KWALIFIKACJA
INF.04

APLIKACJE MOBILNE

DLA STUDENTA I TECHNIKA PROGRAMISTY

Wojciech Jaśkowiec, Krzysztof Kułacz, Marta Kanafa-Suchan
Pod redakcją: Marta Kanafa-Suchan i Krzysztof Kułacz

iTst@rt

Książka **Aplikacje Mobilne dla studenta i technika programisty** to jest przeznaczona dla każdego kto chce zapoznać się z tematem tworzenia aplikacji mobilnych dla systemu Android.

W pierwszej części książki czytelnik zapoznaje się z tajnikami Języka Java. Dogłębnie zapoznaje się z tematyką: programowania obiektowego, oraz współbieżnego. Poruszane są w niej zagadnienia m.in.: klas, obiektów jak i wątków.

W drugiej części zapoznajemy się z tworzeniem aplikacji mobilnych w środowisku Android Studio. Strona wizualna aplikacji tworzona jest z wykorzystaniem języka znaczników xml. Zaś język Java odpowiada za sterowanie działaniem aplikacji.

Książka stanowi szczegółowy poradnik dla każdego kto jest zainteresowany tworzeniem Aplikacji Mobilnych.

Projekt okładki: ***Daniel Pliszka***

Redakcja i skład: ***Marek Smyczek, Marta Kanafa-Suchan, Krzysztof Kułacz***

Wydawnictwo informatyczne

http://www.itstart.pl

email: itstart@itstart.pl

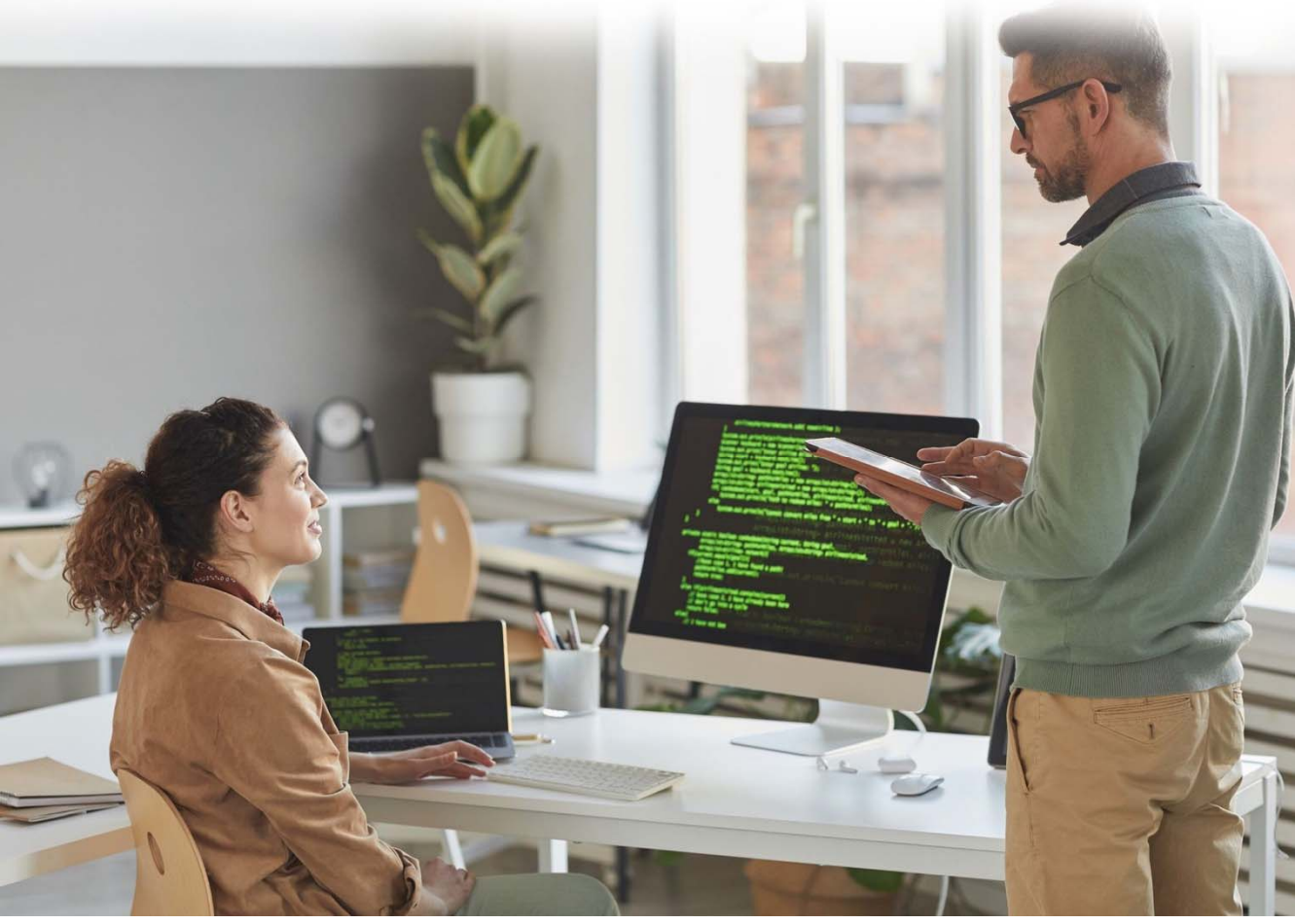
ISBN 978-83-65645-97-5

Wydanie pierwsze – Piekary Śląskie 2024

SPIS TREŚCI

ROZDZIAŁ 1

WSTĘP



1 Wstęp

Książka omawia tworzenie Aplikacji Mobilnych w systemie Android. Tworząc tego typu aplikacje programiści mają obecnie szeroki wybór narzędzi jak i odpowiedniego języka. Samo narzędzie, które omawiane jest w tej książce czyli SDK Android Studio daje możliwość tworzenia aplikacji w kilku językach programowania np. w języku Kotlin, języku Java czy w C++.

Wybór Javy jako języka przewodniego w tym opracowaniu podyktowany jest tym, że w pierwszej kolejności książka ta przeznaczona jest dla uczniów technikum o kierunku technik programista.

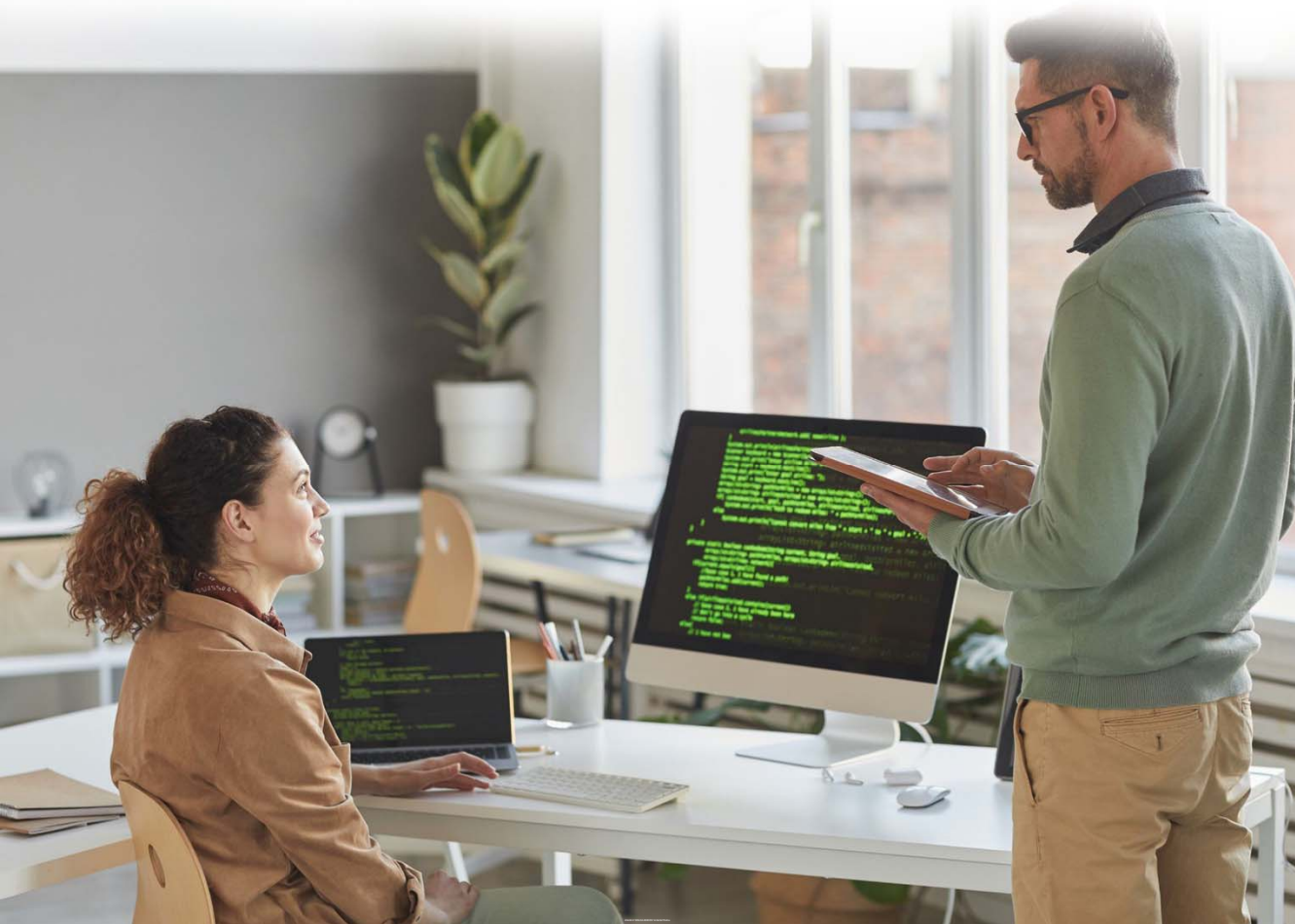
Podstawa programowa tego przedmiotu obejmuje tworzenia aplikacji mobilnych na system Android z wykorzystaniem języka xml, oraz programowania w języku Java. Stąd wybór technologii był oczywisty.

Jeżeli czytelnik zainteresowany jest poszerzeniem swojej wiedzy na temat tworzenia aplikacji mobilnych i chciałby porównać obecny w książce język Java z językiem Kotlin zachęcam serdecznie do przeczytania książki pt.: „Nowoczesne aplikacje mobilne – Kotlin i Android JetPack Compose”.

Wszystkie kody źródłowe dostępne są do pod adresem: **http://aplmob.itstart.pl**. Chcąc uzyskać pełny dostęp do plików, należy się zalogować przy pomocy loginu: **aplmob** i hasła: **apl!@mobDSITP**

ROZDZIAŁ 2

INSTALACJA I KONFIGURACJA OPROGRAMOWANIA



2 Instalacja i konfiguracja oprogramowania

Java to jeden z najpopularniejszych języków programowania. Jest obiektowym, wysokopoziomowym oraz strukturalnym językiem, który opiera się na klasach. Język ten jest dość łatwo przenośny, ponieważ interpretuje go wieloplatformowa maszyna wirtualna Javy (ang. Java Virtual Machine) zmieniająca kod źródłowy na kod bajtowy.

Język ten cechuje się silnym typowaniem, to znaczy, że wyrażenia posiadają już ustalony typ danych i nie może on zostać zmieniony w trakcie działania programu.

Podstawowe koncepcje Javy zostały przejęte od języka **Smalltalk** oraz częściowo od języka C++, przez co składnia Javy jest zbliżona do tej z C++.

Java została zaprojektowana w sposób przyjazny dla użytkownika. Posiada m.in. wiele wbudowanych bibliotek, które dostarczają nam gotowe rozwiązania, co jest sporym ułatwieniem dla programisty.

Jest to język statycznie typowany, dzięki czemu zgodność składniowa oraz typowa będzie sprawdzona pod kątem poprawności zanim nasz program zostanie wykonany. Ułatwia to znacznie znalezienie potencjalnych błędów.

Język ten jest niezależny od platformy przez co może działać na systemach takich jak Windows, Linux czy MacOS. Warto też wspomnieć, że Java jest uniwersalnym językiem. Najczęściej bywa używana przy tworzeniu backendu aplikacji internetowych, ale posiada też wiele innych zastosowań. Java złożona została z trzech głównych komponentów:

- JDK (ang. Java Development Kit) - jest to wieloplatformowe środowisko programistyczne, oferujące biblioteki oraz narzędzia potrzebne do tworzenia aplikacji. W jej skład wchodzą narzędzia takie jak: javac (służy do kompilowania), jdb (służy do debugowania), jar (służy do archiwizacji) i javadoc (służy do generowania dokumentacji).
- JRE (ang. Java Runtime Enviroment) - jest to pewien zestaw komponentów odpowiadających za tworzenie oraz uruchamianie aplikacji.
- JVM (ang. Java Virtual Machine) - jest to maszyna wirtualna pozwalająca na wykonanie kodu bajtowego. Java została stworzona przez grupę roboczą Jamesa Goslinga, który pracował wtedy dla firmy

Sun Microsystems, a sam język pojawił się po raz pierwszy w 1995 roku.

Rysunek 2.1 Zmieniające się logo Java na przestrzeni wielu lat. Od lewej najstarsze, z prawej strony – aktualne i powszechnie stosowane.

Gry takie jak Minecraft, Street Fighter II czy Wolfenstein RPG to tylko przykłady projektów napisanych właśnie w tym języku. Do najbardziej znanych aplikacji należą m.in.: Gmail, OpenOffice, NetBeans czy Eclipse. Znajomość Javy przyda się również przy tworzeniu aplikacji mobilnych. Jak można zauważyć, Java na co dzień jest używana przez najpopularniejsze korporacje na całym świecie.

Rysunek 2.2 Minecraft to jedna z najpopularniejszych gier na całym świecie

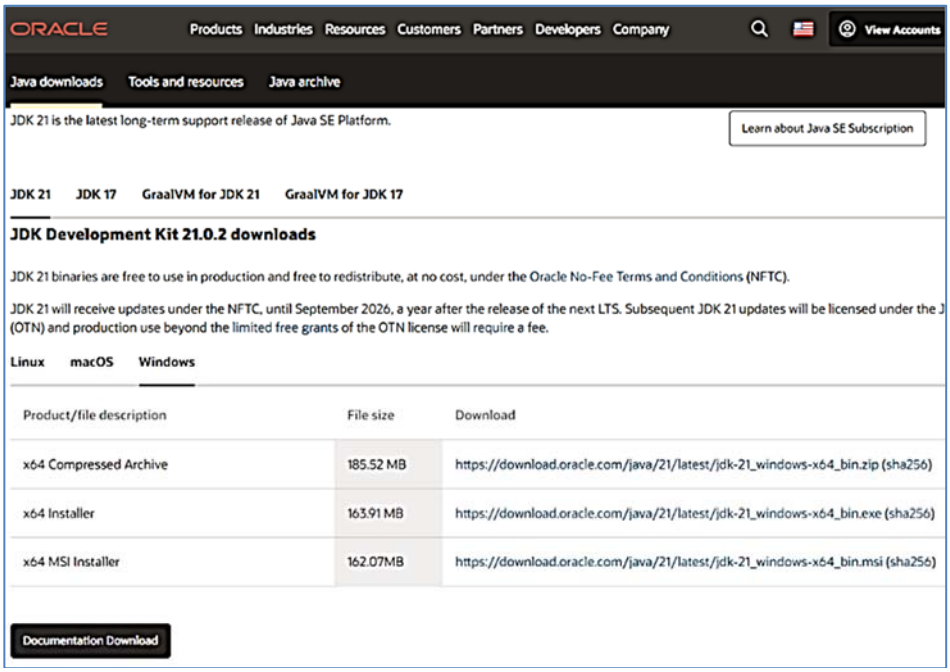
2.1 Instalacja środowiska Java

Rozpoczynając przygodę z programowaniem w Javie musimy przygotować nasz komputer do pracy w tym środowisku. Podstawowe dwa elementy które będą nam potrzebne to wspomniane już wcześniej JDK oraz środowisko programistyczne czyli tzn. IDE (ang. Integrated Development Environment). Na potrzeby naszej pracy będziemy korzystać z programu IntelliJ IDEA produkcji firmy JetBrains. Oczywiście nie jest to jedyny program którego można używać. Na rynku dostępnych jest wiele innych środowisk a wybór należy tylko i wyłącznie do programisty.

2.1.1 Instalacja JDK dla systemu Windows

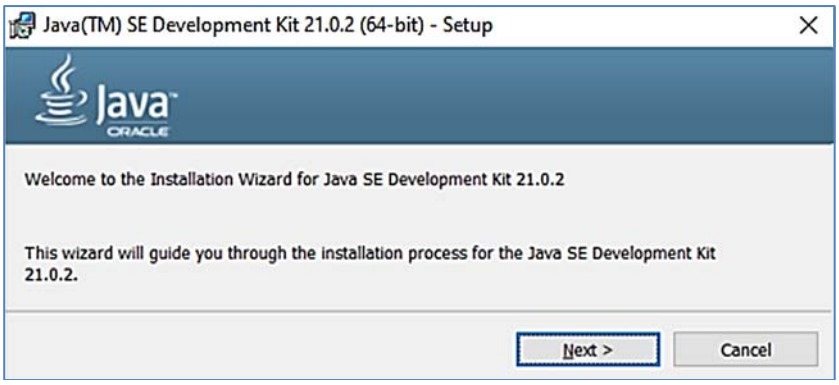
Proces instalacji musimy rozpocząć od pobrania samego JDK. Link do pakietu znajdziemy na stronie *https://www.oracle.com/java/technologies/downloads/*.

Wybieramy odpowiedni plik instalacyjny dedykowany dla naszego systemu. Domyślnie na stronie pobierania wyświetli się najnowsza wersja JDK, ale zdarzają się sytuacje kiedy potrzebne jest starsze wydanie Javy. Dostępne wersje można znaleźć w zakładce **Java archive**.



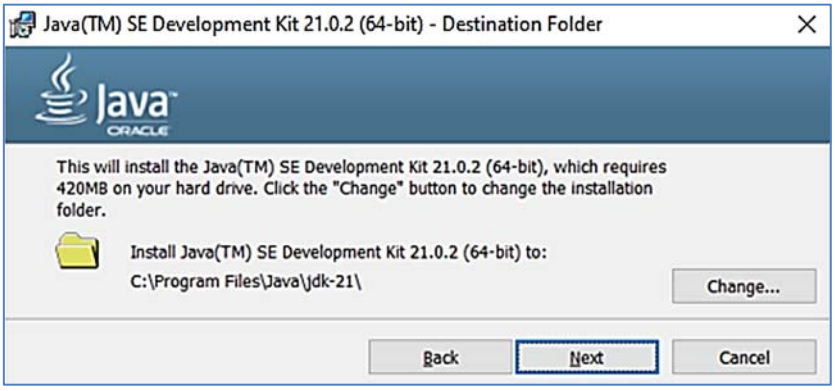
Rysunek 2.3 Środowisko JDK można pobrać ze strony Oracle

Pobieramy i uruchamiamy plik instalacyjny. System wyświetli komunikat z pytaniem o zezwolenie na wprowadzenie zmian na naszym komputerze. Wybieramy **Tak**. Otworzy się okno powitalne programu instalacyjnego w którym klikamy **Next**.



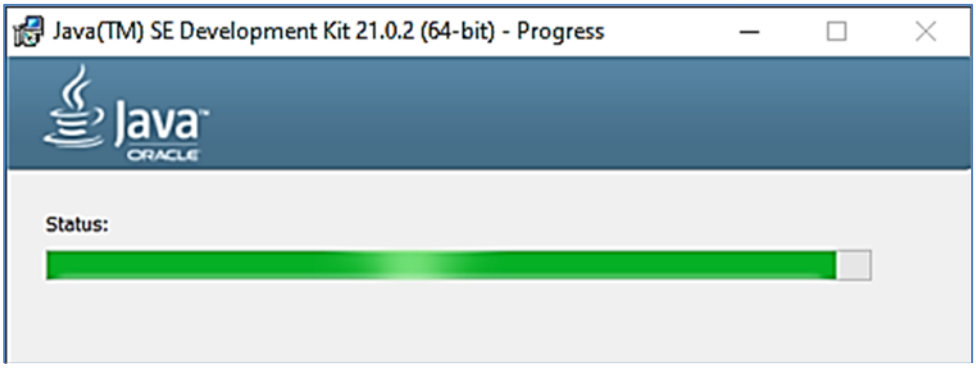
Rysunek 2.4 Okno powitalne instalatora

Teraz wybieramy lokalizację na naszym komputerze w której zostanie zainstalowana Java.



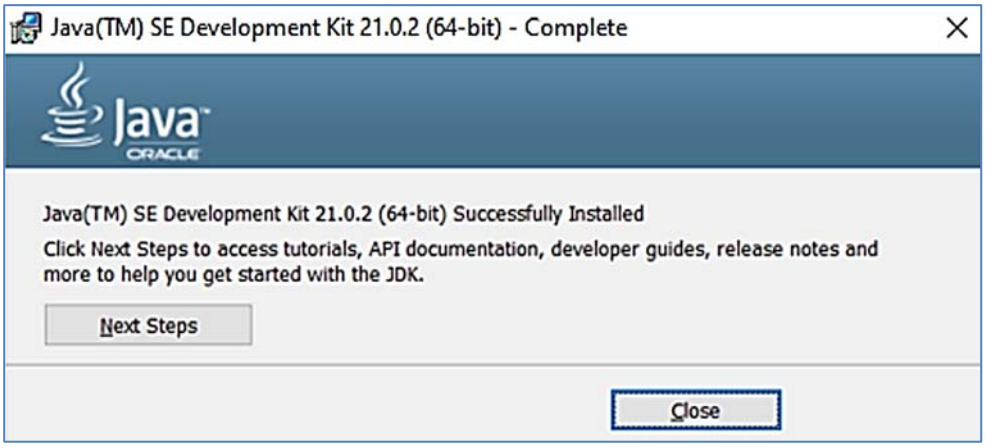
Rysunek 2.5 Określanie lokalizacji dla JDK

Zmieniamy lub pozostawiamy lokalizację i klikamy „**Next**”. Chwilę później rozpocznie się instalacja.



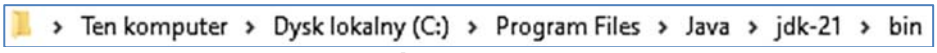
Rysunek 2.6 Postęp instalacji

Po zakończeniu otrzymamy komunikat że, JDK zostało pomyślnie zainstalowane. Klikamy **Close** aby zamknąć program.



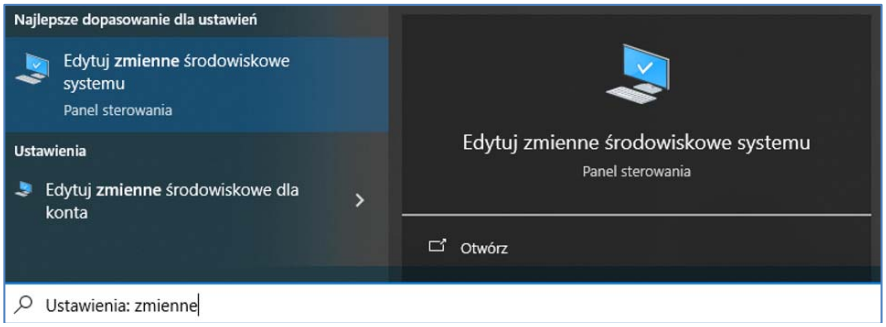
Rysunek 2.7 Pomyślnie zakończona instalacja

Nie jest to jednak koniec naszych działań, ponieważ teraz musimy dodać ścieżkę do Javy w zmiennych środowiskowych. W tym celu potrzebujemy dokładnej lokalizacji w której ją zainstalowaliśmy. Jeżeli w procesie instalacji nie zmienialiśmy domyślnej ścieżki będzie ona wyglądać jak poniżej. Ważne jest aby wejść do folderu JDK a następnie **bin**. Kopiujemy ścieżkę:



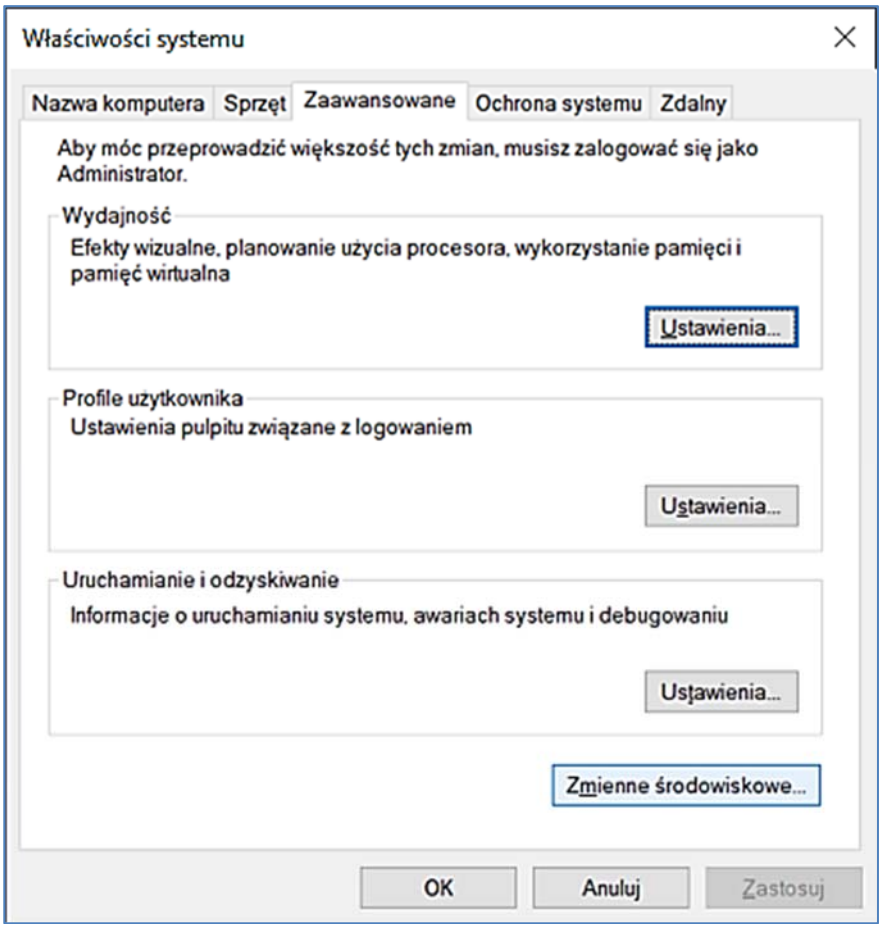
Rysunek 2.8 Ścieżka do folderu "bin"

Teraz klikamy przycisk Windows i wyszukujemy frazę **Edytuj zmienne środowiskowe systemu**.



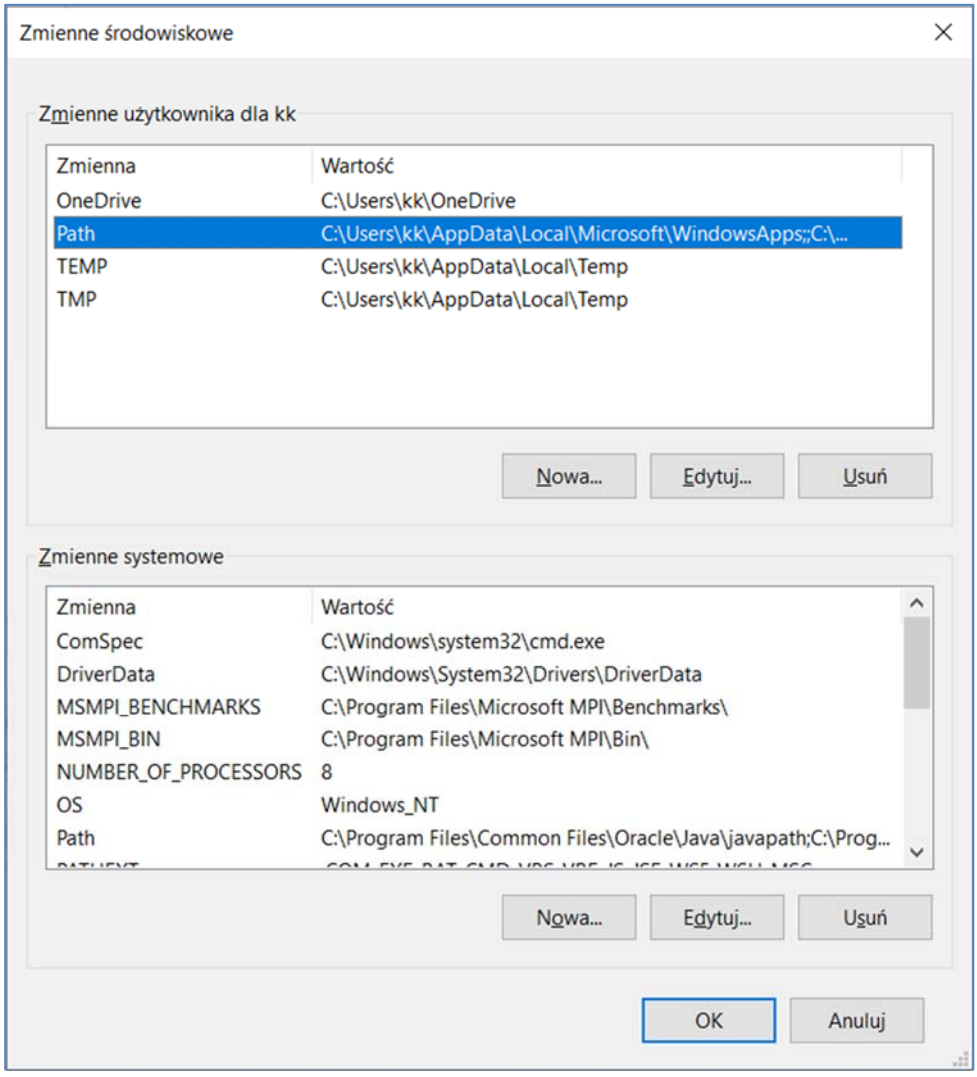
Rysunek 2.9 Wyszukiwanie zmiennych środowiskowych

Wyświetli się okno właściwości systemu. Wchodzimy w „**Zmienne środowiskowe…**”.



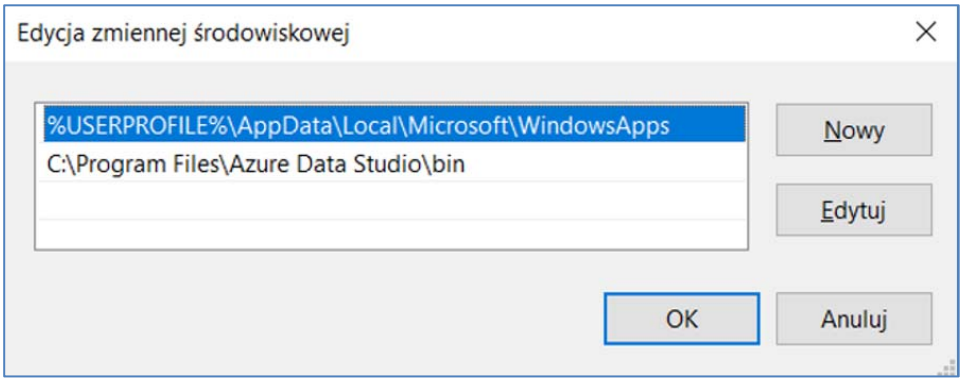
Rysunek 2.10 Okno właściwości systemu

W tabeli zmiennych użytkownika zaznaczamy „**Path**” i klikamy „**Edytuj**”.



Rysunek 2.11 Okno zmiennych środowiskowych

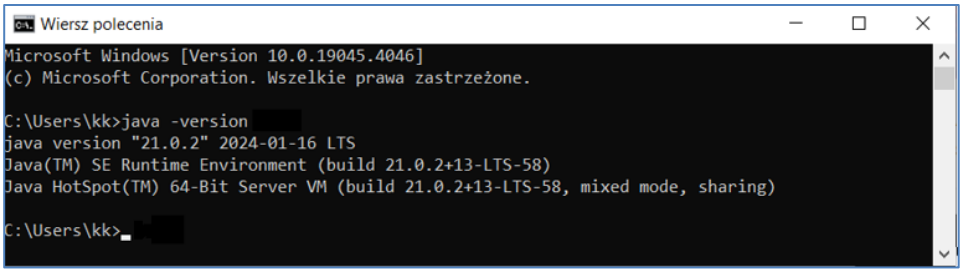
Teraz dodamy nową zmienną. Klikamy „**Nowy**” i wklejamy naszą ścieżkę.



Rysunek 2.12 Dodanie nowej zmiennej

Na koniec zatwierdzamy wszystkie otwarte do tej pory okna przyciskiem „**OK**”.

Ostatnim krokiem jaki możemy wykonać jest sprawdzenie czy JDK zostało pomyślnie zainstalowane. W tym celu uruchamiamy wiersz poleceń CMD i wpisujemy komendę „**java -version**”. Jeśli wszystko przebiegło pomyślnie wyświetli się nam zainstalowana wersja Javy.



Rysunek 2.13 Wyświetlenie informacji o zainstalowanej wersji Javy w konsoli CMD

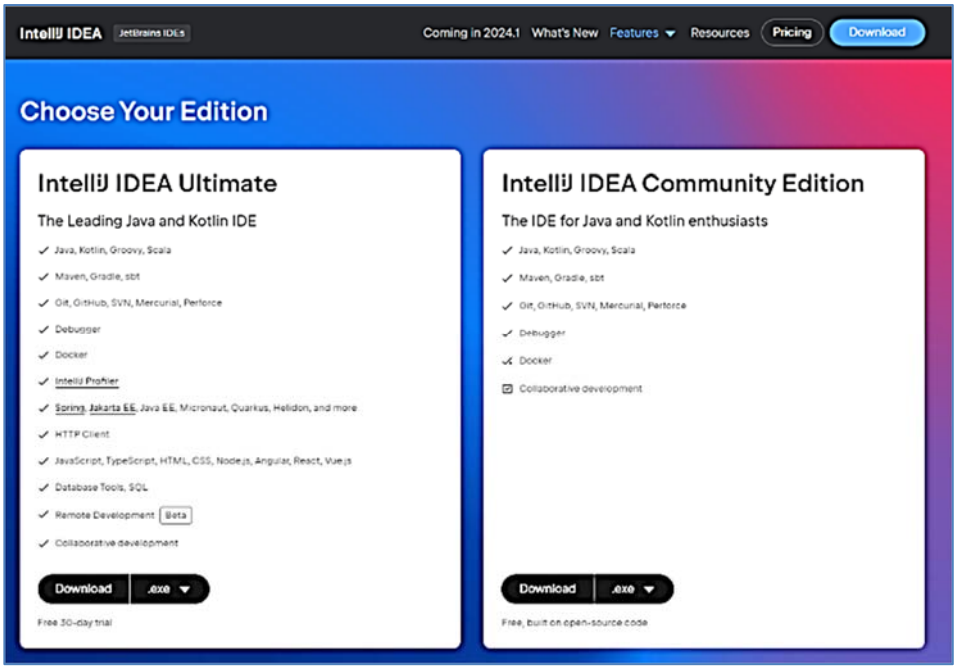
Teraz nasz system jest gotowy do kompilacji i uruchamiania aplikacji napisanych w Javie. Potrzebujemy jeszcze edytora za pomocą którego będziemy zdolni do pisania kodu. Teoretycznie mógłby nam do tego posłużyć zwykły notatnik jednak jest on niepraktyczny i programowanie w nim byłoby to męczące i pracochłonne. Dużo lepiej sprawdzi się dedykowane środowisko programistyczne.

2.2 Instalacja Ide IntelliJ dla systemu Windows

Proces instalacji zaczynamy od pobrania środowiska ze strony:

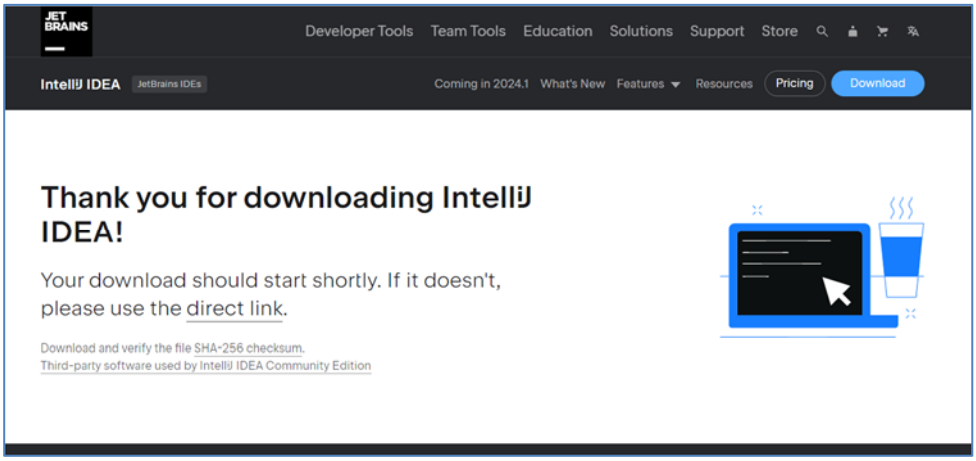
https://www.jetbrains.com/idea/download

Do wyboru mamy wersję **Ultimate** oraz **Community Edition**. Ta pierwsza wymaga od nas posiadania licencji. Istnieje kilka możliwości zdobycia jej bezpłatnie między innymi posiadając status studenta jednak jest ona przeznaczona głównie do tworzenia bardziej skomplikowanych i rozbudowanych projektów. Na nasze potrzeby nie jest nam potrzebna. Wybieramy zatem wersję **Community Edition**.



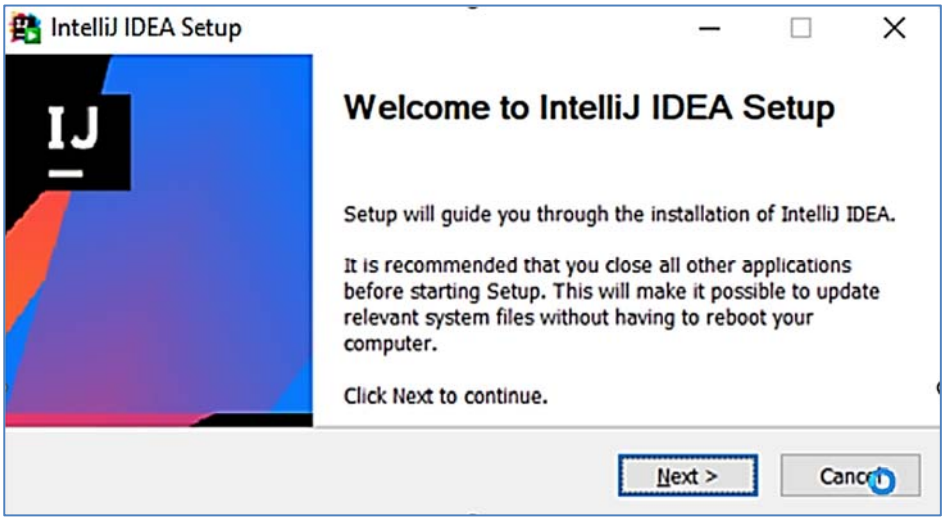
Rysunek 2.14 Pobieranie IntelliJ IDEA ze strony producenta

Klikamy "**Download**" a gdy instalator się pobierze uruchamiamy go.



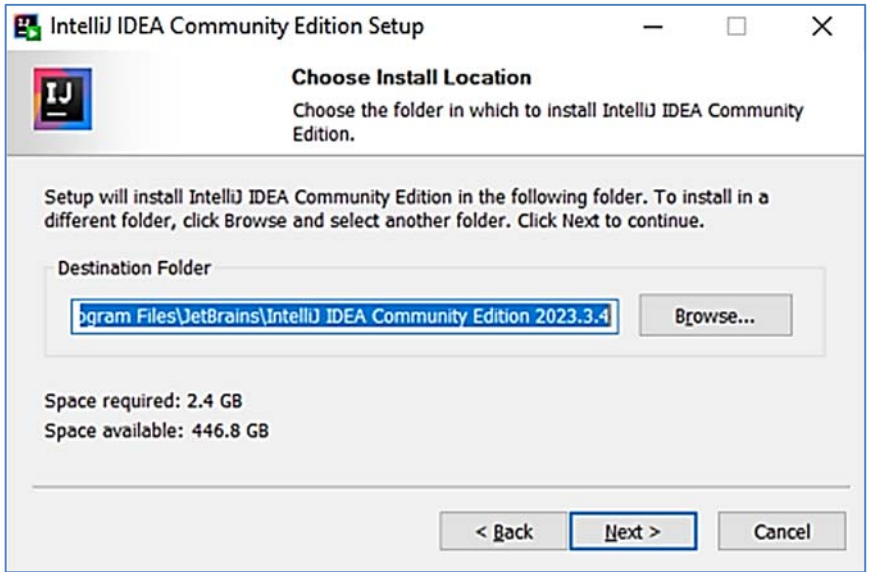
Rysunek 2.15 Strona z potwierdzeniem o pobraniu

Na oknie powitalnym klikamy „**Next**”



Rysunek 2.16 Okno powitalne instalatora IntelliJ

Teraz możemy zdecydować gdzie program zostanie zainstalowany. Następnie przechodzimy dalej.

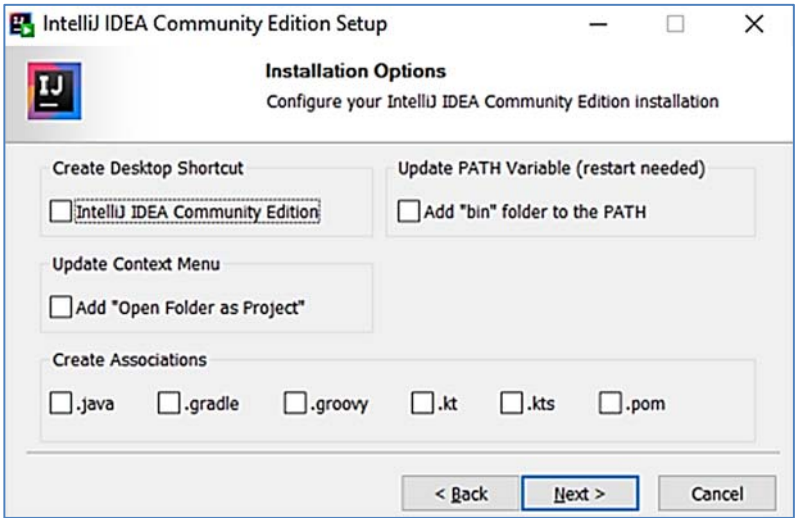


Rysunek 2.17 Wybór lokalizacji do instalacji

Na tym etapie mamy kilka opcji jakie możemy uwzględnić w instalacji. Wszystkie te ustawienia można później zmienić, ale już teraz warto zaznaczyć pola:

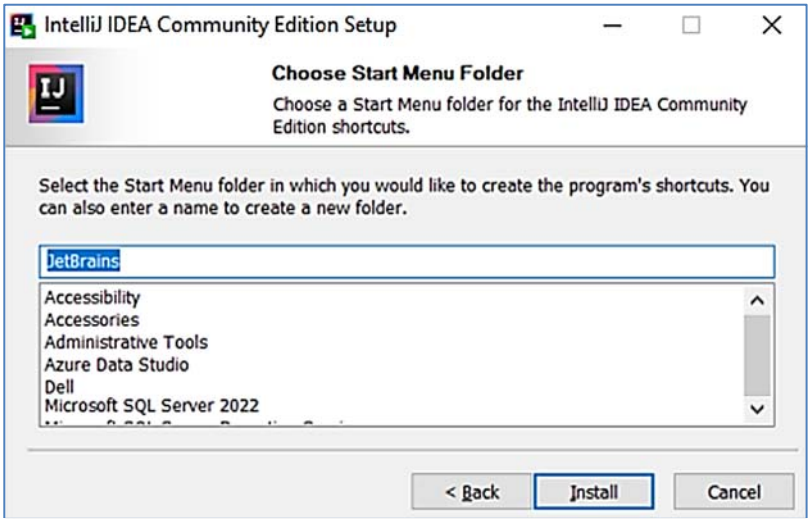
- **Add "Open Folder as Project"**;
- **Add "bin" folder to the PATH"**;
- **.java** oraz **.pom** w zakładce "**Create Associations**".

Następnie klikamy „**Next**".



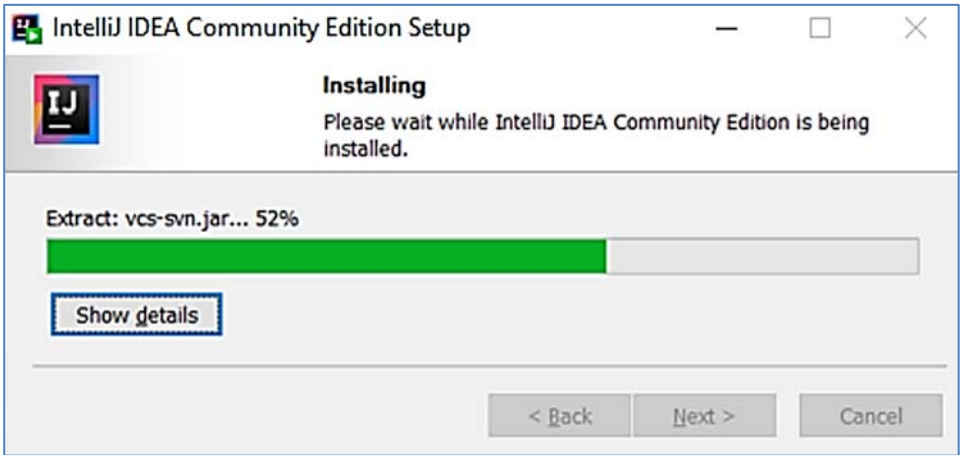
Rysunek 2.18 Okno wyboru konfiguracji dla instalacji

Pojawi się okno z pytaniem o lokalizację skrótów do aplikacji. Najlepiej pozostawić wartość domyślną. Klikamy „**Install**" i czekamy aż program zakończy instalację.



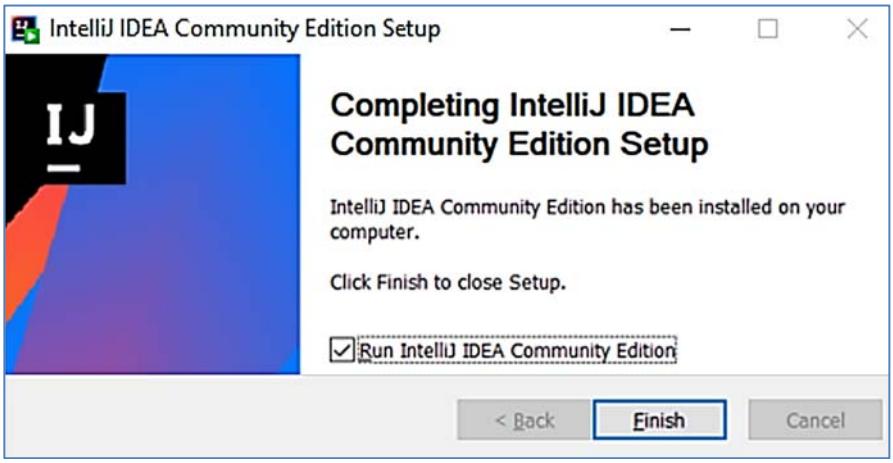
Rysunek 2.19 Okno wyboru folderu Start Menu

Na bieżąco możemy śledzić jej postęp.



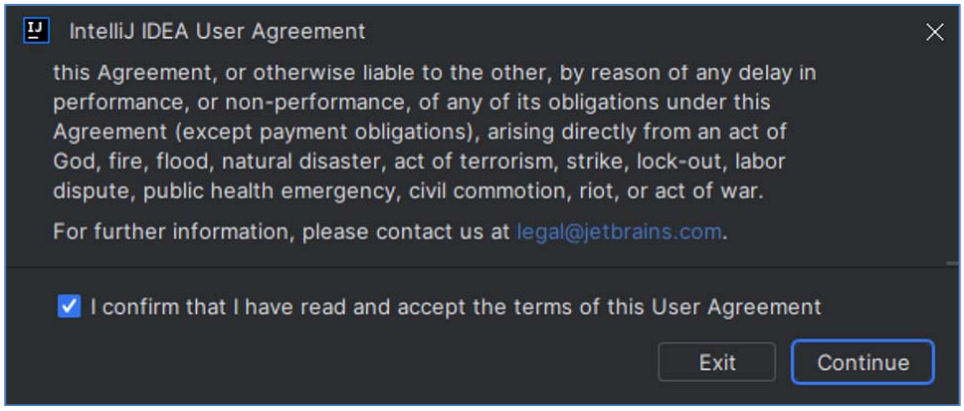
Rysunek 2.20 Pasek postępu instalacji

Gotowe. Program został zainstalowany. Teraz możemy go uruchomić.



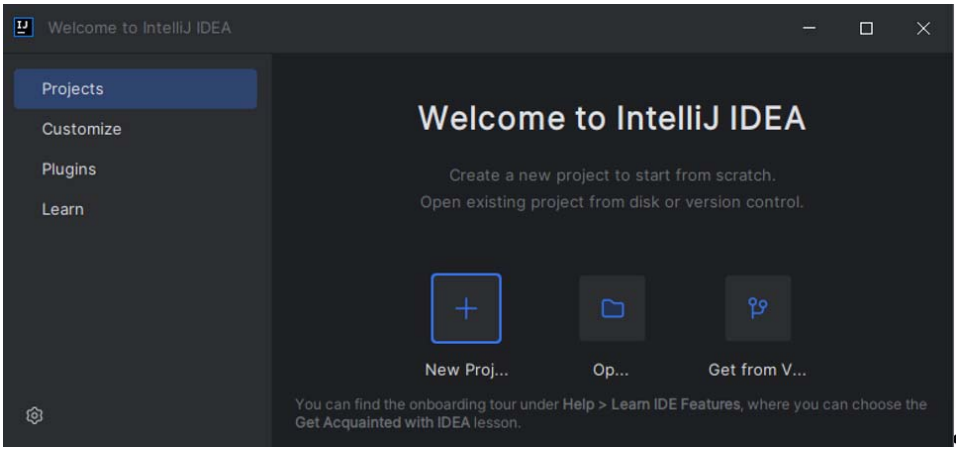
Rysunek 2.21 Instalacja zakończona pomyślnie

Przy pierwszym starcie wyświetli się nam komunikat warunków użytkowania z którym należy się zapoznać i zatwierdzić.



Rysunek 2.22 Warunki korzystania z oprogramowania

Od teraz możemy już swobodnie korzystać z programu. Zacznijmy od stworzenia nowego projektu.



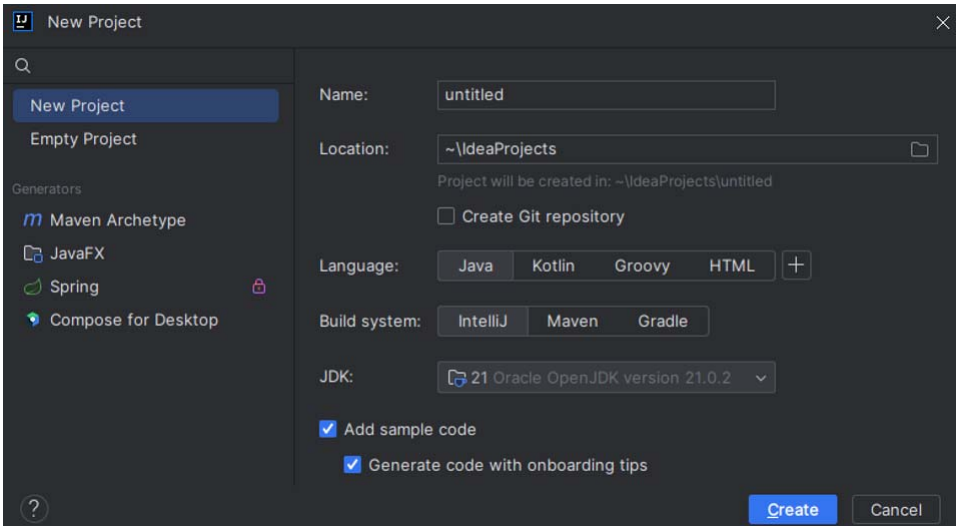
Rysunek 2.23 Panel powitalny IntelliJ IDEA

Po kliknięciu w "**New Project**" otworzy nam się kreator projektu. Musimy tutaj zdecydować o jego budowie za pomocą kilku parametrów:

- „**Name**" - podajemy tu nazwę naszego projektu,
- „**Location**" - określamy ścieżkę w której projekt będzie się znajdował,
- możemy również w tym miejscu powiązać go z repozytorium na GitHubie, ponieważ IntelliJ posiada do tego specjalny moduł,
- „**Language**" - tutaj definiujemy język w którym będziemy pisać. Dzięki temu kompilator będzie rozpoznawał składnię,

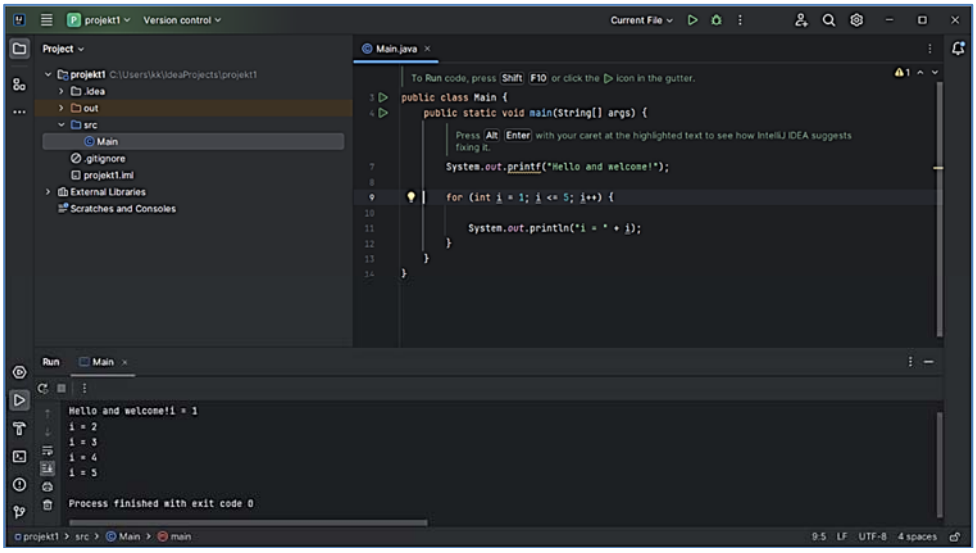
- „**Build system**” - w tym miejscu wybieramy jaki rodzaj systemu ma zarządzać budową naszego projektu,
- „**JDK**” domyślna wartość jaka się pojawi to aktualnie zainstalowana wersja Javy ale możemy również pobrać JDK bezpośrednio z tego poziomu.

Zbudujmy zatem nowy projekt o przykładowej nazwie „**projekt1**”. Lokalizację wybieramy według uznania, język ustawiamy na „**Java**”, strukturę projektu możemy zostawić domyślną czyli „**IntelliJ**”, to samo z JDK. Następnie klikamy „**Create**”.



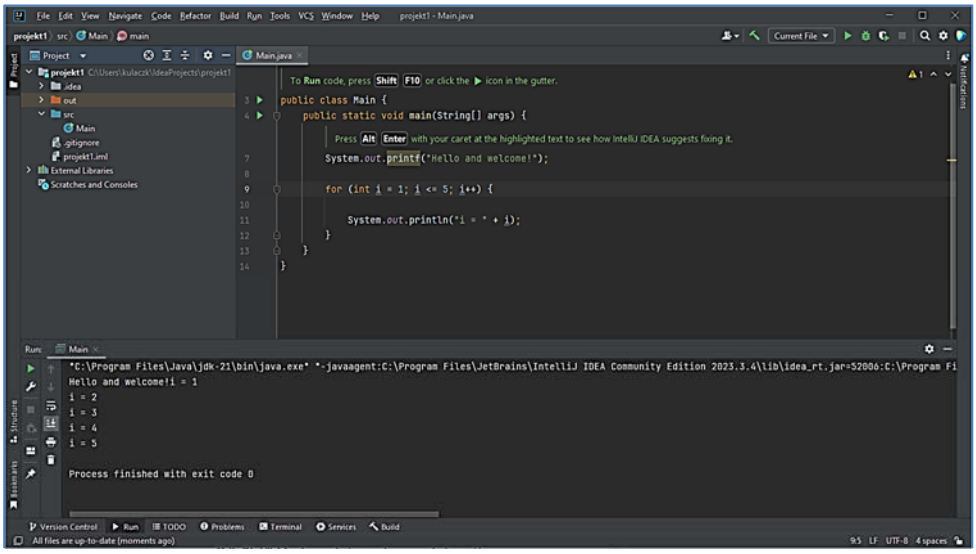
Rysunek 2.24 Kreator nowego projektu

Utworzony został nowy projekt. Od teraz większość pracy związanej z kodowaniem będzie się odbywała właśnie tutaj. Przydatne wskazówki dotyczące poruszania się po programie będą pojawiać się w kolejnych rozdziałach wraz z wprowadzaniem nowych zagadnień.



Rysunek 2.25 Widok środowiska programistycznego

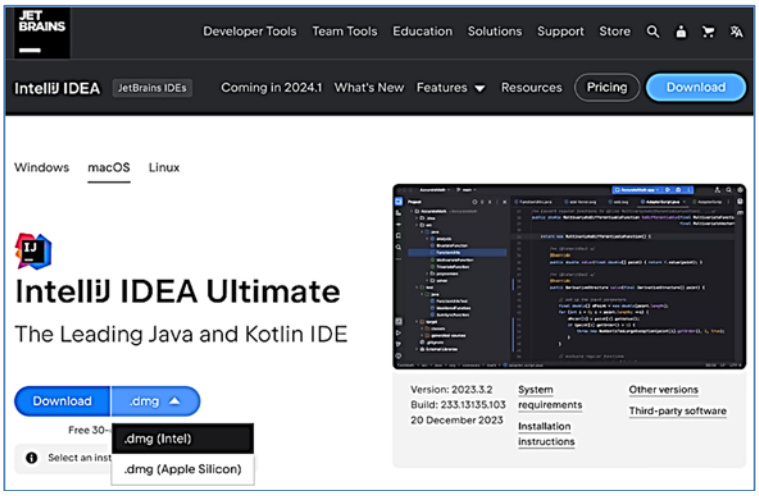
Domyślny widok programu w ostatnich latach uległ drobnym zmianom. Jeśli uważamy, że jest on dla nas nie intuicyjny, możemy przywrócić go do klasycznego wyglądu. W tym celu rozwijamy menu w lewym górnym rogu a następnie w zakładce **File** wybieramy **Settings**. W nowo otwartym oknie wyszukujemy **New UI** i odznaczamy pole **Enable New UI**. Program będzie potrzebował restartu a wygląd naszego systemu się zmieni.



Rysunek 2.26 Klasyczny wygląd programu

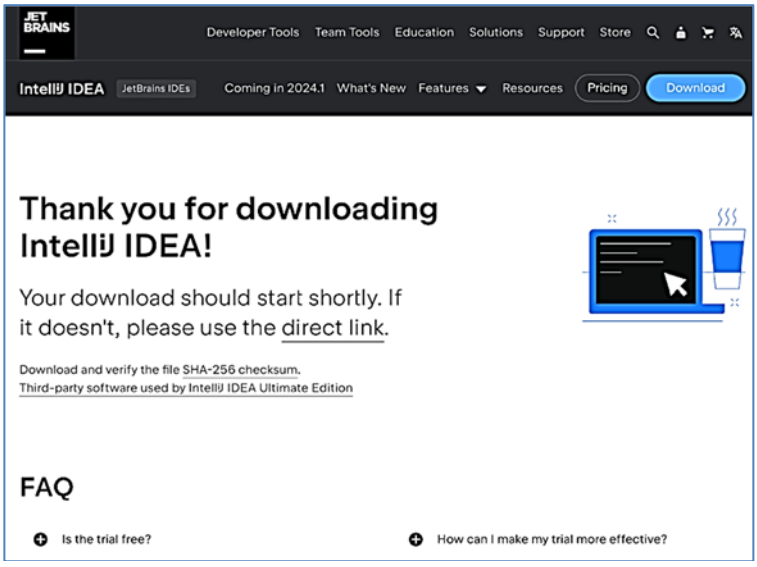
2.3 Instalacja Ide IntelliJ dla systemu MacOS

Proces instalacji środowiska Intellij IDEA na systemach MacOS jest prosty i intuicyjny. Zaczynamy od przejścia na stronę **https://www.jetbrains.com/idea/download**.



Rysunek 2.27 Pobieranie Intellij IDEA ze strony JetBrains

Wybieramy plik instalacyjny odpowiedni dla naszego systemu i klikamy „**Download**”.



Rysunek 2.28 Komunikat o rozpoczęciu pobierania programu

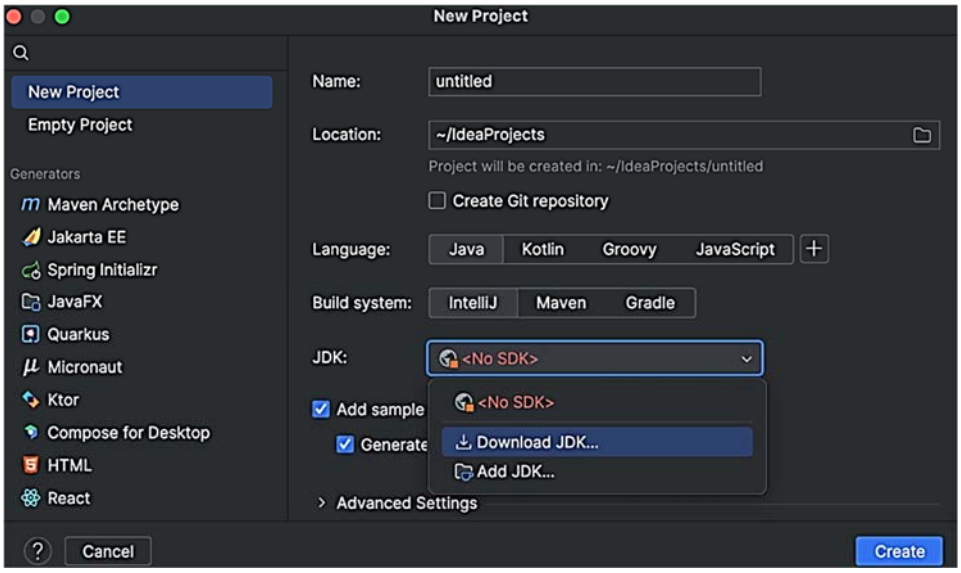
Kiedy instalator się pobierze uruchamiamy go. Wyświetli się okno w którym musimy przeciągnąć nasz program do katalogu "**Applications**".

Rysunek 2.29 Proces instalacji

To wszystko. Nasz program jest gotowy do użycia. Kiedy go uruchomimy wyświetli się nam zgoda użytkownika z którą musimy się zapoznać i zatwierdzić. Po tych czynnościach uruchomi się panel do zarządzania projektami. Od razu możemy stworzyć nowy projekt.

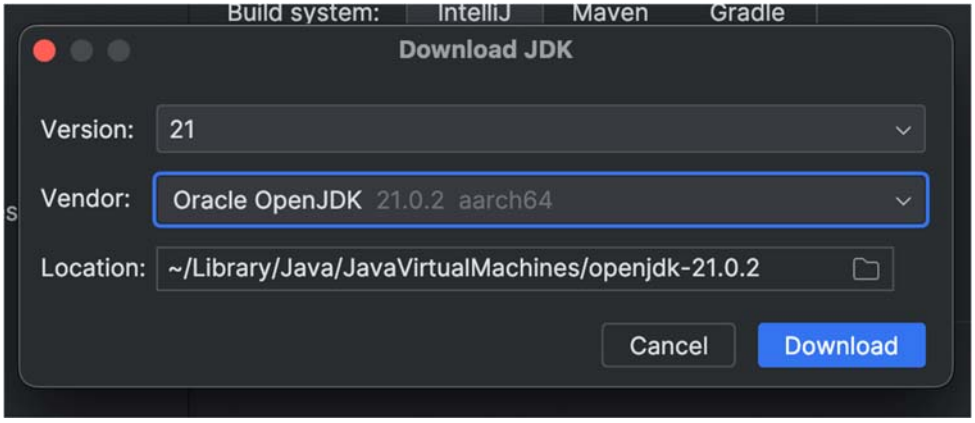
Wybierzmy „**New Project**". W polach „**Name**" i „**Location**" podajmy przykładową nazwę oraz dogodną dla nas lokalizację. Następnie w polu „**Language**" wybierzmy Java. Dzięki temu kompilator będzie radził sobie z rozpoznawaniem składni języka. Musimy jeszcze wybrać który system będzie zarządzał naszym projektem. Jest to potrzebne aby wszystkie pliki i katalogi aplikacji były nadzorowane.

Dodatkowo za pomocą takiego systemu programista w łatwiejszy sposób może dodawać nowe zależności i konfiguracje do projektu. Na początku poznawania języka nie ma znaczenia jaki system wybierzemy dlatego możemy zostawić domyślny. Na koniec pozostało nam wybrać wersję JDK. We wcześniejszym podrozdziale dotyczącym instalacji dla systemu Windows przedstawiono instalację poprzez stronę Oracle. Tutaj zrobimy to z poziomu IDE. Rozwijamy pasek wyboru w polu „**JDK**", a następnie klikamy w „**Download JDK…**".



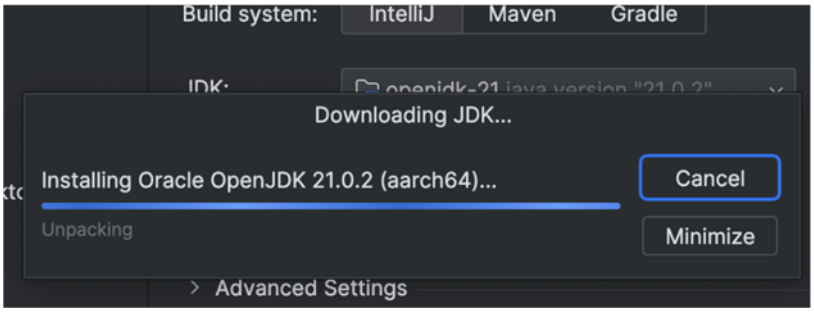
Rysunek 2.30 Kreator tworzenia nowego projektu

Pojawi się nowe okno. Wybieramy w nim wersję Javy która nas interesuje, a następnie dostawcę i lokalizację do zapisu. Na koniec klikamy „**Download**”.



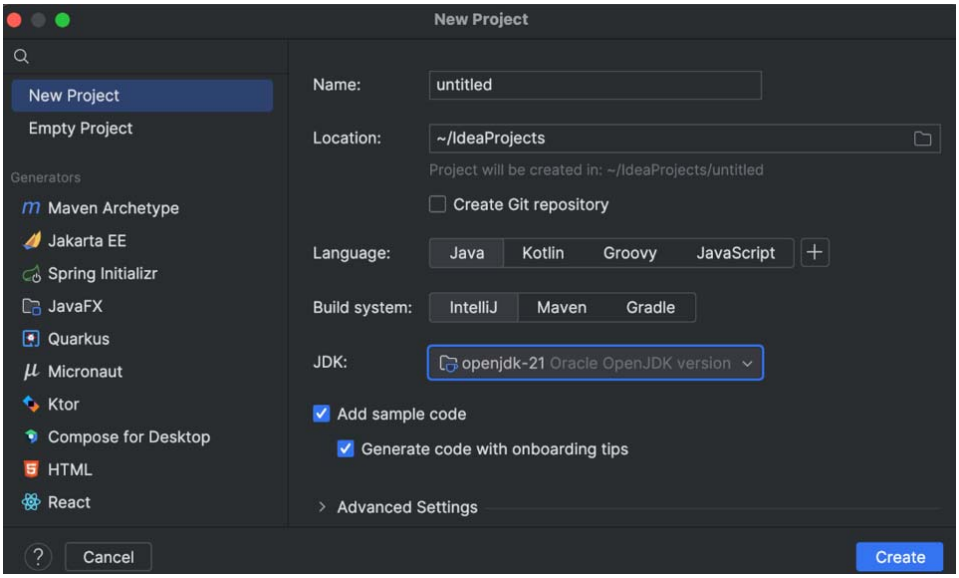
Rysunek 2.31 Wybór i instalacja JDK z poziomu środowiska IntelliJ

IntelliJ pobierze wybrane JDK i przygotuje je do użycia.



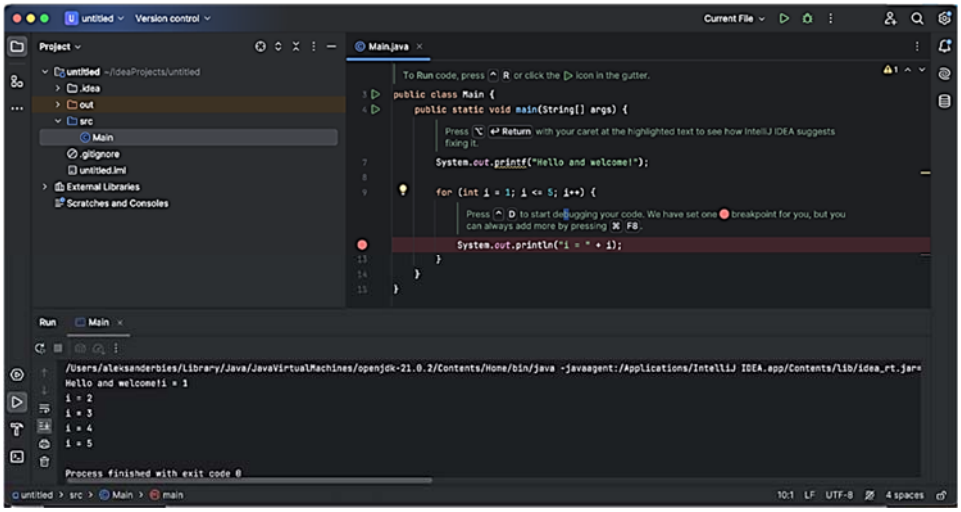
Rysunek 2.32 Pasek postępu pobierania i instalacji

Po zakończonym pobieraniu i instalacji w naszym kreatorze pojawi się możliwość wybrania nowego JDK. Zatwierdzamy je i klikamy „Create” aby utworzyć projekt.



Rysunek 2.33 Konfiguracja budowy nowego projektu

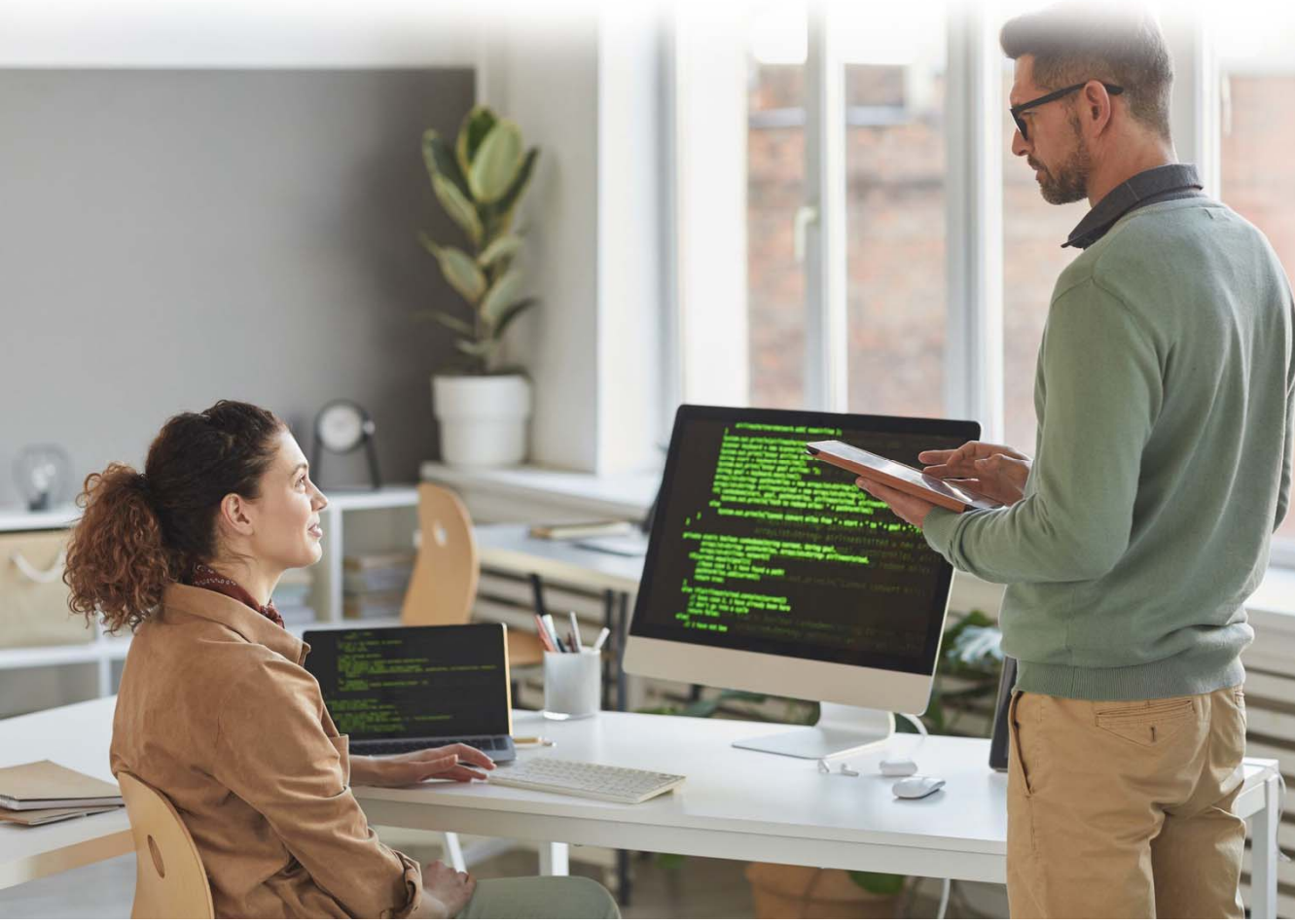
Nowy projekt uruchomi się automatycznie i wczyta potrzebne zależności. Od tego momentu jest gotowy do rozpoczęcia pracy.



Rysunek 2.34 Widok edytora

ROZDZIAŁ 3

PODSTAWY PROGRAMOWANIA W JĘZYKU JAVA



3 Podstawy programowania w języku Java

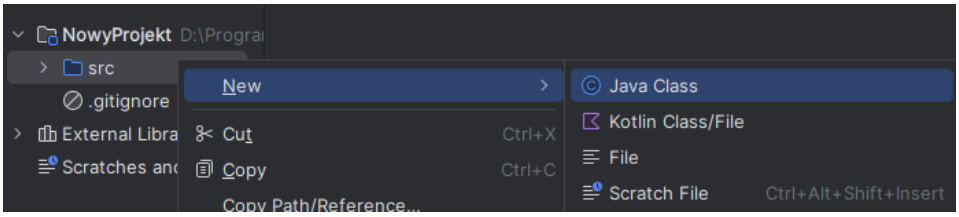
3.1 „Hello World”

Pomyślnie udało nam się przebrnąć przez pierwszy rozdział. Dowiedzieliśmy się w nim co nieco o samym języku, zrozumieliśmy, dlaczego Java jest tak popularna wśród programistów, i przygotowaliśmy nasze środowisko do pracy. Teraz nadszedł moment, który wielu z nas oczekiwało z niecierpliwością -przejdziemy do praktyki i zaczniemy pisać nasze pierwsze linie kodu w języku Java. Rozdział „Hello World” stanowi symboliczny punkt początkowy tej podróży.

Program „Hello, World!” to tradycyjny sposób rozpoczęcia nauki każdego nowego języka programowania. Prosty program wypisujący napis „**Hello, World**”! na ekranie pozwala szybko zweryfikować, czy środowisko jest poprawnie skonfigurowane, a kompilator i interpreter działają bez problemów. To również świetny sposób na zapoznanie się z podstawową składnią języka. Ten rozdział będzie jak pierwszy krok na ścieżce do opanowania sztuki programowania. Nie zostaje nam nic innego, jak zanurzyć się w kodzie i zacząć budować nasze umiejętności programistyczne od podstaw.

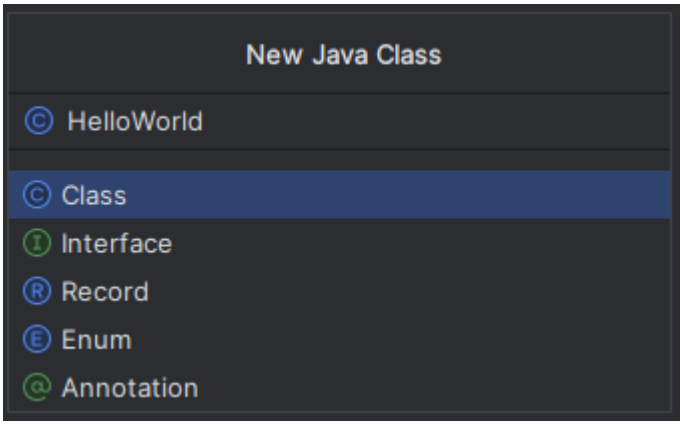
Na początku otwórzmy nasze środowisko programistyczne i stwórzmy projekt, a następnie nową klasę o nazwie „**HelloWorld**”.

Utworzyć ją możemy przez kliknięcie prawym przyciskiem myszy na „**src**” naszego projektu, następnie najechanie na napis „**New**” i wybranie opcji „**Java Class**”.



Rysunek 3.1 Tworzenie nowej klasy HelloWorld

Pojawi się poniższe okienko, w którym podajemy nazwę naszej klasy i naciskamy klawisz **Enter**.



Rysunek 3.2 Tworzenie nowej klasy HelloWorld - kontynuacja

Dla potrzeby pierwszego programu, zalecamy nazwę „**HelloWorld**”. Po utworzeniu, powinniśmy mieć zapisany poniższy kod:

```
public class HelloWorld {
}
```

Listing 3.1

„**Public class HelloWorld**” definiuje klasę o nazwie „HelloWorld”.

Pomiędzy klamrami zapiszemy teraz „**public static void main(String[] args)**” Powinno to wyglądać w ten sposób:

```
public class HelloWorld {
    public static void main(String[] args) {
    }
}
```

Listing 3.2 Kod klasy HelloWorld po dodaniu metody main

W języku Java, metoda „**main**” pełni kluczową rolę i jest punktem wyjścia każdego programu. To w niej zaczyna się wykonywanie kodu.

Na tym etapie nie będziemy jeszcze szczegółowo opisywać zapisanych przez nas słów, ale po krótce opiszemy, co one oznaczają. Mianowicie:

public – jest modyfikatorem dostępu.

- **static** – oznacza, że metoda „**main**” jest związana bezpośrednio z klasą.
- **void** – wskazuje, że metoda „**main**” nie zwraca żadnej wartości.
- **main** – to nazwa metody.

- **String[] args** - jest parametrem metody „**main**”. Pozwala przekazywać argumenty wiersza poleceń do programu podczas jego uruchamiania.

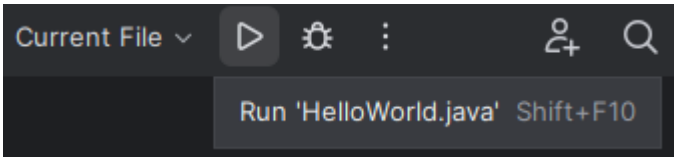
Więcej o powyższych elementach powiemy w dalszej części tej książki. Ostatecznie, linia „**public static void main(String[] args)**” wskazuje na to, że ta konkretna metoda jest punktem startowym dla programu napisanego w języku Java. Gdy program jest uruchamiany, interpreter Javy szuka tej metody, aby rozpocząć wykonywanie kodu.

Teraz wewnątrz tej linii kodu zapiszmy „**System.out.println(„Hello, World!”);**”. Polecenie to wyświetli nam napis podany w nawiasach w konsoli. Cały kod programu powinien wyglądać tak:

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

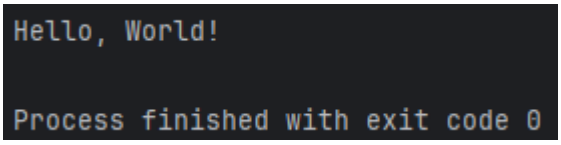
Listing 3.3 Dodanie System.out.println do metody main

Uruchommy teraz nasz pierwszy program. Odszukajmy na górnym pasku środowiska ikonę w kształcie trójkąta i kliknij ją. Możemy to także zrobić klikając skrót Shift+F10 na klawiaturze.



Rysunek 3.3 Pierwsze uruchamianie programu

Jeśli wszystko jest zapisane poprawnie, w konsoli powinno się pojawić:



Rysunek 3.4 Wynik konsoli po uruchomieniu programu

Właśnie w tej chwili odnieśliśmy sukces! Pierwszy program uruchomił się i wypisał „Hello, World!” w konsoli.

Zrozumienie powyższego kodu to pierwszy krok do opanowania Javy. Podczas gdy będziemy poznawać kolejne rozdziały, stanie się on coraz bardziej zrozumiały. Teraz, kiedy ukończyliśmy pierwszy program, możemy z radością patrzeć w przód, gotowi na kolejne wyzwania, które przyniesie nam nauka Javy.

3.2 Komunikacja z użytkownikiem. Operacje wejścia-wyjścia

Po napisaniu swojego pierwszego programu przyszedł czas na zapoznanie się z operacjami wyjścia oraz wejścia. W poprzednim programie nie wykorzystywaliśmy żadnej komunikacji z użytkownikiem. Program wyświetlał tylko informację w konsoli.

Żeby program były ciekawszy i co ważniejsze – bardziej funkcjonalny, będziemy musieli w jakiś sposób pomóc naszemu programowi „**wysłuchać nas**", to znaczy nawiązać z nami komunikacje.

W kolejnym ćwiczeniu napiszemy program będący kalkulatorem. Dane do obliczeń będziemy pobierać od użytkownika, aby aplikacja miała szerszy zakres działania.

3.2.1 Polecenia System.in i System.out

Do wczytywania danych z klawiatury służy funkcja **System.in**, zaś do wypisywania jego działań funkcja **System.out.** Do wykonania obu zadań wykorzystujemy tzw. **strumienie**.

Strumienie I/O (eng. Input/Output) służą do komunikacji z użytkownikiem w programie. W Javie mamy do czynienia z trzema rodzajami strumieni tj.:

- **System.in** – operacja wejścia (input),
- **System.out** – operacja wyjścia (output),
- **System.err** – obsługiwanie wyjątków (lub błędów), rzadziej stosowany rodzaj strumienia wyjścia.

Do utworzenia najprostszego strumienia wejścia możemy użyć **Scannera**. Będzie on obiektem klasy, który odczyta wprowadzane dane.

Zaczniemy od stworzenia klasy o nazwie **InputData** oraz umieszczenia w niej metody **main**. Następnie musimy utworzyć obiekt typu **Scanner**.

Potrzebna nam będzie instancja tego obiektu. Instancja to nic innego jak egzemplarz czy wystąpienie niezależnego kodu, który jest zgodny ze wzorcem. W metodzie **main** umieszczamy poniższy kod:

```
Scanner sc = new Scanner(System.in);
```

Listing 3.4 Utworzenie zmiennej sc1 typu Scanner

Przyjrzyjmy się temu zapisowi dokładniej.

- **Scanner** - nazwa klasy, której instancje chcemy utworzyć;
- **sc1** - nazwa obiektu (instancji);
- „=” - operator przypisania;
- **słowo kluczowe new** - dzięki niemu tworzymy obiekty;
- ponownie nazwa klasy.

Jak można zauważyć – w nawiasie znajduje się operacja wejścia **System.in,** która posłuży do wczytania danych wprowadzonych z klawiatury.

Na końcu umieszczamy średnik – jest on w pewnym sensie odpowiednikiem kropki w zdaniu i oznacza koniec danej linii.

Warto o nim pamiętać, ponieważ jego brak wygeneruje błąd.

Prawdopodobnie nazwa **Scanner** będzie podświetlona na czerwono. Dzieje się tak dlatego, że nie zaimportowaliśmy odpowiedniego pakietu, który udostępni nam obiekt **Scanner**. Na samej górze (nad klasą **InputData**) trzeba zamieścić:

```
import·java.util.
```

Listing 3.5 Zaimportowanie pakietu Scanner

Cały kod powinien teraz wyglądać następująco:

```
import java.util.Scanner;
public class InputData {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
    }
  }
```

Listing 3.6 Pełen kod klasy InputData

Klasa **Scanner** jest bardzo uniwersalna i zezwala na skanowanie tekstu w poszukiwaniu liczb, napisów i innych obiektów.

W ćwiczeniu stworzone zostały właśnie obiekty klasy Scanner, które posłużą nam do wywoływania metody pozwalającej odczytywać różne typy danych.

Ten schemat tworzenia obiektów jest warty zapamiętania, ponieważ będzie używany przy tworzeniu większości obiektów.

Mamy więc już element odpowiadający za wczytywanie danych wpisanych przez użytkownika. Teraz czas na stworzenie programu, umożliwiającego wyświetlenie wartości w konsoli.

Do tego przyda się metoda **System.out**.

Zanim jednak do tego przejdziemy musimy jeszcze dodać do kodu jedną bardzo ważną instrukcję. Na ten moment program zakończy się od razu po uruchomieniu bez możliwości podania żadnych danych. Dzieje się tak, ponieważ sam zadeklarowany obiekt nic nie robi. Po prostu jest.

Wartość, którą chcemy wpisać do programu musimy gdzieś przechować.

Do przechowywania danych w programie służą zmienne. Aby umieścić dane w zmiennej należy ją przypisać w miejsce, które przechowa nam jej wartość.

Do przechowywania danych służą zmienne. One szczegółowo zostaną opisane dopiero w kolejnym podrozdziale, ale na potrzebę przykładu – skorzystamy z nich już teraz.

W funkcji **main** dopisujemy poniższy kod:

```
String tekst = sc.nextLine();
```

Listing 3.7 Dodanie zmiennej typu String oraz przypisanie jej wartości wczytanej z kolejnej linii

Teraz kod będzie wyglądał tak:

```
import java.util.Scanner;
public class InputData {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
String tekst = sc.nextLine();

    }
  }
```

Listing 3.8 Kod klasy InputData po dodaniu zmiennej tekst

Po uruchomieniu trzeba wpisać tekst, a następnie wcisnąć **Enter**.

Program nie zamyka się już od razu po jego uruchomieniu, a dopiero wtedy, gdy przekażemy mu dane i zatwierdzimy klawiszem **Enter**.

3.2.2 Typ String

Typ String służy do przechowywania ciągów znaków, czyli napisów.

Po znaku przypisania „=” wpisujemy wartość, którą ma przechowywać zmienna o podanej nazwie.

Można zauważyć, że string jest podkreślony zieloną linią. Nie należy się tym martwić, gdyż w **IntelliJ** dzieje się tak, kiedy podamy słowo, którego nie ma w słowniku angielskim.

W kolejnej linijce po znaku "=" jest odwołanie do naszego **Scannera** o nazwie **sc** i po kropce metoda, która zostanie wykonana. **Metoda sc.nextLine()** zatrzymuje nam działanie programu i czeka, aż użytkownik wpisze coś w konsoli. Po wprowadzeniu i wciśnięciu klawisza **Enter** program wznawia działanie. Metoda ta czyta podany przez nas wyraz z klawiatury i przypisuje jego wartość do zmiennej „tekst".

Teraz użyjemy wspomnianego chwilę temu **System.out**. W nowej linii wpisujemy podany kod:

```
System.out.println(słowo);
```

Listing 3.9 Metoda System.out.println wypisująca słowo

Całość powinna prezentować się następująco:

```
import java.util.Scanner;

public class InputData {
public static void main(String[] args) {
Scanner sc1 = new Scanner(System.in);
String tekst = sc.nextLine();
System.out.println(słowo);
    }
  }
```

Listing 3.10 Cały kod klasy InputData po dodaniu System.out.println

W poprzednim ćwiczeniu przedstawiona została metoda **System.out.println()**. Wtedy podany tekst był wpisywany w cudzysłowie. W tym programie wypisana zostanie wartość zmiennej przechowywanej przez „**słowo**".

Uruchom teraz program i wpisz dowolny tekst a następnie wciśnij **Enter**.

W kolejnej linii została wypisana wartość zmiennej, czyli tekst który został podany przez użytkownika z klawiatury. Zmodyfikujemy teraz program tak, aby był bardziej funkcjonalny. Zmień nazwę zmiennej „**słowo**" na „**imie**". Oczywiście nie zapomnijmy też jej zmienić w metodzie **System.out.println().** W nawiasie tej metody, przed zmienną „**imie**" dopisz w cudzysłowie „**Witaj** " i postaw znak „+".

Ważne jest, abyś po słowie "**witaj**" zostawił spację. Jeśli tego nie zrobisz to tekst połączy nam się z podanym imieniem, a to sprawi, że wypisana w konsoli zawartość nie będzie czytelna. Kod powinien się prezentować teraz tak:

```
import java.util.Scanner;

public class InputData {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
String imie = sc.nextLine();
System.out.println(„Witaj „ + imie );
```

Listing 3.11 Cały kod klasy InputData po zaktualizowaniu metody System.out.println

Uruchom program, podaj twoje imię i zobacz rezultat.

Program się z nami przywitał!

Takich kombinacji jest nieskończenie wiele i możesz to zrobić w taki sposób jaki tylko będziesz chciał. Aby nie musieć za każdym razem pisać System.out.println() możemy zastosować jedną z dodatkowych funkcjonalności **IntelliJ**.

3.2.3 Komentarze

Załóżmy, że dla ułatwienia chcemy gdzieś w programie zanotować, co robi podana linijka kodu. Istnieje też taka możliwość, by bez ingerowania w działanie programu sporządzać sobie notatki. Do tego służą komentarze.

Napiszemy zatem w komentarzu co robi ostatni wiersz programu, by łatwiej było nam przyswoić informacje na początku naszej przygody z programowaniem. Komentarze przydają się również wtedy, gdy otwieramy program po dłuższym czasie. Dzięki nim jesteśmy w stanie zorientować się do czego służą poszczególne fragmenty kodu czy np. utworzone funkcje, klasy, zmienne.

W celu utworzenia komentarza robimy odstęp pomiędzy ostatnią, a przedostatnią linijką w funkcji głównej main.

Wpisujemy dwa razy znak: "//" - dwa ukośniki. Po nich możemy pisać treść komentarza. Program będzie ignorował te wpisy. Są to informacje tylko dla ciebie. W komentarzu możesz zapisać informacje na przykład w podany sposób.

```
import java.util.Scanner;

public class InputData {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
String imie = sc.nextLine();
//metoda witająca się  z użytkownikiem
System.out.println(„Witaj „ + imie );
    }
        }
```

Listing 3.12 Zastosowanie komentarza w programie

Komentarze są zapisane domyślnie szarą czcionką.

Co jednak, gdy komentarz jest na tyle długi, że nie mieści nam się w długości ekranu? Z pomocą przychodzą tzw. **Komentarze wielolinijkowe**. Zapisujemy je przy użyciu ukośnika i gwiazdki na początku - "/*", a kończymy je w odwrotny sposób - "*/". Wygląda to, w ten sposób:

```
import java.util.Scanner;

public class InputData {
public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
String imie = sc.nextLine();

/*
metoda
witająca się  z użytkownikiem
*/

System.out.println(„Witaj „ + imie );
    }
        }
```

Listing 3.13 Zastosowanie komentarza wieloliniowego w programie

Przy tworzeniu tego typu komentarzy należy uważać, by przypadkiem nie zamknąć w nich linijki poprawnie działającego kodu. W takim przypadku program nie będzie funkcjonował prawidłowo.

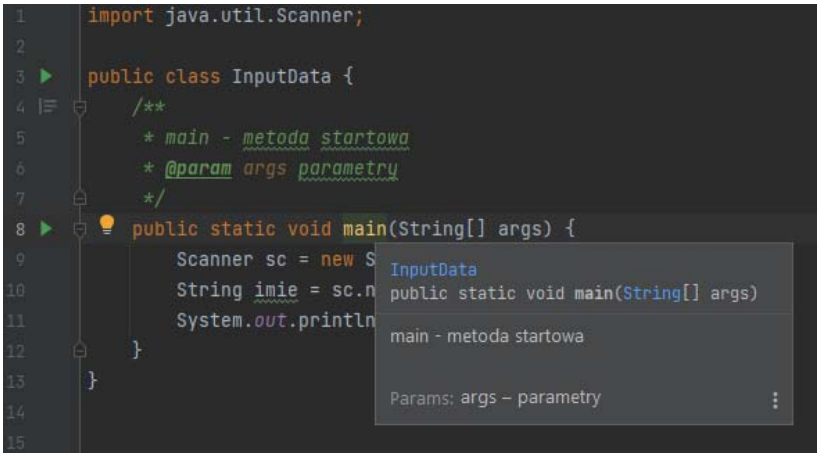
Istnieje jeszcze trzeci rodzaj komentarzy tzw. **komentarze dokumentujące**.

Przypomina on komentarz wieloliniowy, jednak ma inne zastosowanie. Tworzymy go tak samo jak komentarz wieloliniowy, ale dodajemy jeszcze jedną gwiazdkę przy otwarciu.

```
import java.util.Scanner;

public class InputData {

/**
* main - metoda
* @paramargs parametry
**/

public static void main(String[] args) {
Scanner sc = new Scanner(System.in);
String imie = sc.nextLine();
System.out.println(„Witaj „ + imie );
    }
        }
```

Listing 3.14 Zastosowanie komentarza dokumentującego w programie

W pierwszej linijce podajemy opis oraz działanie naszej metody. Następnie automatycznie wygeneruje nam się napis **"@param args**", po którym wpisujemy parametry. Aby zobaczyć rezultat tego komentarza musimy kliknąć myszą na metodę **main**, a następnie z górnego paska **IntelliJ** wybrać **Widok** (eng. View) i **Szybka Dokumentacja** (eng. Quick Documentation) lub drogą na skróty – użyć kombinacji **CTRL + Q.**



Rysunek 3.5 Rezultat zastosowania komentarza dokumentującego w programie

Jak widzimy wyświetla nam się w okienku podany przez nas komentarz. Warto zaznaczyć, że komentarze są przede wszystkim opcjonalne i używanie ich nie jest wymagane. Możemy je stosować, gdy czujemy potrzebę zanotowania czegoś lub tymczasowego wyłączania z działania danego fragmentu kodu. Nie

należy też przesadzać z komentarzami i używać ich zbyt często, gdyż kod nie będzie przejrzysty.

3.3 Typy danych, zmienne i stałe

Napisaliśmy już pierwszy program w języku Java. Wiemy też jak działają operacje wyjścia i wejścia oraz potrafimy umieszczać komentarze.

Zapoznamy się teraz ze **zmiennymi**.

Zmienna jest jednym z podstawowych elementów programowania. Bez ich udziału ciężko stworzyć bardziej zaawansowany i rozbudowany program komputerowy. Zmienna jest to obszar pamięci, w którym podczas wykonywania programu przechowywane są wartości danego typu.

Podstawowe cechy zmiennej to:

- **Typ** - rodzaj informacji, który będzie przechowywany
- **Wartość** - czyli informacja, która jest przechowywana w zmiennej
- **Nazwa** – identyfikuje zmienną, pozwala na odróżnienie jej od innych

Najprościej mówiąc zmienną możemy porównać do pojemnika służącego do przechowywania wartości. W prawie wszystkich językach programowania, zmienne powinny mieć ustalony typ danych, który przechowują.

Nazwa zmiennej powinna się zaczynać od litery oraz nie może zawierać innych znaków niż liczby, litery i znak podkreślenia („_”).

W Javie istnieje również około 50 słów, które nie mogą być nazwą zmiennej. Oto one:

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void

class	finally	long	strictfp	volatile
const	float	native	super	while

Tabela 3.1 Wyrażenia, które nie mogą być zastosowane jako nazwa zmiennej

Słowa te to nazwy funkcji, zdarzeń czy typów. Gdyby były używane jako nazwy zmiennych to komputer miałby problem z rozróżnieniem, dlatego zostały zakazane. Poniżej zostały wypisane ogólne zasady nazywania zmiennych:

- nazwa zmiennej może zawierać litery, liczby, podkreślenia oraz symbol dolara,
- nazwa zmiennej musi się zaczynać od litery,
- nazwa zmiennej powinna się zaczynać małą literą, dolarem lub znakiem podkreślenia,
- nazwa zmiennej nie może się zaczynać białym znakiem,
- nazwy zmiennych są wrażliwe, to znaczy: „mojazmienna” oraz „mojaZmienna” to dwie zupełnie różne zmienne,
- nazwy zmiennych nie mogą być wcześniej przedstawionymi słowami (while, abstract itd.).

Wszystkie wartości w języku Java są obiektami danego typu.

W Javie wyróżniamy kilka typów danych. Zostały one zamieszczone w tabeli poniżej:

Nazwa typu	Liczba bajtów	Dopuszczalne wartości	Wartość domyślna	Opis	Przykład
boolean	1	true lub false	false	przyjmuje wartości logiczne	true
byte	1	-128 do 127	0	wartość całkowita 8-bitowa	1
char	2	0 do 65536	'x0'	kod znaku w 16-bitowym	'a'
short	2	-32 768 do 32 767	0	wartość całkowita 16-	2
int	4	-2^31 do 2^31-1	0	wartość całkowita 32-	-2
long	8	-2^63 do 2^63-1	0	wartość całkowita 32-	1L

float	4	-3.xE+38 do	0.0f	wartość	3.14f
double	8	-1.xE+308 do	0.0d	wartość	1.23d

Tabela 3.2 Podstawowe typy danych w Javie

Ze względu na budowę typy danych w języku Java dzielimy na:

- **Typy prymitywne bądź proste** – int, char, boolean, float, double;
- **Typy nieprymitywne** - String, tablice i klasy;

Ze względu na rodzaj przechowywanych danych dzielimy je na:

- **liczby całkowite** - byte, short, int oraz long;
- **liczby rzeczywiste** - float i double;
- **wartości logiczne** – boolean;
- **znaki Unicode** - char;

Wśród typów całkowitych najbardziej powszechnymi typami są **long** oraz **int**. Są one preferowane i najczęściej stosowane. Rzadziej mamy do czynienia z typami **short** i **byte**.

Jeśli chodzi o liczby rzeczywiste (tzw. zmiennoprzecinkowe) to stosowane są głównie przy różnych obliczeniach naukowych oraz w przetwarzaniu grafiki.

Typ **float** od typu **double** różnią się ilością przechowywanych cyfr dziesiętnych. **Float** przechowuje ich tylko sześć lub siedem, a **double** około piętnastu.

Typy predefiniowane (inaczej typy wbudowane), to typy, które kompilator obsługuje w sposób specjalny. Jednym z przykładów takiego typu jest właśnie **int** jako predefiniowany typ dla wartości całkowitych, które zajmują w pamięci 32 bity.

```
String nazwa = „Minecraft”;
Int rok_wydania = 2011;
```

Listing 3.15 Zadeklarowanie zmiennej i zapisanie działania jako wynik

W poprzednim podrozdziale poznaliśmy funkcję **System.out.print()**, ale nie została ona tam szczegółowo opisana. Funkcja ta służy do wyświetlania tekstu wewnątrz konsoli.

Wewnątrz tej funkcji mamy możliwość wpisania również symboli czy znaków specjalnych. Jednym z takich znaków jest „\n”, który przeniesie wpisany po nim

tekst do następnej linii. Taki sam efekt uzyskamy stosując funkcję: **System.out.println().** Różni się ona od poprzedniej tym, że zawiera przed nawiasami litery ln, które oznaczają, że funkcja wyświetli tekst w nowej linii. Poniżej przykłady działania:

```
System.out.print(„Witaj”);
System.out.print(„kolego”);
```

Listing 3.16 Przykład działania funkcji System.out.print();

Wynik działania programu:

```
Witajkolego
Process finished with exit code 0
```

Rysunek 3.6 Wynik konsoli poprzedniego działania

Jeżeli użyjemy metody **System.out.println()** to otrzymamy poniższy rezultat:

```
Witaj
kolego

Process finished with exit code 0
```

Rysunek 3.7 Wynik konsoli poprzedniego działania z użyciem System.out.println

Jak widzisz funkcja użyta w przykładzie pierwszym wyświetla nam tekst w jednej linii, z kolei ta z drugiego przykładu – każdy tekst osobno. Oprócz tekstu umieszczonego w cudzysłowie możemy też podać nazwę zmiennej lub wpisać samo wyrażenie, którego wynik chcemy uzyskać na przykład:

```
int liczba = 4+2;
System.out.println(liczba);
System.out.println(„6”);
System.out.println(4+2);
```

Listing 3.17 Wypisanie liczby 6 różnymi sposobami

Po zapisaniu i uruchomieniu powyższego kodu w konsoli pojawi się trzykrotnie wartość 6.

