((user) => user[age] > 21);

print(overTwentyOne.length);

}
```

Spójrz na wynik tym razem:

```
//output of code 7.10

true

false

5
```

W sumie jest pięciu użytkowników w wieku powyżej 21 lat.

Następnie możemy uzupełnić ten test metodą singleWhere(), aby potwierdzić, że nie ma użytkowników w wieku poniżej 18 lat.

```
//code 7.11

main(){

var name;

var age;

List<Map<String, dynamic>> users = [

{ name: "Peter", age: 18 },

{ name: "Mira", age: 20 },

{ name: "Jason", age: 22 },

{ name: "Morgan", age: 32 },

{ name: "Mary", age: 50 },

{ name: "Will", age: 86 },

{ name: "Bruce", age: 96 },

];

var isEighteenAndOver = users.every((user) => user[age] >= 18);

print(isEighteenAndOver);

var hasNamesWithLetterA = users.every((user) => user.

toString().startsWith("A"));

print(hasNamesWithLetterA);

var overTwentyOne = users.where((user) => user[age] > 21);

print(overTwentyOne.length);

var underEighteen = users.singleWhere((user) => user[age]

< 18, orElse: () => null);

print(underEighteen);

}
```

Najpierw zobaczmy dane wyjściowe w naszej konsoli, jak pokazano tutaj:

```
//output of code 7.11

/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart

--enable-asserts --enable-vm-service:36063 /home/ss/

IdeaProjects/my_app/main.dart
```

Observatory listening on http://127.0.0.1:36063/vr1OkScPAEw=/

true

false

5

null

Process finished with exit code 0

Ostatni wiersz wyniku mówi nam, że nie ma użytkownika w wieku poniżej 18 lat. Chcemy, abyś spojrzał na ostatni fragment kodu:

```
var underEighteen = users.singleWhere((user) => user[age] < 18,

orElse: () => null);

print(underEighteen);
```

W trybie warunkowym orElse użyliśmy funkcji anonimowej () => null; ta anonimowa funkcja zwraca wartość null tylko wtedy, gdy warunek jest spełniony. Na koniec w Dart zobaczymy kolejną funkcję zbierania, zwaną kolejką.

Kolejka jest otwarta

Kolejka jest przydatna, gdy próbujesz zbudować kolekcję, którą można dodać z jednego końca i usunąć z drugiego końca. Wartości są usuwane lub odczytywane za pomocą indeksu na podstawie kolejności ich wstawiania. Rozważ ten kod:

```
//code 7.12

import 'dart:collection'; // we are about to import some extra

methods from collection library

main(List<String> arguments){

Queue myQueue = new Queue();

print("Default implementation ${myQueue.runtimeType}");

myQueue.add("Sanjib");

myQueue.add(54);

myQueue.add("Howrah");

myQueue.add("sanjib12sinha@gmail.com");

for(var allTheValues in myQueue){

print(allTheValues);

}

print("----------");

print("We are removing the first element ${myQueue.
```

```
elementAt(0)}.");
myQueue.removeFirst();
for(var allTheValues in myQueue){
print(allTheValues);
}
print("----------");
print("We are removing the last element ${myQueue.
elementAt(2 )}.");
myQueue.removeLast();
for(var allTheValues in myQueue){
print(allTheValues);
}
}
```

The output gives us the full lists of what we have added in the Queue.

After that, we have removed the first and last elements.

```
//output of code 7.12
Default implementation ListQueue<dynamic>
Sanjib
54
Howrah
sanjib12sinha@gmail.com
----------
We are removing the first element Sanjib.
54
Howrah
sanjib12sinha@gmail.com
----------
We are removing the last element sanjib12sinha@gmail.com.
54
Howrah
```

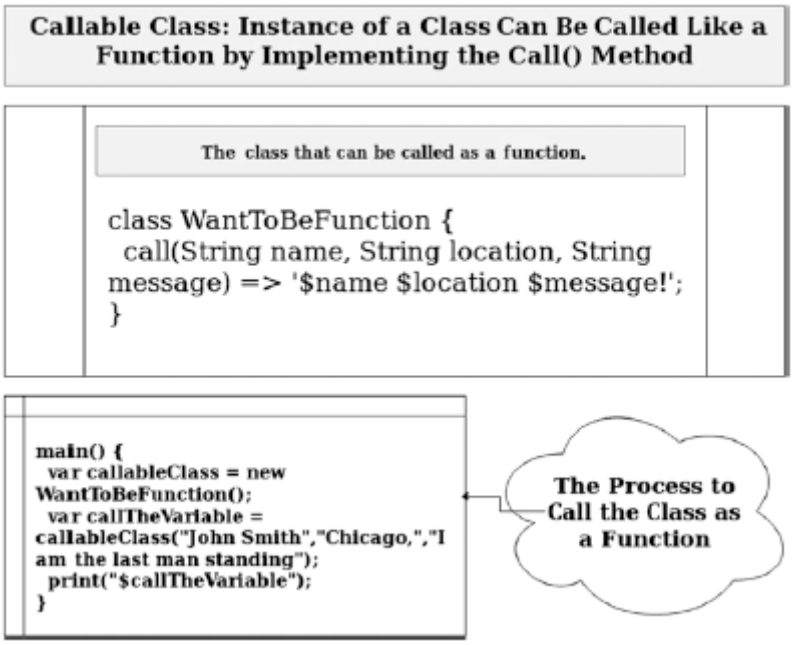
W większości przypadków, jak powiedziałem na początku rozdziału, możemy sobie z tym poradzić za pomocą list i map. Zatem kolejka jest opcją, której możesz czasami potrzebować, ale niezbyt często.

Programowanie wielowątkowe z wykorzystaniem klas przyszłych i wywoływalnych

Jak wiadomo, w Dart wszystko jest przedmiotem. Klasa jest obiektem. Funkcja jest także przedmiotem. Ze względu na podejście obiektowe obiekty powinny zawierać pewne metody, które pozwolą im zachowywać się jak funkcje. W tym rozdziale zobaczysz, jak możemy sprawić, by zachowywały się jak funkcje. Pozwolimy instancji dowolnej klasy zachowywać się jak funkcja. Teraz Dart umożliwia wywoływanie obiektów z metodami wywoływania i jednocześnie przypisywanie ich do zmiennych typu funkcji. W tym rozdziale przyjrzymy się jednemu z najważniejszych aspektów programowania Dart, czyli programowaniu wielowątkowemu z wykorzystaniem klas future i wywoływalnych

Klasy wywoływalne

Wewnętrznie Dart pośrednio zmienia metodę call() (np. SomeVariable. call()) na zamknięcie. Gdy obiektowi zostanie przypisana metoda wywołania do typu funkcji, przyjmuje on cechy funkcji anonimowej. Wywoływanie klasy jak funkcji jest interesującą funkcją w Dart. Wszystko, co musimy zrobić, to po prostu zaimplementować metodę call(). Zanim przetestujemy przykładowy kod, rozważ rysunek



Przetestujmy trochę kodu i zobaczmy wynik.