Kolejnym przykład predefiniowanego typu to **String**. Reprezentuje on ciągi znaków, na przykład “**Java**”.

```
String napis=”Java”;
System.out.println(napis);
```

Listing 3.18 Zadeklarowanie zmiennej typu String i wypisanie jej zawartości

Dla typu **String** Java posiada szereg metod służących do modyfikowania tekstu.

```
String napis = „          Java”;
System.out.println(napis.toUpperCase());
System.out.println(napis.toLowerCase());
System.out.println(napis.trim());
System.out.println(napis.replace('a','o'));
```

Listing 3.19 Działania na typie string

Rysunek 3.8 Wynik konsolowy działań na stringach

Na początku zadeklarowaliśmy zmienną typu string o nazwie napis, która przechowuje nam wartość “ Java”. Nie przypadkowo dodaliśmy większą ilość spacji przed napisem.

Pierwsza metoda: **toUpperCase()** sprawia, że cały napis będzie wyświetlony drukowanymi literami.

Druga z metod - **toLowerCase()** działa przeciwnie do poprzedniej. Cały tekst zostanie napisany małymi literami.

Trim() usuwa z tekstu wszystkie spacje jakie znajdują się przed słowem i po nim, tak by został sam tekst. Przydaje się to na przykład przy tworzeniu programu, w którym musimy podać login, bądź hasło. Jeśli użytkownik poda nam login, który będzie zawierał spację na początku to aby uniknąć kłopotów - możemy zastosować metodę **trim**, która usunie wszystkie spacje.

Ostatnią metoda prezentowana w przykładzie to **replace().** Zamienia ona wybraną literę na inną. W powyższym przykładzie zamieniliśmy literę “**a**” na “**o**”, przez co konsola zwróciła nam “**Jovo**” zamiast “**Java**”.

Stringi jak i inne słowa możemy łączyć w całe zdanie przez użycie znaku “+”:

```
String imie = „Adam”;
String nazwisko = “
Kowalski”;
System.out.println(„Nazywam się: ” + imie + „ ” +
nazwisko);
```

Listing 3.20 Łączenie dwóch stringów w jedno zdanie w metodzie System.out.println

W wyniku działania programu w konsoli otrzymamy:

```
Nazywam się: Adam Kowalski

Process finished with exit code 0
```

Rysunek 3.9 Wynik konsolowy po wypisaniu System.out.println

Jak widzimy wypisaliśmy dwa stringi w jednym zdaniu. Między jedną zmienną, a drugą dodaliśmy pusty cudzysłów, aby je oddzielić. Działają one po prostu jako znak spacji. Łączyć w jedno zdanie można też inne typy danych, np. **string** z **intem**:

```
String nazwa = „Minecraft”;
Int rok_wydania = 2011;
System.out.println(„Gra ”+ nazwa + „ powstała w ” +
rok_wydania + „ roku.”);
```

Listing 3.21 Użycie typu string i int w jednym zdaniu przy zastosowaniu System.out.println

Konsola zwróci nam:

```
Gra Minecraft powstała w 2011 roku.

Process finished with exit code 0
```

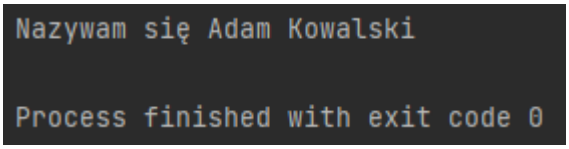
Rysunek 3.10 Wynik konsoli po wypisaniu System.out.println

W tym przykładzie z dwie zmienne „**imie**” oraz „**nazwisko**”, możemy połączyć ze sobą w całość. Jedna zmienna będzie wówczas przechowywała zarówno imię jak i nazwisko. Dwie zmienne typu string łączymy w następujący sposób:

```
String imie = „Adam”;
String nazwisko = „Kowalski”;
String pelnaNazwa = imie + nazwisko;
System.out.println(„Nazywam się „ + pelnaNazwa );
```

Listing 3.22 Dodanie do siebie dwóch zmiennych typu string i przypisanie wyniku do zmiennej

W konsoli ujrzymy:



Rysunek 3.11 Wynik konsoli po dodaniu do siebie dwóch zmiennych i wypisaniu jej wartości

Innym typem predefiniowanym jest typ **boolean**. Dopuszcza on tylko dwie wartości – **false (fałsz)** albo **true (prawda).** Domyślną wartością zmiennej tego typu jest **false**.

Przykład działania zmiennej typu **boolean.**

```
Boolean napis = true;
System.out.println(napis);
```

Listing 3.23 Przykład użycia typu boolean

Zmienna zwróci wartość true co możemy zaobserwować w konsoli.

Wartość **boolean** możemy sprawdzić również podczas porównywania dwóch zmiennych. Załóżmy, że dwie zmienne przechowujące liczby. Sprawdźmy, która z nich ma większą wartość.

W rozwiązaniu pomoże nam metoda **System.out.println().** Zatem:

```
float wartosc1 = 3.4f;
float wartosc2 = 7.3f;
System.out.println( wartosc1 > wartosc2 );
```

Listing 3.24 Porównanie wartości float i wypisanie wyniku w konsoli

Jak widać, w tym przykładzie posłużyliśmy się typem danych **float**. Po uruchomieniu programu, w konsoli pojawi się wartość **true.** Zapis ten pozwala na skrócenie kodu do jednej linijki.

Wystarczy, że w samym nawiasie podamy liczby i w tym przypadku znak większości.

Kod będzie wyglądał następująco:

```
System.out.println( 32 > 35 );
```

Listing 3.25 Porównanie dwóch wartości w System.out.println()

W tym ćwiczeniu uzyskamy wartość **false**.

W trakcie działania programu można zmieniać wartości zmiennych, które wcześniej przechowywały wartość. Przykładowo:

```
float liczba1 = 2.3f;
System.out.println(liczba1);
liczba1 = 2.4f;
System.out.println(liczba1);
```

Listing 3.26 Zmiana wartości zmiennej w trakcie działania programu i wypisanie wartości przed i po zmianie

Konsola w tym przypadku najpierw zwróci nam wartość **2.3**, a przy kolejnym wypisaniu, już **2.4**.

Wartości zmiennych można również zmieniać w taki sposób:

```
float liczba1 = 2.3f;
System.out.println(liczba1 + 2);
```

Listing 3.27 Inny sposób zmiany wartości zmiennej w trakcie działania programu

Do zmiennej dodaliśmy liczbę **2**, a więc jej wartość będzie teraz wynosiła **4.3**, i taką liczbę pokaże nam konsola po skompilowaniu.

Kolejnym typ danych to char. Przechowuje on pojedynczy znak, który może być liczbą, literą lub symbolem. Wartości, które przechowuje typ **char** zapisujemy w apostrofach (nie cudzysłowach!).

Typ **char** umożliwia zapisywania wartości w notacji dziesiętnej lub szesnastkowej.

Dzięki typowi **char** możemy się posługiwać również wartościami w postaci dziesiętnej lub szesnastkowej, które odpowiadają znakom w tabeli **Unicode**.

W Javie wyróżniamy też znaki specjalne, których opis i działanie zostały przedstawione w tabeli poniżej:

Zapis	Opis
\t	umieszcza tab (tabulator) w tekście
\b	umieszcza backspace w tekście, który kasuje
\n	umieszcza nową linię w tekście
\r	umieszcza powrót karetki w tekście
\f	umieszcza kanał formularza w tekście

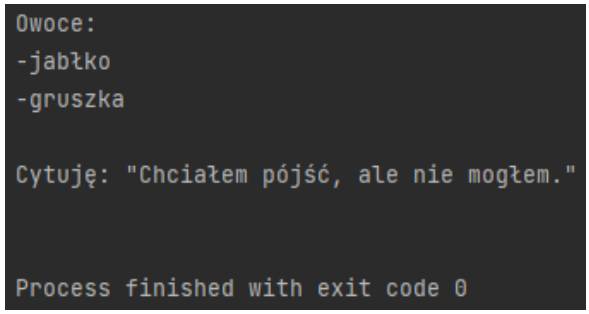
\'	umieszcza pojedynczy apostrof w tekście
\"	umieszcza podwójny apostrof w tekście
\\	umieszcza backslasha w tekście

Tabela 3.3 Znaki specjalnie w Javie

```
System.out.println(„Owoce: \n -jabłko\n-gruszka\n");
System.out.prontln(„Cytuję: \"Chciałem pójść, ale nie
mogłem..\b\"");
```

Listing 3.28 Zastosowanie znaków specjalnych w programie

Zobaczymy teraz, jak wyglądają wybrane z nich w praktyce:



Rysunek 3.12 Wynik konsolowy tego programu

W pierwszej linijce kodu w metodzie **System.out.println()**, umieściliśmy aż trzy znaki nowej linii. W wyniku czego każdy tekst wyświetlił nam się w osobnym wierszu. Znak **\n** na końcu łańcucha spowoduje odstęp pomiędzy pierwszym napisem, a kolejnym.

W drugim przykładzie zaprezentowano w jaki sposób wypisać w konsoli znak specjalny jakim jest **„”**. Bezpośrednio przed symbolem pojawił się znak \, który zmienia znak **„** ze znaku sygnalizującego początek łańcucha na symbol graficzny wypisany w konsoli.

Znak **\b** usuwa poprzedzający go znak. Dlatego w konsoli wypisana została tylko jedna kropka.

```
char znak1 = 'a';
char znak2 = '@';
char znak3 = '2';
char znak4 = '\u00A9';
char znak5 = '©';
System.out.println(znak1 + " " + znak2 + " " + znak3 + "
" + znak4 + " " + znak5);
```

Listing 3.29 Przykładowe użycie typu char

Jak wcześniej wspomniano, chary zapisujemy w apostrofach, a długość przechowywanych znaków nie może być większa niż jeden. W zmiennej o nazwie **znak4** został użyty jeden ze znaków **Unicode**.

Istnieje wiele znaków **Unicode**. Wszystkie umieszczone są w specjalnych tabelach, które bez większych problemów można znaleźć w Internecie. Po uruchomieniu programu widzimy, że **znak4** zwróci nam taką samą wartość co **znak5**.

Zmienne typu char możemy tworzyć również, za pomocą **konstruktora Character**.

W Javie istnieje deklaracja zmiennej, której przechowywana wartość nie ulegnie zmianie przez cały czas działania programu. Inaczej możemy powiedzieć, że jest to zmienna, której wartość nigdy się nie zmieni. Nazywa się ona **stała**. Stosujemy ją, kiedy jesteśmy pewni, że nasza zmienna będzie przechowywała wartość ostateczną. Taką zmienną deklaruje się w identyczny sposób, jak zwykłą, ale przed nazwą typu **(np. int)** dodajemy słowo **final**. Po polsku nazywamy ją finałową zmienną, zmienną stałą lub ostateczną. Poniżej został przedstawiony przykład jej deklaracji:

```
final float pi = 3.14f;
System.out.println(pi);
```

Listing 3.30 Deklaracja finałowej zmiennej

Co się stanie gdy zdecydujemy się zmienić wartość zmiennej stałej?

Na przykład:

```
final float pi = 3.14f;
pi = 3.5f;
System.out.println(pi);
```

Listing 3.31 Próba zmiany wartości finałowej zmiennej

Już przed samą kompilacją programu zauważymy, że nasze pi jest podkreślone na czerwono. Przy uruchomieniu programu, w konsoli pojawi nam się komunikat o treści:

```
java: cannot assign a value to final variable pi
```

Rysunek 3.13 Komunikat błędu w konsoli przy próbie zmiany wartości finałowej zmiennej

Każdy prymitywny typ danych ma przypisane określone wartości, które może przechowywać. A co w sytuacji, kiedy chcielibyśmy, aby **int** przechował

wartość zmiennoprzecinkową typu **float** lub **double?** Takiej zmiany możemy dokonać za pomocą rzutowania typów lub inaczej konwersji. W Javie wyróżniamy dwa typy konwersji:

- konwersja rozszerzająca (automatyczna) – polega na konwertowaniu mniejszego typu danych na większy np. - **byte → short → char → int → long → float → double**;
- konwersja zawężająca (manualna) – polega na konwertowaniu większego typu danych na mniejszy np. - **double → float → long → int → char → short → byte**;

Konwersja rozszerzająca wykonuje się automatycznie w momencie, w którym podamy mniejszy typ danych jako wartość dla większego typu.

```
int zmiennaInt = 6;
float zmiennaFloat = zmiennaInt;
System.out.println(zmiennaInt);
System.out.println(zmiennaFloat);
```

Listing 3.32 Przykład konwersji rozszerzającej

Na początku zadeklarowaliśmy zmienną typu **int**, przechowującą liczbę **6**. Następnie w drugim wierszu przypisaliśmy do zmiennej typu **float** wartość zmiennej typu **int**. Konwersja dokonała się automatycznie.

Konwersja zawężająca niestety musi być wykonana ręcznie poprzez wstawienie nawiasu z typem zmiennej przed jej nazwą. Wygląda to tak:

```
float zmiennaFloat = 3.64f;
int zmiennaInt = (int) zmiennaFloat;
System.out.println(zmiennaFloat);
System.out.println(zmiennaInt);
```

Listing 3.33 Przykład konwersji zawężającej

Tutaj z kolei zaczęliśmy od deklaracji zmiennej typu **float** (bo konwertujemy z większego na mniejszy typ danych). Następnie zmiennej typu **int** przypisaliśmy wartość **int** z zmiennej typu **float**. Po skompilowaniu programu zobaczysz poniższy wynik:

```
3.64
3

Process finished with exit code 0
```

Rysunek 3.14 Wynik konsolowy konwersji zawężającej

Jako, iż **int** nie przechowuje wartości zmiennoprzecinkowych, z zmiennej typu **float** o wartości **3.64** zostaje nam część całkowita o wartości **3**.

3.3.1 Sprawdź się!

1) Jakiego typu danych użyjesz do przechowywania liczb całkowitych w zakresie od -2^31 do 2^31 - 1?

a) int
b) short
c) long

2) Który z poniższych typów danych służy do przechowywania wartości logicznych (true/false)?

a) boolean
b) char
c) double

3) Który z poniższych typów danych służy do przechowywania pojedynczych znaków?

a) int
b) char
c) String

4) Który z poniższych typów danych jest używany do przechowywania liczb zmiennoprzecinkowych o pojedynczej precyzji?

a) double
b) float
c) long

5) Która z poniższych konwersji jest automatyczna w Javie?

a) Zmienna typu double do typu int
b) Zmienna typu int do typu double
c) Zmienna typu String do typu boolean

6) Która z poniższych nazw zmiennych nie jest dozwoloną nazwą w języku Java?

a) "my-variable"

b) "2abc"

c) "_abc"

d) "abc_xyz"

Zadanie 3.3.1

Napisz program w który będzie konwertował wartość liczbową podaną jako String na typ danych zmiennoprzecinkowy **float**. Następnie program ma wyświetlać obie wartości oraz dokładność konwersji.

3.4 Podstawowe operacje arytmetyczne i logiczne

Po poznaniu zmiennych oraz typów danych, przyszedł czas, aby zgłębić kolejny obszar, który jest kluczowy dla tworzenia bardziej zaawansowanych programów. W niniejszym rozdziale skoncentrujemy się na operacjach arytmetycznych i logicznych, które stanowią podstawę dla wielu aspektów programowania. Na początku zajmiemy się operacjami arytmetycznymi, zwanymi także matematycznymi. Operacje arytmetyczne stanowią podstawę każdego języka programowanie, bowiem dzięki nim jesteśmy w stanie manipulować danymi liczbowymi. W języku Java mamy dostęp do standardowych operacji, które pozwalają nam wykonywać podstawowe obliczenia matematyczne. Są nimi:

Operator	Nazwa	Opis	Przykład
+	dodawanie	dodaje do siebie dwie wartości	a + b
-	odejmowanie	odejmuje jedną wartość od drugiej	a - b
*	mnożenie	mnoży dwie wartości	a * b
/	dzielenie	dzieli jedną wartość przez drugą	a / b
%	modulo	zwraca resztę z dzielenia	a % b

Tabela 3.4 Podstawowe operacje arytmetyczne i logiczne

Zastosujemy teraz powyższe operacje w praktyce. Utworzymy nową **klasę** o nazwie „**PodstawoweOperacje**”. W metodzie „**main**” zapiszmy:

```
int wynikDodawania = 3 + 7;
System.out.println(wynikDodawania);
```

Listing 3.34 Zmienna przyjmująca jako wynik sumę dwóch wartości, a następnie wypisanie jej wyniku

Dodawanie w Javie jest intuicyjne i działa zarówno na liczbach całkowitych, jak i zmiennoprzecinkowych. Wartości **3** i **7** są dodawane, a wynik przypisywany do zmiennej „**wynikDodawania**". Następnie wypisujemy w konsoli jej wartość. Powinna pojawić się liczba **10**.

Zapis ten możemy również nieco uprościć:

```
System.out.println(3 + 7);
```

Listing 3.35 Uproszczony zapis poprzedniego kodu

Rezultat będzie ten sam, jednak wyżej wynik dodawania został przypisany do zmiennej, a tutaj po prostu został wyświetlony.

Teraz odejmiemy od siebie dwie liczby. W tej samej klasie zapiszemy:

```
double wynikOdejmowania = 8.40d - 5.20d;
System.out.println(wynikOdejmowania);
```

Listing 3.36 Zmienna przyjmująca jako wynik różnicę dwóch wartości, a następnie wypisanie jej wyniku

Operacja odejmowanie działa na tej samej zasadzie co operacja dodawania. Powyższy przykład ilustruje prostą operację odejmowania dwóch liczb zmiennoprzecinkowych typu **double**.

Warto zauważyć, że na końcu liczby np. **8.40d**, używamy literału „**d**", aby określić, że są to liczby zmiennoprzecinkowe typu **double**. W rezultacie zmienna „**wynikOdejmowania**" przechowuje wynik odejmowania **8.40** od **5.20**. który będzie wynosił **3.2**. Taki wynik powinien się pojawić w konsoli.

Kolejna operacja to mnożenie:

```
int wynikMnozenia = 4*5;
System.out.println(wynikMnozenia);
```

Listing 3.37 Zmienna przyjmująca jako wynik iloczyn dwóch wartości, a następnie wypisanie jej wyniku

Operacja ta, liczy iloczyn dwóch liczb, w tym przypadku **4** i **5**. Wynik będzie równy **20**.

Ostatnia operacja to dzielenie:

```
double wynikDzielenia = 6d/4d;
System.out.println(wynikDzielenia);
```

Listing 3.38 Zmienna przyjmująca jako wynik iloraz dwóch wartości, a następnie wypisanie jej wyniku

Dzielenie w Javie może prowadzić do wartości zmiennoprzecinkowych. Warto używać typu double, aby przechować wynik dzielenia dwóch liczb. W tym przypadku wynik dzielenia będzie równy **1.5**.

Próba podzielenia liczby przez **0** spowoduje błąd programu. W przypadku typu double, w konsoli pojawi się napis „**Infinity**", czyli nieskończoność. Jeśli jednak podzielimy liczby typu int przez **0**, wówczas w konsoli ujrzymy ten błąd:

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at PodstawoweOperacje.main(PodstawoweOperacje.java:15)
```

Rysunek 3.15 Błąd w konsoli przy próbie podzielenia liczby przez zero

W Javie występuje również operator **modulo**, który w wyniku działania wypisuje resztę z dzielenia.

```
double wynikDzieleniaModulo = 7d%3d;
System.out.println(wynikDzieleniaModulo);
```

Listing 3.39 Zmienna przyjmująca jako wynik resztę z dzielenia dwóch wartości, a następnie wypisanie jej wyniku

Wartość zmiennej **wynikDzieleniaModulo** będzie wynosić **1**, bo w liczbie **7** zmieścimy liczbę **3** tylko dwa razy i pozostaje nam niepodzielna **1**.

Te podstawowe operacje arytmetyczne są niezbędne do wykonywania wszelkiego rodzaju obliczeń matematycznych w programowaniu. W praktyce są one często używane w algorytmach, równaniach matematycznych oraz manipulacjach danymi liczbowymi w ogólności. W poprzednich rozdziałach wspomnieliśmy chociażby o inkrementacji np. **i++**.

Jest to przykład operatorów jednoargumentowych, które działają na jednym argumencie, czyli jednej zmiennej lub jednym wyrażeniu. W języku java istnieje kilka operatorów jednoargumentowych. Są to np. operatory inkrementacji i dekrementacji. Odgrywają w programowaniu bardzo ważną rolę. Pierwszy operator służy do zwiększania wartości zmiennej, z kolei drugi – do zmniejszania.

```
int liczba = 6;
liczba++;
System.out.println(liczba);
```

Listing 3.40 Zwiększenie wartości zmiennej przy użyciu operatora inkrementacji

W powyższym przykładzie użyliśmy **inkrementacji** wartości zmiennej **liczba**. Po tej operacji, wartość zmiennej **liczba** będzie równa **7**.

```
int liczba = 6;
liczba--;
System.out.println(liczba);
```

Listing 3.41 Zmniejszenie wartości zmiennej przy użyciu operatora dekrementacji

Teraz z kolei zmniejszyliśmy wartość zmiennej o **jeden** dzięki użyciu **dekrementacji**. Te operatory są szczególnie przydatne w przypadku, gdy chcemy szybko zwiększyć lub zmniejszyć wartość zmiennej o stałą wartość (w tym przypadku o **jeden**). W praktyce są one często używane w **pętlach** i innych sytuacjach, gdzie konieczne jest powtarzanie operacji na zmiennych liczbowych. Oprócz tych dwóch, istnieje też operator **ujemny**. Jego działanie nie jest skomplikowane. Można go użyć, aby zmienić wartość dodatnią na ujemną. Na przykład:

```
int wynik = 20;
wynik = -wynik;
System.out.println(wynik);
```

Listing 3.42 Zmiana wartości zmiennej na przeciwną wartość, w tym przypadku ujemną

Zadeklarowaliśmy zmienną wynik o początkowej wartości równej **20**. W linii niżej przypisaliśmy tej zmiennej wartość zmiennej wynik, ale z **minusem**, który jednocześnie zmienił tę wartość na **ujemną**. W wyniku tego, w konsoli pojawi się liczba **-20**. Te operatory są ważne w różnych kontekstach programowania, a ich zastosowanie zależy od potrzeb danego zadania. Operatory inkrementacji i dekrementacji są przydatne w kontekście manipulacji zmiennymi liczbowymi. Operatory **jednoargumentowe** pozwalają na bardziej złożone operacje na pojedynczych zmiennych.

Używaliśmy już wielokrotnie przypisywania wartości do zmiennej, więc przyjrzymy się teraz nieco bardziej operatorom **przypisania**. Są one znaczącym elementem programowania w języku Java, umożliwiającym przypisanie wartości do zmiennych. Ogólny format operatora przypisania to:

```
int wartosc = 30;
int zmienna = wartosc;
```

Listing 3.43 Przypisanie zmiennej jako wartości dla zmiennej

Wartość po prawej stronie musi być zmienną tego samego typu co zmienna po lewej stronie, co zapewnia poprawne przypisanie danych. Oczywiście w tym przykładzie moglibyśmy po prostu przypisać zmiennej o nazwie zmienna wartość **30**, bez tworzenia dodatkowej zmiennej.

Operator „=" jest podstawowym i najprostszym operatorem przypisania. Są jednak jeszcze mieszane operatory **przypisania**. Są to na przykład znaki „+", „-", czy „/", w połączeniu ze znakiem „=". Inaczej są nazywane operatorami przypisania z operacją arytmetyczną.

```
int x = 5;
x += 3; // x = x + 3
```

Listing 3.44 Zwiększenie wartości zmiennej za pomocą operatora przypisania z operacją arytmetyczną

Zmienna **x** przechowuje początkowo wartość **5**. Następnie używamy operatora przypisania z operacją dodawania liczby **3**, co oznacza, że do wartości **5** dodajemy wartość **3**. Te operatory pozwalają skrócić zapis, zwłaszcza gdy chcemy wykonać operację na zmiennej i przypisać jej wynik z powrotem do tej samej liczbie. W komentarzu znajduje się alternatywny zapis tej linii.

```
int x = 5;
x *= 2; // x = x * 2
System.out.println(x);
```

Listing 3.45 Zwiększenie wartości zmiennej poprzez pomnożenie jej wartości i wypisanie wyniku

W powyższym przykładzie dodawanie zastąpiliśmy mnożeniem. Wartość zmiennej x mnożymy razy dwa, co da nam w konsoli wynik **10**.

W Javie jest możliwość porównywania danych, przykładowo wartości zmiennych. Możemy sprawdzić, która zmienna jest większa spośród dwóch. Do tego używamy operacji **relacyjnych** (porównania). Są wykorzystywane przy porównaniu dwóch wartości i zwracają wynik logiczny **true/false** na podstawie spełnienia określonych warunków. Poniżej w tabeli zostały przedstawione przykładowe operacje relacyjne:

Operator	Opis	Przykład
==	Równy	a == b
!=	Nierówny	a != b
>	większy od	a > b
<	mniejszy od	a < b
>=	większy bądź równy od	a >= b
<=	mniejszy bądź równy od	a <= b

Tabela 3.5 Przykładowe operacje relacyjne

Poniżej znajduje się przykład kodu zawierający operator równości:

```
int liczba1 = 5;
int liczba2 = 5;
boolean czyRowne = liczba1 == liczba2;
```

Listing 3.46 Sprawdzanie czy wartości zmiennych są równe przy użyciu operatora równości

W tym przykładzie zmienna **czyRowne** typu boolean będzie przechowywać wartość **true** lub **false**, w zależności od tego, czy **liczba1** jest równa **liczba2**. W tym przypadku będzie równa **true**, ponieważ obie zmienne przechowują wartość **5**. Teraz użyjemy kilka wybranych z pozostałych operacji porównania. Oto przykładowy kod:

```
int liczba3 = 5;
int liczba4 = 7;
boolean czyNierowne = liczba3 != liczba4;
boolean czyWieksze = liczba3 > liczba4;
boolean czyMniejszeLubRowne = liczba3 <= liczba4;
```

Listing 3.47 Przykład użycia wybranych operacji porównania

W kodzie zdefiniowane są dwie zmienne całkowite **liczba3** i **liczba4**. Następnie użyte są operatory relacyjne do porównań. **Boolean czyNierowne** sprawdza czy **liczba3** jest różna od **liczba4**. W tym przypadku operacja jest prawdziwa **true**, ponieważ **5** nie jest równe **7**. W kolejnej linii znajduje się zmienna **czyWieksze**, która z kolei sprawdza, czy **liczba3** jest większa od **liczba4**. Jak wiemy - nie jest. Stąd też, zmienna **czyWieksze** będzie przyjmowała wartość **false**. W ostatniej linii użyta jest zmienna **czyMniejszeLubRowne**. Ta będzie sprawdzała, czy **liczba3** jest mniejsza, bądź równa **liczba4**. W tej sytuacji **5** jest mniejsze, więc wynik będzie true.

Operacje relacyjne są kluczowe w warunkach logicznych i pętlach, pozwalając programowi na podejmowanie decyzji na podstawie relacji między różnymi wartościami.

Java posiada także operacje logiczne. Są głównym elementem programowania, umożliwiającym programowi podejmowanie decyzji na podstawie określonych warunków. Kluczowym aspektem tego mechanizmu jest zdolność do kontrolowania przepływu programu w zależności od spełnienia lub niespełnienia pewnych warunków logicznych. W tabeli przedstawione zostały najważniejsze informacje na temat tych operacji.

Operator	Opis	Przykład
&&	zwraca true gdy oba warunki są spełnione	a && b
||	zwraca true gdy jeden z warunków jest spełniony	a || b
!	zwraca odwróconą wartość (np. false, gdy wynik jest true)	a ! b

Tabela 3.6 Przykładowe operacje logiczne

Jako pierwszą omówimy operację logiczną **AND („&&”)**. Operacja **AND** zwraca **true** tylko wtedy, gdy oba warunki są spełnione. Poniżej przykład użycia tego operatora:

```
boolean warunek1 = true;
boolean warunek2 = false;
boolean wynikWar = warunek1 && warunek2;
```

Listing 3.48 Przykład zastosowania operatora logicznego „AND”

Wartość zmiennej **wynikWar** będzie wynosiła false, ponieważ nie został spełniony warunek. Dobrym przykładem użycia zmiennej **AND** oraz **OR** będzie sprawdzenie czy liczba jest parzysta lub nieparzysta i większa lub mniejsza od podanej. Wykorzystamy w nim operacje relacyjne:

```
int numer = 15;
boolean warunekAND = (liczba % 2  == 0 ) && (liczba >
10);
boolean warunekOR = (liczba % 2 != 0) || (liczba <= 5);
System.out.println("Czy liczba jest parzysta i większa
od 10? " + warunekAND);
System.out.println("Czy liczba jest nieparzysta lub
mniejsza równa 5? " + warunekOR);
```

Listing 3.49 Przykład zastosowania operatorów logicznych „AND” oraz „OR”, a następnie wypisanie wyniku

W tym przykładzie mamy zmienną numer o wartości **15**. Zmienna **warunekAND** sprawdza, czy ta liczba jest zarówno **parzysta**, jak i większa od **10**. W tym przypadku wynikiem będzie **false**, ponieważ **15** nie jest **parzystą** liczbą. Druga zmienna **boolean warunekOR** będzie sprawdzać, czy liczba jest albo nieparzysta, albo mniejsza bądź równa **5**. W tej sytuacji wynikiem będzie true, bo liczba **15** jest większa niż **5**, czyli spełnia warunek.

W języku Java występuje jeszcze operator **NOT** oznaczany wykrzyknikiem „**!**”. Wykonuje negację wartości logicznej, czyli zamienia wartość **true** na false i odwrotnie. Poniżej przykładowe zastosowanie tego operatora:

```
boolean prawda = true;
boolean falsz = !prawda;
System.out.println(falsz);
```

Listing 3.50 Wykorzystanie operatora negacji i wydrukowanie tej zmiennej

Wykorzystaliśmy zmienne logiczne typu **boolean**. Wartość **true** reprezentuje zmienną **prawda**, natomiast **falsz** przyjmuje wartość negującą zmienną prawda. W efekcie zmienna **falsz** będzie równa **false**.

Operator **NOT** często używany jest w wyrażeniach warunkowych, gdzie chcemy odwrócić wartość logiczną. Operacje logiczne są przydatne w strukturach warunkowych, pętlach, oraz w kontrolowaniu przepływu programu. W praktyce używane są do podejmowania decyzji w zależności od różnych warunków.

W tym rozdziale poznaliśmy operacje na zmiennych i operatorach oraz zgłębiliśmy różnorodne aspekty manipulacji danymi i sterowania przepływem programu. Operacje arytmetyczne dostarczają podstawowych narzędzi do wykonywania obliczeń numerycznych. Zastosowaliśmy je w kontekście różnych problemów, co pozwoliło nam lepiej zrozumieć ich praktyczne zastosowanie.

Przeanalizowaliśmy operatory jednoargumentowe i zauważyliśmy, jak można za ich pomocą manipulować wartościami zmiennych, wprowadzając zmiany w ich wartości w zwięzły sposób. Wiemy też, że operacje przypisania pozwalają na aktualizację wartości zmiennych w trakcie działania programu, co może przyczynić się do bardziej czytelnego kodu. Skoncentrowaliśmy się również na operacjach relacyjnych, które umożliwiają porównywanie wartości. Warunki logiczne tworzone za pomocą tych operatorów decydują o przebiegu programu.

Na koniec wykorzystaliśmy operacje logiczne do tworzenia warunków logicznych, które wpływają na działanie programu w zależności od spełnienia określonych kryteriów. Podsumowując, zrozumieliśmy, jak te operacje integrują

się w składnie języka Java, umożliwiając programistom wykonywanie różnorodnych zadań, od prostych obliczeń po bardziej zaawansowane decyzje sterujące.

3.4.1 Sprawdź się!

1) Który z poniższych operatorów jest używany do logicznego "lub" w języku Java?

a) &&

b) ||

c) !

d) ==

2) Który z poniższych operatorów jest używany do inkrementacji w języku Java?

a) +

b) -

c) ++

d) --

3) Który z poniższych operatorów jest używany do porównania obiektów pod kątem referencji w języku Java?

a) ==

b) ===

c) equals()

d) compare()

Zadanie 3.4.1

Napisz program w Javie, który będzie obliczał podstawowe operacje arytmetyczne na dwóch liczbach i wyświetlał wyniki. Użyj następujących danych wejściowych:

- **liczba1: 25**;
- **liczba2: 7**;

Program powinien obliczyć i wyświetlić następujące operacje: **Sumę, Różnicę, Iloczyn, Iloraz, Resztę** z dzielenia (modulo).

3.5 Instrukcje warunkowe

W poprzednich częściach naszej książki omówiliśmy już wiele aspektów języka Java, zaczynając od podstaw, przechodząc przez typy danych, aż po operacje matematyczne i logiczne.

Nadszedł teraz czas, aby skierować naszą uwagę na instrukcje warunkowe, które są bardzo istotne w programowaniu. Polegają one na decyzyjności i pozwalają programowi adaptować swoje działanie w zależności od zmiennej sytuacji.

W tym rozdziale dowiemy się, jak używać instrukcji warunkowych do sprawdzania warunków logicznych, wybierania między różnymi ścieżkami wykonania programu, oraz jak te elementy mogą uczynić nasze programy bardziej inteligentnymi i responsywnymi.

Na ten moment wiemy już, że Java wspiera użycie operatorów relacyjnych, dzięki którym możemy sprawdzać, na przykład czy dana zmienna jest większa lub równa wartości innej zmiennej. Możemy ich używać do wykonywania różnych działań w przypadku spełnienia różnych warunków. W praktyce działają one często w parze z instrukcjami warunkowymi.

3.5.1 Instrukcje if i else

W Javie głównym mechanizmem realizującym instrukcje warunkowe są konstrukcje **if**, **else if** oraz **else**. Zaczniemy od omówienia tej wymienionej jako pierwsza, najczęściej używanej.

Instrukcja warunkowa **if** pozwala na wykonanie pewnych instrukcji tylko wtedy, gdy określony warunek jest spełniony.

Utwórzmy teraz nową klasę o nazwie **InstrukcjeWarunkowe**, następnie zapiszmy w niej metodę **main**.

```
int liczba = 10;

if(liczba > 0){
    System.out.println("Liczba jest dodatnia");
}
```

Listing 3.51 Przykładowa instrukcja warunkowa „if”

Zadeklarowaliśmy zmienną **liczba**, która przechowuje **wartość 10**.

Tę liczbę będziemy analizować w kontekście instrukcji warunkowej.

W kolejnej linijce kodu użyliśmy instrukcji warunkowej **if**, w której znajduje się warunek **liczba > 0**.

Instrukcja **System.out.println("Liczba jest dodatnia")** będzie wykonana tylko wtedy, gdy warunek ten będzie prawdziwy czyli gdy **liczba** jest większa od zera.

W przypadku, gdyby wartość zmiennej **liczba** jest mniejsza od **0**, ta instrukcja nie zostałaby wykonana, a program kontynuowałby swoje działanie.

A co w sytuacji, gdybyśmy chcieli, by nasz program poinformował nas, że liczba jest ujemna, jeśli nie jest większa od zera?

W takim przypadku, z pomocą przychodzi nam **else**. Ta instrukcja jest używana w połączeniu z instrukcją **if** i pozwala zdefiniować blok kodu, który zostanie wykonany, gdy warunek instrukcji **if** nie jest prawdziwy. Poniżej przykład zastosowania instrukcji **else** w praktyce:

```
int liczba = -5;

if(liczba > 0){
    System.out.println("Liczba jest dodatnia");
}else{
    System.out.println("Liczba nie jest dodatnia");
}
```

Listing 3.52 Instrukcja warunkowa z „else", wyświetlająca określony komunikat

Tym razem zmieniliśmy wartość zmiennej na **-5**. Najpierw program sprawdza, czy **liczba** jest większa od **zera**. W tym przypadku warunek nie jest spełniony, więc program przejdzie do bloku **else**. Kod zapisany wewnątrz **else** zostanie wykonany. W wyniku tego w konsoli ujrzymy komunikat **Liczba nie jest dodatnia**.

W skrócie, jeśli warunek **if** jest spełniony, wykonuje się kod wewnątrz **if**, jeśli nie, to kod wewnątrz **else**. W języku Java znajdziemy również zagnieżdżone instrukcje **else if**.

W kontekście instrukcji warunkowej oznacza to wprowadzenie kolejnego warunku do rozważenia, jeżeli pierwszy warunek w instrukcji **if** nie został spełniony.

Takie rozwiązanie to sposób na uwzględnienie różnych przypadków w programie. Przykładowe użycie **else if**:

```
int liczba = 0;

if(liczba > 0){
    System.out.println("Liczba jest dodatnia");
}else if(liczba < 0){
    System.out.println("Liczba nie jest dodatnia");
}else{
    System.out.println("Liczba jest zerem");
}
```

Listing 3.53 Instrukcja warunkowa z „else if", wyświetlająca określony komunikat

Zmienna **liczba** tym razem przechowuje wartość **0**. Pierwsza instrukcja warunkowa sprawdza, czy liczba jest większa od zera. Jeśli tak, to wypisywany jest podany komunikat, lecz w bieżącym programie nie jest on spełniony. Następnie używamy konstrukcji **else if**. Jeśli warunek związany z pierwszym **if** nie jest spełniony, program przechodzi do kolejnej instrukcji warunkowej. Tasprawdza, czy **liczba** jest mniejsza od **zera**. Jeśli warunek jest spełniony, to program również wyświetli nam odpowiedni komunikat.

Przy wprowadzonych danych warunek nie będzie spełniony. W związku z tym program przechodzi do bloku kodu wewnątrz instrukcji **else**. Właśnie ten komunikat pokaże nam się w konsoli, ponieważ nasza **liczba** jest zerem. Instrukcje **else** oraz **else if** są opcjonalne i mogą występować po instrukcji **if**. Sprawiają, że kod staje się bardziej elastyczny i dostosowuje się do różnych scenariuszy.

Instrukcje warunkowe w Javie mogą być zagnieżdżane, co pozwala na jeszcze bardziej złożone scenariusze decyzyjne. Poniżej przykład:

```
int liczba = 0;

if(liczba > 0){
    System.out.println("Liczba jest dodatnia");
}else if(liczba < 0){
    System.out.println("Liczba nie jest dodatnia");
}else{
    System.out.println("Liczba jest zerem");
}
```

Listing 3.54 Zagnieżdżona instrukcja warunkowa „if else", wyświetlająca określone komunikaty

Ten kod analizuje trzy zmienne całkowite **a**, **b**, **c**. My skupimy się na zmiennej **b**. Przyjmijmy założenie, że program wie o tym, że wartość **c** jest większą od **a**. Dzięki temu, możemy od razu przyjąć, którą w hierarchii podanych liczb jest **b**. Najpierw instrukcja warunkowa **if** sprawdza, czy wartość zmiennej **b** jest

większa niż wartość zmiennej **a**. Warunek będzie spełniony, więc program przejdzie do bloku kodu wewnątrz tej instrukcji. W tym bloku jest jednak kolejna instrukcja warunkowa, sprawdzająca czy liczba **b** jest większa od **c**. To przykład zagnieżdżenia instrukcji warunkowej. Jeśli warunek **b > c** nie zostanie spełniony, wówczas program przechodzi do bloku kodu wewnątrz **else**. W naszym przypadku tak się stanie, a więc program zwróci nam w konsoli komunikat **Liczba b jest drugą największa liczbą spośród trzech zmiennych**. Na koniec mamy **else**, który wykona się, gdy liczba **b** okaże się mniejsza od **a** na samym początku.

Podsumowując, kod ten jest przykładem zagnieżdżonych instrukcji warunkowych, które pozwalają określić relacje między trzema zmiennymi i wypisać odpowiednie komunikaty w zależności od ich wartości. Istnieje krótszy sposób zapisu instrukcji warunkowej. Możemy tego dokonać korzystając z wyrażeń trójargumentowych. W przeciwieństwie do pełnych instrukcji **if-else**, wyrażenia trójargumentowe pozwalają skrócić kod, szczególnie gdy chcemy przypisać wartość do zmiennej w zależności od spełnienia warunku. Ich ogólna struktura wygląda tak:

Rysunek 3.16 Struktura wyrażenia trójargumentowego

Poniżej zastosowanie tego wyrażenia w praktyce:

```
int numer = 15;
String wynik = (numer % 2 == 0) ? "parzysta" :
"nieparzysta";
System.out.println("Liczba jest " + wynik);
```

Listing 3.55 Przykład użycia wyrażenia trójargumentowego

Zmienna **numer** przechowuje wartość **15**. W następnej linii znajduje się zmienna **wynik** typu **string**, która przyjmuje wartość wyrażenia trójargumentowego. Wyrażenie te sprawdza, czy reszta z dzielenia **numer** przez **2** jest równa **0**. Jeśli warunek jest spełniony **czyli liczba jest parzysta**, to do zmiennej **wynik** przypisywany jest tekst **parzysta**, w przeciwnym razie **nieparzysta**. Na koniec program wypisze nam komunikat, w tym przypadku będzie to Liczba jest **nieparzysta**. Poniżej zaprezentowano, jak ten przykład ma się do ogólnej struktury:

- **wyrażenie:** „(numer % 2 == 0)”;
- **wartość_gdy_prawda:** „parzysta”;
- **wartość_gdy_fałsz:** „nieparzysta”;

Ten kod przedstawia sposób wykorzystania wyrażenia trójargumentowego do określenia parzystości lub nieparzystości liczby i wypisania odpowiedniego komunikatu w konsoli. Zastosowanie tego wyrażenia jest dobrym sposobem na skrócenie kodu w przypadku prostych instrukcji warunkowych. Gdybyśmy jednak chcieli zapisać ten kod w tradycyjny sposób, tak jak na początku tego rozdziału, to możemy to zrobić w ten sposób:

```
int numer = 15;
String wynik;

if(numer % 2 == 0){
    wynik = "parzysta";
}else{
    wynik = "nieparzysta";
}

System.out.println("Liczba jest " + wynik);
```

Listing 3.56 Poprzedni kod bez zastosowania wyrażenia trójargumentowego

Jak widać, linii kodu jest trochę więcej, ale działanie programu pozostaje bez zmian. Wyrażenia trójargumentowe posiadają wiele korzyści. Są nimi krótszy zapis, dzięki któremu kod jest bardziej zwięzły, szczególnie w prostych przypadkach. Wyrażenia te są również idealne do użycia w jednolinijkowych sytuacjach, gdzie pełna konstrukcja **if-else** jest niepotrzebna.

Wyrażenia trójargumentowe posiadają niestety swoje wady. Przede wszystkim, są ograniczone do dwóch możliwych wyników, co sprawia, że nie możemy ich użyć do bardziej rozbudowanych problemów. Oprócz tego, w przypadku bardziej skomplikowanych warunków mogą nie być do końca czytelne. Wówczas lepiej sprawdzi się konstrukcja **if-else**.

3.5.2 Switch-case

W poprzednim rozdziale przedstawione zostały instrukcje warunkowe, które umożliwiają programowi podejmowanie różnych działań w zależności od spełnienia określonych warunków. Teraz skupimy się na bardziej zaawansowanych konstrukcjach sterujących, czyli instrukcji **switch**.

Pozwalają one na bardziej czytelne zarządzanie wieloma przypadkami traktowanymi równomiernie.

Instrukcja **switch** w języku Java służy do wybierania jednej z wielu możliwych ścieżek wykonania zależnie od wartości wyrażenia. Jest alternatywą dla długich ciągów instrukcji **if-else**. Przejdziemy teraz do zaprezentowania instrukcji **switch** w praktyce. Zatem utwórzmy nową klasę o nazwie **SwitchCase** i dodajmy w niej metodę **main**. Następnie zapiszmy podany kod:

```
int dzienTygodnia = 3;
switch(dzienTygodnia){
    case 1:
        System.out.println("Poniedziałek");
        break;
    case 2:
        System.out.println("Wtorek");
        break;
    case 3:
        System.out.println("Środa");
        break;
    case 4:
        System.out.println("Czwartek");
        break;
    case 5:
        System.out.println("Piątek");
        break;
    case 6:
        System.out.println("Sobota");
        break;
    case 7:
        System.out.println("Niedziela");
        break;
    default:
        System.out.println("Weekend");
        break;
}
```

Listing 3.57 Przykład instrukcji „switch-case”

Program z dniami tygodnia jest jednym z lepszych przykładów na przedstawienie instrukcji **switch**.

Na początku stworzyliśmy zmienną **dzienTygodnia** typu **int**. Jest zainicjowana wartością **3**, co odpowiada środzie.

Instrukcja **switch** określa wartość, która ma zostać porównana z różnymi przypadkami czyli **case'ami**. Pomiędzy klamrami zapisujemy wartości **case**. Są to możliwe do wystąpienia przypadki. Określają one wartość, z która ma zostać porównane wyrażenie. Jeśli jest wyrażenie jest zgodne z przypadkiem to wykonuje się kod dla danego **case'a**.

W przykładzie każdy **case** będzie wypisywał w konsoli dzień tygodnia odpowiadający numerowi wyrażenia.

Pojawiająca się instrukcja **break**, przerywa wykonywanie instrukcji **switch-case**. Ostatnią opcją konstrukcji jest **default**, czyli opcjonalny blok, który wykonuje się, gdy żaden case nie pasuje do wartości wyrażenia. Jest to odpowiednik **else** w instrukcjach warunkowych.

W naszym programie, jeśli zmienna **dzienTygodnia** przyjmowałaby wartość inną niż od **1** do **7**, to w konsoli pojawiłby się komunikat „**Weekend**".

Podsumowując, program będzie szukał **case'a** odpowiadającego wartości zmiennej **dzienTygodnia** i wykona zawartość bloku kodu odpowiedniego **case'a**. Jeśli napotka w nim słowo **break**, to kończy działanie. Używanie **break** jest zazwyczaj wymagane, aby przerwać wykonanie **switch-case** po spełnieniu warunku. Jego brak może prowadzić, do wykonywania kodu dla kolejnych **case'ów**, nawet jeśli warunek dla nich nie jest spełniony. Warto więc o nim pamiętać.

Dla **dzienTygodnia** równego **3**, (tak jak jest w naszym programie) program wyświetli w konsoli napis **Środa**. Pozostałe przypadki po znalezieniu pasującego **case'a** nie będą sprawdzane, dzięki czemu ścieżki wykonania programu są efektywnie zarządzane.

W instrukcjach **switch-case** można także skorzystać z alternatywnego zapisu z użyciem operatora strzałki. Przykład powyżej możemy przekształcić w podany sposób:

```
int dzienTygodnia = 3;
switch (dzienTygodnia){
    case 1 -> System.out.println("Poniedziałek");
    case 2 -> System.out.println("Wtorek");
    case 3 -> System.out.println("Środa");
    case 4 -> System.out.println("Czwartek");
    case 5 -> System.out.println("Piątek");
    case 6 -> System.out.println("Sobota");
    case 7 -> System.out.println("Niedziela");
    default -> System.out.println("Weekend");
}
```

Listing 3.58 Przykład instrukcji „switch-case" z operatorem strzałki

Operator strzałki → w języku Java jest skróconą formą zapisu dla prostych bloków kodu. Jak widać, przydaje się również w instrukcji **switch-case**. Warto zauważyć, że nie ma potrzeby używania **break**, ponieważ operator strzałki automatycznie kończy działanie danego **case'a** w danym przypadku.

Instrukcja **switch-case** jest przydatna, gdy mamy do czynienia z wieloma przypadkami równorzędnie traktowanymi np. dni tygodnia. W przypadku zmiennych lub bardziej skomplikowanych porównań, lepsze może być skorzystanie z instrukcji warunkowej **if-else**.

Tak o to poszerzyliśmy naszą wiedzę o instrukcjach. Zrozumieliśmy, jak skutecznie porównywać wartości wyrażeń i jak łatwo zarządzać wieloma scenariuszami w zależności od zmiennej wejściowej.

Poznaliśmy składnię instrukcji **switch**, czy zastosowanie operatora strzałki do bardziej zwięzłego zapisu.

Wiemy też, dlaczego **break** jest niezbędny w przypadku tej instrukcji, i jak unikać potencjalnych błędów z nim związanych.

Instrukcja ta jest zwięzła i czytelna, zwłaszcza gdy mamy do czynienia z wieloma opcjami równorzędnymi. Trzeba mieć jednak na uwadze, że nie zawsze **switch** jest najlepszym wyborem i czasem lepiej zastosować instrukcję **if-else**.

3.5.3 Sprawdź się!

1) Który operator logiczny łączy dwa warunki w instrukcji warunkowej w Javie, sprawdzając, czy oba są prawdziwe?

a) &&

b) ||

c) !

d) ==

2) Która z poniższych instrukcji sprawdza, czy dwie wartości są różne od siebie?

a) !=

b) ==

c) >

d) <

3) Która instrukcja warunkowa w Javie umożliwia wybór jednej z wielu opcji na podstawie wartości jednej zmiennej?

a) if

b) else

c) else if

d) switch

4) Ile razy w ramach jednego bloku instrukcji warunkowej może wystąpić instrukcja if, else oraz else if w języku Java?

a) Każda z nich może wystąpić dowolną ilość razy

b) if może wystąpić tylko raz, else może wystąpić tylko raz, else if może wystąpić dowolną ilość razy

c) if może wystąpić dowolną ilość razy, else może wystąpić tylko raz, else if może wystąpić dowolną ilość razy

d) if może wystąpić tylko raz, else może wystąpić tylko raz, else if może wystąpić tylko raz

Zadanie 3.5.1

Napisz program w języku Java, który będzie sprawdzał, czy podana liczba jest **parzysta** czy **nieparzysta**. Następnie program ma wyświetlić odpowiedni komunikat informujący o wyniku sprawdzenia.

Zadanie 3.5.2

Napisz program w języku Java, który na podstawie podanej oceny ucznia w formie **liczby** wyświetli odpowiednią **ocenę** w formie **słownej** według polskiego systemu oceniania. Użyj instrukcji **switch case**. Np. dla liczby **5** zwraca **bardzo dobry**.

3.6 Tablice, listy, mapy

3.6.1 Tablice

W poprzednich rozdziałach mieliśmy do czynienia z zmiennymi, których zadaniem było przechowywanie danych.

Wyobraźmy sobie sytuację, gdy w programie musimy przechować np. **100** wartości tego samego typu? Tworzenie **100** osobnych zmiennych byłoby bardzo czasochłonne, kłopotliwe i przede wszystkim niepraktyczne.

W tym miejscu z pomocą przychodzą nam **tablice**. Tablice to struktury, które gromadzą określoną ilość danych w uporządkowany sposób. Tablicom możemy nadać nazwę, oraz rozmiar.

W języku Java występuje kilka różnych rodzajów tablic. Zaczniemy od najprostszej – tablicy jednowymiarowej.

Tablice deklarujemy w następujący sposób:

```
int[] tablica = new int[10];
```

Listing 3.59 Deklaracja tablicy

Na początku podajemy typ **tablicy**, a obok **kwadratowe nawiasy**, które informują o tym, że w tym miejscu programu znajduje się tablica.

Następnie nadajemy tablicy **nazwę** i za pomocą **znaku równości** tworzymy nową pustą tablicę. Na samym końcu w kwadratowych nawiasach podajemy **rozmiar** tablicy. W tym przypadku wynosi on **10**.

Istnieje również drugi sposób tworzenia tablicy jednowymiarowej.

Prezentuje się on następująco:

```
int tablica[] = new int[10];
```

Listing 3.60 Inny sposób deklaracji tablicy

Różni się on od poprzedniego tylko **miejscem** położenia **kwadratowych nawiasów**. Kolejny sposób tworzy tablice wraz z **wartościami**. Wygląda on tak:

```
int[] tablica = new int[]{1,2,3,4};
```

Listing 3.61 Deklaracja tablicy z wartościami

Jak widzisz zmienia się trochę koncepcja tworzenia tablicy. Po słówku **new int[]** podane są elementy tablicy. Umieszczamy je w nawiasach klamrowych.

W przypadku tworzenia tablicy z wartościami, nie musimy już podawać jej **rozmiaru**.

Do elementów tablicy odwołujemy się za pomocą **indeksów**. Każda tablica zaczyna numerowanie indeksów od **0**. W tym przypadku liczba **1** będzie pod **indeksem 0**. Aby się do tej liczby odwołać będziemy musieli wypisać **tablica[0]**. Oznacza to, że pobieramy z tablicy indeks numer **0**, czyli liczbę **1**. Jeśli będziemy chcieli się odwołać do ostatniego indeksu tablicy to będziemy musieli zapisać **tablica[3]**. Tablica może być dowolnego typu, również **string**. Będzie ona wyglądała wtedy w ten sposób:

```
String[] tablica = new
String[]{"pierwszy","drugi","trzeci"};
```

Listing 3.62 Deklaracja tablicy typu String z wartościami

Zmienną tablicową można również zadeklarować tak jak **zwykłą zmienną** - bez jej natychmiastowego utworzenia. Dokonujemy wtedy takiego zapisu:

```
int[] tablica;
tablica = new int[]{1,2,3,4};
```

Listing 3.63 Deklaracja tablicy bez jej natychmiastowego utworzenia

Do wypisywania elementów zawartych w tablicy możemy oczywiście użyć poznanej już wcześniej funkcji **System.out.println()**. W nawiasach podajemy **nazwę** tablicy oraz jej **indeks**. Taki zapis:

```
String[] tablica = new
String[]{"pierwszy","drugi","trzeci","czwarty"};
System.out.println(tablica[0]);
System.out.println(tablica[2]);
```

Listing 3.64 Wypisywanie elementów tablicy za pomocą System.out.println

Zwróci nam w konsoli następujący zapis:

```
pierwszy
tzeci

Process finished with exit code 0
```

Rysunek 3.17 Wynik konsolowy wypisanych indeksów tablicy

Wiesz już więc, w jaki sposób wypisywać wybrane indeksy z tablicy. Co w przypadku gdy chciałbyś utworzyć tablicę o podanym rozmiarze, nie wiedząc jeszcze, jakie wartości do niej wpisać?

Załóżmy, że twoja tablica ma przechowywać **5** elementów, ale ty na ten moment chcesz wpisać tylko **3**, a w przyszłości uzupełnić resztę. Jest to możliwe i stosunkowo proste. Robimy to w ten sposób:

```
float[] tablica = new float[5];
tablica[4] = 3.14f;
System.out.println(tablica[4]);
```

Listing 3.65 Przypisanie wybranemu indeksowi z tabeli wybranej wartości, a następnie wypisanie go za pomocą System.out.println

W konsoli otrzymamy wówczas:

```
3.14

Process finished with exit code 0
```

Rysunek 3.18 Wynik konsolowy wartości indeksu z tabeli

Jeśli przekroczymy rozmiar tablicy to program zwyczajnie zwróci nam błąd. Gdy indeksowi nie przypiszemy **wartości** to zostanie ona uzupełniona automatycznie i będzie po prostu **wartością domyślną** danego typu. Przykładowo dla tablicy typu **float**, domyślną wartością będzie **0.0f**, a dla tablicy **bool** – będzie to **false**.

Przejdźmy teraz do **tablic wielowymiarowych**. Jak nazwa mówi – mają więcej niż jeden wymiar. Taka tablica może składać się na przykład z **3 wierszy** i **4 kolumn**. Deklaruje się je w podobny sposób do tablic jednowymiarowych, jednak w tych umieszczamy dwa nawiasy kwadratowe:

```
int[][] tablica = new int[3][4];
```

Listing 3.66 Deklaracja tablicy wielowymiarowej złożonej z 3 wierszy i 4 kolumn

Możemy sobie wyobrazić, że tablica wielowymiarowa to po prostu kwadrat o podanych rozmiarach. Tablica może mieć również **trzy wymiary**. Tworzymy ją dodając kolejne kwadratowe nawiasy.

W tej chwili zajmiemy się jednak tablicami dwuwymiarowymi. Tablice wielowymiarowe możemy zainicjować z wartościami tak samo, jak w tablice jednowymiarowe. Dokonujemy tego w ten sposób:

```
int[][] tablica = { {1,2} , {3,5} };
```

Listing 3.67 Deklaracja tablicy wielowymiarowej z wartościami

W tego typach tablicach dane zapisujemy w **klamrach** i oddzielamy dane **przecinkiem**. Jeśli chodzi o wypisywanie wartości z danych indeksów z tabeli, to robimy to w bardzo podobny sposób do tablicy jednowymiarowej:

```
int[][] tablica = { {1,2} , {3,5} };
System.out.println(tablica[0][0]);
System.out.println(tablica[1][1]);
```

Listing 3.68 Deklaracja tablicy wielowymiarowej wraz z wypisaniem konkretnych wartości

Podany zapis wyświetli nam następujące liczby:

```
1
5

Process finished with exit code 0
```

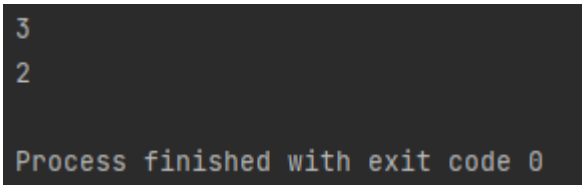
Rysunek 3.19 Wypisanie w konsoli wybranych indeksów tablicy wielowymiarowej

W pierwszym przypadku otrzymaliśmy liczbę **1**, ponieważ pod indeksem pierwszym **[0]** kryją się dwie liczby – **1** oraz **2**. Kolejnym indeksem jest również **[0]** co wskazuje na **jedynkę**. Analogicznie, z drugim wynikiem dzieje się to samo.

Java umożliwia nam sprawdzenie długości naszej tabeli za pomocą funkcji **length**. Ta metoda przyda ci się w pętlach, które będziemy omawiać w dalszym etapie tej książki. Długość tablicy wypisujemy w podany sposób:

```
float[] tablica = new float[3];
int[][] tablica_w = { {1,2} , {3,5} };
System.out.println(tablica.length);
System.out.println(tablica_w.length);
```

Listing 3.69 Sprawdzanie oraz wypisywanie w konsoli długości tablic



Rysunek 3.20 Wypisanie w konsoli długości tablic

W pierwszej tablicy długość możemy odczytać już przy samej inicjacji, natomiast druga nie ma określonego rozmiaru, więc ta metoda jest tutaj przydatna. Tablice bywają ograniczające pod niektórymi względami. Szczególnym ich minusem jest rozmiar, którego nie możemy zmieniać i musi być on ustalony w trakcie deklaracji.

3.6.2 Listy

W języku Java występują również **listy**, które są typem służącym do przechowywania danych. **Lista** jest elastyczną wersją **tablicy**.

Jeśli chodzi o listy, to mamy dwa rodzaje implementacji:

- **ArrayList** – implementacja tablicowa
- **LinkedList** – implementacja wiązana

Implementacje **tablicową** stosujemy najczęściej w momencie, w którym zależy nam na czasie dostępu do danych, zaś implementacje **wiązaną** – gdy będziemy często działać na operacjach **usuwania** i **dodawania**.

Listy deklarujemy identycznie jak inne obiekty. Jeśli zadeklarujemy je jako **List** to nasz kod będzie bardziej uniwersalny, a w dowolnym momencie będziemy mogli podmienić implementację bez zmiany pozostałego kodu. Wygląda to tak:

```
List lista1 = new ArrayList();
List lista2 = new LinkedList();
```

Listing 3.70 Deklaracja list

Gdy wpiszesz poniższy kod w funkcji main, z pewnością napisy List oraz ArrayList i LinkedList podkreślą ci się na czerwono. Dlatego, że musimy zaimportować odpowiednie biblioteki. Dopisz na samej górze:

```
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
```

Listing 3.71 Zaimportowanie bibliotek obsługujących listy

Istnieje również inny sposób tworzenia list:

```
ArrayList<Integer> lista1 = new ArrayList<Integer>();
LinkedList<String> lista2 = new LinkedList<String>();
```

Listing 3.72 Inny sposób zadeklarowania list

Najpierw wybieramy jedną z dwóch implementacji – **Array** lub **Linked**. Następnie pomiędzy znakami „< >” zapisujemy typ listy. Później umieszczamy **nazwę** listy oraz **znak równości**, a na końcu słówko **new**.

W Javie mamy do dyspozycji kilka podstawowych metod do obsługi list niezależnie od tego, którą implementację wybierzemy. Są to:

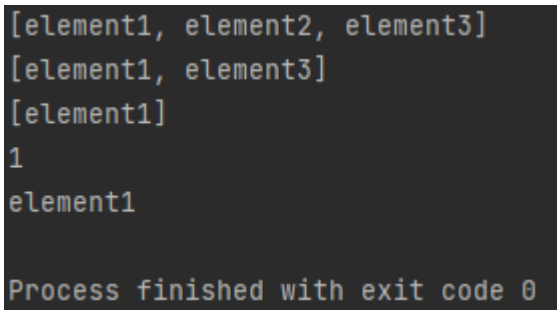
- **add(Object)** – dodawanie elementu do listy;
- **remove(Object/int)** – usuwanie elementu lub określonego indeksu z listy;
- **size()** - sprawdzanie rozmiaru listy (działa tak samo jak length() przy tablicach);
- **get(int)** – pobieranie określonego indeksu z listy;

Teraz pokażemy działanie tych metod w praktyce. Zapisz poniższy kod:

```
ArrayList<String> lista = new ArrayList<String>();
//dodanie elementów do listy, następnie wypisanie
zawartości listy
lista.add("element1");
lista.add("element2");
lista.add("element3");
System.out.println(lista);

//usuwanie elementu z listy o danej nazwie, następnie
wypisanie zawartości listy
lista.remove("element2");
System.out.println(lista);

//usuwanie elementu z listy o danym indeksie, następnie
wypisane zawartości listy
lista.remove(1);
System.out.println(lista);

//wypisanie rozmiaru listy po dodaniu i usunięciu
elementów
System.out.println(lista.size());

//pobranie elementu z listy o indeksie 0, następnie
```

Listing 3.73 Działania na listach

W komentarzach jest opisane dokładne działanie wszystkich metod. Dokładnie to, powinno się wyświetlić w konsoli:



Rysunek 3.21 Rezultat działań na listach w konsoli

Na początku widzimy wszystkie elementy listy – tak jak je dodaliśmy. Następnie wyświetlają się tylko dwa elementy.

Po usunięciu **dwóch** z **trzech** elementów, wyświetla nam się **rozmiar** listy, a na samym końcu wypisujemy element o indeksie **0** – **element1**.

3.6.3 Mapy

W języku Java **Mapa** to struktura danych, która składa się z kluczy **i** odpowiadających im wartości. Można to uznać za parę: **klucz-wartość**. Każdy klucz jest **unikalny** i odpowiada **jednej** wartości.

Mapy są bardzo przydatne, gdy potrzebujemy przechowywać dane w pamięci, a później szybko wyszukiwać **wartości** na podstawie **kluczy**.

W Javie występuje wiele implementacji interfejsu **Map**, ale najczęściej używane to:

- HashMap;
- TreeMap;
- LinkedHashMap.

HashMap jest najczęściej stosowana, ponieważ zapewnia stały czas dostępu do elementów.

TreeMap sortuje elementy według klucza, a **LinkedHashMap** zachowuje kolejność wstawiania elementów.

Przykładowo, możemy utworzyć **HashMapę** przechowującą nazwy państw w Europie jako klucze i odpowiadające im liczbę ludności jako wartości:

```
import java.util.HashMap;
import java.util.Map;
public class InputData9 {
    public static void main(String[] args) {
Map<String, Integer> populacjaPanstw = new HashMap<>();
  populacjaPanstw.put("Niemcy", 82370000);
  populacjaPanstw.put("Francja", 62151000);
  populacjaPanstw.put("Ukraina", 45994000);
  populacjaPanstw.put("Hiszpania", 40491000);
  populacjaPanstw.put("Polska", 38501000);
    }
}
```

Listing 3.74 Utworzenie mapy i dodanie do niej wartości oraz kluczy

W powyższym przykładzie tworzymy **HashMapę** p**opulacjaPanstw**, której **klucze** są typu **String j** (nazwę państwa) i **Integer** jako **wartość** (liczbę ludności).

Następnie dodajemy do niej **pięć** elementów przy użyciu metody **put()**, aby uzyskać wartość dla danego klucza, używamy metody **get()**. Należy też zaimportować odpowiednie dla map biblioteki **Map** oraz **HashMap**.

```
System.out.println("Liczba ludności w Polsce: " +
populacjaPanstw.get("Polska"));
System.out.println("Liczba podanych państw: " +
populacjaPanstw.size());

populacjaPanstw.remove("Niemcy");
System.out.println("Liczba podanych państw: " +
populacjaPanstw.size());
```

Listing 3.75 Przykłady użycia metod na mapach

W przykładzie wypisujemy liczbę **ludności** w Polsce i całkowitą liczbę **państw** w **mapie**.

Elementy z mapy możemy usunąć przy użyciu metody **remove()**. W przykładzie usuwamy **Niemcy**, a następnie wypisujemy zaktualizowaną liczbę państw znajdujących się w mapie.

```
Liczba ludnosci w Polsce: 38501000
Liczba podanych panstw: 5
Liczba podanych panstw: 4

Process finished with exit code 0
```

Rysunek 3.22 Wynik działania metod na mapach

Metoda **get()** użyta na mapie zwróci liczbę ludności w Polsce. Następnie otrzymamy liczbę państw, które posiadamy w naszej mapie. W pierwszym przypadku zwrócona wartość to **5**, a w kolejnym już **4,** ponieważ za pomocą metody **remove()** usunęliśmy państwo **Niemcy**. Ręczne wypisywanie wszystkich elementów mapy za pomocą metody **get()** byłoby zbyt czasochłonne. Posłużymy się pętlą **foreach**, aby je wyświetlić.

```
for(Map.Entry<String, Integer> x :
populacjaPanstw.entrySet()){
    String panstwo = x.getKey();
    int populacja = x.getValue();
    System.out.println("Populacja państwa " + panstwo +
" wynosi: " + populacja);
}
```

Listing 3.76 Wypisywanie elementów mapy za pomocą pętli „foreach”

W tym fragmencie kodu użyto pętli foreach. Służy ona do przejścia przez wszystkie pary **klucz-wartość** w mapie **populacjaPanstw**. W każdej iteracji pętli, zmienna **x** ma typ **Map.Entry<String, Integer>**, co oznacza, że przechowuje ona **jedną** parę **klucz-wartość** w mapie.

Zmienna **panstwo** jest inicjalizowana z kluczem w danej parze **klucz-wartość**, który reprezentuje nazwę kraju, a zmienna **populacja** jest inicjalizowana wartością w danej parze **klucz-wartość**, która reprezentuje populację kraju. Następnie jest wyświetlana informacja o populacji kraju, używając **System.out.println()**, która wyświetla nazwę kraju i jego populację. W konsoli powinno to wyglądać następująco:

```
Populacja panstwa Hiszpania wynosi: 40491000
Populacja panstwa Francja wynosi: 62151000
Populacja panstwa Ukraina wynosi: 45994000
Populacja panstwa Polska wynosi: 38501000

Process finished with exit code 0
```

Rysunek 3.23 Wynik konsoli po wypisaniu elementów mapy

Mapa nie zezwala na posiadanie dwóch identycznych kluczy, ale wartości mogą się już powtarzać. W **HashMap** i **LinkedHashMap** dozwolone są klucze oraz wartości zerowe, z kolei **TreeMap** nie zezwala na nie. Poniżej znajduje się lista wybranych metod z interfejsu map:

- **get()** - zwraca wartość przypisaną do danego klucza lub null, jeśli klucz nie istnieje w mapie.
- **size()-** zwraca liczbę par klucz-wartość w mapie.
- **keySet()** - zwraca set kluczy w mapie.
- **entrySet()** - zwraca set obiektów Map.Entry, które reprezentują pary klucz-wartość w mapie.
- **put()** - dodaje parę klucz-wartość do mapy lub aktualizuje wartość, jeśli klucz już istnieje.
- **clear()** - usuwa wszystkie pary klucz-wartość z mapy.
- **containsKey()** - zwraca true, jeśli mapa zawiera podany klucz.
- **containsValue()** - zwraca true, jeśli mapa zawiera podaną wartość.
- **equals()** - zwraca true, jeśli mapa jest równa podanemu obiektowi.
- **hashCode()** - zwraca hashcode dla mapy.
- **isEmpty()** - zwraca true, jeśli mapa nie zawiera żadnych par klucz-wartość.
- **putAll()** - dodaje wszystkie pary klucz-wartość z podanej mapy do tej mapy.
- **remove()** - usuwa parę klucz-wartość o podanym kluczu.
- **values()** - zwraca zbiór wartości w mapie.

3.6.4 Sprawdź się!

1) Do jakiego typu kolekcji można wykorzystać następujące metody: put(key, value), get(key), containsKey(key), containsValue(value) oraz keySet()?

a) Tylko do tablic;

b) Tylko do list;

c) Tylko do map;

d) Tylko do zbiorów;

e) Zarówno do list, jak i do map;

f) Zarówno do map, jak i do zbiorów;

2) Jak uzyskać długość tablicy w języku Java?

a) tablica.length();

b) tablica.size();

c) tablica.length;

3) Jak dodać parę klucz-wartość do mapy w języku Java?

a) map.add(key, value);

b) map.put(key, value);

c) map.insert(key, value);

Zadanie 3.6.1

Napisz program który oblicza średnią ocen studenta na podstawie zdefiniowanej wcześniej tablicy ocen.

3.7 Pętle

W poprzednich rozdziałach zapoznaliśmy się z tablicami i listami. Obie struktury są powszechnie używane w programowaniu. Za pomocą odpowiednich metod możemy je edytować. Możemy do nich dodawać, usuwać elementy oraz zmieniać ich wartości.

Rozpatrzmy jednak przypadek, gdy mamy **100-elementową** tablicę i chcemy w niej dokonać modyfikacji na większej ilości elementów.

Musielibyśmy wielokrotnie używać tej samej metody. Taka liczba operacji zajęłaby nam mnóstwo czasu. Co w przypadku gdy liczba elementów tablicy będzie wynosić **1000 lub więcej**? Rozwiązaniem tego problemu jest zastosowanie pętli.

Jest to jedna z podstawowych konstrukcji występująca we wszystkich językach programowania. Są one stosowane niemal w każdym programie. Dzięki nim możemy wykonywać określoną metodę, funkcję czy czynność tyle razy ile ustalimy lub do momentu, w którym jakiś warunek się nie spełni. W Javie mamy kilka rodzajów pętli są to:

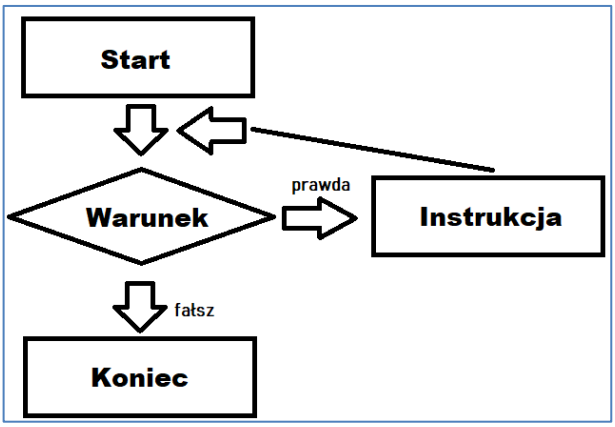
- While;
- do... while;
- for;
- foreach;

3.7.1 Pętla while

Pętli **while** używamy w sytuacji, kiedy nie znamy dokładnej ilości wykonanych operacji. W skrócie – pętla będzie działać **tak długo**, dopóki warunek **będzie prawdziwy**.

Jeśli jednak warunek **nie będzie w ogóle spełniony** (już na początku jej uruchamiania) to nie **uruchomi się** ona wcale.

Schemat blokowy wraz ze schematem ogólnym pętli while zostały zamieszczony poniżej:



Rysunek 3.24 Schemat pętli „while”

```
while(true){
    //instrukcja do wykonania
}
```

Listing 3.77 Schemat ogólny pętli „while"

Na początku słówko kluczowe **while**, następnie w nawiasie znajduje się **warunek**, a w środku pętli – **instrukcje**, które będą się wykonywały podczas trwania tej pętli.

Zobaczmy teraz jak działa pętla **while** w praktyce:

```
int i = 0;
while(i<10){
    i++;
    System.out.println("Pętla wykonała się już po raz "
+ i);
}
```

Listing 3.78 Zastosowanie pętli „while"

Na początku zadeklarowaliśmy zmienną typu **int.** Przechowuje ona określoną wartość – w tym przypadku jest to **liczba 0.**

Następnie pojawia się **pętla while**. Warunkiem jej wykonywania jest porównanie logiczne **i<10**. Oznacza ona, że pętla będzie się wykonywała tak długo, dopóki wartość zmiennej i będzie mniejsza od **liczby 10**.

Dzięki inkrementacji **i++** w ciele pętli zwiększamy wartość zmiennej **i o jeden**. Instrukcja **System.out.println**, wypisze tekst, który będzie nas informował o tym **ile razy** wykonała się pętla. Spójrzmy na działanie programu:

```
Pętla wykonała się już po raz 1
Pętla wykonała się już po raz 2
Pętla wykonała się już po raz 3
Pętla wykonała się już po raz 4
Pętla wykonała się już po raz 5
Pętla wykonała się już po raz 6
Pętla wykonała się już po raz 7
Pętla wykonała się już po raz 8
Pętla wykonała się już po raz 9
Pętla wykonała się już po raz 10

Process finished with exit code 0
```

Rysunek 3.25 Rezultat działania programu pętli while

Jak widać, pętla wykonała się dokładnie **10** razy po czym zakończyła działanie. Na początku nasza zmienna i była równa **0**. Po wykonaniu się pętli po raz pierwszy, jej wartość zwiększyła się i wynosiła **1**. Po **dziesiątym** wykonaniu pętli, wartość wzrosła do **10**, a następnie program zatrzymał pętle, ponieważ warunek przestał być prawdziwy.

3.7.2 Pętla do while

Teraz czas na pętle **do while**. Na pierwszy rzut oka wygląda podobnie do pętli **while**.

Różni się od niej przede wszystkim tym, że bez względu na warunek pętla i tak wykona się przynajmniej jeden raz. Dzieje się tak ponieważ, **warunek** jest sprawdzany dopiero po wykonaniu **instrukcji** pętli, tak więc na początku nie bierze się go w ogóle pod uwagę. Schemat ogólny pętli **do while** wygląda tak:

```
do{
    //instrukcja do wykonania
}while(true);
```

Listing 3.79 Schemat ogólny pętli „do while"

Konstrukcja tej pętli różni od zwykłej pętli **while**. Zaczynamy tym razem od słówka kluczowego **do**, później umieszczamy w środku **instrukcje** do wykonania, a na końcu umieszczamy **warunek** pętli poprzedzony słówkiem kluczowym **while**. Tu powinien znaleźć się także średnik.

Zobaczmy zatem jak działa pętla **do while**:

```
int i = 10;
do{
  i++;
 System.out.println("Zmienna i wynosi: " + i);
}while(i<10);
```

Listing 3.80 Zastosowanie pętli „do while"

A oto rezultat:

```
Zmienna i wynosi: 11

Process finished with exit code 0
```

Rysunek 3.26 Wynik działania programu po użyciu pętli „do while"

Gdybyśmy użyli zwykłej pętli **while** przy zmiennej **i=10**, to ta nie wykonałaby się w ogóle. W przypadku pętli **do while** instrukcja wykona się **jeden raz** bez względu na **warunek**.

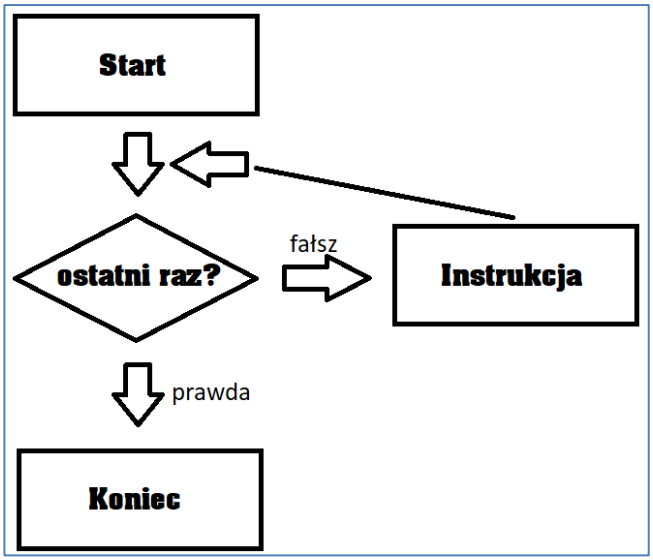
3.7.3 Pętla for

Przyszedł czas na **pętle for** – jedną z najczęściej używanych i zarazem najprostszych pętli w Javie. Używamy jej kiedy znamy dokładną ilość wykonań naszej pętli. Na przykład gdybyśmy chcieli zwiększyć wartość zmiennej **liczba 100** razy.

Schemat ogólny działania pętli prezentuje się następująco:

```
for(int i=0; i<10; i++){
    //instrukcja do wykonania
}
```

Listing 3.81 Schemat ogólny pętli „for”



Rysunek 3.27 Schemat ogólny pętli „for”

Przejdźmy więc do pętli for w praktyce:

```
int liczba = 100;
for(int i=0; i<10; i++){
    liczba--;
}
System.out.println(liczba);
```

Listing 3.82 Zastosowanie pętli „for”

Na początku programu zadeklarowaliśmy zmienną typu **int** o nazwie **liczba**, której wartość początkowa jest równa **100**.

Pętla **for** składa się z trzech instrukcji oddzielonych od siebie średnikami.

- Pierwsze z nich to tzw. **wyrażenie początkowe**;
- W środku znajdziemy **warunek** (podobnie jak w pętli while);
- Ostatni to tzw. modyfikator licznika;

Wewnątrz **pętli** (pomiędzy nawiasami klamrowymi) zamieszczamy **instrukcję do wykonania**.

W powyższym przykładzie **zmienna i**, od której zaczynamy ma **wartość początkową** równą **0**.

Biorąc pod uwagę, że wartość równa się **0** i zwiększamy ją o **jeden** za każdym przejściem pętli, wykona się ona dokładnie **dziesięć razy**, gdyż po przekroczeniu wartości **10** warunek przestaje być prawdziwy.

Działanie programu ilustruje poniższy rysunek:

```
90

Process finished with exit code 0
```

Rysunek 3.28 Wynik działania programu z użyciem pętli „for”

Napiszemy program, który wypisze kolejne liczby parzyste począwszy od **2**.

```
for(int i=0; i<20; i+=2){
    System.out.print(i + ",");
}
```

Listing 3.83 Wypisywanie liczb parzystych przy użyciu pętli „for”

W tym przykładzie zmienna **i** ma na początku wartość **0**.

Następnie jej wartość zwiększa się za każdym razem o **2**, dopóki nie osiągnie wartości większej od **20**. Program w każdym przejściu pętli wypisuje zawartość zmiennej, oraz przecinek.

Wynik działania możemy zobaczyć na rysunku poniżej:

```
0,2,4,6,8,10,12,14,16,18,
Process finished with exit code 0
```

Rysunek 3.29 Liczby parzyste wypisane przez pętle „for”

W kolejnym przykładzie policzymy sumę liczb z przedziału od **0** do **100**, podzielnych przez **5**.

Pętla wykona się **sto razy** – od zadeklarowanego **i = 0** do końca warunku, który wynosi **100**. Podczas działania pętla sprawdzi czy aktualna liczba jest podzielna **przez 5**. Wykorzystana jest w tym miejscu instrukcja warunkowa. Sprawdza ona czy reszta z dzielenia liczby **przez 5** jest równa **0**. Jeżeli tak jest, wybrana liczba zostanie dodana do zmiennej **suma**.

Po przejściu pętli program wypisze zawartość zmiennej **suma.**

```
int suma = 0;
for(int i=0; i<=100; i++){
    if(i % 5 == 0){
        suma+=i;
    }
}
System.out.println("Suma wynosi: " + suma);
```

Listing 3.84 Wykorzystanie pętli „for" do obliczenia sumy liczb podzielnych przez 5 z określonego przedziału

```
Suma wynosi: 1050

Process finished with exit code 0
```

Rysunek 3.30 Wypisanie sumy zmiennej w konsoli

3.7.4 Wykorzystanie pętli w obsłudze tablic i list

Pętle są bardzo przydatne przy obsłudze **tablic** i **list**.

Przydadzą się np. w momencie, w którym chcemy przejść przez każdy element danej **struktury** i wykonać na nim określoną operację.

Na początek zadeklarujemy zwykłą tablicę, a następnie uzupełnimy ją liczbami od **1** do **5**.

```
int[] tablica = new int[5];
for(int i=0; i<tablica.length; i++){
    tablica[i] = i;
}
```

Listing 3.85 Tablica typu int wypełniona liczbami z pętli „for"

Przed użyciem **pętli** została zainicjalizowana **tablica**. Do określenia **warunku** wykorzystano metodę **length**, która **policzy** ilość elementów znajdujących się w tablicy. Z wartością tą będzie porównywane **i**.

Następnie w **pętli** użyliśmy funkcji **tablica.length**, która określa nam ilość elementów czyli długość **tablicy**. To znaczy, że pętla będzie tak długa, jak długa jest tablica. Wewnątrz **for'a** przypisujemy poszczególnym elementom tablicy wartość zmiennej **i.**

W kolejnym przykładzie użyjemy pętli **for**, aby wypisać zawartość tablicy:

```
int[] tablica = new int[5];
for(int i=0; i<tablica.length; i++){
    tablica[i] = i;
}
for(int i=0; i<tablica.length; i++){
   System.out.println("tablica["+i+"] = " + tablica[i]);
}
```

Listing 3.86 Wypisanie wartości z tablicy za pomocą pętli „for"

Do poprzedniego fragmentu dopisujemy nową pętle **for**, w której wypisujemy wartości znajdujące się wewnątrz tablicy. Wynik w konsoli będzie następujący:

```
tablica[0] = 0
tablica[1] = 1
tablica[2] = 2
tablica[3] = 3
tablica[4] = 4

Process finished with exit code 0
```

Rysunek 3.31 Wynik konsolowy wypisania zawartości tablicy

Jak widzisz każdy element tablicy ma wartość równą swojemu indeksowi.

Czasami podczas używania pętli chcielibyśmy, żeby ona się wykonała określoną ilość razy, ale przerwała w jakimś momencie, gdy spełni się na przykład jakiś warunek. Możemy wówczas wykorzystać znaną już instrukcję break.

W kolejnym przykładzie zaprezentujemy jej zastosowanie w pętli.

Program przerwie działanie pętli, kiedy znajdzie **liczbę 3**. Tym razem posłużymy się listą.

Listę uzupełnimy wybranymi liczbami z przedziału **1-7**. Jeśli **pętla** napotka **liczbę 3** to zwyczajnie nam przerwie jej działanie.

Najpierw zaczniemy od stworzenia listy, uzupełnienia jej oraz wypisania wszystkich liczb, które się w niej znajdują. Robimy to, w ten sposób:

```
ArrayList<Integer> lista = new ArrayList<Integer>();

lista.add(6);
lista.add(4);
lista.add(1);
lista.add(3);
lista.add(7);

for(int i=0; i<lista.size(); i++){
    System.out.print(lista.get(i) + ", ");
}
```

Listing 3.87 Zadeklarowanie listy, uzupełnienie jej liczbami i wypisanie za pomocą pętli „for"

Na początek deklarujemy prostą listę typu **int**, a następnie dodajemy do niej **pięć** elementów. Wykorzystujemy w tym miejscu metodę **add**. Pętla **for** ma za zadanie wypisać zawartość listy na ekran. W tym ćwiczeniu metodę **length** zastąpiliśmy metodą **size**. Działa ona podobnie i zwraca dokładnie tą samą wartość – **rozmiar listy**.

Metoda **get()** pobiera numer indeksu każdego z elementów listy. Konsola powinna zwrócić dokładnie to:

```
6, 4, 1, 3, 7,
Process finished with exit code 0
```

Rysunek 3.32 Wynik konsoli po wypisaniu liczb listy

Powyższy kod uzupełniamy następująco:

Na początek tworzymy **pętle**, w której zastosujemy instrukcję **break**. Zostanie ona uruchomiona w momencie, kiedy program pobierze z indeksu **wartość 3**. Do sprawdzenia pobranego **indeksu** wykorzystano **instrukcję warunkową**.

Kod programu będzie wyglądał następująco:

```
ArrayList<Integer> lista = new ArrayList<Integer>();

lista.add(6);
lista.add(4);
lista.add(1);
lista.add(3);
lista.add(7);

for(int i=0; i<lista.size(); i++){
    if(lista.get(i) == 3){
        break;
    } else{
       System.out.println(lista.get(i) + ", ");
        continue;
    }
}
```

Listing 3.88 Wypisywanie wartości z listy za pomocą pętli, dopóki ta nie trafi na liczbę 3

```
6,4,1,
Process finished with exit code 0
```

Rysunek 3.33 Wynik konsoli tego programu

3.7.5 Sprawdź się!

1) **Wyjaśnij, kiedy należy używać pętli do-while zamiast pętli while w Javie. Podaj przykłady sytuacji, w których pętla do-while jest bardziej odpowiednia.**

2) **Ile razy pętla for zostanie wykonana w poniższym kodzie?**

```
int suma = 0;
for (int i = 0; i <= 10; i++) {
    suma += i;
}
```

a) 10 razy

b) 11 razy

c) 9 razy

d) 100 razy

3) Co się stanie, gdy nie podamy warunku w deklaracji pętli while?

a) Kompilator wyświetli błąd i nie będzie można skompilować kodu.

b) Pętla while będzie wykonywana bezwarunkowo, dopóki nie zostanie przerwana przez instrukcję break.

c) Pętla while nie będzie wykonywana ani razu, a program przejdzie do następnej instrukcji poza pętlą.

d) Pętla while będzie kontynuowana w nieskończoność, powodując zawieszenie programu.

4) Jak wykonać pętlę do-while w języku Java?

a) do { } while (condition);

b) while (condition) { } do;

c) while (condition) { } until;

Zadanie 3.7.1

Napisz program w języku Java, który obliczy **silnię** zadanej liczby całkowitej za pomocą **pętli**. Silnia liczby n, oznaczana symbolem **n!**, to iloczyn wszystkich **dodatnich** liczb **całkowitych** mniejszych lub równych **n**. Na przykład, **5! = 5 * 4 * 3 * 2 * 1 = 120**.

Zadanie 3.7.2

Napisz program wyświetlający liczby całkowite z przedziału **<100, 1>** w porządku malejącym oraz skoku równym **2**.

3.8 Zakres zmiennych

W poprzednich rozdziałach poszerzyliśmy naszą wiedzę na temat tworzenia zmiennych, korzystania z instrukcji warunkowych czy pętli. Teraz, zajmiemy się kolejnym ważnym zagadnieniem w programowaniu w języku Java – **zakresem widoczności zmiennych**.

Zakres widoczności zmiennych to po prostu obszar, gdzie nasze zmienne mogą działać i być zrozumiane. To ważne, bo wpływa na to, jak bezpieczny i zorganizowany jest nasz kod.

Rozważmy zatem prosty przykład z **klasą**, gdzie zasięg widoczności zmiennej w klasie jest ogólnodostępny. Przejdźmy do utworzenia klasy **ZakresKlasa**, a następnie zadeklarujmy w niej **zmienną** typu **int** o nazwie **widocznaZmienna**:

```
class ZakresKlasa {
    int widocznaZmienna = 5; //ogólnodostępna zmienna
}
```

Listing 3.89 Klasa z ogólnodostępną zmienną

Teraz, gdy posiadamy tę **zmienną**, sprawdźmy, czy rzeczywiście możemy z niej korzystać, na przykład w **metodach**. W tym celu dodajmy wewnątrz **klasy** dwie nowe metody o dowolnych nazwach. W obu z nich spróbujemy wypisać **wartość** naszej **zmiennej**. Możemy to zrobić w ten sposób:

```
public class ZakresKlasa {
    int widocznaZmienna = 5; //ogólnodostępna zmienna

    public void wypiszWartosc(){
        // możemy korzystać z niej w tej metodzie
        System.out.println("Wartość zmiennej: " +
widocznaZmienna);
    }
    public void podwojWartosc(){
        widocznaZmienna*=2;
        // możemy korzystać z niej w tej metodzie
        System.out.println("Wartość zmiennej: " +
widocznaZmienna);
    }
}
```

Listing 3.90 Klasa z dwoma metodami wypisującymi zmienną

W tym przypadku, **wspolnaZmienna** jest dostępna dla obu **metod** w **klasie**. Można nazwać ją takim narzędziem dostępnym dla wszystkich.

Teraz zastanówmy się, co się stanie, jeśli stworzymy nową zmienną w jednej z metod i spróbujemy użyć jej poza zakresem. Aby to zobaczyć, dodajmy kolejną dowolną metodę na wzór poprzednich i zadeklarujmy w niej zmienną o nazwie **lokalnaZmienna**:

```
public void trzeciaMetoda(){
    int lokalnaZmienna = 6; // lokalna zmienna dostępna
tylko w trzeciaMetoda
    System.out.println("Wartość zmiennej: " +
lokalnaZmienna);
}
```

Listing 3.91 Metoda zawierająca zmienną lokalną i wypisującą jej wartość

Gdybyśmy uruchomili nasz program, wywołując tę metodę, wynik bez problemu pojawiłby się w konsoli. Teraz jednak umieśćmy tę zmienną w jednej z pozostałych metod i zobaczmy, co się stanie podczas próby skompilowania kodu. Na przykład:

```
public void wypiszWartosc(){
    System.out.println("Wartość zmiennej: " +
widocznaZmienna);

    // próba użycia lokalnaZmienna spoza jej zakresu
(nie skompiluje się)
    System.out.println("Wartość zmiennej: " +
lokalnaZmienna);
}
```

Listing 3.92 Umieszczenie zmiennej lokalnej w innej metodzie

Niestety, nie uda nam się skompilować tego kodu ze względu na brak widoczności zmiennej **lokalnaZmienna** przez metodę **wypiszWartosc**. Innymi słowy, **lokalnaZmienna** jest prywatną zmienną jedynie dla metody **trzeciaMetoda**, co oznacza, że jest dostępna tylko w jej obrębie i nie może być używana w innych częściach kodu.

Zrozumieliśmy już, jak kontrolować zmienne w **klasach** i **metodach**. Teraz zajmiemy się **pętlami**, w których najczęściej korzystamy z zmiennych na krótki czas. Utwórzmy w **klasie** nową **metodę**, a w niej **pętle**. Przykładowo:

```
public void petla(){
    for(int i = 0; i < 3; i++){
        // zmienna i jest dostępna tylko wewnątrz pętli
        System.out.println("Obieg pętli nr " + i);
    }
    // poza pętlą zmienna i już nie istnieje
    System.out.println(i);
}
```

Listing 3.93 Metoda zawierającą pętle „for” i zmienną „i” dostępną tylko w pętli

Wartości zmiennych w pętlach są dostępne jedynie wewnątrz pętli, co sprawia, że są używane w konkretnej sytuacji i nie istnieją poza nią. W powyższym przykładzie, taką zmienną jest **i**. Tak jak w poprzednim przypadku **trzeciaMetoda** – nasza metoda **petla** się nie skompiluje.

Na koniec przyjrzymy się **zmiennej lokalnej** dla konkretnego **bloku**. Przechodząc od **pętli** do **zakresu zmiennych** w **blokach**, warto zauważyć, że zmienne lokalne mogą istnieć tylko w określonych fragmentach kodu.

Aby zobrazować to lepiej, utworzymy nową metodę, w której znajdzie się **instrukcja warunkowa** wraz ze **zmienną**. Oto przykład:

```
public void blokowaLogika(boolean warunek){
    int zmiennaZewnetrzna = 10; // zmienna dostępna w
całej metodzie

    if (warunek) {
        int zmiennaBlokowa = 5; // zmienna dostępna
tylko wewnątrz tego bloku
        System.out.println("Warunek spełniony. Wartość
zmiennej: " + zmiennaBlokowa);

        // możemy korzystać z zmiennych zewnętrznych
wewnątrz bloku
        System.out.println("Suma" +
(zmiennaBlokowa+zmiennaZewnetrzna));
    } else {
        // w tym miejscu zmiennaBlokowa nie jest już
dostępna
        System.out.println("Warunek nie jest
spełniony");
    }
    // w tym miejscu zmiennaBlokowa nie jest już
dostępna
}
```

Listing 3.94 Metoda zawiera zmienną warunkową, posiadającą zmienną lokalną

W powyższym przykładzie zmienna „**zmiennaBlokowa**” jest widoczna jedynie w obrębie bloku if. Jeśli warunek jest spełniony, mamy dostęp do tej zmiennej wewnątrz bloku oraz do zmiennej „**zmiennaZewnetrzna**”. Jeśli warunek nie został spełniony to wewnątrz bloku else mamy dostęp tylko i wyłącznie do **„zmiennaZewnetrzna”**. Użycie „**zmiennaBlokowa**” w innymi miejscu niż blok **if** spowoduje błąd.

Zrozumienie zakresu widoczności zmiennych jest kluczowe dla utrzymania porządku w kodzie i unikania błędów. Pozwala nam to kontrolować dostępność zmiennych w różnych fragmentach programu. Wiedząc, gdzie dana zmienna jest dostępna, łatwiej jest śledzić i zrozumieć, jakie wartości może przyjmować oraz w jaki sposób jest wykorzystywana. Dzięki odpowiedniemu zarządzaniu zakresem zmiennych, nasze programy stają się czytelniejsze i bardziej przejrzyste.

3.9 Zarządzanie pamięcią – GarbageCollector

Kiedy piszemy program, koncentrujemy się głównie na projektowaniu i implementacji konkretnych elementów logiki biznesowej oraz na rozwiązywaniu napotkanych problemów. W miarę możliwości poprawiamy jakość samego kodu poprzez stosowanie dobrych praktyk programistycznych. Krótko mówiąc, dążymy do zbudowania funkcjonalnej aplikacji spełniającej nasze wymagania.

Każdy projekt, bez względu na swoją złożoność, dysponuje własną pamięcią, w której przechowuje wartości zmiennych, klasy oraz ich obiekty. Jednakże, nie jest to pamięć nieskończona. Często nie zwracamy uwagi na to, co dzieje się z utworzonymi obiektami kiedy przestajemy ich używać. Gdyby pozostawały w pamięci na stałe szybko mogłyby doprowadzić do jej zapełnienia. Na szczęście istnieje mechanizm który lokalizuje i usuwa nieużywane obiekty. Jest to **Garbage Collector**.

W ogólnym rozumieniu, **Garbage Collector (GC)** jest mechanizmem odpowiedzialnym za automatyczne zarządzanie pamięcią w programach komputerowych. Jego głównym zadaniem jest odzyskiwanie pamięci zajmowanej przez obiekty, które nie są już używane przez program, co pozwala uniknąć wycieków zawartości i utrzymać stabilność oraz wydajność systemu. W Javie, GC jest integralną częścią maszyny wirtualnej (JVM) jednak można go spotkać również w innych językach programowania, takich jak na przykład C# czy Python.

Żeby łatwiej zobrazować sobie na czym polega sprzątanie pamięci z nieużywanych obiektów spójrzmy na poniższy fragment kodu.

```
  public class GarbageCollectorExample {
    public static void main(String[] args) {

        String imie1 = "Adam";
        String imie2 = "Bartek";

        imie1 = imie2;
    }
}
```

Listing 3.95 Przykład utraty referencji do wartości „Adam”

Jak można zauważyć początkowa wartość zmiennej **imie1** została nadpisana. Teraz obie przechowują imię **Bartek**. Co zatem stało się z wartością **Adam**. Nie ma już do niej dostępu, straciła referencję do siebie a jednak nadal zajmuje miejsce w pamięci aplikacji. Gdy Garbage Collector rozpocznie proces sprzątania ta wartość zostanie usunięta. Teraz wyobraźmy sobie podobne nadpisywanie wartości w pętli wykonującej się kilka tysięcy razy. Z punktu widzenia złożoności tego programu mamy do czynienia z trywialnym przykładem a jednak generuje on zbędne śmieci. Wielokrotnie podobne błędy popełniamy w znacznie większych projektach produkując masę zbędnych obiektów.

Jeszcze nie tak dawno, gdy garbage collector nie istniał, programiści byli zmuszeni samodzielnie zarządzać pamięcią w swoich programach. Każda alokacja pamięci musiała być dokładnie śledzona, a każda zwalniana pamięć musiała być ręcznie uwalniana. Ta praktyka była nie tylko niewygodna i czasochłonna, ale także bardzo podatna na błędy.

Sam sposób działania nie jest pojedynczym schematem, lecz opiera się na różnych algorytmach dzięki którym potrafi dostosować swoją strategię oczyszczania do aktualnej sytuacji w której znajduje się aplikacja. Jednym z popularniejszych i łatwiejszych do zrozumienia jest **Mark-Sweep-Compact** (Markowanie, Czyszczenie, Kompaktowanie) który polega na określeniu roota którym jest klasa, a następnie przejście po wszystkich jej instancjach i oznaczanie wszystkich obiektów posiadających referencje. Pozostałe nieoznaczone obiekty są usuwane a pamięć zwalniana.

Garbage Collector jest mechanizmem automatycznym. Nie musimy wywoływać go aby rozpoczął pracę. Nie potrzebuje on też żadnej konfiguracji z naszej strony. Jest natomiast możliwe ręczne wywołanie go za pomocą metody **system.gc()** która wskazuje miejsce gdzie warto przeprowadzić procedurę czyszczenia pamięci. Drugą dostępną metodą jest **finalize()** która pozwala na zaimplementowanie instrukcji która wykona się jako ostatnia czynność przed usunięciem obiektu. Nie jest jednak popularne używanie tych metod ponieważ ich działanie nie zawsze jest przewidywalne a w pewnych sytuacjach może powodować upośledzenia pracy **GC**. Są używane w wyjątkowych sytuacjach i raczej tylko w ostateczności lub podczas przeprowadzania testów aplikacji. Wiedzę o ich istnieniu i zastosowaniu powinno się traktować raczej jako ciekawostkę niemniej jednak wartą zapamiętania.

Podsumowując **Garbage Collector** jest wyjątkowo przydatnym narzędziem które jest w stanie zaoszczędzić mnóstwo pracy programisty. Poprzez sprawowanie kontroli nad wykorzystaniem zasobów pamięci realnie wpływa na wydajność i stabilność aplikacji. Jednak należy pamiętać że żaden mechanizm nie zastąpi zoptymalizowanego kodu. Skupiajmy się zatem na tworzeniu przemyślanych programów zamiast liczyć że system będzie myślał za nas.

3.10 Bloki try-catch

W poprzednich rozdziałach omówiliśmy już instrukcje warunkowe, które pomogły naszym programom podejmować różne decyzje. Teraz nadszedł czas, aby przyjrzeć się czemuś bardziej zaawansowanemu – instrukcji **try-catch**.

To narzędzie pozwala nam radzić sobie z błędami, które mogą pojawić się w trakcie działania naszego programu, ale także, pozwala je przewidywać. Możemy sobie wyobrazić, że **try-catch** to taki zestaw narzędzi służących do naprawy. Pomaga nam najpierw zidentyfikować problem, a następnie go rozwiązać.

Jednym z lepszych przykładów na przedstawienie tej instrukcji będzie próba podzielenia liczby przez zero. Przejdźmy zatem do praktycznego zastosowania try-catch:

```
public class InstrukcjaTryCatch {
    public static void main(String[] args) {
        int dzielna = 6;
        int dzielnik = 0;
        int wynik = dzielna/dzielnik;
    }
}
```

Listing 3.96 Próba podzielenia dzielnej przez 0

Na początku tworzymy nową klasę o nazwie **InstrukcjaTryCatch**, a w niej metodę **main**. Następnie deklarujemy trzy zmienne – **dzielna**, **dzielnik** oraz **wynik**. Każda typu int. **Dzielna** przyjmuje dowolną wartość większa od zera. W tym przykładzie jest to **liczba 6**. **Dzielnik** przyjmuje wartość **0**.

Wynik to iloraz zmiennej **dzielna** przez **dzielnik**. Przy próbie uruchomienia programu wyświetli nam się następujący błąd:

```
Exception in thread "main" java.lang.ArithmeticException Create breakpoint : / by zero
    at InstrukcjaTryCatch.main(InstrukcjaTryCatch.java:6)
```

Rysunek 3.34 Błąd spowodowany próbą dzielenia przez 0

Jak widać, niemożliwe jest podzielenie liczby przez **zero**. Teraz spróbujemy umieścić wcześniej wspomnianą instrukcję **try-catch** w powyższym kodzie. W metodzie **main** zapiszmy:

```
int dzielna = 6;
int dzielnik = 0;

try{
    int wynik = dzielna/dzielnik;
    System.out.println("Wynik dzielenia: " + wynik);
}catch(ArithmeticException e){
    System.err.println("Błąd arytmetyczny o komunikacie:
" + e.getMessage());
}
```

Listing 3.97 Próba dzielenia zmiennej przez 0 w bloku „try-catch"

Pod zmiennymi **dzielna** i **dzielnik**, używamy instrukcji **try** wewnątrz której znajduje się kod, który potencjalnie może spowodować wyjątek. W naszym programie jest to próba dzielenia przez zero, a potem przypisanie rezultatu dzielenia do zmiennej **wynik**.

Następnie wypisujemy wynik za pomocą instrukcji **System.out.println().** Jeśli w trakcie wykonywania kodu w bloku **try** wystąpi błąd (na przykład dzielenie przez zero), program przechodzi do bloku **catch**.

W nawiasie umieszczamy nazwę wyjątku, który „catch" obsłuży oraz nazwę obiektu (najczęściej w przypadku **catch** stosuje się literę **e**). W tym przypadku blok **catch** obsługuje wyjątek **ArithmeticException**, który jest wyświetlany w przypadku błędu arytmetycznego. Komunikat tego błędu wyświetlił się już przy pierwszej próbie kompilacji naszego programu.

Wewnątrz bloku **catch** znajduje się kod obsługujący błąd. Wyświetlamy komunikat o błędzie na standardowym wyjściu błędów **System.err**. Informacje na temat błędu są uzyskiwane z obiektu **e**, który jest instancją klasy **ArithmeticException**. **GetMessagge()** pozwoli pobrać obiektowi **e**, wiadomość na temat tego wyjątku. W konsoli powinien się pojawić poniższy komunikat:

```
Błąd arytmetyczny o komunikacie: / by zero
```

Rysunek 3.35 Wyświetlenie w konsoli komunikatu o błędzie

Ten przykład demonstruje, jak za pomocą instrukcji **try-catch** możemy kontrolować sytuacje, w których występują błędy (przykładowo takie jak dzielenie przez zero) i obsługiwać je w sposób kontrolowany, zamiast

pozostawiać program z widocznym błędem. Teraz pokażemy obsługę innego wyjątku związanego z tablicami. Utwórzmy zatem nową tablicę typu **int**, posiadającą trzy elementy:

```
int[] liczby = {1,2,3};
```

Listing 3.98 Utworzenie tablicy trzyelementowej

W tablicy znajdują się trzy liczby. Sprawdzimy co się stanie, gdybyśmy odwołali się do czwartego indeksu tablicy?

Przykładowo:

```
System.out.println(liczby[4]);
```

Listing 3.99 Próba odwołania do nieistniejącego indeksu tablicy

Po uruchomieniu programu, ten zwróci nam błąd:

```
Exception in thread "main" java.lang
 .ArrayIndexOutOfBoundsException Create breakpoint : Index 4 out of bounds
 for length 3
    at InstrukcjaTryCatch.main(InstrukcjaTryCatch.java:15)
```

Rysunek 3.36 Błąd odwołania do nieistniejącego indeksu tablicy

Nie możemy się odwołać do indeksu, który nie istnieje w danej tablicy. Obsłużymy teraz podany wyjątek za pomocą instrukcji **try-catch**. Cały kod powinien się prezentować następująco:

```
int[] liczby = {1,2,3};
try{
    System.out.println(liczby[4]);
}catch(ArrayIndexOutOfBoundsException e){
    System.err.println("Błąd indeksu tablicy: " +
e.getMessage());
}
```

Listing 3.100 Obsługa wyjątku indeksu tablicy za pomocą „try-catch”

Nasz **System.out.println()** umieściliśmy w bloku **try**, natomiast **System.err.println()** w bloku **catch**. Jeśli uruchomimy program, w konsoli otrzymamy:

```
Błąd indeksu tablicy: Index 4 out of bounds for length 3
```

Rysunek 3.37 Błąd indeksu tablicy wypisany w konsoli

Możemy również skorzystać z ogólnego przechwytywania **Exception**. Zastąpmy napis **ArrayIndexOutOfBoundsException** samym słowem **Exception**. Rezultat będzie ten sam, jednak stosowanie tego jest niezalecane, ponieważ

może utrudnić diagnozowanie i poprawianie błędów. Dodatkowo może ukryć szczegóły konkretnego rodzaju wyjątku.

Instrukcja **try-catch** w Javie umożliwia obsługę różnych rodzajów wyjątków. Te, które zostały użyte w przykładach powyżej – są jednymi z najpopularniejszych. Oprócz nich, popularne są też:

- **NullPointerException** – gdy próbujemy odwołać się do metody lub pola obiektu, który ma wartość „null".
- **NumberFormatException** – gdy próbujemy konwersji ciągu znaków na liczbę, a nie jest to możliwe.
- **IllegalArgumentException** – gdy próbujemy przekazać nieprawidłowe argumenty do metody.

To tylko kilka z dostępnych wyjątków. W Javie jest ich całe mnóstwo.

3.11 Sprawdź się!

1) Jakie są zalety garbage collectora?

a) Ręczne zarządzanie pamięcią, co pozwala na bardziej precyzyjną kontrolę nad zasobami

b) Automatyczne zwalnianie pamięci, eliminując potrzebę ręcznego zarządzania pamięcią i zapobiegając wyciekom pamięci.

c) Wymuszenie czyszczenia pamięci przy każdej operacji, co zapewnia natychmiastowe zwalnianie nieużywanych zasobów.

2) Jaka jest główna rola bloku try-catch w języku Java?

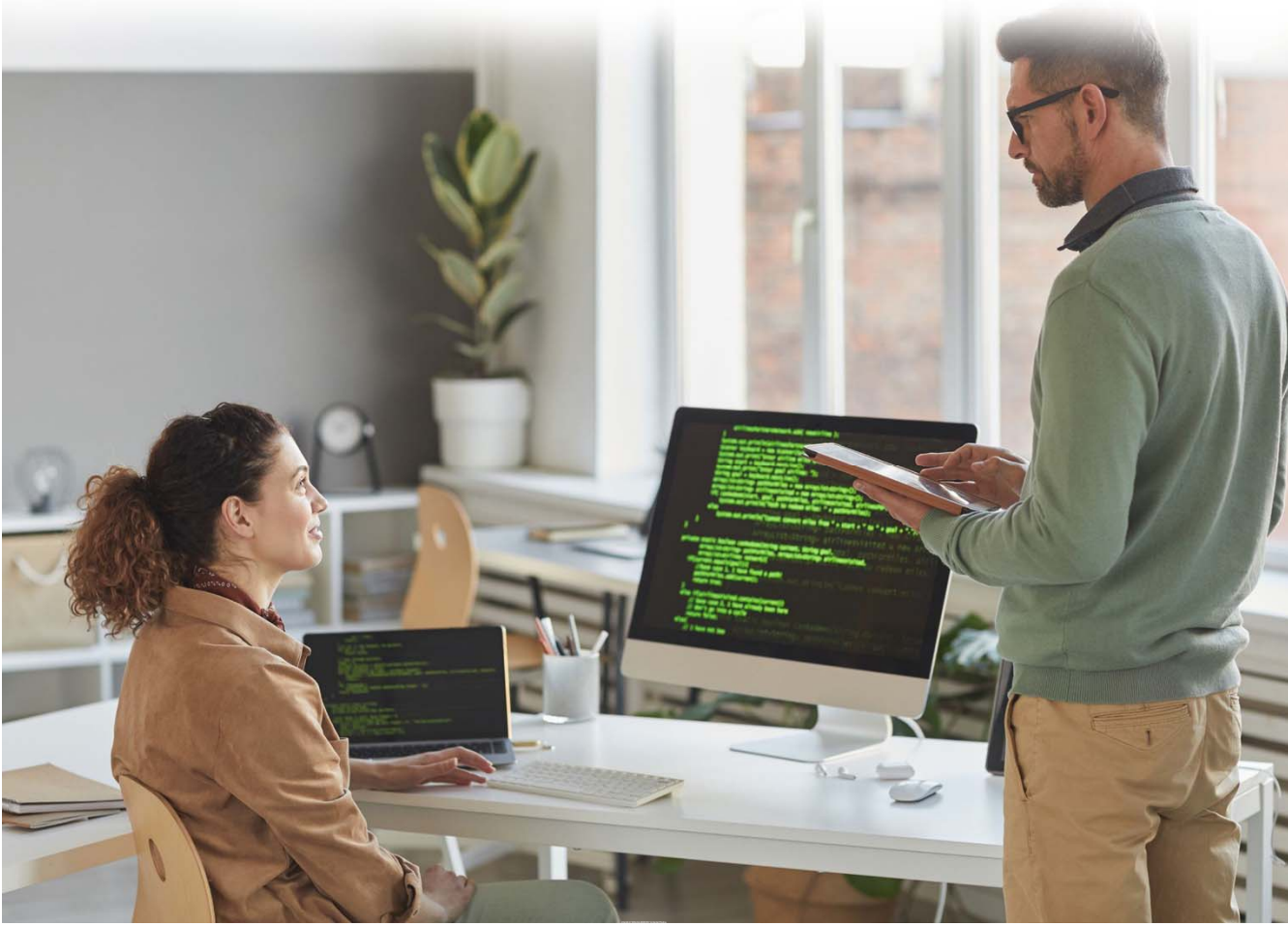
a) Zabezpiecza kod przed błędami i zapewnia alternatywną ścieżkę wykonania w przypadku wystąpienia wyjątku.

b) Zapewnia mechanizm do wykonania kodu wielokrotnie, dopóki nie zostanie osiągnięty określony warunek.

c) Umożliwia tworzenie sekcji kodu, które zostaną wykonane tylko wtedy, gdy żaden wyjątek nie zostanie zgłoszony.

ROZDZIAŁ 4

PROGRAMOWANIE OBIEKTOWE



4 Programowanie obiektowe

4.1 Klasy, metody, obiekty

Klasy są podstawowymi elementami programów w języku Java. Wszystko co tworzymy w Javie jest związane z **klasami** i **obiektami**. Jest to struktura, która definiuje obiekty nazywane **polami** i za ich zachowanie w programie odpowiadają **metody klasy**.

Klasa to szablon bądź wzorzec na podstawie którego tworzone są **obiekty**. Definiujemy ją w osobnym pliku, który nosi taką samą nazwę jak klasa. Jej nazwa zaczyna się zwykle wielką literą.

Specjalną **metodą** każdej **klasy** jest tzw. **konstruktor**. Jest on wywoływany podczas tworzenia **klasy**, a także definiuje sposób tworzenia **obiektów**.

Przejdziemy teraz do praktyki, czyli tworzenia **klas**. W Javie obiekty są tworzone na podstawie **klasy**.

Tworzymy nowy plik w o nazwie **Klasa**. Następnie definiujemy w nim klasę o nazwie **Klasa**, wewnątrz której, umieścimy jedną **zmienną**, a następnie wywołamy ją w metodzie głównej **main**.

Zatem:

```
class Klasa {
    int liczba = 10;

    public static void main(String[] args) {
        Klasa kl = new Klasa();
        System.out.println(kl.liczba);
    }
}
```

Listing 4.1 Utworzenie klasy zawierającej zmienną, a następnie jej obiektu

Na początku tworzymy klasę o nazwie **Klasa** poprzedzając wpisaną nazwę słowem **class**. Wewnątrz niej umieszczamy zmienną **liczba**, której nadajemy wartość liczbową **10**. Następnie pojawia się metoda **main**.

Dla przypomnienia – **main** jest metodą rozruchowa programu. Dzięki niej możemy wykonywać różne instrukcje. Aby ułatwić sobie zadanie i oszczędzić chwilę czasu – możesz ją stworzyć pisząc w programie **psvm**, a następnie wciskając **Enter**. Wtedy zostanie ona automatycznie dodana do kodu.

Obiekt klasy tworzymy wewnątrz metody **main.** Najpierw podajemy **nazwę klasy** dla której tworzymy **obiekt** (w tym przypadku **Klasa**), następnie nazwę naszego obiektu (tutaj **kl**). Później za pomocą słowa kluczowego new (ang. nowy) tworzymy nowy egzemplarz klasy o nazwie, którą podajemy na końcu kodu.

W ten sposób stworzyliśmy obiekt naszej klasy o nazwie **kl**.

W kolejnej linii wywołujemy **zmienną** znajdującą się w **klasie**. Aby tego dokonać, musimy najpierw podać nazwę **obiektu** odpowiedniej **klasy,** później symbol **kropki** „**.**” i **nazwę zmiennej** lub **metody**, która znajduje się w **klasie**. Całość wywołamy za pomocą metody **System.out.println()**.

Po uruchomieniu programu, w konsoli pojawi się liczba **10** – tyle ile wynosi wartość **zmiennej liczba**. W metodzie **main** możemy tworzyć nieskończenie wiele obiektów.

Na przykład:

```
class Klasa {
    int liczba = 10;
    String napis = "Nowy obiekt";

    public static void main(String[] args) {
       Klasa kl1 = new Klasa();
       Klasa kl2 = new Klasa();
       kl1.liczba = 50;
       kl2.napis = "Zmieniony obiekt";
       System.out.println(kl1.liczba);
       System.out.println(kl2.napis);
   }
}
```

Listing 4.2 Utworzenie dwóch obiektów klasy oraz wyświetlenie ich wartości w metodzie „main”

W klasie umieściliśmy dodatkowo nową zmienną typu **string** o nazwie **napis**, która przechowuje **tekst**. W metodzie main zmieniliśmy nazwę obiektu **kl** na **kl1** i dodaliśmy nowy obiekt o nazwie **kl2**. W ten sposób mamy dwa obiekty jednej klasy.

Metoda **System.out.println()** wypisze nam tekst przypisany do obiektu **kl2**. W konsoli ujrzymy następujący rezultat:

```
10
Nowy obiekt

Process finished with exit code 0
```

Rysunek 4.1 Wynik konsolowy poprzedniego programu

Możemy oczywiście zamienić ze sobą te obiekty, tak żeby to obiekt **kl1** wywołał zmienną **napis**, a obiekt **kl2** zmienną **liczba** lub wywołać obie **zmienne** za pomocą tylko jednego **obiektu**. Dzięki obiektom klas mamy możliwość zmiany wartości zmiennych.

Przykładowo:

```
class Klasa {
    int liczba = 10;
    String napis = "Nowy obiekt";

    public static void main(String[] args) {
       Klasa kl1 = new Klasa();
       Klasa kl2 = new Klasa();
       kl1.liczba = 50;
       kl2.napis = "Zmieniony obiekt";
       System.out.println(kl1.liczba);
       System.out.println(kl2.napis);
   }
}
```

Listing 4.3 Zmiana wartości zmiennych przez obiekt klasy

Do poprzedniego programu dodaliśmy dwie nowe linie. Przypisaliśmy w nich dwóm **zmiennym** nowe **wartości**.

Po uruchomieniu programu zostaną one wypisane w konsoli zamiast tych, zadeklarowanych w **klasie**. Takiej zmiany wartości nie będziemy mogli dokonać, w przypadku gdy **zmienna** będzie zmienną typu **final**.

4.1.1 Metoda klasy

O **metodach** już sporo się dowiedziałeś w trakcie czytania tej książki, ale przyszedł czas by poznać je nieco bliżej i bardziej szczegółowo. W języku Java, **metoda** to **blok** kodu, który wykonuje określoną operację lub obliczenia.

Metoda może być wywoływana z każdego miejsca programu, a wynik działania metody może być zwrócony do miejsca, z którego została wykonana.

Metoda może być używana do wykonywania określonych operacji wielokrotnie bez konieczności powtarzania kodu. Składa się zwykle z trzech elementów: **nazwy**, **listy parametrów** (opcjonalnie) i **ciała**, czyli **instrukcji**, które mają być wykonane podczas jej wywołania. Omówimy teraz kilka ważnych metod w Javie, z których na pewno będziesz korzystał w przyszłości.

Jako pierwsze będą to **metody niestatyczne** czyli metody, wywoływane na **obiekcie** danej **klasy**. Są one przydatne do wykonywania operacji na **stanie obiektu**. Przykładem najprostszej takiej metody, z którą każdy z nas się spotkał na początku swojej drogi z programowaniem to metoda zwracająca **Hello World**.

Oto kod tej metody:

```
class Klasa {
    public static void helloWorld(){
        System.out.println("Hello World");
    }

    public static void main(String[] args) {
        helloWorld();
    }
}
```

Listing 4.4 Przykładowa metoda „helloWorld”

W powyższym kodzie słowo **public** oznacza, że metoda jest dostępna z każdego miejsca w programie. **Static** zaś, że **metoda** należy do **klasy,** a nie do jej instancji. **Void** oznacza, że **metoda** nie zwraca żadnej **wartości**, a **helloWorld** to nazwa **metody**.

Metodę tę wywołujemy wewnątrz metody **main**, która jest punktem wejścia do programu. Gdy program zostanie uruchomiony, wyświetli się na ekranie napis **Hello World**. Metoda może być przywoływana nieskończoność razy.

Tak, jak wspomnieliśmy wyżej, **metoda** może przyjmować informacje jako **parametry** i **argumenty**. Parametry zachowują się jak zmienne wewnątrz metody. Są one określane po nazwie metody w **okrągłych nawiasach**.

Poniżej zamieszczony został przykład metody z parametrem:

```
class Klasa {
    static void dodawanieLiczb(int liczba){
        System.out.println(liczba + 5);
    }

    public static void main(String[] args) {
        dodawanieLiczb(3);
    }
}
```

Listing 4.5 Przykład metody z parametrem

W pierwszej kolejności utworzyliśmy nową zmienną statyczną o nazwie **dodawanieLiczb**, której parametrem jest zmienna **liczba** typu **int**. Następnie w metodzie **main** wywołujemy naszą metodę. Jako, że naszym parametrem jest zmienna typu **int**, to musimy w nawiasie podać **liczbę całkowitą** (w tym przypadku jest to **liczba 3**). Samo działanie metody jest bardzo proste - dodawana jest **liczba 5** do podanej przez nas liczby, przy wywoływaniu metody. W konsoli pojawi się **liczba 8**, która jest **sumą** podanych liczb.

Metoda może mieć więcej niż jeden **parametr** i to różnych typów.

Na przykład:

```
class Klasa {
static void ocenaPrzedmiot(int ocena, String przedmiot){
System.out.println("Uczeń otrzymał ocenę " + ocena + " z
przedmiotu" + przedmiot);
    }
public static void main(String[] args) {
        ocenaPrzedmiot(5,"informatyka");
    }
}
```

Listing 4.6 Przykład użycia metody z dwoma parametrami

Kolejny parametr typu **String** został oddzielony **przecinkiem** od poprzedniego. Każdy następny **parametr** również będzie zapisywany po **przecinku**.

Kolejnymi metodami, które zostaną omówione to **metody statyczne**. To metody, które są **wywoływane** bez tworzenia **obiektu** danej **klasy**. Są one przydatne do wykonywania operacji, które nie wymagają **stanu obiektu**.

Słowo kluczowe **void** w powyższych przykładach daje nam informację o tym, że metoda nie zwraca żadnej wartości.

Aby metoda zwracała nam jakąś wartość, musimy **void** zastąpić prymitywnym typem danych (tj. **int, char**, itd.). Dodatkowo w naszej metodzie trzeba umieścić słowo kluczowe **return**. Zatem:

```
class Klasa {
    static int dodawanieLiczb(int liczba){
        return liczba + 5;
    }

    public static void main(String[] args) {
        System.out.println(dodawanieLiczb(3));
    }
}
```

Listing 4.7 Przykład metody zwracającej wartość

Jest to ten sam program, który napisaliśmy poprzednio (metoda z jednym parametrem), lecz został on zaktualizowany tak, aby zwracał wartość. W programie słowo kluczowe **int** zastąpiło nam **void**, co oznacza, że nasza metoda zwraca wartość typu **int**.

Wewnątrz **metody** znajduje się słowo kluczowe **return**, która oznacza zwrócenie **wartości**. Po nim zapisujemy dokładnie to, co metoda ma zwracać. W metodzie **main** wywoływana jest nasza metoda. Aby w konsoli pojawił się wynik, to musi zostać ona zapisana w **System.out.println()**.

Teraz napiszemy program z metodą zwracająca wartość, która będzie miała dwa parametry. Poniżej znajduje się przykładowy program:

```
class Klasa {
    static int odejmowanieLiczb(int x, int y){
        return x-y;
    }

    public static void main(String[] args) {
        int wynik = odejmowanieLiczb(5,3);
        System.out.println(wynik);
    }
}
```

Listing 4.8 Przykład metody zwracającej wartość z dwoma parametrami

Nasza **metoda** będzie polegała na **odjęciu** od siebie dwóch **liczb całkowitych**. Konstrukcja metody jest taka sama jak w poprzednim programie, ale dodatkowo dodajemy jeszcze jeden parametr, oddzielony **przecinkiem** od poprzedniego. W metodzie **main** dodaliśmy zmienną **wynik** typu **int**, która będzie przechowywała wartość zwracaną przez **metodę**. Następnie w **System.out.println()** umieszczamy właśnie ten **wynik**. Jest to inny, bardziej uporządkowany sposób wypisania wartości w konsoli, ponieważ **wynik metody** jest przechowywany przez **zmienną**.

4.1.2 Przeciążenie metod

Przyjrzyjmy się teraz takiemu przykładowi:

```
class Klasa {
static int dodawanieLiczb(int x, int y, int z){
        return x+y+z;
    }
static int odejmowanieLiczb(int x, int y, int z){
        return x-y-z;
    }
public static void main(String[] args) {
int wynikDodawania = dodawanieLiczb(7,4,2);
int wynikOdejmowania = odejmowanieLiczb(5,3,1);
int wynikKońcowy = wynikDodawania-wynikOdejmowania;
        System.out.println(wynikKońcowy);
    }
}
```

Listing 4.9 Przykład metod zwracających wartość z trzema parametrami

W programie znajdują się dwie metody – **dodawanieLiczb** i **odejmowanieLiczb**. W pierwszej, zwróconą wartością będzie suma wszystkich trzech liczb, z kolei w drugiej będzie to ich różnica.

Wyniki wywołania tych metod zapisaliśmy do zmiennych - **wynikDodawania** oraz **wynikOdejmowania**. Następnie różnicę tych dwóch zmiennych przechowuje zmienna **wynikKońcowy**, i to on jest wypisywany przez **System.out.println()**.

Co w przypadku gdybyśmy chcieli dodać do siebie więcej lub mniej niż **trzy** liczby? A może wolelibyśmy żeby to były liczby typu **double**, a nie **int**?

W takiej sytuacji musielibyśmy stworzyć mnóstwo metod o różnych nazwach, tak aby wiedzieć, która odpowiada danym typom i ilościom. Z pewnością nie jest to najlepsze rozwiązanie. Z pomocą przychodzą nam przeciążone metody. **Przeciążanie metody** (ang. method overloading) to mechanizm w języku programowania Java, który umożliwia definiowanie wielu metod o tej samej nazwie, ale różniących się **typami** i/lub liczbą parametrów. Pozwala to na bardziej elastyczne wykorzystanie metody w zależności od potrzeb. Przykładowo, jeśli mamy klasę **Matematyka**, to możemy zdefiniować przeciążoną metodę dodawania, która będzie przyjmować różną liczbę argumentów, np.:

```
public class Matematyka {
    public int dodaj(int a, int b){
        return a + b;
    }
    public double dodaj(double a, double b){
        return a + b;
    }
    public int dodaj(int a, int b, int c){
        return a + b + c;
    }
}
```

Listing 4.10 Przykład przeciążenia metody „dodaj"

Powyżej w przykładzie mamy trzy metody o nazwie **dodaj**, ale różniące się **liczbą** i/lub **typami parametrów**. Pierwsza metoda przyjmuje dwa parametry typu **int**, druga – dwa parametry typu **double**, a trzecia – trzy parametry typu **int**. Przy wywoływaniu metody, kompilator Javy automatycznie dopasuje argumenty do odpowiedniej metody w zależności od ich typów i liczby. Na przykład, gdy wywołujemy metodę **dodaj** z dwoma argumentami typu **int**, zostanie wywołana **pierwsza metoda**, gdy z dwoma argumentami typu **double** - **druga metoda**, a gdy z **trzema argumentami** typu **int** – **trzecia metoda**.

Dzięki **przeciążaniu metody**, możemy uniknąć nadmiernego tworzenia metod o różnych nazwach, co zwiększa czytelność kodu i ułatwia jego utrzymanie. Poniżej znajduje się przykład użycia tych trzech metod **dodaj**:

```
public class Main {
   public static void main(String[] args) {
    Matematyka kalkulator = new Matematyka();

    int wynik1 = kalkulator.dodaj(3,4);
    double wynik2 = kalkulator.dodaj(4.2,5.3);
    int wynik3 = kalkulator.dodaj(2,4,7);

       System.out.println(wynik1);
       System.out.println(wynik2);
       System.out.println(wynik3);
   }
}
```

Listing 4.11 Przykład użycia przeciążonej metody „dodaj"

Wyniki metod dodaj przypisaliśmy kolejno do zmiennych **wynik1** (dla dodania dwóch zmiennych typu **int**), **wynik2** (dla dodania dwóch zmiennych typu **double**) oraz **wynik3** (dla dodania trzech zmiennych typu **int**).

System.out.println() wyświetli nam w konsoli trzy różne wyniki, odpowiadające poszczególnym metodom.

4.1.3 Metody dostępowe getter i setter

Kolejnymi metodami, które omówimy będą metody dostępowe (**gettery i settery**). Są to **metody**, które pozwalają na **odczytanie** i **zapisanie wartości** prywatnych pól (atrybutów) klasy. Dzięki temu unikamy bezpośredniego dostępu do pól klasy z poziomu innych klas, co zwiększa kontrolę nad danymi.

Getter zwraca wartość prywatnego pola, a **setter** ustawia wartość dla tego pola. Oto przykład:

```
class Klasa {
    private String imie;

    //Getter
    public String getImie(){
        return imie;
    }

    //Setter
    public void setImie(String noweImie){
        this.imie = noweImie;
    }
}
```

Listing 4.12 Przykład settera i gettera w klasie

Ten kod definiuje pole prywatne **klasy** o nazwie **imie**. Jest ono typu **String**. **Pole** jest prywatne, co oznacza, że nie jest bezpośrednio dostępne z poziomu innych klas.

Następnie **klasa** ta definiuje dwie metody publiczne: **getImie** oraz **setImie**. Są to tak zwane **gettery** i **settery**. Służą one do pobierania **get** i ustawiania **set** wartości pola **imie**.

Metoda **setImie** ustawia wartość pola **imie** na wartość przekazaną jako parametr **noweImie**. Jest to przykład metody typu **setter**, który umożliwia ustawienie wartości pola z zewnątrz.

Metoda jest również oznaczona modyfikatorem dostępu **public**, aby była dostępna z poziomu innych **klas**. W ciele **metody setImie** wykorzystano słowo kluczowe **this**, które odnosi się do bieżącego obiektu, czyli instancji klasy, w której ta metoda jest wywoływana. Wykorzystanie „**this**” wskazuje na to, że

chcemy ustawić wartość pola **imie** dla bieżącego obiektu, a nie dla innego. **Metoda getImie** zwraca wartość pola **imie**, co pozwala na odczytania aktualnej wartości tego pola z innych **klas**. **Metoda** ta jest również oznaczona modyfikatorem dostępu **public**.

Przykład przedstawia sposób użycia tych metod:

```
class Klasa {
    private String imie;

    //Getter
    public String getImie(){
        return imie;
    }
    //Setter
public void setImie(String noweImie){
        this.imie = noweImie;
    }
public static void main(String[] args) {
        Klasa osoba = new Klasa();
        osoba.setImie("Anna");
System.out.println("Imię osoby to: " + osoba.getImie());
    }
}
```

Listing 4.13 Przykład użycia settera i gettera

W tym przykładzie tworzymy nowy obiekt klasy i ustawiamy jego imię za pomocą metody **setImie**. Następnie używając metody **getImie**, pobieramy wartość pola **imie** i wyświetlamy ją w konsoli. W taki sposób uzyskaliśmy dostęp do prywatnej zmiennej **imie** oraz dokonaliśmy jej użycia.

Kolejną ważną metodą w Javie jest metoda **toString()**. Polega ona na tym, że zwraca reprezentację tekstową obiektu danej klasy. **Metoda** ta jest przydatna do wyświetlania informacji o **obiekcie**. Oto przykład:

```
class Klasa {
    private String imie;

    //Getter
    public String getImie(){
        return imie;
    }

    //Setter
    public void setImie(String noweImie){
        this.imie = noweImie;
    }

    @Override
    public String toString(){
        return "Osoba nazywa się " + imie;
    }
}
```

Listing 4.14 Przykład użycia metody „toString"

W tym programie **klasa** ma jedno pole prywatne **imie**, a jego **wartość** ustawiana jest za pomocą **metody setImie**. W metodzie **toString()** zwracany jest tekst reprezentujący obiekt klasy **Klasa**.

Utwórz teraz nową **klasę** o nazwie **Main** i zapisz w niej poniższy kod:

```
public class Main {
    public static void main(String[] args) {
        Klasa osoba = new Klasa();
        osoba.setImie("Arkadiusz");
        System.out.println(osoba.toString());
    }
}
```

Listing 4.15 Kod nowej klasy

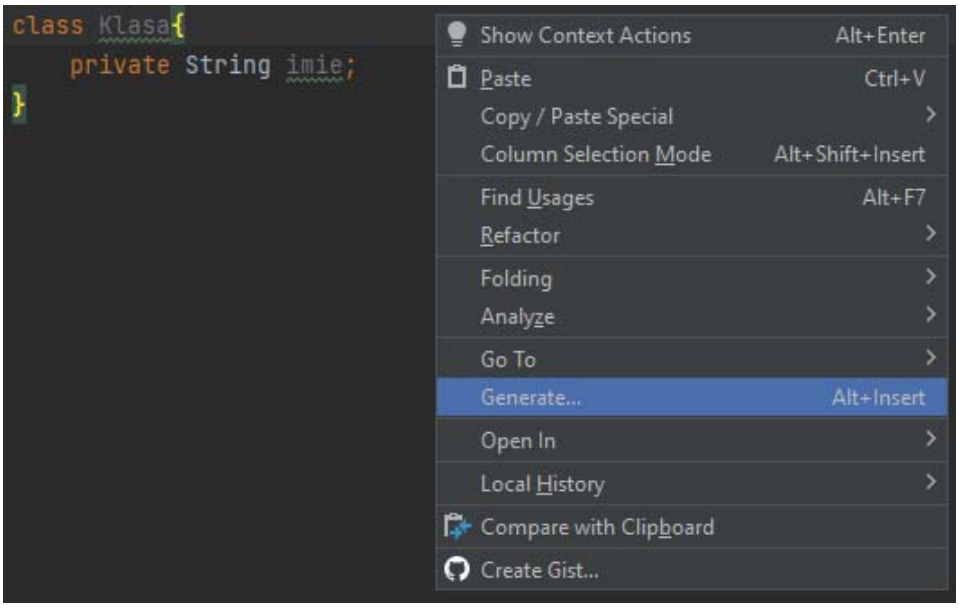
W metodzie „**main()**" tworzony jest nowy obiekt klasy **Klasa**. Imię tej osoby ustawiamy za pomocą **setImie()** i wyświetlamy wynik działania metody **toString()** dla tego obiektu. Konsola zwróci nam:

```
Osoba nazywa sie Arkadiusz

Process finished with exit code 0
```

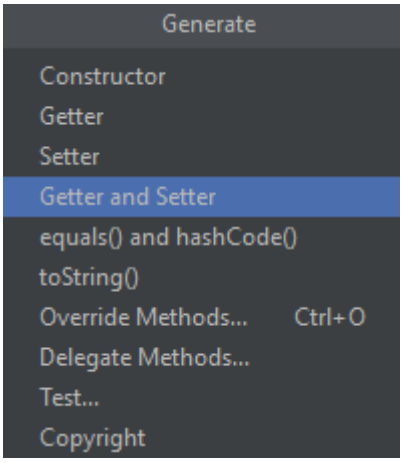
Rysunek 4.2 Wyświetlenie informacji o obiekcie przez metodę „toString"

Przydatną rzeczą w środowisku **IntelliJ** jest automatyczne generowanie powyższych metod tj. **getter**, **setter** i **toString().** Aby tego dokonać wystarczy, że na początku klikniemy prawym przyciskiem w dowolne miejsce naszego

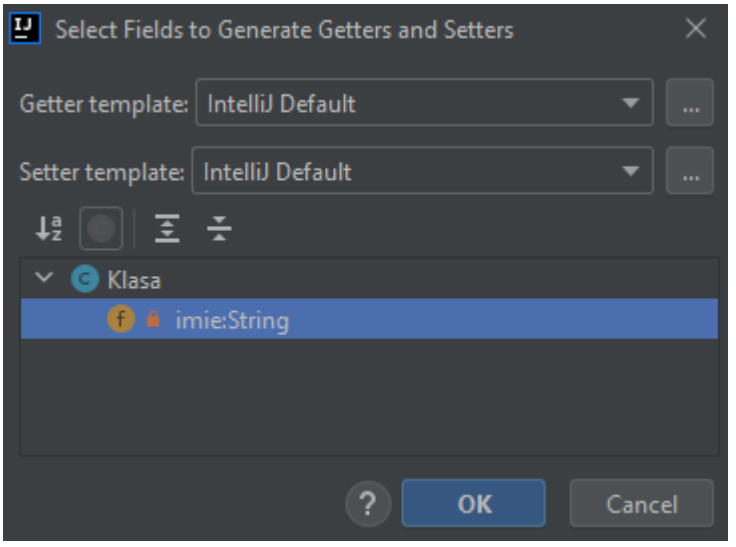
kodu, następnie wybierzemy **Generate...** a następnie jedną z opcji: **Getter**, **Setter**, **Getter and Setter**, **toString()**. Pojawi nam się okienko, w którym wybieramy zmienne będące elementami naszych wybranych metod. W tym przypadku jest to **imie**.



Rysunek 4.3 Kliknięcie prawym przyciskiem myszy na wolny obszar kodu, a następnie wybranie opcji „Generate...”



Rysunek 4.4 Wybranie interesujących nas metod – „Getter and Setter”, w celu wygenerowania ich w programie



Rysunek 4.5 Wybranie zmiennej, będącej elementem naszej wygenerowanej metody

```
class Klasa{
    private String imie;

    public String getImie() {
        return imie;
    }

    public void setImie(String imie) {
        this.imie = imie;
    }
}
```

Rysunek 4.6 Wygenerowane metody „Getter and Setter”

Jak widzisz, program automatycznie nam wygenerował zarówno metodę **Getter** jak i **Setter**. To samo możesz teraz zrobić z metodą **toString()**. Rezultat będzie następujący:

```
class Klasa {
    private String imie;

    //Getter
    public String getImie(){
        return imie;
    }

    //Setter
public void setImie(String noweImie){
        this.imie = noweImie;
    }

    @Override
 public String toString() {
return "Klasa{" + "imie='" + imie + '\'' + '}';
    }
}
```

Listing 4.16 Wygenerowana metoda „toString()"

4.2 Modyfikatory dostępu i enkapsulacja (hermetyzacja)

4.2.1 Modyfikatory dostępu

W języku Java posiadamy dwa rodzaje modyfikatorów. Są nimi **modyfikatory dostępu** oraz **modyfikatory innego typu**.

Tymi drugimi są na przykład **static** czy **abstract**, jednak w tym rozdziale skupimy się tylko na modyfikatorach dostępu.

Modyfikatory te definiują, jakie pola, metody lub klasy są dostępne w jakim zakresie. Poziomy dostępu tych pól możemy regulować, korzystając z odpowiednich modyfikatorów. W Javie mamy do czynienia z czterema różnymi modyfikatorami dostępu:

- **default (domyślny)** – jest domyślnym modyfikatorem dostępu (gdy nie zadeklarujemy żadnego z pozostałych). Zapewnia kontrolę dostępu w ramach określonego pakietu, chroniąc przed niechcianym dostępem z zewnątrz pakietu.
- **private (prywatny)** – ogranicza dostęp do pól i metod jedynie do klasy, w której są zadeklarowane. Chroni dane przed bezpośrednim dostępem z zewnątrz klasy, umożliwiając kontrolowany dostęp poprzez publiczne metody.
- **protected (chroniony)** – pozwala na dostęp do pól i metod w obrębie pakietu oraz przez klasy dziedziczące, zachowując kontrolę dostępu.

- **public (publiczny)** – umożliwia dostęp do pól i metod z dowolnego miejsca w programie.

Poniżej tabela prezentująca widoczność modyfikatorów dostępu w różnych kontekstach:

Modyfikator dostępu	wewnątrz tej samej klasy	w obrębie tego samego	poza pakietem tylko przez klasę	poza pakietem, z dowolnej klasy
Private	Tak	Nie	nie	nie
Default	Tak	Tak	nie	nie
Protected	Tak	Tak	tak	nie
Public	Tak	Tak	tak	tak

Tabela 4.1 Widoczność modyfikatorów dostępu w różnych kontekstach

Teraz przejdziemy do praktyki, czyli pokażemy zastosowanie każdego z powyższych modyfikatorów w kodzie. W pierwszej kolejności utwórzmy klasę **ZakresZmiennych**. Ważne, by w tym przykładzie usunąć słowo **public** sprzed nazwy klasy.

Nad klasą definiujemy pakiet zapisując **package mojPakiet**. Pakiety w Javie pomagają w organizacji kodu, umożliwiając grupowanie klas o podobnej funkcjonalności.

Na ten moment program powinien wyglądać w ten sposób:

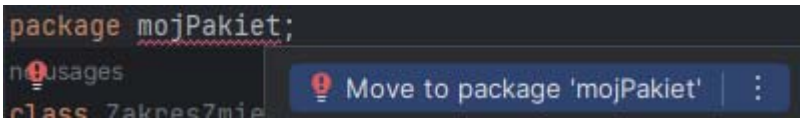
```
package mojPakiet;
class ZakresZmiennych {
}
```

Listing 4.17 Definicja pakietu i utworzenie klasy

Zapewne, nazwa „**mojPakiet**” jest podkreślona na czerwono. To zjawisko wynika z tego, że **klasa** powinna należeć do **pakietu**. Aby to naprawić, wystarczy kliknąć prawym przyciskiem myszy na nazwę „**mojPakiet**” (lub skorzystać z kombinacji **Alt+Enter**).

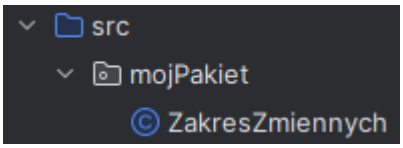
Rysunek 4.7 Pokazanie akcji kontekstowych pakietu

Pojawi się menu z różnymi opcjami do wyboru. My wybieramy **Show Context Actions**, który pokaże nam możliwe rozwiązania. Wybieramy opcję **Move to package 'mojPakiet'**, która przeniesie naszą klasę do pakietu.



Rysunek 4.8 Przeniesienie klasy do pakietu

Zauważamy, że po prawej stronie w katalogu **src** pojawił się nowy pakiet o nazwie, którą wcześniej podaliśmy, a do tego pakietu została przeniesiona nasza klasa **ZakresZmiennych**.

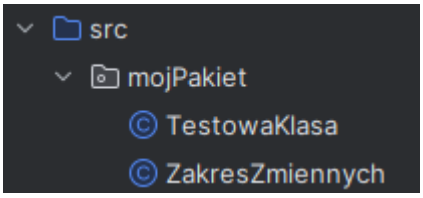


Rysunek 4.9 Pojawienie się nowego pakietu w katalogu „src"

Wewnątrz istniejącego pakietu tworzymy kolejną klasę. Będzie to klasa testowa, w której sprawdzimy dostępność wybranych pól. Klasę tworzymy w standardowy sposób, jednak tym razem klikamy prawym przyciskiem myszy na **mojPakiet**, a nie na **src**, jak to miało miejsce dotychczas. Następnie przechodzimy do opcji **New** i wybieramy **Java Class**.

Rysunek 4.10 Tworzenie nowej klasy w pakiecie

Jako nazwę podajemy **TestowaKlasa**. Teraz obie klasy znajdują się wewnątrz pakietu **mojPakiet**. Gdy wejdziemy w **TestowaKlasa**, zauważymy, że pakiet dla tej klasy utworzył się automatycznie.



Rysunek 4.11 Nowa klasa w pakiecie

Wróćmy jeszcze do klasy **ZakresZmiennych**. Utwórzmy w niej nową **zmienną statyczną liczba** typu **int**, która przechowuje wartość **7**. Powinno to wyglądać tak:

```
package mojPakiet;
class ZakresZmiennych {
    static int liczba = 7;
}
```

Listing 4.18 Klasa posiadająca zmienną statyczną

W tym przykładzie zmienna „**liczba**" jest zadeklarowana jako statyczna, co umożliwia dostęp do niej z innej klasy bez konieczności tworzenia obiektu. Ponieważ jeszcze nie mieliśmy styczności z programowaniem obiektowym, musimy to zrobić właśnie w ten sposób.

Teraz przechodzimy do **TestowaKlasa** i tworzymy w niej metodę **main**. W obrębie tej metody będziemy próbowali uzyskać dostęp do zmiennej „**liczba**" z klasy **ZakresZmiennych**. Wewnątrz metody „main" używamy polecenia **System.out.println** do wyświetlenia wartości zmiennej statycznej **liczba**. Aby w klasie **TestowaKlasa** uzyskać dostęp do tej zmiennej **liczba** z klasy **ZakresZmiennych** używamy składni **ZakresZmiennych.liczba**, co jest możliwe dzięki temu, że obie klasy są w tym samym pakiecie. Kod powinien wyglądać następująco:

```
package mojPakiet;
public class TestowaKlasa {
    public static void main(String[] args) {
        System.out.println("Wartość zmiennej: " +
ZakresZmiennych.liczba);
    }
}
```

Listing 4.19 Utworzenie metody „main" w klasie i wyświetlenie wartości zmiennej innej klasy

Po uruchomieniu programu, w konsoli powinien pojawić się napis „**Wartość zmiennej: 7**". Jeśli powrócimy do tabeli, w której przedstawione są zakresy widoczności modyfikatorów dostępu w różnych kontekstach, zauważymy, że zmienne o modyfikatorze **default** są dostępne w obrębie tego samego pakietu. Gdybyśmy teraz w klasie **ZakresZmiennych** dodali słowo **private** przed nazwą zmiennej statycznej, utracilibyśmy dostęp do niej w klasie **TestowaKlasa**, a program wyświetliłby błąd przy próbie uruchomienia go.

Skoro już mowa o modyfikatorze **private**, to wiemy już, że jest on dostępny jedynie wewnątrz tej samej klasy. Istnieje jednak możliwość korzystania z tych zmiennych w innych klasach, ale do tego celu konieczne jest użycie **getterów** i **setterów**. Te koncepcje programowania obiektowego zostaną dokładniej omówione w późniejszym etapie książki. Teraz utwórzmy nową klasę o nazwie

ZZ_private. Ważne jest, żeby nie umieszczać jej w pakiecie **mojPakiet**. Następnie zadeklarujmy nową zmienną prywatną typu int o nazwie **liczba**, a poniżej dodajmy metodę **main**. Wewnątrz niej spróbujmy wypisać wartość zmiennej prywatnej za pomocą **System.out.println()**. Cały kod powinien wyglądać w ten sposób:

```
public class ZZ_private {
    private int liczba = 6;

    public static void main(String[] args) {
        System.out.println(zmiennaPrywatna);
    }
}
```

Listing 4.20 Próba wypisania zmiennej prywatnej z innej klasy

Zmienna **liczba** będzie w kolorze czerwonym, ponieważ metoda **main** nie ma do niej dostępu. W zasadzie z zmienną prywatną nie możemy za dużo zrobić, ponieważ ma bardzo ograniczoną dostępność. Aby wyświetlić ją w metodzie **main** musimy dodać słowo **static** między słowem **private**, a **int** naszej zmiennej **liczba**. Wtedy będzie możliwe uruchomienie tego programu z wypisaniem **liczby 6**. Jak już wspomnieliśmy, bez **getterów** i **setterów** nie uzyskamy dostępu do zmiennych prywatnych w innych klasach.

Teraz, przechodzimy do modyfikatora dostępu **protected**. Modyfikator ten zajmuje ważne miejsce w kategoriach dostępności pól i metod, szczególnie w kontekście dziedziczenia. Ponownie, więcej o tym dowiemy się dopiero w rozdziałach dotyczących programowania obiektowego, jednak teraz pokażemy prosty przykład użycia **protected**.

Utwórzmy nowy pakiet o nazwie **mojPakiet2**, a w nim dwie klasy o nazwie **ZZ_protected** oraz **TestowaKlasa**. W pierwszej z nich deklarujemy nową statyczną zmienną chronioną typu int o nazwie **liczba**, przyjmującą **wartość 10**. W klasie **TestowaKlasa** zapisujemy metodę **main**, a wewnątrz niej spróbujmy wypisać wartość zmiennej **liczba**. Kod klasy **ZZ_protected** powinien wyglądać następująco:

```
package mojPakiet2;
public class ZZ_protected {
    protected static int liczba = 10;
}
```

Listing 4.21 Klasa ze zmienną chronioną

Z kolei poniżej znajduje się przykład kodu klasy **TestowaKlasa**:

```
package mojPakiet2;

public class TestowaKlasa {
    public static void main(String[] args) {
        System.out.println(ZZ_protected.liczba);
    }
}
```

Listing 4.22 Wyświetlenie zmiennej z innej klasy tego samego pakietu

W tym przypadku, klasa **TestowaKlasa** może bezpośrednio korzystać z zmiennej **protected** z klasy **ZZ_protected** bez wykorzystanie mechanizmu dziedziczenia. Modyfikator **protected** pozwala na dostęp do elementów klasy z poziomu innych klas, ale także występujących w tym samym pakiecie czy w innych pakietach.

Ostatnim z modyfikatorów dostępu jest **public.**

By zaprezentować jego działanie tworzymy kolejną klasę o nazwie **ZZ_public**, a w niej statyczną zmienną publiczną o nazwie **numer**, która przyjmuje wartość **15**. Przykładowy zapis tej klasy:

```
public class ZZ_public {
    public static int numer = 15;
}
```

Listing 4.23 Klasa z zmienną statyczną

Teraz tworzymy nową klasę o nazwie **TestowaPubliczna**. Klasa ta będzie próbować uzyskać dostęp do zmiennej **numer** z klasy **ZZ_public**. Deklarujemy w niej metodę **main**. Wewnątrz tej metody zapisujemy **System.out.println()** do wyświetlenia wartości zmiennej **numer**.

Poniżej znajduje się kod klasy **TestowaPubliczna**:

```
public class PublicznaTestowa {
    public static void main(String[] args) {
        System.out.println(ZZ_public.numer);
    }
}
```

Listing 4.24 Wypisanie publicznej zmiennej z innej klasy

Po uruchomieniu tego programu, w konsoli wypisze się nasz numer.

Modyfikator „public” jest najmniej chronionym modyfikatorem i tym samym najbardziej dostępnym z poziomu innych klas.

Zastosowanie odpowiednich modyfikatorów dostępu pomaga utrzymać bezpieczeństwo i integralność kodu. Ograniczając dostęp do zmiennych i metod

(np. przez private) można zapobiec przypadkowym zmianom. Istnieje tzw. zasada najmniejszej władzy **least privilege**, która sugeruje, aby przypisywać najmniejsze uprawnienia, które są potrzebne. Nie używajmy **public** tam, gdzie **protected** lub **default** są wystarczające.

Zakres widoczności zmiennych i modyfikatory dostępu są kluczowymi elementami zarządzania kodem w języku Java. Poprawne ich zastosowanie wpływa także na czytelność i utrzymanie projektu. Więcej o ich użyciu dowiemy się w przyszłych rozdziałach.

4.2.2 Enkapsulacja

Enkapsulacja to jeden z podstawowych konceptów programowania obiektowego, którego celem jest ukrycie szczegółów implementacyjnych obiektu i ograniczenie dostępu do jego pól i metod.

W Javie **enkapsulacja** jest realizowana poprzez definiowanie pól prywatnych **private** w klasie, co oznacza, że mogą być one modyfikowane tylko wewnątrz tej klasy. Aby dostać się do tych pól z zewnątrz, musimy skorzystać z metod publicznych **gettery** i **settery**, które pozwalają na odczytanie wartości pola lub ustawienie nowej wartości. Dzięki temu zabezpieczamy nasz kod przed niekontrolowanym dostępem i modyfikacją wartości pól, co prowadzi do poprawy bezpieczeństwa, łatwiejszej utrzymanie kodu oraz zapewnienia spójności danych. Przykład:

```
public class Osoba {
    private String imie;
    private String nazwisko;

    public Osoba(){
    }

    public Osoba(String imie, String nazwisko){
        this.imie = imie;
        this.nazwisko = nazwisko;
    }

    public String getImie(){
        return imie;
    }

    public void setImie(String imie){
        this.imie = imie;
    }
```

Listing 4.25 Przykład enkapsulacji

W powyższym przykładzie definiujemy klasę **Osoba**, która posiada pola prywatne **imie** oraz **nazwisko**. Aby uzyskać dostęp do tych pól z zewnątrz, definiujemy **gettery** i **settery**. Metody te są publiczne, co oznacza, że mogą być wywoływane z dowolnego miejsca w programie. W przypadku próby modyfikacji wartości pola bezpośrednio z zewnątrz, np.

```
public class Main {
    public static void main(String[] args) {
        Osoba mezczyzna = new Osoba();
        mezczyzna.imie = "Jan";
    }
}
```

Listing 4.26 Próba modyfikacji wartości pola bezpośrednio z zewnątrz

Napis **imie** podświetli nam się na czerwono, a przy próbie kompilacji - zostanie wygenerowany błąd, ponieważ pole jest prywatne i niedostępne spoza klasy **Osoba**. Dzięki enkapsulacji możemy mieć pewność, że nasz kod działa poprawnie i zapewnia bezpieczeństwo danych. Możliwość wygenerowania „**getterów**" i „**setterów**" w **IntelliJ** dodatkowo ułatwia nam korzystanie z enkapsulacji.

4.3 Konstruktory

W tym rozdziale bardziej szczegółowo omówimy obiekty, które są podstawowymi jednostkami programowania obiektowego w języku Java i stanowią instancję klas. Są to struktury danych, zawierające zmienne i metody, które można wywoływać w ramach tej instancji. Obiekt jest utworzony na podstawie klasy definiującej jego strukturę i zachowanie.

Można powiedzieć, że obiekt ma posiada cechy tj.:

- **stan** (jest reprezentowany przez cechy obiektu),
- **zachowanie** (jest reprezentowane dzięki metodom obiektu),
- **tożsamość** (nadaje obiektowi unikalną nazwę).

Dobrym przykładem na omówienie tego zagadnienia jest obiekt **Samochód**.

- **tożsamość** to nazwa samochodu
- **stan** lub też **cechy** to np. kolor samochodu, model lub rok produkcji
- **zachowaniem samochodu** może być np. jazda, hamowanie czy parkowanie.

Podczas tworzenia obiektu w Javie należy określić, którą klasę obiektu chcemy utworzyć. Aby utworzyć nowy obiekt, należy wywołać konstruktor klasy za pomocą słowa kluczowego **new**. Przykładowo, jeśli chcemy utworzyć obiekt klasy **Samochód**, możemy to zrobić w następujący sposób:

```
class Samochód {
    Samochód auto = new Samochód();
}
```

Listing 4.27 Utworzenie obiektu klasy Samochód

W tym przykładzie **auto** to zmienna, która odwołuje się do nowo utworzonego obiektu klasy **Samochód**. Wywołujemy konstruktor klasy **Samochód**.

Konstruktor to specjalna metoda, która jest wywoływana podczas tworzenia nowej instancji klasy. Służy do inicjalizacji zmiennych obiektu, ustawienia wartości początkowych i wykonania innych czynności, które są wymagane przy tworzeniu nowego obiektu. Konstruktor ma tę samą nazwę co klasa i nie zwraca żadnej wartości, ani nie posiada żadnego typu zwracanego. Konstruktor wywoływany jest za pomocą słowa kluczowego **new**, a jego nazwa musi być taka sama jak nazwa klasy.

Konstruktory mogą mieć różne modyfikatory dostępu (**public**, **private**, **protected**, **bez modyfikatora**). Oznacza to, że możemy zdecydować, czy konstruktor będzie dostępny dla innych klas i obiektów, czy też nie.

Nie zwracają one żadnej wartości, ale też nie są typu **void**. Ich zadaniem jest tylko tworzenie nowych obiektów i inicjalizacja ich pól.

Jeśli nie zdefiniujemy żadnego konstruktora w klasie, to automatycznie zostanie utworzony domyślny konstruktor bezargumentowy (pusty). Pusty konstruktor nie przyjmuje żadnych parametrów. Jego zadaniem jest inicjowanie pól obiektu wartościami domyślnymi.

Oto przykład pustego konstruktora w klasie **Osoba**:

```
public class Osoba {
    private String imie;
    private int wiek;

    public Osoba(){
    }
}
```

Listing 4.28 Przykład pustego konstruktora

Teraz natomiast utworzymy konstruktor domyślny, zawierający argumenty. Przykład zastosowania takiego konstruktora:

```
public class Osoba {
    private String imie;
    private int wiek;

    public Osoba(String imie, int wiek){
        this.imie = imie;
        this.wiek = wiek;
    }
}
```

Listing 4.29 Utworzenie konstruktora klasy

Powyżej tworzony jest konstruktor klasy **Osoba**, który przyjmuje dwa argumenty - **imie** i **wiek**. Wewnątrz konstruktora wartości te są przypisywane do odpowiednich pól za pomocą słowa kluczowego **this**. Teraz można utworzyć nowy obiekt klasy **Osoba** z wykorzystaniem konstruktora:

```
public class Main {
    public static void main(String[] args) {
        Osoba dorosly = new Osoba("Adam",40);
    }
}
```

Listing 4.30 Utworzenie obiektu klasy z wykorzystaniem konstruktora

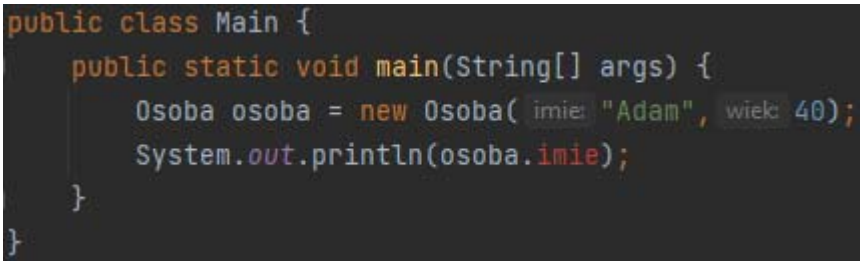
W powyższym przykładzie tworzony jest nowy obiekt klasy **Osoba** o imieniu **Adam** i **wieku 40**.

Konstruktor inicjalizuje wartości pól **imie** oraz **wiek**.

Konstruktory w środowisku **IntelliJ** mogą być generowane automatycznie tak jak metody omówione wcześniej tj.: **Getter**, **Setter** czy **toString()**.

Jeśli chcemy utworzyć więcej niż jedną zmienną, to przytrzymując klawisz **Ctrl** możemy zaznaczyć ich więcej.

Spróbujmy teraz wypisać imię naszej osoby. To, w jaki sposób to zrobimy zależne jest od tego, czy nasza zmienna jest publiczna. W naszym przypadku zmienna jest prywatna, więc samo napisanie nowej linii, w której miałoby wystąpić wypisanie zmiennej, niestety nie zadziała. **Imie** podświetli nam się na czerwono jak na poniższej ilustracji:



Rysunek 4.12 Błąd przy próbie wypisania imienia osoby

Jeśli jednak zmienisz **private** na **public** problem się rozwiąże. Prezentuje to poniższy przykład:

```
public class Osoba {
    public String imie;
    public int wiek;

    public Osoba(String imie, int wiek){
        this.imie = imie;
        this.wiek = wiek;
    }
}
```

Listing 4.31 Zmiana widoku zmiennej „private" na „public"

Wówczas będzie możliwe wypisanie w konsoli imienia naszej osoby. Jeśli zdecydujemy się na pozostanie przy zmiennych prywatnych, to będziemy musieli posłużyć się **Getterami** oraz **Setterami**.

Poprawmy zatem klasę **Osoba**:

```
public class Osoba {
    private String imie;
    private int wiek;

    public String getImie(){
        return imie;
    }
    public void setImie(String imie){
        this.imie = imie;
    }
    public int getWiek(){
        return wiek;
    }
    public void setWiek(int wiek){
        this.wiek = wiek;
    }
    public Osoba(String imie, int wiek){
        this.imie = imie;
        this.wiek = wiek;
    }
}
```

Listing 4.32 Dodanie metod „Getter" i „Setter" do klasy

Z kolei klasa **Main** będzie wyglądać następująco:

```
public class Main {
    public static void main(String[] args) {
        Osoba postac = new Osoba("Adam",40);
        System.out.println("Osoba nazywa się " +
osoba.getImie() + " i ma " + osoba.getWiek() + " lat");
    }
}
```

Listing 4.33 Wypisanie imienia oraz wieku osoby przez użycie metod

Mamy tutaj przykład enkapsulacji.

Aby klasa **Osoba** była bardziej przejrzysta, a zarazem program był bardziej użyteczny, wprowadzimy do niej nowe metody.

Pierwsza będzie pobierała i wypisywała imię osoby, druga jej nazwisko, a trzecia wiek.

Przejdźmy zatem do klasy **Osoba** i dopiszmy do kodu te metody i zmienną:

```
public class Osoba {
    String imie;
    String nazwisko;
    int wiek;

public Osoba(String imie, String nazwisko, int wiek){
        this.imie = imie;
        this.nazwisko = nazwisko;
        this.wiek = wiek;
    }

public void wypiszImie(){
 System.out.println("Osoba ma na imię " + this.imie);
    }
public void wypiszNazwisko(){
 System.out.println("Osoba ma na nazwisko " +
this.nazwisko);
    }
    public void wypiszWiek(){
 System.out.println("Osoba ma " + this.wiek + " lat");
    }
}
```

Listing 4.34 Dodanie nowych metod i zmiennej

Pojawiły się trzy nowe metody **wypiszImie**, **wypiszNazwisko** oraz **wypiszWiek**. Program został też nieco zmodyfikowany, ponieważ nasze zmienne nie są już typu prywatnego, toteż usunięte zostały **Gettery** i **Settery**. Teraz przejdziemy do klasy **Main**. W niej dodamy dwa nowe obiekty **osoba** i użyjemy naszych metod. Program będzie wyglądał następująco:

```
public class Main {
    public static void main(String[] args) {
Osoba czlowiek1 = new Osoba("Adam","Nowak",40);
Osoba czlowiek2 = new Osoba("Alicja","Kowalska",30);
Osoba czlowiek3 = new Osoba("Janusz","Wiśniewski",20);
        czlowiek1.wypiszImie();
        czlowiek2.wypiszNazwisko();
        czlowiek3.wypiszWiek();
    }
}
```

Listing 4.35 Dodanie dwóch nowych obiektów i użycie metod innej klasy

Po uruchomieniu programu, konsola zwróci nam **imię pierwszej osoby**, czyli **Adam**, nazwisko drugiej osoby – **Kowalska** oraz wiek trzeciej osoby – **20**.

Wyobraźmy sobie teraz, że jest pewna osoba, której znane jest imię oraz nazwisko, ale niestety nie wiemy ile ma lat. Chcemy ją dodać do naszego programu jako nowy obiekt **osoba**.

W obecnej sytuacji nie jest możliwe stworzenie obiektu osoba bez podania jej wieku. W takim przypadku możemy posłużyć się przeciążonymi konstruktorami. Przeciążanie poznałeś już przy metodach. Przeciążanie konstruktora polega na zdefiniowaniu w klasie więcej niż jednego konstruktora, ale z różnymi parametrami wejściowymi. Dzięki temu, w zależności od potrzeb, możemy tworzyć obiekty różnymi sposobami, używając odpowiednich konstruktorów. Oto przykład przeciążenia konstruktora w klasie **Osoba**.

```
public class Osoba {
    String imie;
    String nazwisko;
    int wiek;

    public Osoba(){
    }

    public Osoba(String imie){
        this.imie = imie;
    }

    public Osoba(String imie, String nazwisko){
        this.imie = imie;
        this.nazwisko = nazwisko;
    }

public Osoba(String imie, String nazwisko, int wiek){
        this.imie = imie;
        this.nazwisko = nazwisko;
```

Listing 4.36 Przykład przeciążenia konstruktora

W klasie **Osoba** napisaliśmy aż cztery różne **konstruktory.** Pierwszy jest pusty, drugi zawiera tylko jeden argument – **imie**, trzeci posiada dwa argumenty – **imie** oraz **nazwisko**, z kolei czwarty ma trzy argumenty - **imie**, **nazwisko** i **wiek**. Dzięki nim, mamy teraz możliwość utworzenia obiektów na cztery różne sposoby.

Przejdźmy zatem do klasy **Main** i utwórzmy w niej kilka obiektów.

```
public class Main {
    public static void main(String[] args) {
        Osoba osoba1 = new Osoba();
        Osoba osoba2 = new Osoba("Maciej");
        Osoba osoba3 = new Osoba("Jan","Mazur");
        Osoba osoba4 = new Osoba("Janina","Wójcik",25);
    }
}
```

Listing 4.37 Przykład utworzenia obiektów różnych konstruktorów

Pierwszy obiekt to **osoba1**, która nie posiada żadnych argumentów. Druga osoba ma podane tylko **imię**. Trzecia – **imię** oraz **nazwisko**, a czwarta – **imię**, **nazwisko** i **wiek**.

W programie możemy stworzyć dowolną ilość konstruktorów. Przeciążanie konstruktora jest przydatne w sytuacji, gdy chcemy tworzyć obiekty z różnymi zestawami właściwości. Dzięki temu, nasza klasa jest bardziej elastyczna i łatwiej dostosowuje się do różnych wymagań.

4.4 Sprawdź się!

1) Które z poniższych stwierdzeń dotyczących klas i obiektów w języku Java jest prawdziwe?

a) Klasa to obiekt, który można utworzyć na podstawie innych obiektów.

b) Obiekt to instancja klasy, która reprezentuje konkretne zasoby i zachowania.

c) Metoda to instancja obiektu, która definiuje jego właściwości i zachowania.

d) Klasa to zbiór metod, które można wywoływać w innych klasach.

2) Które z poniższych metod jest konstruktorem w języku Java?

a) void calculate(int a, int b)

b) int multiply(int a, int b)

c) Calculator()

d) String toString()

3) W języku Java, co oznacza skrót OOP?

a) Order of Operations in Programming

b) Object-Oriented Programming

c) Optimal Object Procedures

4) W języku Java, co to jest "enkapsulacja"?

a) Proces, w którym jedna klasa może dziedziczyć po innej.

b) Mechanizm, który umożliwia ukrycie implementacji szczegółów wewnętrznych klasy.

c) Specjalna metoda, która inicjalizuje stan obiektu.

4.5 Dziedziczenie i polimorfizm

4.5.1 Dziedziczenie

Dziedziczenie to jeden z najważniejszych konceptów programowania obiektowego, umożliwiający tworzenie hierarchii klas, gdzie klasy pochodne dziedziczą zachowanie swoich klas bazowych. W Javie dziedziczenie umożliwia tworzenie nowych klas na podstawie już istniejących i dodanie do nich nowych cech.

Dziedziczenie odbywa się poprzez utworzenie klasy pochodnej, która dziedziczy pola i metody klasy bazowej. Klasa pochodna może także definiować nowe pola i metody, które nie były dostępne w klasie bazowej.

Dziedziczenie w Javie jest zrealizowane poprzez słowo kluczowe **extends**. Jest także przykładem relacji **IS-A**, czyli tak zwanej relacji rodzic-dziecko.

Oto prosty przykład:

Mamy klasę **Zwierze** i chcemy dla niej stworzyć klasę potomną **Kot**. Klasa **Kot** będzie miała dodatkowe metody i pola specyficzne tylko dla kotów. Poniżej implementacja wymienionych klas:

```
public class Zwierze {
    private String imie;
    private int wiek;

    public Zwierze(){
    }

    public Zwierze(String imie, int wiek){
        this.imie = imie;
        this.wiek = wiek;
    }

    public void dajGlos(){
 System.out.println("Zwierzę wydaje dźwięk!");
    }
}
```

Listing 4.38 Klasa „Zwierze”

```
public class Kot extends Zwierze {
    private String rasa;

    public Kot(){
    }

public Kot(String imie, int wiek, String rasa){
        super(imie, wiek);
        this.rasa = rasa;
    }

    public void mrucz(){
        System.out.println("Kot mruczy!");
    }
}
```

Listing 4.39 Klasa „Kot” dziedzicząca po klasie „Zwierze”

W powyższym przykładzie mamy dwie klasy: **Zwierze** i **Kot**. Klasa **Zwierze** jest klasą bazową, a klasa **Kot** dziedziczy po niej, czyli jest klasą pochodną.

Klasa **Zwierze** posiada dwa prywatne pola: **imie** i **wiek**, oraz publiczną metodę **dajGlos()**. Konstruktor tej klasy przyjmuje dwa argumenty: **imię i wiek zwierzęcia** i przypisuje je do odpowiednich pól. Metoda **dajGlos()** wyświetla w konsoli napis **Zwierzę wydaje dźwięk!**.

Klasa **Kot** dziedziczy po klasie „**Zwierze**”, czyli posiada te same pola i metody, jakie posiada klasa **Zwierze**. Ponadto klasa **Kot** ma dodatkowe pole: **rasa**.

Konstruktor klasy **Kot** przyjmuje trzy argumenty: imię, wiek i rasę kota.

Przywołanie konstruktora klasy bazowej odbywa się poprzez podanie słowa kluczowego **super**.

Klasa **Kot** posiada również dodatkową metodę **mrucz()**, która wyświetla w konsoli napis **Kot mruczy!**.

Dzięki dziedziczeniu, klasa **Kot** dziedziczy po klasie **Zwierze**, co oznacza, że ma dostęp do pól i metod klasy **Zwierze**. W wyniku tego klasa **Kot** nie musi definiować tych samych pól i metod, co klasa **Zwierze**, tylko może korzystać z tych, które już zostały zdefiniowane w klasie bazowej. Ponadto, klasa **Kot** może dodawać swoje własne pola i metody, co pozwala na tworzenie bardziej szczegółowych i rozbudowanych klas.

Jeśli chodzi o dziedziczenie to wyróżniamy w Javie trzy podstawowe typy dziedziczenia:

- **pojedyncze** (klasa b dziedziczy po klasie a),
- **wielokrotne** (klasa c dziedziczy po klasie b, a klasa b dziedziczy po klasie a),
- **hierarchiczne** (zarówno klasa c, jak i klasa b dziedziczą po klasie a)

Poniżej proste przykłady każdego z tych typów dziedziczenia:

```
public class A {
    void metodaA(){
System.out.println("To jest metoda klasy A");
    }
}
public class B extends A{
    void metodaB(){
System.out.println("To jest metoda klasy B");
    }
}
public class Main {
    public static void main(String[] args) {
        B klasaB = new B();
        klasaB.metodaA();
        klasaB.metodaB();
    }
}
```

Listing 4.40 Przykład dziedziczenia pojedynczego

```
public class A {
    void metodaA(){
System.out.println("To jest metoda klasy A");
    }
}
public class B extends A{
    void metodaB(){
System.out.println("To jest metoda klasy B");
    }
}
public class C extends B{
    void metodaC(){
System.out.println("To jest metoda klasy C");
    }
}
public class Main {
    public static void main(String[] args) {
        C klasaC = new C();
        klasaC.metodaA();
        klasaC.metodaB();
        klasaC.metodaC();
    }
}
```

Listing 4.41 Przykład dziedziczenia wielokrotnego

```
public class A {
    void metodaA(){
System.out.println("To jest metoda klasy A");
    }
}
public class B extends A{
    void metodaB(){
System.out.println("To jest metoda klasy B");
    }
}
public class C extends A{
    void metodaC(){
System.out.println("To jest metoda klasy C");
    }
}
public class Main {
    public static void main(String[] args) {
        B klasaB = new B();
        C klasaC = new C();
        klasaB.metodaA();
        klasaB.metodaB();
        klasaC.metodaA();
        klasaB.metodaC();
    }
}
```

Listing 4.42 Przykład dziedziczenia hierarchicznego

W każdym przykładzie, w klasie **Main** znajdują się dostępne dla danego obiektu metody. To znaczy, że np. w ostatnim przykładzie obiekt **klasaC** nie będzie

mógł użyć metody **metodaB**, ponieważ nie dziedziczy on po klasie **B**, tym samym – nie ma do niej dostępu.

Pewnie zastanawiasz się dlaczego w Javie nie jest możliwe wielokrotne dziedziczenie, gdzie np. klasa **C** mogłaby dziedziczyć jednocześnie po klasie **A** oraz **B**. Załóżmy, że klasa **C** dziedziczy właśnie po tych dwóch klasach, a w klasie **A** oraz klasie **B** znajduje się metoda o identycznej nazwie. W przypadku, gdy chciałbyś wywołać taką metodę, program miałby problem z wybraniem metody z klasy **A** lub klasy **B**, ponieważ obie mają te samą nazwę. Wtedy wywołanie będzie niejednoznaczne, dlatego w Javie możemy dziedziczyć tylko po jednej klasie.

Wcześniej w jednym z przykładów pojawiło się słowo kluczowe **@Override** napisane żółtym kolorem. Jako, że poznałeś już dziedziczenie, możemy się bliżej przyjrzeć nadpisywanym metodom, bo za to właśnie to odpowiada.

Nadpisywanie metod w Javie, zwane również przesłanianiem metod, to proces zastępowania implementacji metody dziedziczonej z klasy nadrzędnej przez nową implementację w klasie pochodnej. Aby nadpisać metodę, należy utworzyć w klasie pochodnej metodę o takiej samej nazwie, z takimi samymi parametrami i zwracanym typem co metoda dziedziczona. Przykładem może być klasa **Zwierze** z metodą **dajGlos()**:

```
public class Zwierze {
   public void dajGlos(){
System.out.println("Zwierzę wydaje dźwięk!");
   }
}
```

Listing 4.43 „Zwierze" z metoda „dajGlos"

Teraz załóżmy, że mamy klasę **Kot**, która dziedziczy po klasie **Zwierze** i chcemy zmienić dźwięk, który wydaje kot. W tym celu możemy nadpisać metodę **dajGlos()** w klasie **Kot**:

```
public class Kot extends Zwierze {
    @Override
    public void dajGlos(){
        System.out.println("Miau");
}
```

Listing 4.44 Nadpisanie metody „dajGlos" w klasie „Kot" dziedziczącej po klasie „Zwierze"

W tym przypadku, gdy wywołamy metodę **dajGlos()** na obiekcie klasy **Kot**, zostanie wywołana nadpisana wersja metody, a nie ta z klasy nadrzędnej. To znaczy, że wynikiem w konsoli będzie napis **Miau**.

Dzięki temu mechanizmowi możemy zmienić zachowanie dziedziczonej metody i dostosować ją do potrzeb klasy pochodnej.

Dłuższym i bardziej praktycznym przykładem będzie klasa **FiguraGeometryczna**, w której znajdziemy metodę **obliczPole()**. W związku z tym, że każda z figur ma swój własny wzór na obliczenie pola nie możemy zastosować uniwersalnej metody.

Zapiszemy zatem klasę **FiguraGeometryczna**:

```
public class FiguraGeometryczna {
    private int bok;
    public FiguraGeometryczna(){
    }
    public FiguraGeometryczna(int bok){
        this.bok = bok;
    }
    public int obliczPole(){
        return bok;
    }
}
```

Listing 4.45 Klasa „FiguraGeometryczna”

Klasa ta posiada prywatną zmienną **bok** typu int, dwa konstruktory oraz metodę **obliczPole()**. Jak widzimy, metoda **obliczPole** zwraca nam tylko wartość boku. W klasach pochodnych, takich jak **Kwadrat** czy **Trójkąt**, musimy nadpisać metodę **obliczPole()** i dostosować ją do danej figury.

Przykładowo, dla klasy **Kwadrat**, metoda **obliczPole()** będzie miała następującą implementację:

```
public class Kwadrat extends FiguraGeometryczna {
    private int bok;

    public Kwadrat(int bok){
        this.bok = bok;
    }

    @Override
    public int obliczPole(){
        return bok * bok;
    }
}
```

Listing 4.46 Klasa „Kwadrat” i nadpisanie metody „obliczPole” klasy bazowej

Dla kwadratu, metoda **obliczPole()** zwraca już odpowiednie działanie czyli: bok razy bok. Z kolei klasa **Trojkat** będzie się prezentować w ten sposób:

```
public class Trojkat extends FiguraGeometryczna {
    private int bok;
    private int wysokosc;

    public Trojkat(int bok, int wysokosc){
        this.bok = bok;
        this.wysokosc = wysokosc;
    }

    @Override
    public int obliczPole(){
        return (bok * wysokosc) / 2;
    }
}
```

Listing 4.47 Klasa „Trojkat" i nadpisanie metody „obliczPole()" klasy bazowej

Jeśli chodzi o pole trójkąta to wyliczane jest ono ze wzoru: **½*a*h**. Teraz przejdziemy do utworzenia obiektów danych figur w klasie **Main** i wywołania metod poszczególnych klas. Poniżej znajduje się przykładowy kod:

```
public class Main {
    public static void main(String[] args) {
  FiguraGeometryczna figura = new FiguraGeometryczna(2);
 Kwadrat kwadrat = new Kwadrat(5);
 Trojkat trojkat = new Trojkat(3,4);
 System.out.println("Pole figury wynosi: " +
figura.obliczPole());
 System.out.println("Pole kwadratu wynosi: " +
figura.obliczPole());
 System.out.println("Pole trójkąta wynosi: " +
figura.obliczPole());
    }
}
```

Listing 4.48 Klasa „Main" wraz z obiektami wcześniej zapisanych klas oraz ich metodami

Wynik działania wyświetlony w konsoli po uruchomieniu programu:

```
Pole figury wynosi: 2
Pole kwadratu wynosi: 25
Pole trojkatu wynosi: 6

Process finished with exit code 0
```

Rysunek 4.13 Wypisanie pól różnych figur przez użycie metody „obliczPole"

W przypadku obiektu „figura” jest wywołana oczywiście metoda klasy „FiguraGeometryczna”. Analogicznie zachodzi to też w przypadku obiektu „kwadrat” i „trojkat”.

4.5.2 Polimorfizm

Słowo polimorfizm pochodzi od dwóch greckich słów - **poly** i **morphs**, gdzie **poly** oznacza wiele, a **morphs** znaczy forma.

Tak więc polimorfizm oznacza wiele form. Jest to jedna z ważnych cech programowania obiektowego, która pozwala na tworzenie hierarchii klas i wykorzystywanie wielu typów obiektów jako jednego rodzaju. W języku Java polimorfizm odnosi się do możliwości przypisywania referencji jednego typu do obiektu innego typu, pod warunkiem że oba typy są ze sobą powiązane w hierarchii dziedziczenia.

Polimorfizm umożliwia pisanie kodu, który może działać na wielu typach obiektów, bez potrzeby pisania osobnego kodu dla każdego z tych typów. Dzięki temu kod staje się bardziej elastyczny i łatwiejszy w modyfikacji. Przykład polimorfizmu w kontekście klas abstrakcyjnych i dziedziczenia może wyglądać w ten sposób:

Mamy dwie klasy abstrakcyjne: **Zwierze** i **Ptak**. Klasa **Zwierze** ma posiada metodę abstrakcyjną **dajGlos()**, a klasa **Ptak** metodę abstrakcyjną **lataj()**. Oba typy zwierząt mają różne implementacje tych metod. Może to być zapisane w ten sposób:

```
abstract class Zwierze {
    public abstract void dajGlos();
}
```

Listing 4.49 Klasa abstrakcyjna Zwierze posiadająca metodę „dajGlos()”

```
abstract class Ptak {
    public abstract void lataj();
}
```

Listing 4.50 Klasa abstrakcyjna Ptak z metodą „lataj()”

Następnie tworzymy dwie klasy dziedziczące po klasie **Zwierze**: **Kot** i **Pies**. Oba te zwierzęta mają różne sposoby wydawania dźwięków. Zatem:

```
public class Kot extends Zwierze {
    @Override
    public void dajGlos(){
        System.out.println("Miau!");
    }
}
```

Listing 4.51 Klasa „Kot" posiadająca metodę „dajGlos()", dziedzicząca po klasie „Zwierze"

```
public class Kot extends Zwierze {
    @Override
    public void dajGlos(){
        System.out.println("Miau!");
    }
}
```

Listing 4.52 Klasa „Pies" posiadająca metodę „dajGlos()", dziedzicząca po klasie „Zwierze"

Dodatkowo, tworzymy klasę dziedziczącą po klasie **Ptak** - **Sojka**:

```
public class Sojka extends Ptak {
    @Override
    public void lataj(){
        System.out.println("Sojka fruwa!");
    }
}
```

Listing 4.53 Klasa „Sojka" posiadająca metodę „lataj()", dziedzicząca po klasie „Ptak"

Sojka ma swoją własną implementację metody **lataj()**. Teraz możemy stworzyć tablicę obiektów typu **Zwierzę** i **Ptak**, a następnie iterować po niej i wywoływać z pomocą metody **dajGlos()** lub **lataj()**. Dzięki polimorfizmowi, możemy wywołać te metody na różnych typach obiektów, a metody będą wykonywać odpowiednie implementacje dla każdego typu. Oto kod przykładowej implementacji:

```
public class Main {
    public static void main(String[] args) {
        Zwierze[] zwierzeta = new Zwierze[2];
        zwierzeta[0] = new Kot();
        zwierzeta[1] = new Pies();

        Ptak ptak = new Sojka();
        for(Zwierze zwierze : zwierzeta){
            zwierze.dajGlos();
        }
        ptak.lataj();
    }
}
```

Listing 4.54 Klasa „Main" z trzema obiektami, wywołującymi określone metody

Jak widać, w pętli for wywoływana jest metoda **dajGlos()** na obiektach typu **Kot** i **Pies**, które dziedziczą po klasie **Zwierze**. Dzięki polimorfizmowi, metoda ta wywołuje odpowiednie implementacje dla każdego typu zwierzęcia. Następnie wywoływana jest metoda **lataj()** na obiekcie typu **Sojka**, który dziedziczy po klasie **Ptak**. Ta metoda również wywołuje odpowiednią implementację dla danego typu obiektu. W wyniku działania programu powinno pojawić się:

```
Miau
Hau
Sojka fruwa

Process finished with exit code 0
```

Rysunek 4.14 Wynik działania powyższego programu

4.6 Abstrakcja i interfejsy

4.6.1 Abstrakcja

Skoro znasz już pojęcia dziedziczenie i nadpisywanie metod to czas poznać klasy abstrakcyjne.

Klasy abstrakcyjne to klasy, które nie mogą być bezpośrednio tworzone jako obiekty, ale służą do definiowania interfejsu dla grupy klas pochodnych.

Klasy abstrakcyjne definiują zachowanie, ale pozostawiają implementację szczegółów klasom pochodnym.

W Javie klasę abstrakcyjną oznacza się słowem kluczowym **abstract**. Klasy abstrakcyjne są przydatne w sytuacjach, gdy mamy grupę klas, które są w jakimś sensie powiązane i mają pewne wspólne zachowanie, ale jednocześnie wymagają różnych szczegółowych implementacji.

Przykładem takiej klasy może być klasa **Pojazd**, która ma pewne cechy wspólne dla wszystkich pojazdów, ale jednocześnie wymaga różnych implementacji w zależności od rodzaju pojazdu.

Przykładowa klasa abstrakcyjna w Javie może wyglądać następująco:

```
public abstract class Pojazd {
    private String nazwa;

    public Pojazd(String nazwa){
        this.nazwa = nazwa;
    }
    public String getNazwa(){
        return nazwa;
    }
    public abstract void jedz();
}
```

Listing 4.55 Przykład klasy abstrakcyjnej

W powyższym przykładzie klasa **Pojazd** ma konstruktor, który przyjmuje odpowiednią nazwę, a także prywatne pole **nazwa**, które jest dostępne za pomocą publicznego **gettera**.

Klasa **Pojazd** zawiera również metodę abstrakcyjną **jedz()**, która nie ma implementacji. Każda klasa pochodna klasy **Pojazd** musi zaimplementować tę metodę zgodnie z wymaganiami konkretnej klasy.

Przykładem klasy pochodnej klasy abstrakcyjnej **Pojazd** może być klasa **Samochód**:

```
public class Samochod extends Pojazd {
    private String marka;

public Samochod(String nazwa, String marka){
        super(nazwa);
        this.marka = marka;
    }

    @Override
public void jedz(){
        System.out.println("Samochód jedzie!");
    }
}
```

Listing 4.56 Klasa pochodna klasy abstrakcyjnej „Pojazd”

Klasa „**Samochód**” dziedziczy po klasie abstrakcyjnej **Pojazd** i implementuje metodę abstrakcyjną **jedz()**. Dodatkowo klasa **Samochod** posiada pole **marka**, które nie jest dostępne w klasie **Pojazd**. Teraz utworzymy kolejną klasę pochodną, działającą podobnie do klasy **Samochód**. Będzie to klasa **Samolot**:

```
public class Samolot extends Pojazd{
    private String marka;

    public Samolot(String nazwa, String marka){
        super(nazwa);
        this.marka = marka;
    }

    @Override
    public void jedz(){
        System.out.println("Samolot leci!");
    }
}
```

Listing 4.57 Inna klasa pochodna klasy abstrakcyjnej „Pojazd"

Klasa **Samolot** tak samo jak klasa **Samochód** posiada prywatną zmienną **marka**. Różni się w zasadzie tylko wnętrzem metody **jedz()**.

Dzięki użyciu klas abstrakcyjnych i dziedziczenia możemy tworzyć hierarchie klas, które mają pewne cechy wspólne i jednocześnie posiadają szczegółowe implementacje zgodne z wymaganiami konkretnych zastosowań.

Klasy abstrakcyjne pozwalają na zdefiniowanie podstawowych cech i funkcjonalności dla grupy klas, jednocześnie wymuszając na nich dostosowanie się do tych wymagań.

4.6.2 Interfejsy

Oprócz klas abstrakcyjnych w Javie, mamy też interfejsy. Interfejsy to kolejny sposób na definiowanie **kontraktu** (ang. contract) pomiędzy klasami.

Interfejs definiuje zestaw metod, które klasa implementująca interfejs musi zaimplementować.

Interfejsy są zdefiniowane przez słowo kluczowe **interface** i składają się z nazwy interfejsu, listy metod bez ich implementacji, a także stałych i domyślnych metod. W języku Java klasy abstrakcyjne i interfejsy są narzędziami służącymi do definiowania abstrakcyjnych koncepcji i szkieletów, które są implementowane przez klasy pochodne. Oto kilka różnic między nimi:

- **Implementacja** - Klasa abstrakcyjna może zawierać implementacje metod oraz pól, natomiast interfejs nie może zawierać implementacji żadnej metody, a jedynie deklaracje. Oprócz tego, klasa może dziedziczyć tylko po jednej klasie abstrakcyjnej, ale może implementować wiele interfejsów.

- **Konstruktor** - Klasa abstrakcyjna może mieć konstruktor, natomiast interfejs go nie posiada.
- **Dziedziczenie** - Klasa abstrakcyjna używa słowa kluczowego **extends**, aby być dziedziczona przez inne klasy, natomiast interfejs używa słowa kluczowego **implements** do tego samego celu.
- **Własność** - Klasa abstrakcyjna jest klasą, natomiast interfejs jest swoistego rodzaju kontraktem między klasą a światem zewnętrznym.

Przykładem klasy abstrakcyjnej może być klasa **Zwierze**, która może mieć wiele metod i pól, takich jak **imię**, **waga** i **wiek**, ale nie posiada konkretnej implementacji, ponieważ każde zwierzę ma inną formę i sposób działania.

Klasa **Kot** może dziedziczyć po klasie **Zwierze** i dostosować ją do swoich potrzeb. Przykładem prostego interfejsu może być interfejs **Kwakanie** dla obiektów typu **Kaczka**.

Interfejs ten mógłby zawierać jedną metodę **kwacz()**, która jest implementowana przez różne klasy kaczek w zależności od ich specyficznych zachowań.

Interfejs **Kwakanie** może wyglądać następująco:

```
public interface Kwakanie {
    void kwacz();
}
```

Listing 4.58 Przykład interfejsu

Interfejs ten posiada jedną metodę **kwacz()**. Teraz klasy kaczek, które chcą implementować ten interfejs, muszą zaimplementować metodę **kwacz()** w swoim ciele klasy. Na przykład:

```
public class DzikaKaczka implements Kwakanie {
    public void kwacz(){
        System.out.println("Kwa! Kwa!");
    }
}
```

Listing 4.59 Przykład klasy implementującej interfejs

Klasa **DzikaKaczka** implementuje interfejs Kwakanie poprzez dostarczenie własnej implementacji metody **kwacz()**. Z kolei, inna klasa **GumowaKaczka** może implementować ten sam interfejs w inny sposób, np. przez wydawanie dźwięku **piszcz!**.