```
//code 8.1

class CallableClassWithoutArgument {

String output = "Callable class";

void call() {

print(output);

}
```

```
}
call(String name) => "$name";
}
main(){
var withoutArgument = CallableClassWithoutArgument();
var withArgument = CallableClassWithArgument();
withoutArgument(); // it is equivalent to withoutArgument.
call()
print(withArgument("John Smith")); //OK.
// withArgument(); //it'll give error
print(withArgument.call("Calling John Smith"));
}
```

Oto wynik poprzedniego kodu:

```
//output of code 8.1
Callable class
John Smith
Calling John Smith
```

Możemy również użyć klasy wywoływalnej, aby mogła przyjmować opcjonalny parametr. W poniższym kodzie parametr [name] jest opcjonalny w wywoływalnej klasie o nazwie Person:

```
//code 8.2
//when dart class is callable like a function, use call() function
class Person{
String name;
String call(String message, [name]){
return "This message: '$message', has been passed to the
person $name.";
}
}
main(List<String> arguments){
var John = Person();
John.name = "John Smith";
```

String name = John.name;

String msgAndName = John("Hi John how are you?", name);

print(msgAndName);

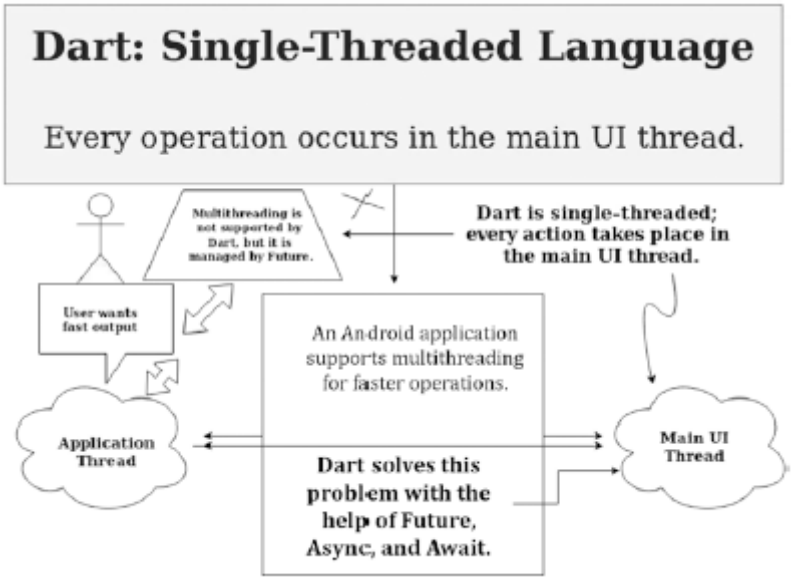
Oto dane wyjściowe:

This message: 'Hi John how are you?', has been passed to the person John Smith.

Tutaj John jest zmienną, a Person() jest klasą. Klasę Person wywołuje się podobnie jak funkcję, ponieważ zaimplementowaliśmy funkcję call(), przez którą przekazaliśmy dwa parametry: wiadomość typu String oraz opcjonalną nazwę parametru. Na koniec przekazaliśmy oba i przechwyciliśmy wartość za pomocą msgAndName.

Programowanie przyszłości, asynchroniczne, oczekujące i asynchroniczne

Ponieważ Dart jest językiem jednowątkowym, błędne jest założenie, że w Dart nie możemy używać wielowątkowego programowania asynchronicznego. Zanim zagłębisz się w programowanie asynchroniczne i w to, jak Dart sobie z tym radzi, musisz zrozumieć podstawowy mechanizm dowolnej aplikacji na Androida. Za każdym razem, gdy włączamy dowolne urządzenie z Androidem, rozpoczyna się domyślny proces. Działa w głównym wątku interfejsu użytkownika. Ten główny wątek interfejsu użytkownika obsługuje wszystkie podstawowe czynności, takie jak klikanie przycisków, wszelkiego rodzaju czynności na ekranie dotykowym itp. Jednak nie są to jedyne rzeczy, których możemy oczekiwać od urządzenia z Androidem. Powinniśmy być w stanie wykonywać niektóre ciężkie operacje, takie jak sprawdzanie poczty, pobieranie plików, oglądanie filmów, granie w gry itp. Programowanie wielowątkowe. Otwiera wątek aplikacji, w którym zarządzane są wszystkie ciężkie operacje. Kiedy w tle trwają ciężkie operacje, nasz interfejs użytkownika musi reagować; i z tego powodu Android umożliwia przetwarzanie równoległe. Jest to normalna procedura programowania asynchronicznego. Ponieważ Dart jest jednowątkowym językiem programowania, zarządza programowaniem asynchronicznym za pomocą funkcji o nazwie Future. W Dart SDK w wersji 1.9 język Dart dodał obsługę asynchronii. Teraz łatwiej jest pisać i czytać asynchroniczny kod Dart. Zobaczymy to za chwilę. Spróbujmy najpierw zrozumieć całą koncepcję, jak pokazano na rysunku.



Zanim funkcje asynchronizacji i oczekiwania zostały wprowadzone do zestawu Dart SDK w wersji 1.9, Dart polegał głównie na Future i wtedy. Przyjrzymy się przykładowemu kodowi, aby zobaczyć, jak Dart Future zarządza wątkiem aplikacji równolegle do głównego wątku interfejsu użytkownika. Rozważ następujący kod, w którym musimy zaimportować biblioteki asynchroniczne; importujemy predefiniowane klasy z bibliotek Dart, aby dodać specjalne funkcje w naszym programie:

```
//code 8.3

import 'dart:async';

// our all operations will use the main UI thread

// since dart and flutter are single threaded we need to use

Future, Async amd Await APIs

void main(){

// the main UI thread starts after that the heavy operations

will take place

print("The main UI thread is starting on here.");

print("Now it will take 10 seconds to display news

headlines.");

displayNews();

print("The main UI thread ends.");

// this program remains incomplete, we don't get the result

// it is because the main UI thread is overlapping before 10

seconds

// therefore we need await and async APIs to block main UI

thread for 10 seconds

}

// this is where our heavy operations are taking place

Future<String> checkingNewsApp() {

// since we are returning a string value

// by delaying the main UI thread for 10 seconds

Future<String> delayingTenSeconds = Future.

delayed(Duration(seconds: 10), (){

return "The latest headlines are displayed here after

10 seconds.";
```

```
});
// after 10 seconds the news headlines will be displayed
return delayingTenSeconds;
}
void displayNews() {
// here we will primarily display the news headline
Future<String> displayingNewsHeadlines = checkingNewsApp();
// inside then we need an anonymous function like lambda or
anonymous function
displayingNewsHeadlines.then((displayString){
// it will give the future object
print("Displaying news headlines here:
$displayingNewsHeadlines");
});
}
```

Na wyjściu otrzymaliśmy wartość jako obiekt Future.

```
//output of code 8.3
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
--enable-asserts --enable-vm-service:42565 /home/ss/
IdeaProjects/my_app/main.dart
Observatory listening on http://127.0.0.1:42565/vR6Xhf8qofo=/
The main UI thread is starting on here.
Now it will take 10 seconds to display news headlines.
The main UI thread ends.
Displaying news headlines here: Instance of 'Future<String>'
Process finished with exit code 0
```

Jeśli przeczytasz komentarze w programie, zrozumiesz, jak działa obiekt Przyszłość. Jednak nasz cel był inny; chcieliśmy wyświetlać nagłówki wiadomości zamiast obiektu Przyszłość. W tej linijce został popełniony błąd:

```
// it will give the future object
print("Displaying news headlines here:
$displayingNewsHeadlines");
```

```
});
```

W metodzie Future przekazaliśmy anonimową funkcję, w której użyliśmy parametru. Musimy jeszcze raz sprawdzić tę linię. Napiszmy kod w ten sposób (zmiany zaznaczono pogrubioną czcionką):

```
//code 8.4

import 'dart:async';

// our all operations will use the main UI thread

// since dart and flutter are single threaded we need to use

Future, Async amd Await APIs

void main(){

// the main UI thread starts after that the heavy operations

will take place

print("The main UI thread is starting on here.");

print("Now it will take 10 seconds to display news headlines.");

displayNews();

print("The main UI thread ends.");

// this program remains incomplete, we don't get the result

// it is because the main UI thread is overlapping before

10 seconds

// therefore we need the Future API to block main UI thread

for 10 seconds

}

// this is where our heavy operations are taking place

Future<String> checkingNewsApp() {

// since we are returning a string value

// by delaying the main UI thread for 10 seconds

Future<String> delayingTenSeconds = Future.

delayed(Duration(seconds: 10), (){

return "The latest headlines are displayed here after

10 seconds.";

});

// after 10 seconds the news headlines will be displayed
```

```
return delayingTenSeconds;
}
void displayNews() {
// here we will primarily display the news headline
Future<String> displayingNewsHeadlines = checkingNewsApp();
// inside then we need an anonymous function like lambda or
anonymous function
displayingNewsHeadlines.then((displayString){
print("Displaying news headlines here: $displayString");
});
}
```

Teraz dane wyjściowe zostały zmienione. Oto nagłówki wiadomości, jakie początkowo chcieliśmy:

```
//output of code 8.4
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
--enable-asserts --enable-vm-service:42565 /home/ss/
IdeaProjects/my_app/main.dart
Observatory listening on http://127.0.0.1:42565/vR6Xhf8qofo=/
The main UI thread is starting on here.
Now it will take 10 seconds to display news headlines.
The main UI thread ends.
Displaying news headlines here: The latest headlines are
displayed here after 10 seconds.
Process finished with exit code 0
```

Teraz, wraz z pojawieniem się nowoczesnych wersji Darta, mamy asynchronizację i oczekiwanie. Pomagają nam pisać czysty kod asynchroniczny. Trzeba jednak wiedzieć, jak prawidłowo korzystać z tych funkcji. Pomagają nam pisać kod asynchroniczny, który wygląda jak kod synchroniczny, jednocześnie korzystając z Future API. W Dart 2 zamiast wstrzymywać, używa funkcji oczekujących i asynchronicznych do synchronicznego wykonywania. Rozważ poniższy kod, w którym nie wykorzystaliśmy tych funkcji prawidłowo. Zwróć uwagę na długi wyjątek.

```
//code 8.5
import 'dart:async';
void main(){
Future checkVersion() async {
```

```
var version = await checkVersion();
// Do something with version
try {
return version;
} catch (e) {
// React to inability to look up the version
return e;
}
}
print(checkVersion());
}
```

W poprzednim kodzie próbowaliśmy wydrukować wersję pakietu Dart SDK. Otrzymaliśmy następujące dane wyjściowe:

```
//output of code 8.5
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
--enable-asserts --enable-vm-service:34325 /home/ss/
IdeaProjects/my_app/main.dart
Observatory listening on http://127.0.0.1:34325/pJXWzoNT9F0=/
Instance of 'Future<dynamic>'
Unhandled exception:
Stack Overflow
#0 _FutureListener.stateThenOnerror (dart:async/future_
impl.dart:66:20)
#1 Future._thenNoZoneRegistration (dart:async/future_impl.
dart:256:22)
#2 _awaitHelper (dart:async-patch/async_patch.dart:110:17)
#3 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:5:19)
#4 _AsyncAwaitCompleter.start (dart:async-patch/async_
patch.dart:49:6)
#5 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
```

```
main.dart:4:22)
#6 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:5:37)
#7 _AsyncAwaitCompleter.start (dart:async-patch/async_
patch.dart:49:6)
#8 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:4:22)
#9 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:5:37)
#10 _AsyncAwaitCompleter.start (dart:async-patch/async_
patch.dart:49:6)
11 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:4:22)
#12 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:5:37)
#13 _AsyncAwaitCompleter.start (dart:async-patch/async_
patch.dart:49:6)
#14 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:4:22)
#15 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:5:37)
#16 _AsyncAwaitCompleter.start (dart:async-patch/async_
patch.dart:49:6)
....
#11119 _AsyncAwaitCompleter.start (
dart:async-patch/async_
patch.dart:49:6)
#11120 main.checkVersion (file:///home/ss/IdeaProjects/my_app/
main.dart:4:22)
#11121 main (file:///home/ss/IdeaProjects/my_app/main.
dart:15:21)
```

```
#11122 _startIsolate.<anonymous closure> (dart:isolate-patch/
isolate_patch.dart:301:19)
#11123 _RawReceivePortImpl._handleMessage (dart:isolate-patch/
isolate_patch.dart:172:12)
Process finished with exit code 255
```

Dla zwięzłości skróciłem dane wyjściowe. Zgłoszono długi wyjątek

z powodu naszego błędu. Gdybyśmy użyli instrukcji print, nasz problem nie zostałby rozwiązany. Jaka będzie zatem właściwa forma pisania asynchronicznego i oczekującego?

Rozważ ten sam kod tutaj:

```
//code 8.6
import 'dart:async';
void main(){
print("The main UI thread is starting on here.");
print("Now it will take 3 seconds to display the version of
Dart.");
checkVersion();
print("The main UI thread ends.");
}
Future<String> checkingVersion() {
// since we are returning a string value
// by delaying the main UI thread for 3 seconds
Future<String> delayingTenSeconds = Future.
delayed(Duration(seconds: 3), (){
return "The version 2.1 is displayed here after 3 seconds.";
});
// after 3 seconds the version will be displayed
return delayingTenSeconds;
}
void checkVersion() async {
String version = await checkingVersion();
// Do something with version
```

```
try {
print("Displaying version here: $version");
} catch (e) {
// React to inability to look up the version
return e;
}
}
```

W powyższym kodzie te linie są ważne:

```
Future<String> checkingVersion() {
// since we are returning a string value
// by delaying the main UI thread for 3 seconds
Future<String> delayingTenSeconds = Future.
delayed(Duration(seconds: 3), (){
....
void checkVersion() async {
String version = await checkingVersion();
```

Jakiej metody przyszłości używamy? Strunowy. Dlatego metody async i oczekuj powinny podążać za tym. Teraz nasze dane wyjściowe są czystsze niż wcześniej, jak pokazano tutaj:

```
//output of code 8.6
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
--enable-asserts --enable-vm-service:41595 /home/ss/
IdeaProjects/my_app/main.dart
Observatory listening on http://127.0.0.1:41595/hMelJx-vdlw=/
The main UI thread is starting on here.
Now it will take 10 seconds to display news headlines.
The main UI thread ends.
Displaying version here: The version 2.1 is displayed here
after 3 seconds.
Process finished with exit code 0
```

Teraz możemy użyć większej liczby funkcji asynchronicznych i poczekać, aż zrozumiemy, jak faktycznie działają z Future. Możemy wrócić do aplikacji z wiadomościami. Tym razem użyjemy async i Wait

zamiast Future. Rozważ następujący kod, w którym nie użyliśmy funkcji async i Wait. Główny wątek interfejsu użytkownika został zakończony i po dziesięciu sekundach otrzymujemy obiekt Future!