```
public class GumowaKaczka implements Kwakanie {
    public void kwacz(){
        System.out.println("Piszcz!");
    }
}
```

Listing 4.60 Przykład klasy implementującej interfejs

Dzięki zastosowaniu interfejsów możemy zapewnić, że różne klasy, które implementują ten sam interfejs, mają tę samą metodę, ale posiadają różne implementacje, co umożliwia różne zachowania w zależności od kontekstu użycia. Klasa może oczywiście implementować więcej interfejsów. W takim przypadku, każdy kolejny zapisujemy po przecinku.

4.7 Kompozycja i agregacja

4.7.1 Agregacja

Agregacja to jeden z mechanizmów relacji między obiektami w programowaniu obiektowym. Opisuje związek typu „**część-całość**" między klasami. Inaczej mówiąc, jedna klasa **całość** zawiera w sobie inną klasę **część**, jednak obiekty tej drugiej klasy mogą istnieć niezależnie.

To znaczy, część elementów klasy może działać sama, bez względu na pozostałą całość. Agregacja może tworzyć hierarchie obiektów. W Javie, agregację możemy podzielić na słabą lub mocną. W przypadku mocnej agregacji obiekt części nie może należeć do więcej niż jednej całości.

Najczęściej w Javie stosuje się słabą agregację, która jest implementowana za pomocą pól obiektowych, gdzie jedna klasa zawiera pole będące obiektem innej klasy. Poniżej przykład zastosowania agregacji. Zacznijmy od stworzenia klasy **Szkola**, klasy **Uczen** oraz klasy **Main**:

```
public class Szkola {
}
public class Uczen{
}
public class Main{
    public static void main(String[] args) {
    }
```

Listing 4.61 Przykład zastosowania agregacji

Zajmijmy się najpierw klasą „**Uczen**". Utworzymy dla niej dwa pola – **imię** i **numer szafki**, konstruktor oraz metodę **getImie()**. Zatem:

```
public class Uczen{
    private String imie;
    private int numerSzafki;

    public Uczen(String imie, int numerSzafki){
        this.imie = imie;
        this.numerSzafki = numerSzafki;
    }

    public String getImie(){
        return imie;
    }
}
```

Listing 4.62 Utworzenie nowych pól, metod oraz konstruktora w klasie „Uczeń”

Zmienna prywatna „**imie**” typu String, reprezentuje imię ucznia, z kolei zmienna **numerSzafki** typu int, przechowuje numer szafki ucznia. Następnie konstruktor, który jest wywoływany podczas tworzenia nowego obiektu **Uczen**. Przyjmuje on dwa parametry – czyli nasze zmienne. Metoda **getImie** posłuży do zwracania imienia ucznia.

Teraz przejdźmy do klasy **Szkola**:

```
public class Szkola {
    private Uczen klasa1B;

    public Szkola(Uczen klasa1B){
        this.klasa1B = klasa1B;
    }
    public void informacjeOSzkole(){
        System.out.println("Klasa 1B ma ucznia o
imieniu: " + klasa1B.getImie());
    }
}
```

Listing 4.63 Klasa „Szkola”

Klasa **Szkola** reprezentuje szkołę i w tym przykładzie posiada jednego ucznia (klasa podrzędna). **Uczen** reprezentuje ucznia w szkole.

W pierwszej linii klasy **Szkola** skorzystaliśmy z agregacji.

Mamy pole **klasa1B** typu **Uczen**. To oznacza, że szkoła zawiera jednego ucznia jako swoją część. Następnie utworzony został konstruktor, w którym przekazujemy obiekty klasy **Uczen**, tworząc związek między szkołą a uczniem. Dodatkowo, dodaliśmy metodę **informacjeOSzkole()**, aby wyświetlić proste informacje na temat szkoły i ucznia.

Na koniec przechodzimy do klasy **Main**, w której tworzymy obiekt klasy **Uczen** (uczen1) i obiekt klasy **Szkola** (szkola), przekazując ucznia do szkoły:

```
public class Main{
    public static void main(String[] args) {
  Uczen uczen1 = new Uczen("Jan Kowalski",27);
   Szkola szkola = new Szkola(uczen1);
    szkola.informacjeOSzkole();
    }
}
```

Listing 4.64 Klasa „Main" z obiektem klasy „Uczen" oraz obiektem klasy „Szkola"

Po uruchomieniu programu otrzymamy informacje o szkole i jednym uczniu:

```
Klasa 1B ma ucznia o imieniu: Jan Kowalski
```

Rysunek 4.15 Wyświetlenie informacji o Uczniu dzięki metodzie „informacjeOSzkole"

W ten sposób, poprzez agregację, tworzymy związek między obiektami szkoły, a studenta gdzie student jest częścią szkoły, ale może też istnieć niezależnie od niej. To umożliwia tworzenie lepiej zorganizowanych struktur obiektowych.

4.7.2 Kompozycja

W przypadku kompozycji, obiekt jednego typu staje się integralną częścią obiektu drugiego. Istnieje między nimi silny związek, gdzie obiekt **część** nie może funkcjonować niezależnie od obiektu **całość**.

W Javie przypomina to trochę jak budowanie domku z poszczególnych elementów. Dom składa się z różnych części, np. drzwi, ściany czy dach. W kompozycji jedna klasa jest jak jedna z tych części, a klasa nadrzędna używa tych części, aby stworzyć coś większego i bardziej zorganizowanego. W skrócie, klasa nadrzędna zawiera obiekt klasy podrzędnej.

Kompozycja tworzy silną zależność między klasami. Obiekt klasy podrzędnej nie może istnieć niezależnie od obiektu klasy nadrzędnej. Wracając do przykładu domu – drzwi są jego częścią, ale samodzielnie nie mogą istnieć i niczego tworzyć. Innym przykładem może być samochód i silnik. Na przykład, jeśli mamy klasy **Silnik** i **Samochod**, to kompozycja oznacza, że klasa **Samochod** zawiera obiekt klasy **Silnik**. **Samochód** nie tylko ma silnik, ale jest od niego zależny - bez silnika samochód nie może działać.

Przedstawimy ten przykład w praktyce. Najpierw utwórzmy klasę **Silnik**, a w niej tworzymy pole, konstruktor i metodę **uruchom**. Na przykład:

```
public class Silnik {
    private String typ;

    public Silnik(String typ){
        this.typ = typ;
    }
    public void uruchom(){
        System.out.println("Silnik uruchomiony");
    }
}
```

Listing 4.65 Klasa „Silnik" z polem, konstruktorem i metodą „uruchom"

Klasa **Silnik** posiada pole **typ**, które określa rodzaj silnika (benzynowy, diesel, itp.). Konstruktor klasy **Silnik** inicjalizuje typ silnika. Metoda **uruchom()** przedstawia komunikat o uruchomieniu silnika.

Klasa Samochod:

```
public class Samochod {
    private Silnik silnik;

    public Samochod(Silnik silnik){
        this.silnik = silnik;
    }

    public void rusz(){
        silnik.uruchom();
        System.out.println("Samochód rusza");
    }
}
```

Listing 4.66 Klasa „Samochod" z polem będącym obiektem klasy „Silnik", konstruktorem oraz metodą „rusz", zawierającą metodę „uruchom"

Klasa **Samochod** zawiera pole **silnik**, które jest obiektem klasy **Silnik**. To oznacza, że samochód ma silnik.

Konstruktor w tej klasie przyjmuje obiekt klasy **Silnik** jako argument i przypisuje go do pola **silnik**.

Metoda **rusz()** wywołuje metodę **uruchom()** z klasy **Silnik** i wyświetla podany komunikat.

W klasie **Main** utworzymy obiekty i sprawdzimy działanie programu. Zapiszmy w niej poniższy kod:

```
public class Main{
    public static void main(String[] args) {
            Silnik benzynowySilnik = new
Silnik("Benzynowy");
                Samochod mojSamochod = new
Samochod(benzynowySilnik);
        mojSamochod.rusz();
    }
}
```

Listing 4.67 Klasa „Main" z obiektami i użyciem metody „rusz"

W przykładzie tworzymy obiekt klasy **Silnik** (benzynowySilnik) i obiekt klasy **Samochod** (mojSamochod), przekazując tym samym silnik do samochodu. Po uruchomieniu programu otrzymamy komunikat, że silnik zostanie uruchomiony, a następnie, że samochód rusza. Tak jak poniżej:

```
Silnik uruchomiony
Samochód rusza
```

Rysunek 4.16 Wynik konsolowy metody „rusz"

Agregacja i kompozycja sprawiają, że projektowanie staje się bardziej elastyczne. Możemy tworzyć skomplikowane struktury, które są jednocześnie zorganizowane i zrozumiałe. Dzięki nim nasze programy są bardziej przygotowane na zmiany. Możemy dodawać nowe elementy lub dostosowywać struktury bez większego problemu.

Agregacja organizuje kod, dzięki czemu jest bardziej czytelny i zrozumiały. Rozdziela odpowiedzialności między różne klasy. Kompozycja z kolei umożliwia strukturyzację kodu, co czyni kod przejrzystym i zrozumiałym. Oparta jest też na silnym powiązaniu między klasami.

4.8 Refaktoryzacja

Dotarliśmy już do momentu, w którym jesteśmy w stanie stworzyć działający program, korzystając z różnorodnych instrukcji, funkcji i konceptów programowania obiektowego, które tworzą strukturę programu. Chcąc tworzyć bardziej zaawansowane programy, potrzebujemy umieścić w nim większą ilość metod i pól. Jednak im bardziej rozbudowujemy program, tym trudniej zapanować nad wszystkim. Z czasem struktura programu może stać się skomplikowana i trudna do zrozumienia.

Z pomocą przychodzi nam wówczas **refaktoryzacja** – proces, który pomaga poprawić jakość naszego kodu, nie zmieniając działania programu.

Zamiast pisać wszystko od nowa, po prostu upraszczamy już istniejący kod i modyfikujemy go, aby był bardziej czytelny, zorganizowany i łatwiejszy w utrzymaniu. Nie będziemy dodawać nowych funkcji, ale poprawimy je w taki sposób, aby były w jak najlepszej formie. Przejdźmy zatem do kilku przykładów refaktoryzacji kodu w Javie.

Pierwszym z nich jest ekstrakcja metody. Na początek tworzymy nową klasę, a w niej metodę, którą potem przekształcimy:

```
public void obliczWynik(int a, int b){
    int suma = a - b;
    int wynik = suma * 4;
    System.out.println("Wynik: " + wynik);
}
```

Listing 4.68 Metoda „obliczWynik”

Metoda **obliczWynik()** wykonuje dwie operacje. Najpierw odejmuje dwie liczby **a** i **b**, a następnie mnoży wynik przez **4**, i ostatecznie wypisuje go na ekranie. Choć kod działa, może być bardziej czytelny. Utworzymy w tym celu nową metodę.

Kod będzie wyglądał w ten sposób:

```
public void obliczWynik(int a, int b){
    int suma = obliczSume(a,b);
    int wynik = suma * 4;
    System.out.println("Wynik: " + wynik);
}
private int obliczSume(int a, int b){
    return a + b;
}
```

Listing 4.69 Przykład refaktoryzacji funkcji

Po refaktoryzacji, funkcja **obliczWynik()** nadal robi to samo, ale teraz używa dodatkowej metody **obliczSume()**, która została wydzielona do osobnej funkcji. Ten proces nazywany jest ekstrakcją metody.

Działa to tak, jakbyśmy wydzielili pewien fragment kodu do osobnej **funkcji-pomocnika**, aby zwiększyć czytelność i umożliwić ponowne użycie.

Dodanie **obliczSume()** jako osobnej metody pomaga w:

- **czytelności kodu** – nazwa metody „obliczSume” jasno wskazuje, co ta funkcja robi, co ułatwia zrozumienie kodu.

- **ponownym użyciu** – jeśli gdzieś indziej w programie potrzebujemy obliczyć sumę dwóch liczb, możemy teraz użyć funkcji „obliczSume()” zamiast powtarzać ten sam kod.

To właśnie jedno z wielu podejść do refaktoryzacji, mające na celu uczynienie kodu bardziej zrozumiałym, elastycznym i łatwiejszym do zarządzania. Kolejnym podejściem będzie podział na klasy.

Utwórzmy klasę **Kalkulator**, a w niej zapiszmy:

```
public class Kalkulator {
    public int dodaj(int a, int b){
        return a + b;
    }
    public int pomnoz(int a, int b){
        return a * b;
    }
}
```

Listing 4.70 Klasa posiadająca dwie metody

Mamy klasę **kalkulator**, która zawiera dwie metody: **dodaj()** i **pomnoz()**. Ta klasa pełni dwie różne funkcje – zarówno dodawanie, jak i mnożenie. Choć jest to zgodne z zasadami programowania obiektowego, czasem można poprawić strukturę kodu. Możemy te klasę podzielić na dwie oddzielne. Na przykład **Dodawacz** i **Mnoznik**.

Zatem:

```
public class Dodawacz {
    public int dodaj(int a, int b){
        return a + b;
    }
}
```

Listing 4.71 Klasa „Mnoznik” z metodą „pomnoz”

Każda z tych klas wykonuje teraz tylko jedno konkretne zadanie – **Dodawacz** dodaje liczby, a **Mnoznik** je mnoży. Jest to korzystne ze względu na klarowność i zrozumiałość. Każda klasa ma jedno jasno zdefiniowane zadania. Gdy funkcje są podzielona na dwie osobne klasy, zarządzanie nimi staje się prostsze. Jeśli problem wystąpi z dodawaniem to zaglądamy do klasy **Dodawacz**, jeśli z mnożeniem, to do klasy **Mnoznik**. Gdybyśmy w przyszłości chcieli utworzyć dodatkowe operacje matematyczne, możemy łatwo dodać nowe klasy, bez konieczności modyfikowania istniejących.

Kolejnym przykład refaktoryzacji kodu polega na poprawie nazw zmiennych. W nowej klasie utwórzmy metodę **obliczPole()**:

```
public double obliczPole(int r){
    double p = 3.14 * r * r;
    return p;
}
```

Listing 4.72 Metoda „obliczPole" zwracająca „p"

W powyższym kodzie mamy funkcję, która oblicze pole koła na podstawie promienia, ale zmienna **p** jako nazwa dla pola koła jest mało opisowa. Zmodyfikujmy nieco tę metodę:

```
public double obliczPoleKola(int promien){
    double poleKola = 3.14 * promien * promien;
    return poleKola;
}
```

Listing 4.73 Metoda „obliczPoleKola" zwracająca „poleKola"

W tej refaktoryzacji zmieniliśmy nazwę funkcji na **obliczPoleKola()**, co bardziej precyzyjnie opisuje, co ta funkcja robi. Ponadto, zmieniliśmy nazwę zmiennej z **p** na **poleKola**, co sprawie, że jest jasne, czym metoda się zajmuje.

Opisowe nazwy zmiennych i funkcji pomagają w zrozumieniu, co dokładnie robi dany fragment kodu. Nazwa **p** mogłaby sugerować, że chodzi o **punkt**, **prędkość** lub inne pojęcie. Nazwa **poleKola** nie sprawia nam żadnych wątpliwości, co oznacza ta zmienna.

Ważne jest, aby utrzymać konwencje nazewnicze, takie jak używanie **camelCase** (pierwsze słowo zaczynające się małą literą, a każde kolejne wielką) czy opisowe nazwy zmiennych.

Ostatnim sposobem refaktoryzacji, który przedstawimy będzie użycie interfejsów. Dla przykładu utwórzmy klasę **Kwadrat**, a następnie metodę **obliczPole()** typu double:

```
public class Kwadrat {
    public double obliczPole(double bok){
        return bok * bok;
    }
}
```

Listing 4.74 Klasa „Kwadrat" z metodą „obliczPole"

Metoda **obliczPole()** oblicza pole kwadratu na podstawie długości boku. Teraz, utwórzmy interfejs **Figura** z metodą **obliczPole():**

```
public interface Figura {
    double obliczPole(double bok);
}
```

Listing 4.75 Interfejs „Figura” z metodą „obliczPole”

Klasie **Kwadrat** dodajmy implementację z klasy **Figura** i nadpiszmy metodę **obliczPole()**:

```
public class Kwadrat implements Figura{
    @Override
    public double obliczPole(double bok){
        return bok * bok;
    }
}
```

Listing 4.76 Klasa „Kwadrat” implementująca z klasy „Figura”, nadpisująca metodę „obliczPole”

Po refaktoryzacji, wprowadziliśmy interfejs **Figura**, który definiuje metodę **obliczPole()**. Klasa **Kwadrat** implementuje ten interfejs, co oznacza, że musi dostarczyć implementację metody **obliczPole().** Korzystając z interfejsu, oddzielamy definicję metody od jej implementacji, co sprawia, że kod staję się bardziej zwięzły. Jeśli w przyszłości chcemy dodać więcej figur geometrycznych, możemy po prostu utworzyć nowe klasy implementujące interfejs **Figura**. Całość daje nam bardziej elastyczną strukturę kodu i ułatwia dodawanie nowych figur geometrycznych.

Podsumowując, celem refaktoryzacji jest przede wszystkim poprawa jakości kodu poprzez eliminację zbędnych fragmentów, ulepszanie nazw zmiennych i funkcji, a także zastosowanie bardziej czytelnych oraz efektywnych konstrukcji. Są to kluczowe kroki, które pomogą nam w doskonaleniu programu.

Refaktoryzacja to nie tylko techniczne poprawki, lecz również głębsze zrozumienie kodu. Dzięki temu nasze projekty stają się łatwiejsze do rozwijania, a co najważniejsze – bardziej przyjazne i przejrzyste dla innych programistów, którzy będą mieli styczność z naszym kodem.

4.9 Sprawdź się!

1) Które z poniższych stwierdzeń dotyczących getterów i setterów w języku Java jest prawdziwe?

a) Getter służy do ustawiania wartości pola klasy, a setter do pobierania wartości.

b) Getter i setter są metodami, które umożliwiają dostęp do prywatnych pól klasy.

c) Getter i setter są automatycznie generowane przez kompilator wraz z utworzeniem klasy.

d) Getter i setter są dostępne tylko dla klas abstrakcyjnych.

2) Które z poniższych słów kluczowych jest używane do dziedziczenia klas w języku Java?

a) new

b) extends

c) implements

d) abstract

3) Co robi słowo kluczowe this w języku Java?

a) Wskazuje na obiekt wywołujący daną metodę.

b) Tworzy nową instancję klasy.

c) Zamyka strumień danych.

d) Umożliwia dostęp do zmiennych statycznych.

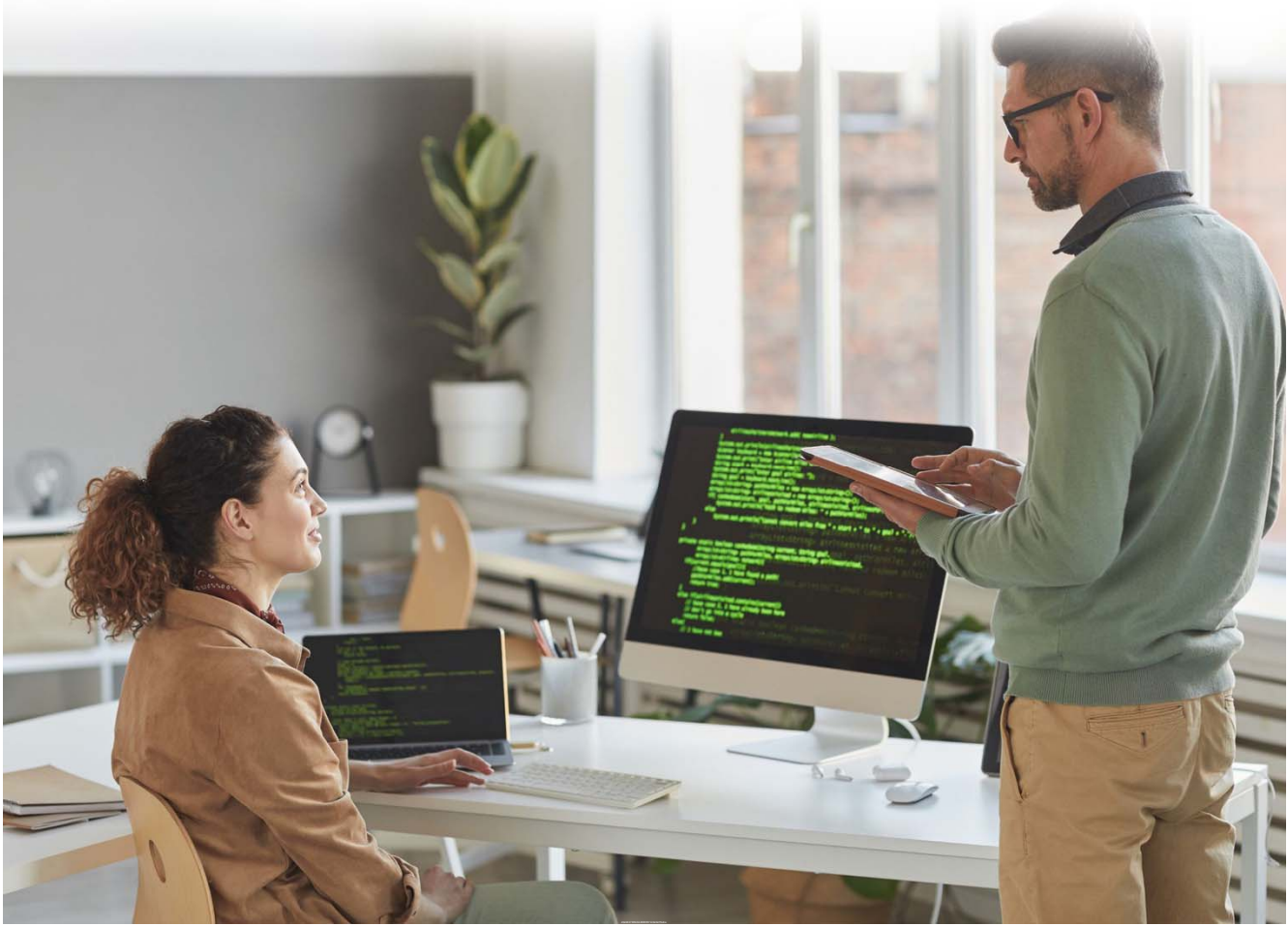
Zadanie 4.9.1

Napisz klasę „**Stack**” reprezentującą strukturę danych stosu. Klasa powinna zawierać następujące metody:

- **push(element)**: Dodaje nowy element na wierzch stosu.
- **pop()**: Usuwa i zwraca element znajdujący się na wierzchu stosu.
- **peek()**: Zwraca element znajdujący się na wierzchu stosu, nie usuwając go.
- **isEmpty()**: Zwraca true, jeśli stos jest pusty, w przeciwnym razie false.

ROZDZIAŁ 5

PROGRAMOWANIE WSPÓŁBIEŻNE



5 Programowanie współbieżne

Wielowątkowość to kluczowy aspekt współczesnych technologii, który odgrywa istotną rolę w naszym codziennym życiu. Choć wielu z nas może nie zdawać sobie z tego sprawy, doświadczamy jej wpływu każdego dnia, korzystając z różnych urządzeń, takich jak komputer, telefon czy tablet.

Pomyślmy o sytuacji, gdy przeglądamy Internet na swoim komputerze. Jesteśmy w stanie otworzyć kilka kart przeglądarki jednocześnie, sprawdzić pocztę elektroniczną, słuchać muzyki na platformie streamingowej, a jednocześnie pisać raport do pracy. Ponadto nawet poza naszą wiedzą na bieżąco aktualizowana jest godzina czy informacje o pogodzie. Wszystkie te działania są możliwe dzięki wielowątkowości, która umożliwia jednoczesne wykonywanie wielu zadań.

Podobnie dzieje się na naszych telefonach komórkowych. Możemy jednocześnie wysyłać wiadomości tekstowe, przeglądać media społecznościowe, sprawdzać pogodę i korzystać z nawigacji GPS. Dzięki wielowątkowości, nasze telefony są w stanie obsługiwać wiele aplikacji jednocześnie, zapewniając płynne i responsywne doświadczenie użytkownika.

Podobnie jak zastosowanie wielordzeniowych procesorów w naszych urządzeniach, tak wielowątkowość umożliwia realizację wielu zadań jednocześnie, co prowadzi do znacznego zwiększenia wydajności i efektywności naszych programów. Podobnie jak procesory wykorzystują wielowątkowość do jednoczesnego wykonywania różnych operacji, programowanie wielowątkowe w Javie daje nam narzędzia do tworzenia aplikacji, które mogą równocześnie przetwarzać różne zadania.

Dzięki programowaniu wielowątkowemu w Javie możemy tworzyć aplikacje, które wykorzystują pełnię potencjału naszych komputerów, umożliwiając jednoczesne wykonywanie zadań obliczeniowych, operacji wejścia/wyjścia oraz obsługi interfejsu użytkownika. To oznacza, że nasze aplikacje mogą być bardziej responsywne, wydajne i skalowalne, co przekłada się na lepsze doświadczenia użytkowników oraz efektywniejsze wykorzystanie zasobów sprzętowych.

W tym rozdziale zgłębimy tajniki programowania wielowątkowego w języku Java i poznamy różnorodne techniki i narzędzia, które pozwalają nam wykorzystać potencjał wielowątkowości. Przyjrzymy się praktycznym

zastosowaniom, jak również omówimy znane praktyki i wzorce projektowe, które pomogą nam tworzyć wydajne, skalowalne i niezawodne aplikacje w środowisku Javy.

5.1 Program sekwencyjny, a program współbieżny

Dowiedzieliśmy się już czym są wątki i jak potężne dają korzyści. Teraz musimy zrozumieć na czym polega fundamentalna różnica pomiędzy programem sekwencyjnym z programem współbieżnym.

Program sekwencyjny to tradycyjny program który wykonuje krok po kroku kolejne operacje. Każde zadanie musi zaczekać na zakończenie poprzedniego aby rozpocząć swoją pracę. Jest to program najbardziej pospolity, każdy dotychczas omawiany w tej książce temat był napisany właśnie sekwencyjnie. Nie oznacza to jednak, że w tym programie wątki nie występują. Otóż każdy program wykonywany jest w ramach jednego wątku, zwanego głównym.

Program współbieżny z kolei pozwala na wykonywanie wielu zadań jednocześnie, dzięki użyciu wątków. Każdy wątek może pracować niezależnie, co umożliwia efektywne wykorzystanie zasobów systemu i zwiększa responsywność aplikacji.

Różnice pomiędzy obydwoma programami można zobrazować poprzez przykład z życia. Wyobraźmy sobie kolejkę w sklepie. Klienci do kasy to operacje do wykonania, a kasjer to pojedynczy wątek. W przypadku programu sekwencyjnego mamy do czynienia z jedną otwartą kasą. Wszyscy zostaną obsłużeni jednak może to zająć dłuższą chwilę. Gdy na kasach zacznie obsługiwać więcej kasjerów, ta sama ilość klientów zostanie obsłużona w znacznie krótszym czasie.

5.2 Wybór między programowaniem sekwencyjnym a współbieżnym

W praktyce, wybór między programem sekwencyjnym a współbieżnym zależy od specyfiki projektu. Zastosowanie wątków staje się istotne, gdy mamy do czynienia z równoczesnymi operacjami, takimi jak obsługa wielu klientów w serwerze czy przetwarzanie danych w czasie rzeczywistym. Warto jednak pamiętać, że programowanie współbieżne wymaga także odpowiednich mechanizmów synchronizacji, aby uniknąć konfliktów i zagwarantować poprawność działania aplikacji.

5.3 Równoległość a współbieżność, proces a program, przeplot

W informatyce istnieje wiele kluczowych koncepcji związanych z przetwarzaniem danych i wykonywaniem zadań. Równoległość, współbieżność, proces, program i przeplot to pojęcia fundamentalne dla zrozumienia sposobu, w jaki komputer wykonuje instrukcje oraz zarządza zadaniami.

5.3.1 Równoległość vs. Współbieżność

Równoległość i współbieżność to dwie kluczowe koncepcje związane z jednoczesnym przetwarzaniem. Równoległość odnosi się do faktycznego jednoczesnego wykonywania wielu zadań na wielu procesorach lub rdzeniach, co przyspiesza czas wykonania. Z kolei współbieżność dotyczy jednoczesnego przetwarzania zadań, ale niekoniecznie w tym samym momencie. W kontekście Javy, wielowątkowość wprowadza elementy współbieżności, umożliwiając jednoczesne przetwarzanie różnych wątków.

5.3.2 Proces a Program

Rozróżnienie między procesem a programem jest bardzo istotne. Proces to instancja programu uruchomiona w systemie operacyjnym, która ma swój własny obszar pamięci, zasoby i stan. Może zawierać wiele wątków, które wykonują różne części programu jednocześnie. Z kolei program to abstrakcyjna koncepcja, opisująca algorytmy i logikę przetwarzania danych, niezależnie od konkretnego wykonania.

5.3.3 Mechanizm przeplotu

Przeplot to technika, w ramach której różne wątki lub procesy są wykonywane naprzemiennie, dzieląc czas procesora. Mechanizm ten pozwala na równomierne przydzielanie zasobów, a także unika blokowania całego programu w przypadku jednego zatrzymującego się wątku.

5.4 Tworzenie wątków

Teraz kiedy już poznaliśmy teoretyczne znaczenie programowania wielowątkowego oraz definicji z nim związanych możemy przejść do ich praktycznego poznania.

Na początek warto wiedzieć że wątki posiadają cykl życia który składa się z kilku etapów:

- **New** - to nowoutworzony wątek, jednak wciąż nie wykonujący żadnej czynności;
- **Runnable** - następuje po wykonaniu metody uruchamiającej;
- **Terminated** – następuje dla każdego wątku który skończył swoją pracę;
- **Blocked** – stan w którym wątek jest zatrzymany;
- **Time Waiting** – stan w którym wątek jest wstrzymany na określony czas;
- **Waiting** – wątek został wstrzymany, np. przez metodę „join()”;

Tworzenie nowych wątków może odbywać się na różne sposoby w zależności od wymagań aplikacji. My spróbujemy je napisać na trzy sposoby.

5.4.1 Klasa Thread

Thread jest klasą dostarczaną przez Javę która odpowiada za zarządzanie wątkami. My wykorzystamy ją żeby zrobić własny. Musimy nadpisać w niej metodę **run** która będzie zawierała instrukcje do wykonania przez nasz wątek. Dodatkowo za jej pomocą spróbujemy go uruchomić.

Zacznijmy od stworzenia własnej klasy. Możemy ją nazwać **Watki**, ważne żeby rozszerzała klasę **Thread**. W ciele klasy umieszczamy metodę **run** i dopisujemy jej implementację. Na początku możemy wyświetlić nazwę naszego wątku. Powinna wyglądać jak poniżej.

```
public class Watki extends Thread{

    public void run() {
        System.out.println(currentThread().getName());
    }
}
```

Listing 5.1 Zwrócenie nazwy aktualnego wątku

Wykorzystane w instrukcji **print** metody odpowiadają kolejno za zwrócenie aktualnego wątku **currentThread** oraz wyciągnięcie z niego samej nazwy **getName**. Teraz przejdźmy do utworzenia obiektu naszej klasy oraz uruchomienia programu. Nazwijmy go **watek1**. Następnie wykonajmy na nim metodę **run()**.

```
public static void main(String[] args) {
    Watki watek1 = new Watki();
    watek1.run();
}
```

Listing 5.2 Uruchomienie wątku metodą „run”

Uruchommy program. W konsoli dostaniemy wynik jak poniżej.

```
main


Process finished with exit code 0
```

Rysunek 5.1 Wynik programu w konsoli

Jak widać wyświetlona została wartość **main**. Jak dobrze pamiętamy metoda „**run**” którą nadpisaliśmy ma wypisać nazwę obecnego wątku, to właśnie ona.

Możemy to trochę poprawić. Nadajmy własną nazwę dla naszego wątku za pomocą metody **setName()**. Dodajmy ją w kodzie przed wywołaniem **watek1.run();** i ustawmy nową nazwę na **MojWatek**.

```
public static void main(String[] args) {

    Watki watek1 = new Watki();
    watek1.setName("MojWatek");
    watek1.run();

}
```

Listing 5.3 Zmiana nazwy wątku

Ponownie uruchamiamy program.

```
main


Process finished with exit code 0
```

Rysunek 5.2 Wynik nieudanej zmiany nazwy

Niestety program nie zadziałał. Nazwa nie zmieniła się, a wina leży w wywołaniu metody **run()** na naszym wątku. Otóż faktycznie uruchamia ona jego działanie ale jedynie w ramach wątku głównego aplikacji, tzn. że nie działa

on współbieżnie do programu. Jeśli przyjrzymy się dokładniej Klasie **Thread** (w IDE klikamy na nazwę klasy **CTRL+LPM**) zobaczymy że posiada ona również metodę **start()**. To właśnie za jej pomocą uruchomimy nasz obiekt jako osobny wątek.

Wprowadźmy zatem poprawki. Musimy zamienić wywołanie metody **run()** na **start()**.

```
public static void main(String[] args) {

    Watki watek1 = new Watki();
    watek1.setName("MojWatek");
    watek1.start();

}
```

Listing 5.4 Zmiana metody startowej z „run" na „start"

Znów uruchommy nasz program.

```
MojWatek


Process finished with exit code 0
```

Rysunek 5.3 Wypisanie nowej nazwy w konsoli

Tym razem faktycznie wyświetlona została nazwa którą ustawiliśmy. Oznacza to że został ożywiony nowy wątek. Zróbmy jeszcze jeden przykład. Usuńmy linijkę w której ustawialiśmy nazwę dla wątku, a następnie dodajmy kolejne dwa obiekty klasy Watek. Nazwijmy je **watek2** oraz **watek3** oraz wykonajmy metodę **start()** na pierwszym i **run()** na drugim.

```
public static void main(String[] args) {

    Watki watek1 = new Watki();
    Watki watek2 = new Watki();
    Watki watek3 = new Watki();
    watek1.start();
    watek2.start();
    watek3.run();

}
```

Listing 5.5 Utworzenie kilku wątków

Teraz uruchommy nasz program.

```
main
Thread-0
Thread-1

Process finished with exit code 0
```

Rysunek 5.4 Wynik działania programu

W wyniku otrzymaliśmy **3 nazwy** wątków. Jak widać jedna z nich to wątek główny. Oczywiście pochodzi ona z obiektu **watek3**. Zastanawiające może być dlaczego wyświetlił się on jako pierwszy. Ogólnie rzecz biorąc programowanie wielowątkowe jest w dużym stopniu zależne od sprzętu na którym uruchamiamy aplikację. Jeśli nie zarządzamy w żaden sposób kolejnością wykonywanych zadań to wyniki mogą się różnić mimo identycznych wartości na wejściu. Możemy wykorzystać metodę klasy **Thread**, a mianowicie **setPriotity()** która przyjmuje kolejno wartości:

- **1** – dla najmniejszego priorytetu,
- **5** – dla normalnego priorytetu,
- **10** – dla największego priorytetu.

Nie ma jednak pewności jak bardzo wpłynie to na faktyczny priorytet wątku. Jest to raczej sugestia dla programu niż precyzyjna deklaracja.

Gdybyśmy chcieli skrócić zapis tworzenia wątków i poprawić czytelność możemy także wykonać za pomocą wyrażenia lambda. Może to wyglądać tak jak poniżej.

```
public static void main(String[] args) {

    Thread thread = new Thread(() ->
            System.out.println("Nazwa wątku: " +
Thread.currentThread().getName()));

    thread.start();
}
```

Listing 5.6 Utworzenie nowego wątku za pomocą wyrażenia lambda

W sytuacji tworzenia tak krótkich wątków może to być przydatne.

5.4.2 Interfejs Runnable

Teraz do utworzenia wątków użyjemy interfejsu **Runnable**. Zaletą wykorzystania go zamiast użycia samej klasy **Thread** jest oczywiście ukrywanie szczegółów implementacji, a zatem zastosowanie abstrakcji w naszym programie co z punktu widzenia poprawnego pisania aplikacji jest dobrą praktyką. **Runnable** posiada tylko jedną metodę i jest to „**run**".

Zacznijmy od stworzenia nowej klasy o nazwie np. „**WatekRunnable**". Następnie wskażmy że klasa implementuje **Runnable**. W klasie musimy napisać implementację dla metody „**run**". Będziemy chcieli zobaczyć jak będą wykonywać się zadania dwóch(lub więcej) wątków. Innymi słowy zobaczymy czy faktycznie dojdzie do wielozadaniowości w aplikacji. W tym celu najlepiej będzie posłużyć się zwykłą pętlą która wypisze nam nazwę wątku oraz numer iteracji. Całość powinna wyglądać jak poniżej.

```
 public class WatekRunnable implements Runnable{
    public void run() {
        for(int i = 0; i < 99; i++){

System.out.println(Thread.currentThread().getName() + ",
i = " + i );
        }
    }
}
```

Listing 5.7 Nadpisanie metody „run" z interfejsu „Runnable"

Warte zauważenia jest że aby dostać się do metody „**currentThread()"** musimy wykorzystać klasę **Thread**. Będzie nam ona potrzebna również przy uruchomieniu wątków ponieważ na jej obiekcie będziemy wywoływać napisaną przez nas metodę **run**. Przejdźmy zatem do utworzenia obiektów.

Zacznijmy od stworzenia dwóch instancji klasy **WatekRunnable** które przekażemy jako parametry przy tworzeniu dwóch instancji klasy **Thread**. Następnie możemy zmienić nazwy wątków które stworzyliśmy za pomocą metody **setName()**. Ułatwi to analizę wyników w konsoli. Na koniec wykonujemy na obiektach metodę **start()**. Całość powinna wyglądać jak poniżej.

```
public static void main(String[] args) {

    WatekRunnable mojWatek1 = new WatekRunnable();
    WatekRunnable mojWatek2 = new WatekRunnable();

    Thread thread1 = new Thread(mojWatek1);
    Thread thread2 = new Thread(mojWatek2);

    thread1.setName("pierwszy");
    thread2.setName("drugi");

    thread1.start();
    thread2.start();

}
```

Listing 5.8 Uruchomienie programu z dwoma nowymi wątkami

Teraz możemy uruchomić nasz program. W konsoli wypisane zostaną kolejne iteracje pętli z obu wątków jednak przeplatają się między sobą.

```
drugi, i = 0
drugi, i = 1
pierwszy, i = 0
drugi, i = 2
pierwszy, i = 1
drugi, i = 3
pierwszy, i = 2
drugi, i = 4
pierwszy, i = 3
drugi, i = 5
pierwszy, i = 4
drugi, i = 6
pierwszy, i = 5
drugi, i = 7
pierwszy, i = 6
drugi, i = 8
pierwszy, i = 7
drugi, i = 9
pierwszy, i = 8
drugi, i = 10
```

Rysunek 5.5 Wynik działania w konsoli

Jak widzimy oba wątki pracują jednocześnie ale nie zawsze jest to tak wyrównany wyścig. W zależności od obciążenia procesora w danym momencie może dość do sytuacji że dysproporcja między oboma wątkami będzie znaczna. Aby to zaobserwować warto uruchomić program kilka razy, obserwując kolejne wyniki. Mimo że kod programu jest identyczny to kolejność wyników prawie zawsze będzie się różnić.

Oto przykład w którym wątek drugi otrzymał przez moment więcej pamięci obliczeniowej procesora, przez co wyprzedził wątek pierwszy

```
pierwszy, i = 5
drugi, i = 8
drugi, i = 9
pierwszy, i = 6
drugi, i = 10
drugi, i = 11
pierwszy, i = 7
drugi, i = 12
drugi, i = 13
drugi, i = 14
drugi, i = 15
drugi, i = 16
pierwszy, i = 8
drugi, i = 17
pierwszy, i = 9
drugi, i = 18
pierwszy, i = 10
drugi, i = 19
pierwszy, i = 11
drugi, i = 20
```

Rysunek 5.6 Wskazanie fragmentu gdzie występuje przewaga jednego z wątków

5.4.3 Klasa ExecutorService

Klasa **ExecutorService** to narzędzie, które pozwala nam zarządzać pulą wątków do wykonywania zadań w naszych aplikacjach. Dzięki niej możemy

kontrolować, jak wiele zadań jest wykonywanych jednocześnie, oraz w jaki sposób są one przetwarzane. Możemy również zdecydować, czy chcemy, aby nasza pula wątków była ograniczona do określonej liczby wątków (na przykład, aby uniknąć przeciążenia systemu), czy też miała dynamicznie dostosowywać się do obciążenia aplikacji.

Żeby napisać prosty przykład działania posłużymy się klasą **WatekRunnable**, którą napisaliśmy we wcześniejszym rozdziale. Zmodyfikujemy jedynie klasę w której uruchamiamy program. Zacznijmy od utworzenia dwóch obiektów naszej klasy oraz jednego obiektu klasy **ExecutorService**. Następnie określmy liczbę dostępnych wątków w puli. Aby to zrobić przypisujemy obiektowi naszego serwisu wartość **Executors.newFixedThreadPool(2) Liczba 2** oznacza że do wykonania zadań dostępne będą dwa wątki. Teraz za pomocą metody **submit()** dodajmy zadania do wykonania Całość powinna wyglądać jak poniżej.

```
public static void main(String[] args) {

    WatekRunnable watek1 = new WatekRunnable();
    WatekRunnable watek2 = new WatekRunnable();
    ExecutorService executorService;

    executorService= Executors.newFixedThreadPool(2);
    executorService.submit(watek1);
    executorService.submit(watek2);

}
```

Listing 5.9 Utworzenie puli wątków

Teraz możemy uruchomić naszą aplikację.

```
pool-1-thread-1, i = 0
pool-1-thread-2, i = 0
pool-1-thread-1, i = 1
pool-1-thread-2, i = 1
pool-1-thread-1, i = 2
pool-1-thread-2, i = 2
pool-1-thread-1, i = 3
pool-1-thread-1, i = 4
pool-1-thread-1, i = 5
pool-1-thread-2, i = 3
pool-1-thread-1, i = 6
pool-1-thread-2, i = 4
pool-1-thread-2, i = 5
pool-1-thread-2, i = 6
pool-1-thread-2, i = 7
pool-1-thread-2, i = 8
pool-1-thread-2, i = 9
pool-1-thread-2, i = 10
pool-1-thread-1, i = 7
pool-1-thread-1, i = 8
```

Rysunek 5.7 Wynik działania programu

Jak widać aplikacja zaczęła wykonywać pracę współbieżnie. Jednak przyjrzyjmy się ostatniej linijce wyświetlonej w konsoli.

```
pool-1-thread-1, i = 90
pool-1-thread-1, i = 91
pool-1-thread-2, i = 98
pool-1-thread-1, i = 92
pool-1-thread-1, i = 93
pool-1-thread-1, i = 94
pool-1-thread-1, i = 95
pool-1-thread-1, i = 96
pool-1-thread-1, i = 97
pool-1-thread-1, i = 98
```

Rysunek 5.8 Brak linii informującej o zakończeniu pracy programu

Zakres pętli który ustawialiśmy w klasie **WatekRunnable** ustawiony był na **99**, a zatem wygląda na to, że oba wątki zakończyły przechodzenie pętli. Jednak program nie zakończył pracy. Przy prawidłowym wykonaniu programu powinniśmy otrzymać w konsoli wiadomość **Process finished with exit code 0**. Tutaj jednak go nie ma. Dodatkowo jeśli spojrzymy naszemu IDE to nad edytorem będzie miał dostępny przycisk **Stop** co świadczy o ciągłej pracy aplikacji. Jest to spowodowane przez **ExecutorService** który nie zamyka się gdy wątki skończą pracę tylko jest ciągle aktywny. Aby to zmienić musimy dodać w kodzie instrukcję **shutdown()**.

```
public static void main(String[] args) {

    WatekRunnable watek1 = new WatekRunnable();
    WatekRunnable watek2 = new WatekRunnable();
    ExecutorService executorService;

    executorService= Executors.newFixedThreadPool(2);
    executorService.submit(watek1);
    executorService.submit(watek2);

    executorService.shutdown();

}
```

Listing 5.10 Dodanie metody zamykającej pulę po zakończeniu pracy.

Jeśli teraz ponownie uruchomimy program powinien on zakończyć swoją pracę normalnie.

```
pool-1-thread-1, i = 94
pool-1-thread-1, i = 95
pool-1-thread-1, i = 96
pool-1-thread-1, i = 97
pool-1-thread-1, i = 98

Process finished with exit code 0
```

Rysunek 5.9 Wynik przedstawiający poprawne zakończenie działania programu

Korzystając z **ExecutorService**, możemy tworzyć bardziej wydajne, skalowalne i responsywne aplikacje, dlatego jest to rozwiązanie lepsze niż stosowanie samej klasy **Thread**.

5.5 Metody sleep() i join()

W programowaniu współbieżnym, korzystanie z metod **sleep()** i **join()** jest ważne dla efektywnego zarządzania wątkami. Przyjrzymy się sposobom ich działania aby zrozumieć jak wpływają na pracę programu.

Pierwszą rzeczą o której trzeba pamiętać podczas korzystania z tych metod to obsługa wyjątku **InterruptedException** który jest rzucany w sytuacji gdy inny wątek przerywa działanie

5.5.1 Metoda sleep()

Metoda **sleep()** jest wykorzystywana do zawieszenia wykonania wątku na określony czas. Podczas gdy wątek **śpi**, inne wątki mogą kontynuować swoje działanie. Jest to przydatne, gdy chcemy wywołać opóźnienie w działaniu wątku lub kontrolować tempo przetwarzania.

Możemy przykładowo wstrzymać działanie wątku głównego aplikacji na **4** sekundy. Aby to zrobić napiszemy program który odczyta godzinę rozpoczęcia programu oraz zakończenia a następnie obliczy różnicę w zmierzonym czasie. Będzie nam do tego potrzebne zaimportowanie takich klas jak **Duration**, **LocalDateTime** oraz **DateTimeFormatter**. W ciele metody pobieramy aktualny czas do zmiennej oraz dodajemy formatowanie dla lepszego wyglądu.

```
LocalDateTime start = LocalDateTime.now();
DateTimeFormatter formatowanie =
DateTimeFormatter.ofPattern("HH:mm:ss");
```

Listing 5.11 Pobieranie aktualnej godziny oraz zadeklarowanie formatu dla wyświetlania czasu

Następnie umieszczamy blok **try-catch** w którym określamy na jak długo uśpiony jest wątek oraz dodajemy obsługę wyjątku **InterruptedException**.

```
try {
    Thread.sleep(4000);
}
catch (InterruptedException e) {
    System.out.println(e);
}
```

Listing 5.12 Obsługa wyjątku „InterruptedException" blokiem try-catch

Na koniec do nowej zmiennej przypisujemy aktualny czas oraz obliczamy różnicę pomiędzy startem i końcem działania aplikacji. Warto dodać kilka

instrukcji **print**, które ułatwią czytanie wiadomości z konsoli. Całość powinna wyglądać jak poniżej.

```
public static void main(String[] args){

    LocalDateTime start = LocalDateTime.now();
    DateTimeFormatter formatowanie =
DateTimeFormatter.ofPattern("HH:mm:ss");

    System.out.println("Program rozpoczął pracę o: " +
start.format(formatowanie));

    System.out.println("Proszę czekać...");

    try {
        Thread.sleep(4000);
    }
    catch (InterruptedException e) {
        e.printStackTrace();

    }

    LocalDateTime koniec = LocalDateTime.now();
    System.out.println("Wątek zakończył pracę o: " +
koniec.format(formatowanie));

    Duration duration = Duration.between(start, koniec);
    long roznica = duration.getSeconds();
    System.out.println("Czas pracy: " + roznica + "
s.");
}
```

Listing 5.13 Program usypiany jest na cztery sekundy

Teraz możemy uruchomić program.

```
Program rozpoczął pracę o: 13:35:01
Proszę czekać...
Wątek zakończył pracę o: 13:35:05
Czas pracy: 4 s.

Process finished with exit code 0
```

Rysunek 5.10 Wynik z konsoli. Obliczenie czasu pracy programu

Jak można zauważyć, zaraz po uruchomieniu programu w konsoli wyświetliły się pierwsze dwie linijki, następnie program natrafił na instrukcję uśpienia która spowodowała wstrzymanie działania programu na **4 sekundy**.

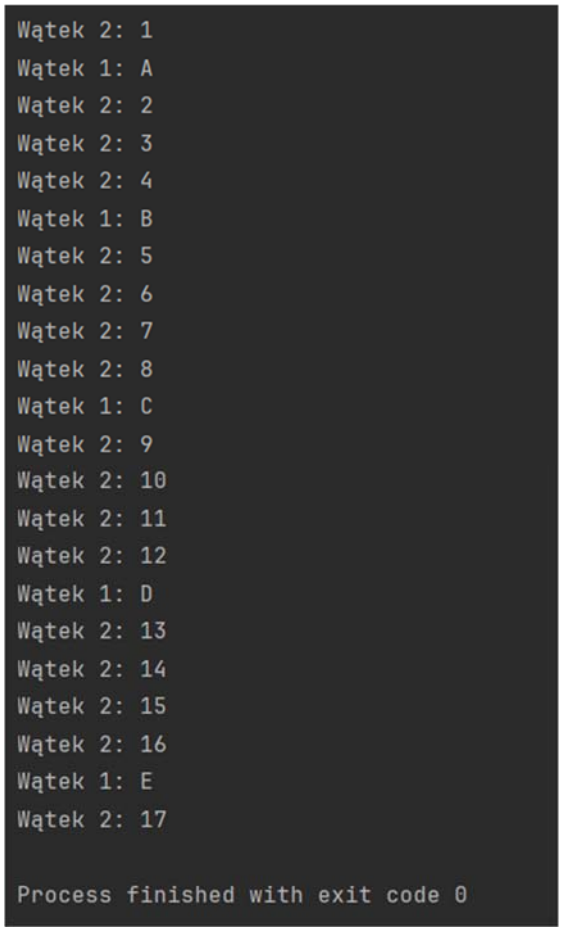
To jednak przykład zatrzymania jedynie wątku głównego aplikacji. Spróbujmy teraz wykonać przykład na dwóch wątkach. Pierwszy będzie wypisywał za pomocą pętli litery od **A** do **E**, a drugi będzie wypisywał liczby od **1** do **16**.

Dodamy metodę **sleep()** z wartością **4000** dla wątku wyświetlającego litery i **1000** dla liczb. Program powinien wyglądać jak poniżej.

```
public static void main(String[] args) {

    Thread watek1 = new Thread(() -> {
        for (char c = 'A'; c <= 'E'; c++) {
            System.out.println("Wątek 1: " + c);
            try {
                Thread.sleep(4000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    watek1.start();

    Thread watek2 = new Thread(() -> {
        for (int i = 1; i <= 17; i++) {
            System.out.println("Wątek 2: " + i);
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
    watek2.start();
}
```

Listing 5.14 Podgląd programu wykonującego współbieżnie dwa zadania z różnym czasem uśpienia

Naszym założeniem jest że oba wątki będą pracowały jednocześnie jednak z różnym opóźnieniem. Licząc ilość elementów do wyświetlenia liczb mamy **4 razy** więcej, ale za to litery mają 4 razy większe uśpienie. Oznacza to że wątki powinny zakończy swoją pracę w bardzo zbliżonym czasie. Uruchommy zatem program.



Rysunek 5.11 Wynik końcowy z konsoli

Obserwując pojawiające się wiadomości w konsoli widać że liczby pojawiają się co sekundę, natomiast litery co 4 sekundy.

5.5.2 Metoda join()

Metoda **join()** jest używana do czekania na zakończenie wykonania innego wątku. Główny lub inny wątek wykonujący wywołuje **join()** na innym wątku i zostaje wstrzymany do momentu zakończenia działania bieżącego. Jest to szczególnie przydatne, gdy konieczne jest wykonanie operacji w głównym wątku po zakończeniu działania innego.

Napiszemy program który w wątku głównym będzie wykonywał pętlę wypisującą nazwę oraz aktualną iterację. W chwili gdy iteracja dojdzie do pewnej liczby wywołana zostanie metoda **join()**, wtedy postęp zostanie

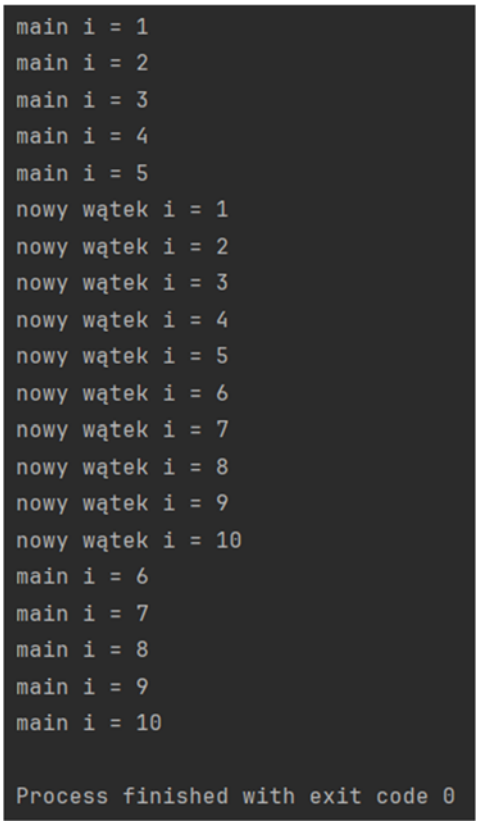
wstrzymany aż drugi wątek nie zakończy swojego działania. Zacznijmy od stworzenia wątku który będzie przechodził po pętli **for 10 razy**, a przy każdej iteracji wypisze swoją nazwę oraz numer przejścia. Dodatkowo dla większej czytelności nadajmy mu własną nazwę.

Następnie napiszmy drugą pętlę która będzie częścią wątku głównego. Również niech wykona się **10 razy** i wypisuje swoją nazwę i numer przejścia. Jednak dodamy tutaj instrukcję warunkową sprawdzającą czy **i = 5**, gdy wynik będzie prawdziwy program ma przejść do bloku **try-catch** i uruchomić stworzony wcześniej wątek oraz wstrzymać pracę dopóki nowy wątek nie zakończy swojej. Całość powinna prezentować się następująco.

```
public static void main(String[] args) {

    Thread thread1 = new Thread(()-> {
        for(int i = 1; i < 11; i++){

System.out.println(Thread.currentThread().getName() + "
i = " + i );
        }
    });

    thread1.setName("nowy wątek");

    for (int i = 1; i < 11; i++) {

System.out.println(Thread.currentThread().getName() + "
i = " + i);
        if (i == 5) {
            try {
                thread1.start();
                thread1.join();
            } catch (InterruptedException e) {
                System.out.println("Główny wątek został
przerwany!");
            }
        }
    }
}
```

Listing 5.15 Program przedstawiający działanie metody „join()”

Linijka „**thread1.start()**;” znajduje się dopiero w bloku **try-catch** aby w konsoli lepiej było widać kiedy następuje przerwanie wątku głównego. Uruchomimy program.



Rysunek 5.12 Wynik działania programu

Jak widać program rozpoczął swoje działanie, a w momencie gdy iteracja wyniosła **5**, wątek został wstrzymany. Ruszył dopiero gdy **nowy wątek** zakończył swoją pracę.

5.6 Synchronizacja

Jednym z wyzwań w programowaniu współbieżnym jest zapewnienie spójności i poprawności działania aplikacji w przypadku, gdy wiele wątków próbuje korzystać z tych współdzielonych zasobów jednocześnie. Bez odpowiednich mechanizmów synchronizacji wątków, może dochodzić do sytuacji, w której jeden wątek modyfikuje dane, podczas gdy inne próbują odczytać lub zmodyfikować te same dane, co może prowadzić do błędów

Jednym ze sposobów radzenia sobie z problemem braku synchronizacji jest wykorzystanie słowa kluczowego **synchronized** które możemy stosować na różnym zakresie kodu np. na całych metodach lub konkretnych blokach kodu. Zalecane jest aby nie przesadzać z używaniem synchronizacji kiedy nie jest to

konieczne. Jest to rozwiązanie wielu problemów w czasie tworzenia aplikacji ale jednocześnie może wygenerować inne o których powiemy jeszcze trochę później.

Spróbujmy teraz napisać program który wygeneruje nam problem z dostępem do danych przez dwa wątki jednocześnie. Następnie naprawimy to stosując wcześniej opisaną metodę.

Zacznijmy od stworzenia nowej klasy reprezentującej konto bankowe. Zadeklarujmy w niej pole typu **int** będące odpowiednikiem stanu konta. Następnie dodajmy trzy metody które kolejno będą odpowiadać za wpłatę, wypłatę oraz sprawdzenie stanu konta. Klasa powinna wyglądać jak poniżej.

```
class KontoBankowe {

    private int stanKonta;

    public void wplac(int kwota) {
        stanKonta += kwota;
    }
    public void wyplac(int kwota) {
        stanKonta -= kwota;
    }
    public int getStanKonta() {
        return stanKonta;
    }
}
```

Listing 5.16 Klasa „KontoBankowe”

Teraz napiszemy logikę w której utworzymy dwa wątki z czego jeden będzie dokonywał wpłaty a drugi wypłaty środków z konta. Ponieważ wystąpienie błędu nie zdarza się co chwilę musimy operować na wielu powtórzeniach w pętli. Załóżmy że dokonujemy wpłaty **1000 razy** i tak samo wypłaty. Na koniec wyświetlamy stan naszego konta. Kod programu może wyglądać następująco.

```
public static void main(String[] args) {

    KontoBankowe konto = new KontoBankowe();

    Thread watek1 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            konto.wplac(10);
        }
    });

    Thread watek2 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            konto.wyplac(5);
        }
    });

    watek1.start();
    watek2.start();

    try {
        watek1.join();
        watek2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Stan konta po operacjach: " +
konto.getStanKonta());
}
```

Listing 5.17 Metoda „main" z logiką obsługującą transakcje

W powyższym kodzie „**wątek1**" wykona wpłatę na konto **1000 razy** po **10** co daje nam ostateczną kwotę **10000**, natomiast „**watek2**" pobierze z konta 1000 razy po **5**, co da nam **5000**. Możemy z tego wyliczyć że po wszystkich transakcjach stan konta powinien wynosić **5000**. Uruchamiamy zatem nasz program.

```
Stan konta po operacjach: 3220

Process finished with exit code 0
```

Rysunek 5.13 Błędna wartość spowodowana brakiem synchronizacji dostępu do zasobów

Jak widzimy wartość którą zwrócił program jest inna niż zakładana. Oznacza to że w trakcie pracy doszło do błędu który spowodował naliczanie nieprawidłowych wartości. Mamy tutaj do czynienia z problemem znanym jako

wyścig wątków czyli rzeczywistą walką o dostęp do zasobów. Teraz spróbujmy zabezpieczyć naszą aplikację przed takimi sytuacjami.

Musimy zadbać żeby w trakcie dokonywania zmian przez dany wątek, żaden inny nie miał możliwości modyfikacji. W tym celu musimy wrócić do klasy **KontoBankowe** i zabezpieczyć za pomocą **synchronized** miejsca w których dochodzi do modyfikacji pola. Powinno to wyglądać jak poniżej.

```
class KontoBankowe {
    private int stanKonta;
    public void wplac(int kwota) {
        synchronized (this) {
            stanKonta += kwota;
        }
    }
    public void wyplac(int kwota) {
        synchronized (this) {
            stanKonta -= kwota;
        }
    }
    public int getStanKonta() {
        return stanKonta;
    }
}
```

Listing 5.18 Dodanie instrukcji „synchronied" do narażonych metod

Ponownie spróbujemy uruchomić aplikację.

```
Stan konta po operacjach: 5000


Process finished with exit code 0
```

Rysunek 5.14 Prawidłowy wynik programu

Teraz wartość zwrócona w konsoli odpowiada założeniom.

5.7 Klasa Lock i implementacja ReentrantLock

Zarządzanie synchronizacją dostępu do wspólnych zasobów w aplikacjach umożliwia również klasa **Lock** implementacja **ReentrantLock**, które umożliwiają kontrolowanie blokad i zapewnienie spójności danych. Poprawne użycie tych mechanizmów może znacząco zwiększyć niezawodność i wydajność aplikacji wielowątkowych, minimalizując ryzyko zakleszczeń i wyścigów wątków.

Sama budowa jest podobna do opisanej wyżej instrukcji **synchronized** dlatego posłużymy się tamtym przykładem. Wprowadzimy jedynie drobne modyfikacje.

Zaczniemy od klasy **KontoBankowe**. Musimy w niej dodać pole **private Lock lock = new ReentrantLock();** oraz zamienić wszystkie bloki **synchronized** na **try-finaly**.

```
lock.lock();
try {
    stanKonta += kwota;
} finally {
    lock.unlock();
}
```

Listing 5.19 Konstrukcja try-finally dla blokady dostępu

Będzie to powodowało zablokowanie dostępu do zmiennej **stanKonta** do czasu aż nie zostanie ono odblokowane. Podobnie robimy to dla metody **wyplac**. Całość powinna wyglądać jak poniżej.

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

class KontoBankowe {
    private int stanKonta;
    private Lock lock = new ReentrantLock();
    public void wplac(int kwota) {
        lock.lock();
        try {
            stanKonta += kwota;
        } finally {
            lock.unlock();
        }
    }
    public void wyplac(int kwota) {
        lock.lock();
        try {
            stanKonta -= kwota;
        } finally {
            lock.unlock();
        }
    }
    public int getStanKonta() {
        return stanKonta;
    }
}
```

Listing 5.20 Klasa „KontoBankowe" przebudowana pod użycie blokad

Teraz przejdźmy do klasy w której implementujemy logikę. Nie musimy jej zmieniać ponieważ będziemy wykorzystywać tę samą klasę co poprzednio i użyjemy tych samych wartości. Tak więc powinna wyglądać jak poniżej.

```
public static void main(String[] args) {

    KontoBankowe konto = new KontoBankowe();

    Thread watek1 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            konto.wplac(10);
        }
    });

    Thread watek2 = new Thread(() -> {
        for (int i = 0; i < 1000; i++) {
            konto.wyplac(5);
        }
    });

    watek1.start();
    watek2.start();

    try {
        watek1.join();
        watek2.join();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }

    System.out.println("Stan konta po operacjach: " +
konto.getStanKonta());
}
```

Listing 5.21 Metoda „main" z obsługą transakcji

Teraz uruchomimy program.

```
Stan konta po operacjach: 5000


Process finished with exit code 0
```

Rysunek 5.15 Poprawny wynik w konsoli

Jak widać program zakończył swoją pracę z prawidłowym wynikiem, co oznacza że nie doszło do kolizji w dostępie do zasobów.

5.8 Problemy synchronizacji. Deadlock i Livelock

5.8.1 Deadlock

Deadlock to sytuacja w kilka wątków nie może kontynuować swojej pracy, ponieważ każdy z nich czeka na zasoby, które są zablokowane przez inne wątki w tym samym czasie. W rezultacie żaden nie może zakończyć swojej pracy ani zwolnić zasobów, które blokują, co prowadzi do zastoju w aplikacji.

Przykładem sytuacji gdzie zachodzi **Deadlock** jest klasa poniżej.

```
public class Deadlock {
    private static final Object wartosc1 = new Object();
    private static final Object wartosc2 = new Object();

public static void main(String[] args) {

  Thread watek1 = new Thread(() -> {
   synchronized (wartosc1) {
 System.out.println("Wątek 1: Trzyma wartosc 1...");
  try { Thread.sleep(100);
}
catch(InterruptedException e) {}
System.out.println("Wątek 1: Oczekuje na wartosc 2...");
 synchronized (wartosc2) {
System.out.println("Wątek 1: Trzyma wartosc 1 i wartosc
2...");
                    }
                }
            });

Thread watek2 = new Thread(() -> {
 synchronized (wartosc2) {
System.out.println("Wątek 2: Trzyma wartosc 2...");

try { Thread.sleep(100);
}
catch (InterruptedException e) {
}
System.out.println("Wątek 2: Oczekuje na wartosc 1...");
 synchronized (wartosc1) {
System.out.println("Wątek 2: Trzyma wartosc 2 i wartosc
1...");
                    }
                }
            });

            watek1.start();
            watek2.start();
        }
    }
```

Listing 5.22 Przykład wystąpienia problemu Deadlock w programie

Jak można zauważyć każdy z wątków trzyma jedną wartość i potrzebuje chwycić drugą. Niestety nie może tego zrobić ponieważ ta jest zajęta. Program na tym etapie już nie zakończy działania.

5.8.2 Livelock:

Livelock to podobna sytuacja jak wspomniany **Deadlock**, w której wątki nie mogą kontynuować swojej pracy. Jednak w przeciwieństwie do niego, wątki są zablokowane i czekają na siebie nawzajem. **Livelock** może być trudny do

wykrycia i diagnozowania, ponieważ wątki nadal są aktywne i mogą wykonywać pewne operacje, ale nie są w stanie zakończyć swojej pracy.

Przykład wystąpienia **Livelock** można zaobserwować w przykładzie poniżej.

```
public class Livelock {
    static class Osoba {
        public synchronized void akcja(Osoba drugaOsoba)
{
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

System.out.println(Thread.currentThread().getName() + "
próbuje wykonać akcję.");
            drugaOsoba.akcja(this);
        }
    }

    public static void main(String[] args) {
        final Osoba osoba1 = new Osoba();
        final Osoba osoba2 = new Osoba();

        new Thread(() -> osoba1.akcja(osoba2), "Wątek
1").start();
        new Thread(() -> osoba2.akcja(osoba1), "Wątek
2").start();
    }
}
```

Listing 5.23 Przykład wystąpienia problemu Livelock w programie

Problem jaki pojawia się w tym przykładzie wynika z faktu że każdy z wątków chce wykonać zadanie w ramach metody **akcja**. Posiada ona oznaczenie **synchronized**, co powoduje że inne wątki nie mają do niej dostępu dopóki aktualny nie zakończy pracy. Niestety jednym z poleceń jest wywołanie tej metody na drugim wątku, co jest nie możliwe i nigdy się nie odbędzie. Oba utknęły w nieskończonej pętli.

5.9 Problem pięciu filozofów

Problem pięciu filozofów jest klasycznym przykładem problemu synchronizacji i współbieżności w informatyce. Polega on na rozważeniu sytuacji, w której pięciu filozofów siedzi wokół okrągłego stołu, a między każdymi dwoma są umieszczone widelce. Filozofowie mogą albo jeść, albo myśleć. Aby zjeść posiłek, filozof musi podnieść dwa widelce znajdujące się po jego lewej i prawej stronie. Problem polega na zaprojektowaniu algorytmu, który zapewni, że żaden

z filozofów nie będzie głodować (nie będzie mógł zjeść), jednocześnie unikając sytuacji zakleszczenia (wszyscy filozofowie trzymają dwa widelce, ale żaden nie może zacząć jeść).

Rozwiązanie problemu pięciu filozofów często opiera się na zastosowaniu algorytmu hierarchii zasobów lub asymetrycznych widelców. W przypadku algorytmu hierarchii zasobów filozofowie podnoszą najpierw widelec o niższym numerze, a dopiero potem o wyższym. Natomiast w przypadku asymetrycznych widelców jeden z filozofów podnosi najpierw lewy widelec, a dopiero potem prawy.

Przykładowe rozwiązanie znajduje się poniżej:

```
public class Filozof {
    public static void main(String[] args) {
        Object[] widelce = new Object[5];

        for (int i = 0; i < 5; i++) {
            widelce[i] = new Object();
        }

        for (int i = 0; i < 5; i++) {
            final int numer = i;
            Thread filozof = new Thread(() -> {
                while (true) {
                    try {
                        System.out.println("Filozof " +
numer + " myśli");
                        Thread.sleep((long)
(Math.random() * 1000));
                        synchronized (widelce[numer]) {
                            synchronized (widelce[(numer
+ 1) % 5]) {

System.out.println("Filozof " + numer + " je");
                                Thread.sleep((long)
(Math.random() * 1000));
                            }
                        }
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            });
            filozof.start();
        }
```

Listing 5.24 Przykład rozwiązania problemu pięciu filozofów

W tym konkretnym przypadku rozwiązanie oparte jest na hierarchii - każdy filozof próbuje podnieść najpierw lewy widelec, a następnie prawy. Jest to prostsze rozwiązanie niż asynchroniczne widelce, ponieważ nie wymaga

dodatkowej koordynacji między wątkami ani zastosowania dodatkowych mechanizmów synchronizacji. Każdy filozof działa niezależnie od pozostałych, próbując podnieść swoje widelce w ustalonej kolejności. Dzięki temu rozwiązaniu unikamy zakleszczeń i zapewniamy, że każdy filozof będzie mógł zjeść, gdy tylko oba widelce będą dla niego dostępne.

Podsumowując problem pięciu filozofów jest ważnym zagadnieniem w dziedzinie informatyki, ponieważ ilustruje wyzwania związane z synchronizacją i współbieżnością w systemach wieloprocesorowych.

5.10 Dobre praktyki przy programowaniu współbieżnym

Programowanie współbieżne może być wyzwaniem, ale stosowanie pewnych praktyk może zminimalizować ryzyko błędów. Oto kilka wskazówek przydatnych przy programowaniu współbieżnym:

1. **Minimalizacja współbieżności**: Jeśli to możliwe, unikaj stosowania wielowątkowości w miejscach, gdzie nie jest to konieczne. Staraj się ograniczyć je do tych fragmentów kodu, które faktycznie wymagają równoczesnego wykonania.
2. **Unikanie współdzielonych zmiennych**: Współbieżność może prowadzić do problemów z synchronizacją, dlatego należy minimalizować ich użycie.
3. **Synchronizacja dostępu do współdzielonych zasobów**: Jeśli już konieczne jest używanie współdzielonych zmiennych, upewnij się, że operacje na nich są odpowiednio synchronizowane za pomocą mechanizmów takich jak **synchronized** czy klasy **Lock**.
4. **Unikanie blokowania**: Staraj się unikać długotrwałych blokad, które mogą spowodować opóźnienia i potencjalnie prowadzić do wstrzymania aplikacji (**deadlock**).
5. **Testowanie i debugowanie**: Programowanie współbieżne może być trudne do testowania ze względu na nieprzewidywalne zachowanie dlatego należy testować zwracane wyniki i udoskonalać kod.
6. **Monitorowanie i optymalizacja wydajności**: Regularnie monitoruj aplikację pod kątem wydajności i identyfikuj miejsca, w których występują problemy związane z wielowątkowością. Optymalizuj kod i zasoby w tych obszarach, aby zapewnić jak najlepszą wydajność.

7. **Dokumentacja**: Dokumentuj sposób synchronizacji i zarządzania współbieżnością w kodzie, abyś inni mogli łatwo zrozumieć jego zachowanie. Komunikuj również wyraźnie wszelkie założenia dotyczące współbieżności w kodzie.

Stosowanie tych praktyk może pomóc w zminimalizowaniu problemów związanych z programowaniem współbieżnym i zapewnieniu stabilności oraz wydajności aplikacji działających w środowisku wielowątkowym.

5.11 Sprawdź się!

1) Co to jest wątek (thread) w języku Java?

a) Niezależna sekwencja instrukcji wykonywana równolegle z innymi sekwencjami instrukcji w programie.

b) Specjalna klasa, która reprezentuje jednostkę pracy w programie.

c) Kolekcja obiektów, która zarządza wykonaniem różnych części programu w określonym czasie.

2) Co to jest synchronizacja wątków i dlaczego jest ważna?

a) Synchronizacja wątków polega na równoczesnym uruchamianiu wielu wątków.

b) Synchronizacja wątków polega na kontrolowaniu dostępu wielu wątków do współdzielonych zasobów, zapewniając bezpieczeństwo operacji na nich.

c) Synchronizacja wątków polega na ograniczaniu liczby dostępnych wątków w programie.

3) Jakie jest główne zastosowanie metody sleep() w języku Java?

a) Metoda sleep() służy do zatrzymywania wątku na określony czas.

b) Metoda sleep() służy do usypiania wszystkich wątków w programie.

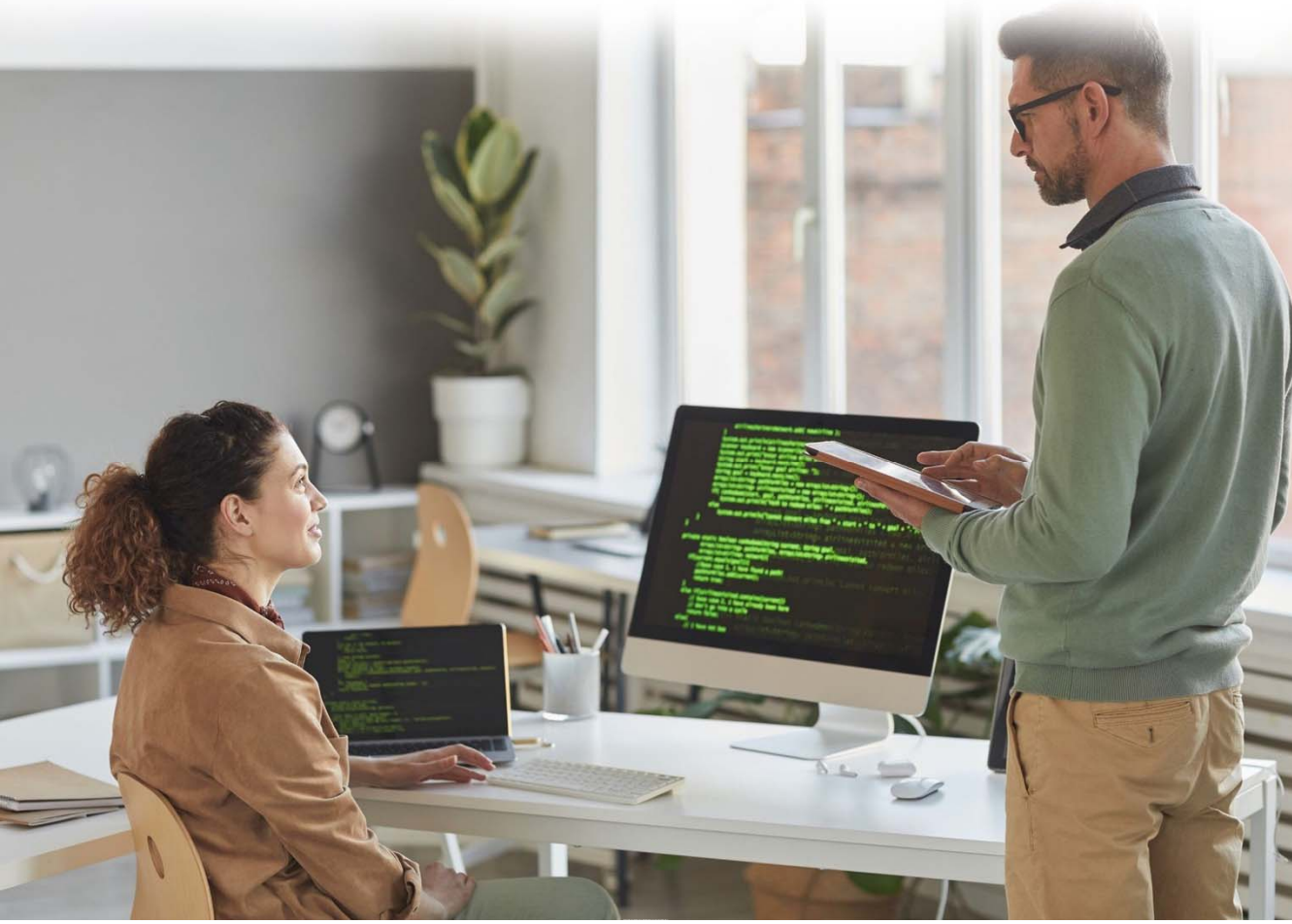
c) Metoda sleep() służy do usuwania wątków z pamięci.

Zadanie 5.1

Napisz program, który tworzy dwa wątki. Pierwszy dodaje liczby parzyste od **1** do **100**, a drugi nieparzyste od **1** do **100**. Każdy wątek powinien działać równolegle, a ich wyniki (sumy liczb parzystych i sumy liczb nieparzystych) powinny być wyświetlone na zakończenie pracy wątków.

ROZDZIAŁ 6

APLIKACJE MOBILNE



6 Aplikacje mobilne

6.1 Wstęp do aplikacji mobilnych

6.1.1 Aplikacje mobilne

Aplikacja mobilna jest oprogramowaniem, które działa na urządzeniach przenośnych. Urządzeniami takimi są m.in.: smartfony czy tablety.

Początkowo aplikacje mobilne pełniły funkcje typowo użytkowe. Były to przede wszystkim aplikacje służące do przeglądania stron internetowych (przeglądarki internetowe), aplikacje do obsługi poczty elektronicznej, kalendarz, aplikacje do odtwarzania filmów bądź plików muzycznych.

Dynamiczny rozwój urządzeń przenośnych: smartfonów i tabletów spowodował rozwój również aplikacji mobilnych.

Do najbardziej popularnych aplikacji należą aplikacje obsługujące bankowość mobilną, aplikacje popularnych sklepów sieciowych zastępujące popularne karty lojalnościowe, aplikacje oferujące kupony rabatowe w restauracjach, gry, komunikatory.

Większość portali społecznościowych posiada odpowiedniki aplikacji internetowych w postaci aplikacji mobilnych.

Za pierwszy telefon komórkowy posiadający wbudowaną aplikację uznaje się **Psion Organizer** stworzony przez brytyjską firmę Psion w 1984 roku. Ten kieszonkowy komputer opary był na 8-bitowym procesorze Hitachi 6301, posiadał pamięć operacyjną 2 kB. Jako pierwsze urządzenie w historii opierał się na bazie danych, był wyposażony w aplikację kalkulator i zegar.

Pierwszym mobilnym systemem operacyjnym był **Symbian**, który zadebiutował w 2001 roku. Powstał z inicjatywy firmy Nokia.

W dzisiejszych czasach aplikacje mobilne są łatwo dostępne, można je pobrać ze sklepów **Google Play** – w przypadku systemu operacyjnego Android, oraz **App Store** dla telefonów z firmy Apple. W większości są bezpłatne. Część z nich wymaga założenia konta.

6.1.2 System operacyjny Android

Android jest systemem operacyjnym dedykowany aplikacjom mobilnym. System został stworzony przez amerykański startup – **Android Inc**. Trzej założyciele tej spółki intensywnie pracowali nad systemem operacyjnym, który byłby konkurencją dla systemu Symbian.

Firma w lipcu 2005 roku została kupiona przez giganta Google za 50 milionów dolarów.

Android oparty jest na jądrze systemu Unix. W większości dostępny na licencji GNU GPL. Jest systemem otwartym czyli takim który nie jest dedykowany konkretnym urządzeniom (jak iPhone).

Wszystkie nazwy systemu Android zaczynają się od kolejnych liter alfabetu i są nazwami słodyczy.

Pierwszym telefonem z systemem Android był HTC Dream. Obecnie system Android wykorzystywany jest do tworzenia aplikacji nie tylko na smartfony czy tablety, ale również na telewizory i smartwatche.

6.2 Android Studio

Android Studio jest oficjalnym środowiskiem programistycznym do tworzenia aplikacji mobilnych na system operacyjny Android. Środowisko to zostało zaprezentowane w 2013 roku i zastąpiło najpopularniejsze na tamte czasy środowisko Eclipse. Android Studio zostało zbudowane na podstawie komercyjnego oprogramowania IntelliJ IDEA stworzonego przez firmę JetBrains. Jest oprogramowaniem dostępnym na licencji Apache License, w związku z tym jest wolnym oprogramowaniem, które bez konsekwencji można pobrać z Internetu i zainstalować na własnym komputerze.

Środowisko daje możliwość tworzenia oprogramowania w języku Java, C++, oraz Kotlin i może zostać zainstalowane na systemach operacyjnych Windows, macOS, a także Linuxie.

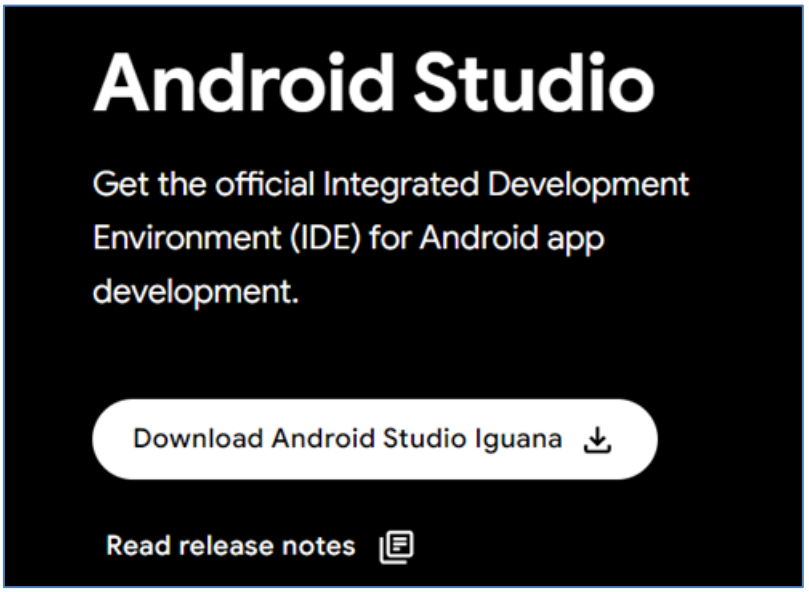
W chwili pisania tej książki (kwiecień 2024 roku) aktualną wersją Android Studio jest Iguana o numerze 2023.1.2, które wydano w marcu 2024 roku.

6.2.1 Instalacja Android Studio

Android Studio można pobrać ze strony:

https://developer.android.com/studio

Klikamy po porostu w przycisk **Download Android Studio Iguana**, co pokazuje poniższy rysunek.



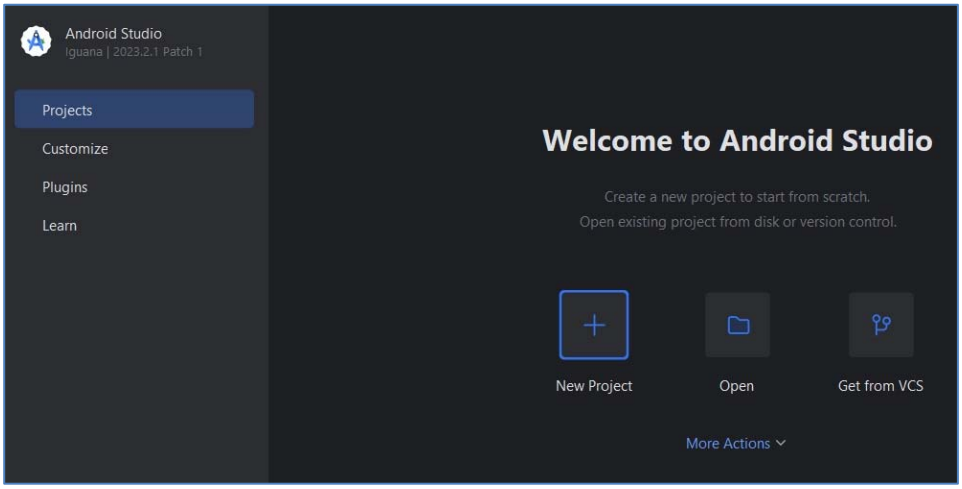
Rysunek 6.1 Oficjalna strona Android Developers

W oknie pojawia się umowa licencyjna którą musi zaakceptować, następnie zaznaczamy okienko, i klikamy przycisk odpowiedzialny za pobieranie oprogramowania.

Pakiet po pobraniu należy zainstalować. Zanim jednak zainstalujemy Android Studio należy przed tym zainstalować pakiet Java. Można go bezpiecznie pobrać ze strony: **https://www.oracle.com/pl/java/technologies/downloads/#jdk19-windows**. Po uruchomieniu Android Studio zobaczymy ekran startowy, jak na poniższym rysunku.

Rysunek 6.2 Ekran startowy aplikacji Android Studio

Po załadowaniu aplikacji otrzymujemy widok jak na poniższym rysunku.



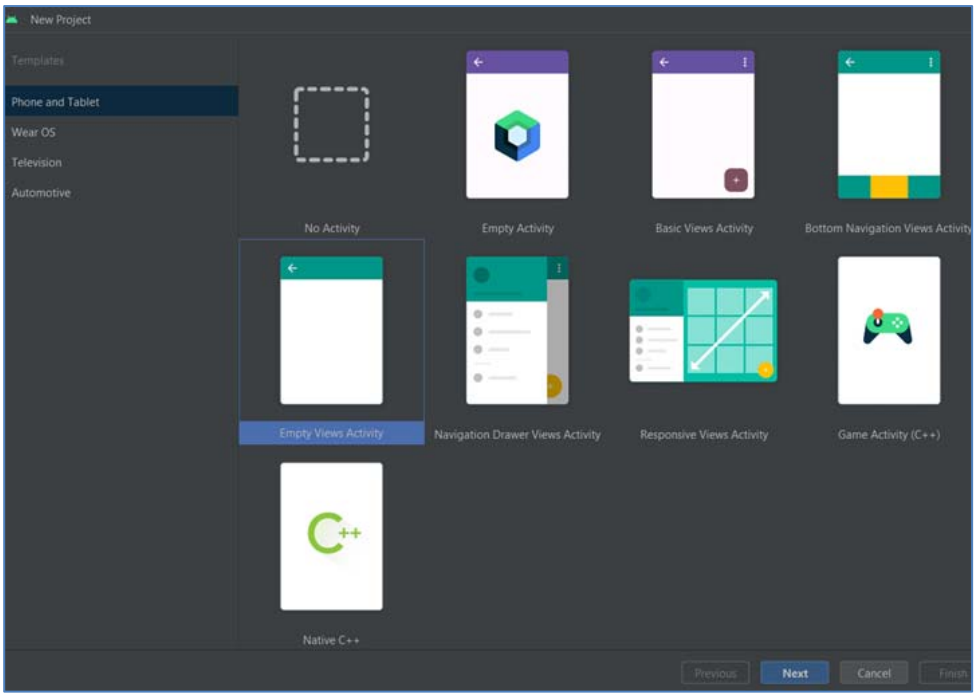
Rysunek 6.3 Okno wyboru aplikacji w Android Studio

Mamy możliwość załadowania jednej z ostatnio edytowanych aplikacji. Możemy utworzyć nowy projekt używając przycisku **New Project**, ewentualnie otworzyć aplikacje zapisaną na Dysku za pomocą **Open**.

6.2.2 Tworzenie pierwszej aplikacji

Na początek utworzymy prosty projekt. W tym celu klikamy **New Project**. Widzimy okno wyboru aktywności. Po lewej stronie w oknie **Templates** możemy zdecydować na jaki rodzaj urządzenia chcemy utworzyć aplikację. **Android Studio** pozwala na tworzenie aplikacji na takie urządzenia jak: tablety

i telefony - **Phone and Tablet**, smartwatche - **Wear OS**, telewizory- **Android TV**, panele w samochodach- **Automotive**. Po prawej stronie okna wybieramy rodzaj aktywności. Na początek będzie to **Empty Views Activity** i taką opcję zaznaczamy. Następnie klikamy przycisk **Next**.



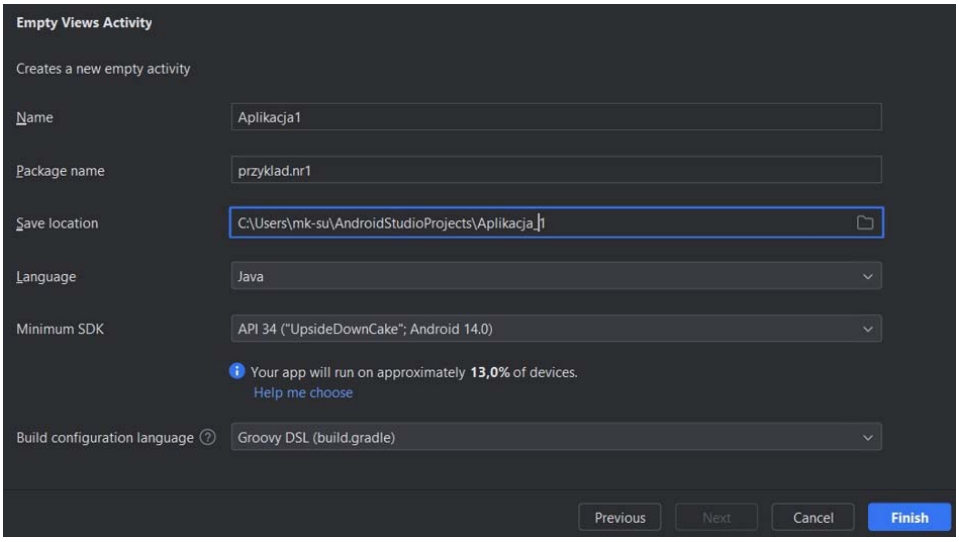
Rysunek 6.4 Okno wyboru aktywności w Android Studio

Na ekranie pojawia się okno tworzenia aplikacji. Uzupełniamy w nim następujące pola:

- **Name** (nazwa) – podajemy nazwę Aplikacji – np. Aplikacja1;
- **Package** name (nazwa pakietu) – podajemy nazwę pakietu w której będzie umieszczona aplikacja – np. przykład.nr1;
- **Save location** (lokalizacja aplikacji) – wybieramy miejsce na dysku w którym zapisujemy aplikacje. Android Studio tworzy folder o nazwie AndroidStudioProject i tam można umieszczać wszystkie aplikacje;
- **Language** (język) – wybieramy język w którym chcemy tworzyć aplikację. Do wyboru mamy Java i Kotlin. Na początku wybieramy język programowania Java.

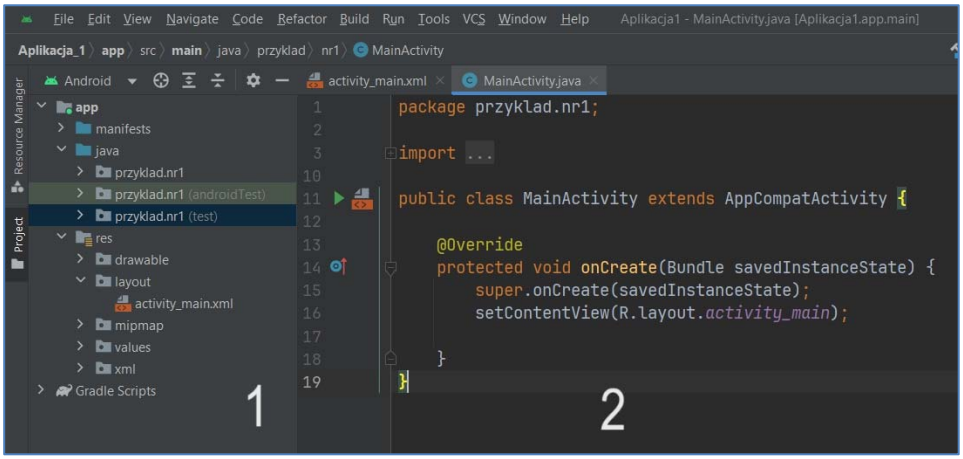
Minimum SDK (minimalny poziom API Android) – wybieramy w której wersji Androida będzie pisana nasza aplikacja. Według najnowszych wytycznych Androida aplikacje tworzymy w najwyższej wersji API, ponieważ takie aplikacje będą bez problemu działać na urządzeniach o starszej wersji Androida.

Po uzupełnieniu okna tworzenia aplikacji klikamy **Finish** (zakończ). Następuje załadowanie elementów aplikacji.



Rysunek 6.5 Okno tworzenia nowej aplikacji

Po chwili na ekranie pojawia się okno podobne do poniższego rysunku.



Rysunek 6.6 Okno główne Android Studio

W polu oznaczonym **nr 1** znajduje się Drzewo katalogów aplikacji. Aplikacja w systemie Android jest złożona z kilkunastu plików. Wszystkie pliki mają

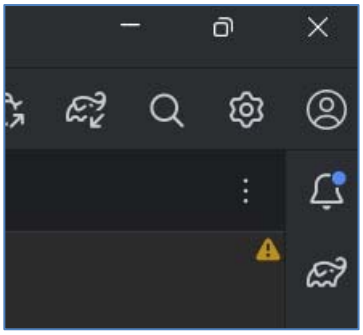
swoje unikalne znaczenie. Ich zastosowanie zostanie omówione w dalszej części książki

W polu oznaczonym **nr 2** znajduje się okno do edycji kodu. To w tym oknie będziemy programować naszą aplikację.

Widoczny projekt składa się z dwóch okien. Prezentują się one w formie zakładek umieszczonych na górze okna.

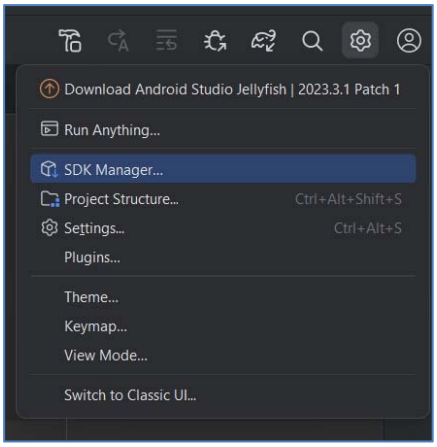
6.2.3 Zmiana wyglądu GUI

W Android Studio istnieje możliwość zmiany interfejsu użytkownika. Po pierwsze mamy do wyboru starszą wersję GUI, oraz nowsze bardziej nowoczesne. Jeżeli po instalacji nie wyświetla się nowsza wersją interfejsu można ją samodzielnie zmienić. W tym celu klikamy ikonę koła zębatego znajdującą się w górnym prawym rogu aplikacji.



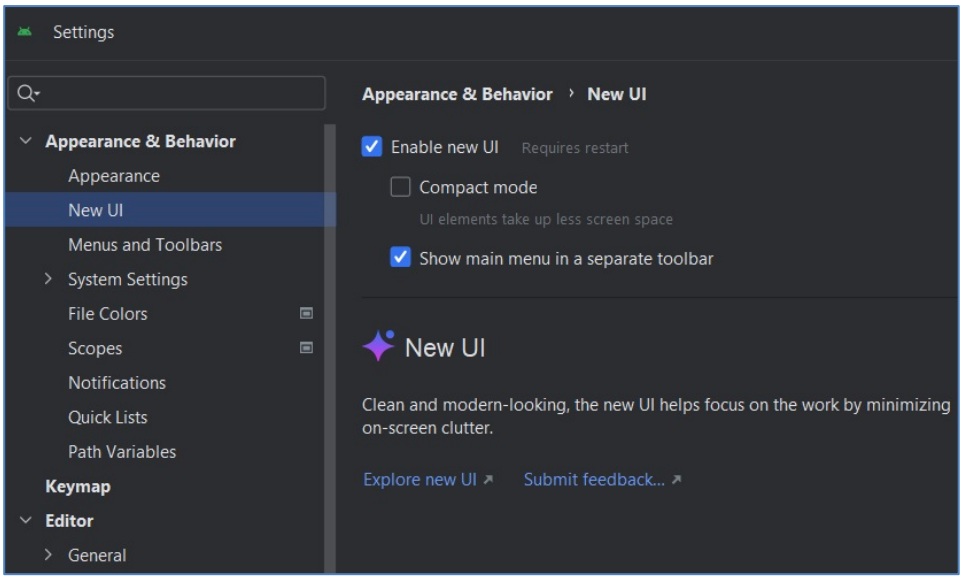
Rysunek 6.7 Ikona SDK Manager

Z rozwijanego menu wybieramy opcję: **SDK Manager.**



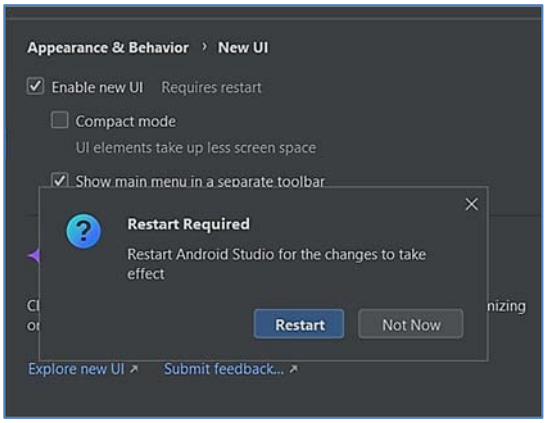
Rysunek 6.8 Menu podręczne – podstawowe operacje w Android Studio

Przechodzimy do zakładki **Appearance & Behavior** i zaznaczamy **New UI**. W następnym kroku zaznaczmy pole **Enable New UI** i klikamy przycisk **OK**.



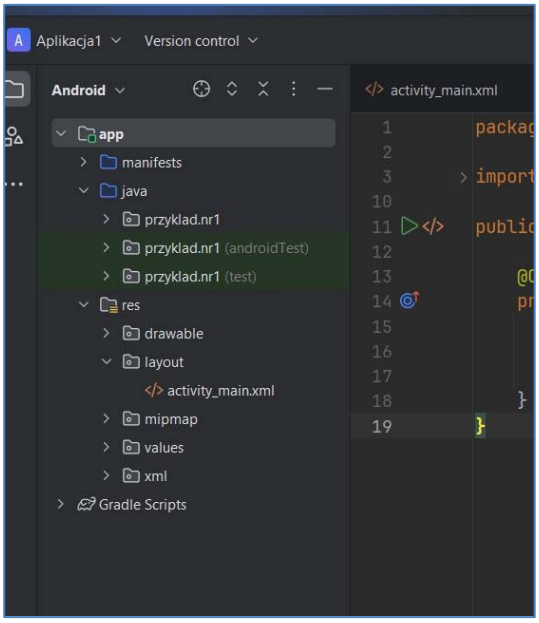
Rysunek 6.9 Okno Settings

Program zapyta czy chcemy, aby zrestartować Android Studio. Klikamy Restart



Rysunek 6.10 Restart Required: Potwierdzenie Restartu Android Studio

Program zamknie się i otworzy ponownie z nowym UI. Jego wygląd prezentuje poniższy rysunek.



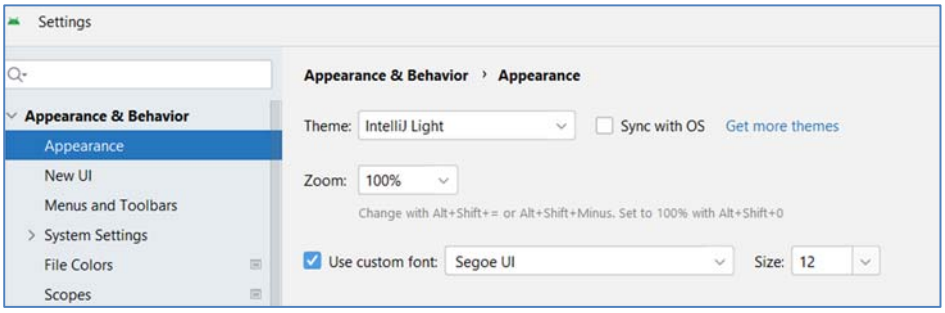
Rysunek 6.11 Okno główne Android Studio po zastosowaniu nowego UI

Przełączenie się na starszą wersję GUI jest operacją odwrotną. Przechodzimy do zakładki **Appearance & Behavior** i zaznaczamy **New UI**. Odznaczmy okienko **Enable New Gui** i klikamy przycisk **OK**. Program ponownie spyta nas czy chcemy go zrestartować?

6.2.4 Zmiana motywu aplikacji

Android Studio umożliwia ustawienie ciemnego i jasnego motywu. Zmienimy teraz motyw na jasny.

Z menu **New UI** przechodzimy do **Appearance**. W zakładce tej mamy możliwość wybrania motywu, a także ustawienie innych elementów takich jak: rozmiar i rodzaj wyświetlanej czcionki.



Rysunek 6.12 Ustawienie jasnego motywu SDK

W książce będziemy wykorzystywać motyw: **IntelliJ Light**. Z rozwijanego menu wybieramy taką opcję i klikamy **OK**. Android Studio będzie odtąd wyglądać tak jak na poniższym rysunku:

Rysunek 6.13 Okno Android Studio z zastosowaniem jasnego motywu

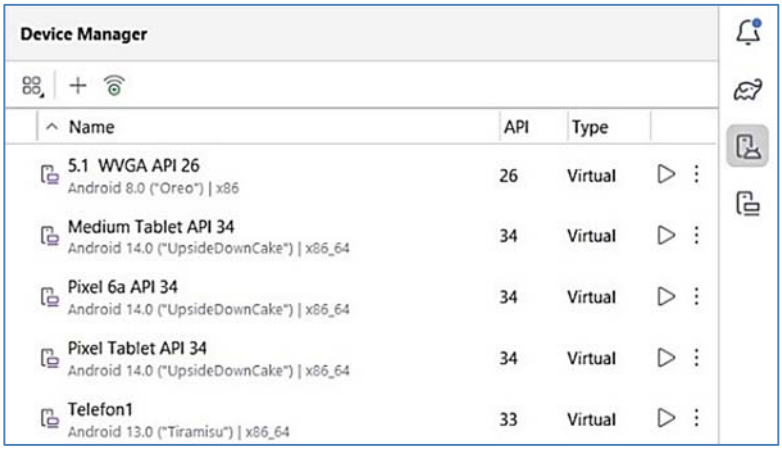
6.2.5 Emulator

Po utworzeniu aplikacji każdy programista chciałby zobaczyć efekt działania swojej pracy. Większość języków programowania posiada swoje środowisko programistyczne czyli aplikację, która umożliwia tworzenie, edycję kodu, kompilację a także uruchamianie programu.

W Android Studio do uruchomienia aplikacji będziemy potrzebować narzędzia zwanego **Emulatorem**.

Emulator jest to wirtualny system operacyjny, który umożliwia zobaczenie aplikacji tak jakby była uruchomiona na prawdziwym urządzeniu bez konieczności podłączania telefonu. Można również podłączyć własny telefon i zobaczyć na nim działanie aplikacji. Jak to zrobić omówię w następnym kroku. Póki co skupmy się na emulatorze.

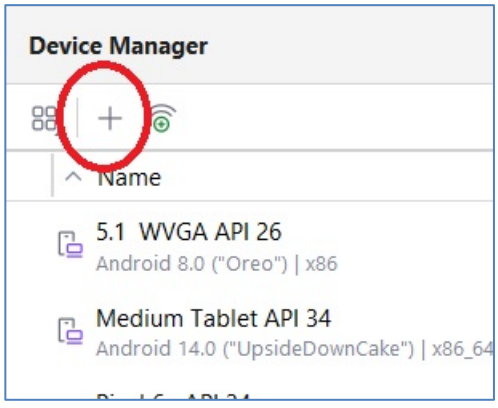
Aby korzystać z emulatora trzeba go najpierw uruchomić i podłączyć do projektu. Po prawej stronie **SDK** mamy zakładkę o nazwie **Device Manager**. Kliknięcie jej otworzy **Menedżera Urządzeń** gdzie widoczne są wszystkie uruchomione Emulatory. W tym miejscu możemy również dodać kolejny z nich.



Rysunek 6.14 Okno Device Manager – lista zainstalowanych Emulatorów

Na rysunku widzimy listę wszystkich zainstalowanych Emulatorów. W pierwszej kolumnie widnieje ich nazwa. W kolumnie **API** widzimy jaki poziom ma wybrany Emulator. Ikona trójkąta pozwala na uruchomienie wybranego Systemu.

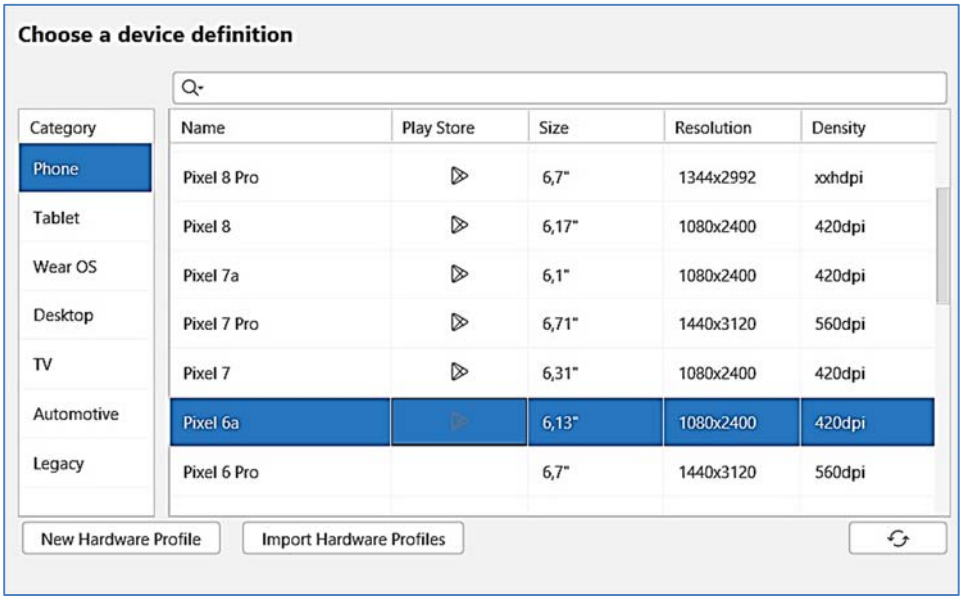
Tworzenie nowego środowiska zaczniemy od kliknięcia przycisku **Create Virtual Device** – który ma kształt znaku plus.



Rysunek 6.15 Przycisk umożliwiający dodanie nowego Emulatora

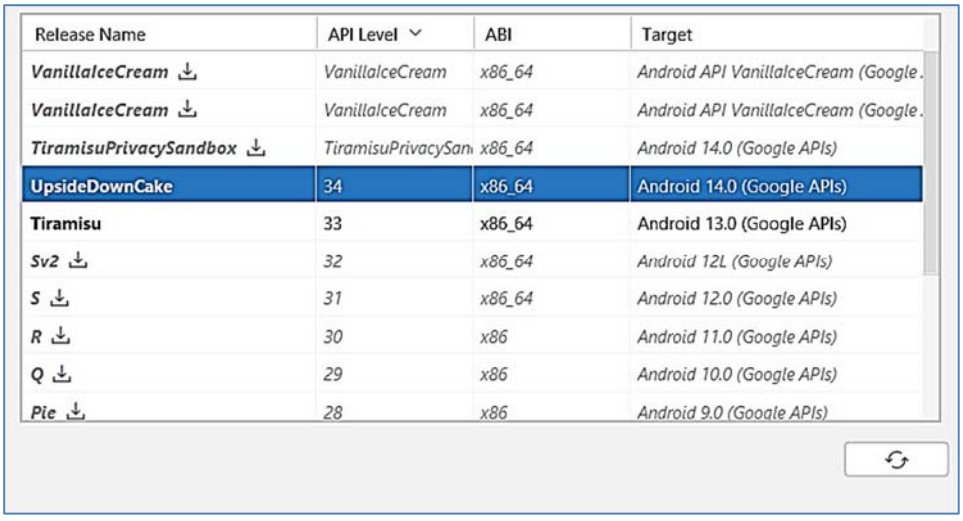
Po wciśnięciu tego przycisku pojawia nam się okno wyboru urządzenia. Android Studio daje możliwość przygotowania aplikacji na różne urządzenia: m.in.: na telefony, tablety czy telewizory. Na początek skupimy się na przygotowaniu aplikacji na telefon.

Wybieramy kategorie **Phone**. Z listy urządzeń wybieramy np. **Pixel 6a**. Zaznaczamy nazwę i klikamy przycisk **Next**.

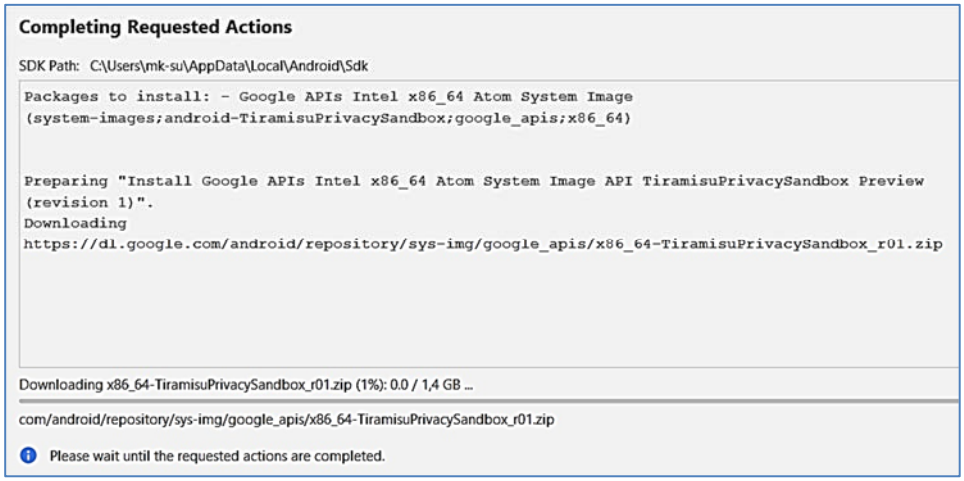


Rysunek 6.16 Okno wybory urządzenia

W następnym oknie **System Image** wybieramy poziom API czyli wersję systemu. Jeżeli wcześniej nie tworzyliśmy Emulatora w konkretnej wersji należy najpierw ściągnąć odpowiednie pakiety. Klikamy przycisk **małej strzałki** i czekamy, aż w następnym oknie zostaną pobrane potrzebne pliki.



Rysunek 6.17 Okno wyboru poziomu API

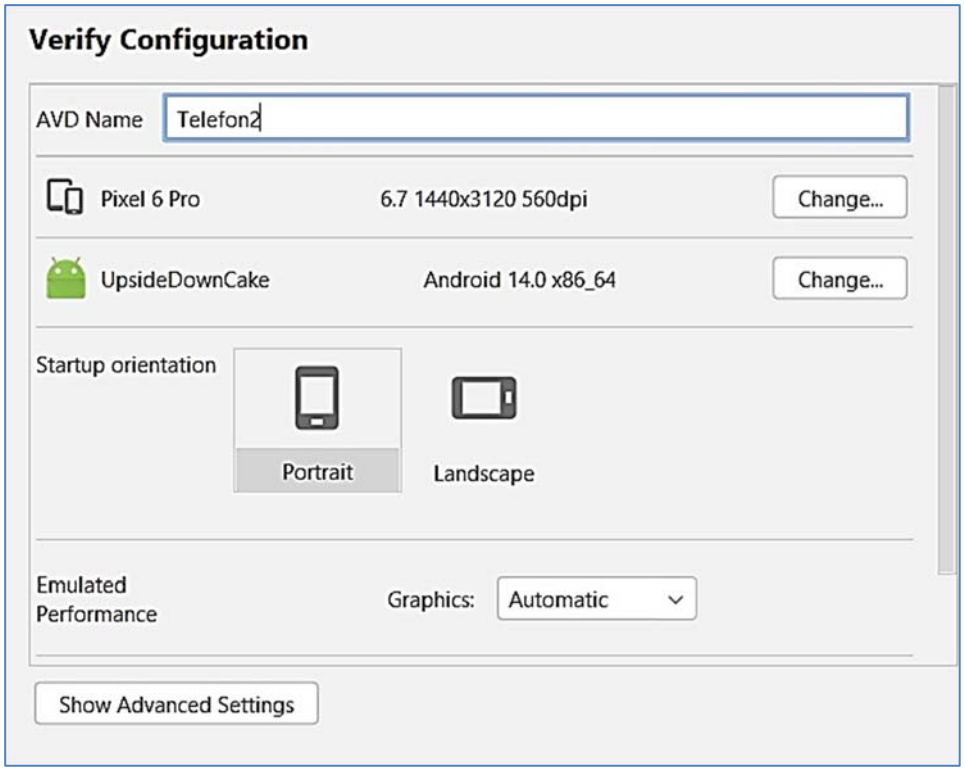


Rysunek 6.18 Pobieranie plików instalacyjnych API 34

Po załadowaniu pakietów klikamy przycisk **Finish**.

Wracamy z powrotem do okna **System Image**. Tam klikamy przycisk Next.

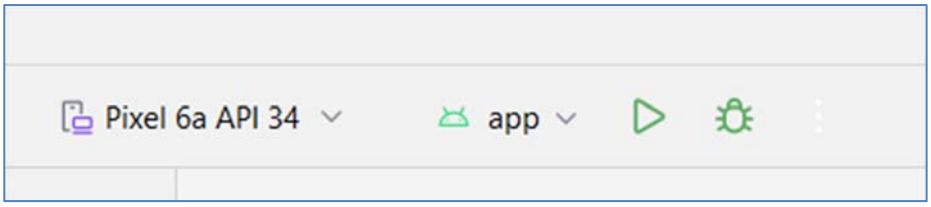
Otworzymy kolejne okno **Verify Configuration.**



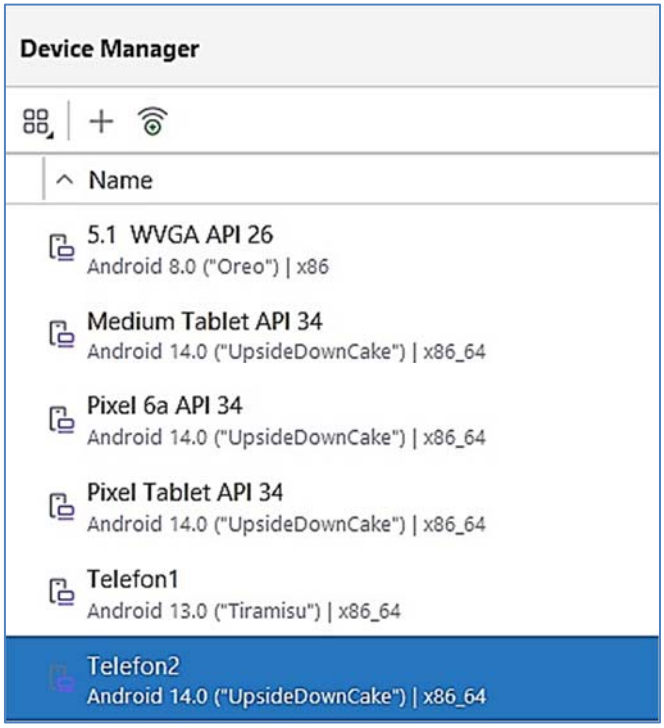
Rysunek 6.19 Edycja ustawień nowego urządzenia wirtualnego – Emulatora

Możemy tutaj zmienić nazwę urządzenia wpisując ją w polu **AVD Name**. Kliknięcie przycisku **Finish** spowoduje utworzenie Emulatora.

Po wykonaniu tych wszystkich czynności możemy uruchomić aplikację. Klikamy symbol zielonego trójkąta.

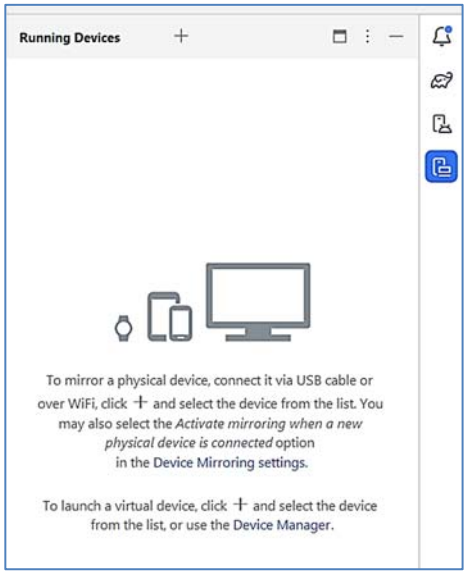


Rysunek 6.20 Ikona uruchamiająca działanie



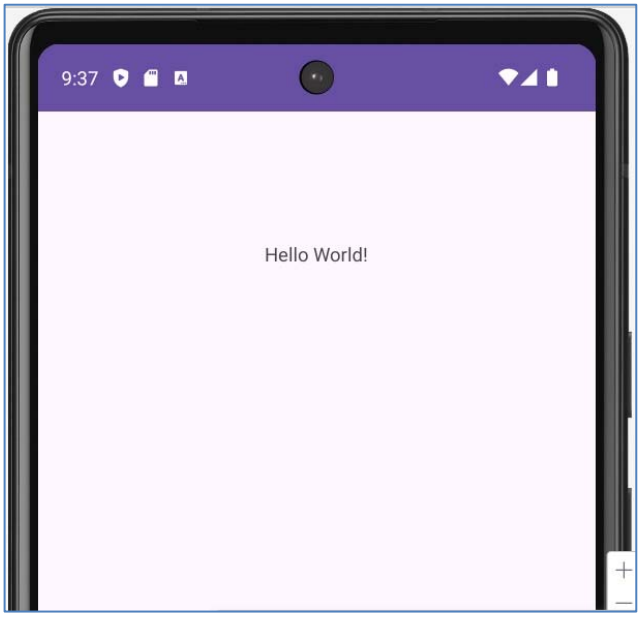
Rysunek 6.21 Okno Device Manager z zainstalowanym nowym Emulatorem

Aby zobaczyć efekt symulacji Emulatora wybieramy zakładkę **Runing Devices** , którą znajdziemy po prawej stronie panelu głównego.



Rysunek 6.22 Okno Running Devices.

Kod aplikacji znajdziesz w **Folderze Aplikacja1**. Pamiętaj, aby wszystkie pliki aplikacji umieścić w katalogu Android Studio znajdującym się w folderze utworzonym przez aplikację.



Rysunek 6.23 Emulator po uruchomieniu domyślnej aplikacji

6.2.6 Testowanie aplikacji na fizycznym urządzeniu.

Aplikacje tworzone poprzez **Android Studio** można również uruchamiać na fizycznym telefonie. IDE instaluje wówczas aplikacje na telefonie podłączonym do systemu. Aplikacje uruchamiamy bezpośrednio na urządzeniu.

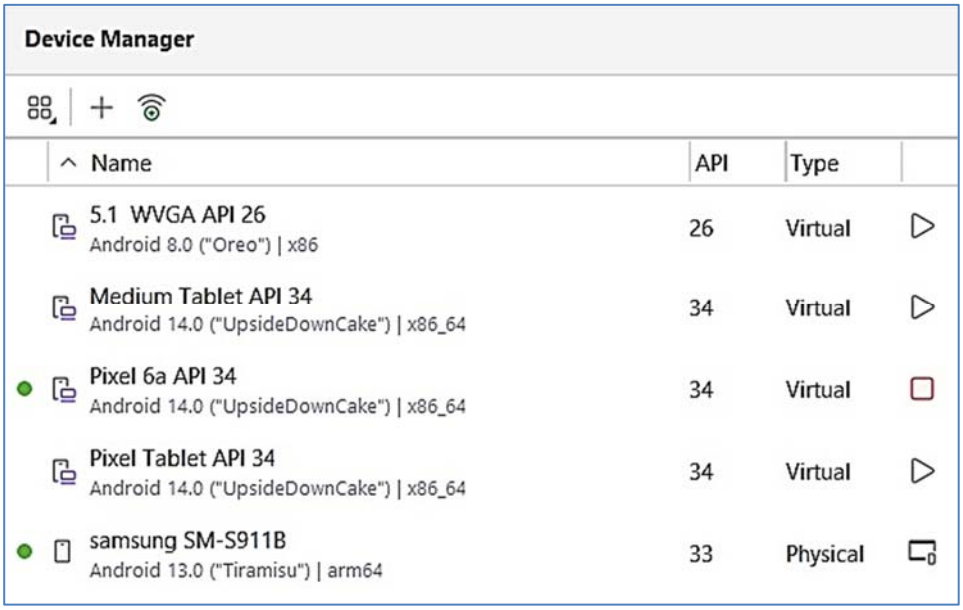
Jeżeli masz pod ręką telefon z systemem operacyjnym Android możesz podłączyć go do Android Studio.

Pierwszą czynnością jaką należy zrobić to przygotowanie telefonu. Urządzenie, które chcemy podłączyć pod Android Studio musi być ustawione na tryb programisty lub opcje programisty. Informacje na ten temat można znaleźć na stronie producenta twojego smartfonu.

Jeżeli posiadasz telefon marki Samsung możesz wykonać poniższe czynności

1. Otwórz ustawienia.
2. Przejdź do sekcji **Telefon – informacje**. Otwórz je kliknięciem.
3. Kliknij w kartę Informacje o oprogramowaniu.
4. Znajdź sekcję: **Numer wersji** i kliknij tą opcję siedem razy.
5. Na ekranie pojawi się informacja, że telefon został uruchomiony w trybie programisty.
6. Podłącz telefon do komputera za pomocą kabla USB.
7. Wyszukaj w telefonie ustawień: **Opcje programisty**.
8. Przejdź do sekcji **Debugowanie** i przesuń przycisk **Switch: Debugowanie USB**.
9. Telefon poprosi o pozwolenie.
10. **Zezwól** na korzystanie z Debugowania USB.

Następne czynności wykonujemy już w Android Studio. Przechodzimy do karty **Device Manager**, która znajduje się po prawej stronie okna. Na liście urządzeń powinien pojawić się nasz telefon. Wyróżnia się on na tle innych emulatorów, gdyż w polu **Type** jest ustawiona wartość **Physical**.



Rysunek 6.24 Okno Device Manager z zainstalowanym fizycznym urządzeniem

W jaki sposób uruchomić aplikację na telefonie? Po prostu przejdź do aplikacji którą chcesz uruchomić. W oknie wyboru urządzenia wybierz swój telefon.

Kliknij ikonę **zielonej strzałki** znajdującą się obok. Android Studio zainstaluje aplikacje na Twoim telefonie. Jeżeli chcemy otworzyć aplikację ponownie wystarczy kliknąć **jej ikonę** na **fizycznym urządzeniu**. Można to zrobić również wówczas, gdy telefon nie jest podłączony do komputera. Aplikacja została zainstalowana na telefonie i pozostanie tam aż do jej fizycznego usunięcia przez użytkownika. Jeżeli chcesz wyłączyć telefon z trybu programisty wystarczy przesunąć przycisk **Switch** w oknie **Opcje programisty**.

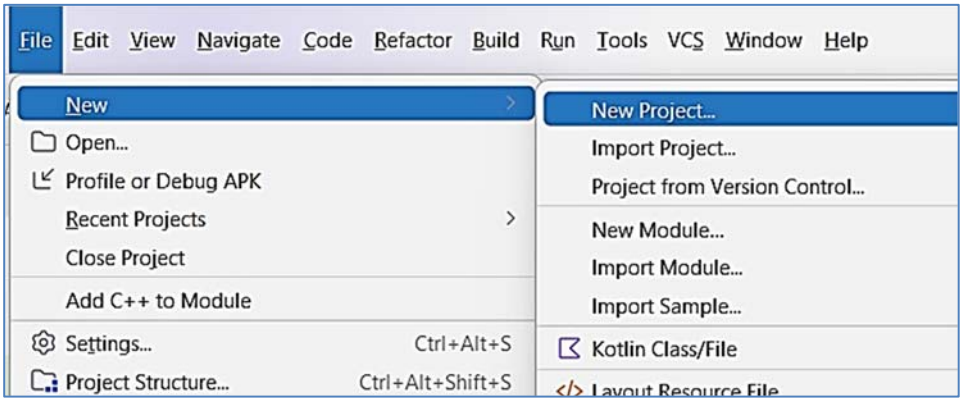
6.3 Tworzenie aplikacji

Przejdziemy teraz do tworzenia własnej aplikacji.

Na początek utworzymy prostą aplikację zawierającą **tekst** i pojedynczy przycisk tzw. **Button**

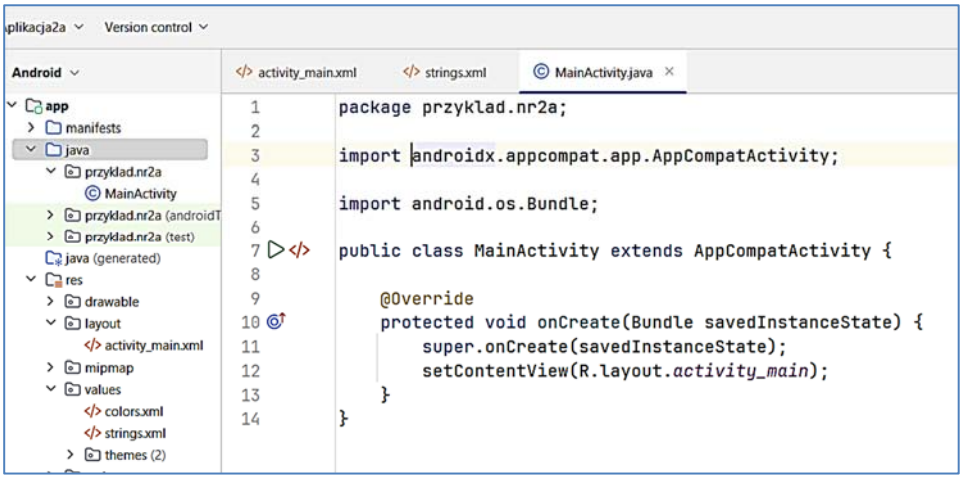
6.3.1 Tworzenie aplikacji z poziomu otwartego okna

Tworzenie nowej aplikacji rozpoczynamy od otwarcia **nowego projektu**. W lewym górnym rogu okna wybieramy zakładkę **File**, a następnie: **New** i **New Project**.



Rysunek 6.25 Tworzenie nowego projektu z poziomu Aplikacji

Jako szablon aplikacji wybieramy **Empty Views Activity**. Jeżeli w zadaniu nie podano innego szablonu zawsze będziemy wybierać właśnie ten. Wypełniamy okno tworzenia aplikacji i klikamy **Finish**.



Rysunek 6.26 Widok nowej (pustej) aplikacji po utworzeniu nowego projektu

W środku okna widzimy edytor pliku aktywności **MainActivity.java**. W drugiej zakładce znajduje się edytor graficzny pliku układu.

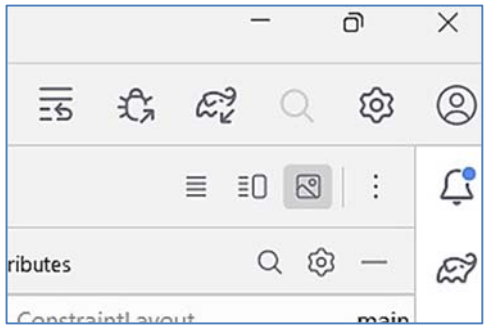
Układ aplikacji to rozmieszczenie elementów umieszczonych w aplikacji. Układ widzi to, co użytkownik uruchamiający aplikację.

Układ w aplikacjach mobilnych tworzymy w języku **xml**, który jest nieco zbliżony do języka **CSS**. Edytować układ możemy na dwa sposoby:

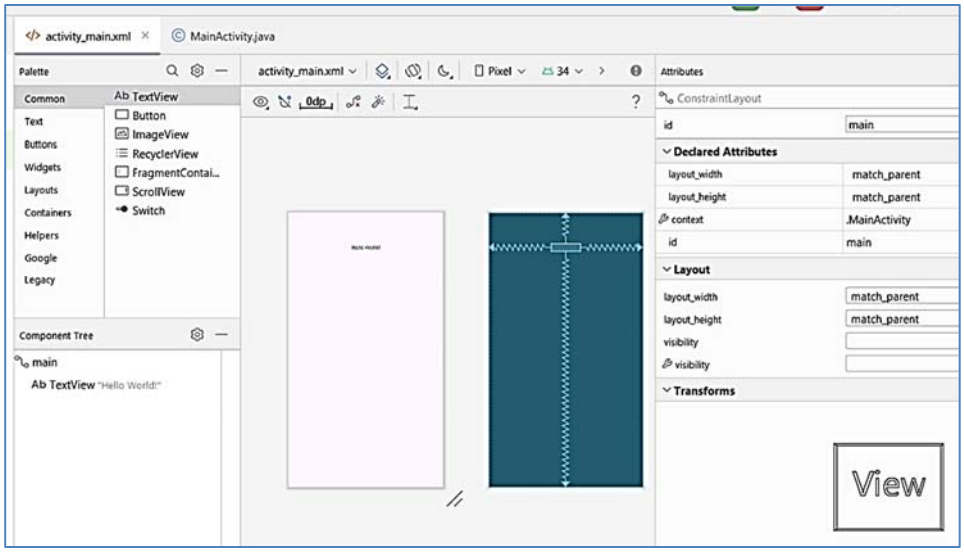
- **Graficznie** – przeciągając, rozmieszczając bądź powiększając elementy za pomocą myszki.
- **Tekstowo** – wpisując odpowiednie polecenia języka xml w edytorze tekstowym.

Pomiędzy widokami możemy przemieszczać się klikając odpowiednią kartę.

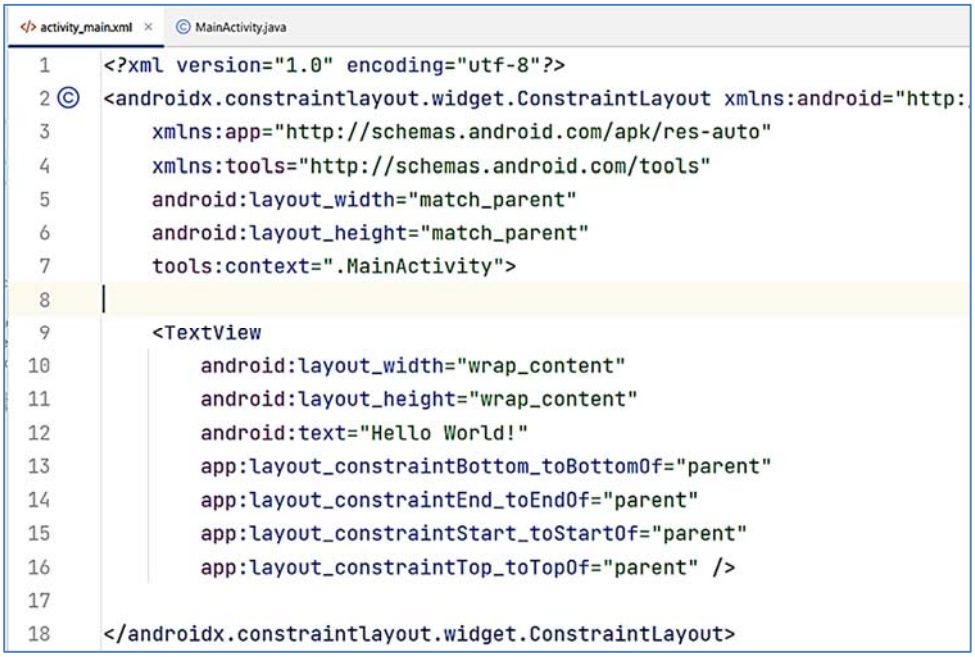
- **Karta Code** – umożliwia tekstową edycję pliku.
- **Karta Design** – otwiera edytor graficzny.
- **Karta Split** – jest połączeniem obu edytorów.



Rysunek 6.27 Zakładki umożliwiające zmianę widoku układu



Rysunek 6.28 Graficzny edytor układu.



Rysunek 6.29 Tekstowy edytor układu.

W tej części podręcznika będziemy tworzyć układ za pomocą edytora tekstowego. Edytor graficzny jest domyślny, a co za tym idzie każda nowo otwarta aplikacja otwiera się w tym widoku.

6.3.2 Podstawowe pojęcia

Aby rozpocząć programowanie aplikacji mobilnych musimy zapoznać się z podstawowymi pojęciami, które będą pojawiać się w tej książce.

Układ – układ to sposób rozmieszczenia elementów w aplikacji. Android Studio umożliwia utworzenie kilku rodzajów układów. Każdy z nich ma swoje specyficzne właściwości i zastosowanie. Pliki w których znajduje się kod układu znajdziecie w katalogu **res/layout** i zawsze mają rozszerzenia xml.

Aktywność – aktywność jest to działanie jakie wykonuje aplikacja. Może to być reakcja na wciśnięcie przez użytkownika przycisku czy wybranie opcji z listy rozwijanej. Aktywności znajdziemy w katalogu Java. Są to pliki z rozszerzeniem ***.java** i są zaprogramowane właśnie w tym języku.

Widok – (inaczej widżet bądź komponent) to element umieszczony w układzie. Widokiem jest pole tekstowe: **TextView**, a także przycisk **Button**. Powyższe

pojęcia będą pojawiać się w tym opracowaniu wielokrotnie. Dlatego też ich definicje pojawiają się na samym początku przygody z Androidem.

6.3.3 Układ liniowy

Kod tworzonej aplikacji znajdziesz w folderze **Aplikacja2.** Jako pierwszy z układów omówimy układ liniowy. Jego charakterystyczną cechą jest to, że każdy z elementów (komponentów) aplikacji, który umieszczamy w projekcie jest układany jeden obok drugiego w takiej kolejności w jakiej jest umieszczony w pliku układu.

Zaczniemy od utworzenia „**ramy**" naszego układu czyli zdefiniowania układu liniowego.

W pliku **activity_main.xml** umieszczamy kod z poniższego Listingu.

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

</LinearLayout>
```

Listing 6.1 Układ liniowy

Najważniejsze elementy:

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

Listing 6.2 Dodanie aplikacji przestrzeni nazw

Za pomocą tych linijek dodajemy do aplikacji tzw. **Przestrzeń nazw**, która umożliwia nam na korzystanie z języka **xml**.

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

Listing 6.3 Ustawienie szerokości i wysokości widoku

Ustawiamy szerokość i wysokość układu.

```
android:orientation="vertical"
```

Listing 6.4 Układ liniowy

Ustawiamy orientacje elementów w układzie.

Ten atrybut może przyjmować dwie opcje:

- **Verical** – element są umieszczone jeden pod drugim;
- **Horizontal** – element są ułożone jeden obok drugiego;

```
tools:context=".MainActivity">
```

Listing 6.5 Odniesienie do pliku aktywności

Do pliku układu podłączamy plik aktywności. (domyślnie powinien być to plik **.MainActivity.java**). Wszystkie z powyższych atrybutów (oprócz android:orientation) znajdują się w każdej aplikacji bez względu od użytego układu.

Atrybuty **android:layout_width**, oraz **android:layout_height** mogą przyjmować różne wartości. Możemy skorzystać z wartości:

- **match_parent** – która dopasuje wielkość układu do elementu nadrzędnego tzw. rodzica. W przypadku układu jest to okno aplikacji.
- **wrap_content** – który dopasuje wielkość elementu do swojej zawartości np. napisu.

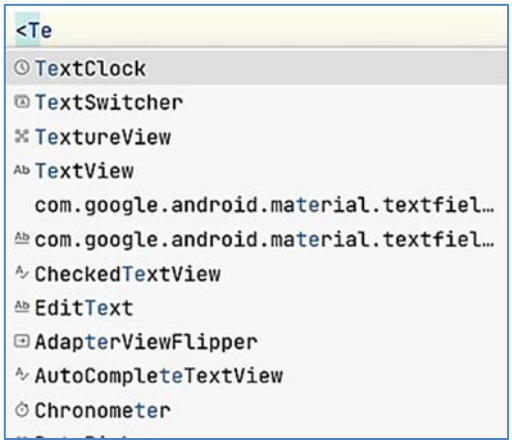
Atrybutom tym można również nadać wartość jednostki liczbowej np. **25dp**. Aplikacja nie zawiera jeszcze żadnych widocznych elementów. Utworzyliśmy na razie samą ramkę. Dodajmy do niej teraz prosty tekst.

Do wyświetlania tekstu aplikacji służy widget: **TextView**. Nasz napis ma znajdować się wewnątrz widoku **LinearLayout**. Dlatego w kodzie musimy umieścić go pomiędzy znacznikami:

```
<LinearLayout

</LinearLayout>
```

Listing 6.6 Układ liniowy

Najlepiej ustawmy się kursorem **poniżej linijki** z wpisem **tools:context=".MainActivity">.** Klikamy **kursor** i wpisujemy znaki **<T -** program sam podpowie nam co powinniśmy wpisać. Z rozwijanej listy wybieramy **TextView** i klikamy. Program sam utworzy widżet z obowiązkowymi elementami.



Rysunek 6.30 Okno Autouzupełnienie

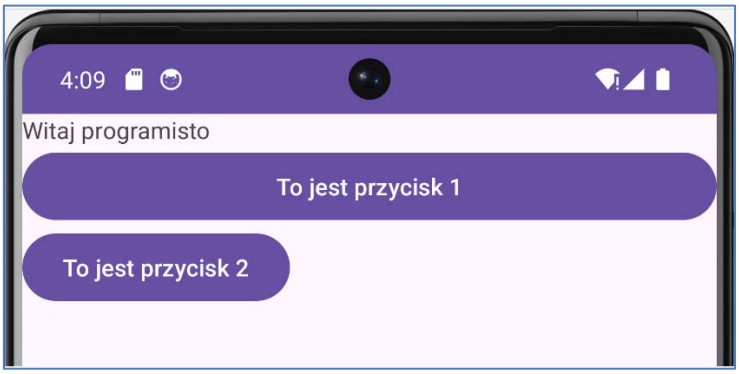
Uzupełniamy wartości atrybutów **android:layout_width** i **android:layout_height** ustawiając wartości **match_parrent**. Dodajemy również atrybut android:text w którym umieścimy napis. Zostanie on wypisany na ekranie aplikacji.

Do aplikacji dodamy teraz dwa przyciski. Przycisk wstawiamy za pomocą widżetu **Button**. Pod elementem **TextView** umieszczamy w pliku układu poniższy kod. Będzie on teraz wyglądał następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    tools:context=".MainActivity">
  <TextView
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:text="Witaj programisto"
     />
  <Button
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:text="To jest przycisk 1" />
  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="To jest przycisk 2"
      />
</LinearLayout>
```

Listing 6.7 Układ liniowy

Aplikacja po uruchomieniu będzie wyglądać tak jak na poniższym rysunku.



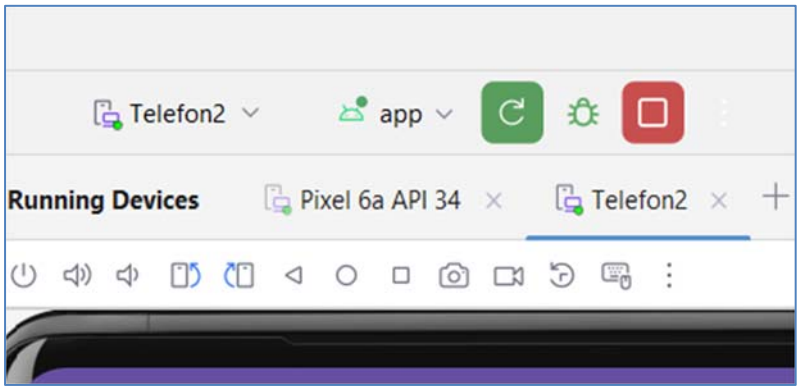
Rysunek 6.31 Aplikacja prezentująca działanie właściwości match_parent i wrap_content.

W aplikacji zauważyć można różny sposób wyświetlania przycisków. **Przycisk 1** jest rozciągnięty na szerokość układu, gdyż zastosowano wartość **match_parent** dla atrybutu **android:layout_width**. Szerokość przycisku została dostosowana do szerokości układu.

Przycisk 2 jest znacznie węższy, gdyż jego szerokość ustawiona jest na **wrap_content**. Szerokość przycisku została dostosowana do długości napisu. Gdy go zastąpimy tekst „**To jest przycisk 2**" napisem „**Przycisk 2**" jego szerokość dostosuje się do długości zawartości.

6.3.4 Ponowne uruchomienie aplikacji po wprowadzeniu zmian w kodzie.

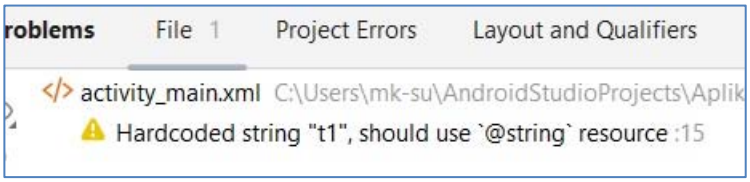
Jeżeli dokonujemy zmian w aplikacji i chcemy zobaczyć podgląd po edycji musimy odświeżyć aplikację w emulatorze. W tym celu trzeba kliknąć w ikonę **zielonej zaokrąglonej strzałki** a górze edytora.



Rysunek 6.32 Przycisk ponownego uruchomienia aplikacji

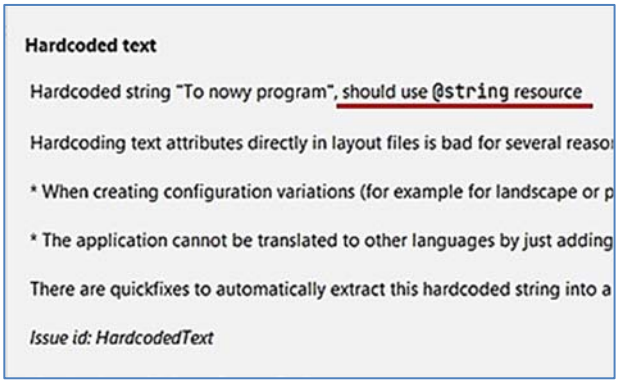
6.3.5 Plik przechowujący łańcuchy znaków - strings.xml

Zmodyfikowany kod aplikacji znajdziesz w folderze **Aplikacja2a.** W poprzednim ćwiczeniu umieściliśmy w układzie kilka elementów. Każdy z nich zawierał atrybut **android:text**. Wyświetlany w aplikacji napis wpisywaliśmy bezpośrednio jako wartość atrybutu. Zajrzyjmy do edytora i sprawdźmy **komunikaty warning**, które pojawiały się w edytorze.



Rysunek 6.33 Treść zwracanego ostrzeżenia: Hardcoded text

Jak widać na rysunku. Wstawiony przez nas widok **TextView** zgłasza zastrzeżenie. Brzmi on: **Hardcoded text**. **Hardcode** to rodzaj praktyki programistycznej. Polega on na tym, że dane w programie umieszczamy bezpośrednio w docelowym elemencie. Jak sami widzimy Android Studio zgłasza nam kod jako niekoniecznie pewnego rodzaju błąd, a raczej ostrzeżenie. Gdy przyjrzymy się bliżej, Android podpowie nam co powinniśmy zrobić, żeby ten błąd poprawić.

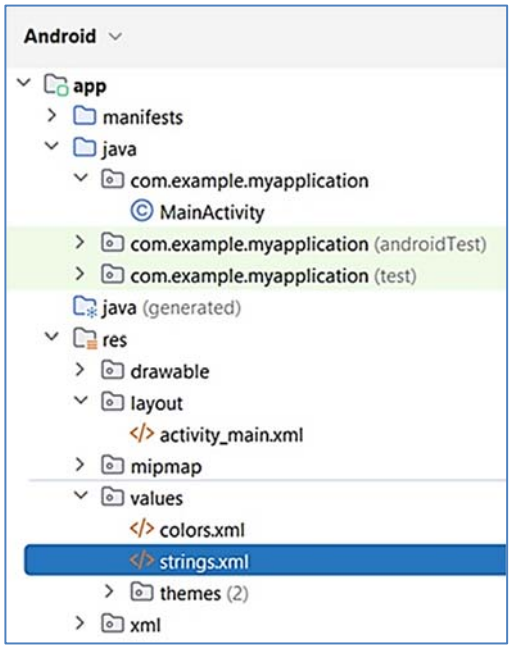


Rysunek 6.34 Informacja systemowa o możliwym rozwiązaniu

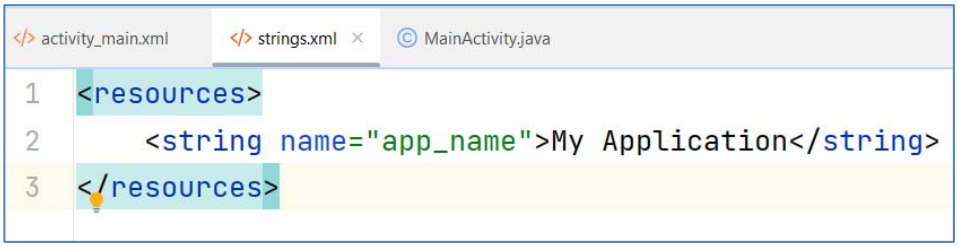
Android podpowiada, że powinniśmy użyć tekstu pochodzącego z pliku **@string**.

Plik **strings.xml** znajdziemy w drzewie aplikacji czyli liście plików i katalogów które budują nasz projekt. Jeżeli klikniemy po **prawej stronie edytora** zakładkę **Project** otworzy nam się lista plików.

Trzy najważniejsze z nich to: **manifest**, **java** i **res**. Plik zasobów **strings.xml** znajdziemy w katalogu **res**, a właściwie w jego podkatalogu o nazwie **values**. Kliknięcie na plik doda nam zakładkę do edytora i pozwoli nam na jego edycję.



Rysunek 6.35 Drzewo aplikacji z plikiem strings.xml



Rysunek 6.36 Zawartość początkowa pliku strings.xml

Teraz będziemy edytować plik **strings.xml** dodając do niego nasze napisy. Wszystkie wpisy dotyczące napisów będziemy umieszczać pomiędzy znacznikami **<resources>** i **</resources>.** Pusty plik **strings.xml** zawiera następujący kod.

```
<resources>
   <string name="app_name">Aplikacja2a</string>
</resources>
```

Listing 6.8 Plik strings.xml

Pojawiają się tutaj wcześniej wspomniane znaczniki **resource**, oraz łańcuch **app_name**, któremu przypisana jest nazwa naszej aplikacji. Uzupełnimy kod pliku o napisy, które noszą nasze widoki.

```
<resources>
<string name="app_name">Aplikacja2</string>
<string name="tekst1">
Witaj programisto</string>
<string name="przycisk1">To jest przycisk 1</string>
<string name="przycisk2">To jest przycisk 2</string>
</resources>
```

Listing 6.9 Plik strings.xml z uzupełnioną wartością.

Teraz odnośniki do napisów musimy umieścić w pliku układu. Przechodzimy do pliku **activity_main.xml** i uzupełniamy atrybuty **android:text** o następujące wpisy.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
 xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    tools:context=".MainActivity">
  <TextView
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:text="@string/tekst1" />
  <Button
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      android:text="@string/przycisk1" />
  <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/przycisk2"
      />
</LinearLayout>
```

Listing 6.10 Plik układu z ustawionymi widokami: TextView i Button.

Zastępujemy tekst odnośnikiem do pliku string. Znak @ informuje, że będziemy sięgać do innego pliku. Nazwa po znaku @ oznacza plik do którego odnosi się zasób. W przypadku tego ćwiczenia jest to plik zasobów tekstowych string. Po znaku „/” wpisujemy nazwę łańcucha definiującego napis.

Wygląd aplikacji w tym przypadku nie zmieni się. Z edytora znikną tylko ostrzeżenia.

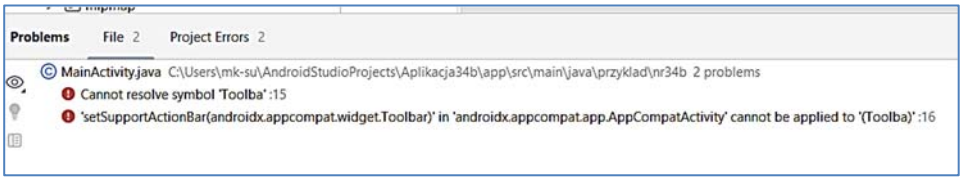
6.3.6 Warning a Error

W poprzednim ćwiczeniu mieliśmy do czynienia z tzw. **warningiem**. Jest to ostrzeżenie o pewnych nieprawidłowościach, które występują w kodzie i powinniśmy je poprawić.

Jednak każdy nawet bardzo zaawansowany programista popełnia bardziej poważne błędy. Android Studio wyłapie wszystkie nieprawidłowości w kodzie i zgłosi je jako błąd czyli Error. Czym zatem różni się **Warning** od **Error**?

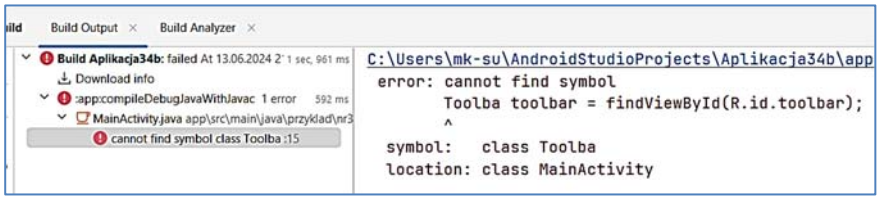
Jak już wielokrotnie wspomniano Warning to tylko ostrzeżenie – taka żółta kartka dla programisty. Zresztą informacja o tego typu błędach jest wyświetlana na żółto. Warningi powinniśmy poprawiać jednak nie mają one istotnego wpływu na działanie aplikacji. Emulator bez problemu uruchomił pierwszą aplikację pomimo ostrzeżeń.

Z **Errorem** jest zupełnie inaczej. Po pierwsze **Error** oznaczony jest kolorem **czerwonym**. Jego wystąpienie ma istotny wpływ na działanie aplikacji i **Emulator** niestety nie uruchomi aplikacji w której znajdzie **Error**.



Rysunek 6.37 Błędy zgłaszane przez Android Studio

Rysunek przedstawia fragment kodu po usunięciu z niego frazy **LinearLayout**. W lewym górnym rogu widzimy czerwone ostrzeżenie. Większość kodu również się nieco „zaczerwieniła". Gdy spróbujemy uruchomić aplikację z powyższym błędem **Emulator** zbuntuje się. Aplikacja się nie uruchomi, a na ekranie pojawią się następujące komunikaty.



Rysunek 6.38 Program po uruchomieniu aplikacji z błędami.

6.4 Aktywność

Kod aplikacji znajdziesz w folderze **Aplikacja3.**

Wszystkie aplikacje w systemie Android składają się z co najmniej dwóch elementów. Pierwszym z nich jest **układ**. Drugim elementem są **aktywności**. **Aktywność** odpowiada za działanie aplikacji – reakcje na przyciśnięcie przycisku, wyświetlenie komunikatu.

Układy definiowane są w plikach z rozszerzeniem ***.xml** – głównym z nich jest zazwyczaj plik **activity_main.xml**. Aktywność zaś definiowana jest w pliku z rozszerzeniem **java** bądź **kt**. W zależności czy jest pisana w **Javie** czy w języku **Kotlin**. Główny plik aktywności nosi zazwyczaj nazwę **MainActivity.java** (w przypadku języka Kotlin **MainActivity.kt**). Aktywność i układy tworzą ze sobą pary.

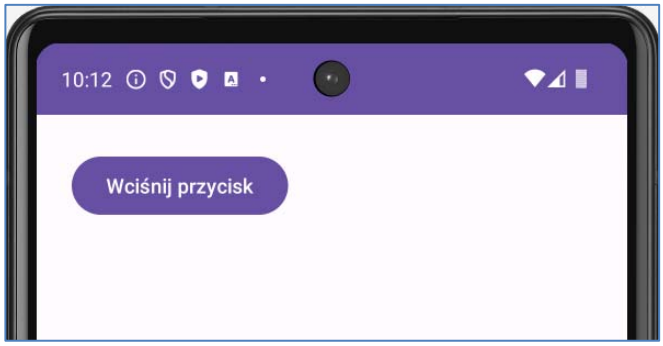
Stwórzmy nową aplikację. Jej gotowy kod znajdziesz w folderze o nazwie **Aplikacja3.** Zacznijmy od wstawienia przycisku który zareaguje na nasze działanie. Wstawiamy go w środku układu liniowego – **LinearLayout**. Kod układu **activity_main.xml** będzie wyglądał następująco:

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

Listing 6.11 Kod układu aplikacji Aplikacja3

Nowym atrybutem jest tutaj **android:layout_margin** i przyjmuje on wartość **25dp**. Jednostka **dp** jest charakterystyczna dla systemu Android. **1dp** oznacza odległość bądź długość **1 piksela** bez względu na gęstość ekranu. Jest więc niezależna od rozdzielczości urządzenia na którym pracuje aplikacja.

Sam atrybut zaś będzie tworzył margines pomiędzy widokiem a krawędzią ekranu, bądź innym elementem. Zanim uruchomisz aplikację zadbaj jeszcze o uzupełnienie pliku **strings.xml**. Po uruchomieniu aplikacja wygląda jak na obrazie poniżej.



Rysunek 6.39 Okno aplikacji po uzupełnieniu kodu układu

Zanim zaprogramujemy aktywność przyjrzyjmy się elementom pliku aktywności. Plik aktywności powinien otworzyć się jako zakładka edytora. Jeżeli się nie wyświetla trzeba go samodzielnie otworzyć. Znajduje się on w katalogu **Java** i nosi nazwę **MainActivity**.

```
package przyklad.nr3;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;

public class MainActivity extends AppCompatActivity {

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

Listing 6.12 Kod aplikacji Aplikacja3

Poszczególne linijki oznaczają:

- **Package** – jest to tzw. paczka. Nazwa która się wyświetla to nazwa projektu. Jeżeli nie pamiętasz nazwy paczki zawsze możesz ją sprawdzić w drzewie projektu po lewej stronie edytora.
- **Import** – pozwala na umieszczenie w aplikacji klas, które są dołączone do projektu. Im bardziej rozbudowana jest aplikacja (a konkretnie aktywność) tym więcej klas jest zaimportowanych. Edytor zaznacza na szaro te klasy, które są dołączone, a które nie są w danym projekcie potrzebne.

W tej aplikacji mamy na początku podłączone dwie klasy.

- **androidx.appcompat.app.AppCompatActivity** – importuje klasę bazową. Jest to podklasa klasy Activity, która ma za zadanie zapewnić możliwość korzystania z funkcji Android dostępnych w starszych wersjach API.
- **android.os.Bundle** – jest ona odpowiedzialna za komunikację komponentów w aplikacji, oraz przechowywanie danych podczas działania aplikacji.

Kolejna linijka kodu oznacza klasę główną aplikacji w której będziemy zapisywać wszystko, co w aktywności się dzieje. Klasa **MainActivity** dziedziczy po klasie **AppCompatActivity**.

- **@Override** - jest to informacja dla kompilatora, że klasa będzie przysłaniać inne (bardziej szczegółowe informacje o przesłanianiu klas znajdziesz w części dotyczącej języka Java).
- **protected void onCreate(Bundle savedInstanceState)** – Metoda onCreate jest wywoływana w chwili tworzenia aktywności po raz pierwszy
- **setContentView(R.layout.activity_main)** – w tej linijce najprościej mówiąc informujemy aktywność z jakiego pliku układu ma korzystać (kolokwialnie mówiąc jest to podłączenie aktywności pod konkretny układ).

W ostatniej linijce należy zwrócić również uwagę na **literkę R**, gdyż ma ona bardzo duże znaczenie w tworzeniu aktywności.

Litera R jest w pewnym sensie łączem z plikiem **R.java**. Znajduje się on w plikach aplikacji w katalogu o ścieżce **app/build/generated/source/r/debug**. Jeżeli zachodzi taka potrzeba plik ten można otworzyć i sprawdzić jego zawartość. Plik **R.java** aktualizuje się sam w trakcie tworzenia aplikacji. Zawiera definicje wszystkich elementów jakie zawiera nasza aplikacja.

Gdy już zapoznaliśmy się z podstawowym wyglądem pliku aktywności możemy przejść do edycji aplikacji. Na początku będziemy edytować kod układu dopisując funkcje dzięki której przycisk **Button** będzie reagował na wciśnięcie.

```
android:onClick="napiszCos"
```

Listing 6.13 Przypisanie metody do przycisku

Dodanie powyższej linijki w pliku **activity_main.xml** w części odpowiadającej za przycisk spowoduje, że po kliknięciu przycisku uruchomi się metoda o nazwie **napiszCos**, której definicja znajdzie się w pliku **ActivityMain.java**.

Chcemy, aby aplikacja wyświetlała tekst po wciśnięciu przycisku. Żeby tekst mógł się znaleźć na ekranie musimy „przygotować" mu miejsce.

W pliku układu musimy dodać element **TextView**. Uzupełniony kod będzie wyglądał w następujący sposób:

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

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="25dp"
        android:text="@string/Przycisk"
        android:onClick="napiszCos"/>
    <TextView
        android:id="@+id/napis1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:textSize="30sp"
        android:text="" />

</LinearLayout>
```

Listing 6.14 Kod układu z elementem TextView

Element TextView na początku jest pusty. Jeżeli chcemy, żeby przed przyciśnięciem przycisku znajdował się tam tekst, musimy przygotować dla niego łańcuch w pliku strings.xml. W widoku **TextView** pojawiły się też nowe atrybuty:

```
android:id="@+id/textView1"
```

Listing 6.15 Nadanie id.

Atrybut ten nadaje widokowi id czyli unikalną nazwę. Poprzez tą nazwę będzie można odwołać się do tego widoku w pliku aktywności.

```
android:textSize="30sp"
```

Listing 6.16 Ustawienie rozmiaru czcionki

Ten atrybut nadaje rozmiar czcionki. Tutaj ustawiony jest na wartość **30sp**. **Sp** jest jednostką występującą w Androidzie. Jest ona bardzo podobna do jednostki **dp**, z tą różnicą, że uwzględnia ona ustawienia jakich w systemie dokonał użytkownik telefonu.

Gdyż już dokonaliśmy zmian w pliku układu przechodzimy do edycji Aktywności. W tym celu przechodzimy do pliku **ActivityMain.java**.

Na początek wpiszemy metodę, która ma zareagować na wciśnięcie przycisku. Zgodnie z tym co zapisane jest w pliku układu w miejscu dotyczącym przycisku Button metoda będzie nosiła nazwę **napiszCos**. Umieścimy ją wewnątrz klasy MainActivity pod klasą **onCreate**.

Metoda będzie typu **View**. Klasa View w Androidzie reprezentuje tzw. widoki (lub widżety) czyli elementy które można wstawić do aplikacji. Jednym z widżetów jest przycisk Button. Inne widżety poznamy w dalszej części książki. Plik MainActivity.java po dodaniu pustej metody napiszCos będzie wyglądał następująco.

```
package przyklad.nr3;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void napiszCos (View view) {

    }
}
```

Listing 6.17 Kod aktywności głównej

Uzupełniamy kod tak, aby po wciśnięciu przycisku pojawił się napis.

```
TextView tekst = findViewById(R.id.napis1);
```

Listing 6.18 Utworzenie referencji do elementu z pliku układu

W klasie **napiszCos** deklarujemy obiekt typu **TextView** i nadajemy mu nazwę **tekst**. Utworzonemu obiektowi przypisujemy referencję do elementu **TextView**. Poprzez funkcję **findViewById** odnajdujemy w pliku układu odpowiedni element. Zwróć uwagę, że wyszukujemy element o odpowiednim id. W przypadku tego fragmentu kodu jest to id o nazwie **tekst1**, a właśnie taką nazwę nadaliśmy elementowi **TextView** w pliku układu.

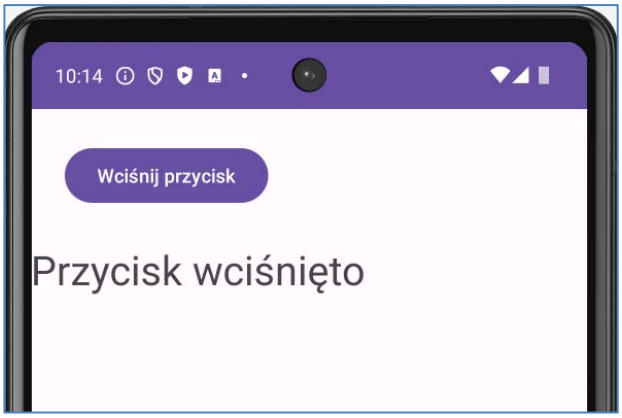
```
tekst.setText("Przycisk wciśnięto");
```

Listing 6.19 Ustawienie łańcucha znaków

Zadaniem tej metody będzie przypisanie zmiennej tekst (za pomocą **setText**) napisu, który zostanie wyświetlony po przyciśnięciu przycisku. Poniżej znajduje się kompletny kod obu pików. Sprawdź czy kod zgadza się z tym co udało Ci się napisać, a następnie uruchom aplikację. Kod metody **napiszCos**:

```
public void napiszCos (View view) {
TextView tekst = findViewById(R.id.napis1);
tekst.setText("Przycisk wciśnięto");
   }
```

Listing 6.20 Definicja metody napiszCos



Rysunek 6.40 Okno aplikacji po uruchomieniu aktywności

6.5 Komunikat Toast

Kod aplikacji znajdziesz w folderze **Aplikacja4**.

Komunikat **Toast** jest krótką informacją pojawiającą się na dole aplikacji. Wykorzystujemy go do szybkiej komunikacji z użytkownikiem. Tworzymy aplikację zawierającą jeden przycisk **Button.** Kod układu będzie wyglądał tak:

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

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_margin="25dp"
        android:text="@string/Przycisk1"
        android:onClick="toast"/>

</LinearLayout>
```

Listing 6.21 Plik układu aplikacji Aplikacja4.

W pliku aktywności definiujemy metodę o nazwie toast, która będzie reagowała na wciśnięcie przycisku. Definicja metody będzie miała następujący kod:

```
String tekst = "Brawo uruchomiłeś Toast";
```

Listing 6.22 Zdefiniowanie łańcucha znaków.

Zmienna typu **String** będzie przechowywała treść komunikatu.

```
int duration = Toast.LENGTH_SHORT;
```

Listing 6.23 Zmienna duration z przypisaną wartością. Długością trwania komunikatu Toast.

Zmienna typu int ma przypisaną długość komunikatu, która jest wyznaczona za pomocą wartości **LENGTH.SHORT**.

```
Toast toast = Toast.makeText(this, tekst, duration);
```

Listing 6.24 Utworzenie obiektu klasy Toast.

Zmiennej toast, która jest obiektem typu **Toast** przypisujemy wynik działania metody **makeText**. Metoda ta, ma trzy parametry.

Pierwszy z nich to tzw. **Context** czyli miejsce w którym będzie uruchomiona metoda. Biorąc pod uwagę, że metodę tę wykonujemy w środku klasy w której się znajduje **Context** ustawiamy na wartość **this**.

Drugi parametr to nazwa zmiennej przechowującej komunikat, który ma być wyświetlony w okienku **Toast**. Ostatni parametr to zmienna **duration**, czyli ta która będzie przechowywać czas trwania komunikatu.

Parametr **duration** może przyjmować dwie wartości:

- **LENGTH.SHORT** – krótki czas trwania
- **LENGTH.LONG** – dłuższy czas trwania

Ostatnim zadaniem klasy jest wyświetlenie komunikatu toast i tutaj pomoże nam metoda show.

```
toast.show();
```

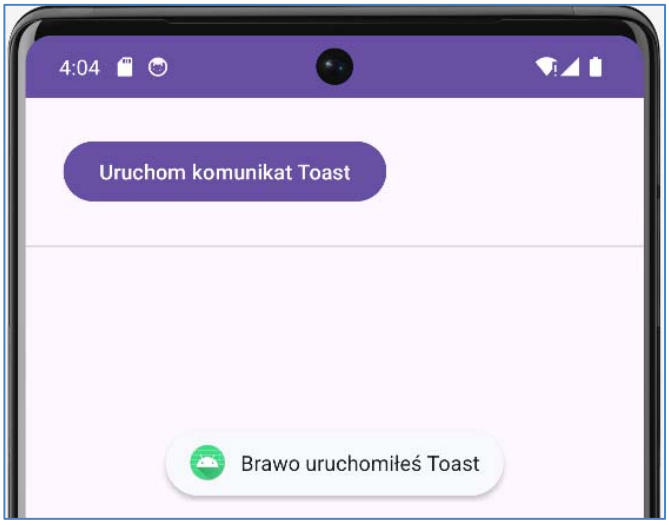
Listing 6.25 Uruchomienie komunikatu Toast.

Cały kod aktywności będzie miał następującą postać:

```
package przyklad.nr4;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void toast(View view) {
 String tekst = "Brawo uruchomiłeś Toast";
 int duration = Toast.LENGTH_LONG;
 Toast toast = Toast.makeText(this, tekst,
duration);
  toast.show();
    }
}
```

Listing 6.26 Kod aktywności aplikacji Aplikacja4.

Po uruchomieniu aplikacji i wciśnięciu przycisku na dole telefonu zobaczymy komunikat **Toast**.



Rysunek 6.41 Komunikat Toast

6.6 Widoki

O widokach wspominaliśmy w poprzednim rozdziale. Były to m.in.: przyciski: **Button**, oraz pole tekstowe: **TextView.** Teraz poznamy inne rodzaje widoków, które przydadzą się nam przy pracy nad aplikacjami.

Ważną informacją jest fakt, że wszystkie widgety są elementami GUI czyli Graficznego Interfejsu Użytkownika (ang, graphical user interface). Są to elementy, które są pochodnymi poznanej już klasy **View**.

6.6.1 EditText – pole tekstowe

Kod aplikacji znajdziesz w folderze **Aplikacja5.**

Pierwszym elementem GUI będzie pole tekstowe. Jest to okienko w którym możemy wpisywać tekst. W Android Studio pole tekstowe reprezentowane jest przez widget – **EditText**.

Kod umieszczający w układzie element **EditText** będzie wyglądał następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <EditText
        android:id="@+id/pole1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/pole1"
        android:inputType="textCapSentences"
        android:textColorHint="#8D6E63"
        />
</LinearLayout>
```

Listing 6.27 Widok EditText w zdefiniowany w pliku układu

Nowe atrybuty:

```
android:hint="@string/pole1"
```

Listing 6.28 Atrybut ustawiający podpowiedź.

to tzw. podpowiedź. Tekst w cudzysłowie będzie wyświetlał się po uruchomieniu aplikacji, zanim użytkownik coś wpisze. Jego treść zdefiniowana jest w pliku **strings.xml.**

```
android:inputType="textCapSentences"
```

Listing 6.29 Deklaracja atrybutu inputType

Atrybut określa rodzaj wprowadzanych w polu danych. Możliwe opcje tego atrybutu znajdziesz w poniższej tabeli.

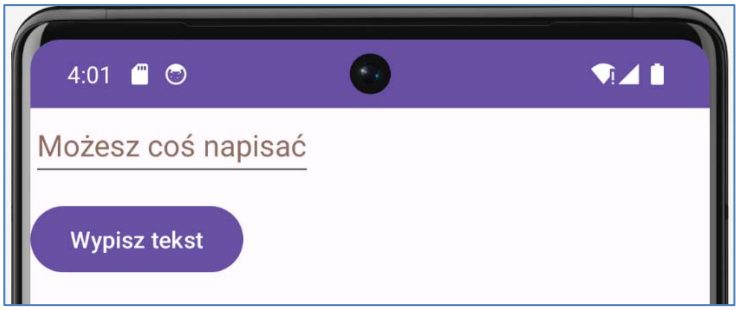
Wartość	Rodzaj wprowadzonych danych
Numer	Umożliwia wprowadzanie liczb
Phone	Umożliwia wprowadzanie numeru telefonu
textPassword	Umożliwia wpisanie hasła; każdy z wprowadzonych
textCapSentences	Umożliwia wpisanie tekstu; Każde nowe zdanie
textAutoCorrect	Umożliwia wprowadzenie tekstu; Automatycznie

Tabela 4.1: Możliwe wartości atrybutu android:inputType

```
android:textColorHint="#8D6E63"
```

Listing 6.30 Atrybut ustalający kolor podpowiedzi.

Określa kolor tekstu wyświetlanego jako podpowiedź. Kolor ustawiony jest w notacji szesnastkowej. Uruchom teraz aplikacje i sprawdź jak wygląda.



Rysunek 6.42 Pole edycyjne EditText

Wykorzystamy teraz widok **EditText** w pliku aktywności.

Aplikacja będzie pobierać tekst z pola tekstowego i wyświetli go na ekranie. Do utworzenia aplikacji będziemy potrzebować komponenty: **EditText, TextView**, oraz przycisk **Button**. Plik układu aplikacji znajdziesz poniżej:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <EditText
        android:id="@+id/pole1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="@string/pole1"
        android:inputType="textCapSentences"
        android:textColorHint="#8D6E63"
        />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/przycisk1"
        android:text="@string/przycisk1"
        android:layout_marginTop="10dp"
        android:onClick="wypiszTekst"
        />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/tekst1"
        android:text=""
        android:layout_marginTop="10dp"/>
</LinearLayout>
```

Listing 6.31 Pełny kod układu.

Teraz zajmiemy się plikiem aktywności **MainActivity.java**.

Do pobrania tekstu z pola tekstowego będziemy potrzebować obiektów typu **EditText, TextView** z pliku układu. Przyda nam się również zmienna pobrany. Będzie ona typu znakowego String. Zmienna pobierze i przechowa wpisany przez użytkownika tekst.

```
EditText tekst = findViewById(R.id.pole1);
String pobrany = tekst.getText().toString();
TextView napis = findViewById(R.id.tekst1);
napis.setText(pobrany);
```

Listing 6.32 Utworzenie referencji do elementów z pliku układu: EditText i TextView.

Pierwsza linijka tworzy obiekt klasy **EditText**, o nazwie tekst i tworzy referencję do elementu w pliku układu o nazwie **pole1**.

W drugiej linijce zmiennej o nazwie pobrany przypisujemy tekst, który został pobrany z pola tekstowego za pomocą metody **getText**. Funkcja **toString** ma za zadanie przekształcić tekst (wpisany przez użytkownika) na łańcuch znaków.

Pobrany napis musimy teraz wstawić w miejsce napisu w polu **TextView**. Potrzebna będzie nam referencja do odpowiedniego widoku w pliku układu i tu z pomocą przychodzi nam metoda **findViewById**. W ostatnim kroku ustawiamy tekst metodą **set.text**.

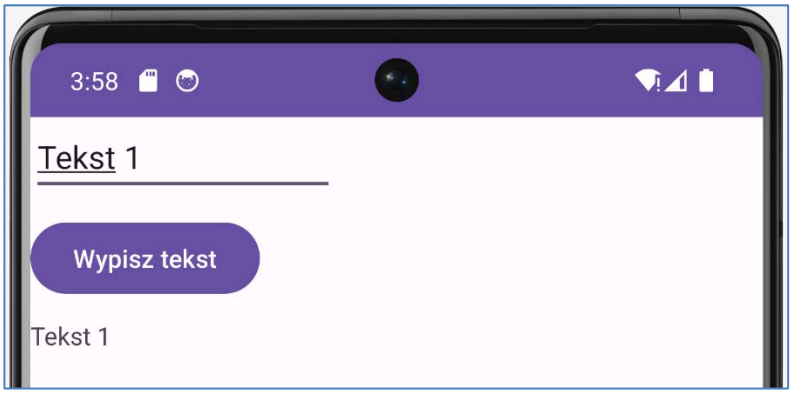
W związku z tym, że aplikacja ma zadziałać po przyciśnięciu przycisku kod pobrania i wypisania tekstu trzeba umieścić w metodzie przypisanej przyciskowi **Button**.

Pełny kod aktywności znajduje się poniżej.

```
package przyklad.nr5;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.EditText;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    @Override
protected void onCreate(Bundle savedInstanceState) {

        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void wypiszTekst (View view) {
 EditText tekst = findViewById(R.id.pole1);
 String pobrany = tekst.getText().toString();
 TextView napis = findViewById(R.id.tekst1);
        napis.setText(pobrany);

    }
}
```

Listing 6.33 Kod aktywności programu Aktywność5.

Po uruchomieniu i wpisaniu tekstu w okienku **EditText** i przyciśnięciu przycisku okno aplikacji będzie wyglądać jak na obrazku poniżej.



Rysunek 6.43 Aplikacja5 po uruchomieniu

6.6.2 Wstawianie grafiki - ImageView

Kod aplikacji znajdziesz w folderze **Aplikacja6**.

Chcąc aby aplikacja wyglądała bardziej atrakcyjnie możemy do niej wstawić obrazek. Do napisania następnej aplikacji przygotuj obrazek o niedużych rozmiarach. Jeżeli nie dysponujesz np. zdjęciem możesz pobrać grafikę z Internetu. Pamiętaj tylko o zachowaniu praw autorskich.

Na początek tworzymy układ liniowy w którym umieścimy widok **ImageView**.

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

  <ImageView
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:src="@drawable/fot1"
    android:contentDescription="@string/fot1"/>

</LinearLayout>
```

Listing 6.34 Widok ImageView w pliku układu.

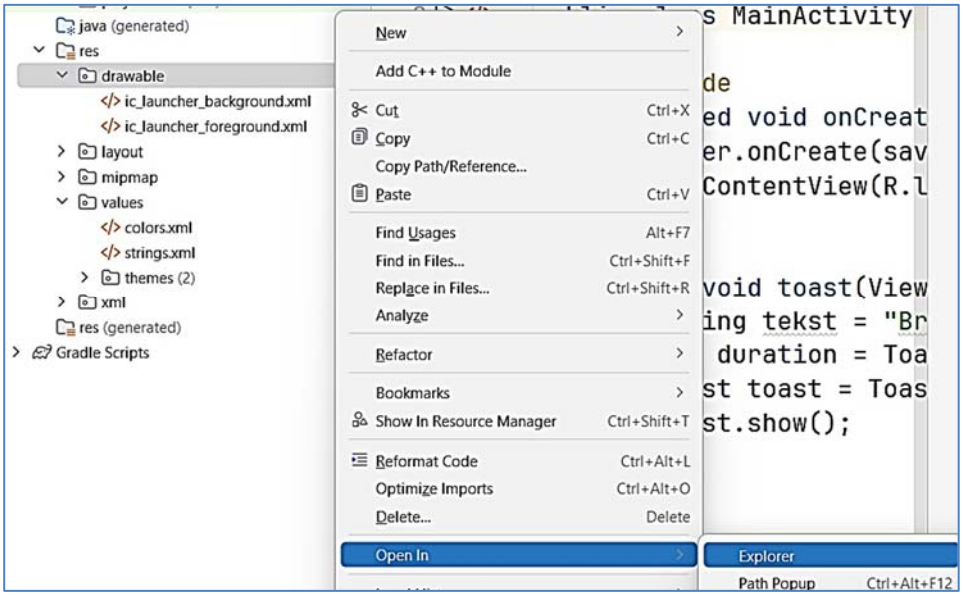
Poniższy kod:

```
android:src="@drawable/fot1"
```

Listing 6.35 Odnośnik do pliku graficznego.

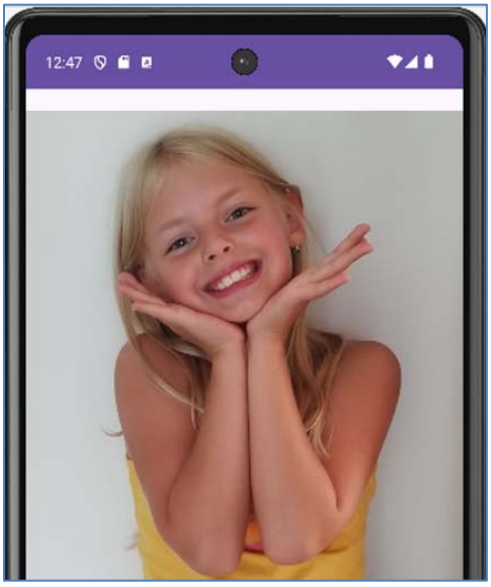
mówi nam o tym, że będziemy wstawiać obrazek o nazwie **fot1** i pobierać go z katalogu o nazwie **drawable**.

Aby obrazek był widoczny w programie musi być umieszczony w folderze **drawable**. Katalog ten jest tworzony razem z projektem. Znajduje się, więc w plikach aplikacji. Znajdziemy go wybierając odpowiednią ścieżkę dostępu lub klikając w nazwę katalogu w drzewie projektu i wybierając – „**Open in Explorer**”.



Rysunek 6.44 Otwieranie katalogu drawable z poziomu okna aplikacji

Na komputerze otworzy się nam katalog **res**. Wystarczy tylko wejść do katalogu **drawable** i tam wkleić obrazek. Ważne, żeby nazwa obrazka zgadzała się z nazwą umieszczoną w kodzie aplikacji i rozpoczynała się od litery. Akceptowane są pliki z rozszerzeniem ***.jpg** jak i ***.png**. Gdy umieścimy obrazek w środku katalogu, możemy uruchomić aplikację.



Rysunek 6.45 Aplikacja z widokiem ImageView

6.6.3 ToggleButton

Kolejnym widżetem będzie przycisk przełączania **ToggleButton**. Poniżej znajduje się kod za pomocą którego można go wstawić do aplikacji.

```
<ToggleButton
    android:id="@+id/toggle_button"
    android:layout_width="200dp"
    android:layout_height="wrap_content"
    android:textOff="@string/on"
    android:textOn="@string/off" />
```

Listing 6.36 Element ToggleButton w pliku układu

Nowe atrybuty występujące w tym kodzie to:

```
android:textOff="@string/on"
android:textOn="@string/off"
```

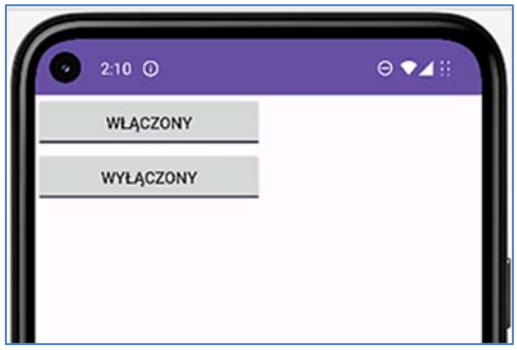
Listing 6.37 Atrybuty textOff i textOn

Za pomocą tych atrybutów określamy tekst wyświetlany na włączonym i wyłączonym przycisku. Powyższe napisy zdefiniowane są w pliku strings.xml, którego kod wygląda następująco:

```
<resources>
<string name="app_name">Aplikacja7</string>
<string name="on">Wyraź zgodę</string>
<string name="off">Wycofaj zgodę</string>
<string name="napis1">Czy wyrażasz zgodę marketingową?
</string>
</resources>
```

Listing 6.38 Plik strings.xml

Przycisk **ToggleButton** będzie wyglądał w aplikacji tak jak na obrazie poniżej. Pierwszy z nich jest w trybie **włączony**, drugi **wyłączony**.



Rysunek 6.46 Przełącznik ToggleButton

By zaprezentować działanie przełącznika **ToggleButton** przygotujemy aplikację. Kod aplikacji znajdziesz w folderze o nazwie **Aplikacja7.**

Aplikacja będzie wyświetlać informację o zgodzie użytkownika na wysyłanie reklam. W pliku układu umieszczamy dwa widoki: **ToggleButton**, oraz **TextView**, który będzie się zmieniał w zależności od tego czy użytkownik wyrazi zgodę marketingową.

Plik układu będzie wyglądał następująco:

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

    <ToggleButton
        android:id="@+id/toggle_button"
        android:layout_width="200dp"
        android:layout_height="wrap_content"
        android:textOff="@string/on"
        android:textOn="@string/off"
        android:onClick="wlaczony"
        />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/tekst1"
        android:text="@string/napis1"
        android:textSize="30sp"
        android:layout_marginTop="10dp"/>

</LinearLayout>
```

Listing 6.39 Plik układu aplikacji

ToggleButton podobnie jak zwykły przycisk **Button** również posiada atrybut **onClick** i w tej aplikacji go wykorzystamy. Przypisujemy mu wartość **wlaczony** i tak samo będzie nazywać się metoda, która uruchomi się po kliknięciu przycisku. Metoda **wlaczony** będzie miała następujący kod:

```
public void wlaczony(View view) {
TextView tekst1 = findViewById(R.id.tekst1);
boolean stan = ((ToggleButton) view).isChecked();
    if (stan) {
tekst1.setText("Wyraziłeś zgodę marketingową");
    }
   else {
tekst1.setText("Nie wyraziłeś zgody marketingowej");
    }
}
```

Listing 6.40 Definicja metody włączony

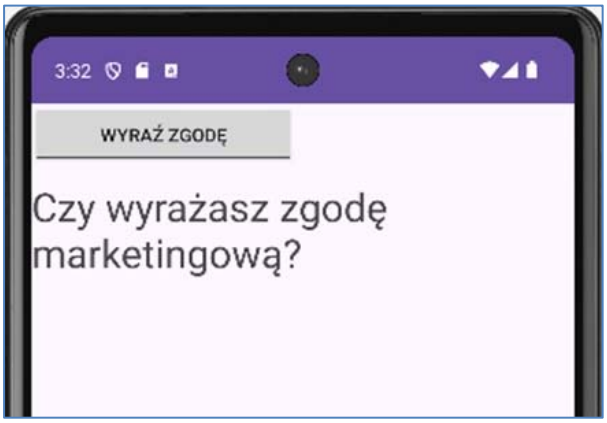
Na początek tworzymy obiekt typu **TextView**. Nadajemy mu nazwę **tekst1** i pobieramy referencję do widoku z pliku układu. Potrzebna nam będzie również zmienna typu **boolean.** Zmienna taka przyjmuje jedną z dwóch wartości **true** (prawda) bądź **false** (fałsz). Będzie ona pobierać odpowiednią wartość w zależności od tego jaki stan będzie przyjmował **ToggleButton** – **On** czy **Off**. Dla tego posługujemy się tutaj metodą **isChecked**.

Jeśli przycisk jest zaznaczony (czyli checked) zmienna przyjmie wartość **true**. W przeciwnym wypadku false. Utworzoną zmienną wykorzystamy w instrukcji warunkowej **if**. Obie opcje będą wypisywać odpowiedni tekst w zależności od tego jaką decyzję podejmie użytkownik. Kod aktywności będzie następujący:

```
package przyklad.nr7;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import android.widget.ToggleButton;

public class MainActivity extends AppCompatActivity {

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

    public void wlaczony(View view) {
TextView tekst1 = findViewById(R.id.tekst1);
boolean stan = ((ToggleButton) view).isChecked();
        if (stan) {
tekst1.setText("Wyraziłeś zgodę marketingową");
        }
        else {
tekst1.setText("Nie wyraziłeś zgody marketingowej");
        }
    }
}
```

Listing 6.41 Kod aktywności programu Aktywnosc7

Wynik działania aplikacji można zobaczyć na poniższym rysunku.



Rysunek 6.47 Aplikacja po uruchomieniu przycisku ToggleButton

6.6.4 Przełącznik Switch

Kod aplikacji znajdziesz w folderze o nazwie **Aplikacja8.**

Kolejny przycisk również pracuje w dwóch stanach. Od **ToggleButton** różni się tylko formą. Jest przesuwanym przełącznikiem. Element ten to przełącznik **Switch**. Kod za pomocą którego można go umieścić w pliku układu wygląda następująco:

```
<Switch
android:id="@+id/switch1"
android:layout_width="wrap_content"
android:layout_height="wrap_content"
/>
```

Listing 6.42 Przełącznik switch

Kod aktywności w której wykorzystamy przełącznik będzie przypominał ten z poprzedniego ćwiczenia. Kod układu będzie zawierał element **Switch**, oraz **TextView**.

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

    <Switch
        android:id="@+id/switch1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="wlaczony"
        />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/tekst1"
        android:text="@string/napis1"
        android:textSize="30sp"
        android:layout_marginTop="10dp"/>

</LinearLayout>
```

Listing 6.43 Kod układu aplikacji Aplikacja8

```
Kod aktywności MainActivity.java
package przyklad.nr8;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.view.View;
import android.widget.Switch;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void wlaczony(View view) {
  TextView tekst1 = findViewById(R.id.tekst1);
 boolean stan = ((Switch) view).isChecked();
        if (stan) {
tekst1.setText("Wyraziłeś zgodę marketingową");
        }
        else {
tekst1.setText("Nie wyraziłeś zgody marketingowej");
        }
    }
}
```

Listing 6.44 CheckBox

Kod aplikacji znajdziesz w folderze **Apliakcja9**.

Teraz przyjrzymy się polom wyboru. Pierwszy ich rodzaj to checkbox. Jest to maleńki kwadracik, który może przyjmować dwa stany. Z polem **chceckbox** spotykamy się potwierdzając znajomość Regulaminu bądź gdy wyrażamy zgodę marketingową podczas wypełniania internetowego formularza. Kod, który pozwoli nam na umieszczenie **checkbox'a** w aplikacji wygląda następująco:

```
<CheckBox
    android:id="@+id/dostawa"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/dostawa"
    android:textSize="12pt"
    android:onClick="policz"
    />
```

Listing 6.45 Widżet CheckBox w pliku układu.

Użyjemy go w aplikacji, która będzie liczyła dodatkowe koszty zamówienia. W układzie umieścimy trzy pola **checkbox: dostawa**, **płatność za pobraniem**, **ozdobne opakowanie**, oraz dwa pola **TextView**. Pełny kod układu będzie wyglądał następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout width="wrap content"
    android:layout height="wrap content"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/tekst2"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:textSize="12pt"
        android:text="@string/dodatkowe uslugi"
        />
    <CheckBox
        android:id="@+id/dostawa"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/dostawa"
        android:textSize="12pt"
        android:onClick="policz"
        />
```

```
    <CheckBox
        android:id="@+id/platnosc"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/platnosc"
        android:textSize="12pt"
        android:onClick="policz"
        />

    <CheckBox
        android:id="@+id/opakowanie"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:text="@string/opakowanie"
        android:textSize="12pt"
        android:onClick="policz"
        />

    <TextView
        android:id="@+id/tekst"
        android:layout width="wrap content"
        android:layout height="wrap content"
        android:textSize="12pt"
        android:text=""
        />
</LinearLayout>
```

Listing 6.46 Pełny kod układu

Dla pola **CheckBox** można przypisać atrybut **onClick**. W tej aplikacji będzie to metoda policz, którą zdefiniujemy w aktywności głównej. Jej kod będzie wyglądał następująco:

```
public void policz(View view) {
boolean checked = ((CheckBox) view).isChecked();
    switch((view.getId()) ) {
        case R.id.dostawa:
            if (checked)
                kwota = kwota + 10;
            else
                kwota = kwota - 10;
            break;

        case R.id.platnosc:
            if (checked)
                kwota = kwota + 5;
            else
                kwota = kwota -5;
            break;
```

```
        case R.id.opakowanie:
            if (checked)
                kwota = kwota + 20;
            else
                kwota = kwota - 20;
            break;
    }
TextView textView = findViewById(R.id.tekst);
textView.setText("Do zapłaty dodatkowo " + kwota + "
złotych");
}
```

Listing 6.47 Definicja metody policz

Zaczynamy od utworzenia zmiennej typy boolean.

```
boolean checked = ((CheckBox) view).isChecked();
```

Listing 6.48 Deklaracja zmiennej checked.

Będzie ona sprawdzać, czy pole jest zaznaczone. W dalszej części mamy instrukcje wyboru switch. Instrukcja Switch zdefiniowana w kodzie powyżej sprawdzi id opcji za pomocą metody: **(view.getId())** i zareaguje odpowiednio na zaznaczenie pola.

Wewnątrz każdego case umieszczona jest instrukcja warunkowa. Jeżeli pole będzie zaznaczone doda odpowiednią kwotę do zamówienia. Jeżeli użytkownik odznaczy opcje kwota zostanie pomniejszona.

```
TextView textView = findViewById(R.id.tekst);
textView.setText("Do zapłaty dodatkowo " + kwota + "
złotych");
```

Listing 6.49 Ustawienie referencji do elementu TextView, oraz ustawienie tekstu do wyświetlenia.

Referencja do pola **TextView** pobierze tekst. A metoda **setText** ustawi napis. W łańcuchu znajdziemy nazwę zmiennej **kwota**. Jest to wartość, która zostanie policzona po zaznaczeniu opcji. Zmienną **kwota** trzeba zadeklarować w klasie głównej.

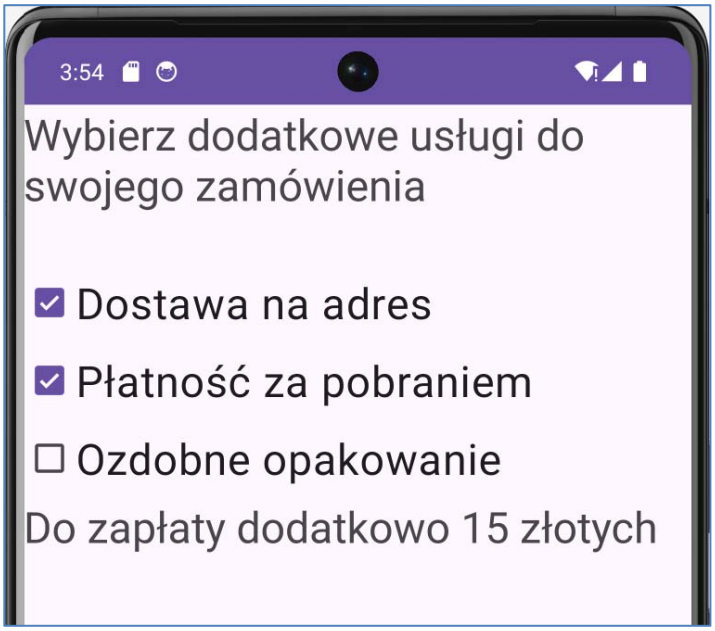
```
int kwota = 0;
```

Listing 6.50 Deklaracja zmiennej kwota

W aplikacji potrzebny nam będzie również plik strings.xml

```
<resources>
<string name="app_name">Aplikacja9</string>

<string name="dostawa">Dostawa na adres</string>

<string name="platnosc">Płatność za pobraniem</string>

<string name="opakowanie">Ozdobne opakowanie</string>

<string name="dodatkowe_uslugi">Wybierz dodatkowe usługi
do swojego zamówienia \n</string>

</resources>
```

Listing 6.51 Plik strings.xml



Rysunek 6.48 Aplikacja prezentująca działanie widoku CheckBox

6.6.5 Przyciski opcji - Radio

Przyciski opcji **Radio** podobnie jak **checkbox** pozwala na zaznaczenie wybranej opcji. Różnica pomiędzy nimi jest taka, że w grupie przycisków można zaznaczyć tylko jedną odpowiedź. W przypadku **chceckbox'a** można zaznaczyć wszystkie opcje jednocześnie.

Kod aplikacji znajdziesz w folderze **Aplikacja10**.

Żeby wstawić pole typu radio (przycisk opcji) najpierw trzeba utworzyć grupę przycisków tzw. **Radio Group.**

```
<RadioGroup android:id="@+id/radio_group"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">
</RadioGroup>
```

Listing 6.52 Widżet RadioGroup w pliku układu

Powyższy kod utworzy tylko grupę przycisków, aby pojawiły się przyciski opcji trzeba je ustawić wewnątrz **RadioGroup** pola **RadioButton**:

```
<RadioGroup android:id="@+id/radio_group"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:orientation="vertical">

    <RadioButton android:id="@+id/czerwony"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
    <RadioButton
        android:id="@+id/zielony"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
/>
    <RadioButton
        android:id="@+id/niebieski"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        />
</RadioGroup>
```

Listing 6.53 Pola RadioButton w pliku układu

Kolejna aplikacja będzie zmieniała kolor napisu w zależności od tego która opcja pola **radio** zostanie zaznaczona. Na początek przygotujemy plik układu w którym umieścimy grupę przycisków **RadioGroup**, a w niej trzy pola radio. Przydatny będzie również napis, który będzie zmieniał swój kolor w zależności od wyboru użytkownika. Plik układu będzie wyglądał tak jak poniżej.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity"
    android:orientation="vertical">
<RadioGroup
 android:id="@+id/radio_group"
 android:layout_width="match_parent"
 android:layout_height="wrap_content"
 android:orientation="vertical">

<RadioButton
 android:id="@+id/czerwony"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@string/czerw"
 android:onClick="zaznaczona" />

<RadioButton
 android:id="@+id/zielony"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@string/ziel"
 android:onClick="zaznaczona" />

<RadioButton
 android:id="@+id/niebieski"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="@string/nieb"
 android:onClick="zaznaczona" />

</RadioGroup>
<TextView
 android:id="@+id/tekst"
 android:layout_width="wrap_content"
 android:layout_height="wrap_content"
 android:text="Ten tekst zmieni kolor ;-)"
 android:textSize="25dp" />

</LinearLayout>
```

Listing 6.54 Układ liniowy wykorzystujący pola wyboru typu radio

W pliku aktywności zdefiniujemy metodę „zaznaczona”, która będzie reagowała na zaznaczenie przycisku i zmieniała kolor tekstu.

```
RadioGroup radioGroup = findViewById(R.id.radio_group);
int id = radioGroup.getCheckedRadioButtonId();
TextView tekst1 = findViewById(R.id.tekst);
```

Listing 6.55 Tworzenie referencji do widoku RadioGroup i pola TextView

Na początek tworzymy **referencję** do pliku układu dla elementów **RadioGroup** i **TextView**. Tworzymy również zmienną typu int, która pobierze **id** zaznaczonego przycisku. Wartość **id** wykorzystamy w warunku wyboru switch.

```
switch(id) {
      case R.id.czerwony:
tekst1.setTextColor(Color.parseColor("#FF0000"));
break;
```

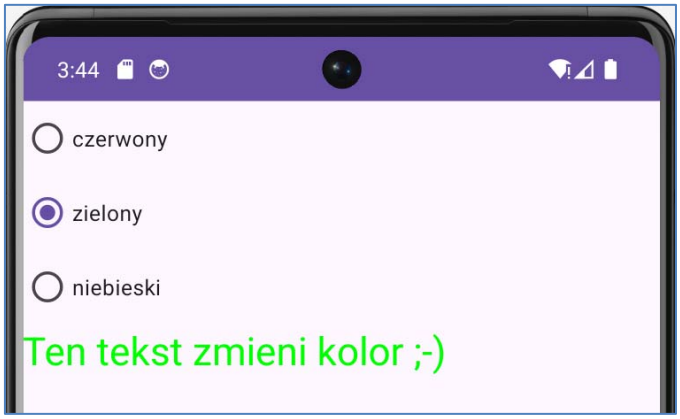
Listing 6.56 Instrukcja switch w kodzie aktywności

Wewnątrz każdej opcji **case** wykorzystujemy metodę **setTextColor**, która zamieni aktualny kolor napisu na wybrany. Metoda **parseColor** zamieni zapis szesnastkowy na wartość **int**. A oto pełny kod aktywności.

```
package przyklad.nr10;

import androidx.appcompat.app.AppCompatActivity;

import android.graphics.Color;
import android.os.Bundle;
import android.view.View;
import android.widget.RadioGroup;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void zaznaczona(View view) {

RadioGroup radioGroup = findViewById(R.id.radio_group);
int id = radioGroup.getCheckedRadioButtonId();
TextView tekst1 = findViewById(R.id.tekst);

 switch(id) {
     case R.id.czerwony:
tekst1.setTextColor(Color.parseColor("#FF0000"));
     break;
     case R.id.zielony:
tekst1.setTextColor(Color.parseColor("#00FF00"));
     break;
     case R.id.niebieski:
tekst1.setTextColor(Color.parseColor("#0000FF"));
     break;
  }
 }
}
```

Listing 6.57 Kod aktywności aplikacji Aplikacja10

Aplikacja po uruchomieniu będzie wyglądać jak na poniższym rysunku:



Rysunek 6.49 Aplikacja prezentująca działanie pola RadioButton

6.6.6 Lista rozwijana - Spinner

Android Studio umożliwia tworzenie **rozwijanej listy wyboru**. Listę możemy wstawić do aplikacji za pomocą elementu **Spinner**. Kod aplikacji znajdziesz w folderze **Aplikacja11**.

Tworzymy aplikację w której wykorzystamy **kontrolkę Spinner**. Najpierw umieszczamy komponent **Spinner** do pliku układu. Można to zrobić za pomocą poniższego kodu.

```
<Spinner
    android:id="@+id/spinner"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:entries="@array/opcjelisty"
    android:layout_marginTop="25dp"
    />
```

Listing 6.58 Widok Spinner w pliku układu

Powyższy kod utworzy komponent w układzie. Trzeba jeszcze dodać opcje do listy rozwijanej. Możemy to zrobić w pliku **strings.xml**. Za przekierowanie listy do pliku **strings.xml** odpowiada atrybut:

```
android:entries="@array/opcjelisty"
```

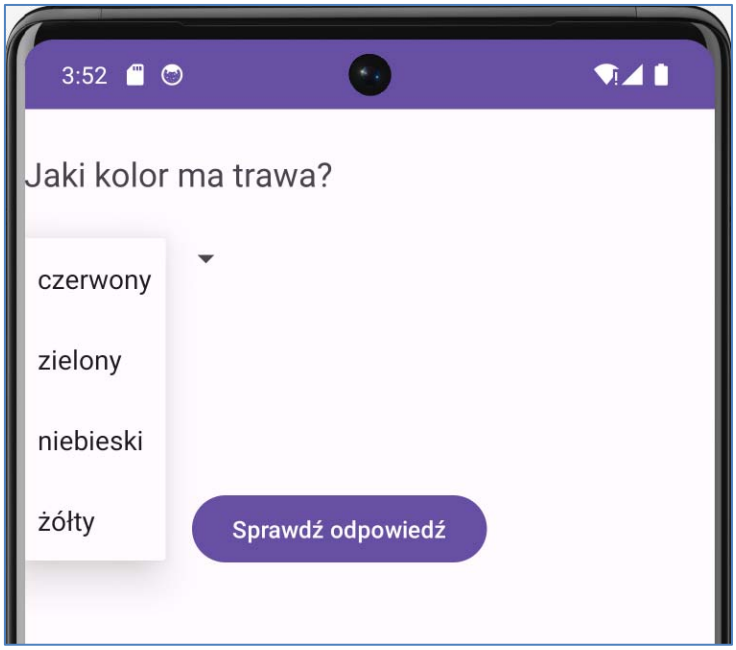
Listing 6.59 Odnośnik do elementu array

Atrybut **entries** pozwala na utworzenie opcji, które znajdą się na liście. **Array** oznacza tablicę opcji, dzięki której będziemy mogli dodać elementy listy. Kod pliku **strings.xml** będzie wyglądał następująco:

```
<resources>
<string name="app_name">widgety</string>
<string name="pytanie">Jaki kolor ma trawa?</string>
<string name="przycisk">Sprawdź odpowiedź</string>
    <string-array name="opcjelisty">
        <item>czerwony</item>
        <item>zielony</item>
        <item>niebieski</item>
        <item>żółty</item>
    </string-array>
</resources>
```

Listing 6.60 Plik strings.xml z deklaracją elementów zawartych na liście rozwijanej typu Spinner

Każdy wpis pomiędzy znacznikami **<item>** i **</item>** utworzy jedną opcję wyboru. Ważne, aby wartość **name** była taka sama jak nazwa po wpisie **@array/.** Lista rozwijana po uruchomieniu aplikacji będzie wyglądać w ten sposób:



Rysunek 6.50 Lista rozwijana Spinner, wraz z opcjami do wyboru

Wykorzystamy teraz komponent **Spinner** w pliku aktywności. Do układu dodamy **TextView**, oraz przycisk **Button**. Uzupełniony kod układu będzie wyglądał następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/tekst"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/pytanie"
        android:textSize="20sp"
        android:layout_marginTop="25dp"
        />
    <Spinner
        android:id="@+id/spinner"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:entries="@array/opcjelisty"
        android:layout_marginTop="25dp"
        />
    <Button
        android:id="@+id/przycisk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:onClick="sprawdzOdpowiedz"
        android:layout_marginTop="25dp"
        android:text="@string/przycisk" />
</LinearLayout>
```

Listing 6.61 Kod układu aplikacji Aplikacja11

W polu **TextView** pojawi się pytanie: „**Jaki kolor ma trawa?**”. Aplikacja będzie sprawdzać prawidłową odpowiedź. Trzeba zdefiniować metodę **sprawdzOdpowiedz**, która uruchomi się po wybraniu przez użytkownika odpowiedzi i wciśnięciu przycisku.

Zaczniemy od zdefiniowania dwóch komunikatów **Toast**. Pierwszy będzie się wyświetlał, gdy użytkownik wybierze prawidłową odpowiedź. Drugi uruchomi się gdy odpowiedź użytkownika będzie błędna.

```
String tekst = "Odpowiedź prawidłowa";
String tekst2 = "Odpowiedź błędna";
int duration = Toast.LENGTH_LONG;
Toast toast = Toast.makeText(this, tekst, duration);
Toast toast2 = Toast.makeText(this, tekst2, duration);
```

Listing 6.62 Definicja elementów komunikatu Toast.

Następnie zajmiemy się pobraniem wartości z komponentu **Spinner**.

```
Spinner kolor = findViewById(R.id.spinner);
String wybor = String.valueOf(kolor.getSelectedItem());
```

Listing 6.63 Referencja do elementu Spinner

Zmienna **kolor** pobierze referencję do **kontrolki Spinner** znajdującej się w pliku układu. Druga zmienna typu **String** zapamięta wybór użytkownika. Czyli ustawi swoją wartość na opcję pobraną z listy. Teraz należy wykorzystać pobraną wartość, żeby uruchomić odpowiedni komunikat **Toast**. Przyda się tutaj prosta instrukcja warunkowa.

```
if(wybor.equals("zielony"))
     toast.show();
 else
     toast2.show();
```

Listing 6.64 Instrukcja warunkowa sprawdzająca wybór użytkownika

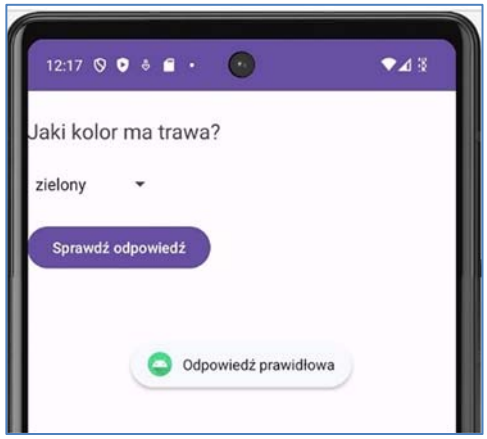
Instrukcja sprawdzi czy wybrana przez użytkownika opcja jest równa **zielony**. Jeżeli tak wyświetlony zostanie komunikat **Toast nr 1.**

W przeciwnym wypadku czyli kiedy użytkownik wybierze inną opcję zostanie wyświetlony komunikat **Toast nr 2**. Pełny kod aktywności:

```
package przyklad.nr11;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.Spinner;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
public void sprawdzOdpowiedz(View view) {
Spinner kolor = findViewById(R.id.spinner);
String wybor = String.valueOf(kolor.getSelectedItem());
String tekst = "Odpowiedź prawidłowa";
String tekst2 = "Odpowiedź błędna";
int duration = Toast.LENGTH_LONG;
Toast toast = Toast.makeText(this, tekst, duration);
Toast toast2 = Toast.makeText(this, tekst2, duration);
       if(wybor.equals("zielony"))
            toast.show();
        else
            toast2.show();
        }
    }
```

Listing 6.65 Pełny kod aktywności aplikacji Aplikacja11

A aplikacja po uruchomieniu będzie wyglądać następująco:



Rysunek 6.51 Komunikat Toast wyświetlający się po wybraniu prawidłowej odpowiedzi na pytanie

6.6.7 ListView

Kod aplikacji znajdziesz w folderze **Aplikacja12**.

Kontener **ListView** wyświetla informację w postaci listy. Jej kod przypomina **kontener Spinner**. Kod układu będzie wyglądał tak jak poniżej:

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

    <ListView
        android:id="@+id/lista"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:entries="@array/opcje"
        />
</LinearLayout>
```

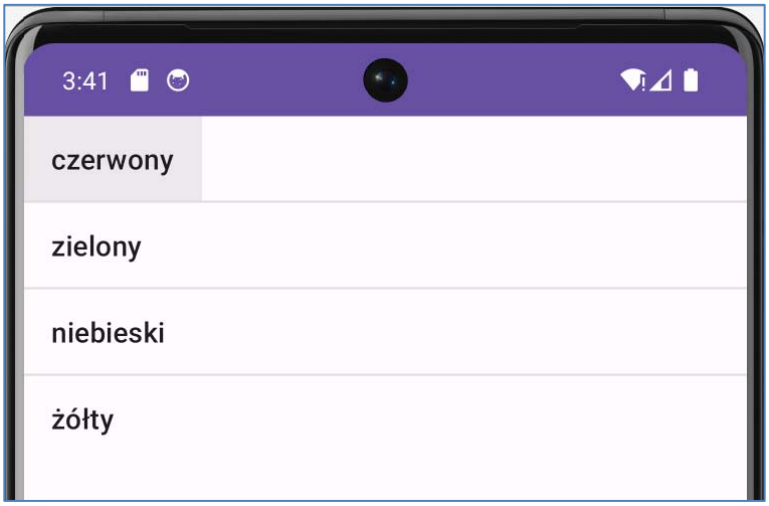
Listing 6.66 Kod układu zawierający element ListView

W dalszej kolejności zajmiemy się zdefiniowaniem opcji umieszczonych na liście. Będziemy to robić w pliku strings.xml. Podobnie jak w przypadku **Spinnera**. Kod umieszczony w pliku **strings.xml** będzie wyglądał w ten sposób:

```
<resources>
  <string name="app_name">Aplikacja12</string>
    <string-array name="opcje">
        <item>czerwony</item>
        <item>zielony</item>
        <item>niebieski</item>
        <item>żółty</item>
    </string-array>
</resources>
```

Listing 6.67 Zawartość pliku strings.xml

Po uruchomieniu, okno aplikacji powinno przypominać poniższy rysunek:



Rysunek 6.52 Aplikacja prezentująca działanie ListView

6.6.8 AutoCompleteTextView

Jest to kontrolka, której zadanie jest wyświetlanie podpowiedzi dla użytkownika. Podpowiedzi czerpane są z tablicy przygotowanej przez programistę aplikacji.

Kod aplikacji znajdziesz w folderze **Aplikacja13**.

W pierwszej kolejności stworzymy plik układu **activity_main.xml** w którym umieścimy dwa widoki: **TextView**, oraz **AutoCompleteTextView**.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Wybierz miesiąc: "
        android:id="@+id/tekst1"
        android:textSize="25sp" />
    <AutoCompleteTextView
        android:id="@+id/tekst2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:completionThreshold="1"/>
</LinearLayout>
```

Listing 6.68 Kod układu z widokiem AutoCompleteTextView

Atrybut **android:completionThreshold** określa po ilu wprowadzonych znakach zostaje wyświetlona podpowiedź. Teraz będziemy tworzyć kod aktywności. Po pierwsze musimy przygotować listę z której będziemy mieli możliwość wyboru miesiąca.

```
private static final String[] Miesiac = {
            "Styczeń",
            "Luty",
            "Marzec",
            "Kwiecień",
            "Maj",
            "Czerwiec",
            "Lipiec",
            "Sierpień",
            "Wrzesień",
            "Październik",
            "Listopad",
            "Grudzień"
};
```

Listing 6.69 Tablica definiująca elementy listy

Jest to prywatna statyczna tablica łańcuchów. Każdą pozycję na liście zapisujemy w cudzysłowie i oddzielamy przecinkami. W dalszej części definiujemy adapter, oraz tworzymy obiekt obsługujący komponent **AutoCompleteTextView**. W tym miejscu musimy wspomnieć o **adapterze**. Czym jest adapter i po co go stosujemy?

Adapter to nic innego jak łącznik. **Adapter** łączy źródło z którego pochodzą dane i widok umieszczony w układzie.

W przypadku naszej aplikacji łączymy tablicę typu **String** z pliku aktywności z komponentem **AutoCompleteTextView** znajdującym się w pliku układu. Adaptera nie będziemy potrzebować jeżeli dane będą pochodzić z innego pliku xml np. z pliku **strings.xml.** – Jak w przypadku **Spinnera**. W pliku aktywności tworzymy adapter dla danych typu **String** wygląda on w następujący sposób:

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>
(this,android.R.layout.simple_dropdown_item_1line,
Miesiac);
```

Listing 6.70 Adapter dla danych typu String

Z pomocą przychodzi nam klasa generyczna **ArrayAdapter** dla typu **String**. Zastosowany zasób **android.R.layout.simple_dropdown_item_1line** spowoduje utworzenie wpisów na liście. Jest to plik, który został wcześniej

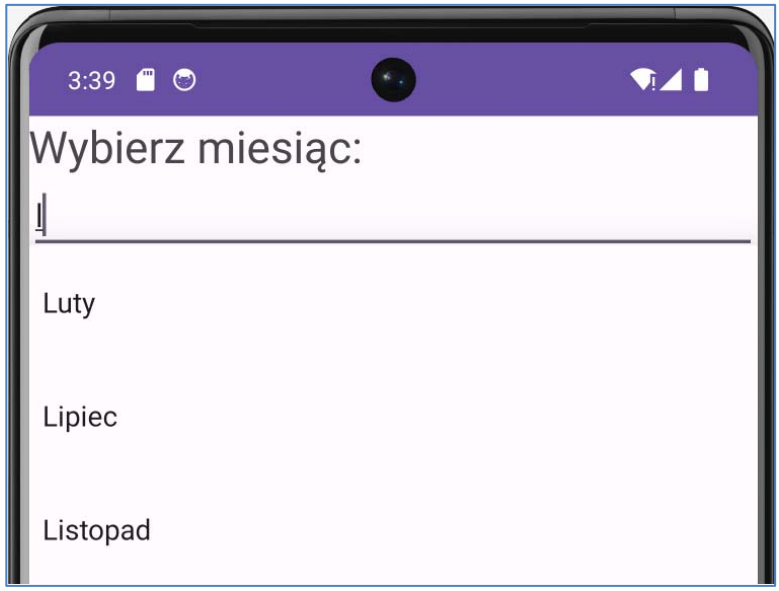
zdefiniowany my korzystamy z niego jako „gotowiec”. Zasób ten możemy równie dobrze zdefiniować sami.

Ostatnim argumentem jest **Miesiac** czyli nazwa tablicy w której znajdują się podpowiedzi do listy. Pozostaje nam tylko uzyskanie referencji do kontrolki poprzez metodę **findViewById** gdzie podajemy id nadane w pliku układu. Ostatnia linijka kodu to ustawienie **adaptera**. Kompletny plik aktywności będzie wyglądać tak jak poniżej:

```
package przyklad.nr13;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.ArrayAdapter;
import android.widget.AutoCompleteTextView;
public class MainActivity extends AppCompatActivity {
    private static final String[] Miesiac = {
                "Styczeń",
                "Luty",
                "Marzec",
                "Kwiecień",
                "Maj",
                "Czerwiec",
                "Lipiec",
                "Sierpień",
                "Wrzesień",
                "Październik",
                "Listopad",
                "Grudzień"
        };
   @Override
  protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);

ArrayAdapter<String> adapter = new ArrayAdapter<>(this,
android.R.layout.simple_dropdown_item_1line, Miesiac);

AutoCompleteTextView textView =
findViewById(R.id.tekst2);
   textView.setAdapter(adapter);
 }
}
```

Listing 6.71 Kod aktywności aplikacji o nazwie Aplikacja13

Wynik działania aplikacji przedstawiono na rysunku poniżej.



Rysunek 6.53 Działanie kontrolki AutoCompleteTextView

6.7 Paski postępu

6.7.1 ProgressBar

Kod aplikacji znajdziesz w folderze **Aplikacja14**.

Gdy aplikacja potrzebuje czasem więcej czasu na reakcje potrzebować będziemy ikonek, które będą pojawiać się na ekranie w czasie oczekiwania. Pierwszym z widżetów tego typu będzie **ProgressBar**. Będzie on wyglądał jak obracający się, kolorowy fragment koła.

Umieszczamy go jako widok w układzie. Przykładowy kod będzie przedstawiony w Listiningu poniżej:

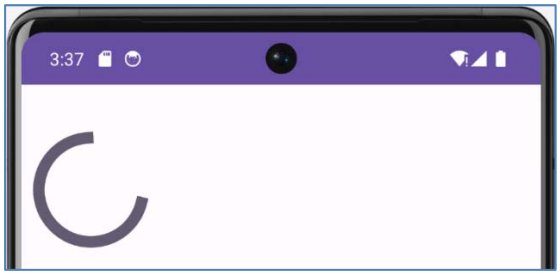
```
<ProgressBar
    android:id="@+id/progressBar"
    style="?android:attr/progressBarStyle"
    android:layout_width="110dp"
    android:layout_height="wrap_content"
    android:layout_weight="1"
/>
```

Listing 6.72 Pasek postępu ProgressBar w pliku układu

Na uwagę zasługuje tutaj **atrybut style**. Inne opcje stylu to np.:

- Widget.ProgressBar.Horizontal;
- Widget.ProgressBar.Small;
- Widget.ProgressBar.Large;
- Widget.ProgressBar.Inverse;
- Widget.ProgressBar.Small.Inverse;
- Widget.ProgressBar.Large.Inverse;

Warto przetestować wszystkie dostępne style. Aplikacja po uruchomieniu:



Rysunek 6.54 Pasek postępu ProgressBar

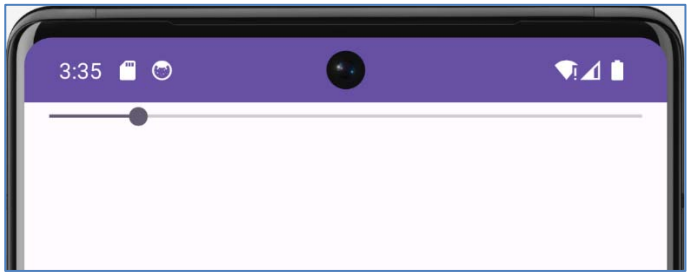
6.7.2 SeekBar

SeekBar jest rodzajem **suwaka**. Wyposażony jest w poziomą linię i okrągły suwak, który przesuwamy. Kod aplikacji znajdziesz w folderze **Aplikacja15**. W kodzie układu **SeekBar** wygląda w ten sposób:

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

    <SeekBar
        android:id="@+id/seekBar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1" />
</LinearLayout>
```

Listing 6.73 Element SeekBar w pliku układu

W aplikacji będzie on wyglądał następująco:



Rysunek 6.55 Pasek postępu SeekBar

Za pomocą **aktywności** możemy dodać do elementu **SeekBar** komunikat **Toast**, który będzie informował o tym jaki procent całości jest zaznaczony poprzez widok.

Kod układu będzie przypominał ten poniżej:

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

    <SeekBar
        android:id="@+id/seekBar"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="5"
        android:max="1000"
        android:progress="150"/>

</LinearLayout>
```

Listing 6.74 Kod układu aplikacji Aplikacja15

Na uwagę zasługuję atrybuty widoku **SeekBar**:

- **Android:max** – będzie on wyznaczał maksymalną wartość jaką może wybrać użytkownik.
- **Android:progres** – wartość ustawiona w momencie uruchomienia aplikacji.

Przykładowy kod aplikacji będzie wyglądał w ten sposób:

```
package przyklad.nr15;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.SeekBar;
import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        SeekBar seekbar;
        seekbar= findViewById(R.id.seekBar);
        seekbar.setOnSeekBarChangeListener(new
SeekBar.OnSeekBarChangeListener() {
    int postep = 1;
public void onProgressChanged(SeekBar seekBar, int
progress, boolean fromUser) {
     postep = progress;
}

public void onStartTrackingTouch(SeekBar seekBar) {
}

public void onStopTrackingTouch(SeekBar seekBar) {
                Toast.makeText(MainActivity.this,
"Maksymalna cena to: " + postep + " zł",
                        Toast.LENGTH_SHORT).show();
   }
  });
 }
}
```

Listing 6.75 Przykładowy kod aktywności wykorzystujący element SeekBar

Po pierwsze tworzymy obiekt klasy **SeekBar**:

```
SeekBar seekbar;
```

Listing 6.76 Deklaracja obiektu klasy SeekBar

Do zmiennej **seekbar** ustawiamy **referencję** obiektu z pliku układu.