```
//code 8.7

import 'dart:async';

// our all operations will use the main UI thread

// since dart and flutter are single threaded we need to use

Future, Async amd Await APIs

// however, we have not used it here and got the future object

instead of headlines

void main(){

// the main UI thread starts after that the heavy operations

will take place

print("The main UI thread is starting on here.");

print("Now it will take 10 seconds to display news headlines.");

displayNews();

print("The main UI thread ends.");

// this program remains incomplete, we don't get the result

// it is because the main UI thread is overlapping before

10 seconds

// therefore we need await and async APIs to block main UI

thread for 10 seconds

}

// this is where our heavy operations are taking place

Future<String> checkingNewsApp(){

// since we are returning a string value

// by delaying the main UI thread for 10 seconds

Future<String> delayingTenSeconds = Future.

delayed(Duration(seconds: 10), (){

return "The latest headlines are displayed here after

10 seconds.";

});
```

```
// after 10 seconds the news headlines will be displayed

return delayingTenSeconds;

}

void displayNews(){

// here we will primarily display the news headline

Future<String> displayingNewsHeadlines = checkingNewsApp();

print("Displaying news headlines here:

$displayingNewsHeadlines");
```

Naszym celem było wyświetlenie nagłówków wiadomości. Zamiast tego otrzymaliśmy obiekt Przyszłość.

```
//output of code 8.7

/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart

--enable-asserts --enable-vm-service:35735 /home/ss/

IdeaProjects/my_app/main.dart

Observatory listening on http://127.0.0.1:35735/q812ySn2w1s=/

The main UI thread is starting on here.

Now it will take 10 seconds to display news headlines.

Displaying news headlines here: Instance of 'Future<String>'

The main UI thread ends.

Process finished with exit code 0
```

Teraz będziemy prawidłowo korzystać z funkcji asynchronizacji i oczekiwania, aby wyświetlić na ekranie żądane nagłówki wiadomości.

```
//code 8.8

import 'dart:async';

// our all operations will use the main UI thread

// since dart and flutter are single threaded we need to use

Future, Async amd Await APIs

void main(){

// the main UI thread starts after that the heavy operations

will take place

print("The main UI thread is starting on here.");

print("Now it will take 10 seconds to display news
```

```
headlines.");
displayNews();
print("The main UI thread ends.");
// this program remains incomplete, we don't get the result
// it is because the main UI thread is overlapping before
10 seconds
// therefore we need await and async APIs to block main UI
thread for 10 seconds
}
// this is where our heavy operations are taking place
Future<String> checkingNewsApp() {
// since we are returning a string value
// by delaying the main UI thread for 10 seconds
Future<String> delayingTenSeconds = Future.
delayed(Duration(seconds: 10), (){
return "The latest headlines are displayed here after
10 seconds.";
});
// after 10 seconds the news headlines will be displayed
return delayingTenSeconds;
}
void displayNews() async {
// here we will primarily display the news headline
String displayingNewsHeadlines = await checkingNewsApp();
print("Displaying news headlines here:
$displayingNewsHeadlines");
}
```

Tym razem stwierdzamy, że dane wyjściowe uległy zmianie i po zakończeniu głównego wątku interfejsu użytkownika po dziesięciu sekundach wyświetliły się nagłówki wiadomości.

```
//output of code 8.8
/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart
```

--enable-asserts --enable-vm-service:33305 /home/ss/

IdeaProjects/my_app/main.dart

Observatory listening on http://127.0.0.1:33305/FpBV0pOM2qc=/

The main UI thread is starting on here.

Now it will take 10 seconds to display news headlines.

The main UI thread ends.

Displaying news headlines here: The latest headlines are

displayed here after 10 seconds.

Process finished with exit code 0

W kolejnej części poznasz biblioteki i pakiety Dart; ponadto musisz wiedzieć, że biblioteki Dart mają wiele funkcji zwracających obiekty Future. Wszystkie te funkcje są synchroniczne. Obsługują czasochłonne, ciężkie operacje (takie jak we/wy). Aby to zrobić, funkcje te używają słów kluczowych async i oczekuj; pozwala nam to napisać kod asynchroniczny, który wygląda jak kod synchroniczny. Dlatego, aby poprawnie obsłużyć Future i uzyskać pełny wynik Future, musimy użyć async i czekać; ponadto możemy używać starych metod Future API, takich jak then(), catchError() iWhenComplete().

Więcej o Future API

Spójrzmy na więcej przykładów, aby zrozumieć, jak działa Future API

Pracuje. Rozważmy następujący kod, w którym używamy metody Future Opóźnione(), a następnie za pomocą metody then() przekazujemy funkcję lambda w celu wydrukowania wartości.

```
//code 8.9

import 'dart:async';

void main(){

Future<int>.delayed(

Duration(seconds: 6),

() { return 200; },

).then((value) { print(value); });

print('Waiting for a value for 6 seconds...');

}
```

Opóźniliśmy cały proces o sześć sekund; następnie zwracamy wartość.

```
//output of code 8.9

/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart

--enable-asserts --enable-vm-service:35393 /home/ss/

IdeaProjects/my_app/main.dart
```

Observatory listening on http://127.0.0.1:35393/ushFPI8yST4=/

Waiting for a value for 6 seconds...

200

Process finished with exit code 0

Jak widać, w zależności od wyniku wartości, którą mamy zamiar wydrukować, musimy wspomnieć, jakiego typu obiektu Future będziemy używać. W poprzednim kodzie ta linia jest ważna:

Future<int>.delayed()//

Musimy zadeklarować typ obiektu Future. Tutaj jest to liczba całkowita, ponieważ zwracamy liczbę całkowitą. W następnym fragmencie kodu zamiast zwracać konkretną wartość, rzucimy wyjątek:

```
//code 8.10

import 'dart:async';

void main(){

Future<int>.delayed(

Duration(seconds: 6),

() => throw 'We are throwing some error here.',

).then((value) {

print(value);

});

print('Waiting for a value');

}
```

Całkiem naturalnie pojawia się błąd na wyjściu.