```
seekbar=findViewById(R.id.seekBar);
```

Listing 6.77 Referencja do elementu SeekBar.

Następnie ustawiamy tzw. **Obiekt nasłuchujący** (setOnSeekBarChangeListener), który będzie reagował na zmiany na pasku. Zmienna typu całkowitego **int** o nazwie **postęp** będzie ustawiała wartość liczbową.

```
seekbar.setOnSeekBarChangeListener(new
SeekBar.OnSeekBarChangeListener() {
int postep = 0;
```

Listing 6.78 Obiekt nasłuchujący setOnSeekBarChangeListener

W dalszej kolejności wykorzystujemy trzy metody. Pierwsza z nich to:

```
public void onProgressChanged(SeekBar seekBar, int
progress, boolean fromUser)
{
   postep = progress;
}
```

Listing 6.79 Definicja metody onProgressChanged.

Parametrami tej metody są:

- **obiekt klasy SeekBar** – którego dotyczy aplikacja;
- **zmienna całkowita** – przechowująca wartość zaznaczoną przez **komponent;**
- **zmienna typu boolean** – jest zależna od działania użytkownika i zmienia się wraz z przesunięciem suwaka.

W metodzie zmiennej postęp, przypisujemy wartość pobraną z paska **SeekBar**. Druga metoda **public void onStartTrackingTouch(SeekBar seekBar)**. Ma jeden parametr i jest to zmienna typu **SeekBar**. Funkcja określa początek działania widoku **SeekBar**.

Ostatnia metoda to public **void onStopTrackingTouch(SeekBar seekBar)**. Ona podobnie jak poprzednia ma również jeden parametr czyli nazwę obiektu klasy **SeekBar**, którego dotyczy aplikacja. Funkcja ma za zadanie zadziałać w momencie, kiedy użytkownik ustawi na suwaku wartość. W tej metodzie tworzymy komunikat **Toast** i go definiujemy.

```
Toast.makeText(MainActivity.this, "Maksymalna cena to: "
+ postep + " zł",
Toast.LENGTH_SHORT).show();
```

Listing 6.80 Definicja elementu Toast.

Po uruchomieniu aplikacji i przesunięciu suwaka na dole aplikacji pojawi się komunikat Toast. Pełny kod aktywności jest następujący:

```
package przyklad.nr15;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.SeekBar;
import android.widget.Toast;
public class MainActivity extends AppCompatActivity {
    @Override
 protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 SeekBar seekbar;
 seekbar = findViewById(R.id.seekBar);

 seekbar.setOnSeekBarChangeListener(new
SeekBar.OnSeekBarChangeListener() {
         int postep = 1;

public void onProgressChanged(SeekBar seekBar, int
progress, boolean fromUser) {
    postep = progress;
}

public void onStartTrackingTouch(SeekBar seekBar) {

}

public void onStopTrackingTouch(SeekBar seekBar) {
                Toast.makeText(MainActivity.this,
"Maksymalna cena to: " + postep + " zł",
                        Toast.LENGTH_SHORT).show();

   }
  });
 }
}
```

Listing 6.81 Pełny kod aktywności

6.7.3 RatingBar

RatingBar umożliwia wystawienie oceny. Ma on postać pięciu gwiazdek. Kod aplikacji znajdziesz w folderze **Aplikacja16**. W pliku układu umieszczamy RatingBar w następujący sposób:

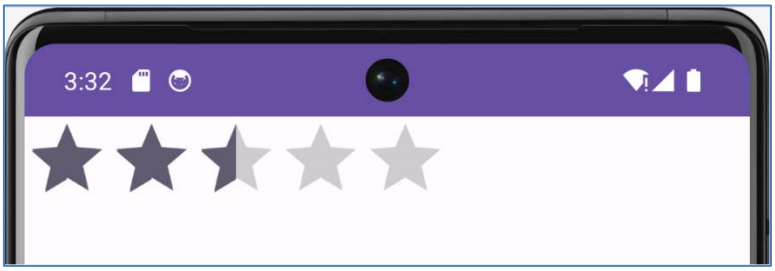
```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

<RatingBar
  android:id="@+id/RatingBar"
  style="@style/Widget.AppCompat.RatingBar"
  android:layout_width="wrap_content"
  android:layout_height="wrap_content"
  android:numStars="5"
  android:stepSize="0.5"/>
</LinearLayout>
```

Listing 6.82 Układ zawierający element RatingBar

Zwróćmy uwagę na atrybuty:

- **android:numStars** – będzie on określał maksymalną ilość gwiazdek, które będą wyświetlana.
- **android:stepSize** – będzie mówił o ile można podnieść ocenę. W przypadku tego kodu będzie to wartość pół gwiazdki.

Aplikacja będzie wyglądać tak jak na poniższym rysunku:



Rysunek 6.56 System ocen RatingBar

6.8 Intencje

W tym ćwiczeniu zajmiemy się komunikacją pomiędzy aktywnościami. Każda aplikacja mobilna złożona jest zazwyczaj z większej ilości aktywności. Do komunikacji pomiędzy dwoma aktywnościami służą intencje.

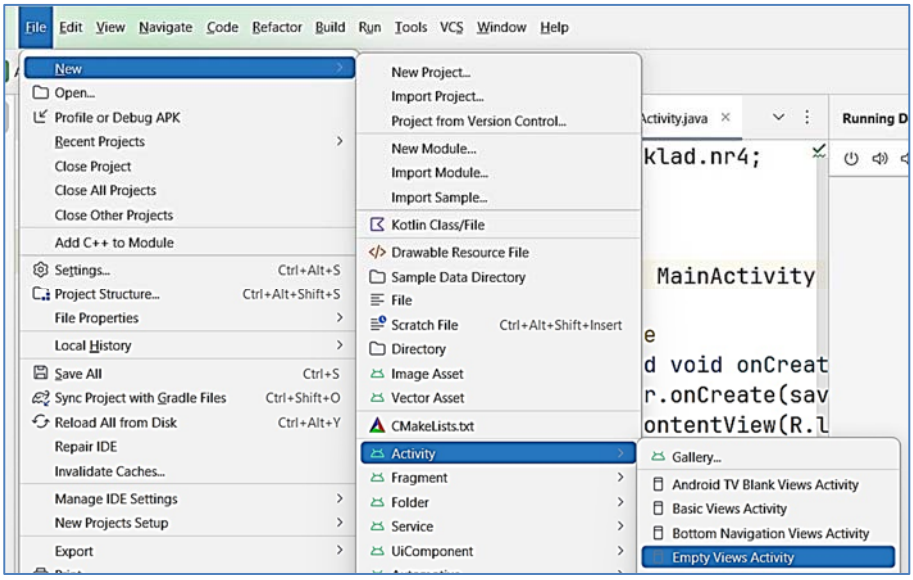
6.8.1 Intencja bez przekazywania wartości

Zaczniemy od prostego przykładu. Napiszemy aplikację w której na początek umieścimy przycisk Button. Kod do przykładu znajdziesz w folderze o nazwie **Aplikacja17**. Kod układu będzie wyglądał następująco:

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

    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/przycisk"
        android:onClick="intencja"
        />
</LinearLayout>
```

Listing 6.83 Kod układu z przyciskiem Button

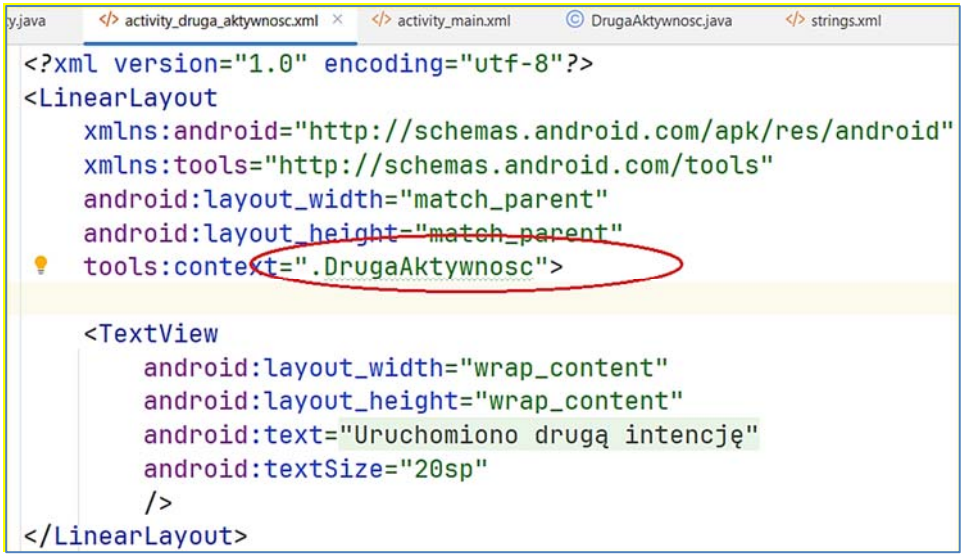
Jak wspomniano **intencja** służy do przesyłania danych pomiędzy aktywnościami. Wynika z tego, że w aplikacji będzie potrzebna druga **aktywność** i drugi **plik układu**. Przechodzimy zatem do lewej strony edytora. Z drzewa zasobów rozwijamy **gałąź java**. Klikamy na nią prawym przyciskiem myszy i wybieramy po kolei: **New → Activity → Empty Views Activity**.



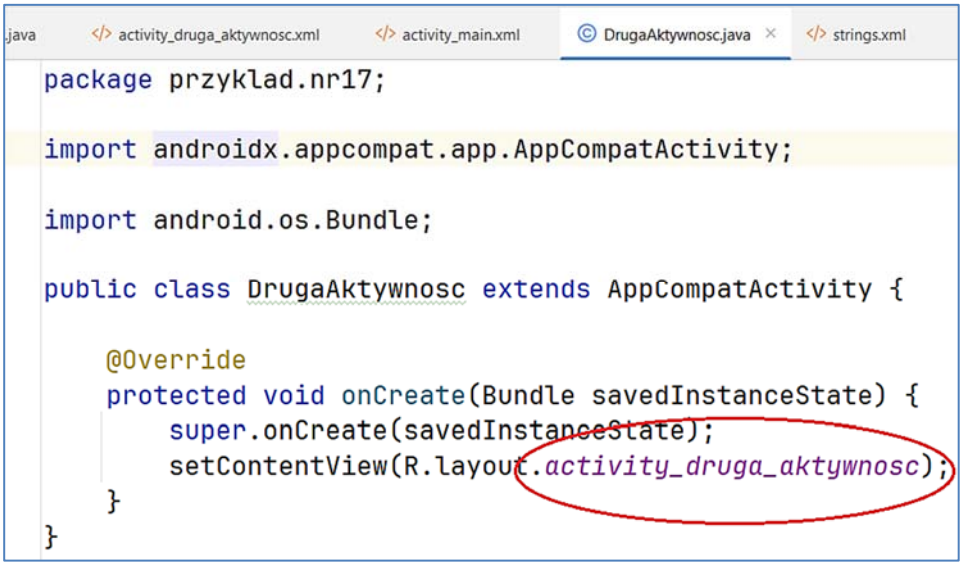
Rysunek 6.57 Tworzenie nowej aktywności

W oknie, które się otworzy wpisujemy nazwę nowej Aktywności - **DrugaAktywnosc**. Zwróć uwagę, że wraz z aktywnością tworzy się plik układu o nazwie **activity_druga_aktywnosc.xml**. Będzie to układ współpracujący z naszą Aktywnością. Wystarczy kliknąć **Finish**. W programie mamy **dwie Aktywności** i **dwa Układy**.

Gdy otworzymy teraz plik Układu **activity_druga_aktywnosc.xml** i plik Aktywności **DrugaAktywnosc.java** możemy zobaczyć, że są one ze sobą połączone.



Rysunek 6.58 Kod układu activity_druga_aktywnosc.xml z zaznaczonym odniesieniem do pliku aktywności



Rysunek 6.59 Kod aktywności DrugaAktywnosc.java z zaznaczonym odniesieniem do pliku układu

Zajmiemy się teraz utworzeniem układu drugiej aktywności. Będziemy wypisywać napis dlatego potrzebny będzie nam widok **TextView**.

Kod drugiego układu:

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/intencja"
        android:textSize="20sp"
        />
</LinearLayout>
```

Listing 6.84 Kod układu activity_druga_aktywnosc.xml

Zajmiemy się teraz plikiem aktywności głównej. Definiujemy w niej metodę **intencja**, która uruchomi się po wciśnięciu przycisku **Button**.

```
public void intencja(View view) {

    startActivity(new Intent(this,
DrugaAktywnosc.class));
}
```

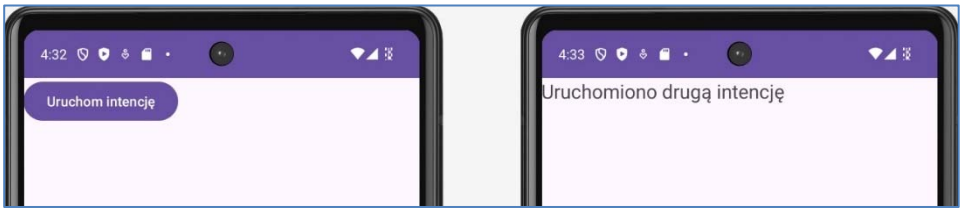
Listing 6.85 Deklaracja metody intencja

Uruchamiamy intencję za pomocą **startAcivity**. Tworzymy obiekt klasy **Intent**. **Konstruktor** tej klasy ma dwa argumenty: pierwszy to **aktywność** w której uruchamiamy **intencje**, druga to nazwa **aktywności docelowej**, czyli tej do której wysyłamy **intencje**.

```
Kod pliku MainActivity.java
package przyklad.nr17;

import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
public class MainActivity extends AppCompatActivity {

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

public void intencja(View view) {
startActivity(new Intent(this, DrugaAktywnosc.class));
 }
}
```

Listing 6.86 Kod aktywności

Uruchamiamy program. Aplikacja będzie wyglądać następująco:



Rysunek 6.60 Aplikacja przed (obrazek po lewej stronie, oraz po (obrazek po prawej stronie) uruchomieniu intencji.

6.8.2 Intencja przekazująca dane

Kolejna aplikacja również będzie wykorzystywała **intencje**. Jednak pomiędzy aktywnościami zostanie przekazana informacja. Kod aplikacji znajdziesz

w folderze **Aplikacja17a**. Program będzie pobierał od użytkownika informacje z pola tekstowego i przesyłał do drugiej aktywności.

W pliku układu **activity_main.xml** umieścimy dwa pola **EditText**, oraz przycisk **Button**. W układzie **activity_druga_aktywnosc.xml** pobrane dane wyświetlimy w dwóch polach **TextView**. Kod układu **activity_main.xml**:

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

    <EditText
        android:id="@+id/ulica"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint1"
        android:inputType="textCapSentences" />
    <EditText
        android:id="@+id/nr_domu"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint2"
        android:inputType="number"
        />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/przycisk"
        android:onClick="intencja"
        />
</LinearLayout>
```

Listing 6.87 Pełny kod układu głównego

Kod układu **activity_druga_aktywnosc.xml**:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/napis"
        android:textSize="20sp"
        />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/tekst1"
        android:text=" "
        android:textSize="20sp"
        />
</LinearLayout>
```

Listing 6.88 Pełny kod układu drugiej aktywności

Przejdziemy teraz do pliku aktywności głównej **MainActivity.java**. Zdefiniujemy **metodę intencja**, która uruchomi się po wciśnięciu przycisku **Button**.

```
public void intencja(View view) {
    EditText ul = findViewById(R.id.ulica);
    EditText nrd = findViewById(R.id.nr_domu);
    String ul2 = ul.getText().toString();
    int nrd2 =
Integer.parseInt(String.valueOf(nrd.getText()));
    Intent nowaIntencja = new Intent(this,
DrugaAktywnosc.class);
    nowaIntencja.putExtra("ulica", ul2);
    nowaIntencja.putExtra("nr_domu", nrd2);
    startActivity(nowaIntencja);
}
```

Listing 6.89 Definicja metody intencja przekazującej dane

Tworzymy referencję do pół **EditText**, które pobierają dane od użytkownika. Następnie pobrane wartości musimy przekonwertować na odpowiednie typy.

```
EditText ul = findViewById(R.id.ulica);
EditText nrd = findViewById(R.id.nr_domu);
```

Listing 6.90 Tworzenie referencji do pól EditText

W przypadku ulicy będzie to typ **String**. W przypadku **nr domu** będzie to wartość liczbowa **int**.

```
String ul2 = ul.getText().toString();
int nrd2 =
Integer.parseInt(String.valueOf(nrd.getText()));
```

Listing 6.91 Konwersja pobranych danych

Tworzymy intencje o nazwie **nowaIntencja.**

```
Intent nowaIntencja = new Intent(this,
DrugaAktywnosc.class);
```

Listing 6.92 Tworzenie intencji

Kolejnym krokiem jest przygotowanie **intencji** do przekazania danych. Potrzebna będzie metoda **putExtra**, która wyśle wartości do **drugiej aktywności**.

```
nowaIntencja.putExtra("ulica", ul2);
nowaIntencja.putExtra("nr_domu", nrd2);
```

Listing 6.93 Intencja przekazująca informacje

Pierwszym argumentem jest tzw. **klucz**. Jest to nazwa po której będzie można zidentyfikować dane. **Drugi argument** to wartość, która jest przekazywana do drugiej aktywności. W przypadku tej aplikacji są to nazwy dwóch zmiennych, które przechowują pobrane od użytkownika wartości. Trzeba jeszcze uruchomić **intencje**:

```
startActivity(nowaIntencja);
```

Listing 6.94 Uruchomienie intencji

Aktywność odbierająca **intencje** czyli klasa **DrugaAktywnosc.java** będzie wyglądała następująco:

```
package przyklad.nr17a;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;

public class DrugaAktywnosc extends AppCompatActivity {

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

Bundle bundle = getIntent().getExtras();
TextView komunikat = findViewById(R.id.tekst1);

if (bundle != null) {
String napis = bundle.getString("ulica");
 int liczba = bundle.getInt("nr_domu");
 komunikat.setText("Twoje miejsce zamieszkania to:
\nulica: " + napis + " nr "+ liczba);

  }
 }
}
```

Listing 6.95 Kod pliku DrugaAktywnosc.java

Wszystkie najważniejsze operacje dzieją się w **klasie onCreate**. Na początek tworzymy obiekt klasy **Bundle**, który pobierze **intencję** i dodatkowe informację.

```
Bundle bundle = getIntent().getExtras();
```

Listing 6.96 Tworzenie obiektu Bundle pobierającego dane z intencji

Następnie tworzymy referencję do pola tekstowego w którym wyświetlana będzie informacja pobrana z głównej aktywności.

```
TextView komunikat = findViewById(R.id.tekst1);
```

Listing 6.97 Tworzenie referencji do elementu układu

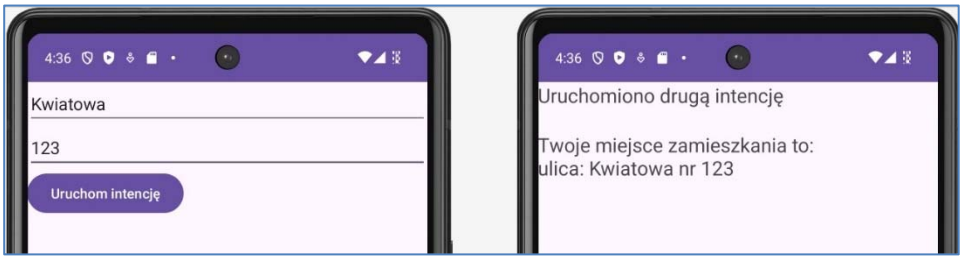
Instrukcja warunkowa sprawdzi w warunku czy użytkownik wpisał dane. Jeżeli tak to do dwóch zmiennych **String napis** i **int liczba** zostaną przekazane wartości z pierwszej aktywności. Za to działanie odpowiadać będą dwie metody **get.String** i **get.Int**.

```
if (bundle != null) {
 String napis = bundle.getString("ulica");
 int liczba = bundle.getInt("nr_domu");
```

Listing 6.98 Instrukcja warunkowa sprawdzająca czy użytkownik uzupełnił formularz

Metody getString i **getInt** są w pewnym sensie przeciwieństwem metod z pierwszej aktywności: **putString** i **putInt**. Metody z przedrostkiem **put** wysyłają dane, a **get** pobierają.

Ostatnim zadaniem jest wypisanie pobranych danych w polu **TextView**. Działanie aplikacji można sprawdzić na poniższym rysunku:



Rysunek 6.61 Wynik działania aplikacji wykorzystującej intencje z przesyłaniem danych.

6.8.3 Intencja niejawna

W przykładzie, który przed chwilą omówiliśmy mieliśmy do czynienia z tzw. **intencją jawną**. Intencja jawna jest typem intencji, która trafia w konkretne z góry ustalone miejsce. W przypadku naszego zadania była to **druga aktywność**. Intencja została wysłana z **aktywności pierwszej** do **drugiej**.

Miejsce w którym została odebrana **intencja** było **jawne** i z góry ustalone. Podaliśmy **intencji** informację, którą **klasę** ma uruchomić.

```
Intent intent = new Intent(this, DrugaAktywnosc.class);
```

Listing 6.99 Tworzenie intencji

W tworzeniu aplikacji mobilnych bywa czasem tak, że tworzymy **intencje**, ale do końca nie wiemy gdzie ta intencja trafi. Ma to miejsce wówczas, gdy intencje wysyłamy do innej aplikacji. Wyobraźmy sobie, że aplikacja którą tworzymy ma za zadanie wysłać wiadomość. Wiemy tylko, że ma to być wiadomość w formie tekstu, nie wiemy czy ma to być **wiadomość SMS** czy **e-mail**. W chwili kliknięcia przycisku wysyłającego na ekranie pojawi się lista aplikacji, które umożliwią nam wysłanie wiadomości. Będzie to np. program do wysłania wiadomości e-mail.

W przypadku **intencji niejawnej** to Android ma zadecydować, o doborze odpowiedniej aktywności. Pomaga mu w tym filtr intencji. Jest on zdefiniowany w **pliku manifestu**. Jego kod będzie wyglądał podobnie do tego poniżej.

```
<intent-filter>
<action android:name="android.intent.action.MAIN" />
<category
android:name="android.intent.category.LAUNCHER"/>
</intent-filter>
```

Listing 6.100 Deklaracja filtra intencji niejawnej

Intencje niejawną tworzymy wykorzystując następujący kod, który umieszczamy w odpowiedniej aktywności.

```
Intent intent = new Intent(działanie);
```

Listing 6.101 Tworzenie intencji niejawnej

W miejscu działania wpisujemy rodzaj akcji, która ma się wydarzyć po uruchomieniu intencji np. **Intent.ACTION_VIEW**, **Intent.ACTION_SEND**. Pierwsza stała będzie pomocna przy wyświetlaniu widoku, a druga pozwoli na wysłanie np. wiadomości.

6.9 Układy

Układ to sposób ułożenia elementów w aplikacji. Do tej pory korzystaliśmy z jednego rodzaju układu. Był to układ liniowy – **LinearLayout**. Jednak aplikacje możemy tworzyć również z wykorzystaniem innych dostępnych układów. Zaczniemy jednak od przypomnienia i dokładnego omówienia układu liniowego.

6.9.1 Linear Layout

Podstawową zasadą w układzie liniowym jest to, że elementy układane są **jeden pod drugim** lub **jeden obok drugiego** w kolejności w jakiej są umieszczone w **pliku xml**. Jeżeli chodzi o sposobie wyświetlenia elementów największy wpływ będzie miał tutaj parametr **android:orientation**.

6.9.2 FrameLayout

Kolejnym układ to **FrameLayout**. Jego cechą charakterystyczną jest to, że elementy są układane **jeden na drugim** w kolejności odwrotnej do tego w jakiej są umieszczone w pliku **układu xml**. Na elementy ułożone w układzie ramki patrzymy tak jak na warstwy w pliku graficznym.

Do ilustracji działania **FrameLayout** posłużymy się przykładem. Mamy aplikacje w której umieścimy **trzy obrazki**. Każdy z obrazków to kwadrat

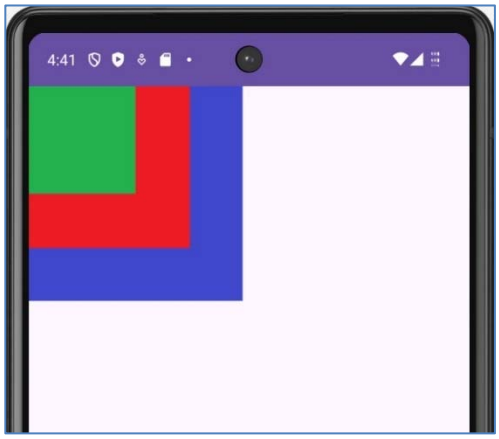
w określonym kolorze: niebieskim, czerwonym i zielonym. W pliku układu ułożone są one tak jak w tym kodzie poniżej.

Kod aplikacji znajdziesz w folderze **Aplikacja18**.

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/blue"
        />
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/red"
        />
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/green"
        />
</FrameLayout>
```

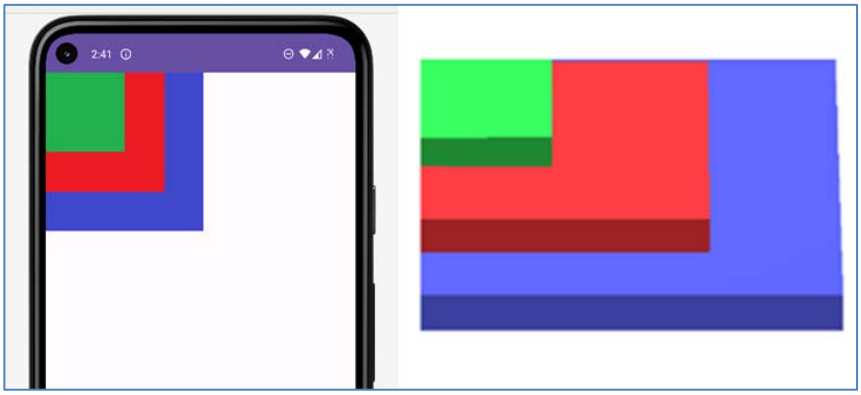
Listing 6.102 Układ FrameLayout

Po uruchomieniu aplikacja wygląda tak jak na ilustracji poniżej:



Rysunek 6.62 Aplikacja wykorzystująca układ liniowy - FrameLayout

Na dole znajduje się niebieski kwadrat, który w pliku układu jest umieszczony jako pierwszy. Gdybyśmy spojrzeli na obrazek z boku. Tak jakbyśmy trzymali telefon w poziomie **FrameLayout** wyglądałby tak jak na ilustracji poniżej.



Rysunek 6.63 Ułożenie elementów w układzie liniowym w modelu 3D

6.9.3 RelativeLayout

RelativeLayout określa położenie elementów układu na podstawie pozycji komponentu względem innych elementów. Jest to tzw. względny sposób rozmieszczanie elementów.

W układzie **RelativeLayout** możemy ustawić widoki względem elementów nadrzędnych. Ma to największe znaczenie, gdy projektujemy aplikacje dla urządzeń o różnej rozdzielczości i wielkości np. smartfon i tablet. Na obu typach urządzeń aplikacja powinna wyglądać tak samo, albo podobnie.

6.9.3.1 Rozmieszczenie elementów względem układu.

Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja19**.

Elementem nadrzędnym (tzw. rodzicem) dla widoku najczęściej jest układ. Dlatego też położenie np. przycisku będziemy odnosić do układu RelativeLayout.

W przykładzie poniżej umieścimy trzy przyciski: **Przycisk1, Przycisk2, Przycisk3** w trzech różnych miejscach wykorzystując **RelativeLayout**. Najpierw jednak przyjrzyjmy się efektowi działania **układu**.

Rysunek 6.64: Ułożenie przycisków po zastosowaniu układu RelativeLayout

Kod układu może wyglądać w ten sposób:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/przycisk1"
        android:text="@string/p1"
        android:onClick="Przycisk1"
        android:layout_alignParentBottom="true"
        android:layout_centerHorizontal="true"
        />
    <Button
        android:id="@+id/przycisk3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_alignParentLeft="true"
        android:onClick="Przycisk3"
        android:text="@string/p3" />

</RelativeLayout>
```

Listing 6.103 Układ RelativeLayout.

O położeniu elementów w układzie decydują linijki w których umieszczone są wpisy:

```
android:layout_alignParentLeft="true"
android:layout_centerHorizontal="true"
android:layout_alignParentBottom="true"
android:layout_centerHorizontal="true".
```

Listing 6.104 Atrybuty układu relatywnego.

Wszystkie powyższe parametry są ustawione na wartość **true**. Domyślnym ustawieniem tych atrybutów jest wartość **false** dlatego jeżeli chcemy je uaktywnić musimy zmienić ich wartość na przeciwną.

Zwróć uwagę na to, że w przypadku **RelativeLayout** nie ma znaczenia konieczność ułożenia elementów w pliku układu.

Przycisk1 ma ustawioną wartość **alignParentBottom**, która umieszcza element na dole ekranu, dodatkowo dołączona jest druga opcja: **center_Horizontal**, która umieszcza element dokładnie na środku w poziomie. Jak widać opcję mogą się ze sobą łączyć. Lista dostępnych opcji dla atrybutu **android:layout…** przedstawia poniższa tabela:

Atrybut	Działanie
android:layout_alignParentBottom	Umieszcza element na dole układu
android:layout_alignParentTop	Umieszcza element na górze układu
android:layout_alignParentLeft	Umieszcza element po lewej stronie
android:layout_alignParentRight	Umieszcza element po prawej stronie
android:layout_centerInParent	Umieszcza element dokładnie na środku układu
android:layout_centerHorizontal	Umieszcza element pośrodku układu w poziomie
android:layout_centerVertical	Umieszcza element pośrodku w pionie

Tabela 6.1 Wartości atrybutów wykorzystywanych w układzie RelativeLayout

Są to najczęściej używane opcje w układzie **RelativeLayout**. Porównaj teraz kod aplikacji z tabelą. Czy atrybuty przypisane widokom są zgodne z tabelą?

6.9.3.2 Rozmieszczenie elementów względem siebie.

Kod aplikacji znajdziesz w folderze **Aplikacja19a**. Jeżeli potrzebujemy umieścić dwa elementy w grupie np. obok siebie możemy również zastosować **RelativeLayout**. Musimy tylko określić od jakiego elementu zależy ten drugi. Zaprojektujmy dwa przyciski, które będą zależne od siebie – są ułożone jeden pod drugim.

Kod takiej aplikacji powinien być zbliżony do kodu przedstawionego na poniższym listingu:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/przycisk1"
        android:text="@string/p1"
        android:onClick="Przycisk1"
        android:layout_centerInParent="true"
        />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/przycisk2"
        android:text="@string/p2"
        android:onClick="Przycisk2"
        android:layout_alignLeft="@id/przycisk1"
        android:layout_below="@id/przycisk1"
        android:layout_centerHorizontal="true"
        />

</RelativeLayout>
```

Listing 6.105 Układ RelativeLayout z elementami zależnymi od siebie

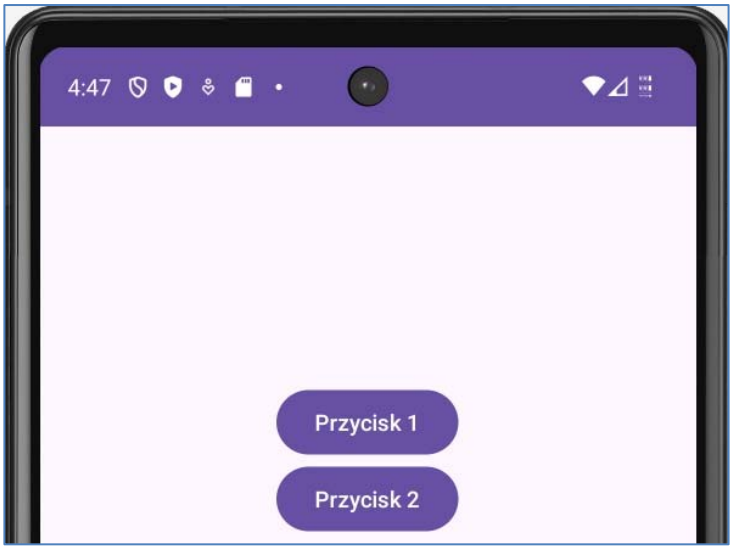
Jako element nadrzędny będziemy traktować **Przycisk1**. Ustawiony on jest za pomocą atrybutu **android:layout_centerInParent** na środku ekranu. Przycisk drugi został do niego przypisany za pomocą dwóch atrybutów:

- **instrukcji:android:layout_alignLeft=”@+id/przycisk1”** – za jej pomocą przycisk drugi jest wyrównany do lewej krawędzie przycisku pierwszego, oraz drugiej
- **instrukcji android:layout_below=”@+id/przycisk1” –** która ustawia przycisk pod przyciskiem1. Inne atrybuty obsługujące ustawienie elementów względem siebie przedstawia poniższa tabela.

Atrybut	Działanie
layout_above	Umieszcza widok powyżej nadrzędnego
layout_below	Umieszcza widok poniżej nadrzędnego
layout_aligntop	Wyrównuje widoki do górnej krawędzi
layout_alignbottom	Wyrównuje widoki do dolnej krawędzi
layout_alignleft, layout_alignstart	Wyrównuje widoki do lewej strony

layout_alignright, layout_alignend	Wyrównuje widoki do prawej krawędzi
layout_toleftof, layout_tostartof	Umieszcza element po lewej stronie elementu nadrzędnego
layout_torightof, layout_toendof	Umieszcza element po prawej stronie elementu nadrzędnego

Tabela 6.2 Wartości atrybutów układu RelativeLayotu.



Rysunek 6.65 Ułożenie przycisków w układzie RelativeLayout rozmieszczonych względem siebie.

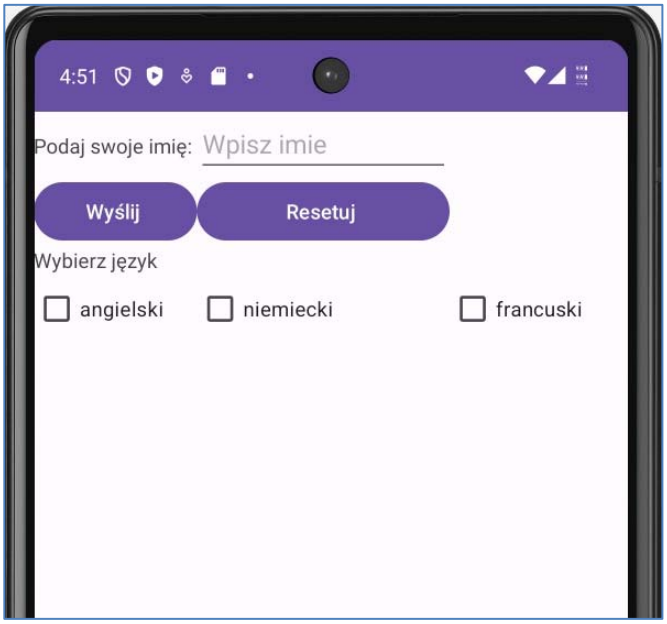
6.9.4 TableLayout

Układ tabelaryczny przedstawia układ w formie **tabeli**. Możemy utworzyć układ dodając do niego **wiersze**. Każdy dodany do wiersza element tworzy **komórkę**. Szerokość komórki jest równa szerokości najszerszego z elementów.

Kod aplikacji znajduje się w folderze o nazwie **Aplikacja20**. Utwórzmy prosty układ tabelaryczny składający się z:

- pierwszego wiersza w którym umieścimy napis i pole tekstowe do uzupełnienia;
- drugiego wiersza z dwoma przyciskami: wyślij i wyczyść;
- trzeciego wiersza z napisem;
- czwartego wiersza z trzema polami wyboru typu checkbox;

Układ będzie wyglądał jak na obrazie poniżej:



Rysunek 6.66 Widoki ułożone za pomocą układu TableLayout

Jak widać na rysunku utworzone zostały cztery wiersze elementów. Każdy z **wierszy** został podzielony na **kolumny**. Każdy nowododany element w wierszu tworzy kolumnę. Zwróć uwagę na kolumnę środkową. Została ona poszerzona do **175dp**, a w związku z tym wszystkie elementy w tej kolumnie mają tą samą szerokość, ponieważ pole tekstowe w pierwszym wierszu jest najszerszym elementem w kolumnie.

Kod układu będzie wyglądał tak jak poniżej:

```
<? xml version="1.0" encoding="utf-8"?>
<TableLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    tools:context=".MainActivity">
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <TextView
            android:id="@+id/imie"
            android:layout_column="0"
            android:text="@string/tekst1" />
        <EditText
            android:id="@+id/odpowiedz"
            android:layout_width="175dp"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:hint="@string/hint1"
            android:inputType="textAutoCorrect" />
```

```
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <Button
            android:id="@+id/button1"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="0"
            android:text="@string/tekst2" />
        <Button
            android:id="@+id/button2"
            android:layout_width="wrap_content"
            android:layout_height="wrap_content"
            android:layout_column="1"
            android:text="@string/tekst3" />
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">

        <TextView
            android:id="@+id/napis"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_column="0"
            android:text="@string/tekst4" />
    </TableRow>
    <TableRow
        android:layout_width="match_parent"
        android:layout_height="match_parent">
        <CheckBox
            android:id="@+id/angielski"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_column="0"
            android:text="@string/tekst5" />
        <CheckBox
            android:id="@+id/niemiecki"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_column="1"
            android:text="@string/tekst6" />
        <CheckBox
            android:id="@+id/francuski"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            android:layout_column="2"
            android:text="@string/tekst7" />
    </TableRow>
</TableLayout>
```

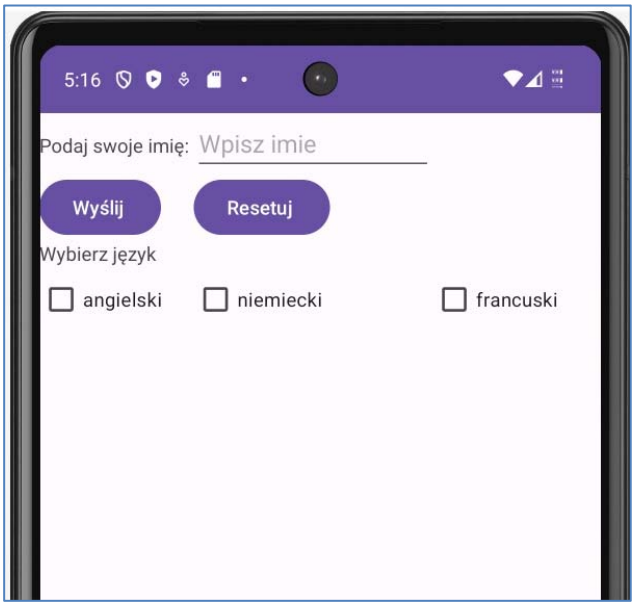
Listing 6.106 Układ TableLayout

Nowym atrybutem jest: **android:layout_column**. Przypisane do niego są kolejne wartości zaczynając od **0**. Wartości te oznaczają kolejność umieszczenia elementu w wierszu.

6.9.5 GridLayout

Kod poniższej aplikacji znajdziesz w folderze o nazwie **Aplikacja21a**. **GridLayuout** to układ siatki. Tworzy w układzie siatkę wierszy i kolumn dzięki którym można umieszczać elementy w układzie.

GridLayout podobnie jak **TableLayout** układa elementy w komórki. Różnica jest taka, że każdemu elementowi w układzie nadajemy **numer kolumny**, ale również i numer **wiersza**. Nie ma potrzeby separować wierszy jak w poprzednim układzie. Na rysunku przedstawiony jest wygląd aplikacji zawierający te same elementy jak w poprzednim ćwiczeniu z tą różnicą, że wykorzystujemy tutaj **GridLayout**.



Rysunek 6.67 Widoki rozmieszczone za pomocą układu GridLayout

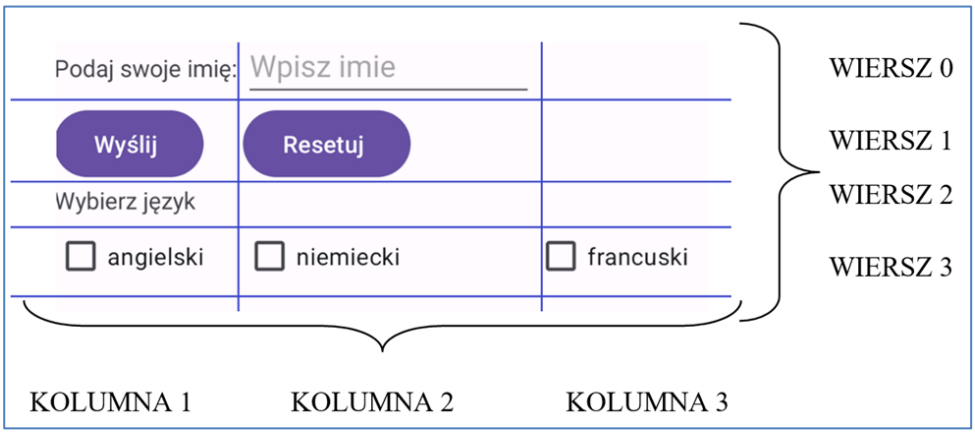
Jak widać układ wygląda dokładnie tak samo jak w poprzednim ćwiczeniu. Spójrz teraz na kod aplikacji oparty na **GridLayout**:

```
<?xml version="1.0" encoding="utf-8"?>
<GridLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    tools:context=".MainActivity"
    android:columnCount="3"
    android:rowCount="4">
```

```
    <TextView
        android:text="Podaj swoje imię: "
        android:id="@+id/imie"
        android:layout_column="0"
        android:layout_row="0"
        />
   <EditText
        android:hint="Wpisz imie"
        android:layout_width="175dp"
        android:layout_height="wrap_content"
        android:id="@+id/odpowiedz"
        android:layout_column="1"
        android:layout_row="0"
        android:inputType="textAutoCorrect" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Wyślij "
        android:id="@+id/button1"
        android:layout_column="0"
        android:layout_row="1"/>
    <Button
        android:layout_width="wrap_content"
        android:layout height="wrap content"
        android:id="@+id/button2"
        android:text="Resetuj "
        android:layout_column="1"
        android:layout_row="1"/>
    <TextView
        android:text="Wybierz język "
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/napis"
        android:layout_column="0"
        android:layout_row="2"/>
    <CheckBox
        android:text="angielski "
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/angielski"
        android:layout_column="0"
        android:layout_row="3"/>
    <CheckBox
        android:text="niemiecki "
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/niemiecki"
        android:layout_column="1"
        android:layout_row="3"/>
    <CheckBox
        android:text="francuski "
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:id="@+id/francuski"
        android:layout_column="2"
        android:layout_row="3"/>

</GridLayout>
```

Listing 6.107 Układ GridLayout

W układzie siatki umieszczamy wszystkie elementy w jednej sekcji bez podziału na wiersze. W każdym widoku dodajemy dodatkową linijkę kodu: **android:layout_row** i podobnie jak w **android:layout_column** nadajemy im wartości począwszy od 0. Każdy nowy wiersz ma kolejną o jeden większą wartość.



Rysunek 6.68 Schemat ułożenia widoków

Patrząc na nasz **układ** nie trudno dostrzec, że ma on pewne wady. W związku z tym, że kolumna druga ma ustawiona większą szerokość tzn. że pozostałe komórki (zwłaszcza w wierszach drugim i trzecim) są zbyt szerokie. **GridLayout** daje nam jednak możliwość scalenia dwóch komórek (zupełnie jak w arkuszu kalkulacyjnym bądź edytorze tekstowym). Wyobraźmy sobie, że nasz układ jest tabelą. Jeżeli weźmiemy pod uwagę, że **EditText** wygląda lepiej, gdy jest szerszy to tabela będzie wyglądać mniej więcej tak jak na schemacie poniżej:

TextView	EditText	
Button1	Button2	
TextView		
CheckBox1	CheckBox2	CheckBox3

Tabela 6.3 Schemat układu GridLayout

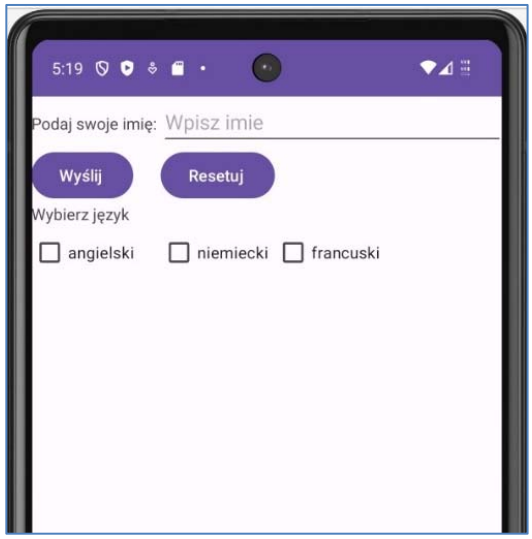
Kod aplikacji znajdziesz w folderze **Aplikacja21b**. Spróbujmy teraz dostosować nasz układ tak, żeby wyglądał lepiej.

Użyjemy w tym celu atrybut **android:layout_columnSpan=”2**” Atrybut ten informuje o tym, że komórka będzie zajmować obszar wielkości dwóch komórek. Gotowy kod aktywności po wprowadzonych zmianach będzie wyglądać tak:

```
<TextView
    android:text="Wybierz język "
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/napis"
    android:layout_column="0"
    android:layout_row="2"
    android:layout_columnSpan="2"/>
<CheckBox
    android:text="angielski "
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:id="@+id/angielski"
    android:layout_column="0"
    android:layout_row="3"/>
```

Listing 6.108 Układ GridLayout z zastosowaniem atrybutu android:layout_columnSpan

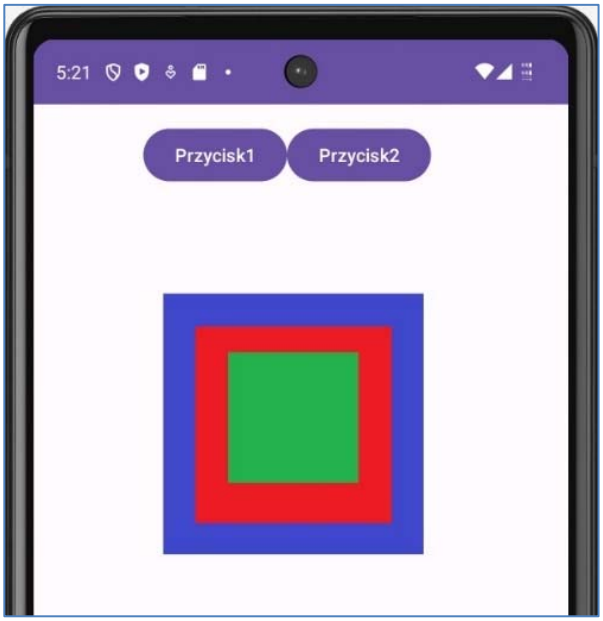
A po uruchomieniu aplikacja wygląda teraz tak:



Rysunek 6.69 Ułożenie widoków z zastosowaniem atrybutu columnSpan

6.9.6 Układy zagnieżdżone

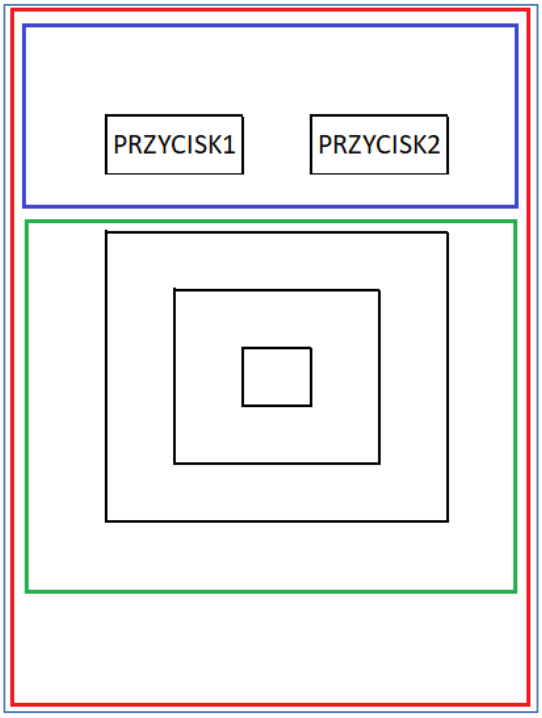
Kod utworzonej aplikacji znajdziesz w folderze o nazwie **Aplikacja22**. Spróbujmy stworzyć układ z poniższego rysunku:



Rysunek 6.70 Układ zagnieżdżony

Mamy tutaj dwa przyciski: **PRZYCISK1**, oraz **PRZYCISK2**, a także trzy kolorowe obrazki na których znajdują się trzy kolorowe kwadraty. Obrazki ułożone są jeden na drugim. W związku z tym najprościej będzie użyć w tym przypadku układu **FrameLayout**. Natomiast do ułożenia przycisków będzie nam potrzebny **LinearLayout**. Przy tego typu łączeniu układów najlepiej wykorzystać ich zagnieżdżenie.

Zagnieżdżenie układów polega na umieszczeniu jednego układu w środku drugiego. Zanim zaczniemy tworzyć kod powyższego układu spróbujmy wykorzystać poniższy schemat.



Rysunek 6.71 Schemat prezentujący zagnieżdżenie widoków

W pliku aktywności będziemy mieli **trzy układy**. Pierwszy z nich (oznaczony kolorem czerwonym) będzie **układem głównym** - **linearnym** o orientacji **vertical**. Chcemy, aby **dwa wewnętrzne układy** były ułożone jeden pod drugim. Układ ten składać się będzie z dwóch elementów. Pierwszym będzie kolejny **układ liniowy** (oznaczony kolorem niebieskim). Tym razem o orientacji **horizontal**, gdyż chcemy, żeby przyciski były ułożone obok siebie. Drugim **układ ramki** (kolor zielony) **FrameLayout** w którym umieścimy trzy obrazki. Trzeba jeszcze przesunąć elementy wykorzystując **parametry margin**.

Kod układu może wyglądać jak na poniższym listingu.

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

```
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="145dp"
        android:orientation="horizontal">

    <Button
        android:id="@+id/przycisk1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="85dp"
        android:layout_marginTop="15dp"
        android:onClick="Przycisk"
        android:text="Przycisk1" />

    <Button
        android:id="@+id/przycisk2"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="0dp"
        android:layout_marginTop="15dp"
        android:onClick="Przycisk2"
        android:text="Przycisk2" />
 </LinearLayout>

    <FrameLayout
        android:layout_width="match_parent"
        android:layout_height="257dp"
        android:layout_marginLeft="0dp"
        android:layout_marginTop="0dp">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="100dp"
        android:layout_marginTop="0dp"
        android:src="@drawable/blue" />

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginLeft="125dp"
        android:layout_marginTop="25dp"
        android:src="@drawable/red" />

     <ImageView
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_marginLeft="150dp"
         android:layout_marginTop="45dp"
         android:src="@drawable/green" />
    </FrameLayout>
</LinearLayout>
```

Listing 6.109 Kod aplikacji z układem zagnieżdżonym

6.9.7 ConstraintLayout

Kod aplikacji znajduje się w folderze o nazwie **Aplikacja23**.

ConstraintLayout czyli układ z ograniczeniami jest układem w którym elementy rozmieszczamy za pomocą **edytora graficznego**, a nie tak jak do tej pory poprzez umieszczanie kodu w **pliku xml**. Utwórzmy teraz nowy projekt. Domyślnie ustawionym układem będzie **ConstraintLayout**. Jego kod wygląda jak poniżej:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">
  </androidx.constraintlayout.widget.ConstraintLayout>
```

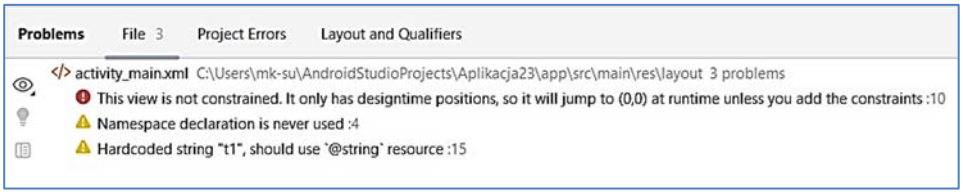
Listing 6.110 ConstraintLayout

Dodajmy do układu prosty widok tekstowy **TextView** w którym umieścimy tekst **Dzień dobry**.

```
<?xml version="1.0" encoding="utf-8"?>
<android.support.constraint.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Dzień dobry" />

</android.support.constraint.ConstraintLayout>
```

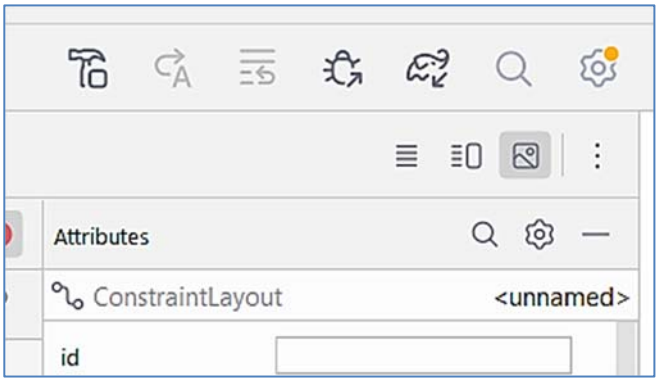
Listing 6.111 Układ ConstraintLayout z widokiem TextView

Gdy przyjrzymy się aplikacji zauważymy, że pojawia się **błąd**. Treść błędu to: **<TextView>: Missing Constraints in ConstraintLayout.** Element **TextView** nie ma ograniczenia – czyli **Constraints**.

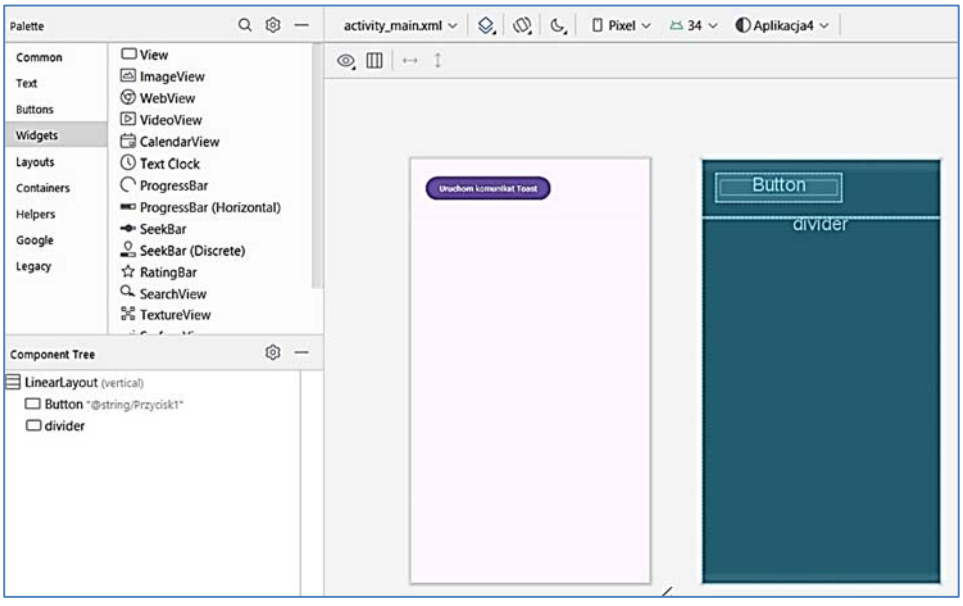


Rysunek 6.72 Błąd informujący o braku ograniczenia widoku

Usuńmy teraz **błąd** dodając **ograniczenia** dla **elementu**. W tym celu musimy zmienić **widok pliku xml**. Przejdźmy do zakładki **Design** znajdującej się w prawym górnym rogu edytora.

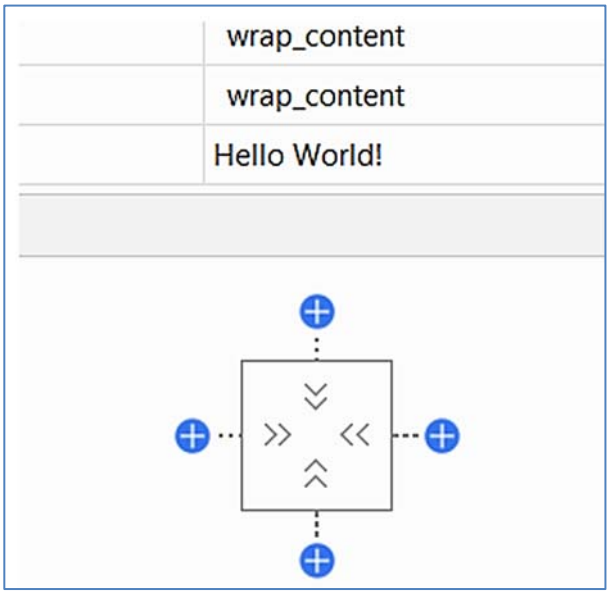


Rysunek 6.73 Ikony umożliwiające przemieszczanie się pomiędzy edytorami układu (tekstowym, połączonym, graficznym)



Rysunek 6.74 Edytor graficzny układu

W oknie widzimy nasz element TextView. Znajduje się on w lewym górnym rogu. Musimy nadać mu ograniczenie. Zaznaczamy widok klikając na niego. W oknie po prawej stronie pojawi się obszar o nazwie Layout.



Rysunek 6.75 Okienko umożliwiające ustawienie ograniczeń dla widoku

Ograniczenia nadamy poprzez kliknięcie **niebieskiego znaku plus**. Nadać musimy co najmniej dwa ograniczenia. Jedno pionowe i jedno poziome. Możemy również nadać ograniczenia dla wszystkich czterech obszarów: **góra, dół, lewo, prawo**.

Rysunek 6.76 Ustawianie ograniczeń dla widoku

Po nadaniu ograniczeń **znika błąd** brakującego ograniczenia. W oknie **Layout** widzimy pozycję widoku. W tej chwili jest on ustawiony domyślnie na wartości **0 i 0**. Oznaczają one margines elementu czyli odległość od krawędzi lewej i górnej. Element możemy przesunąć wybierając wartość marginesu z rozwijanej listy, bądź przesuwając element ręcznie w edytorze.

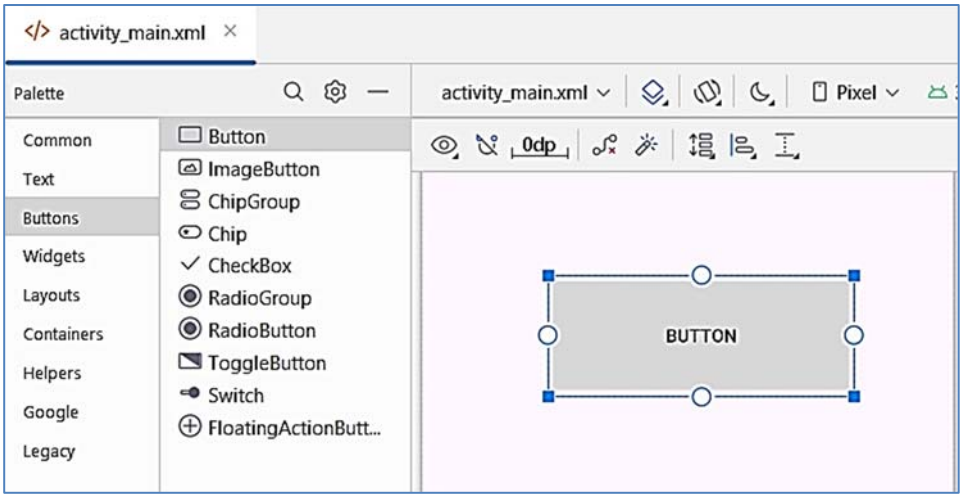
Klikamy na element i ustawiamy się w **jego środku**. Z **wciśniętym przyciskiem myszy przesuwamy element** w dowolne miejsce. Wartości marginesów zmienią się odpowiednio do położenia elementu. Zmiany możemy również podejrzeć w edytorze tekstowym. Kod układu będzie wyglądać tak:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_marginStart="176dp"
        android:layout_marginTop="260dp"
        android:text="@string/t1"
 app:layout_constraintStart_toStartOf="parent"
 app:layout_constraintTop_toTopOf="parent" />
</androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 6.112 Kod układu ConstraintLayout w edytorze tekstowym

Zauważmy, że atrybuty **app:layout_constraintStart_toStartOf** i **app:layout_constraintTop_toTopOf** znajdowały się domyślnie, kiedy utworzyliśmy projekt. Dodajmy teraz kolejny element wykorzystując układ z ograniczeniami i edytor graficzny.

Przechodzimy do zakładki **Design**. Po lewej stronie edytora mamy listę dostępnych widoków. Przejrzyj ją w celu sprawdzenia możliwości jakimi dysponujesz. My wstawimy **przycisk**. Wybieramy z listy **Button**. Z wciśniętym klawiszem myszki przesuwamy przycisk na pole układu.



Rysunek 6.77 Wstawianie przycisku Button do układu.

Teraz możemy ustawić położenie przycisku najpierw jednak musimy nadać mu ograniczenia. Postępujemy dokładnie tak samo jak z polem tekstowym.

6.10 Style

Chcemy utworzyć aplikację, która składa się z **dziewięciu przycisków** ułożonych w trzy rzędy po trzy kolumny. Przyciski mają mieć naprzemienne kolory – tak, aby wyglądały jak szachownica.

W kodzie układu umieścimy dziewięć **przycisków Button** i każdemu z nich przypiszemy odpowiednie ustawienia. Takie rozwiązanie ma dwie wady. Po pierwsze **zwielokrotniamy kod**, gdyż każdy z przycisków musimy opisać osobno, a poza tym gdyby zaszła **potrzeba drobnej zmiany jednego elementu** musielibyśmy zmiany wprowadzić w każdym elemencie osobno.

Możemy tutaj wykorzystać **style**. Style (podobnie jak w kaskadowych arkuszach styli css) umożliwiają zgrupowanie ustawień w jednym miejscu i przypisanie określonego stylu do widoku.

6.10.1 Tworzenie stylów

Kod aplikacji znajduje się w folderze o nazwie **Aplikacja24**.

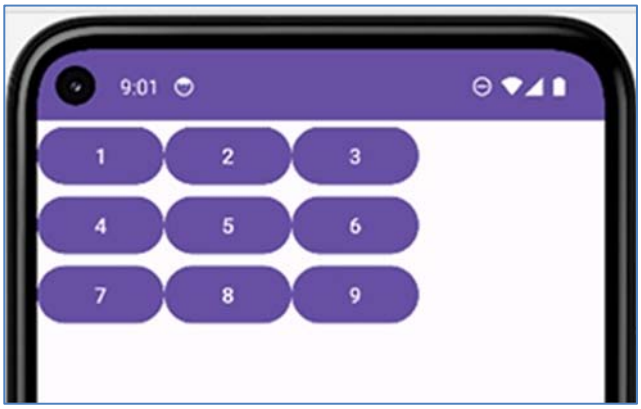
Tworzymy nową aplikację. Do rozmieszczenia elementów wykorzystamy **GridLayout**. W układzie umieścimy dziewięć przycisków. Początkowo kod pliku **activity_.main.xml** będzie wyglądał następująco:

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

    <Button
        android:text="1"
        android:layout_column="0"
        android:layout_row="0"
        />
    <Button
        android:layout_column="1"
        android:layout_row="0"
        android:text="2" />
    <Button
        android:text="3"
        android:layout_column="2"
        android:layout_row="0"
        />
    <Button
        android:layout_column="0"
        android:layout_row="1"
        android:text="4" />
    <Button
        android:text="5"
        android:layout_column="1"
        android:layout_row="1"
        />
    <Button
       android:layout_column="2"
       android:layout_row="1"
       android:text="6" />
    <Button
        android:text="7"
        android:layout_column="0"
        android:layout_row="2"
        />
    <Button
        android:text="8"
        android:layout_column="1"
        android:layout_row="2"
        />
    <Button
        android:text="9"
        android:layout_column="2"
        android:layout_row="2"
        />
</GridLayout>
```

Listing 6.113 Początkowy kod układu aplikacji

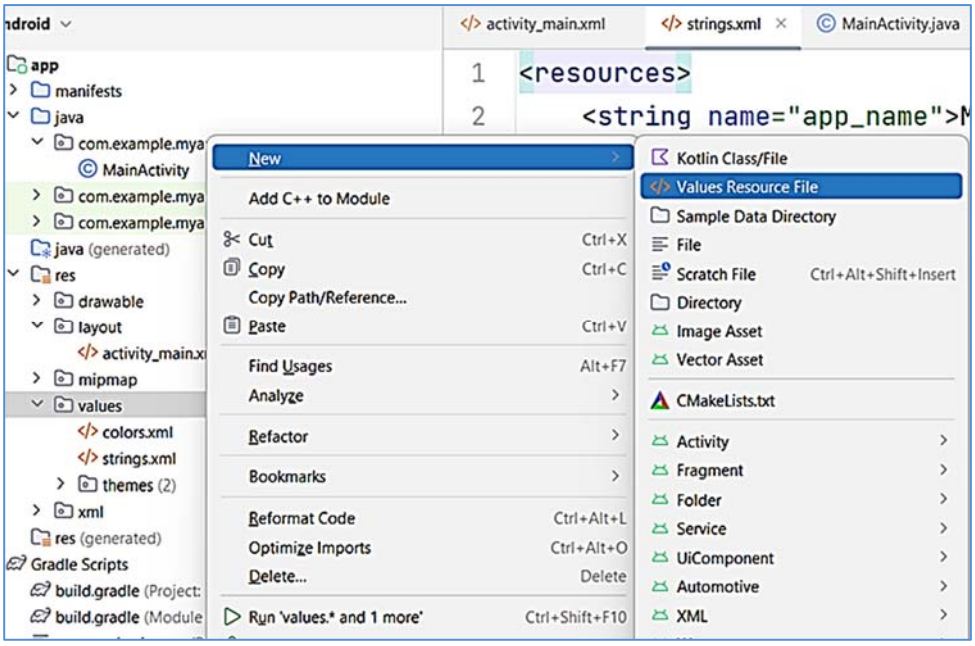
Różnić się będą tylko wartości w parametrach **android:layout_column** i **android:layout_row**.

Aplikacja po uruchomieniu będzie wyglądać tak:



Rysunek 6.78 Aplikacja ze stylem domyślnym

Style można tworzyć w pliku **styles.xml**. Jest to plik zasobów aplikacji i znaleźć go można w drzewie po lewej stronie. Jeżeli plik się tam nie znajduje trzeba go utworzyć samodzielnie. Klikamy prawym przyciskiem myszy na **gałąź values** i z menu, które nam się otwiera wybieramy **New → Values Resource File.**



Rysunek 6.79 Tworzenie nowego pliku zasobów – styles.xml

Nadajemy plikowi nazwę **styles**. Rozszerzenie **xml** zostanie nadane automatycznie. Utworzymy **styl** o nazwie **Szachownica.**

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

<style name="Szachownica">
<item name="android:layout_width">130dp</item>
<item name="android:layout_height">130dp</item>
<item name="android:background">#FFFFFF</item>
<item name="android:textColor">#FF0000</item>
<item name="android:typeface">serif</item>
<item name="android:textSize">55sp</item>
</style>

</resources>
```

Listing 6.114 Definicja stylu Szachownica

Po pierwsze nadajemy nazwę stylu: **name=”Szachownica”**. Kolejne atrybuty informują o **szerokości** i **długości** widoku. Natomiast o kolorze tła zadecyduje atrybut **android:background**. Jest on ustawiony na biały, gdyż chcemy, aby wszystkie przyciski były białe. **Kolor** możemy zmienić poprzez kliknięcie **małego kwadratu** znajdującego się po **lewej stronie okna**.

Rysunek 6.80 Zawartość pliku colors.xml

Kliknięcie spowoduje wyświetlenie **palety**.

Rysunek 6.81 Okno wyboru koloru

Kolor wybieramy poprzez kliknięcie w **odpowiedni obszar**. Możemy to zrobić również przesuwając **dostępne suwaki**. Pierwszy steruje **odcieniem**, drugi **przeźroczystością**. Skorzystać możemy również z przygotowanej listy, a także wpisać wartości **RGB ręcznie**. Dla koloru białego będzie to wartość **255, 255, 255**. Zapis w pliku **styles.xml** jest **zapisem szesnastkowym**.

Kolejne atrybuty to ustawienia koloru tekstu:

```
<item name="android:textColor">#FF0000</item>
```

Listing 6.115 Ustawienie koloru czcionki

Określa rodzinę czcionek:

```
<item name="android:typeface">serif</item>
```

Listing 6.116 Ustawienie rodziny czcionek

Określa wielkość czcionki:

```
<item name="android:textSize">55sp</item>
```

Listing 6.117 Ustawienie rozmiaru czcionki

W pliku układu możemy teraz przypisać style do odpowiednich kwadratów. Zrobimy to za pomocą poniższego kodu:

```
style="@style/NazwaStylu"
```

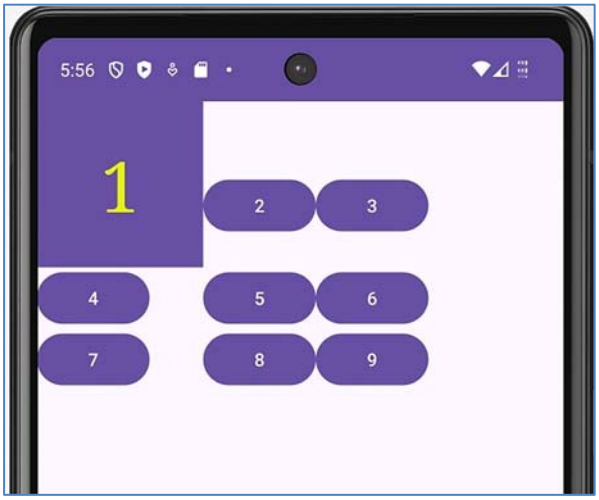
Listing 6.118 Przypisanie stylu do widoku.

w miejscu **NazwaStylu** wpisujemy nazwę **Szachownica**. Zmienimy ustawienie stylu dla pierwszego przycisku. Kod przycisku będzie wyglądał następująco:

```
<Button
    style="@style/Szachownica"
    android:text="1"
    android:layout_column="0"
    android:layout_row="0"
    />
```

Listing 6.119 Kod przycisku Button z zadeklarowanym stylem

Aplikacja po uruchomieniu powinna przypominać poniższy rysunek:



Rysunek 6.82 Przycisk 1 po zastosowaniu stylu

Na rysunku widać, że przycisk zmienił **rozmiar**, a także **kształt**. Zmienił się również **rozmiar i kolor czcionki**. Nie zmienił się jednak **kolor przycisku**. Aplikacja działa w domyślnym stylu, który jest zapisany w pliku **Manifest.xml**. Dlatego musimy zmienić to ustawienie. Na początek jednaj musimy ustalić swój motyw aplikacji. W pliku **styles.xml** umieszczamy następujący kod:

```
<style name="NowyTemat"
parent="Theme.AppCompat.Light.NoActionBar">
</style>
```

Listing 6.120 Stworzenie nowego stylu aplikacji

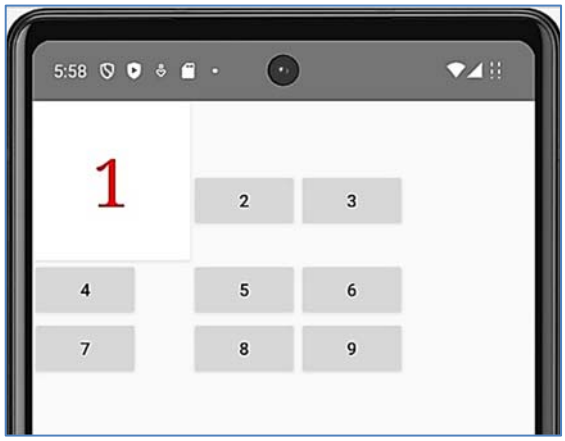
Tworzymy nowy **styl** o nazwie **NowyTemat**. W atrybucie **parent** wpisujemy nazwę stylu **nadrzędnego – rodzica**, który nieco zmieni wygląd aplikacji.

Ustawienia motywu będą szerzej omówione w następnym rozdziale. W tym momencie wystarczą nam tylko te dwie powyższe linijki. Następnie przechodzimy do pliku **manifestu** i zmieniamy tam jego nazwę.

```
<application
    android:allowBackup="true"

android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/NowyTemat"
    tools:targetApi="31">
```

Listing 6.121 Fragment pliku manifestu

W miejscu **android: theme** zmieniamy domyślną nazwę na nazwę naszego stylu czyli: **NowyTemat**. Po uruchomieniu aplikacji nasz przycisk zmieni kolor na właściwy.



Rysunek 6.83 Aplikacja po zastosowaniu własnego tematu aplikacji

Zauważyć można również, że zmianę pozostałych przycisków, a także, zmianę koloru paska. Wynika to z faktu, że zmieniliśmy cały temat kolorystyczny aplikacji.

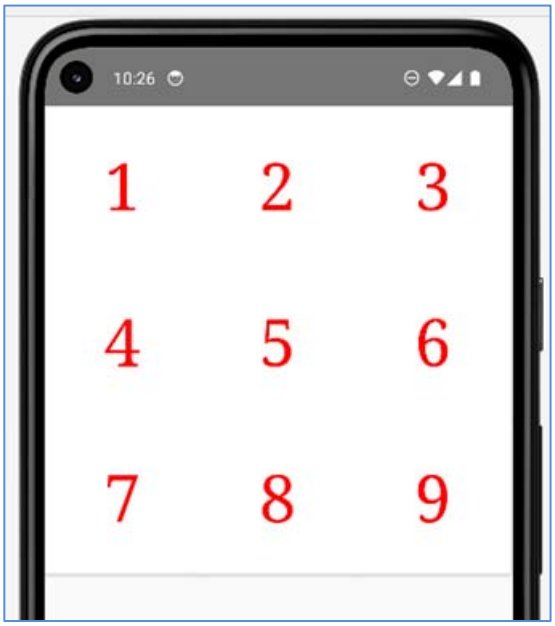
6.10.2 Dziedziczenie stylów

Kod poniższej aplikacji znajdziemy w folderze **Aplikacja24a**.

Podczas tworzenia większych i bardziej skomplikowanych aplikacji wykorzystanie pojedynczej grupy styli może być niewystarczające. Często

potrzebne jest wiele tego typu grup, które często różnią się tylko pewnymi elementami. Wówczas możemy wykorzystać część atrybutów z jednego stylu.

Jako przykład zmodyfikujemy poprzednią aplikację. Na początek przypiszmy **styl Szachownica** do wszystkich widoków w układzie. Aplikacja będzie wyglądać tak jak na poniższym rysunku.



Rysunek 6.84 Aplikacja z zastosowaniem własnego stylu

Wszystkie pola są **koloru białego**. Teraz musimy znaleźć sposób jak najprościej i najszybciej dokonać zmian w kodzie. Wykorzystamy w tym celu **dziedziczenie stylów**. Dziedziczenie jak wspomniano na początku umożliwia **zmianę** tylko **wybranych parametrów**. Resztę zostawimy bez zmian.

Tworzymy styl o nazwie **Szachownica.Czarne**. Nazwa zawiera informacje o tym, że głównym stylem jest styl o nazwie **Szachownica**, a **kropka** informacje, że styl **Czarne** po nim dziedziczy. W stylu **Szachownica.Czarne** wpisujemy informacje o kolorze tła jakie ma mieć widok. Kod pliku **styles.xml** będzie wyglądał tak jak poniżej:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

    <style name="NowyTemat"
parent="Theme.AppCompat.Light.NoActionBar">
    </style>

    <style name="Szachownica">
<item name="android:layout_width">130dp</item>
<item name="android:layout_height">130dp</item>
<item name="android:background">#FFFFFF</item>
<item name="android:textColor">#FF0000</item>
<item name="android:typeface">serif</item>
<item name="android:textSize">55sp</item>
    </style>

    <style name="Szachownica.Czarne">
<item name="android:background">#000000</item>
    </style>
</resources>
```

Listing 6.122 Definicja odziedziczonego stylu o nazwie Szachownica.Czarne

W pliku układu trzeba dopisać informację o **nowym stylu** dla **widoków**, które mają mieć **czarny kolor**. Kod pliku układu będzie miał po zmianach następującą postać:

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

    <Button
        style="@style/Szachownica"
        android:text="1"
        android:layout_column="0"
        android:layout_row="0"
        />
    <Button
        style="@style/Szachownica.Czarne"
        android:layout_column="1"
        android:layout_row="0"
        android:text="2" />
```

```
    <Button
        style="@style/Szachownica"
        android:text="3"
        android:layout_column="2"
        android:layout_row="0"
        />
    <Button
        style="@style/Szachownica.Czarne"
        android:layout_column="0"
        android:layout_row="1"
        android:text="4" />
    <Button
        style="@style/Szachownica"
        android:text="5"
        android:layout_column="1"
        android:layout_row="1"
        />
    <Button
        style="@style/Szachownica.Czarne"
        android:layout_column="2"
        android:layout_row="1"
        android:text="6" />
    <Button
        style="@style/Szachownica"
        android:text="7"
        android:layout_column="0"
        android:layout_row="2"
        />
    <Button
        style="@style/Szachownica.Czarne"
        android:text="8"
        android:layout_column="1"
        android:layout_row="2"
        />
    <Button
        style="@style/Szachownica"
        android:text="9"
        android:layout_column="2"
        android:layout_row="2"
        />
</GridLayout>
```

Listing 6.123 Kod układu z przypisanymi stylami

Aplikacja będzie wyglądać tak jak na poniższym rysunku:

Rysunek 6.85 Aplikacja wykorzystująca dziedziczenie stylu

6.11 Tematy, motywy.

Tematy lub motywy omawiane już były w poprzednim rozdziale. Zagadnienie to jest nieco szersze i przybliżymy je w tym rozdziale.

Temat lub **motyw** jest **stylem** stosowanym w **aplikacji**. Temat podobnie jak styl można zdefiniować i wykorzystać w całym programie. Style wykorzystuje się do części aplikacji np. **widoku**. Temat stosuje się do całej aplikacji. Są to ustawienia wyglądu aktywności od strony graficznej: **kolory, rodzaje czcionek**, a także **kolory pasków w oknie**.

6.11.1 Korzystanie z wbudowanych tematów

Tworzymy nową aplikację z plikiem układu zawierającym pole **tekstowe** i **przycisk**.

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    tools:context=".MainActivity">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="32dp"
        android:text="Hello World!"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.125"
        android:textSize="25pt"/>

    <Button
        android:layout_width="260dp"
        android:layout_height="97dp"
        android:layout_marginTop="150dp"
        android:text="Przycisk"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.498"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintVertical_bias="0.071" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Listing 6.124 Plik układu aplikacji

Następnie dodajemy plik **styles.xml**. Potrzebny również będzie plik **manifestu**. W pliku **styles.xml** tworzymy nowy **motyw**. Nadajmy mu nazwę **NowyTemat2** i tą właśnie nazwę wpiszemy w pliku manifestu. Plik **styles.xml** powinien wyglądać teraz tak:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
  <style name="NowyTemat2" parent="Theme.AppCompat">
    </style>

</resources>
```

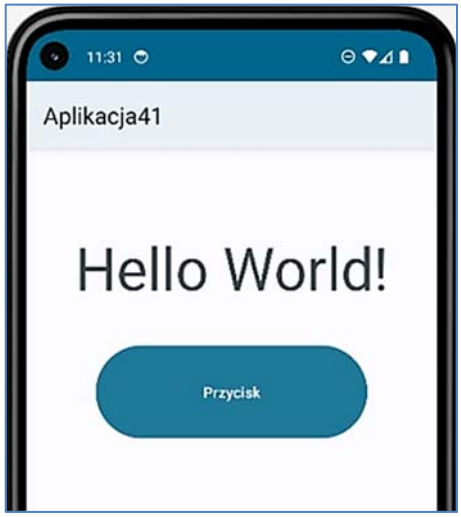
Listing 6.125 Utworzenie nowego stylu

W atrybucie **parent** możemy wybrać jeden z wielu dostępnych motywów aplikacji. Najlepszym wyborem będą **tematy** z kolekcji **Material3**. Są one bardzo nowoczesne i oparte są na technologii **Materia Design wersji 3**. Poniżej znajdują się przykłady aplikacji wykorzystujące różne wbudowane motywy.

Theme.AppCompat:

Rysunek 6.86 Aplikacja po zastosowaniu motywu "Theme.AppCompat"

Theme.Material3.DynamicColors.Light:



Rysunek 6.87 Aplikacja po zastosowaniu motywu Theme.Material3.DynamicColors.Light"

Theme.Material3.Dark:

Rysunek 6.88 Aplikacja po zastosowaniu motywu "Theme.Material3.Dark"

6.11.2 Tworzenie własnego motywu

Kod aplikacji znajduje się w folderze o nazwie **Aplikacja25**.

Tworzenie własnego **motywu** trzeba zacząć od ustawienia dowolnego z domyślnych **tematów**. Robimy to w pliku **styles.xml**. Pamiętaj również, żeby w pliku **manifestu** dodać **główną nazwę stylu**.

Wszystkie kolory, które będziemy używać w aplikacji należy wcześniej zadeklarować i ustawić w pliku **colors.xml**. Znajduje się on w drzewie aplikacji w gałęzi **Values**. W pliku można dopisać wartości do tych, które już tam się znajdują. Zajmiemy się teraz edycją tego pliku. Utworzymy wyraziste, kolory które silnie będą akcentować elementy aplikacji. Dopisujemy następujące kolory:

```
<color name="zielony">#7CB342</color>
<color name="czerwony">#D50000</color>
<color name="zolty">#FFD600</color>
<color name="jasnyniebieski">#55B7F1</color>
<color name="ciemnoniebieski">#283593</color>
<color name="różowy">#D1119C</color>
```

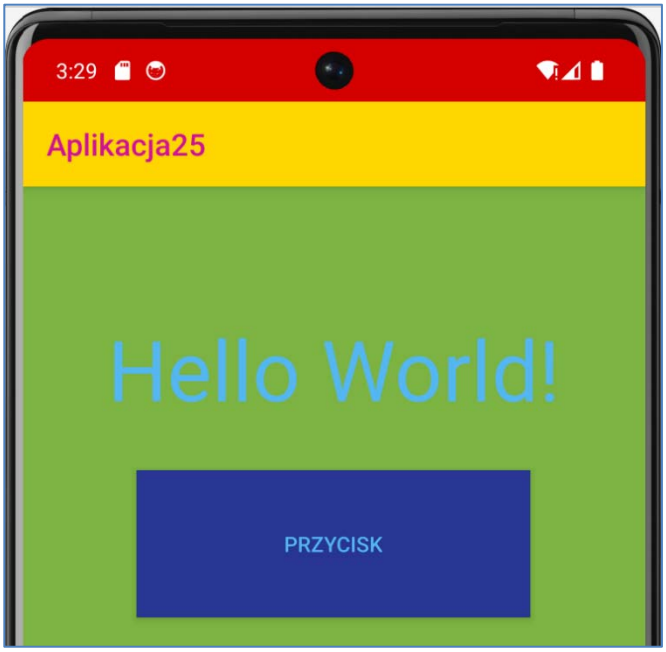
Listing 6.126 Plik colors.xml

Następnie w pliku **styles.xml** zmieniamy ustawienia kolorów elementów występujących w aplikacji. Kod w pliku **styles.xml** wygląda tak jak poniżej:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
<style name="NowyTemat2" parent="Theme.AppCompat">
<item
name="android:windowBackground">@color/zielony</item>
<item name="colorPrimary">@color/zolty</item>
<item name="colorPrimaryDark">@color/czerwony</item>
<item
name="android:textColor">@color/jasnyniebieski</item>
<item name="colorAccent">@color/ciemnoniebieski</item>
<item
name="android:textColorPrimary">@color/różowy</item>
    </style>

 <style name="Przycisk">
 <item
name="android:background">@color/ciemnoniebieski</item>
<item
name="android:textColor">@color/jasnyniebieski</item>
</style>
</resources>
```

Listing 6.127 Plik styles.xml z zdefiniowanym własnym stylem

Utworzyliśmy również dodatkowy **styl** dla **przycisku**. Na rysunku widzimy w którym miejscu aplikacji ustawiły się zadeklarowane kolory.



Rysunek 6.89 Aplikacja z własnym motywem

Jak widzimy **colorPrimary** to kolor znajdujący się na górze aplikacji tzw. **pasek ToolBar. ColorPrimaryDark** to kolor paska znajdującego się na samej górze aplikacji. **Android:windowBackground** to kolor tła w całej aplikacji, a **android:textColor** to kolor tekstu. Podobnie będą działać te ustawienia dla poszczególnych widoków. Jako przykład w tej aplikacji to **przycisk Button**.

6.12 Cykl życia aktywności

Każda aktywność, która zostanie uruchomiona poprzez system operacyjny rozpoczyna swój cykl życia. Aplikacja może przyjmować kilka różnych stanów. Są to:

- **Stan aktywny** – aktywność jest widoczna na pierwszym planie;
- **Stan wstrzymania** – aplikacja jest widoczna częściowo, ale nadal jest możliwość jej przywrócenie (nie została zamknięta);
- **Stan zatrzymania** – aplikacja jest nie widoczna, ale jeszcze nie zostały zwolnione wszystkie zasoby;
- **Stan zniszczenia** – aktywność kończy swoje działanie;

Przy starcie aplikacji swoje działanie rozpoczyna metoda **onCreate**, która zdefiniowana jest w pliku aktywności. Metoda **onCreate** „ożywia" aktywność i umożliwia jej działanie. Kiedy aktywność kończy działanie jest uruchamiana metoda **onDestroy**. Jest to ostatnia metoda w cyklu życia aktywności. Jej zadaniem jest wyczyszczenie danych, które zawierała aplikacja. Wraz z zamknięciem aplikacji wszystkie zasoby z jakich korzystała zostają zwolnione, a aktywność przechodzi w stan zniszczenia.

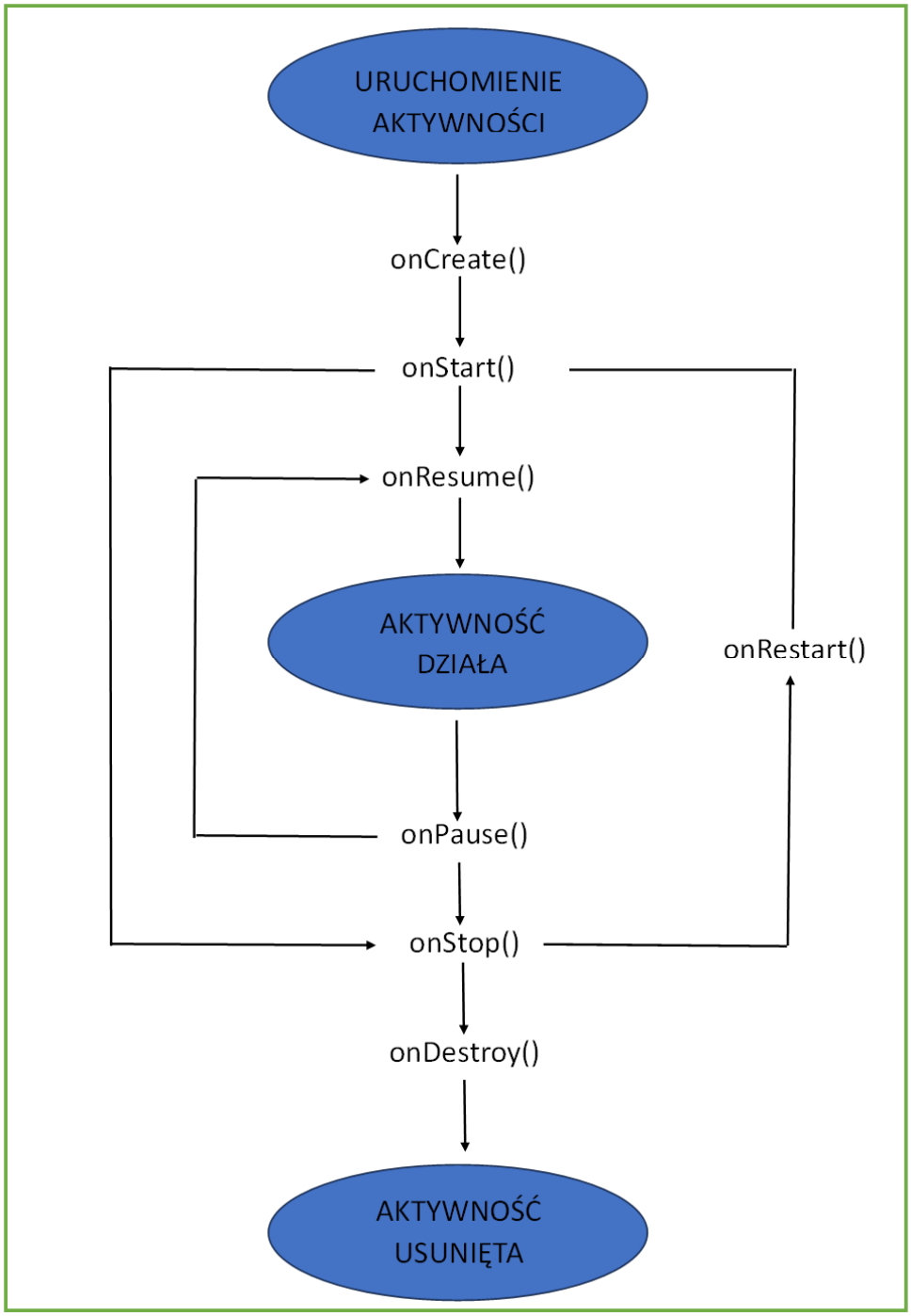
Jednak podczas użytkowania aplikacji zdarzają się czasami sytuacje, kiedy działanie aplikacji nie zostaje zatrzymane, ale czasem musi zostać **chwilowo przerwane**. Sytuacja taka może zdarzyć się wówczas, gdy podczas korzystania z aplikacji np. zadzwoni telefon i odbieramy połączenie. Aplikacja przyjmuje **stan uśpienia**. Dobrymi przykładami są gry, słuchanie muzyki czy oglądanie filmu na urządzeniu mobilnym. Pomocne będą tutaj kolejne metody, które zadbają o to, żeby podczas takich sytuacji działanie aplikacji nie zostało zakłócone. Zanim jednak omówimy te metody przeanalizujmy cykl życia aktywności przedstawiony na poniższym rysunku.

Uruchomienie aktywności wywołuje metodę **onCreate()**. Gdy aktywność (a przede wszystkim aplikacja) zaczyna działać, ale nie koniecznie jest widoczna

na pierwszym planie urządzenia, swoje działanie rozpoczyna metoda **onStart()**. Gdy aplikacja jest już widoczna na pierwszym planie zaczyna pracować metoda **onResume()**.

Gdy aplikacja zostaje zamykana przez użytkownika zaczynają działać metody, które będą czuwały nad prawidłowym przebiegiem tego procesu. Pierwsza z nich **onPause()** zaczyna działać w momencie, kiedy aplikacja znika z pierwszego planu czyli np. użytkownik odebrał połączenie telefoniczne lub otworzył inną aplikację. Jeżeli wznowiono działanie programu wywoływana jest metoda **onResume()**. Jednak jeśli użytkownik zdecyduje o wyłączeniu aplikacji następną uruchomioną metodą będzie **onStop()**. Przy zakończeniu działania aplikacji ostatnią metodą będzie metoda **onDestroy()**, która wyczyści wszystkie zasoby zarezerwowane przez aplikacje.

Jeżeli z jakiegoś powodu użytkownik postanowi jednak wznowić działanie aplikacji czyli przywrócić ją na pierwszy plan urządzenia, pomocna będzie metoda **onRestart()**, zaczyna ona działać tuż przed wznowieniem działania aplikacji, po niej uruchamiana jest ponownie metoda **onStart()**.



Rysunek 6.90 Schemat cyklu życia aktywności.

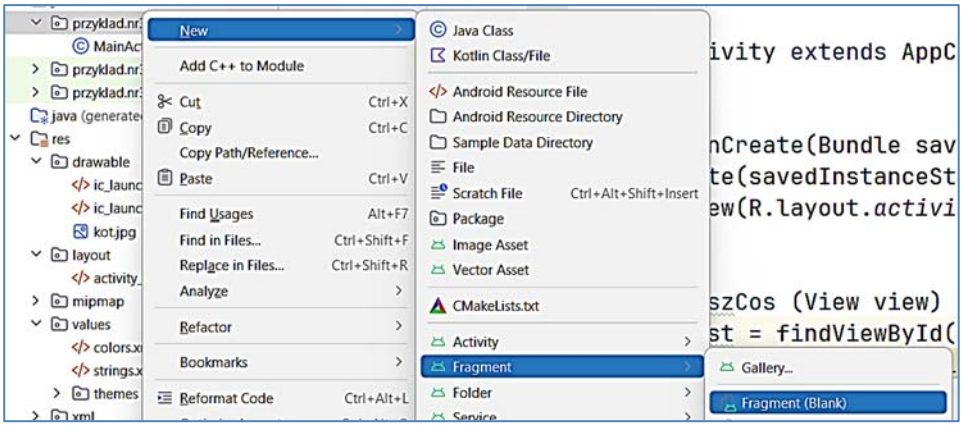
6.13 Fragmenty

Dla lepszej organizacji i funkcjonalności aplikacji układ można podzielić na mniejsze części. **Część układu** to po prostu **fragment**. Fragmenty podobnie jak układy definiujemy w pliku układu z rozszerzeniem ***.xml**.

By zaprezentować działanie fragmentów przygotujmy kolejną aplikację. Układ aplikacji będzie podzielony na **trzy części** wobec czego utworzymy sobie **trzy pliki układu** i **trzy nowe aktywności** o następujących nazwach: **fragment_1.xml, fragment_2.xml, fragment_3.xml, Fragment1.java, Fragment2.java, Fragment3.java**.

Zaczniemy od dodania do projektu **Fragmentów**. Otwieramy **drzewo aplikacji** znajdujące się po lewej stronie okna Android Studio. Rozwijamy je i klikamy prawym przyciskiem myszy na **katalog layout** umieszczony w **katalogu res**.

Tworzymy **nowy Fragment**: **New → Fragment → Fragment(Blank)**. W oknie tworzenia fragmentu wpisujemy nazwę **Fragment1**. Zatwierdzamy przyciskiem **Finish**. W plikach java (na drzewie aplikacji) pojawi się **nowa klasa Fragment1.java**, a w pliku **zasobów layout** mamy nowy plik układu o nazwie **fragment_1.xml**.



Rysunek 6.91 Tworzenie nowego Fragmentu

Operacje powtarzamy **trzykrotnie** tworząc **Fragmenty** o nazwach **Fragment1, Fragment2**, oraz **Fragment3**. Dodane pliki są widoczne w drzewie aplikacji. Kod aplikacji znajduje się w folderze **Aplikacja26**.

Jako pierwszy zdefiniujemy **Fragment1**. Zaczniemy od **pliku układu**. **Fragment** fizycznie jest częścią **aktywności** dlatego możemy go definiować tak jak aktywność, np. nadawać mu te same rodzaje **Layoutów** i określone dla

aktywności atrybuty. Pierwszy z fragmentów zdefiniujemy jako układ **LinearLayout** zawierający wyświetlany tekst: „**Jestem pierwszym fragmentem**”, oraz przycisk z napisem: „**Fragment1**”. Dla odróżnienia fragmentu od pozostałych nadajmy mu również tło w **kolorze zielonym**.

Drugi fragment zdefiniujmy jako układ **FrameLayout** i umieścimy w nim obrazek na którym będzie znajdować się biały napis: „**Jestem fragmentem drugim**”. Tło fragmentu ustawimy na **kolor niebieski**.

Trzeci fragment będzie zdefiniowany w układzie **RelativeLayout**. Będzie zawierał widok **TextView** z napisem: „**To ja Fragment nr 3**” i tło **koloru żółtego**. Tekst będzie znajdował się dokładnie w centrum układu: zarówno w **pionie** jak i w **poziomie**.

Kody wszystkich fragmentów będą wyglądać w następujący sposób:

Plik fragment_1.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/green"
    android:orientation="vertical"
    tools:context=".Fragment1">

<TextView
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Jestem pierwszym fragmentem"/>
<Button
   android:layout_width="wrap_content"
   android:layout_height="wrap_content"
   android:text="Fragment1" />
</LinearLayout>
```

Listing 6.128 Plik układu pierwszego fragmentu

Plik fragment_2.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/blue"
    tools:context=".Fragment2">

    <ImageView
        android:layout_width="150dp"
        android:layout_height="350dp"
        android:layout_marginLeft="25dp"
        android:src="@drawable/zdj12" />

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Jestem fragmentem drugim"
        />

</FrameLayout>
```

Listing 6.129 Plik układu drugiego fragmentu

Plik fragment_3.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:background="@color/yellow"
    tools:context=".Fragment3">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerInParent="true"
        android:text="To ja Fragment nr 3"
        />

</RelativeLayout>
```

Listing 6.130 Plik układu trzeciego fragmentu

Kolejnym zadaniem jest edycja plików aktywności. Tak, aby nasze klasy stały się klasami obsługującymi fragmenty. Kod pliku **Fragment1.java** będzie wyglądał jak poniżej:

```
package przyklad.nr26;

import android.os.Bundle;
import androidx.fragment.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Fragment1 extends Fragment {

        @Override
public View onCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
return inflater.inflate(R.layout.fragment_1, container,
false);
        }
    }
```

Listing 6.131 Plik aktywności dla pierwszego fragmentu

Na początek musimy zaimportować potrzebne w tym **fragmencie klasy**. Następnie tworzymy **klasę Fragment1**, która będzie **dziedziczyła** po wbudowanej **klasie Fragment**. **Metoda onCreateView** jest uruchamiana w momencie, kiedy fragment znajduje się na ekranie aplikacji. Zawiera ona trzy parametry:

- **Inflater** – umieszcza we fragmencie odpowiedni układ;
- **Container** – jest to komponent rodzicielski z którym łączymy wszystkie kontrolki występujące we fragmencie;
- **savedInstanceState** – odpowiada za odczytanie poprzedniego stanu fragmentu;

W pierwszym parametrze **metody inflate** umieszczamy informację o pliku układu, który ma obsługiwać nasz fragment. Stąd w tym kodzie umieszczony został **plik fragment_1**. Pozostałe pliki fragmentów (**Fragment2.java** i **Fragment3.java**) będą miały na początku podobny kod i będą wyglądać następująco:

```
package przyklad.nr26;

import android.os.Bundle;
import androidx.fragment.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Fragment2 extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_2,
container, false);
    }
}
```

Listing 6.132 Plik aktywności dla drugiego fragmentu

Fragment3.java:

```
package przyklad.nr26;

import android.os.Bundle;
import androidx.fragment.app.Fragment;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;

public class Fragment3 extends Fragment {

    @Override
    public View onCreateView(LayoutInflater inflater,
ViewGroup container, Bundle savedInstanceState) {
        return inflater.inflate(R.layout.fragment_3,
container, false);
    }
}
```

Listing 6.133 Plik aktywności dla trzeciego fragmentu

Teraz nasze **Fragmenty** musimy połączyć w całość. W tym celu będziemy edytować główny plik **układu activity_main.xml**. Chcemy, żeby nasze **Fragmenty** ułożyły się jak na rysunku poniżej. Najlepiej, więc będzie wykorzystać zagnieżdżony układ **LinearLayout**. Kod aktywności będzie wyglądał tak:

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

  <fragment
   android:id="@+id/fragment1"
   android: name="przyklad.nr26.Fragment1"
   android:layout_width="match_parent"
   android:layout_height="match_parent"  />
  <fragment
   android:id="@+id/fragment2"
   android:name="przyklad.nr26.Fragment2"
   android:layout_width="280dp"
   android:layout_height="wrap_content"  />
    </LinearLayout>

  <fragment
   android:id="@+id/fragment3"
   android:name="przyklad.nr26.Fragment3"
   android:layout_width="match_parent"
   android:layout_height="match_parent"  />

</LinearLayout>
```

Listing 6.134 Plik activity_main.xml z ułożonymi wewnątrz fragmentami

A aplikacja po uruchomianiu będzie przypominała tą z poniższego rysunku:

Rysunek 6.92 Aplikacja z zastosowaniem trzech fragmentów

6.14 Powiadomienia

Powiadomieniami w aplikacji mobilnej są krótkie wiadomości przekazywane przez system użytkownikowi. Powiadomieniem może być np. informacja o odebraniu **wiadomości e-mail**, albo **przypomnienie z kalendarza**. Na ekranie pojawia się wówczas okienko z informacją, a na pasku wyskakuje ikona przypominająca.

Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja27**.

Tworzenie prostego powiadomienia rozpoczniemy od przygotowania pliku układu. Umieścimy w nim **przycisk Button**, który będzie uruchamiał powiadomienie, oraz **dwa widoki TextView**. **Pierwszy** z nich wyświetli komunikat: **Czy powiadomienia są włączone?**. Drugi wyświetli odpowiedź: **tak** lub **nie**. Działanie takie spowodowane jest wprowadzeniem zmian w wyświetlaniu powiadomień. Zmiany pojawiły się w **API 34**. Mają na celu umożliwić użytkownikowi blokadę niechcianych powiadomień. Zanim aplikacja uruchomi powiadomienie jej użytkownik musi wyrazić zgodę. Jeżeli taka zgoda istnieje powiadomienie pojawia się automatycznie. Kod układu będzie wyglądał następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.appcompat.widget.LinearLayoutCompat
    xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    android:padding="16dp"
    tools:context=".MainActivity">

    <Button
      android:id="@+id/przycisk1"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:layout_gravity="center"
      android:text="Pokaż powiadomienie"
      android:onClick="uruchom"/>
   <LinearLayout
       android:layout_width="match_parent"
       android:layout_height="match_parent"
       android:orientation="horizontal"
       android:padding="8dp">
```

```
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="1"
        android:gravity="center"
        android:text="Powiadomienia włączone?"    />
      <TextView
         android:id="@+id/tekst1"
         android:layout_width="wrap_content"
         android:layout_height="wrap_content"
         android:layout_weight="1"
         android:gravity="center"
         tools:text="TAK" />

     </LinearLayout>

</androidx.appcompat.widget.LinearLayoutCompat>
```

Listing 6.135 Kod układu activity_main.xml

Przejdziemy teraz do pliku **aktywności**. Tworzenie **powiadomienia** rozpoczniemy od **deklaracji zmiennych**.

```
private static final String CHANNEL_ID = "channel_01";
private static ActivityResultLauncher<String>
requestPermissionsLauncher;
private NotificationManager notificationManager;
```

Listing 6.136 Deklaracja zmiennych

Pierwsza z nich **CHANNEL_ID** utworzy id kanału powiadomień. **Druga** to zmienna będzie przechowywać informacje o tym, czy użytkownik udzielił pozwolenia na wyświetlanie powiadomienia. **Trzecia** to zmienna tworząca menedżera powiadomień. Kolejne działanie, to uzupełnienie metody **onCreate**.

```
notificationManager =
getSystemService(NotificationManager.class);
Uruchamiamy menedżera powiadomień

requestPermissionsLauncher =
registerForActivityResult(new
ActivityResultContracts.RequestPermission(), isGranted -
> {
    refreshUI();
    if (isGranted) sendNotification();
});
```

Listing 6.137 Kod uzupełniający metodę onCreate

Tworzymy **nowy obiekt**, który, sprawdzi czy powiadomienia są włączone. **Instrukcja warunkowa** sprawdzi czy takie powiadomienie zostało udzielone,

jeżeli tak, zostanie uruchomiona **metoda sendNotification**, której zadaniem będzie uruchomienie powiadomienia. Definicję metody omówimy za chwilę.

```
createNotificationChannel();
refreshUI();
```

Listing 6.138 Wywołanie metody tworzącej kanał powiadomienia

Tworzymy kanał powiadomienia i za pomocą **metody refreshUI** wyświetlamy odpowiedni komunikat o tym czy użytkownik wydał zgodę na powiadomienie.

```
public void uruchom(View view){
    if (ActivityCompat.checkSelfPermission(this,
android.Manifest.permission.POST_NOTIFICATIONS) !=
PackageManager.PERMISSION_GRANTED) {

requestPermissionsLauncher.launch(android.Manifest.permi
ssion.POST_NOTIFICATIONS);
        return;
    }
    sendNotification();
}
```

Listing 6.139 Definicja metody uruchom

Metoda **uruchom** zadziała po wciśnięciu przycisku **Button**. Sprawdzi ona czy aktywne jest pozwolenie użytkownika. Jeżeli taka zgoda jest udzielona uruchomione zostanie powiadomienie. Kolejna metoda utworzy **kanał powiadomienia**.

```
private void createNotificationChannel() {
 CharSequence name = "notification_channel";
 String description =
"notification_channel_description";
 int importance =
NotificationManager.IMPORTANCE_DEFAULT;
NotificationChannel channel = new
NotificationChannel(CHANNEL_ID, name, importance);
    channel.setDescription(description);

notificationManager.createNotificationChannel(channel);
}
```

Listing 6.140 Definicja metody tworzącej kanał powiadomienia

CharSequence name – nadaje **nazwę kanału**, która będzie nam potrzebna do jego utworzenia. Zmienna **importance** ustala **priorytet** kanału, który decyduje o jego ważności wówczas gdy mamy ich w jednej aplikacji więcej.