```
//output of code 8.10

/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart

--enable-asserts --enable-vm-service:43091 /home/ss/

IdeaProjects/my_app/main.dart

Observatory listening on http://127.0.0.1:43091/8Zr4UnbBJMA=/

Waiting for a value_

Unhandled exception:

We are throwing some error here

#0 main.<anonymous closure> (file:///home/ss/IdeaProjects/

my_app/main.dart:6:13)

#1 new Future.delayed.<anonymous closure> (dart:async/
```

future.dart:316:39)

#2 Timer._createTimer.<anonymous closure> (dart:async-patch/

timer_patch.dart:21:15)

#3 _Timer._runTimers (dart:isolate-patch/timer_impl.

dart:382:19)

#4 _Timer._handleMessage (dart:isolate-patch/timer_impl.

dart:416:5)

#5 _RawReceivePortImpl._handleMessage (

dart:isolate-patch/

isolate_patch.dart:172:12)

Process finished with exit code 255

Na koniec chcielibyśmy zobaczyć, jak działają niektóre stare metody Future, takie jak catchError() iWhenComplete(), jak pokazano w poniższym kodzie:

```
//code 8.11

import 'dart:async';

void main(){

Future<int>.delayed(

Duration(seconds: 6),

() { return 100; },

).then((value) {

print(value);

}).catchError(

(err) {

print('Caught $err');

},

test: (err) => err.runtimeType == String,

).whenComplete(() { print("Process completed."); });

print('The main UI thread is waiting');

}
```

Jak widać, główny wątek interfejsu użytkownika czeka przez sześć sekund, a następnie generuje dane wyjściowe.

```
//output of code 8.11
```

/home/ss/Downloads/flutter/bin/cache/dart-sdk/bin/dart

--enable-asserts --enable-vm-service:36707 /home/ss/

IdeaProjects/my_app/main.dart

Observatory listening on http://127.0.0.1:36707/AIbh2kXqMxM=/

The main UI thread is waiting

100

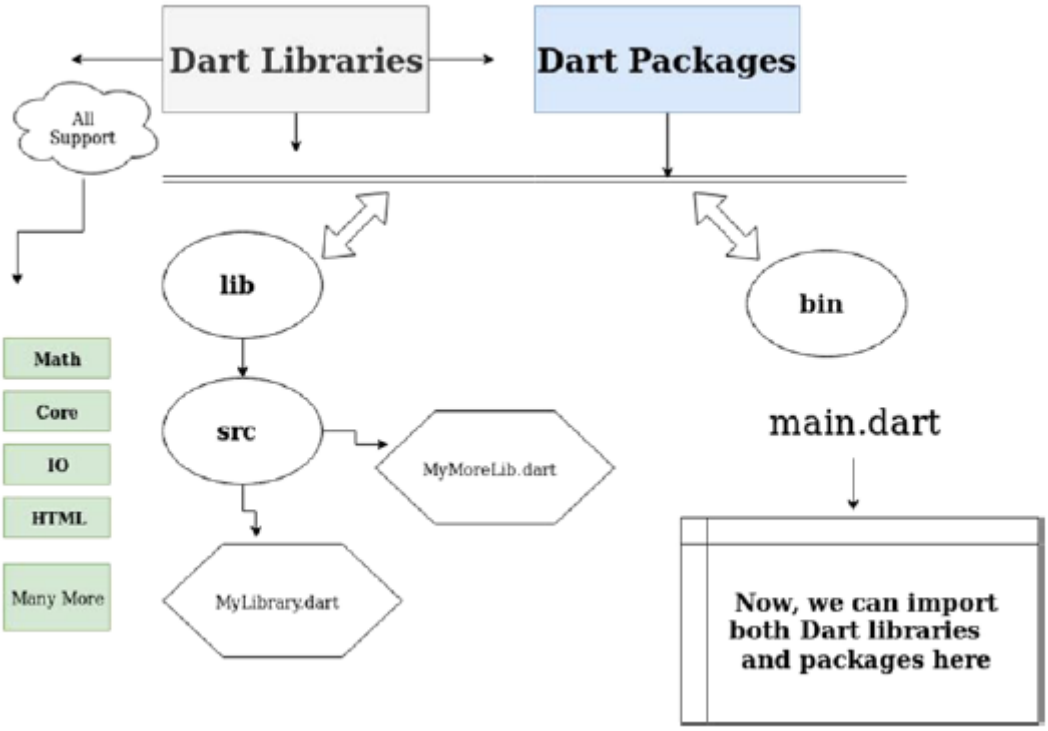
Process completed.

Process finished with exit code 0

Zasadniczo Future opóźnia wątek o kilka sekund, jak wspomniano wcześniej; wówczas generuje albo kompletne dane, albo błąd!

Pakiety i biblioteki Dart

Programowanie Dart w dużej mierze opiera się na bibliotekach, które zawierają krytyczne zestawy wbudowanych funkcjonalności. Dostępnych jest kilka popularnych bibliotek; ponadto można programować modułowo za pomocą bibliotek. Widzieliśmy już wiele przykładów, takich jak predefiniowane metody zbierania danych, wiele funkcji matematycznych itp. Kilka popularnych bibliotek służy wielu celom przy tworzeniu aplikacji Dart. Do tej pory widzieliście wiele wbudowanych funkcji, których używaliśmy w naszych wielu funkcjach zdefiniowanych przez użytkownika. Na przykład biblioteki dart:core zapewniają pomoc dotyczącą liczb, operacji specyficznych dla ciągów i kolekcji. Za pomocą dart:math możemy dość łatwo wykonywać wiele rodzajów operacji matematycznych. Możemy także budować własne biblioteki. Tak naprawdę w miarę postępów poczujesz potrzebę tworzenia własnych bibliotek. Ponadto możesz uzyskać dodatkowe biblioteki, importując je z pakietów. Za pośrednictwem pakietów możemy udostępniać oprogramowanie, takie jak biblioteki i narzędzia. Zasadniczo pomoc możemy uzyskać z obu typów bibliotek (wbudowanej i niestandardowej). Rysunek pokazuje, jak możemy używać bibliotek i pakietów w Dart.



Powinieneś także wiedzieć, dlaczego potrzebujemy bibliotek. Aby stworzyć modułową i współdzieloną bazę kodu, baza kodu musi być dobrze zorganizowana. W rzeczywistości jest to istotna część programowania obiektowego. Biblioteki nie tylko zapewniają wsparcie dla programowania modułowego, obiektowego, ale także zapewniają pewnego rodzaju prywatność we własnym kodzie. Identyfikatory zaczynające się od podkreślenia (_) są widoczne tylko w Twoich bibliotekach. Znak podkreślenia poprzedzający oznacza, że ta funkcja jest prywatna dla biblioteki. Oznacza to, że nie można używać tej funkcji w innych bibliotekach. Jest to typowa funkcja Darta. Dart obsługuje widoczność za pomocą tego poprzedzającego podkreślenia. Biblioteki pomagają także uniknąć konfliktów nazw, co jest istotną częścią kodowania. Spójrzmy na przykład, aby wyjaśnić te punkty.

Importowanie pakietów

Najpierw utwórzmy plik RelationalOperators.dart w folderze lib.

```
//code 9.1

//lib/ RelationalOperators.dart

class TrueOrFalse{

int firstNum = 40;

int secondNum = 40;

int thirdNum = 74;

int fourthNum = 56;

void BetweenTrueOrFalse(){

if (firstNum == secondNum || thirdNum == fourthNum){

print("If choice between 'true' or 'false', in this case
```

```
the 'TRUE' gets the precedence. $firstNum is equal to
$secondNum");
} else print("Nothing happens.");
}
void BetweenTrueAndFalse(){
if (firstNum == secondNum && thirdNum == fourthNum){
print("It will go to else clause");
} else print("If choice between 'true' and 'false', in
this case the 'FALSE' gets the precedence. $thirdNum is
not equal to $fourthNum");
}
}
```

Następnie utwórz plik o nazwie PowProject.dart w folderze lib.

```
//code 9.2
//lib/PowProject.dart
class PowProject{
void MultiplyByAGivenNumber(int fixedNumber, int givenNumber){
int result = fixedNumber * givenNumber;
print(result);
}
void pow(int x, int y){
int addition = x + y;
print(addition);
}
}
```

Teraz spójrz na treść funkcji main(), pokazaną tutaj:

```
//code 9.3
import 'dart:math' as math;
import 'package:IdeaProjects/PowProject.dart';
import 'package:IdeaProjects/RelationalOperators.dart' as
relation;
```

```
main(List<String> arguments){

print("Printing 2 to the power 5 using Dart's built-in

'dart:math' library.");

var int = math.pow(2, 5);

print(int);

print("Now we are going to use another 'pow()' function from

our own library.");

var anotherPowObject = PowProject();

anotherPowObject.MultiplyByAGivenNumber(4, 3);

anotherPowObject.pow(2, 12);

print("Now we are going to use another library to test the

relational operators.");

var trueOrFalse = relation.TrueOrFalse();

trueOrFalse.BetweenTrueOrFalse();

trueOrFalse.BetweenTrueAndFalse();

}
```

W folderze lib (lub bibliotekach) utworzyliśmy dwie klasy. Jedna z nich ma funkcję o nazwie pow(). Ale wbudowana biblioteka dart:math ma funkcję o tej samej nazwie: pow(). Nie możemy używać obu funkcji o tej samej nazwie w tym samym kodzie. Dałoby nam to błędy. Aby więc uniknąć konfliktu nazw, musimy stworzyć własną bibliotekę i zdefiniować ją wewnątrz klasy. Całkiem naturalnie, na potrzeby książki, stworzona przez nas funkcja pow() robi coś innego niż obliczanie potęgi liczby. Spójrz na górę funkcji main(), pokazaną tutaj:

```
import 'dart:math' as math;

import 'package:IdeaProjects/PowProject.dart';

import 'package:IdeaProjects/RelationalOperators.dart' as Relation;
```

Użyliśmy słowa kluczowego import, aby określić, w jaki sposób można używać naszych bibliotek, oprócz bibliotek podstawowych. Nasz katalog projektu to IdeaProjects, a PowProject.dart znajduje się w katalogu lib. Ścieżka po katalogu projektu pochodzi z katalogu lib. Po imporcie musimy przekazać argument, który jest niczym innym jak jednolitym identyfikatorem zasobu (URI) określającym biblioteki. Dla wszelkich wbudowanych bibliotek identyfikator URI ma specjalny schemat dart:.... W przypadku innych bibliotek można użyć ścieżki systemu plików lub schematu pakietu:.... Kiedy bezpośrednio korzystamy z bibliotek, używamy normalnej linii takiej jak ta:

```
import 'package:IdeaProjects/PowProject.dart';
```

W takim przypadku możemy bezpośrednio utworzyć obiekt klasy należący do tej konkretnej biblioteki w następujący sposób:

var anotherPowObject = PowProject();

Istnieje jednak inna dobra metoda; możemy wywołać dowolną bibliotekę, używając nazwy takiej jak ta:

import 'package:IdeaProjects/RelationalOperators.dart' as

relation;

Zaletą tego jest to, że teraz możemy utworzyć dowolny obiekt klasy należący do tej biblioteki, używając nowej nazwy, w następujący sposób:

var trueOrFalse = relation.TrueOrFalse();

Te przedrostki są używane, aby uniknąć konfliktów nazw i uprościć długie nazwy pakietów. Możesz pisać klasy o tej samej nazwie w bibliotekach i możesz z nich korzystać, nadając im nazwę.

Korzystanie z wbudowanych bibliotek Dart

Kilka dobrych wbudowanych bibliotek jest dostarczanych z Dartem; nie musisz ich pisać ponownie. Tutaj jest kilka z nich:

• dart:core: Daje nam to wiele podstawowych funkcjonalności. Jest automatycznie importowany do każdego programu Dart.

• dart:math: Widziałeś, jak w naszym programie wykorzystaliśmy podstawową bibliotekę matematyczną. Za pomocą tej biblioteki możemy wykonywać wiele rodzajów operacji matematycznych, takich jak generowanie liczb losowych.

• dart:convert: Dzięki tej bibliotece konwersja pomiędzy różnymi reprezentacjami danych jest prosta; ta konwersja obejmuje JSON i UTF-8.

Pisanie serwera za pomocą Darta

Korzystając z domyślnych bibliotek Dart, możemy łatwo zbudować serwer lokalny, zażądać strony HTML i uzyskać odpowiedź. W pierwszej połowie tej sekcji zobaczymy jak umieścić plik HTML w naszym programie i uzyskać odpowiedź w przeglądarce klienta. Dla relacji serwer-klient, która jest podstawą każdego rodzaju aplikacji webowej, rolę serwera będzie pełnił nasz program Dart, a klientem będzie przeglądarka, z której będziemy korzystać.

Pokazuje Jakiś Prosty Tekst

Napiszmy prosty kod serwera Dart, który wygeneruje odpowiedź w postaci ciągu znaków

//code 9.4

import 'dart:io';

import 'dart:async';

Future main() async {

var myServer = await HttpServer.bind(

'127.0.0.1',

8080,