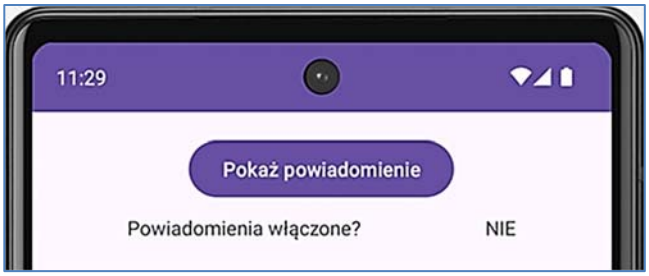
Obiekt klasy **NotificationChannel,** o przykładowej nazwie **channel** tworzy kanał powiadomienia. Wykorzystane są tutaj parametry utworzonej we wcześniejszej części kodu: **id kanału**, **name** i **description**. Ostatecznie

uruchamiamy kanał za pomocą menedżera powiadomień - **notificationManager**.

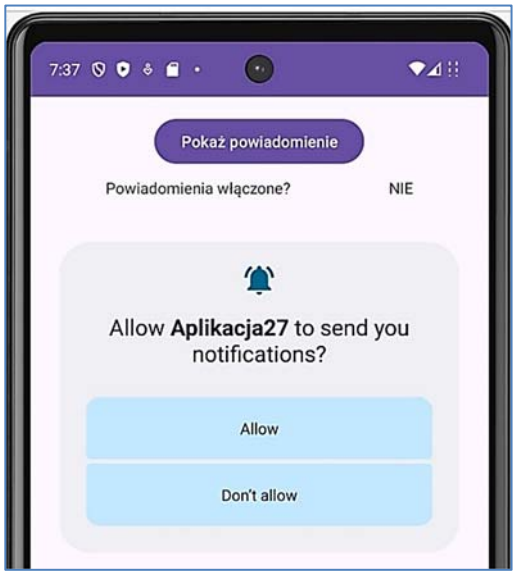
```
private void refreshUI() {
TextView textView = findViewById(R.id.tekst1);

textView.setText(notificationManager.areNotificationsEna
bled() ? "TAK" : "NIE");
}
```

Listing 6.141 Deklaracja metody refreshUI

Metoda **refreshUI** ma za zadanie wyświetlić odpowiedź **TAK** lub **NIE** w zależności od tego czy użytkownik wydał zgodę na powiadomienia. Odpowiedź zostanie wyświetlona w polu **TextView** do którego pobierzemy referencję. Jeżeli aplikacja nie posiada odpowiednich uprawnień zapyta użytkownika o pozwolenie.



Rysunek 6.93 Aplikacja po uruchomieniu.



Rysunek 6.94 Android pyta użytkownika czy wyraża zgodę na uruchomienie powiadomienia

```
private void sendNotification() {
NotificationCompat.Builder builder = new
NotificationCompat.Builder(this, CHANNEL_ID)

.setSmallIcon(R.drawable.ic_launcher_foreground)
.setContentTitle("Powiadomienie")
.setContentText("Udało Ci się utworzyć
powiadomienie.")
.setPriority(NotificationCompat.PRIORITY_DEFAULT)
;
notificationManager.notify(1, builder.build());
}
```

Listing 6.142 Tworzenie prostego powiadomienia

Ostatnia klasa **sendNotification** utworzy powiadomienie. Do utworzenia powiadomienia wykorzystamy klasę **NotificationCompat.Builder**. Potrzebne nam będą dwa parametry **pierwszy** z nich to tzw. context o wartości **this**. **Drugim** parametrem jest **CHANNEL_ID** czyli unikalne ID kanału utworzone wcześniej.

Podobnie jak każdy widok, również powiadomienia powinny zawierać pewne podstawowe ustawienia. Są nimi:

- Mała ikonka, która zostaje wyświetlona na górze aplikacji - **setSmallIcon**;
- Tytuł powiadomienie - **setContentTitle**;
- Treść powiadomienia - **setContentText**;

Wszystkie elementy umieszczone są w przykładowym kodzie przedstawionym na powyższym listingu.

Kolejny z zaprezentowanych ustawień to tzw. **priorytet powiadomienia**. On informuje o tym jak bardzo ważne jest powiadomienie na tle innych. Istnieje siedem stopni priorytetu. Każdy z nich ma ustawioną wartość liczbową informującą o wielkości priorytetu i są to:

- IMPORTANCE_MAX - 5;
- IMPORTANCE_HIGH - 4 ;
- IMPORTANCE_DEFAULT - 3 ;
- IMPORTANCE_LOW - 2 ;
- IMPORTANCE_MIN - 1 ;

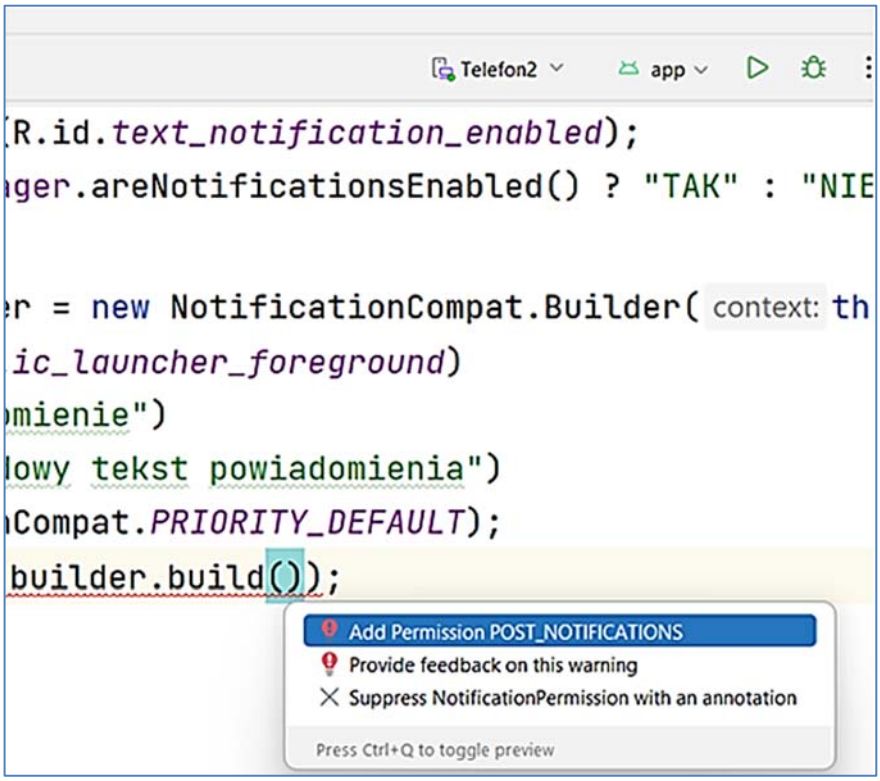
- IMPORTANCE_NONE – 0;
- IMPORTANCE_UNSPECIFIED - 1000.

Ustawień powiadomienia jest znacznie więcej. Omówione w tym przykładzie zostały tylko te obowiązkowe. Ostatnia linijka przykładowego kodu:

```
notificationManager.notify(NOTIFICATION_ID,builder.build
());
```

Listing 6.143 Uruchomienie powiadomienia w aktywności

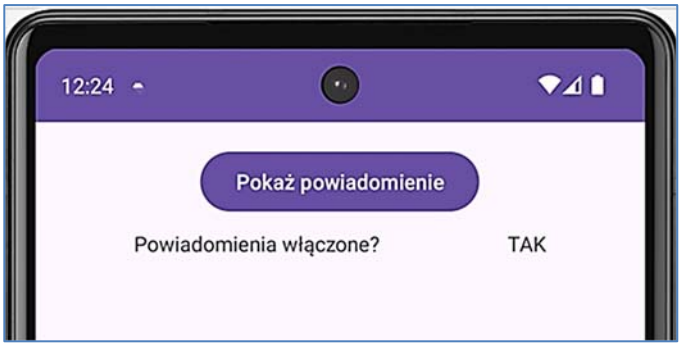
uruchamia utworzone powiadomienie. Parametr **NOTIFICATION_ID** jest unikalnym **ID** nadawanym powiadomieniu. **Drugi** z parametrów jest obiektem klasy **NotificationCompat** utworzonym na początku. Jak można zauważyć ostatnia linijka w kodzie podkreślona jest na czerwono i zgłasza błąd.



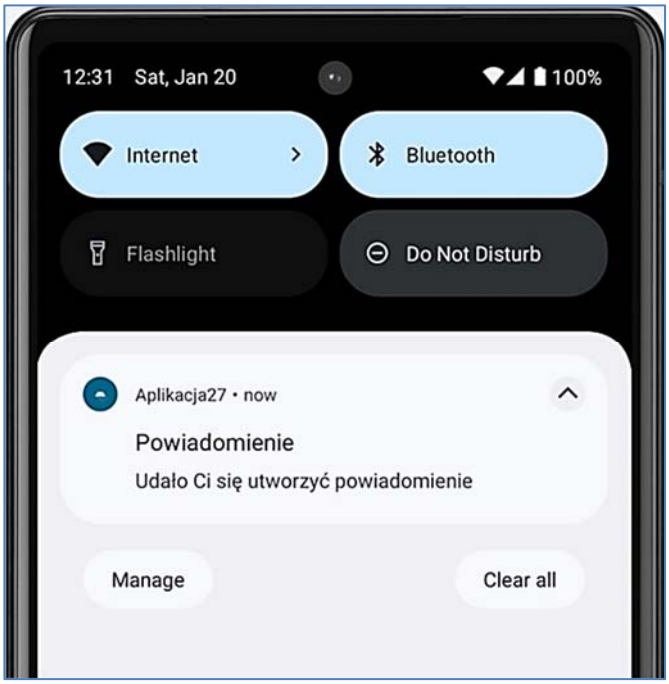
Rysunek 6.95 Błąd aplikacji i propozycja rozwiązania

Dzieje się tak dlatego, że w pliku **manifestu** należy umieścić zgodę na wysyłanie powiadomień. Jeżeli klikniemy na podkreślony fragment kodu i wybierzemy opcję: **Add Permission POST_NOTIFICATION** program

automatycznie doda odpowiedni wpis w pliku manifestu. Aplikacja po uruchomieniu wygląda następująco:



Rysunek 6.96 Aplikacja po wydaniu przez użytkownika zgody na uruchomienie powiadomienia



Rysunek 6.97 Uruchomione okno powiadomienia.

6.15 Okna dialogowe

Okna dialogowe służą do komunikacji aplikacji z użytkownikiem. Są to proste okienka pokazujące się w trakcie działania aplikacji.

6.15.1 Okno informacyjne

Kod aplikacji znajduje się w folderze **Aplikacja28.**

W kolejnym przykładzie tworzymy prosty **przycisk**. W pliku aktywności zaś umieszczamy poniższy kod:

```
package przyklad.nr28;

import androidx.appcompat.app.AppCompatActivity;
import android.app.AlertDialog;
import android.os.Bundle;
import android.view.View;
public class MainActivity extends AppCompatActivity {
  @Override
  protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void dialog1(View view) {
  AlertDialog.Builder dialogBuilder = new
AlertDialog.Builder(this);
  dialogBuilder.setTitle("Okno informacyjne");
  dialogBuilder.setMessage("Informuje o uruchomieniu
aplikacji");
  AlertDialog komunikat = dialogBuilder.create();
        komunikat.show();
    }
}
```

Listing 6.144 Plik aktywności z kodem tworzącym okno dialogowe

W metodzie **onClick** tworzymy obiekt klasy **AlertDialog**.

```
AlertDialog.Builder dialogBuilder = new
AlertDialog.Builder(this);
```

Listing 6.145 Tworzenie nowego obiektu klasy Dialog

Za pomocą metody **setTitle** nadajemy tytuł okienka, a za pomocą **setMessage** określamy wyświetlany komunikat.

```
dialogBuilder.setTitle("Okno informacyjne");
dialogBuilder.setMessage("Informuje o uruchomieniu
aplikacji");
```

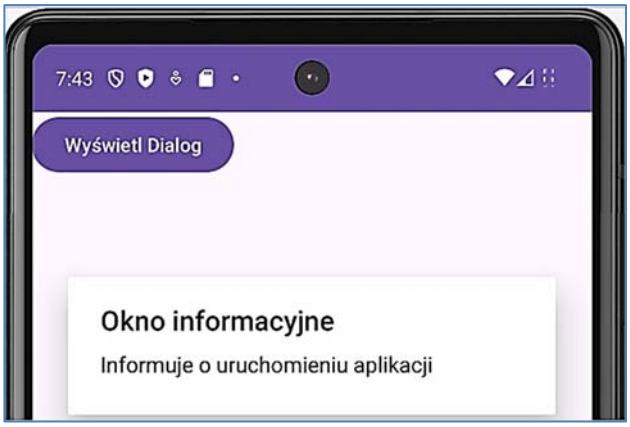
Listing 6.146 Ustawienie okna dialogowego

Tworzymy **obiekt** o nazwie **komunikat**, a następnie wyświetlamy go za pomocą **metody show**.

```
AlertDialog komunikat = dialogBuilder.create();
komunikat.show();
```

Listing 6.147 Uruchamianie okna dialogowego

Okno aplikacji będzie wyglądać jak poniżej:



Rysunek 6.98 Aplikacja z uruchomionym oknem informacyjnym

6.15.2 Okno informacyjne reagujące na działanie użytkownika

Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja28a**.

W poprzednim przykładzie uruchomiliśmy proste okno informacyjne, ale nie pozwoliło ono użytkownikowi na **reakcje**. Kolejny przykład pokaże w jaki sposób wykorzystać reakcje użytkownika. Aplikacja będzie składać się z prostego **układu** z **jednym przyciskiem** i **polem TextView**. **Przycisk** po **wciśnięciu** pokaże okno dialogowe.

Będzie się ono składało z **komunikatu** i **dwóch przycisków** dla użytkownika - **Tak**, oraz **Nie**. Po wybraniu dowolnego przycisku w oknie aplikacji pokaże się przypisany tekst, a okno dialogowe się zamknie. Kod aktywności będzie prezentował się na poniższym listingu:

```
package przyklad.nr28a;

import androidx.appcompat.app.AppCompatActivity;

import android.app.AlertDialog;
import android.content.DialogInterface;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
```

```
public class MainActivity extends AppCompatActivity {
    @Override
protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
     setContentView(R.layout.activity_main);
    }

    public void dialog1(View view) {

 AlertDialog.Builder dialogBuilder = new
AlertDialog.Builder(this);
 dialogBuilder.setTitle("Zadam Ci pytanie");
 dialogBuilder.setMessage("Czy masz dzisiaj dobry
nastrój?");
 dialogBuilder.setCancelable(false);
 dialogBuilder.setPositiveButton("Tak", new
DialogInterface.OnClickListener() {

public void onClick(DialogInterface dialog, int which) {
   TxtView tekst = findViewById(R.id.info);
   String napis = "Wybrałeś tak";
   tekst.setText(napis);
  }
});

dialogBuilder.setNegativeButton("Nie", new
DialogInterface.OnClickListener() {

public void onClick(DialogInterface dialog, int which) {
   TextView tekst = findViewById(R.id.info);
   String napis = "Wybrałeś nie";
   tekst.setText(napis);
 }
});
AlertDialog dialog = dialogBuilder.create();
   dialog.show();
 }
}
```

Listing 6.148 Kod aktywności z oknem dialogowym oczekującym na reakcje użytkownika

Co dzieje się w kodzie? Kod z poprzedniej aplikacji uzupełniliśmy o metody, które będą reagowały na wciśnięcie poszczególnych przycisków. Są to metody:

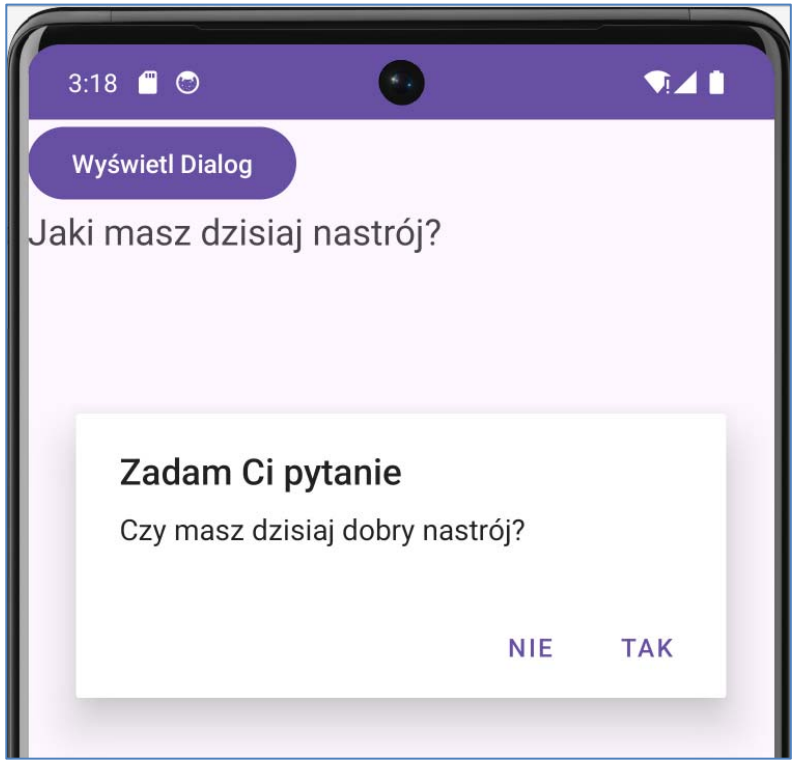
- **dialogBuilder.setPositiveButton** – będzie reagować, kiedy odpowiedź jest pozytywna;
- **dialogBuilder.setNegativeButton** – będzie reagować, kiedy odpowiedź jest negatywna.

```
dialogBuilder.setPositiveButton("Tak", new
DialogInterface.OnClickListener() {
public void onClick(DialogInterface dialog, int which) {
   TextView tekst = findViewById(R.id.info);
    String napis = "Wybrałeś tak";
     tekst.setText(napis);
           }
       });
```

Listing 6.149 Definicja klasy onClick działającej po reakcji użytkownika

Obie z metod mają **dwa parametry**. **Pierwszy** z nich jest **typu tekstowego** i zawiera tekst wyświetlony na przycisku. Odpowiednio są to: **Tak** i **Nie**. **Drugi** parametr to obiekt nasłuchujący **OnClickListener**.

Wewnątrz **każdej z metod** znajduje się kolejna **metoda onClick**, która zostanie uruchomiona dokładnie w momencie wciśnięcie odpowiedzi. W funkcjach tych pobierana jest **referencja** do **widoku TextView** znajdującego się w pliku układu, a następnie zostaje on zastąpiony odpowiednim komunikatem w zależności od tego, który z przycisków został kliknięty. Aplikacja będzie wyglądała podobnie do tej na poniższym rysunku:



Rysunek 6.99 Okno informacyjne wymagające reakcji od użytkownika

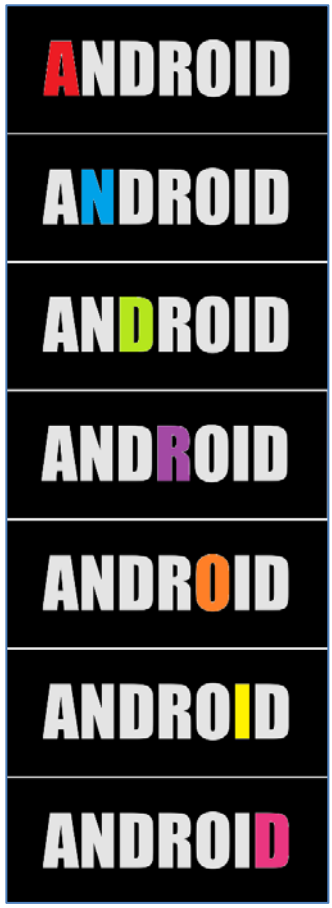
6.16 Animacje

6.16.1 Animacja poklatkowa

Kod aplikacji znajduje się w folderze o nazwie **Aplikacja29**.

Android Studio umożliwia stworzenie prostej animacji składającej się z kilku obrazków. Program zmienia obrazy jeden po drugim w określonym interwale czasowym co daje efekt animacji.

Tworzenie aplikacji rozpoczniemy od przygotowania obrazów. Będzie to **siedem plików *.png.** Każdy z nich będzie zawierał **napis ANDROID**. Tło rysunków będzie czarne, a pierwotny kolor liter to biały. W każdym z obrazów pojedyncza litera będzie miała inny kolor. Na rysunku przedstawiono grafiki potrzebne do utworzenia animacji.



Rysunek 6.100 Klatki animacji

Tak przygotowane obrazki umieszczamy w katalogu **drawable**. Kolejnym krokiem będzie przygotowanie **pliku układu**. Jedynym elementem, który tam umieścimy będzie **widżet ImageView**. Nie musisz umieszczać w kodzie odwołania do konkretnego pliku graficznego. Wszystko to będziemy programować w **pliku MainActivity.java**.

W tej aplikacji będzie nam potrzebny również plik obsługujący animacje. Utworzymy go w **katalogu drawable**. W tym celu najeżdżamy na jego nazwę w **drzewie aplikacji**. Klikamy **prawym przyciskiem myszy** i wybieramy **New** z wysuwanego menu i wybieramy **drawable Resource File**. Nowemu plikowi nadajemy nazwę **animacja**. W drzewie projektu pojawi się nowy plik z rozszerzeniem **xml**. Plik uzupełniamy o poniższy kod.

```
<animation-list
xmlns:android="http://schemas.android.com/apk/res/android"
    android:oneshot="false">
<item android:drawable="@drawable/z1" android:duration="250"/>
<item android:drawable="@drawable/z2" android:duration="250"/>
<item android:drawable="@drawable/z3" android:duration="250"/>
<item android:drawable="@drawable/z4" android:duration="250"/>
<item android:drawable="@drawable/z5" android:duration="250"/>
<item android:drawable="@drawable/z6" android:duration="250"/>
<item android:drawable="@drawable/z7" android:duration="250"/>
</animation-list>
```

Listing 6.150 Plik animacja.xml

Atrybut **android:oneshot** ustawiony został na wartość **false**. Oznacza to, że animacja będzie się powtarzać **bez końca**. Zmiana wartości na **true** spowoduje to, że animacja wyświetli się **tylko raz**.

W poszczególnych polach **item** umieszczamy dwie informacje: pierwsza to **odnośnik** do obrazka w katalogu **drawable**. Drugi **android:duration** określa czas trwania poszczególnej klatki. Wartość wyrażona jest w milisekundach. Co w przypadku naszej aplikacji oznacza, że każdy obrazek będzie wyświetlony przez **250ms czyli ¼ sekundy**. W aplikacji trzeba jeszcze uzupełnić kod pliku **MainActivity.java.**

```
package przyklad.nr29;

import androidx.appcompat.app.AppCompatActivity;

import android.graphics.drawable.AnimationDrawable;
import android.os.Bundle;
import android.view.View;
import android.webkit.WebView;
import android.widget.ImageView;

public class MainActivity extends AppCompatActivity {

    @Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 ImageView imageView = findViewById(R.id.obrazek);

imageView.setBackgroundResource(R.drawable.animacja);

AnimationDrawable animacja = (AnimationDrawable)
imageView.getBackground();
        animacja.start();
 }
}
```

Listing 6.151 Kod aktywności uruchamiający animację poklatkową

W klasie **onCreate** tworzymy kilka obiektów. Pierwszy z nich jest obiektem klasy **ImageView**. Tworzymy do niego **referencję**. Następnie ustawiamy go jako tło komponentu **ImageView** z pliku układu. Jako miejsce docelowe podajmy nazwę pliku animacja.

W następnym kroku utworzony jest obiekt klasy **AnimationDrawable** o nazwie **animacja**. Za pomocą **metody start** uruchamiamy animację. Dla lepszego efektu tło aplikacji można ustawić na czarne.

6.16.2 Animowanie widoków

Kod aplikacji znajduje się w folderze **Aplikacja30**.

Android umożliwia kilka operacji graficznych, które możemy wykonać na widokach. Animacji możemy dokonać w czterech obszarach w których wykorzystamy cztery znaczniki:

- **alpha** – odpowiada za zanikanie elementu;
- **scale** – służy do zmiany wielkości elementu;

- **translate** – odpowiada za przesunięcie elementu;
- **rotate** – służy do wyzwalania obrotu;

Animacja 1:

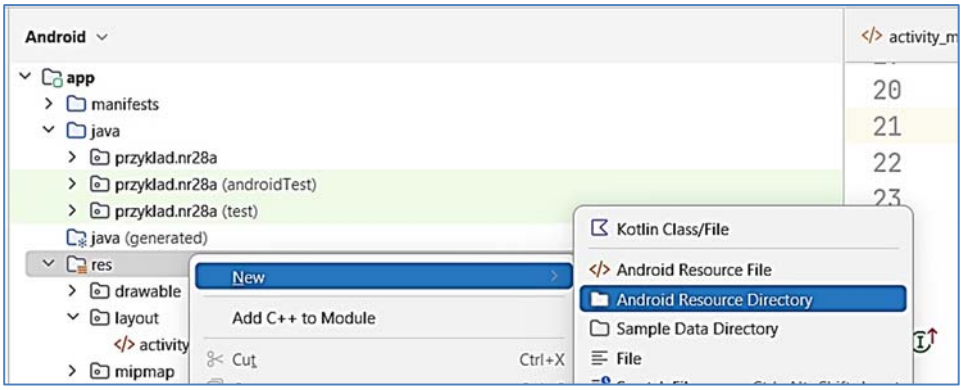
Pliku układu będzie zawierał **przycisk Button**, oraz **widok ImageView**. W folderze drawable umieszczamy obrazek.

Rysunek 6.101 Aplikacja po uruchomieniu

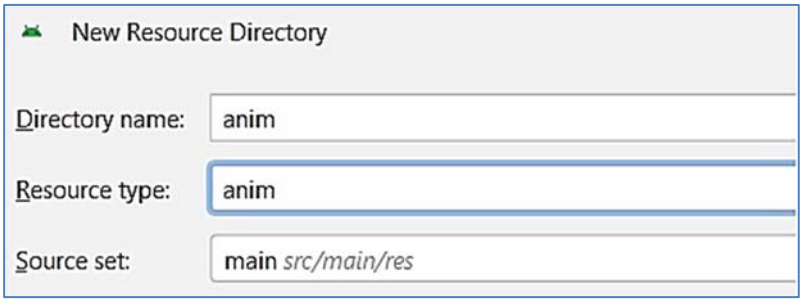
Wobec tego, że aplikacja będzie składała się z kilku animacji utworzymy specjalny katalog o nazwie **anim**. Katalog ten umieścimy w drzewie aplikacji. Na początek ustawmy się kursorem w miejscu gdzie znajduje się katalog **res**. Klikamy prawym przyciskiem myszy i wybieramy **New → Android Resource Directory**.

W okienku, które pojawi się po kliknięciu w przycisk trzeba wpisać nazwę katalogu. W pierwszym z okienek formularza wpisujemy nazwę **anim**, a w kolejnym wybieramy **typ** przechowywanych danych i tutaj również

wybieramy **anim**. Klikamy **OK** i w drzewie aplikacji będzie widoczny utworzony katalog.



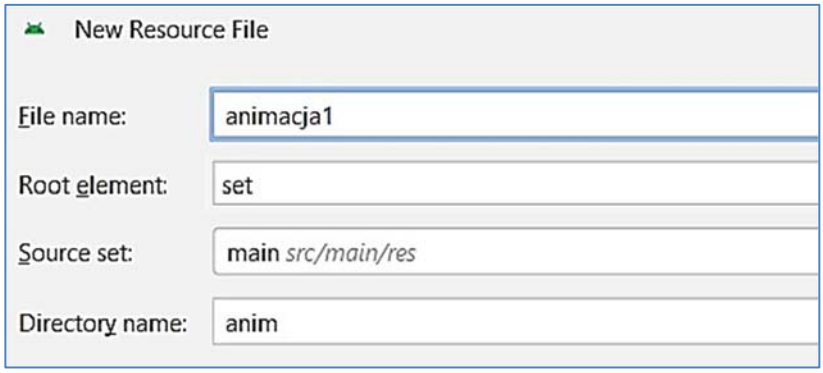
Rysunek 6.102 Tworzenie nowego folderu z zasobami aplikacji



Rysunek 6.103 Tworzenie folderu anim

W utworzonym katalogu **anim** tworzymy pierwszy z plików o nazwie **animacja1.xml**. Klikamy na katalog prawym przyciskiem myszy i wybieramy **New**, a następnie **Animation Resource File**. W okienku wpisujemy nazwę pliku czyli **animacja1**. W miejscu **Root element** zostawiamy **set**.

Po kliknięciu przycisku **OK** w drzewie aplikacji znajdziemy utworzony plik z rozszerzeniem **xml**.



Rysunek 6.104 Tworzenie pliku zasobów animacja1.xml

Plik należy teraz uzupełnić o następujący kod:

```
<set
xmlns:android="http://schemas.android.com/apk/res/android" >
    <alpha
        android:duration="5000"
        android:fromAlpha="1.0"
        android:toAlpha="0.0" />
</set>
```

Listing 6.152 Plik animacja1.xml ze zdefiniowaną akcją animacji

Znacznik **set** pozwala na ustawianie w aplikacji poniższych właściwości.

Pierwsza animacja będzie powodowała **zanikanie wyświetlonego obrazka**. Operacja ta będzie polegała na zmianie wartości **kanału alpha** w czasie. Dlatego w tej części kodu pojawił się znacznik **alpha**. Pierwszy atrybut **android:duration** określa czas całej animacji. Jest on wyrażany w **milisekundach**. W tym przykładzie zanikanie obrazka będzie trwało **5 sekund**.

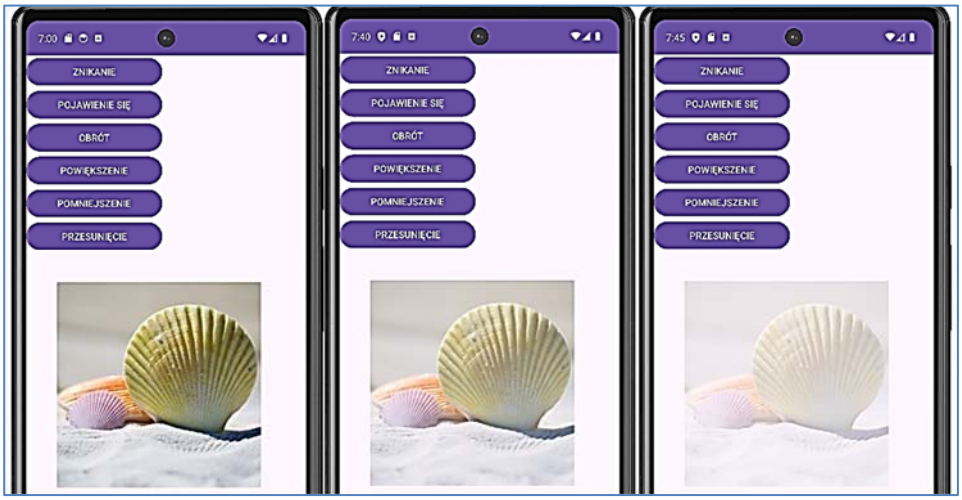
Kolejny atrybut **android:fromAlpha** wyraża początkową wartość kanału alpha, a drugi z nich **android:toAlpha** określa wartość przy jakiej zostanie zakończona animacja. Animacja da nam wrażenie, że obrazek będzie znikał.

Kolejną czynnością, którą należy teraz zrobić jest edycja kodu pliku **MainActivity.java**. Zajmiemy się zdefiniowaniem **klasy onClick**, która uruchomi się po wciśnięciu przycisku. Ze względu na to, że poza funkcją zanikanie, zaprogramujemy również inne akcje, w atrybucie **onClick** umieścimy taką samą nazwę klasy jaką ma plik w folderze **anim** i będzie to **anim1**. Kod **klasy anim1** będzie wyglądał tak jak na poniższym listingu:

```
public void anim1(View view){
        Animation animacja1=
AnimationUtils.loadAnimation(this, R.anim.animacja1);
        ImageView obrazek = findViewById(R.id.obrazek);
        obrazek.startAnimation(animacja1);
    }
```

Listing 6.153 Uruchomienia animacji w pliku aktywności

W aplikacji będziemy korzystać z **klasy Animation**. Tworzymy zatem **obiekt** tej klasy o nazwie **animacja1** i przypisujemy mu dane pobrane z piku zanikanie znajdującego się w katalogu **anim**. Następnie tworzymy **referencję** do obrazka znajdującego się w **pliku układu**. Uruchamiamy aplikacje za pomocą metody **startAnimation**. Uruchomienie aplikacji i wciśnięcie przycisku **ZNIKANIE** spowoduje powolne znikanie obrazka z aplikacji.



Rysunek 6.105 Animacja zanikania

Animacja 2:

Dodajmy do aplikacji kolejne animacje. Na początek będzie to animacja w której będzie pojawiał się obrazek. Procedura jest dokładnie taka sama jak przy pierwszym przykładzie. Dodajemy **plik animacja2.xml** do katalogu **anim**. Uzupełniamy go o kod działający odwrotnie do poprzedniego przykładu.

Następnie do **układu activity_main.xml** dodajemy kolejny przycisk z napisem **POJAWIANIE SIĘ**. Zdefiniować musimy jeszcze klasę odpowiedzialną za uruchomienie aplikacji. Kliknięcie przycisku ma spowodować pojawienie się obrazka w aplikacji. Zdefiniowana do przycisku **POJAWIENIE** będzie wyglądać tak jak na poniższym listingu:

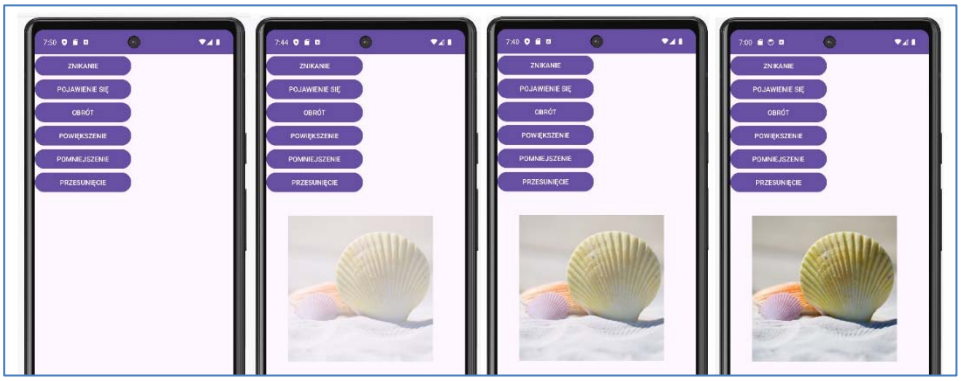
```
public void anim2(View view){
Animation  animacja1= AnimationUtils.loadAnimation(this,
R.anim.animacja2);
ImageView obrazek = findViewById(R.id.obrazek);
    obrazek.startAnimation(animacja1);
}
```

Listing 6.154 Kod metody anim2

Plik **animacja2.xml** będzie miał następujący kod:

```
<set
xmlns:android="http://schemas.android.com/apk/res/android" >
    <alpha
        android:duration="5000"
        android:fromAlpha="0.0"
        android:toAlpha="1.0" />
</set>
```

Listing 6.155 Plik animacja2.xml ze zdefiniowaną akcją animacji



Rysunek 6.106 Animacja pojawiania się

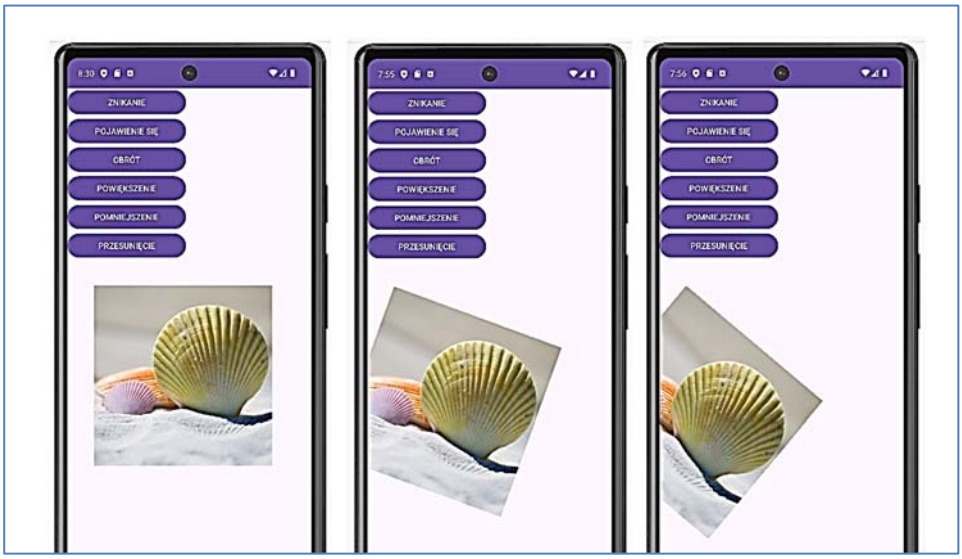
Animacja 3:

Zdefiniujemy teraz kolejne pliki w katalogu **anim**. Plik **animacja3.xml** przedstawia poniższy listing:

```
<set
xmlns:android="http://schemas.android.com/apk/res/android">
<rotate
    android:fromDegrees="0"
    android:toDegrees="360"
    android:duration="1000" >
</rotate>
</set>
```

Listing 6.156 Plik animacja3.xml ze zdefiniowaną akcją animacji

Kod spowoduje obrót obrazka o **360 stopni**. Zacznie się on od **punku 0** (co jest ustawione w atrybucie **android:fromDegrees**). Ilość stopni może być większa niż **360**. Każda **wielokrotność** spowoduje **więcej niż jeden obrót**. Gdy atrybut **android:toDegrees** będzie miał wartość **720** (czyli 2 razy 360) obrazek obróci się dwa razy. Obrót będzie odbywać się wokół lewego górnego rogu.



Rysunek 6.107 Animacja obrotu. Oś obrotu ustawiona na lewy górny róg

Do przesunięcia osi obrotu możemy użyć atrybutów **android:pivotX** i **android:pivotY**. Pierwszy atrybut przesunie oś obrotu w poziomie, drugi w pionie. Jednostką jaką należy tutaj użyć jest wartość procentowa. Użycie zapisu **android:pivotX=”50%”** spowoduje przesunięcie osi obrotu na środek górnej krawędzi.

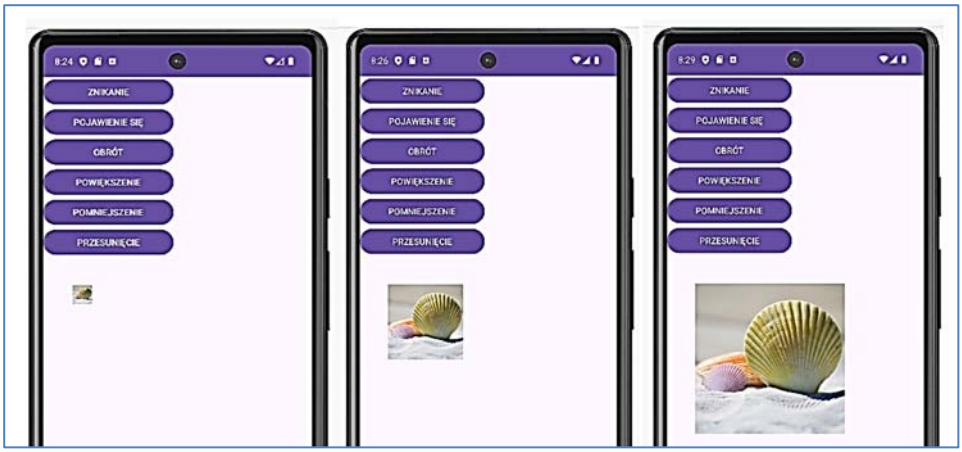
Rysunek 6.108 Animacja obrotu. Oś obrotu ustawiona na środku obrazka

Animacja 4 i Animacja 5:

Kod pliku **animacja4.xml** będzie wyglądać tak:

```
<set
xmlns:android="http://schemas.android.com/apk/res/android" >
    <scale
        android:duration="1000"
        android:fromXScale="0.5"
        android:fromYScale="0.5"
        android:toXScale="1"
        android:toYScale="1" >
    </scale>
</set>
```

Listing 6.157 Plik animacja4.xml ze zdefiniowaną akcją animacji

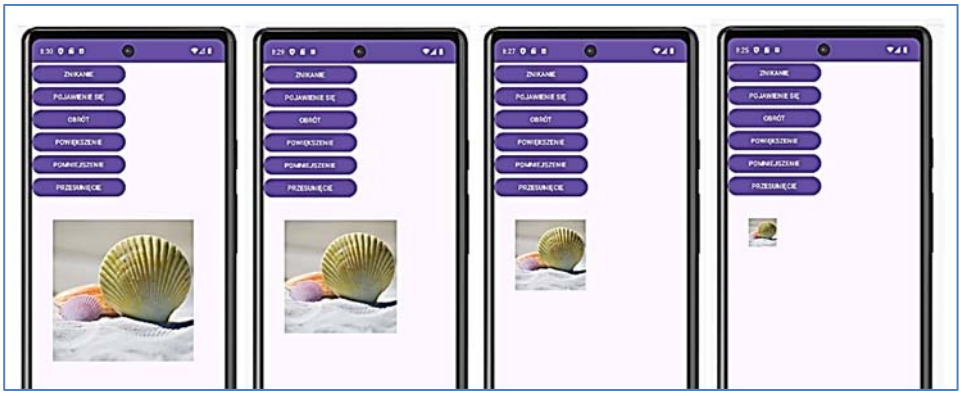


Rysunek 6.109 Animacja powiększenia.

Kod pliku animacja5.xml:

```
<set
xmlns:android="http://schemas.android.com/apk/res/android" >
    <scale
        android:duration="1000"
        android:fromXScale="1"
        android:fromYScale="1"
        android:toXScale="0.5"
        android:toYScale="0.5" >
    </scale>
</set>
```

Listing 6.158 Plik animacja5.xml ze zdefiniowaną akcją animacji



Rysunek 6.110 Animacja zmniejszenia.

Pojawiają się tutaj nowe atrybuty. Pierwsza z nich **android:fromXScale** - określa wielkość obrazka w poziomie przy starcie animacji. **Wartość 1.0** odpowiada **100%** wielkości, **wartość 0.5 - 50%**. Atrybut **android:fromYScale** działa tak samo jak poprzedni z tą różnicą, że określa wielkość obrazka w poziomie. Atrybuty **android:ToXScale** i **android:ToYScale** będą określać jak duży ma być obrazek przy zatrzymaniu animacji.

Operacje powiększania i pomniejszania rozpoczynają się w lewym górnym rogu obrazka. Gdy chcemy przesunąć ten punkt używamy atrybutów **pivot**.

Animacja 6

Kod pliku **animacja6.xml** będzie wyglądać tak:

```
<set
xmlns:android="http://schemas.android.com/apk/res/android" >
    <translate
        android:fromXDelta="0%"
        android:toXDelta="70%"
        android:fromYDelta="0%"
        android:toYDelta="70%"
        android:duration="1000" />
</set>
```

Listing 6.159 Plik animacja6.xml ze zdefiniowaną akcją animacji

Atrybuty **fromXDelta** i **fromYDelta** określają jaki procent obrazka ma być ukryte w momencie rozpoczęcia aplikacji. Pierwszy z atrybutów dotyczy osi **X** - osi poziomej, drugi osi **Y** - osi pionowej. Wartości te wyrażane są w procentach. Dwa kolejne atrybuty **toYDelta** i **toXDelta** określają w którym punkcie animacja się zatrzyma. W przypadku naszej aplikacji obraz zostanie przesunięty od **0%** zarówno pionowo i poziomo do **70%** obrazka . Punkt w którym rozpoczyna się przesuwanie to ponownie lewy górny róg.

Rysunek 6.111 Animacja przesunięcia.

Inne istotne atrybuty, które możemy wykorzystać przy animowaniu elementów:

- **fillAfter** — Atrybut może przyjmować dwie wartości: true i false. Wartość true spowoduje, że animacja po zatrzymaniu nie wróci do początkowego miejsca, a ponowne jej uruchomienie nastąpi w miejscu

zatrzymania. Wartość false przywróci animacje do pierwotnego położenia.

- **repeatCount** — określa ilość powtórzeń animacji. Jeżeli jako parametr podamy wartość -1 animacja będzie powtarzać się w nieskończoność.
- **repeatMode** — jeżeli atrybut repeatCount przyjmuje wartość inną niż zero (czyli animacja ma się powtórzyć) parametr repeatMode będzie decydował w jaki sposób ponownie będzie uruchomiona.

Parametr ten może przyjmować następujące wartości: Dostępne są dwie wartości:

  - **restart** — animacja rozpocznie nowy cykl od początku (wartość domyślna).
  - **reverse** — animacja zostanie odtworzona od tyłu.

- **startOffset** – opóźnienie. Parametr określa po jakim czasie od uruchomienia animacja zacznie działać. Jednostką jest czas podawany w milisekundach.

Kod pliku activity_main.xml:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <Button
        android:layout_width="200dp"
        android:layout_height="wrap_content"
        android:text="@string/anim1"
        android:id="@+id/anim1"
        android:onClick="anim1"
        />
    <Button
        android:layout width="200dp"
        android:layout_height="wrap_content"
        android:text="@string/anim2"
        android:id="@+id/anim2"
        android:onClick="anim2"
        />
     <Button
         android:layout_width="200dp"
         android:layout_height="wrap_content"
         android:text="@string/anim3"
         android:id="@+id/anim3"
         android:onClick="anim3"
         />
     <Button
        android:layout_width="200dp"
        android:layout_height="wrap_content"
        android:text="@string/anim4"
        android:id="@+id/anim4"
        android:onClick="anim4"
        />
    <Button
         android:layout_width="200dp"
         android:layout_height="wrap_content"
         android:text="@string/anim5"
         android:id="@+id/anim5"
         android:onClick="anim5"
         />
    <Button
        android:layout_width="200dp"
        android:layout_height="wrap_content"
        android:text="@string/anim6"
        android:id="@+id/anim6"
        android:onClick="anim6"
        />
    <ImageView
        android:id="@+id/obrazek"
        android:layout_width="300dp"
        android:layout_height="300dp"
        android:scaleType="centerCrop"
        android:src="@drawable/plaza"
        android:layout_marginLeft="45dp"
        android:layout_marginTop="45dp"
        />
</LinearLayout>
```

Listing 6.160 Kod układu aplikacji.

Kod aktywności głównej: **MainActivity.java.**

```
package przyklad.nr30;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.view.animation.Animation;
import android.view.animation.AnimationUtils;
import android.widget.ImageView;

public class MainActivity extends AppCompatActivity {

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

public void anim1(View view){
Animation animacja1=
AnimationUtils.loadAnimation(this,R.anim.animacja1);
ImageView obrazek = findViewById(R.id.obrazek);
 obrazek.startAnimation(animacja1);
 }

public void anim2(View view){
Animation animacja1 =
AnimationUtils.loadAnimation(this,R.anim.animacja2);
ImageView obrazek = findViewById(R.id.obrazek);
   obrazek.startAnimation(animacja1);
 }

public void anim3(View view){

Animation animacja1=
AnimationUtils.loadAnimation(this,R.anim.animacja3);
ImageView obrazek = findViewById(R.id.obrazek);
 obrazek.startAnimation(animacja1);
 }

public void anim4(View view){

Animation animacja1=
AnimationUtils.loadAnimation(this,R.anim.animacja4);
ImageView obrazek = findViewById(R.id.obrazek);
 obrazek.startAnimation(animacja1);
 }

public void anim5(View view){

Animation animacja1=
AnimationUtils.loadAnimation(this,R.anim.animacja5);

ImageView obrazek = findViewById(R.id.obrazek);
  obrazek.startAnimation(animacja1);
}

public void anim6(View view){
Animation animacja1=
AnimationUtils.loadAnimation(this,R.anim.animacja6);
ImageView obrazek = findViewById(R.id.obrazek);
   obrazek.startAnimation(animacja1);
 }
}
```

Listing 6.161 Kod aktywności

Potrzebny będzie również plik **strings.xml**.

```
<resources>
 <string name="app_name">Aplikacja28</string>
 <string name="anim1">ZNIKANIE</string>
 <string name="anim2">POJAWIENIE SIĘ</string>
 <string name="anim3">OBRÓT</string>
 <string name="anim4">POWIĘKSZENIE</string>
 <string name="anim5">POMNIEJSZENIE</string>
 <string name="anim6">PRZESUNIĘCIE</string>
</resources>
```

Listing 6.162 Plik strings.xml

6.17 Kontenery RecyclerView i CardView

RecyclerView jest bardziej zaawansowaną formą **widoku listy**. Umożliwia tworzenie układów z wykorzystaniem **CardView**. Ma on postać **przewijanych kart** na których można umieścić inne widoki w formie powtarzającej się grupy. Tworzenie tego typu widoków jest bardziej skomplikowane od zwykłej listy, gdyż wymaga od programisty utworzenia własnego **adaptera**. Jest jednak bardziej efektowna w wyglądzie, łatwiejsza i wygodniejsza w edycji.

Kod aplikacji znajdziesz w folderze **Aplikacja31**.

W projekcie będziemy potrzebować czterech obrazków umieszczonych w katalogu **drawable**. W przypadku tego przykładu będą to zdjęcia kolorowych sałatek. Tworzenie aplikacji rozpoczniemy od edycji pliku układu **activity_main.xml**. Umieścimy w nim układ **RelativeLayout** wewnątrz, którego wykorzystamy widżet **RecyclerView**. Kod wspomnianego układu będzie wyglądał następująco:

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

    <androidx.recyclerview.widget.RecyclerView
        android:id="@+id/widok1"
        android:scrollbars="vertical"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
</RelativeLayout>
```

Listing 6.163 Kod układu z wykorzystaniem kontenera Recyclerview

Kolejnym krokiem jest określenie wyglądu pojedynczej karty. Układ karty będziemy definiować w osobnym pliku układu. Tworzymy zatem nowy plik układu o nazwie **salatki.xml**. Plik będzie znajdował się w tym samym katalogu co plik aktywności głównej **activity_main.xml** czyli **layout**. Jako układ umieścimy tutaj kontener **CardView**. Zadeklarujemy go za pomocą poniższego kodu:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.cardview.widget.CardView
xmlns:android="http://schemas.android.com/apk/res/android"
xmlns:card_view="http://schemas.android.com/apk/res-auto"
    android:id="@+id/card_view"
    android:layout_width="match_parent"
    android:layout_height="250dp"
    android:layout_margin="4dp"
    card_view:cardElevation="2dp"
    card_view:cardCornerRadius="20dp">

    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
    <ImageView
        android:id="@+id/obrazek1"
        android:layout_height="200dp"
        android:layout_width="match_parent"/>

    <TextView
        android:id="@+id/nazwa"
        android:layout_marginLeft="4dp"
        android:layout_marginBottom="4dp"
         android:textSize="10pt"
         android:textColor="@color/black"
         android:layout_height="wrap_content"
         android:layout_width="match_parent"/>

    </LinearLayout>
</androidx.cardview.widget.CardView>
```

Listing 6.164 Kontener CardView

W powyższym fragmencie kodu pojawiają się nowe **atrybuty**, które bezpośrednio będą definiować wygląd karty. Pierwszy atrybut **card_view:cardElevation** – spowoduje wyświetlenie cienia pod kartą. Drugi z nich **card_view:cardCornerRadius** – powoduje zaokrąglenie rogów karty. Oba atrybuty mają znaczenie estetyczne.

Wewnątrz widżetu **CardView** umieścimy układ **LinearLayout**, a w nim obrazek (**ImageView**), oraz prosty podpis w postaci **TextView**.

Oznacza to, że każda karta będzie zawierała w sobie obrazek i podpis, które umieszczone zostaną jako układ liniowy. Oba widoki trzeba ze sobą połączyć. Trzeba też zdefiniować elementy poszczególnych kart (**obrazek i tekst**).

Wszystkie te zadania wykonamy w pliku aktywności **MainActivity.java**. Na początek tworzymy trzy obiekty. Obiekt **RecyclerView** klasy **RecyclerView**, menedżer widoku, który będzie odpowiadał za prawidłowy wygląd układu, oraz listę, która będzie przechowywała informacje o obrazkach i nazwach sałatek.

```
RecyclerView RecyclerView;
private RecyclerView.LayoutManager menedzer1;
ArrayList<Salatka> salatka;
```

Listing 6.165 Tworzenie listy przechowującej dane

W wbudowanej klasie **OnCreate**:

Obiektowi **RecyclerView** przypisujemy **referencję** do widoku umieszczonego w pliku układu (**activity_main.xml**) z id **widok1**, który jest widżetem **RecyclerView**.

```
RecyclerView = findViewById(R.id.widok1);
```

Listing 6.166 Referencja do elementu RecyclerView

Tworzymy obiekt **salatka**, który będzie przechowywał dane pobrane za pomocą metody **zrobSalatke()**; - metodę tą zdefiniujemy w dalszej części kodu.

```
salatka = zrobSalatke();
```

Listing 6.167 Tworzenie obiektu salatka

Obiektowi **menedzer1** przypisujemy nowy obiekt metody **GridLayoutManager**, który jest tzw. menedżerem układu siatki. Wykorzystujemy tutaj **konstruktor klasy**, który posiada dwa parametry. **Pierwszy** z nich tzw. **Context** będzie informował z jakiej klasy mamy pobrać dane. W związku z tym, że w tym miejscu korzystamy z klasy w której się znajdujemy, używając słowa kluczowego **this**. **Drugi** parametr jest liczbą całkowitą i określa na ile kolumn ma być podzielony układ.

```
menedzer1 = new GridLayoutManager(this,1);
```

Listing 6.168 Tworzenie obiektu klasy GridLayoutManager

Następnie za pomocą metody **setLayoutManager** zależącej od obiektu **RecyclerView** ustawiamy menedżer **layoutu** na nasz obiekt **menedżer1**.

```
RecyclerView.setLayoutManager(menedzer1);
```

Listing 6.169 Wykorzystanie metody setLayoutManager

Tworzymy **Adapter1** o nazwie **adapter** i tworzymy jego nową wersję o nazwie **salatka**, a następnie przypisujemy go do obiektu **RecyclerView** za pomocą metody **setAdapter**.

```
Adapter1 adapter = new Adapter1(salatka);
RecyclerView.setAdapter(adapter);
```

Listing 6.170 Utworzenie adaptera

Kolejnym elementem jest **klasa** o nazwie **Salatka**. Będzie ona definiować elementy, które będziemy wykorzystywać na każdej z kart widoku.

```
private static class Salatka {
    int obrazek;
    String nazwa;

    public Salatka (int obrazek, String nazwa){
        this.obrazek = obrazek;
        this.nazwa = nazwa;
    }
}
```

Listing 6.171 Definicja klasy Salatka

Potrzebować będziemy zmiennej typu **całkowitego**, która będzie przechowywać **id** obrazka, oraz zmiennej **łańcuchowej String**, która będzie zawierać **tekst**, wyświetlany w opisie (nazwie) sałatki. Do klasy musimy również stworzyć konstruktor. Kolejnym zadaniem jest stworzenie tablicy przechowującej dane każdej z **kart**.

```
private ArrayList<Salatka> zrobSalatke(){

ArrayList<Salatka> salatka = new ArrayList<>();
  salatka.add(new Salatka(R.drawable.jajko,"Sałatka z
jajkiem"));
  salatka.add(new Salatka(R.drawable.caprese,"Sałatka
Caprese"));
  salatka.add(new Salatka(R.drawable.krewetki,"Sałatka z
krewetkami"));
  salatka.add(new Salatka(R.drawable.owocowa, "Sałatka
owocowa"));
   return salatka;
}
```

Listing 6.172 Tablica przechowywująca dane z kart

Za pomocą metody **add** dodajemy do tablicy jeden obiekt składający się z **odnośnika** do obrazka, oraz **tekstu**, który będzie opisywał sałatkę. W tej aplikacji mamy czteroelementową tablicę. Tworzymy klasę **Adaptera**:

```
public static class Adapter1 extends
RecyclerView.Adapter<ViewHolder> {
```

Listing 6.173 Tworzenie klasy Adapter

W klasie tej tworzymy **tablicę**:

```
ArrayList<Salatka> s1;
```

Listing 6.174 Deklaracja tablicy s1.

Adapter ten potrzebuje jeszcze kilku **metod** do prawidłowego działania. Pierwsza z nich to **konstruktor**.

```
public Adapter1(ArrayList<Salatka> salatka1) {
    s1 = salatka1;
}
```

Listing 6.175 Definicja metody Adapter1

Konstruktor ten, będzie zawierał tylko jeden obiekt typu **Array** i będzie to nasza **tablica**, która przechowuje listę sałatek. Kolejną metodą jest **onCreateViewHolder.**

```
public ViewHolder onCreateViewHolder(ViewGroup parent,int
viewType) {
    View v =
LayoutInflater.from(parent.getContext()).inflate(R.layout
.salatki, parent, false);
    return new ViewHolder(v);
}
```

Listing 6.176 Definicja metody ViewHolder

Tutaj tworzony jest tzw. wzorzec **ViewHolder**. Za jego pomocą definiujemy w jaki sposób ma wyglądać pojedynczy element – pojedyncza karta. Stąd odwołanie do pliku układu **salatki.xml**. Metoda **onBindViewHolder** łączy dane z widokiem.

```
public void onBindViewHolder(ViewHolder holder, int
position) {
    Salatka salata = s1.get(position);

holder.obrazekSalatki.setImageResource(salata.obrazek);
holder.nazwaSalatki.setText(salata.nazwa);
}
```

Listing 6.177 Definicja metody onBindViewHolder

W metodzie pobierane jest miejsce elementu na liście. Ostatnią metodą będzie **getItemCount**.

```
public int getItemCount() {
    return s1.size();
}
```

Listing 6.178 Definicja metody getItemCount

Pobrana tutaj będzie ilość elementów, które będzie zawierać **lista**.

```
private static class ViewHolder extends
RecyclerView.ViewHolder {

public ImageView obrazekSalatki;
public TextView nazwaSalatki;
public ViewHolder(View v) {

super(v);
obrazekSalatki = v.findViewById(R.id.obrazek1);
nazwaSalatki =  v.findViewById(R.id.nazwa);
 }
}
```

Listing 6.179 Definicja klasy ViewHolder

Ostatnią klasą będzie klasa **ViewHolder** dziedzicząca po klasie **RecyclerView**. Do obiektów holder pobierane są referencję z pliku układu. W przypadku zdjęcia będzie to obrazek **id** widoku **ImageView** z pliku **salatki.xml**. Dla tekstu będzie to nazwa czyli **id TextView** z tego samego pliku układu. Na początek kod wydaje się dość skomplikowany. Cała zawartość aktywności **MainActivity.java** znajduje się poniżej.

```
package com.example.aplikacja31;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.ImageView;
import android.widget.TextView;
import androidx.appcompat.app.AppCompatActivity;
import androidx.recyclerview.widget.GridLayoutManager;
import androidx.recyclerview.widget.RecyclerView;

import java.util.ArrayList;

public class MainActivity extends  AppCompatActivity  {

RecyclerView RecyclerView;
private RecyclerView.LayoutManager menedzer1;
    ArrayList<Salatka> salatka;
    @Override
```

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

   RecyclerView = findViewById(R.id.widok1);
   salatka = zrobSalatke();
   menedzer1 = new GridLayoutManager(this,1);
        RecyclerView.setLayoutManager(menedzer1);
Adapter1 adapter = new Adapter1(salatka);
RecyclerView.setAdapter(adapter);
 }
private static class Salatka {
    int obrazek;
    String nazwa;
    public Salatka (int obrazek, String nazwa){
            this.obrazek = obrazek;
            this.nazwa = nazwa;
        }
    }
private ArrayList<Salatka> zrobSalatke(){
ArrayList<Salatka> salatka = new ArrayList<>();

salatka.add(new Salatka(R.drawable.jajko, "Sałatka z
jajkiem"));
salatka.add(new Salatka(R.drawable.caprese, "Sałatka
Caprese"));
salatka.add(new Salatka(R.drawable.krewetki, "Sałatka z
krewetkami"));
salatka.add(new Salatka(R.drawable.owocowa, "Sałatka
owocowa"));
        return salatka;
    }
public static class Adapter1 extends
RecyclerView.Adapter<ViewHolder> {
  ArrayList<Salatka> s1;
  public Adapter1(ArrayList<Salatka> salatka1) {
  s1 = salatka1;
}
  @Override
public ViewHolder onCreateViewHolder(ViewGroup parent,
int viewType) {
View v =
LayoutInflater.from(parent.getContext()).inflate(R.layou
t.salatki, parent, false);
       return new ViewHolder(v);
 }
        @Override
public void onBindViewHolder(ViewHolder holder, int
position) {
```

```
Salatka salata = s1.get(position);
holder.obrazekSalatki.setImageResource(salata.obrazek);
holder.nazwaSalatki.setText(salata.nazwa);
  }
   @Override
 public int getItemCount() {
      return s1.size();
  }
 }
private static class ViewHolder extends
RecyclerView.ViewHolder {
  public ImageView obrazekSalatki;
  public TextView nazwaSalatki;
  public ViewHolder(View v) {
  super(v);
obrazekSalatki = v.findViewById(R.id.obrazek1);
nazwaSalatki =  v.findViewById(R.id.nazwa);
  }
 }
}
```

Listing 6.180 Kod aktywności MainActivity.java

Aplikacja po uruchomieniu będzie wyglądać tak jak na poniższym rysunku:

Rysunek 6.112 Aplikacja z zastosowaniem kontenerów CardView

Na ekranie widać jednocześnie tylko trzy obrazki, wraz z opisami. Widok kart ma zawartą właściwość przewijania. Wystarczy, że przesuniemy ekran aplikacji, a pojawi się ostatnia pozycja na liście:

6.18 Multimedia

Nowoczesna aplikacja poza intuicyjną obsługą i ładną szatą graficzną powinna zawierać także elementy multimedialne. We wcześniejszych rozdziałach wstawialiśmy już obrazy, które poddawaliśmy animacją. Tworzyliśmy również animacje poklatkową. W aplikacji możemy również wykorzystywać dźwięk i filmy.

6.18.1 Odtwarzanie muzyki

Kod aplikacji i potrzebne do jej działania zasoby znajdują się w folderze **Aplikacja32**.

W kolejnym przykładzie stworzymy prostą aplikację do odtwarzania plików dźwiękowych. Będziemy potrzebować:

- Krótkiego pliku muzycznego w formacie **wav** lub **mp3**;
- Trzech niewielkich obrazków, które posłużą nam jako przyciski: **Start**, **Stop** i **Pauza**.

Tworzenie aplikacji rozpoczniemy od przygotowania trzech obrazków. W przykładzie zostaną wykorzystane poniższe obrazki o wymiarach: **wysokość** 100px, **szerokość** 100px każdy. Najlepiej jeżeli tło obrazków będzie przeźroczyste.

Rysunek 6.113 Ikony służące w aplikacji do odtwarzania pliku muzycznego

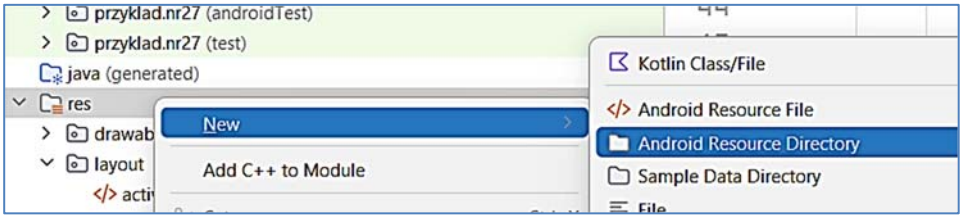
Tworzymy układ **LinearLayout** z orientacją poziomą – **horizontal**. Do układu wstawiamy trzy widżety **ImageButton**. Widoki te działają tak samo jak przyciski typu **Button** i posiadają takie same atrybuty. Również atrybut **onClick**, który będzie w tej aplikacji niezbędny.

Różnica pomiędzy nimi jest taka, że do widoku **ImageView** zamiast tekstu możemy wstawić obrazek. Dlatego też atrybut **android:text** jest zamieniony na **android:src** gdzie wstawiamy odnośnik do obrazka umieszczonego w katalogu drawable. Plik układu będzie wyglądał jak poniżej:

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="horizontal"
    tools:context=".MainActivity" >

    <ImageButton
        android:id="@+id/button1"
        android:layout_width="150dp"
        android:layout_height="100dp"
        android:layout_marginLeft="5dp"
        android:src="@drawable/play2"
        android:onClick="Play"
         />
   <ImageButton
       android:id="@+id/button2"
       android:layout_width="150dp"
       android:layout_height="100dp"
       android:layout_marginLeft="10dp"
       android:src="@drawable/stop2"
       android:onClick="Stop"
       />

   <ImageButton
       android:id="@+id/button3"
       android:layout_width="150dp"
       android:layout_height="100dp"
       android:layout_marginLeft="10dp"
       android:src="@drawable/pause2"
       android:onClick="Pauza"
        />
</LinearLayout>
```

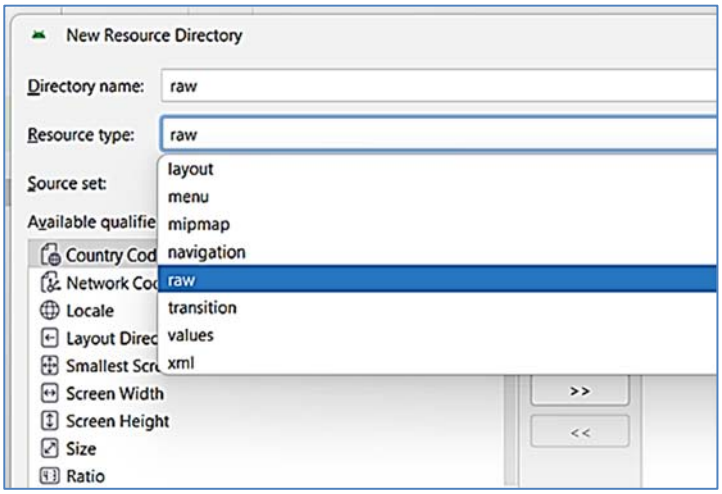
Listing 6.181 Kod układu aplikacji

Plik dźwiękowy trzeba umieścić w nowym katalogu, który trzeba będzie utworzyć w zasobach aplikacji. Katalog tworzymy klikając na drzewie projektów gałąź **res**. Wybieramy **New**, a następnie **Android Resource Directory**.



Rysunek 6.114 Tworzenie nowego katalogu zasobów

W polu **Resource type** wybieramy **raw**. Nazwa katalogu: **Directory type** zmieni się automatycznie. Wystarczy kliknąć **ok** i w drzewie katalogów pojawi się nowy folder.



Rysunek 6.115 Tworzenie katalogu zasobów raw

Odszukujemy na dysku komputera katalog **raw** i w nim umieszczamy plik dźwiękowy, który będziemy wykorzystywać w aplikacji. Można również użyć opcji **Open in Explorer**. Teraz czas na edycję pliku aktywności **MainActivity.java**.

Do obsługi plików dźwiękowych będzie nam potrzebna biblioteka **MediaPlayer**. Na początek tworzymy obiekt klasy **MediaPlayer**, który definiujemy w głównej klasie aktywności.

```
MediaPlayer muzyka;
```

Listing 6.182 Utworzenie obiektu klasy MediaPlayer

Następnie utworzony obiekt powiążemy z odpowiednim plikiem dźwiękowym umieszczonym w katalogu **raw**. Dokonujemy tego w metodzie **onCreate**.

```
muzyka = MediaPlayer.create(this, R.raw.music2);
```

Listing 6.183 Definicja obiektu muzyka

Metoda **create** obiektu **MediaPlayer** ma dwa argumenty. Pierwszy z nich to tzw. **context** dlatego mamy w tym miejscu słówko kluczowe **this** (zapis ten był wielokrotnie wcześniej omawiany). Drugim argumentem jest **plik dźwiękowy**. Jest on umieszczony w katalogu **raw** i w przypadku tego przykładu ma nazwę **music2**. Do obsługi przycisków trzeba utworzyć trzy metody. Będą to: **Play**, **Stop**, oraz **Pause**.

Metoda **Play** będzie uruchamiać dźwięk, **Stop** zatrzymywać bez możliwości ponownego uruchomienia, a metoda **Pause** będzie zatrzymywać odtwarzanie dźwięku, ale umożliwi ponowne uruchomienie w miejscu zatrzymania po ponownym kliknięciu przycisku **Play**. Wymienione metody są bardzo proste w obsłudze, a ich kod będzie wyglądał następująco:

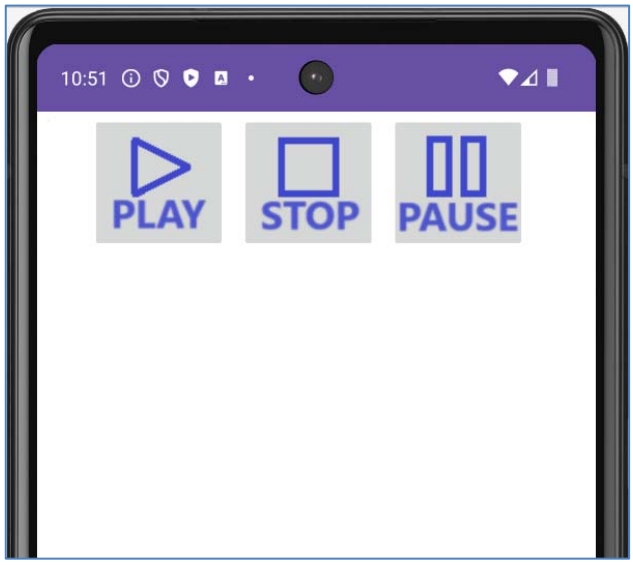
```
public void Play(View view)
{
    muzyka.start();
}
public void Stop(View view)
{
    muzyka.stop();
}
public void Pauza(View view)
{
    muzyka.pause();
}
```

Listing 6.184 Definicja metod obsługujących odtwarzanie dźwięku

Cały kod aktywności ma postać:

```
package przyklad.nr32;
import androidx.appcompat.app.AppCompatActivity;

import android.media.MediaPlayer;
import android.os.Bundle;
import android.view.View;

public class MainActivity extends AppCompatActivity {

    MediaPlayer muzyka;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        muzyka = MediaPlayer.create(this, R.raw.music2);
    }
    public void Play(View view)
    {
        muzyka.start();
    }
    public void Stop(View view)
    {
        muzyka.stop();
    }
    public void Pauza(View view)
    {
        muzyka.pause();
    }
}
```

Listing 6.185 Kod aktywności aplikacji



Rysunek 6.116 Aplikacja po uruchomieniu

6.18.2 Odtwarzanie filmów

Kod aplikacji znajduje się w folderze **Aplikacja33**.

Android Studio umożliwia odtwarzanie plików filmowych. Przydatna będzie w tym przypadku klasa **VideoView**. Napiszemy teraz prostą aplikację, która będzie odtwarzać film. W aplikacji **Video** będziemy wykorzystywać krótki film, który umieścimy w zasobach aplikacji.

Podobnie jak w poprzednim zadaniu, również tutaj musimy utworzyć katalog do przechowywania multimediów. Ponownie będzie to katalog **raw** znajdujący się w zasobie **res**. W utworzonym katalogu umieszczamy plik wideo. W pliku układu dodajemy widok umożliwiający odtwarzanie filmu i będzie to **VideoView**. Plik układu będzie wyglądał jak w poniższym kodzie.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/androi
d"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

    <VideoView
        android:id="@+id/film"
        android:layout_width="450dp"
        android:layout_height="250dp"
       />

</LinearLayout>
```

Listing 6.186 Kod układu aplikacji z elementem VideoView

W pliku aktywności dodajemy obiekt klasy **VideoView** i przypisujemy do niego odpowiedni element z pliku układu. Następnie obiekt łączymy z plikiem filmowym znajdującym się w katalogu **raw**. Teraz wystarczy tylko uruchomić film metodą **start**. Całość kodu umieszczamy w **klasie onCreate**. Kod pliku **MainActivity.java** przedstawiony jest poniżej:

```
package przyklad.nr33;

import androidx.appcompat.app.AppCompatActivity;
import android.net.Uri;
import android.os.Bundle;
import android.widget.VideoView;
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        VideoView video = findViewById(R.id.film);

video.setVideoURI(Uri.parse("android.resource://" +
getPackageName() + "/" + R.raw.film1));
        video.start();
    }
}
```

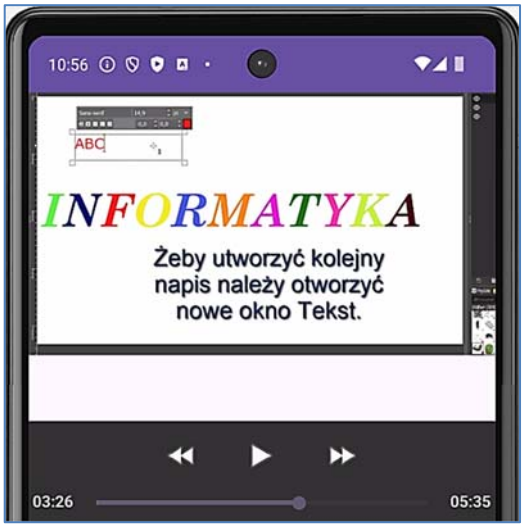
Listing 6.187 Kod aktywności aplikacji

W aplikacji można dodać panel sterujący odtwarzaniem. Do pliku aktywności wystarczy dołączyć dodatkowo poniższe linijki:

```
MediaController mediaController = new
MediaController(this);
video.setMediaController(mediaController);
```

Listing 6.188 Umieszczenie w aplikacji paska nawigacji video

Aplikacja z wykorzystaniem kontrolera będzie wyglądać jak na poniższym rysunku:



Rysunek 6.117 Aplikacja z uruchomionym filmem.

6.19 Nawigacja i tworzenie menu

Aplikacja mobilna składa się z kilku aktywności. Do sprawnego poruszania się po niej potrzebne będzie **menu** i **intuicyjna nawigacja**. Dobrym rozwiązaniem w takim przypadku jest zastosowanie paska narzędzi **Toolbar**.

6.19.1 Pasek Toolbar

Zacznijmy od początku. Pierwszym narzędziem wykorzystywanym do nawigacji był tzw. **AppBar** czyli pasek aplikacji. Jest to nic innego jak kolorowy obszar znajdujący się na górze okna. Jego podstawowym elementem jest **tytuł**. Można było do niego również dodać prosty przycisk menu.

Następcą AppBar jest pasek akcji czyli **ActionBar**. Pojawił się po raz pierwszy w **API 11**. **ActionBar** można było dostosowywać do potrzeb aplikacji umieszczając na nim m.in.: ikonę aplikacji, przycisk akcji, oraz rozszerzone menu bądź tzw. szufladę nawigacyjną. **Toolbar** czyli pasek narzędziowy jest następcą paska **Actionbar**. Jest jego nowocześniejszą i bardziej funkcjonalną wersją. Pojawił się po raz pierwszy w **API 21**. Oprócz elementów które mogliśmy wykorzystywać w **ActionBar** umożliwia również umieszczanie w nim innych widoków.

Utworzymy teraz aplikacje, która zaprezentuje działanie paska nawigacyjnego. Kod aktywności znajduje się w katalogu **Aplikacja34**.

W układzie umieszczamy widok reprezentujący pasek **ToolBar**.

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

    <androidx.appcompat.widget.Toolbar
        android:id="@+id/toolbar"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:background="@color/jasnoniebieski" />
</LinearLayout>
```

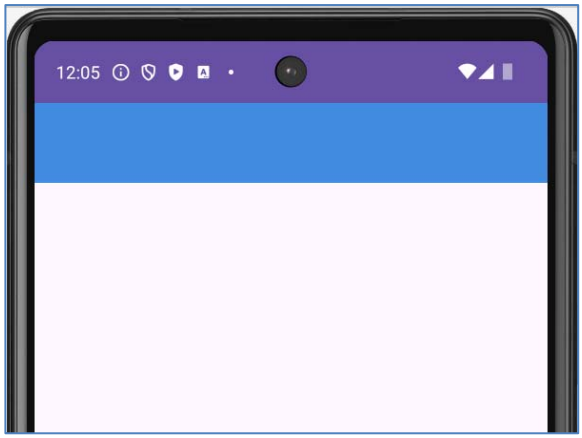
Listing 6.189 Kod układu wstawiający pasek ToolBar

Jak widać pasek **ToolBar** dodajemy do aplikacji tak jak zwykły **widok**. W pliku **colors.xml** utworzyliśmy nowy kolor, który wykorzystamy w aplikacji:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
 <color name="black">#FF000000</color>
 <color name="white">#FFFFFFFF</color>
 <color name="jasnoniebieski">#418CE1</color>
</resources>
```

Listing 6.190 Plik colors.xml z definicją nowego koloru

Po uruchomieniu aplikacja będzie wyglądać w następujący sposób:



Rysunek 6.118 Aplikacja z paskiem ToolBar

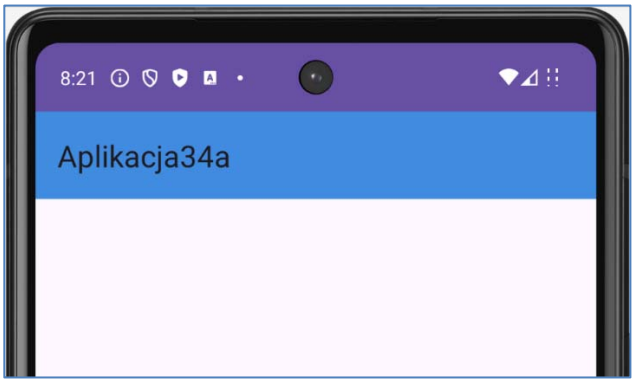
W oknie widzimy **niebieski pasek** – to jest właśnie pasek **ToolBar**. Kolejnym krokiem jest dodanie do niego nazwy aplikacji. Dokonać tego można w pliku aktywności **MainActivity.java**.

```
public class MainActivity extends AppCompatActivity {

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

Listing 6.191 Kod aktywności wstawiający pasek Toolbar

W klasie **onCreate** tworzymy obiekt **toolbar** klasy **Toolbar** i przypisujemy mu odpowiedni **widok** znajdujący się w pliku **układu**. Metoda **setSupportActionBar** zadba o prawidłowe działanie paska **Toolbar**.

Po uruchomieniu aplikacji na pasku **ToolBar** pojawi się jej tytuł. Kod aplikacji z tytułem można znaleźć w katalogu o nazwie **Aplikacja34a**.



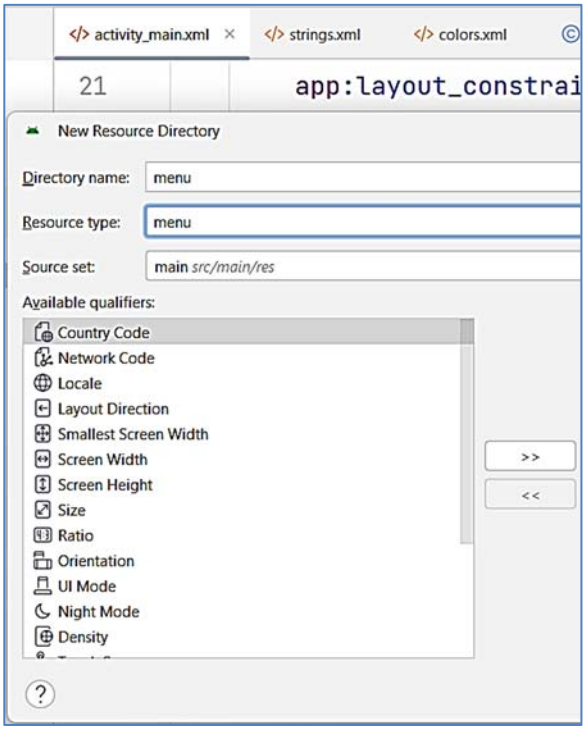
Rysunek 6.119 Tytuł aplikacji wyświetlony na pasku ToolBar

6.19.2 Tworzenie menu

Pasek narzędzi jak sama nazwa wskazuje ma swoje określone zadanie. Na jego powierzchni można umieszczać elementy, które pomogą w obsłudze aplikacji. Najważniejszą z zalet **ToolBar** jest możliwość stworzenia prostego menu. Tworzymy nową aplikację w której umieścimy kod z poprzedniego zadania, a następnie go zmodyfikujemy. Kod aplikacji znajduje się w katalogu **Apliakacja34b**.

Zaczniemy od menu zawierającego odnośniki do innych elementów aplikacji. Pracę rozpoczniemy od utworzenia kolejnego pliku zasobów o nazwie **menu** w którym będziemy definiować elementy i wygląd menu aplikacji.

W drzewie aplikacji klikamy na gałąź **res**. Z menu podręcznego wybieramy opcję **new** a następnie **New Resource File**. W polu **File name**: wpisujemy **menu_glowne,** a **Resource type** ustawiamy na **menu**. W drzewie katalogów pojawi się nowy katalog o nazwie **menu** w którym znajdziemy nowy plik zasobów **menu_glowne.xml**.



Rysunek 6.120 Tworzenie katalogu menu

W pliku **menu** umieszczamy elementy, które znajdziemy na naszym **pasku Narzędzi**. Kod pliku będzie wyglądał jak poniżej:

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

    <item android:id="@+id/menu1"
        android:title="Plik1"
        android:orderInCategory="1"
        app:showAsAction="never" />

    <item android:id="@+id/menu2"
        android:title="Plik2"
        android:orderInCategory="2"
        app:showAsAction="never" />

    <item android:id="@+id/menu3"
        android:title="Plik3"
        android:orderInCategory="3"
        app:showAsAction="never" />
</menu>
```

Listing 6.192 Kod pliku menu_glowne.xml

Pojedyncze pole **item** będzie tworzyć jedno pole w **menu**. Atrybut **android:title** nada każdemu z elementów tytuł, który będzie wyświetlany w menu aplikacji. **Android:orderInCategory** będzie decydować o miejscu wyświetlania pozycji w menu. W pierwszej kolejności wyświetlane są pola z mniejszą wartością tego atrybutu. **App:showAction** to atrybut, który określa sposób wyświetlania elementu. Ustawiona w przykładzie wartość „**never**" oznacza, że element będzie schowany (zwinięty) w menu. Inne opcje tego atrybutu to:

- **ifRoom** – umieści element na pasku tylko, gdy będzie na nie odpowiednia ilość miejsca. Gdy miejsca będzie zbyt mało element trafi do rozwijanego menu;
- **withText** – poza ikoną wyświetli w menu również tytuł zamieszczony w atrybucie **title**;
- **always** – umieści element na pasku. Zawsze będzie on na wierzchu.

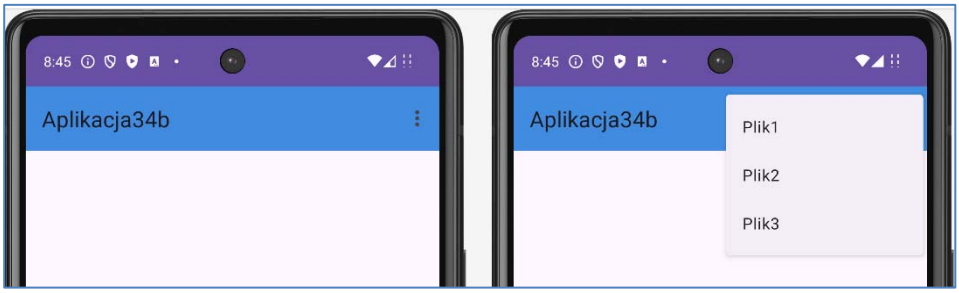
Utworzone menu trzeba połączyć z resztą plików aplikacji. Zrobimy to w pliku aktywności **MainActivity.java**.

```
public class MainActivity extends AppCompatActivity {

    @Override
protected void onCreate( Bundle savedInstanceState ) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
Toolbar toolbar = findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
    @Override
public boolean onCreateOptionsMenu(Menu menu) {
getMenuInflater().inflate(R.menu.menu_glowne, menu);
   return super.onCreateOptionsMenu(menu);
 }
}
```

Listing 6.193 Plik aktywności definiujący obsługę menu

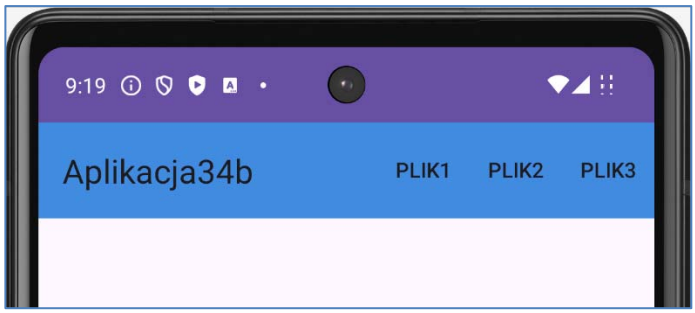
W klasie głównej dodajemy nową **metodę** typu **boolean**, o nazwie **onCreateOptionsMenu**. Tworzymy **instancje** do menu za pomocą **getMenuInflater** i zwracamy listę elementów menu.

Wygląd aplikacji po uruchomieniu przedstawia poniższy rysunek.



Rysunek 6.121 Rozwijane menu na pasku ToolBar – zwinięte (lewa strona), rozwinięte (prawa strona)

Na pasku pojawiła się ikona menu (trzy kropki). Kliknięcie ich wysunie opcje menu. Jeżeli w pliku **menu_glowne.xml** zmienimy opcję atrybutu **app:showAsAction** na wartość **always** aplikacja będzie wyglądać nieco inaczej.



Rysunek 6.122 Przyciski akcji widoczne na pasku TollBar

6.19.3 Tworzenie nawigacji – przycisk Powrót

Naszą aplikację wyposażymy teraz w dodatkowy **przycisk akcji**, którego zadaniem będzie powrót do głównej aktywności. Zaczniemy edycję od utworzenia **drugiej aktywności**.

Kolejnym krokiem jest edycja **pliku manifestu**. Musimy w nim umieścić informacje, która aktywność jest nadrzędna. Biorąc pod uwagę budowę aplikacji, na górze hierarchii będzie znajdować się **MainActivity.java** i taką adnotację trzeba umieścić w pliku **AndroidManifest.xml**. W pliku manifestu znajdujemy fragment odpowiedzialny za drugą aktywność czyli **Menu1.java.**

```
<activity
    android:name=".Menu1"
    android:exported="false" />
```

Listing 6.194 Fragment pliku manifestu

W tym miejscu trzeba umieścić informacje o tym, że **aktywnością rodzicem** jest **MainActivity**.

```
<activity
    android:name=".Menu1"
    android:exported="false"
    android:parentActivityName=".MainActivity"/>
```

Listing 6.195 Fragment pliku manifestu z ustawioną hierarchią aktywności

Modyfikujemy kod aktywności głównej. Dodajemy do kodu metodę **onOptionsItemSelected**, która będzie reagowała na przyciśnięcie przycisku menu. Kod tej metody znajduje się poniżej:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    if (item.getItemId() == R.id.menu1) {
        Intent intent = new Intent(this, Menu1.class);
        startActivity(intent);
        return true;
    }
    return super.onOptionsItemSelected(item);
}
```

Listing 6.196 Definicja metody onOptionsItemSelected

Metoda pobiera **id** elementu **item**. Wykorzystuje go w instrukcji wyboru **if**. Po kliknięciu w przycisk menu uruchamia się **intencja**, która przenosi nas do aktywności **Menu1.java**. W tej części aplikacji mamy zdefiniowaną **akcję** do przycisku **PLIK1**. Uzupełnienie kodu dla pozostałych przycisków nie powinno stanowić problemu. Zachęcam do samodzielnego uzupełnienia metody.

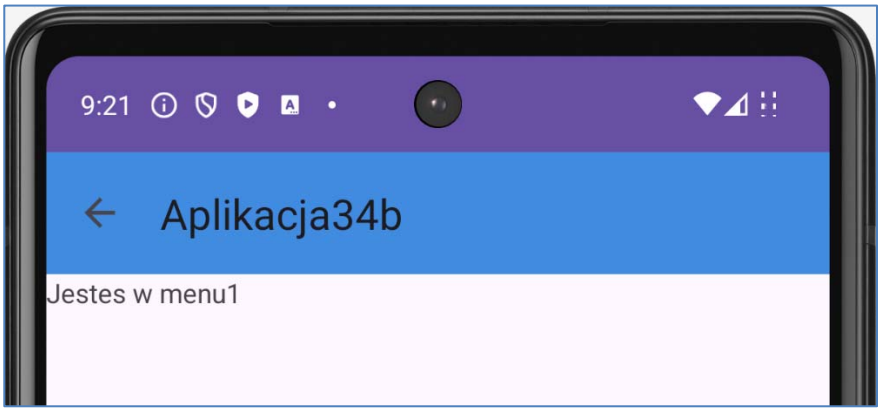
W drugiej aktywności tworzymy przycisk sterujący „**w górę**”.

```
Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
 setSupportActionBar(toolbar);
 ActionBar actionBar = getSupportActionBar();
 actionBar.setDisplayHomeAsUpEnabled(true);
```

Listing 6.197 Tworzenie obiektu Toolbar i nadanie mu funkcjonalności

Kod ten umieszczamy w funkcji **onCreate**. Dwie pierwsze linijki są nam znane. Tworzą odwołanie do pliku układu, a konkretnie do umieszczonego tam elementu **ToolBar**. Dwie ostatnie tworzą obiekt typu **ActionBar** i przycisk nawigacji „**w górę**”.

Teraz uruchamiamy aplikację. Po wciśnięciu przycisku **Menu1** zostaniemy przeniesiemy do **drugiej aktywności**, gdzie na **pasku ToolBar** pojawi się **ikona strzałki**. Kliknięcie jej spowoduje powrót do aktywności głównej.



Rysunek 6.123 Okno aplikacji z widoczną ikoną Powrót

Pełny kod aktywności głównej **MainActivity.java**:

```
package przyklad.nr34a;

import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.MenuItem;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate( Bundle savedInstanceState )
{
```

```
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        Toolbar toolbar = (Toolbar)
findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);
    }
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
       getMenuInflater().inflate(R.menu.menu_glowne,
menu);
        return super.onCreateOptionsMenu(menu);
    }

    @Override
    public boolean onOptionsItemSelected(MenuItem item)
{
 if (item.getItemId() == R.id.menu1) {
  Intent intent = new Intent(this, Menu1.class);
            startActivity(intent);
            return true;
        }
        return super.onOptionsItemSelected(item);
    }
}
```

Listing 6.198 Kod aktywności głównej

Kod aktywności **Menu1.java**:

```
package przyklad.nr34a;

import androidx.appcompat.app.ActionBar;
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import android.os.Bundle;
public class Menu1 extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_menu1);
Toolbar toolbar = findViewById(R.id.toolbar);
setSupportActionBar(toolbar);
ActionBar actionBar = getSupportActionBar();
actionBar.setDisplayHomeAsUpEnabled(true);
 }
}
```

Listing 6.199 Kod aktywności pierwszej opcji menu

6.19.4 Tworzenie własnych ikon.

Pasek **ToolBar** umożliwia wyświetlanie na pasku własnych ikon zamiast napisów. W programie można użyć ikon zaprojektowanych samodzielnie w formie niewielkich obrazków. Umieszczamy je w katalogu **Drawable**. Można również skorzystać z bazy ikon, które znajdziemy w **Internecie**. Całkiem sporą ich bazę znajdziemy na stronie:

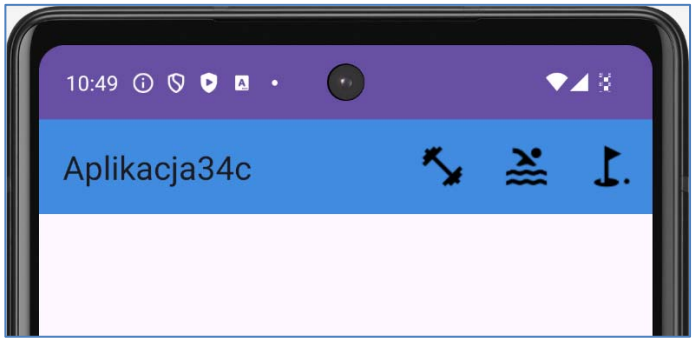
https://fonts.google.com/icons?icon.set=Material+Symbols.

Pobrane obrazki również umieszczamy w katalogu **Drawable**. W następnym przykładzie wykorzystamy gotową aplikację z przykładu **Aplikacja34a**. A pełny kod aplikacji znajduje się w katalogu **Aplikacja34c**. W bieżącym przykładzie nazwy tekstowe **itemów** zastąpimy **ikonami**. Kod pliku **menu_glowne.xml** będzie wyglądał tak jak poniżej:

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

    <item android:id="@+id/menu1"
        android:title="Plik1"
        android:orderInCategory="2"
        app:showAsAction="always"
        android:icon="@drawable/ik1"/>

    <item android:id="@+id/menu2"
        android:title="Plik2"
        android:orderInCategory="3"
        app:showAsAction="always"
        android:icon="@drawable/ik2"/>

    <item android:id="@+id/menu3"
        android:title="Plik3"
        android:orderInCategory="4"
        app:showAsAction="always"
        android:icon="@drawable/ik3"/>
</menu>
```

Listing 6.200 Kod układu z odnośnikami do ikon obrazkowych

Do każdego z **itemów** została dodana jedna linijka z atrybutem: **android:icon**, a w nim odwołanie do katalogu **drawable** i odpowiedniego **pliku graficznego**. Po uruchomieniu aplikacja będzie wyglądać tak jak poniżej.

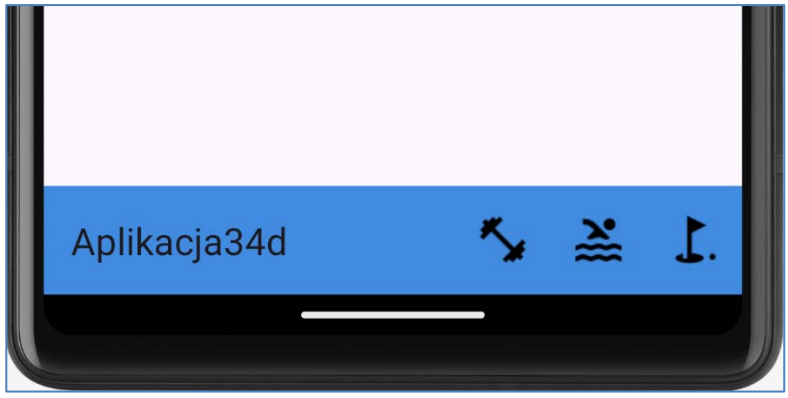


Rysunek 6.124 Pasek ToolBar z obrazkowymi ikonami

6.19.5 Zmiana miejsca położenia paska TollBar.

Pasek narzędzi **ToolBar** jest traktowany tak jak każdy inny widok. W związku z tym mamy możliwość przemieszczania go po ekranie aplikacji, a co za tym idzie zmieniać jego domyślne miejsce położenia. W następnym ćwiczeniu przesuniemy pasek **ToolBar na dół aplikacji**.