```
);
print("The server is alive on the above mentioned port and
it's listening "
"on ${myServer.port}/");
myServer.listen((HttpRequest myRequest){
myRequest.response
..write("Bonjour mademoiselle, comment appelez vous?")
..close();
});
}
```

Funkcja main() zaczyna się od Future i async, a później używamy funkcji Wait. Omówiliśmy te pojęcia w poprzednim rozdziale. Następnie za pomocą metody HttpServer.bind() tworzymy obiekt HttpServer. W metodzie bind() przekazujemy dwa parametry: host (127.0.0.1) i port (8080). W tym przypadku używamy prostej instrukcji print, która daje proste dane wyjściowe pokazujące, że nasłuchujemy wcześniej wspomnianego portu.

Teraz, zgodnie ze strukturą relacji serwer-klient, nasz nowy obiekt serwera powinien nasłuchiwać nowego obiektu HttpRequest (tutaj myRequest). A po otrzymaniu żądania obiekt serwera powinien odpowiedzieć. Obiekt odpowiedzi wywołuje metodę write(), która daje nam proste wyniki, takie jak to:

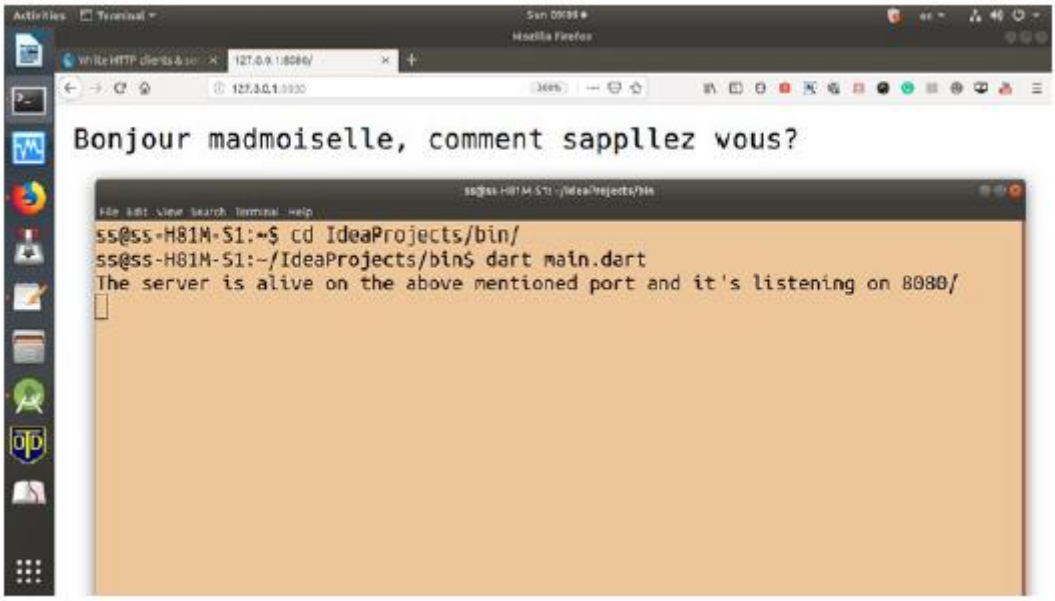
„Bonjour mademoiselle, skomentuj appelez vous?”

To francuskie zdanie, które oznacza: „Dzień dobry, panienko, kim jesteś?"

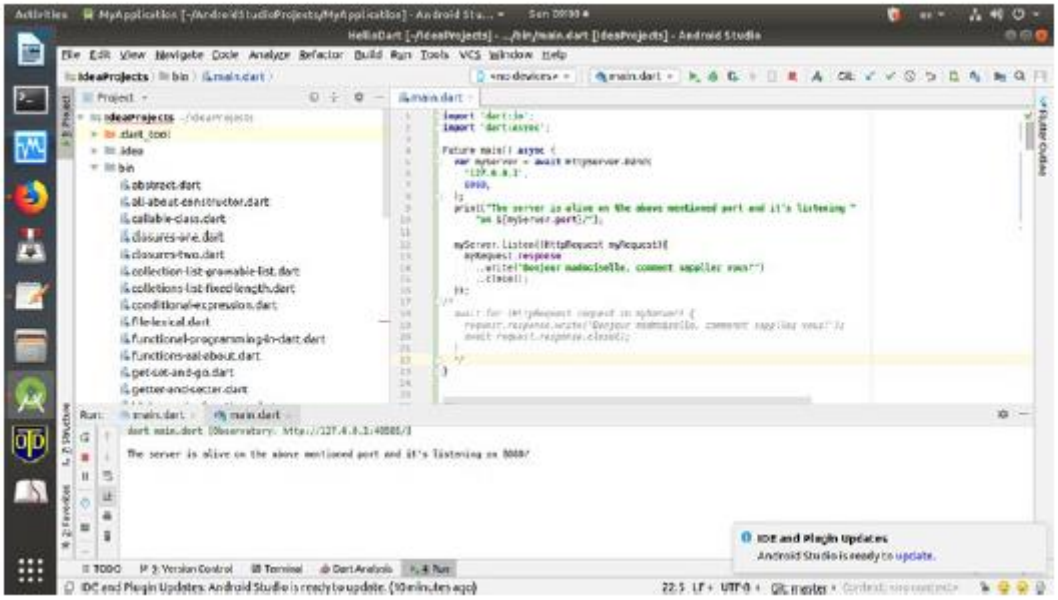
Teraz po uruchomieniu tego programu powinniśmy w dowolnej przeglądarce wpisać http://127.0.0.1:8080. Da to wyjście. Program ten możemy uruchomić na dwa sposoby. Możesz po prostu uruchomić go na Android Studio lub IntelliJ Community, a w konsoli dostaniesz komunikat, że serwer nasłuchuje. Następnie możemy otworzyć przeglądarkę, aby zobaczyć wynik. W inny sposób możemy wykorzystać nasz terminal. Otwórz katalog bin folderu projektu i wpisz następujące polecenie:

```
//kod 9.5
dart main.dart
```

Spowoduje to również uruchomienie programu



Oto zastrzeżenie. Nie należy uruchamiać równolegle dwóch programów nasłuchujących na tym samym porcie. Jeśli port jest już używany, połączenie zostanie automatycznie odrzucone. Możesz użyć dowolnego numeru portu od 1024 wzwyż. Jeśli uruchomisz ten sam program w Android Studio, w konsoli pojawi się komunikat „serwer nasłuchuje”.



Aby zatrzymać program w terminalu, możesz nacisnąć Control+C, a w dowolnej konsoli IDE kliknąć czerwony przycisk po lewej stronie. Zobaczysz to również w prawym górnym rogu. Po zatrzymaniu programu otrzymujemy następujący wynik:

//code 9.6

/home/ss/flutter/bin/cache/dart-sdk/bin/dart --enable-vm-service:

40505 /home/ss/IdeaProjects/bin/main.dart

Observatory listening on http://127.0.0.1:40505/

The server is alive on the above mentioned port and it's

listening on 8080/

Process finished with exit code 137 (interrupted by signal 9: SIGKILL)

Do tej pory mogliśmy podać prostą odpowiedź jako wynik za pośrednictwem naszego programu klienta-serwera zaplecza. Co więcej, możemy sprawić, że nasz serwer zaplecza będzie wyświetlał stronę HTML.

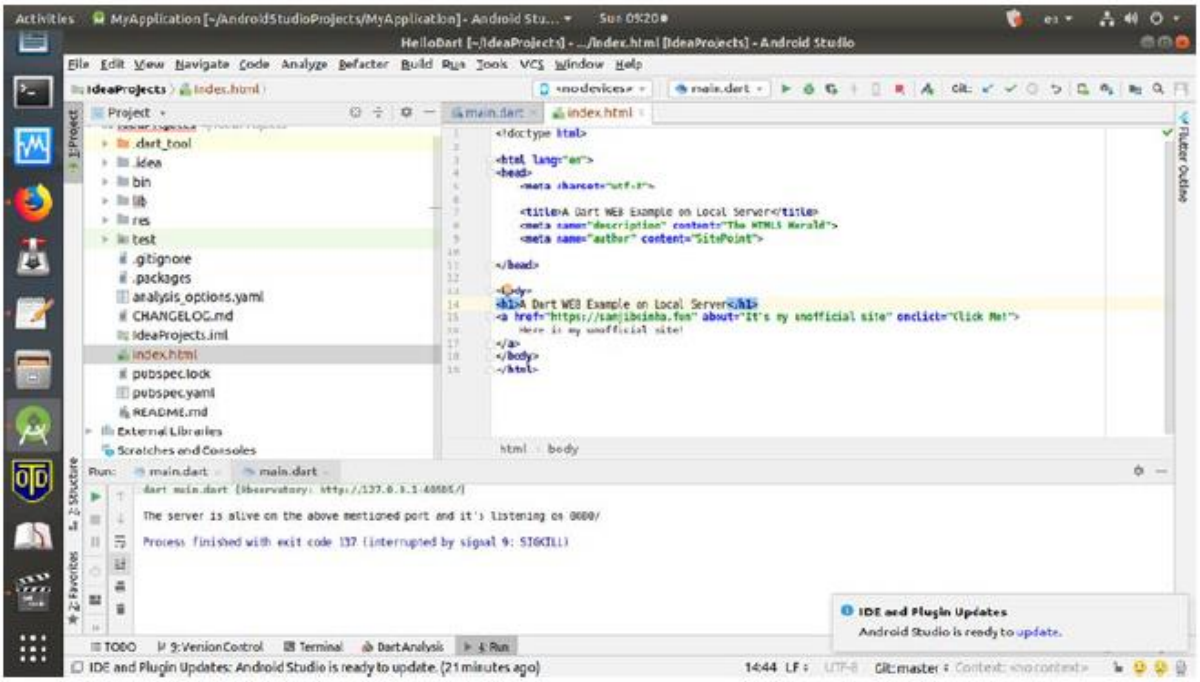
Wyświetlanie strony HTML

Proces nie jest bardzo skomplikowany. Jedyne, czego potrzebujemy, to najpierw plik HTML. Stwórzmy prosty plik HTML5 o nazwie indeks.html w naszym katalogu głównym.

//code 9.7

```
<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>A Dart WEB Example on Local Server</title>

<meta name="description" content="The HTML5 Herald">

<meta name="author" content="SitePoint">

</head>

<body>

<h1>A Dart WEB Example on Local Server</h1>

<a href="https://sanjibsinha.fun" about="It's my unofficial

site" onclick="Click Me!">

Here is my unofficial site!

</a>

</body>

</html>
```

Najpierw zobaczmy plik w Android Studio

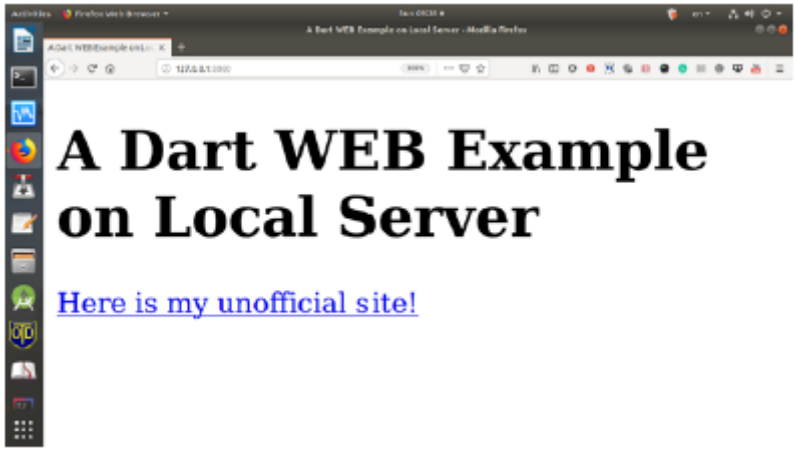


Jak widziałeś wcześniej, biblioteki Dart posiadają wszystkie narzędzia do tworzenia dowolnego rodzaju aplikacji internetowych. Tutaj musimy użyć naszego obiektu odpowiedzi, aby zwrócić zawartość naszego pliku HTML, ustawiając jego nagłówek Content-Type na HTML:

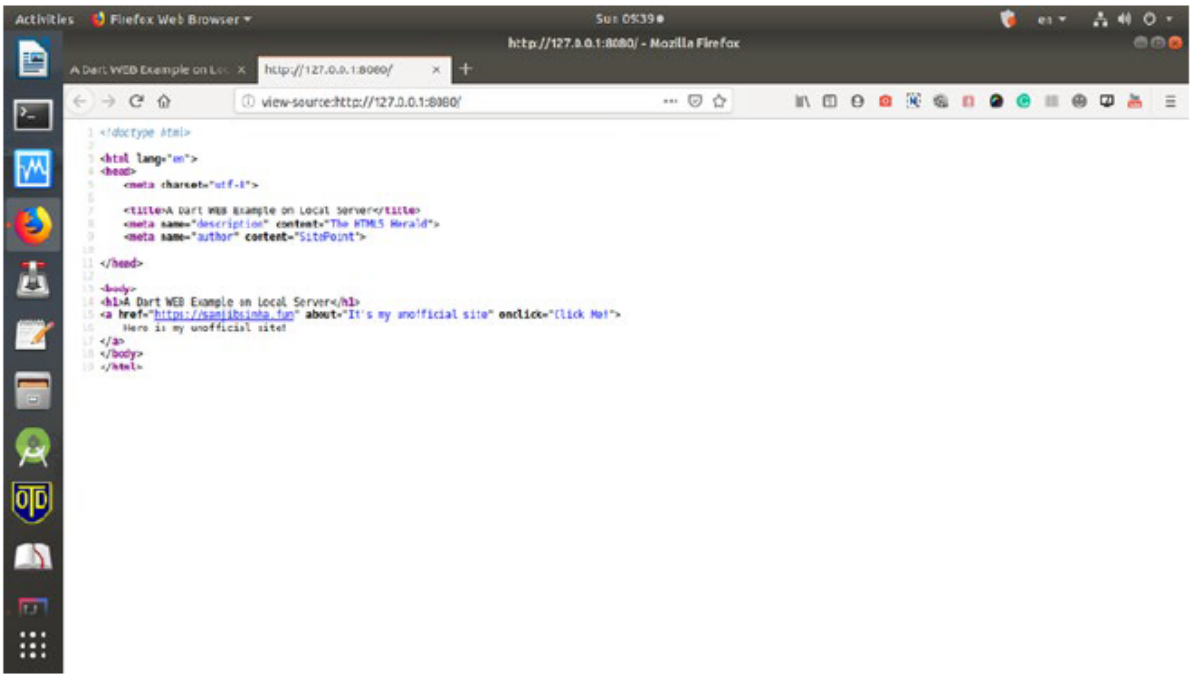
```
//code 9.8

import 'dart:io';

import 'dart:async';

final File myFile = File("index.html");

Future main() async {

var myServer = await HttpServer.bind(

'127.0.0.1',

8080,

);

print("The server is alive on the above mentioned port and

it's listening "

"on ${myServer.port}/");

// we are going to use the await from dart async library

await for (HttpRequest myRequest in myServer) {

if(await myFile.exists()){

print("We're going to serve ${myFile.path}");
```

```
myRequest.response.headers.contentType = ContentType.html;

await myFile.openRead().pipe(myRequest.response);

}

}

}
```

Po przeczytaniu pliku użyliśmy metody pipe() aby umieścić zawartość pliku w odpowiedzi, co da nam odpowiedź



Teraz możemy nawet sprawdzić źródło, aby zobaczyć, czy plik HTML został użyty w tym celu



Widziałeś, jak możemy wykorzystać biblioteki Dart do tworzenia wielu rodzajów skomplikowanych aplikacji. Możemy również łatwo zbudować własne pakiety, korzystając z tych bibliotek, aby ponownie wykorzystać ten kod w innych aplikacjach.

Co dalej

Nie ma wątpliwości, że Dart będzie w przyszłości jeszcze bardziej popularny. Jest nie tylko popularny w świecie iOS i Androida, ale jest używany w aplikacjach internetowych. Ta książka była krótkim wprowadzeniem do Darta. Powodzenia w przyszłości.