Kod aplikacji znajdziesz w katalogu **Aplikacja34d**. Najprostszym sposobem będzie wykorzystanie układu **RelativeLayout** zamiast zwykłego **LinearLayout**. Zmian dokonamy w pliku **activity_main.xml**. Pierwszym krokiem będzie wspomniana zmiana układu. Następnie w ustawieniach widoku **android.support.v7.widget**.**Toolbar** dodajemy linijkę, która ustawi widok na dole układu: **androidx.appcompat.widget.Toolbar**. **Pasek ToolBar** ustawi się na dole okna.



Rysunek 6.125 Pasek ToolBar umieszczony na dole aplikacji

6.19.6 Menu wysuwane – Szuflada nawigacyjna

Kod aplikacji i zasoby potrzebne do jej stworzenia znajdziesz w katalogu o nazwie **Aplikacja35**.

W kolejnej aplikacji utworzymy tzw. **szufladę nawigacyjną**. Będzie to pewnego rodzaju lista, która za pomocą przycisku menu (tzw. menu hamburgera) będzie wysuwać opcję do wyboru. W aplikacji wykorzystamy obrazki, które wykorzystaliśmy przy tworzeniu **CardView** i stworzymy prostą książkę kucharską. Ponieważ aplikacja będzie opierać się na fragmentach w następnym kroku utworzymy cztery fragmenty z nazwami sałatek.

Będą to odpowiednio:

- **FCaprese** – sałatka Caprese;
- **FKrewetki** – sałatka z krewetkami;
- **FOwocowa** – sałatka owocowa;
- **FJajko** – sałatka z jajkiem;

Każdy z **fragmentów** będzie miał swój **plik układu**. Ich nazwy utworzą się same wraz z utworzeniem fragmentu. **Nowy fragment** tworzymy klikając nazwę pakietu w **katalogu Java** znajdującego się w drzewie katalogów. Operacja ta była wykonywana w rozdziale o **Fragmentach**. Jeżeli potrzebujesz podpowiedzi w jaki sposób to zrobić, wróć się kilka stron wcześniej. Utworzone **Fragmenty** trzeba teraz uzupełnić. Zaczniemy od pliku układu **fragmentu FCaprese** o nazwie **fragment_fcaprese.xml**.

Wykorzystamy tradycyjnie układ liniowy: **LinearLayout**. Umieścimy w nim:

- **Pole tekstowe** – **TextView** w którym umieścimy nazwę sałatki;
- **Obrazek** – **ImageView** w którym umieścimy obrazek sałatki;
- **Pole tekstowe** – **TextView** – tutaj umieścimy listę składników potrzebnych do wykonania sałatki.

Kod pliku **fragment_f_caprese.xml**:

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

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Sałatka Caprese"
        android:textSize="15pt"
        />
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:src="@drawable/caprese"
        />
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Składniki:\n -mozarella \n-pomidor \
świeża bazylia \n-oliwa z oliwek"
        android:textSize="10pt"
        />

</LinearLayout>
```

Listing 6.201 Kod układu tworzącego fragment1

Pozostałe pliki dotyczące fragmentów sałatek będą wyglądały bardzo podobnie. Zmienić trzeba tylko nazwę sałatki w widoku **TextView**, nazwę obrazka w **ImageView**, oraz listę składników.

W plikach aktywności umieszczamy poniższy kod aktywności **FCaprese.java**:

```
package przyklad.nr35;

import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import androidx.fragment.app.Fragment;

public class FCaprese extends Fragment {

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

Listing 6.202 Plik aktywności głównej

Pozostałe pliki aktywności czyli: **FJajko.java**, **FKrewetki.java**, oraz **FOwocowa.java** będą miały podobne kody. Zmienić należy tylko nazwę **klasy głównej**, oraz nazwę **pliku układu**.

Tworzymy **pasek ToolBar**. Utworzymy go w osobnym pliku, aby łatwiej można było go dodawać do innych elementów programu. Tworzymy nowy plik zasobów i nazywamy go **toolbar.xml**, jego kod przedstawiono poniżej:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.appcompat.widget.Toolbar
  android:id='@+id/toolbar'
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="?attr/actionBarSize"
  android:background="#D57EBF"
  android:theme="@style/ThemeOverlay.AppCompat.Dark.ActionBar"
/>
```

Listing 6.203 Plik toolbar.xml

Zmieniamy domyślny styl aplikacji w **pliku manifestu**. Nowy styl nazwiemy **Menu** i taką właśnie nazwę wpiszemy w pliku **AndroidManifest.xml**.

```
<application
    android:allowBackup="true"

android:dataExtractionRules="@xml/data_extraction_rules"
    android:fullBackupContent="@xml/backup_rules"
    android:icon="@mipmap/ic_launcher"
    android:label="@string/app_name"
    android:roundIcon="@mipmap/ic_launcher_round"
    android:supportsRtl="true"
    android:theme="@style/Menu"
    tools:targetApi="31">
```

Listing 6.204 Fragment pliku manifestu

Dodany **styl** edytujemy w pliku **styles.xml**. Plik powinien znajdować się w **katalogu values**. Jeżeli jeszcze go tam nie ma trzeba go utworzyć. Gotowy plik uzupełniamy poniższym kodem:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>

<style name="Menu"
parent="Theme.AppCompat.Light.NoActionBar">

<item name="colorPrimary">@color/menu1</item>
<item name="colorPrimaryDark">@color/menu2</item>
<item name="colorAccent">@color/menu3</item>

</style>
<style name="Menu.menu1">
<item name="android:textSize">12pt</item>
</style>
</resources>
```

Listing 6.205 Definicja tematu tworzącego menu

Użyte w stylu **kolory** trzeba zdefiniować w pliku **colors.xml**. W aplikacji zostały użyte wybrane kolory. Można je jednak zmienić według własnego uznania.

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color name="black">#FF000000</color>
    <color name="white">#FFFFFFFF</color>
    <color name="menu1">#EA0C63</color>
    <color name="menu2">#EC75A3</color>
    <color name="menu3">#EAC3D2</color>
</resources>
```

Listing 6.206 Plik colors.xml

Przyszedł czas na utworzenie **pliku menu**. Będzie to plik zasobów o nazwie **menu.xml**, który utworzymy w **podkatalogu menu**. W pliku będą się znajdować cztery opcje **item**. Każda z nich będzie reprezentować **jedno pole** na karcie.

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

    <item
        android:id="@+id/Caprese"
        android:title="Caprese"
        android:checked="true" />
    <item
        android:id="@+id/Jajko"
        android:title="Jajko" />
    <item
        android:id="@+id/Krewetki"
        android:title="Krewetki" />
    <item
        android:id="@+id/Owocowa"
        android:title="Owocowa" />

</menu>
```

Listing 6.207 Plik menu.xml

Do **paska ToolBar** dodamy **przycisk** za pomocą, którego będzie można otworzyć menu. Najpierw tworzymy obiekt klasy **Toolbar** i tworzymy **referencję** do **widoku** z pliku układu.

```
Toolbar toolbar = findViewById(R.id.toolbar);
```

Listing 6.208 Referencja do paska ToolBar

Tworzymy kolejny obiekt tym razem jest to **obiekt drawer** pochodzący z klasy **DrawerLayout** i tworzymy referencję do elementu **drawer_layout** znajdującym się w pliku układu **activiy_mail.xml**.

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

Listing 6.209 Tworzenie obiektu DrawerLayout

Instrukcja **ActionBarDrawerToogle** utworzy w naszej aplikacji tzw. menu hamburgera czyli ikonę składającą się z trzech kresek, których kliknięcie spowoduje wyświetlenie menu bocznego.

Konstruktor tej **klasy** ma aż pięć parametrów:

- **Context** – odnosi się do klasy w której będziemy tworzyć menu;
- **Układ DrawerLayout** – drawer;
- **Pasek Toolbar** – toolbar;
- **Łańcuch znaków** – wyświetlany podczas otwierania menu - R.string.Otworz;
- **Łańcuch znaków** – wyświetlany podczas zamykania menu - R.string.Zamknij;

```
ActionBarDrawerToggle toggle =
   new ActionBarDrawerToggle(this, drawer, toolbar,
R.string.Otworz,
R.string.Zamknij);
```

Listing 6.210 Tworzenie obiektu ActionBarDrawerToggle

Oba łańcuchy zostały zdefiniowane w pliku **strings.xml**. Ich deklaracja wygląda następująco:

```
<string name="Otworz">Otwórz Menu</string>
<string name="Zamknij">Zamknij Menu</string>
```

Listing 6.211 Definicja łańcuchów w pliku strings.xml

Funkcja **drawer.addDrawerListener(toggle)** – doda przełącznik do układu. Ostatnia funkcja zdefiniowana we fragmencie kodu synchronizuje działanie ikony – menu hamburgera.

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

Listing 6.212 Synchronizacja działania menu hamburgera

Pozostało nam jeszcze zdefiniowanie akcji wciśnięcia przycisku, co prezentuje poniższy kod:

```
implements
NavigationView.OnNavigationItemSelectedListener {
```

Listing 6.213 Zdefiniowanie akcji wciśnięcia przycisku

Umieszczamy go w klasie głównej pliku **MainActivity.java**. Jest on odpowiedzialny za reakcję aplikacji na kliknięcie opcji przez użytkownika. Natomiast te dwie linijki umieszczamy są w **klasie OnCreate**:

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

Listing 6.214 Tworzenie obiektu nasłuchującego

Ich zadaniem jest zakomunikowanie aplikacji, kiedy użytkownik kliknął wybraną opcję. Kolejną czynnością jest zdefiniowanie klasy, która w reakcji na kliknięcie opcji otworzy żądany fragment.

```
public boolean onNavigationItemSelected (MenuItem item){
    int id = item.getItemId();
    Fragment fragment;
    switch (id) {
        case R.id.Jajko:
            fragment = new FJajko();
            break;
        case R.id.Krewetki:
            fragment = new FKrewetki();
            break;
        case R.id.Owocowa:
            fragment = new FOwocowa();
            break;
        default:
            fragment = new FCaprese();
    }

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

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

Listing 6.215 Definicja klasy onNavigationItemSelected

Klasa **onNavigationItemSelected** ma jeden **parametr** i jest nim nasze menu. Najpierw tworzone są trzy obiekty:

- **Id** – typu całkowitego – który będzie pobierał id pola item z pliku **menu.xml**;
- **Fragment typu fragment** – obiekt ten zajmie się uruchomieniem odpowiedniego pliku fragmentu;

- **Prosta instrukcja switch** – zależna od id pobranego pola item, co uruchomi odpowiedni plik fragmentu.

Jeżeli nie zostanie wybrana żadna z opcji fragmentem domyślnym będzie pierwszy element na liście czyli w tym przypadku fragment **FCaprese**. Kompletny kod aktywności **ActivityMain.java** wygląda następująco:

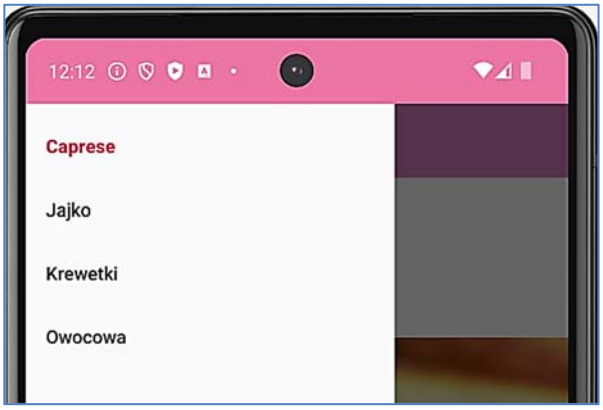
```
package przyklad.nr35;

import androidx.appcompat.app.ActionBarDrawerToggle;
import androidx.appcompat.app.AppCompatActivity;
import androidx.appcompat.widget.Toolbar;
import androidx.core.view.GravityCompat;
import androidx.drawerlayout.widget.DrawerLayout;
import androidx.fragment.app.Fragment;
import androidx.fragment.app.FragmentTransaction;

import android.annotation.SuppressLint;
import android.os.Bundle;
import android.view.MenuItem;

import
com.google.android.material.navigation.NavigationView;

public class MainActivity extends AppCompatActivity
  implements
NavigationView.OnNavigationItemSelectedListener {

  @Override
  protected void onCreate (Bundle savedInstanceState){
  super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
Toolbar toolbar = findViewById(R.id.toolbar);
 setSupportActionBar(toolbar);
DrawerLayout drawer = findViewById(R.id.drawer_layout);
 ActionBarDrawerToggle toggle =
 new ActionBarDrawerToggle(this,drawer,toolbar,
 R.string.Otworz,
 R.string.Zamknij);
 drawer.addDrawerListener(toggle);
 toggle.syncState();
 NavigationView navigationView = (NavigationView)
findViewById(R.id.nav_view);

navigationView.setNavigationItemSelectedListener(this);
```

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

        @Override
public boolean onNavigationItemSelected (MenuItem item){
 int id = item.getItemId();
 Fragment fragment;
switch (id) {
              case R.id.Jajko:
              fragment = new FJajko();
               break;
              case R.id.Krewetki:
              fragment = new FKrewetki();
               break;
              case R.id.Owocowa:
              fragment = new FOwocowa();
               break;
              efault:
              fragment = new FCaprese();
}
FragmentTransaction ft =
getSupportFragmentManager().beginTransaction();
ft.replace(R.id.content_frame, fragment);
  ft.commit();

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

Listing 6.216 Kod aktywności głównej

Wynik działania aplikacji prezentuje poniższy rysunek:

Rysunek 6.126 Szuflada nawigacyjna – wersja zwinięta



Rysunek 6.127 Szuflada nawigacyjna – wersja rozwinięta

6.20 Kalendarz i Zegar – Klasy DataPicker i TimePicker

6.20.1 Wstawianie kalendarza

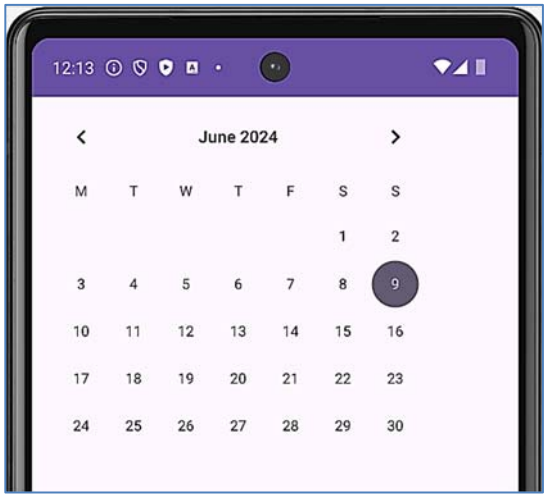
Kod aplikacji znajduje się w folderze o nazwie **Aplikacja36**.

Data i czas są ważnymi elementami aplikacji. W kolejnym przykładzie zaprezentujemy w jaki sposób w aplikacji można wstawić kalendarz. Za umieszczenie kalendarza odpowiadać będzie widok **CalendarView**. Kod układu z widokiem kalendarza będzie prezentował się następująco:

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <CalendarView
        android:id="@+id/kalendarz1"
        android:layout_width="wrap_content"
        android:layout_height="0dp"
        android:layout_weight="1" />

</LinearLayout>
```

Listing 6.217 Układ z widokiem Calendar View

Prosty kalendarz w aplikacji będzie wyglądał tak jak ilustruje to poniższy rysunek:



Rysunek 6.128 Kalendarz w aplikacji

Widok **CalendarView** ma kilka ciekawych atrybutów, które mogą zmienić jego wygląd. Jak widać na powyższym rysunku pierwszym dniem tygodnia w kalendarzu jest niedziela (ang. Sunday). Atrybut **android:firstDayOfWeek** ustawiony na wartość **2** zmieni pierwszy dzień tygodnia na poniedziałek.

Kolejny atrybut: **android:dateTextAppearance** zmieni wielkość czcionki. Posiada on następujące opcje:

- **@android:style/TextAppearance.Large** – zmienia na dużą czcionkę;
- **@android:style/TextAppearance.Small** – zmienia na czcionkę;
- **@android:style/TextAppearance.Medium** – zmienia na średnią czcionkę;
- **@android:style/TextAppearance.Large.Inverse** – zmienia na dużą czcionkę i jednocześnie odwraca kolory.

Warto wypróbować kilka z nich.

6.20.2 Wstawienie zegara

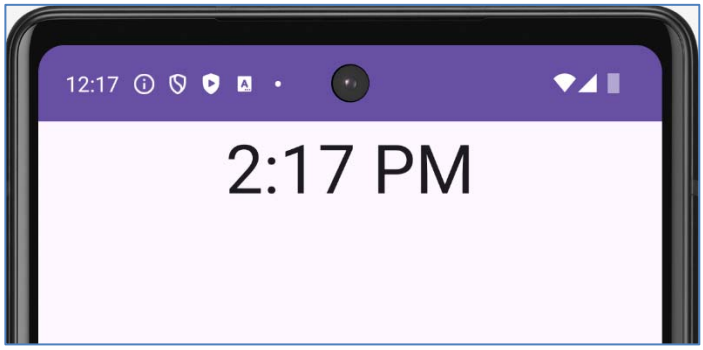
Android umożliwia wstawienia zegara cyfrowego. Pomocny będzie tutaj widok **TextClock**. Kod wstawiający podany widżet będzie wyglądał następująco:

Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja37**.

```
<?xml version="1.0" encoding="utf-8" ?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <TextClock
        android:id="@+id/zegar1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:textSize="20pt"

        />
</LinearLayout>
```

Listing 6.218 Układ z widokiem TextClock

Po uruchomieniu aplikacji zegar będzie wyglądał w ten sposób:



Rysunek 6.129 Aplikacja wyświetlająca czas w formacie 12-godzinnym

W pierwszej kolejności zwracamy uwagę na formę zegara. Jest on domyślnie ustawiony w zapisie **12-godzinnym**. Zwyczajowo używamy zegara **24-godzinnego**. Spróbujemy „naprawić" nasz zegar. W pliku układu w widoku TextClock dodajemy dodatkowy atrybut:

```
android:format24Hour="HH:mm:ss"
```

Listing 6.219 Ustawienie formatu wyświetlenia zegara

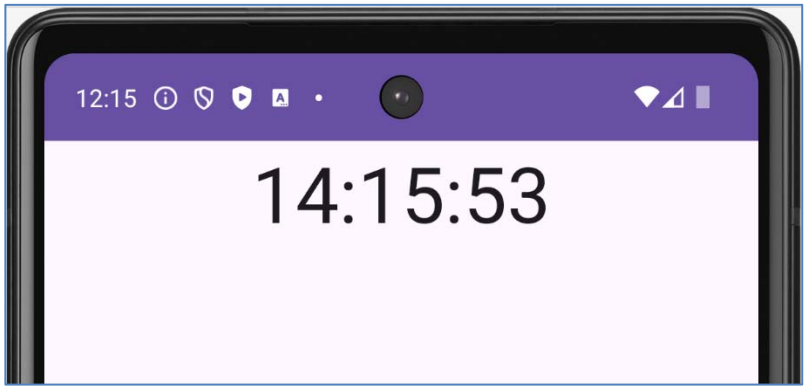
Sygnalizujemy, iż chcemy żeby nasz zegar był w formacie 24-godzinnym i żeby obok godzin i minut wyświetlał również sekundy. Zmian należy również dokonać w pliku aktywności. Kod aktywności będzie wyglądał następująco:

```
package przyklad.nr37;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextClock;
public class MainActivity extends AppCompatActivity {
    private TextClock zegar1;
    @Override
  protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_main);
  this.zegar1 = this.findViewById(R.id.zegar1);
  this.zegar1.setFormat12Hour(null);
    }
}
```

Listing 6.220 Pełny kod aktywności

Tworzymy obiekt klasy TextClock o nazwie zegar1. W klasie **onCreate** tworzymy **referencje** tego obiektu do widoku **TextClock** znajdującego się w pliku układu. Następnie wyłączamy wyświetlanie zegara w formie 12-godzinnej poprzez ustawienie metody **setFormat12Hours** na **null**.

Po wprowadzeniu zmian w kodzie zegar będzie wyglądał następująco:



Rysunek 5.119: Aplikacja wyświetlająca czas w formacie 24-godzinnym

Widzimy zmianę formatu. Godzina wyświetla się w notacji 24-godzinnej. Oprócz godzin i minut widać również sekundy.

Zadbać musimy jeszcze o jeden szczegół. Mianowice ustawienie strefy czasowej. Do kodu aktywności trzeba dodać jedną linijkę kodu:

```
zegar1.setTimeZone("Europe/Warsaw");
```

Listing 6.221 Ustawienie strefy czasowej

Ustawiamy strefę czasową na czas europejski w Warszawie. Aplikacja zegar jest w tym momencie gotowa.

6.20.3 Pobieranie daty

Użytkownik korzystający z aplikacji ma możliwość pobierania daty i czasu i przekazania ich np. do bazy danych, albo innego miejsca w aplikacji. Z pomocą przyjdą nam tutaj dwie klasy:

- **Datapicker** – która pomoże w pobraniu od użytkownika daty;
- **Timepicker** – która pobierze godzinę.

Napiszemy teraz aplikację o nazwie **Datapicker**, która pobierze od użytkownika **datę**. Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja38**.

Kod układu aplikacji będzie przypominał ten poniżej:

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

    <DatePicker
    android:id="@+id/PobierzDate"
    android:layout_width="match_parent"
    android:layout_height="match_parent">
</DatePicker>

</LinearLayout>
```

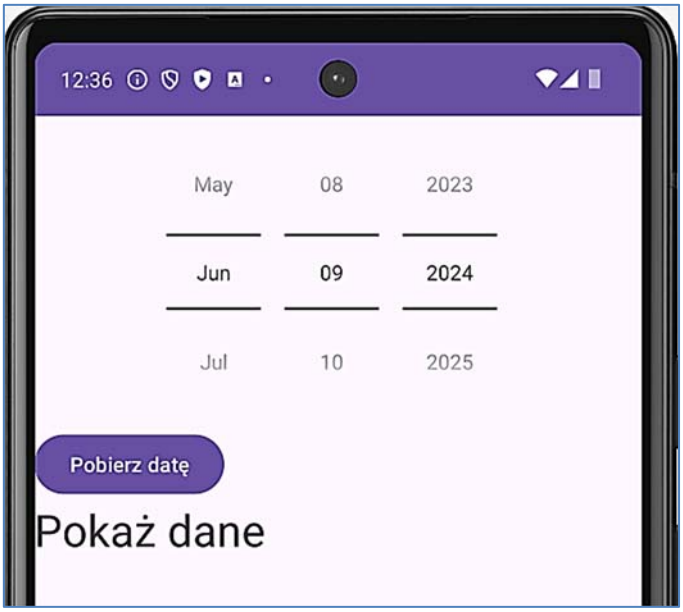
Listing 6.222 Plik układu z elementem DataPicker

Aplikacja po uruchomieniu będzie przedstawiała kalendarz. Zupełnie jak w ćwiczeniu z widżetem **CalendarView**. W wypadku **DatePicker** mamy również możliwość wyświetlenia pola wyboru daty w formie **Spinnera**. W kodzie układu wprowadzamy następujące zmiany:

```
<DatePicker
    android:id="@+id/PobierzDate"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:datePickerMode="spinner"
    android:calendarViewShown="false" >
</DatePicker>
```

Listing 6.223 Definicja widżetu DataPicker

Atrybut **android:datePickerMode** ustawiamy na **Spinner**, a widok **calendarViewShow** wyłączamy ustawiając na **false**. Widżet po uruchomieniu aplikacji będzie wyglądał tak:



Rysunek 6.130 DataPicker – narzędzie służące do ustawienia daty. Wersja: spinner

Napiszemy teraz aplikację, która pobierze dane z kalendarza. Tworzymy **układ liniowy** w którym umieścimy widżety: **Datapicker**, **Button**, oraz **TextView.**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <DatePicker
        android:id="@+id/PobierzDate"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:firstDayOfWeek="2"
        android:layout_gravity="center"/>
```

```
    <Button
        android:id="@+id/Przycisk"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Pobierz datę"
        android:onClick="Pobierz"/>
    <TextView
        android:id="@+id/text1"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Pokaż dane"
        android:textSize="30dp"/>

</LinearLayout>
```

Listing 6.224 Układ liniowy z widokiem DataPicker

Dla tak przygotowanego układu tworzymy aktywność.

```
package przyklad.nr38;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.DatePicker;
import android.widget.TextView;
import android.view.View;

public class MainActivity extends AppCompatActivity {
    TextView textView1;
    DatePicker datePicker;
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView1=findViewById(R.id.text1);
        datePicker=findViewById(R.id.PobierzDate);
    }
    public  void  Pobierz(View view)
    {
        int dzien=datePicker.getDayOfMonth();
        int miesiac=datePicker.getMonth();
        int rok=datePicker.getYear();
        miesiac=miesiac+1;
        textView1.setText("Dzień: "+ dzien +"\nMiesiąc:
"+ miesiac + "\nRok: " + rok);
    }
}
```

Listing 6.225 Kod aktywności MainActivity.java

Tworzymy dwa obiekty napis typu **TextView**, oraz data typu **DatePicker.** Wykorzystamy je w dalszej części kodu.

```
TextView textView1;
DatePicker datePicker;
```

Listing 6.226 Utworzenie obiektów TextView i DatePicker.

W klasie **onCreate** pobieramy referencję do widoków **DataPicker** i **TextView** znajdujących się w pliku układu i zapisujemy je we wcześniej utworzonych zmiennych.

```
textView1 = findViewById(R.id.text1);
datePicker = findViewById(R.id.PobierzDate);
```

Listing 6.227 Utworzenie referencji do obiektów

Następnym zadaniem będzie zdefiniowanie klasy **Pobierz**, która zostanie wywołana w momencie kliknięcia przycisku.

```
int dzien=datePicker.getDayOfMonth();
int miesiac=datePicker.getMonth();
int rok=datePicker.getYear();

textView1.setText("Dzień: "+ dzien +"\nMiesiąc: "+
miesiac + "\nRok: " + rok);
```

Listing 6.228 Utworzenie zmiennych przechowujących dane pobrane z kalendarza

W klasie tej tworzymy trzy **zmienne** typu **int** i przypisujemy im **wartości** pobrane z k**alendarza**. Korzystamy tutaj z trzech **funkcji** operujących na datach:

- **getYear** – pobiera zaznaczony rok;
- **getDayOfMonth** – pobiera zaznaczony dzień miesiąca;
- **getMonth** – pobiera numer miesiąca;

Następnie pobrane dane wykorzystujemy do utworzenia łańcucha znaków, który wyświetlimy w miejscu pola **TextView**. Na uwagę zasługuje tutaj znak **\n**, który będzie przenosił tekst do następnej linijki. Aplikacja po uruchomieniu, zaznaczeniu bieżącej daty i wciśnięciu przycisku będzie zwracać następujące informacje:

Rysunek 6.131 Aplikacja pobierająca datę z kalendarza

Widzimy jednak, że coś się tutaj nie zgadza. Rok i dzień są prawidłowe. Problem pojawia się z numerem miesiąca. Wartości miesięcy ułożone są w tablicy. Pierwszy dzień miesiąca czyli styczeń zajmuje w tej tablicy pierwsze miejsce, a pobrany bieżący miesiąc czyli listopad jest tam na jedenastym miejscu. Problem wynika z budowy owej tablicy.

Pamiętać musimy, że tablice są indeksowane od zera. Zatem styczeń będzie się znajdował na miejscu 0, a listopad 10. Musimy zatem dodać jeszcze jedną prostą operację. Zanim wyświetlimy numer miesiąca (czyli zmienną miesiac) musimy go zwiększyć o 1. Dodajemy wobec tego poniższą linijkę do kodu programu:

```
miesiac=miesiac+1;
```

Listing 6.229 Zmiana wartości zmiennej miesiac

6.20.4 Pobieranie czasu – TimePicker

Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja39**.

Do pobrania czasu będzie nam służył komponent **TimePicker**. Zaczniemy od przedstawienia go w najprostszej postaci. Wstawimy go jako **widok** do pliku **układu**.

TimePicker podobnie jak **DatePicker** może przyjmować dwie różne formy. Pierwsza przypomina zegar w którym ustawiamy czas poprzez przesunięcie wskazówki. Druga forma przypomina **DataPicker** z ustawieniem **Spinner**. Poniższy kod wstawi w aplikacji obie formy dla porównania.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/LinearLayout1"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">
    <TimePicker
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/timePicker2"
        android:layout_gravity="center"
        android:timePickerMode="clock"/>
    <TimePicker
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/timePicker"
        android:layout_gravity="center"
        android:timePickerMode="spinner"/>

</LinearLayout>
```

Listing 6.230 Układ liniowy z widokiem TimePicker

Rysunek 6.132 TimePicker – wersja w postaci zegara (na górze), oraz wersja w postaci Spinnera(na dole)

Do ćwiczeń zastosujemy zegar ze wskazówkami czyli ten z atrybutem **android:timePickerMode** ustawionym na wartość **clock**. Przejdziemy do pliku aktywności głównej. Tam w klasie **onCreate** dodajemy kod, który ustawi oba zegary w notacji 24-godzinnej.

```
TimePicker picker = findViewById(R.id.zegar1);
picker.setIs24HourView(true);
```

Listing 6.231 Utworzenie referencji do elementu TimePicker

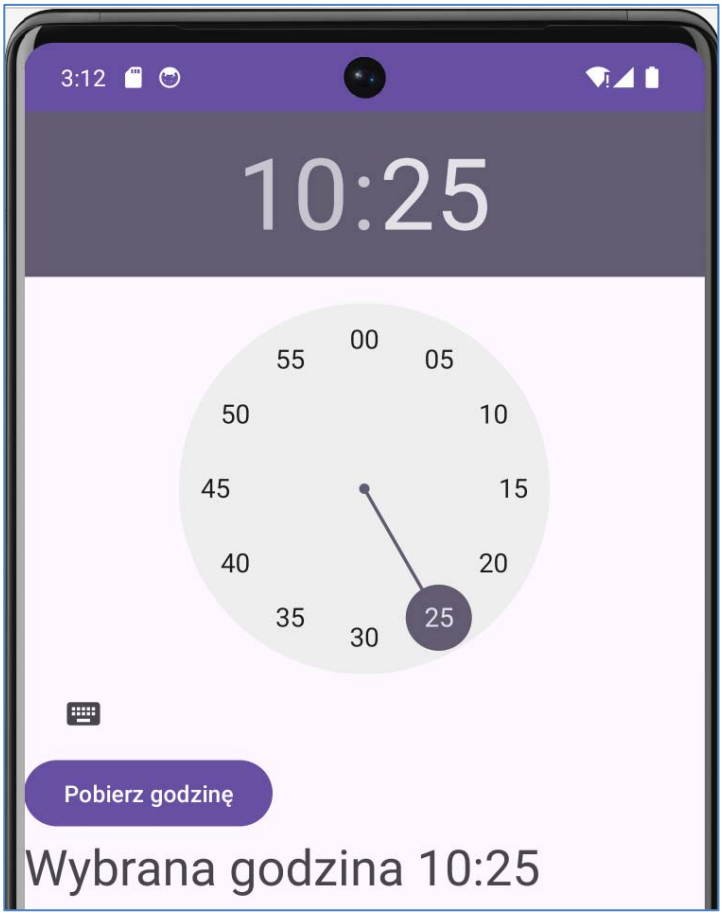
Następnym zadaniem będzie pobranie godziny z zegara. Przygotujmy najpierw plik układu, który będzie zbliżony do tego z poprzedniego zadania. Będzie on zawierał element T**imePicker**, przycisk **Button**, oraz **TextView**. W pliku **MainActivity.java** umieszczamy następujący kod:

```
package przyklad.nr39;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.TextView;
import android.widget.TimePicker;
import android.view.View;

public class MainActivity extends AppCompatActivity {
    TimePicker czas;
    TextView textView1;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        textView1 = findViewById(R.id.text1);
        czas = findViewById(R.id.zegar1);
        czas.setIs24HourView(true);
    }
    public void Pobierz(View view) {
        int hour, minute;
        hour = czas.getHour();
        minute =
czas.getMinute();textView1.setText("Wybrana godzina "+
hour +":"+ minute);
    }
}
```

Listing 6.232 Kod aktywności głównej

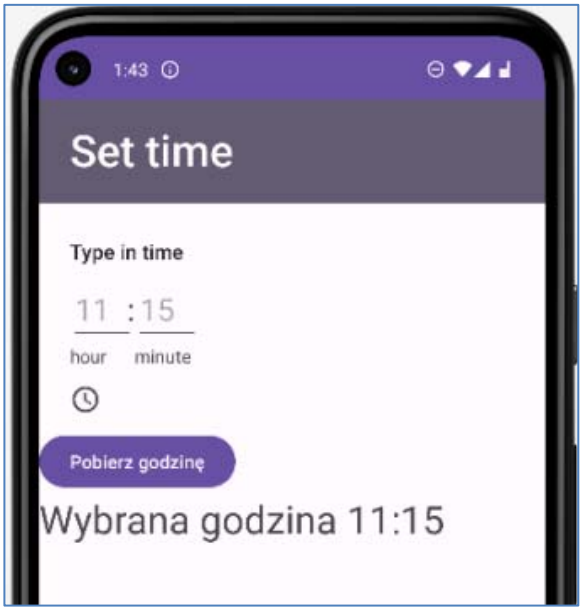
Tworzymy dwie zmienne typu **Timepicker** o nazwie **czas**, oraz **TextView** o nazwie **textView1**. W klasie **onCreate** tworzymy referencję do tych obiektów.

Klasa **Pobierz**, która zostanie uruchomiona po kliknięciu przycisku **Button,** co spowoduje pobranie godziny i minuty za pomocą dwóch metod: **getHour** i **getMinute**. Następnie w widoku **textView1** zostanie wyświetlona informacja o pobranym czasie.



Rysunek 6.133 Aplikacja pobierająca czas za pomocą ikony zegara

Jeżeli przyjrzymy się dokładnie aplikacji nad przyciskiem **Button** możemy zauważyć ikonę klawiatury. Jest to wbudowany mechanizm dołączony do widoku **TimePicker** z opcją **Clock**. Gdy przyciśniemy ikonę pojawi się nam formularz w którym ręcznie będziemy mogli wpisać wybraną godzinę.



Rysunek 6.134 Aplikacja pobierająca czas za pomocą formularza

Wciśnięcie ikonki zegara przełączy widok z powrotem.

6.21 Przechowywanie danych w aplikacji

6.21.1 Zapisywanie preferencji użytkownika

Każda rozbudowana aplikacja z której korzystamy bardzo często posiada pewne stałe ustawienia. Przykładem takiej aplikacji może być np. klient pocztowy czyli aplikacja do odbierania i wysyłania wiadomości e-mail. Stałe ustawienia to np. domyślny adres nadawcy (gdy korzystamy w programie z więcej niż jednego konta) czy podpis wiadomości.

Do ustalenia takich stałych zapisanych ustawień przydadzą nam się **Preferencje** użytkownika, które w **Android Studio** wykorzystują klasę **SharedPreferences**. Tworzymy prostą aplikację zawierającą układ składający się z jednego pola typu **checkbox**, pola **EditText** i przycisku **Button**.

Pełny kod aplikacji znajduje się a katalogu o nazwie **Aplikacja40**.

```
<LinearLayout
xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <CheckBox
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/regulamin"
        android:id="@+id/Regulamin"  />

    <EditText
        android:id="@+id/editText"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" />

    <Button
        android:id="@+id/button"
        android:text="Zapisz"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:onClick="onClick"/>
</LinearLayout>
```

Listing 6.233 Kod układu activity_main.xml

W pliku aktywności tworzymy preferencję:

```
SharedPreferences appPrefs =
getPreferences(MODE_PRIVATE);
```

Listing 6.234 Użycie metody getPreferences

Metoda **getPrefereces** pobiera ustawienia użytkownika i zachowuje je w obiekcie **appPrefs**. Argument **MODE_PRIVATE** przechowa wprowadzone ustawienia tylko w obszarze aplikacji dzięki czemu dane niewydostaną się na zewnątrz. Następnie pobieramy ustawienia elementów układu, aby można je było zapamiętać.

```
String tekst = appPrefs.getString("tekst", "");
EditText editText = (EditText)
findViewById(R.id.editText);
editText.setText(tekst);

boolean check = appPrefs.getBoolean("checkBox", false);
CheckBox checkBox = (CheckBox)
findViewById(R.id.Regulamin);
checkBox.setChecked(check);
```

Listing 6.235 Zapisywanie pobranych preferencji użytkownika

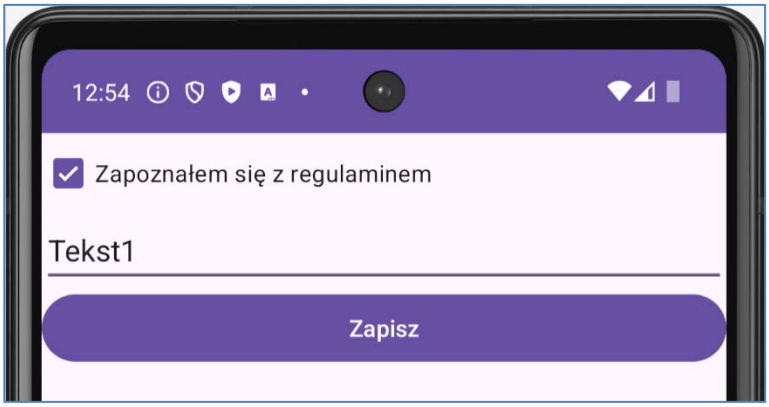
Mamy tutaj dwie metody **getString**, która pobierze tekst wprowadzony w polu **Edit** tekst, oraz **getBoolen**, która pobierze wartość z pola **checkbox**. Oczywiście mogą się tutaj pojawić dwie wartości: **zaznaczona** lub niezaznaczona. Wykorzystane są tutaj również dwie metody **setText**, oraz **setChecked**, które po pobraniu zapiszą ustawienia w aplikacji. Kolejnym etapem jest zdefiniowanie metody **OnClick**.

```
public void onClick(View view) {
    SharedPreferences sharedPreferences =
getPreferences(MODE_PRIVATE);
    SharedPreferences.Editor prefsEditor =
sharedPreferences.edit();

EditText editText = findViewById(R.id.editText);
String tekst = editText.getText().toString();
prefsEditor.putString("tekst", tekst);

CheckBox checkBox = findViewById(R.id.Regulamin);
    boolean cb = checkBox.isChecked();
    prefsEditor.putBoolean("checkBox", cb);

    prefsEditor.commit();
}
```

Listing 6.236 Zdefiniowanie metody onClick

Pobieramy utworzone ustawienia (**preferencje**). Następnie pobieramy wpisany tekst z pola **EditText**. Metoda **putString** zapisze plik w **preferencji**. Podobnie będziemy postępować w przypadku pola **chceckbox**. Tutaj mamy do czynienia z metodą **putBoolean**, która zapiszę stan przycisku: czy jest on **zaznaczony** czy **niezaznaczony**. Metoda **apply** zatwierdzi wprowadzone zmiany.

Uruchomimy teraz aplikację. Zaznaczymy okienko **CheckBox** i wpiszemy prosty tekst w **EditText**. Po wciśnięciu przycisku **Button** zaznaczone pola zostaną zapamiętane. Działanie aplikacji możemy sprawdzi po ponownym uruchomieniu. Okienko wyboru pozostanie zaznaczone, a w polu tekstowym będzie wprowadzony w poprzednim kroku tekst.



Rysunek 6.135 Formularz pobierający preferencję

Kompletny plik **MainActivity.java** wygląda w następujący sposób:

```
package com.example.aplikacja36;

import androidx.appcompat.app.AppCompatActivity;
import android.content.SharedPreferences;
import android.os.Bundle;
import android.view.View;
import android.widget.CheckBox;
import android.widget.EditText;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 SharedPreferences appPrefs =
getPreferences(MODE_PRIVATE);
String tekst = appPrefs.getString("tekst", "");
EditText editText = findViewById(R.id.editText);
editText.setText(tekst);

boolean check = appPrefs.getBoolean("checkBox", false);
CheckBox checkBox = findViewById(R.id.Regulamin);
        checkBox.setChecked(check);
    }
    public void onClick(View view) {
        SharedPreferences sharedPreferences =
getPreferences(MODE_PRIVATE);
        SharedPreferences.Editor prefsEditor =
sharedPreferences.edit();

EditText editText = findViewById(R.id.editText);
String tekst = editText.getText().toString();
prefsEditor.putString("tekst", tekst);
```

```
CheckBox checkBox =  findViewById(R.id.Regulamin);
 boolean cb = checkBox.isChecked();
        prefsEditor.putBoolean("checkBox", cb);
        prefsEditor.apply();
    }
}
```

Listing 6.237 Kod aktywności głównej MainActivity.java

6.21.2 Pobieranie i zapisywanie danych z wykorzystaniem plików.

Dane w aplikacji możemy pobrać z pliku znajdującego się zasobach aplikacji. Mamy również możliwość zapisania takich danych do pliku. Pierwsza wersja programu będzie zapisywać dane do pliku. Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja41**.

Tworzymy prosty układ **LinearLayout** zawierający: pole **EditText** z którego będziemy pobierać tekst do przesłania, oraz przycisk **Button**.

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

    <EditText
        android:id="@+id/tekst"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:hint="Wpisz tutaj tekst"
        android:textSize="15pt" />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Zapisz tekst"
        android:onClick="Zapisz" />

</LinearLayout>
```

Listing 6.238 Plik układu z formularzem do pobierania danych

W pliku aktywności tworzymy obiekt **EditText**, a w klasie **onCreate** pobieramy do niego **referencję** z pliku **układu**.

```
EditText editText;
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    editText = findViewById(R.id.tekst);
}
```

Listing 6.239 Tworzenie referencji do pola tekstowego

Najważniejsza część kodu będzie znajdować się w funkcji **Zapisz**, która uruchomi się po wciśnięciu przycisku **Button**.

```
String tekst = editText.getText().toString();
```

Listing 6.240 Tworzenie zmiennej tekstowej

Tworzymy zmienną typu **String** i przekazujemy do niej tekst pobrany z pola **EditText**.

```
try {
FileOutputStream file = openFileOutput("plik1.txt",
MODE_PRIVATE);
OutputStreamWriter stream = new
OutputStreamWriter(file);
    stream.write(tekst);
    stream.close();
    editText.setText("");
}
```

Listing 6.241 Sekcja try

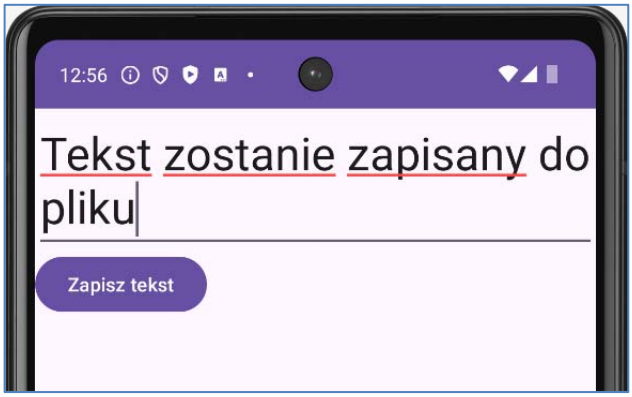
Następnie w sekcji **try** otwieramy nasz plik tekstowy. Wykorzystujemy do tego klasę **FileOutputStream**, która tworzy tzw. **strumień wyjściowy**. Metoda **openFileOutput** zawiera dwa parametry, pierwszy z nich to nazwa pliku w którym będziemy zapisywać dane. Jeżeli plik o podanej nazwie nie istnieje zostanie utworzony. Drugi parametr to tzw. **tryb otwarcia**. Wykorzystany w przykładzie tryb **MODE_PRIVATE** spowoduje usunięcie zapisanych wcześniej danych z pliku. Istnieje również drugi tryb **MODE_APPEND**, który spowoduje dopisanie do już istniejącego pliku nowych danych.

Za pomocą obiektu klasy **OutputStreamWriter** tworzymy strumień zapisujący dane. Metoda write zapisze dane do pliku, a **close** zamknie strumień po zakończeniu operacji. Na końcu metoda **set.text** obiektu **editText** zamieni wpisany przez użytkownika tekst na pusty łańcuch (czyli wyczyści pole). Zostaje nam jeszcze do zdefiniowania wyjątek.

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

Listing 6.242 Zdefiniowanie wyjątku

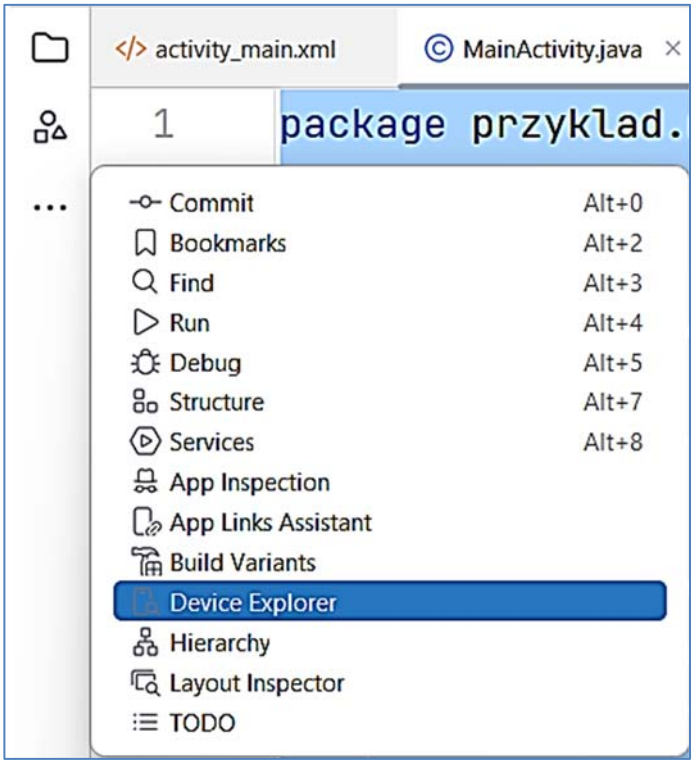
Jeżeli w podczas otwarcia pliku bądź zapisu do niego danych pojawią się błędy wyjątek wyświetli kod błędu.



Rysunek 6.136 Formularz pobierający tekst do pliku

Jeżeli udało nam się prawidłowo uruchomić aplikację i zapisaliśmy do niej dane, dobrze byłoby sprawdzić czy w pliku znajduje się podany przez nas tekst. Odnalezienie właściwego pliku nie jest takie proste. Zaczniemy od włączenia karty **Device File Explorer**.

Po lewej stronie edytora odnajdujemy ikonę **trzech kropek**. Klikamy w nią i z menu kontekstowego wybieramy opcję: **Device Explorer**.



Rysunek 6.137 Menu za pomocą, którego otwieramy narzędzie Device Explorer

Po kliknięciu w nią po prawej stronie pojawi się przycisk umożliwiający otwarcie eksploratora plików.

Name	Permissions	Date	Size
/	drwxr-xr-x	2009-01-01 00:00	4 KB
acct	drwxr-xr-x	2009-01-01 00:00	4 KB
apex	drwxr-xr-x	2024-05-26 06:59	1,4 KB
bin	lrw-r--r--	2009-01-01 00:00	11 B
cache	drwxrwx---	2009-01-01 00:00	4 KB
config	drwxr-xr-x	2024-05-26 06:58	0 B
d	lrw-r--r--	2009-01-01 00:00	17 B
data	drwxrwx--x	2024-05-26 06:59	4 KB
debug_ramdisk	drwxr-xr-x	2009-01-01 00:00	4 KB
dev	drwxr-xr-x	2024-05-26 06:59	2,6 KB

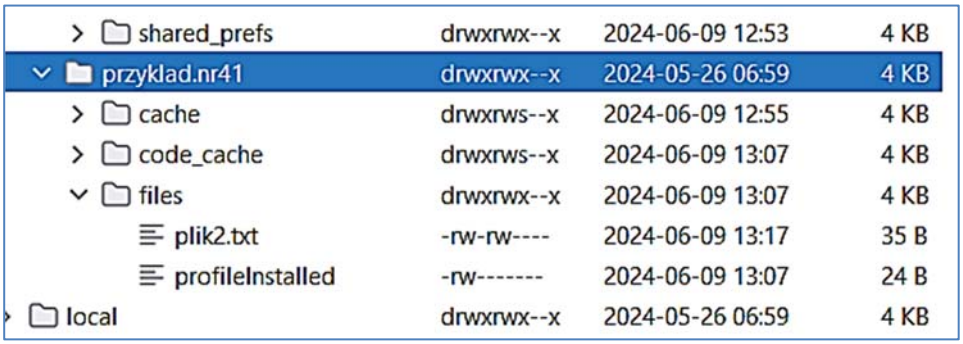
Rysunek 6.138 Device Explorer – Eksplorator plików zawartych w aplikacji

Na karcie tej pojawi się lista plików, które zostały utworzone przez Android Studio podczas całego czasu działania. Zapisane są na niej dane wszystkich

stworzonych aplikacji. Jeżeli podczas testowania używaliśmy różnych Emulatorów, również każda z wersji jest zapisywana.

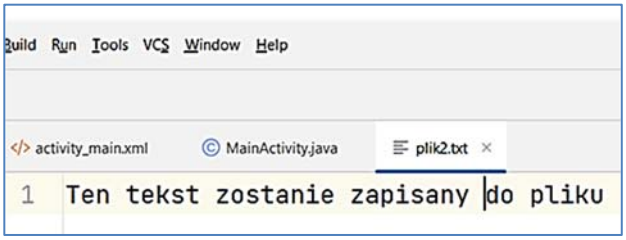
Poszukiwanie plików rozpoczniemy od rozwinięcia gałęzi Data, a następnie rozwinięcia kolejnego folderu o tej samej nazwie. W katalogu wewnętrznym wyszukujemy nazwy naszej aplikacji. Jej nazwa rozpocznie się od nazwy **package**. Tworzona w tym ćwiczeniu aplikacja będzie miała zatem nazwę **przykład.nr41.Aplikacja41** W katalogu z odpowiednią nazwą znajduje się podkatalog **Files**. Po rozwinięciu go znajdziemy nasz plik tekstowy.

Rysunek 6.139 Katalog Data



Rysunek 6.140 Plik z zapisanymi danymi widocznym w eksploratorze plików

Wystarczy go kliknąć, a w oknie aplikacji pojawi się karta z wypisanym tekstem. W łatwy sposób można odczytać jego zawartość.



Rysunek 6.141 Uruchomiony plik w oknie aplikacji

Jeżeli chcemy otworzyć plik na dysku komputera wystarczy w miejscu kodu kliknąć prawym klawiszem myszy i z menu kontekstowego wybrać opcję **Open in Explorer**.

6.21.3 Odczytanie danych z pliku.

Do układu naszej aplikacji dołożymy kolejny przycisk. Jego zadaniem będzie odczyt danych z pliku i wyświetlenie go w aplikacji. Kod aplikacji znajduje się w katalogu **Aplikacja41a**.

Do pliku aktywności dodajemy klasę **Przeczytaj**, która będzie się uruchamiała po wciśnięciu przycisku.

```
public void Przeczytaj(View view){
 try
 {
FileInputStream file2 = openFileInput("plik2.txt");
InputStreamReader stream = new InputStreamReader(file2);
BufferedReader reader = new BufferedReader(stream);
StringBuilder tekst = new StringBuilder();
  String linia;
   while ((linia = reader.readLine()) != null){
   tekst.append(linia).append("\n");
 }
   reader.close();
   editText.setText(tekst.toString());
  }
   catch (IOException ex) {
        ex.printStackTrace();
    }
}
```

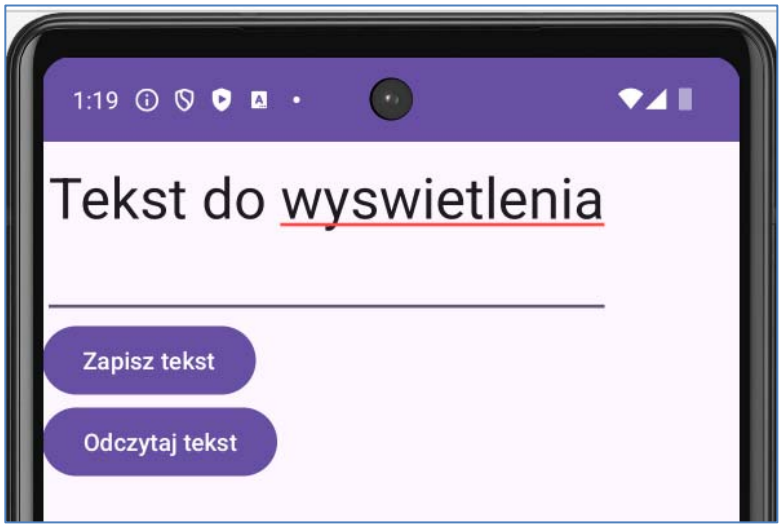
Listing 6.243 Definicja metody Przeczytaj

Tworzymy kolejny **strumień** tym razem otwierający plik do odczytu. W **atrybucie** metody **openFileInput** podajemy nazwę pliku, którego zawartość chcemy odczytać. Ponieważ będziemy odczytywać informację, którą zapisaliśmy w pliku **plik1.txt** podajemy jego nazwę.

Klasa **BufferedReader** ma za zadanie odczytać zawartość pliku. Będzie w tym celu korzystać ze specjalnego bufora w którym będzie przechowywać pobrane w danej chwili dane. Obiekt klasy **StringBuilder** będzie z pobranych danych tworzył łańcuch znaków. Pobrane dane mają postać binarną, a klasa ma za zadanie przywrócić dane do postaci napisu.

Pętla **while** będzie wykonywać swe działanie do momentu, aż nie pobierze pustej linijki. Nastąpi to wówczas, gdy dojdziemy do końca wczytywanego pliku. Po napotkaniu znaku przejścia do nowej linii program zacznie wypisywać pobrany tekst od nowego wiersza.

Zamykamy **strumień** metodą **close,** a pobrany tekst umieszczamy w polu **EditText**. W tej klasie również mamy sekcje **catch**, która uruchomi swoje działanie w razie kłopotów z pobraniem danych i wówczas wyśle błąd. Działanie aplikacji po uruchomieniu będzie wyglądało jak na poniższym rysunku:



Rysunek 6.142 Aplikacja pobierająca i wyświetlająca dane z i do pliku

Pełny kod pliku **MainActivity.java** wygląda w następujący sposób:

```
package przyklad.nr41a;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

import java.io.BufferedReader;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;

public class MainActivity extends AppCompatActivity {
    EditText editText;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        editText = findViewById(R.id.tekst);
    }
```

```
public void Przeczytaj(View view){
        try
        {
 FileInputStream file2 = openFileInput("plik2.txt");
 InputStreamReader stream = new
InputStreamReader(file2);
 BufferedReader reader = new BufferedReader(stream);
 StringBuilder tekst = new StringBuilder();
   String linia;
  while ((linia = reader.readLine()) != null) {
                tekst.append(linia).append("\n");
 }
 reader.close();
 editText.setText(tekst.toString());
  }
   catch (IOException ex) {
     ex.printStackTrace();
   }
    }

public void Zapisz(View view){
String tekst = editText.getText().toString();
        try {
FileOutputStream file = openFileOutput("plik2.txt",
MODE_PRIVATE);
OutputStreamWriter stream = new
OutputStreamWriter(file);
            stream.write(tekst);
            stream.close();
            editText.setText("");
  }
        catch (IOException wyjatek){
            wyjatek.printStackTrace();
  }
 }
}
```

Listing 6.244 Plik MainActivity.java

6.22 Bazy danych SQLite

6.22.1 Tworzenie prostej bazy danych

Pobieranie i zapisywanie danych z wykorzystaniem plików tekstowych jest prostą operacją. Jednak w aplikacjach lepszym rozwiązaniem jest przechowywanie informacji w bazach danych. Idealnym rozwiązaniem w aplikacjach mobilnych jest skorzystanie z języka **SQLite**. Jest to w pewnym

sensie uproszczona wersja popularnego **MySQL** i obsługuje się ją podobnymi poleceniami. Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja42**.

W ćwiczeniu wykorzystamy prostą bazę danych służącą do przechowywania informacji o uczniach. W relacyjnych bazach danych do porządkowania informacji wykorzystuje się tabele. Zanim zaczniemy programować należy przygotować tabelę z danymi, które chcemy w bazie umieścić. Jak wspomniano na początku baza będzie przechowywać informacje o uczniach. Tabela, która będziemy się posługiwać będzie wyglądała następująco:

Id	Imię	Nazwisko	Klasa
1	Jan	Kowalski	1PT
2	Ewa	Nowak	2PT
3	Stanisław	Malinowski	3PT

Tabela 6.4 Tabela danych użytych w bazie danych

Tworzenie aplikacji rozpoczniemy od przygotowania pliku układu. Będzie on zawierał pole **TextView** za pomocą, którego będziemy wypisać zawartość bazy.

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

    <TextView
        android:id="@+id/tekst1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/tekst1"
        android:textSize="15pt"
        android:textColor="@color/black"
        />
</LinearLayout>
```

Listing 6.245 Plik układu main_activity.xml

Kolejnym krokiem jest stworzenie klasy o nazwie Asystent.java. Klasa będzie odpowiedzialna za obsługę bazy.

```
public class Asystent extends SQLiteOpenHelper
```

Listing 6.246 Deklaracja klasy Asystent dziedziczącej po klasie SQLiteOpenHelper

Klasa **Asystent** będzie dziedziczyć po klasie **SQLiteOpenHelper**. Klasa ta zajmuje się obsługą i edycją baz danych. Do działania potrzebuje **konstruktora**, którego tworzymy w następujący sposób:

```
public Asystent(Context context){
    super(context, "szkola.db", null, 1);
}
```

Listing 6.247 Konstruktor klasy Asystent

Za pomocą konstruktora przekażemy cztery parametry: context, nazwę bazy danych, informacje o tzw. kursorze, oraz numer wersji bazy danych.

Obsługa **bazy danych** wymaga jeszcze **dwóch klas**, które zadbają o jej prawidłowe działanie. Pierwsza z nich **onCreate** w której będziemy tworzyć tabelę. W naszej aplikacji klasa **onCreate** będzie wyglądać następująco:

```
@Override
public void onCreate(SQLiteDatabase bd1) {
    bd1.execSQL("CREATE TABLE UCZEN (id INTEGER PRIMARY
KEY AUTOINCREMENT, "+ "IMIE text," + "NAZWISKO text, " +
"KLASA text);");
}
```

Listing 6.248 Tworzenie tabeli UCZEN

Argumentem klasy **onCreate** będzie nazwa bazy danych. Metoda **execSQL** pozwala na wykonanie instrukcji języka SQL. Tworzymy tabelę o nazwie uczeń, która zawiera cztery kolumny. Zgodnie z tabelką, którą przygotowaliśmy wcześniej. Zauważyć można, że polecenie tworzące tabelę wygląda bardzo podobnie do tego w języku MySQL.

Tworzymy cztery kolumny:

- Id;
- Imie;
- Nazwisko;
- Klasa;

Kolumna **id** będzie typu całkowitego int. Dodatkowo kolumna będzie tzw. kluczem głównym, o czym mówi nam **PRIMARY KEY**. Funkcja

Autoincrement spowoduje, że każdy nowy rekord w tabeli będzie miał nadany kolejny numer począwszy od **1**.

Pozostałe kolumny są typu text co pozwala na wprowadzenie danych w formie tekstu. Kolejną klasą, którą musimy utworzyć jest klasa **onUpgrate**. Będzie ona pomocna w momencie, kiedy będziemy zamieniać jedną bazę danych na drugą np. po edycji danych. Klasa zawiera trzy argumenty. Pierwszy z nich to **nazwa bazy**. Drugi jest **numerem nowej wersji bazy danych**, a trzeci **numerem obecnej**.

```
@Override
public void onUpgrade(SQLiteDatabase bd1, int nowa, int
stara) {

}
```

Listing 6.249 Deklaracja metody onUpgrade

Na tym etapie wszystkie niezbędne składniki **Asystenta** są gotowe. Pozostałe klasy są tworzone z myślą o konkretnej bazie danych i nie zawsze muszą występować. Baza danych **szkola** będzie potrzebowała jeszcze kilku dodatkowych **klas** do jej obsługi. Pierwszą z nich będzie **dodaj**. Jej zadaniem będzie dodawanie rekordów do bazy danych.

```
public void dodaj(String imie, String nazwisko, String
klasa)
{
    SQLiteDatabase bd1 = getWritableDatabase();
    ContentValues dane = new ContentValues();
    dane.put("imie",imie);
    dane.put("nazwisko",nazwisko);
    dane.put("klasa", klasa);
    bd1.insertOrThrow("UCZEN",null,dane);
}
```

Listing 6.250 Definicja metody dodaj

Klasa ma trzy argumenty typu **String**, gdyż będziemy przekazywać do niej rekordy będące **tekstem**. **Argumenty** te to: **imię**, **nazwisko**, **klasa**. Są one **nazwami kolumn** budujących tabelę bazy danych. Użyta metoda **getWritabledatabase** pozwoli na otwarcie bazy i daje możliwość jej edycji. Metodę przypisujemy obiektowi **bd1** – czyli naszej bazie danych.

Obiekt klasy **ContentValues** będzie przechowywać przekazane wartości. Metoda **put** ma za zadanie przekazać do bazy **klucz** – wartość w cudzysłowie –

i **nazwę kolumny**. (Podobnie jak przy przekazywaniu intencji). Ostatnia metoda **insertOrThrow** przekaże informacje do tabeli **Uczen**.

Do wypisywania zawartości bazy danych będziemy potrzebować klasy **wypiszCalosc**. Będzie to klasa oparta o **obiekt** klasy **Cursor**. Kursor to rodzaj wskaźnika który wskazuje wybrane fragmenty bazy danych. Ma on również możliwość poruszania się po elementach bazy.

```
public Cursor wypiszCalosc()
{
SQLiteDatabase bd1 = getReadableDatabase();
Cursor kursor = bd1.rawQuery("SELECT * from UCZEN ",
null);
   return kursor;
    }
```

Listing 6.251 Definicja metody wypiszCalosc

Tworzymy obiekt **SQLiteDatabase** i za pomocą **getReadableDatabase** otwieramy bazę do odczytu. Metoda **getReadableDataBases** różni się od **getWritableDatabase** tym, że umożliwia tylko odczyt danych bez możliwości edycji. Następnie tworzymy obiekt klasy **Cursor** i za pomocą metody **rawQuery** tworzymy zapytanie do bazy. Jest to zwyczajne zapytanie **MySQL**, które wypisze wszystkie dane z bazy **UCZEN**. Kursor pobierze i przechowa pobrane dane.

Na końcu zwracamy obiekt kursor, który przechowuje informację o zapytaniu. Klasa **Asystent** jest już gotowa do działania. Cały jej kod prezentuje się następująco:

```
package przyklad.nr42;

import android.content.ContentValues;
import android.content.Context;
import android.database.Cursor;
import android.database.sqlite.SQLiteDatabase;
import android.database.sqlite.SQLiteOpenHelper;
public class Asystent extends SQLiteOpenHelper {
 public Asystent(Context context){
  super(context, "szkola.db", null, 1);
}
 @Override
public void onCreate(SQLiteDatabase bd1) {
 bd1.execSQL("CREATE TABLE UCZEN (id INTEGER PRIMARY KEY
```

```
AUTOINCREMENT, "+ "IMIE text," + "NAZWISKO text, " +
"KLASA text);");
 }

    @Override
public void onUpgrade(SQLiteDatabase bd1, int nowa, int
stara) {

    }
    public void dodaj(String imie, String nazwisko,
String klasa)
    {
  SQLiteDatabase bd1 = getWritableDatabase();
  ContentValues dane = new ContentValues();
        dane.put("imie",imie);
        dane.put("nazwisko",nazwisko);
        dane.put("klasa", klasa);
        bd1.insertOrThrow("UCZEN",null,dane);
    }
    public Cursor wypiszCalosc()
    {
    SQLiteDatabase bd1 = getReadableDatabase();
   Cursor kursor = bd1.rawQuery("SELECT * from UCZEN ",
null);
        return kursor;
 }
}
```

Listing 6.252 Kod klasy Asystent

Czas na edycję **MainActivity.java**. Rozpoczniemy od utworzenia dwóch obiektów. Pierwszy o nazwie **asystent** będzie obiektem klasy **Asystent** i posłuży do jej obsługi. Drugi obiekt klasy **Cursor** o nazwie **k**.

```
Asystent asystent = new Asystent(this);
Cursor k;
```

Listing 6.253 Utworzenie obiektu klasy Asystent

Tworzymy prostą klasę, która wypełni tabelę bazy danych.

```
public void dodajDane()
{
    asystent.dodaj("Jan","Kowalski","1PT");
    asystent.dodaj("Ewa","Nowak","2PT");
    asystent.dodaj("Stanisław","Malinowski","3PT");
}
```

Listing 6.254 Definicja metody dodajDane

Klasa ta będzie dodawać do bazy kolejne rekordy.

Posługuje się ona obiektem asystent i klasą dodaj, którą zdefiniowaliśmy wcześniej. Zauważyć można, że w każdy z rekordów wpisujemy tylko trzy pozycje. Są to: **imię**, **nazwisko** i **klasa**. Zawartość kolumny **ID** jest dodawana automatycznie, gdyż są to pola zdefiniowane jako AUTOINCREMENT. Pola tej kolumny będą się wypełniać kolejnymi wartościami liczbowymi. Przechodzimy do klasy **onCreate** w której pobieramy **referencję** do pola **TextView** za pomocą, którego będziemy wypisywać dane z bazy:

```
TextView tekst = findViewById(R.id.tekst1);
```

Listing 6.255 Referencja do widoku TextView

Uruchamiamy klasę **dodajDane**:

```
dodajDane();
```

Listing 6.256 Wywołanie klasy dodajDane

A także wypiszCalosc:

```
k = asystent.wypiszCalosc();
```

Listing 6.257 Wywołanie metody wypiszCalosc

Jak powiedzieliśmy wcześniej, **kursor** jest pewnym rodzajem wskaźnika. W momencie pobrania danych za pomocą klasy **wypiszCalosc** kursor ustawił się na pierwszym pobranym elemencie. Aby klasa wypisująca mogła wypisać wszystkie potrzebne dane trzeba odpowiednio przesunąć kursor. Wykorzystamy w tym celu prostą pętle **while:**

```
while(k.moveToNext())
{
    int id=k.getInt(0);
    String imie = k.getString(1);
    String nazwisko = k.getString(2);
    String klasa = k.getString(3);
    tekst.setText(tekst.getText()+"\n " + id + ". " +
imie + " "+ nazwisko +  " " +klasa );
}
```

Listing 6.258 Pętla odczytująca i wypisująca wszystkie rekordy z bazy danych

Warunek pętli uzależniony jest od **metody** obsługującej **kursor** - **moveToNext**. **Pętla** będzie działać do momentu, kiedy nie napotka na ostatni element. Metoda **moveToNext** ma za zadanie przesunąć kursor - jak sama nazwa wskazuje - do następnego elementu. Jeżeli kursor nie będzie mógł się dalej przesunąć znaczy to, że dotarł do ostatniego elementu i pętla zakończy działanie.

Istnieją cztery metody do obsługi kursorów. Poza wspomnianą **moveToNext** możemy wyróżnić również:

- **moveToFirst()** – przesunie kursor na pierwszy element;
- **moveToLast()** – przesunie kursor na ostatni element;
- **moveToPrevious()** – przesunie kursor o jeden element do tyłu – do elementu poprzedniego;

Po uruchomieniu pętli powstaje pewnego rodzaju **tablica danych**. Kursor będzie przesuwał się pomiędzy elementami i pobierał dane, które następnie umieści w przygotowanych zmiennych. W ostatniej linijce tworzymy napis za pomocą metody **setText** i umieszczamy go w widoku **TextView**. Bazę danych należy jeszcze zamknąć i tutaj posłużymy się kodem:

```
k.close();
```

Listing 6.259 Zamknięcie kursora

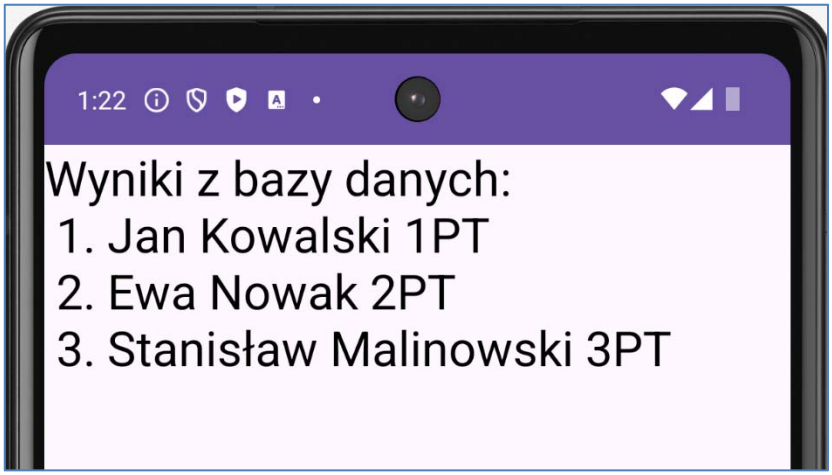
Całość aktywności **MainActivity.java** będzie wyglądać następująco:

```
package przyklad.nr42;

import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.database.Cursor;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    Asystent asystent = new Asystent(this);
    Cursor k;
    public void dodajDane(){
asystent.dodaj("Jan","Kowalski","1PT");
asystent.dodaj("Ewa","Nowak","2PT");
asystent.dodaj("Stanisław","Malinowski","3PT");
}
  @Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
 TextView tekst = findViewById(R.id.tekst1);

 dodajDane();
 k = asystent.wypiszCalosc();

 while(k.moveToNext()){
int id=k.getInt(0);
String imie = k.getString(1);
```

```
String nazwisko = k.getString(2);
String klasa = k.getString(3);
tekst.setText(tekst.getText()+"\n " + id + ". " + imie +
" "+ nazwisko +  " " +klasa );
  }
      k.close();
 }
}
```

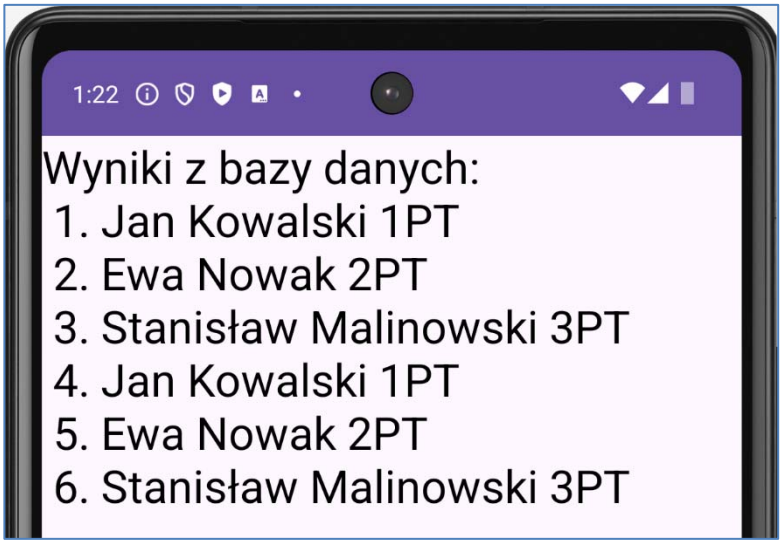
Listing 6.260 Plik aktywności MainActivity.java

Po uruchomieniu aplikacji widzimy zawartość bazy danych:



Rysunek 6.143 Wypisane informacje zawarte w bazie danych

Kłopot pojawi się, gdy ponownie uruchomimy aplikację. Dane do bazy dodawane są za każdym razem, gdy ją uruchamiamy. W wyniku ponownego otwarcia aplikacji otrzymujemy następujące dane z bazy.



Rysunek 6.144 Wypisane informacje zawarte w bazie danych po ponownym uruchomieniu aplikacji

Wyobraźmy sobie teraz sytuację, gdy aplikację uruchamiamy 100, 1000 lub więcej razy. Dlatego też trzeba zadbać o to, żeby aplikacja umieszczała dane w bazie tylko, wtedy, gdy jest ona pusta. Program policzy ilość wpisanych rekordów i dopisze dane tylko wtedy, gdy baza jest pusta. W klasie **Asystent** tworzymy kolejną metodę:

```
public int policz()
{
    SQLiteDatabase bd1 = getReadableDatabase();
    String licznik = "SELECT count(*) FROM UCZEN";
    Cursor kursor2 = bd1.rawQuery(licznik, null);
    cursor.moveToFirst();
    int ilosc = kursor2.getInt(0);
    return ilosc;
}
```

Listing 6.261 Definicja metody policz

Otwieramy bazę do odczytu. Tworzymy zmienną **licznik**, która skieruje odpowiednie **zapytanie** do bazy. **Zapytanie** ma policzyć ilość wprowadzonych do bazy **rekordów**. Następnie przekazać **wynik** do **kursora**, a ten do zmiennej **ilość**. Zmienna **ilość** wykorzystamy w aktywności głównej. Modyfikujemy nieco funkcję **dodajDane**:

```
public void dodajDane()
{
 int ilosc2 = asystent.policz();
   if (ilosc2 == 0) {
   asystent.dodaj("Jan", "Kowalski", "1PT");
   asystent.dodaj("Ewa", "Nowak", "2PT");
   asystent.dodaj("Stanisław", "Malinowski", "3PT");
    }
}
```

Listing 6.262 Definicja metody dodajDane

Pobieramy dane przekazane z funkcji **policz** i przypisujemy je zmiennej **ilosc2**. Nazwa zmiennej celowo została zmieniona, żeby pokazać, że jest to zupełnie inna zmienna niż w klasie **Asystent**. Przed wprowadzeniem danych do bazy **instrukcja warunkowa** sprawdza czy w bazie znajdują się już dane. Jeżeli w bazie znajdują się już wprowadzone dane to funkcja nie zadziała. Po kilkukrotnym uruchomieniu aplikacji baza danych wypisze te same dane, co za pierwszym razem.

6.22.2 Dodawanie danych za pomocą formularza.

Bazy danych są praktycznym narzędziem do przechowywania i przetwarzania danych w aplikacji, Zwłaszcza takiej w której gromadzimy spore ilości danych. Do tej pory korzystaliśmy z bazy danych, która uruchamiała się wraz z aplikacją. Jednak baza danych powinna dawać również możliwość edycji i dodawania danych według potrzeb użytkownika.

W następnym ćwiczeniu wykorzystamy bazę danych Uczen. Dane do niej będziemy jednak dodawać za pomocą prostego formularza. Kod aplikacji znajduje się w katalogu **Aplikacja42a**.

W pliku układu nowej aplikacji tworzymy formularz składający się z trzech pól **EditText**. Za jego pomocą będziemy dodawać do bazy danych informacje o uczniu: **Imie**, **Nazwisko**, **Klasę**. Do zatwierdzenia wprowadzonych danych będzie nam potrzebny przycisk **Button**. Kod układu będzie wyglądał następująco:

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

    <EditText
        android:id="@+id/imie"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint1"
        android:inputType="textCapSentences" />
  <EditText
        android:id="@+id/nazwisko"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint2"
        android:inputType="textCapSentences"
      />
  <EditText
      android:id="@+id/klasa"
      android:layout_width="match_parent"
      android:layout_height="wrap_content"
      android:hint="@string/hint3"
      android:inputType="textCapSentences"
      />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/przycisk"
        android:onClick="dodaj"
        />
</LinearLayout>
```

Listing 6.263 Kod układu pobierającego dane do bazy

W bazie przyda nam się klasa **Asystent**. Tworzymy ją dokładnie tak samo jak w poprzednim przykładzie. W związku z tym, że dane będą dodawane do bazy z zewnątrz można w niej pominąć metodę **policz**. Teraz w pliku **Aktywności** głównej tworzymy klasę, która pobierze dane z pól **EditText**, a następnie wyśle je do bazy. Będzie to klasa **dodaj**, która jest podpięta pod przycisk i uruchomi się, gdy użytkownik wciśnie przycisk. Kod klasy będzie prezentował się jak na następnym listingu. Na początek tworzymy **referencję** do pól **EditText**: **imie**, **nazwisko** i **klasa**.

```
public void dodaj(View view) {
    EditText im = findViewById(R.id.imie);
    EditText naz = findViewById(R.id.nazwisko);
    EditText kl = findViewById(R.id.klasa);
```

Listing 6.264 Definicja metody dodaj

Za pomocą metody **getText** pobieramy tekst wpisany przez użytkownika w odpowiednim polu. Metoda **toString** zamieni wpisane dane na napis **String**. Pobrane napisy przypisujemy odpowiednim zmiennym.

```
String im2 = im.getText().toString();
String naz2 = naz.getText().toString();
String kl2 = kl.getText().toString();
```

Listing 6.265 Tworzenie zmiennych łańcuchowych

Informacje do bazy przekażemy za pomocą **intencji**. Tworzymy zatem obiekt klasy **Intent** i wysyłamy dane do nowej **aktywności**, którą nazwiemy **DrugaAktywnosc**.

```
Intent nowaIntencja = new Intent(this,
DrugaAktywnosc.class);
```

Listing 6.266 Tworzenie intencji

Metoda **putExtra** użyta trzykrotnie – do każdego z napisów osobno – wyśle dane do drugiej aktywności. Ostatnią czynnością będzie uruchomienie intencji.

```
nowaIntencja.putExtra("imie", im2);
nowaIntencja.putExtra("nazwisko", naz2);
nowaIntencja.putExtra("klasa", kl2);
startActivity(nowaIntencja);
```

Listing 6.267 Dane przekazywane przez intencje

Do odebrania informacji z intencji utworzymy wspomnianą wcześniej klasę: **DrugaAktywnosc.java**. Plik układu drugiej aktywności o nazwie **activity_druga_aktywnosc.xml** będzie zawierał jeden widok **TextView** za pomocą, którego będziemy chcieli wyświetlić zawartość bazy danych. W klasie **DrugaAktywnosc** odbierzemy dane z intencji:

```
Bundle bundle = getIntent().getExtras();
```

Listing 6.268 Tworzenie obiektu bundle

Pobieramy z intencji przekazane napisy:

```
TextView tekst = findViewById(R.id.tekst1);
String imie = bundle.getString("imie");
String nazwisko = bundle.getString("nazwisko");
String klasa = bundle.getString("klasa");
pm.dodaj(imie, nazwisko, klasa);
```

Listing 6.269 Pobieranie danych z intencji

Za pomocą metody **dodaj**, umieszczamy informacje w bazie danych, a następnie wyświetlamy zawartość całej bazy danych na ekranie. Kod klasy **DrugaAktywnosc** wygląda następująco:

```
package przyklad.nr42a;


import androidx.appcompat.app.AppCompatActivity;

import android.database.Cursor;
import android.os.Bundle;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;
public class DrugaAktywnosc extends AppCompatActivity {
    Asystent pm = new Asystent(this);
    @Override
    protected void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_druga_aktywnosc);
 Bundle bundle = getIntent().getExtras();
TextView tekst = findViewById(R.id.tekst1);
String imie = bundle.getString("imie");
String nazwisko = bundle.getString("nazwisko");
String klasa = bundle.getString("klasa");
   pm.dodaj(imie, nazwisko, klasa);
        Cursor k = pm.wypiszCalosc();

 while (k.moveToNext()) {
   int id = k.getInt(0);
   String imie2 = k.getString(1);
   String nazwisko2 = k.getString(2);
   String klasa2 = k.getString(3);
tekst.setText(tekst.getText() + "\n" + id + ". " + imie2
+ " " + nazwisko2 + " " + klasa2);
  }
 }
}
```

Listing 6.270 Kod drugiej aktywności

6.22.3 Rozdzielenie operacji dodawania i wypisywania danych do bazy.

Każdorazowe wypisywanie danych z bazy jest czasochłonne i często niepotrzebne. Dlatego też warto te dwie operacje rozdzielić i stworzyć osobno klasę do wprowadzania danych i osobno do wypisywania. Zrobimy to w następnym ćwiczeniu. Kod aplikacji znajduje się w katalogu o nazwie **Aplikacja42b**.

Tworzymy nową aplikację, która na początek będzie składała się z klas: **Asystent** i **MainActivity**, oraz pliku układu **activity_main.xlm**. W pliku układu **activity_main.xml** umieścimy pole **TextView** z napisem zachęcającym do skorzystania z bazy danych, oraz dwóch przycisków **Button** – „**Wprowadź dane**”, oraz „**Wypisz Dane**”. Kod pliku **activity_main.xml**:

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

   <TextView
     android:id="@+id/nr_domu"
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="- - - - - Witaj w bazie danych - - - - -\n
     - - - - Zdecyduj co chcesz zrobic - - -"
     android:textColor="@color/darkblue"
     android:layout_gravity="center_horizontal"
     android:textSize="25sp"/>

   <Button
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/button1"
     android:layout_marginTop="10dp"
     android:textSize="25sp"
     android:layout_gravity="center_horizontal"
     android:onClick="wpisz"/>
```

```
    <Button
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/button2"
      android:textSize="25sp"
      android:layout_gravity="center_horizontal"
      android:onClick="wypisz" />

</LinearLayout>
```

Listing 6.271 Kod układu activity_main.xml

Do przycisków **Button** zostały podłączone dwie metody „**wpisz**" i „**wypisz**". Metody te zostaną zdefiniowane w pliku głównej aktywności **ActivityMain.java**. Kod aktywności wygląda następująco:

```
package przyklad.nr42b;

import androidx.appcompat.app.AppCompatActivity;

import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class MainActivity extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
    public void wpisz(View view) {

startActivity(new Intent(this, WprowadzDane.class));
    }
    public void wypisz(View view) {
startActivity(new Intent(this, WypiszDane.class));
    }
}
```

Listing 6.272 Kod aktywności głównej

Zarówna metoda "**wpisz**" jak i „**wypisz**" uruchamiają intencję do odpowiedniej klasy. Pierwsza z nich posłuży do wpisania danych do bazy, druga do ich wypisania na ekranie aplikacji. Kliknięcie przycisku **Wprowadź dane** spowoduje uruchomienie formularza służącego do wpisania danych. Uruchomiona zostanie zatem klasa **WprowadźDane**. Będzie ona przypominała klasę z poprzedniego ćwiczenia. Pominiemy w niej tylko automatyczne wypisywanie danych. Plik układu połączonego z aktywnością będzie zawierał trzy pola **EditText** – służące do wpisania danych, oraz przycisk. Pełny kod tego układu będzie wyglądać następująco:

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

    <EditText
        android:id="@+id/imie"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint1"
        android:inputType="textCapSentences" />
    <EditText
        android:id="@+id/nazwisko"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint2"
        android:inputType="textCapSentences"
        />
    <EditText
        android:id="@+id/klasa"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:hint="@string/hint3"
        android:inputType="textCapSentences"
        />
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:textSize="25sp"
     android:layout_gravity="center_horizontal"
        android:text="@string/button1"
        android:onClick="dodaj"
        />
</LinearLayout>
```

Listing 6.273 Kod formularza dodającego dane do bazy

Metoda **dodaj** spowoduje umieszczenia danych w bazie. Pełny kod aktywności **WprowadzDane.java**:

```
package przyklad.nr42b;

import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.EditText;

public class WprowadzDane extends AppCompatActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
setContentView(R.layout.activity_wprowadz_dane);
 }
public void dodaj(View view) {
EditText im = findViewById(R.id.imie);
EditText naz = findViewById(R.id.nazwisko);
EditText kl = findViewById(R.id.klasa);
String im2 = im.getText().toString();
String naz2 = naz.getText().toString();
String kl2 = kl.getText().toString();
Intent nowaIntencja = new Intent(this,
DodajDoBazy.class);
   nowaIntencja.putExtra("imie", im2);
   nowaIntencja.putExtra("nazwisko", naz2);
   nowaIntencja.putExtra("klasa", kl2);
   startActivity(nowaIntencja);
 }
}
```

Listing 6.274 Kod aktywności WprowadźDane

Kliknięcie przycisku **WprowadźDane** po uzupełnieniu formularza uruchomi **intencje** przekazującą pobrane z **formularza** dane kolejnej **aktywności**. W pliku aktywności **DodajDoBazy** odebrane poprzez formularz z klasy **WprowadzDane** informacje zostają umieszczone w bazie. Działa to dokładnie tak samo jak w poprzednim ćwiczeniu. Kod aktywności **DodajDoBazy.java** przedstawia poniższy listing:

```
package przyklad.nr42b;

import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;

public class DodajDoBazy extends AppCompatActivity {
    Asystent pm = new Asystent(this);
    @Override
    protected void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
setContentView(R.layout.activity_dodaj_do_bazy);
Bundle bundle = getIntent().getExtras();
String imie = bundle.getString("imie");
String nazwisko = bundle.getString("nazwisko");
String klasa = bundle.getString("klasa");
    pm.dodaj(imie, nazwisko, klasa);
 }
public void wypisz(View view) {
startActivity(new Intent(this, WypiszDane.class));
  }
public void wpisz(View view) {
startActivity(new Intent(this, WprowadzDane.class));
    }
}
```

Listing 6.275 Aktywność DodajDoBazy.java

W klasie zostały dwa przycisku **Button**, których działanie jest identyczne do tych w pliku aktywności głównej – **ActivityMain.java**. Pierwszy z nich umożliwi wypisanie danych z bazy, a drugi ponownie otworzy formularz dodawania do bazy z pliku **WprowadzDane.java**. Plik układu **activity_dodaj_do_bazy.xml** ma następująco postać:

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

    <TextView
     android:id="@+id/tekst1"
     android:layout_width="match_parent"
     android:layout_height="wrap_content"
     android:textSize="25sp"
```

```
     android:layout_gravity="center_horizontal"
     android:text="@string/tekst1"/>

    <Button
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:layout_marginTop="10dp"
     android:textSize="25sp"
     android:layout_gravity="center_horizontal"
     android:text="@string/button2"
     android:onClick="wypisz" />

    <Button
     android:layout_width="wrap_content"
     android:layout_height="wrap_content"
     android:text="@string/button1"
     android:layout_marginTop="10dp"
     android:textSize="25sp"
     android:layout_gravity="center_horizontal"
     android:onClick="wpisz" />

</LinearLayout>
```

Listing 6.276 Kod układu activity_dodaj_do_bazy.xml

Do omówienia zostaje ostatnia **klasa**. Uruchamia się ona wówczas, gdy chcemy wypisać dane zawarte w bazie. Kod aktywności **WypiszDane.java** wygląda kod następująco:

```
package przyklad.nr42b;

import androidx.appcompat.app.AppCompatActivity;
import android.content.Intent;
import android.database.Cursor;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;

public class WypiszDane extends AppCompatActivity {
    Asystent pm = new Asystent(this);
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_wypisz_dane);
 TextView tekst = findViewById(R.id.tekst1);
    Cursor k = pm.wypiszCalosc();

 while (k.moveToNext()) {
  int id = k.getInt(0);
  String imie2 = k.getString(1);
```

```
   String nazwisko2 = k.getString(2);
   String klasa2 = k.getString(3);
  tekst.setText(tekst.getText() + "\n" + id + ". " +
imie2 + " " + nazwisko2 + " " + klasa2);
   }
  }
     public void wpisz(View view) {
startActivity(new Intent(this, WprowadzDane.class));
     }
}
```

Listing 6.277 Kod aktywności - WypiszDane

W **klasie** mamy dokładnie tę samą **pętle** co w poprzednim ćwiczeniu. Ma ona za zadanie pobrać dane z bazy – linijka po linijce. Wykorzystuje w tym celu **kursor**. Następnie pobrane informacje są przetworzone w napis za pomocą **setText**. Po wypisaniu danych pojawiają się również przycisk za pomocą którego możemy po raz kolejny przenieść się do formularza i wpisać kolejną osobę do bazy danych. Do prawidłowego działania aplikacji będzie nam potrzebny również plik **strings.xml**:

```
<resources>
<string name="app_name">Aplikacja47</string>
<string name="button1">Wprowadź dane</string>
<string name="button2">Wypisz dane</string>
<string name="hint1">Wpisz imię</string>
<string name="hint2">Wpisz nazwisko</string>
<string name="hint3">Wpisz klasę</string>
<string name="tekst1">Dane zostały dodane do bazy
</string>
<string name="button3">Wprowadź kolejną osobę</string>
</resources>
```

Listing 6.278 Plik strings.xml

Omówiony przykład jest bardzo uproszczony. Został zaprezentowany po to, aby przyjrzeć się mechanizmowi działania bazy danej. Poszczególne klasy i działania bazy można byłoby przenieść do prostego menu – np. **szuflady nawigacyjnej**. Baza byłaby bardziej funkcjonalna i na pewno wyglądała bardziej profesjonalnie. Do bazy można również dodać formularze przeszukujące dane i filtrujące informacje.

6.23 Wyświetlanie aplikacji na urządzeniach o różnej wielkości ekranu

Aplikacje mobilne są obecnie bardzo popularne. W swoich smartfonach mamy ich dziesiątki. Jednak telefony nie są jednymi urządzeniami, które dają możliwość ich instalacji.

Aplikacje można instalować i uruchamiać również na tabletach. Jak doskonale wiemy, są to urządzenia o wiele większych ekranach niż telefony komórkowe. Dzięki czemu dają lepsze możliwości rozłożenia widoków.

Aplikacje mobilne można tworzyć jednocześnie na mniejsze jak i na większe ekrany. Kolejna aplikacja pokaże w jaki sposób zrobić to najprościej. Kod aplikacji i potrzebne do jej stworzenia zasoby znajdują się w folderze o nawie **Aplikacja43**.

Zacznijmy od instalacji odpowiedniego emulatora. Poza dostępnym już smartfonem musimy zainstalować emulator symulujący działanie tabletu. Instalacja nowego wirtualnego urządzenia została omówiona w poprzednich rozdziałach.

Gdy już będziemy dysponowali odpowiednim emulatorem możemy zacząć testować nasze aplikacje na urządzeniu z większym wyświetlaczem. Zaczniemy od stworzenia prostego układu w którym umieścimy kilka widoków.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/
android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/napis"
        android:textSize="30sp"
        />
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="300dp"
        android:scaleType="center"
        android:id="@+id/obrazek"
        android:src="@drawable/obrazek"/>
    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="300dp"
        android:scaleType="center"
        android:id="@+id/obrazek2"
        android:src="@drawable/obrazek2"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/przycisk"
        />
</LinearLayout>
```

Listing 6.279 Kod układu activity_main.xml

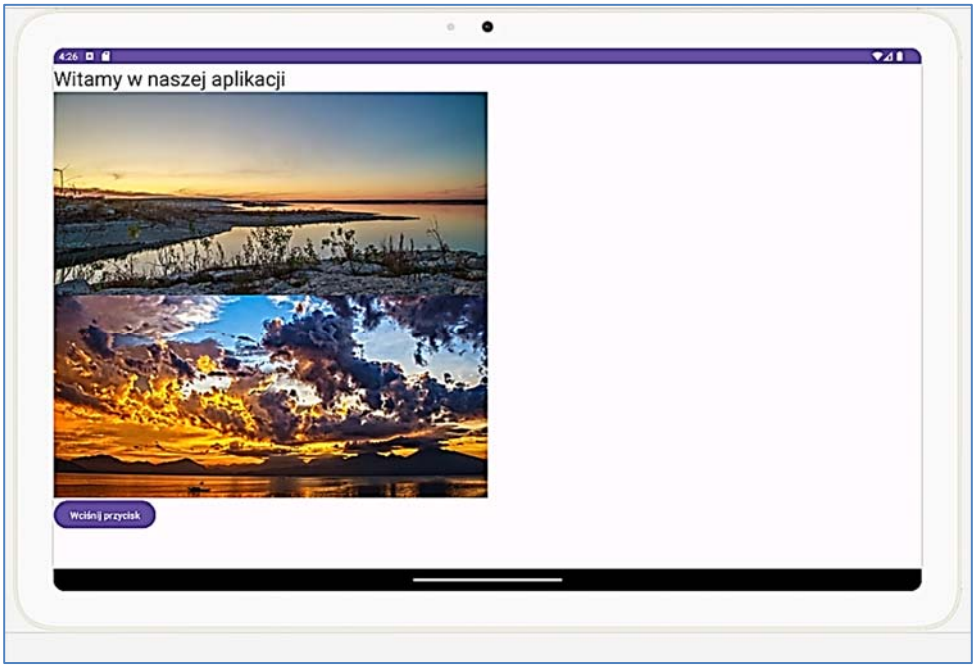
W układzie tworzymy:

- **TextView** – w którym umieszczamy napis z czcionką wielości 30sp.
- **Dwa widżety ImageView** – w których szerokość jest ustawiona na wrap_content, a wysokość ma wartość 300dp.
- Prosty przycisk – Button.

Po uruchomieniu aplikacji na Smartfonie wygląda ona następująco:

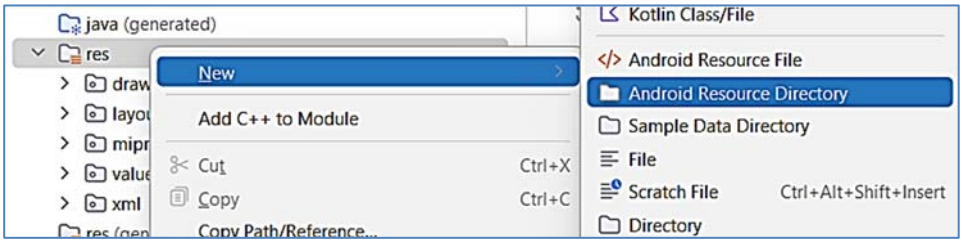
Rysunek 6.145 Układ widocznym na urządzeniu typu Smartfon

Gdy uruchomimy aplikację na większym urządzeniu będzie ona wyglądała w ten sposób, jaki ilustruje poniższy rysunek.

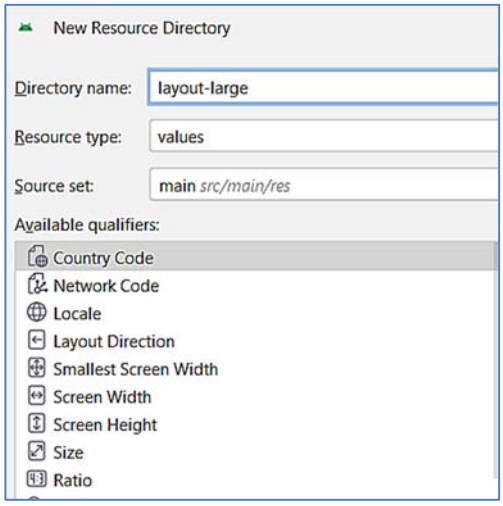


Rysunek 6.146 Układ widoczny na urządzeniu o większych rozmiarach: Tablet

Na dużym urządzeniu aplikacja wygląda w zasadzie tak samo jak na mniejszym. Elementy ułożone są według takiego samego układu i nie wyglądają dobrze. W aplikacji jest mnóstwo wolnego miejsca, które możemy wykorzystać lepiej. Aby utworzyć inny układ dla większych ekranów należy na nowo zdefiniować plik **main_activity.xml**. Zanim jednak przystąpimy do pracy trzeba do aplikacji dodać dodatkowy plik układu przeznaczony na duże urządzenia. W zasobach aplikacji tworzymy dodatkowy katalog i nazywamy go **layout-large**.

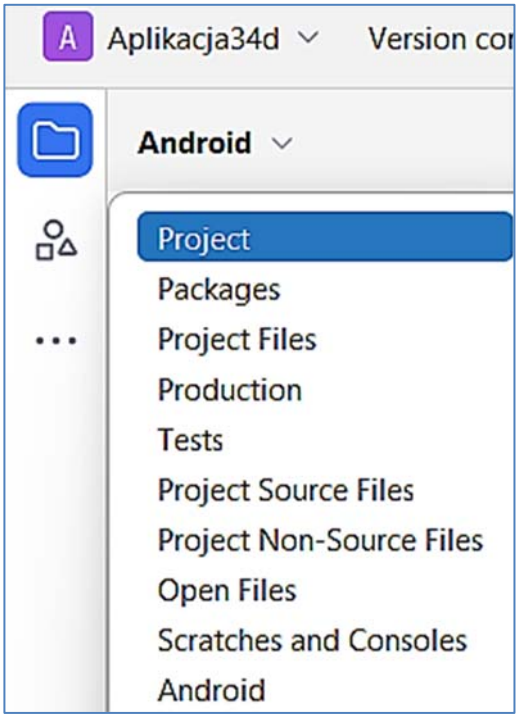


Rysunek 6.147 Tworzenie nowego katalogu



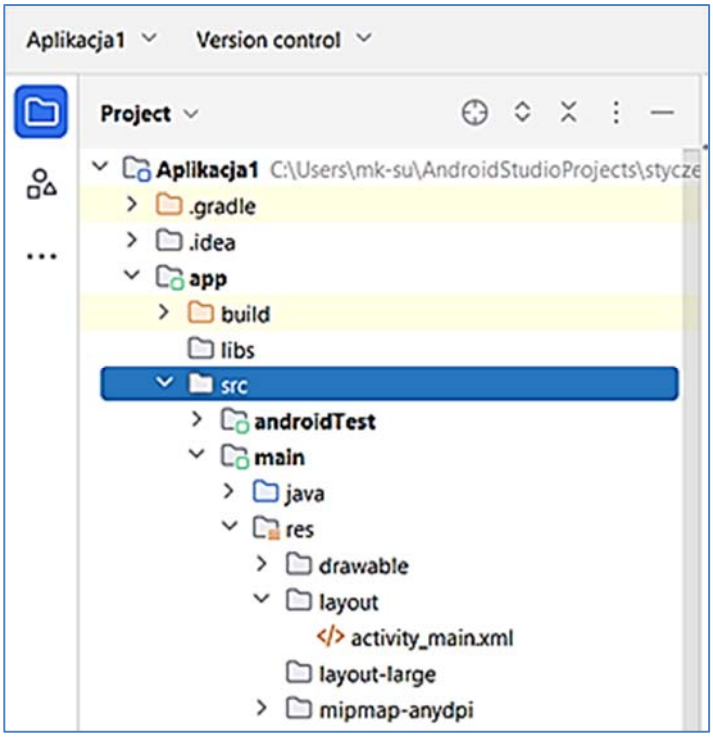
Rysunek 6.148 Tworzenie katalogu layout-large przechowującego układy dla urządzeń o większym ekranie

Po utworzeniu katalogu nie będzie on widoczny w drzewie aplikacji. Aby go wyświetlić przechodzimy do zakładki **Project**.



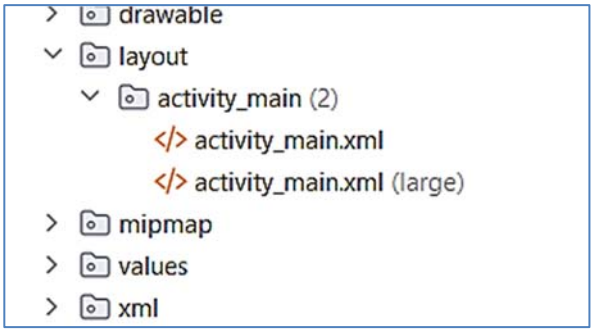
Rysunek 6.149: Zakładka Projekt w drzewie projektu

Rozwijamy gałąź **aplikacja**, a następnie przechodzimy po kolei przez następujące gałęzie: **aplikacja → app → src → main → res**. W tym miejscu już widać utworzony **katalog**. **Katalog** jest pusty o czym świadczy brak strzałki, która umożliwia jego rozwinięcie.



Rysunek 6.150 Katalog layout-large widoczny w drzewie projektu

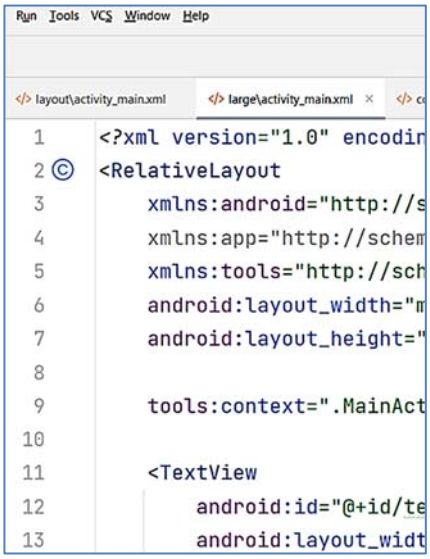
W katalogu umieszczamy skopiowany z katalogu **layout** plik **activity_main.xml**. Teraz w drzewie aplikacji będą dostępne oba pliki.



Rysunek 6.151 Pliki activity_main.xml, oraz activity_main.xml z opcją large na drzewie aplikacji

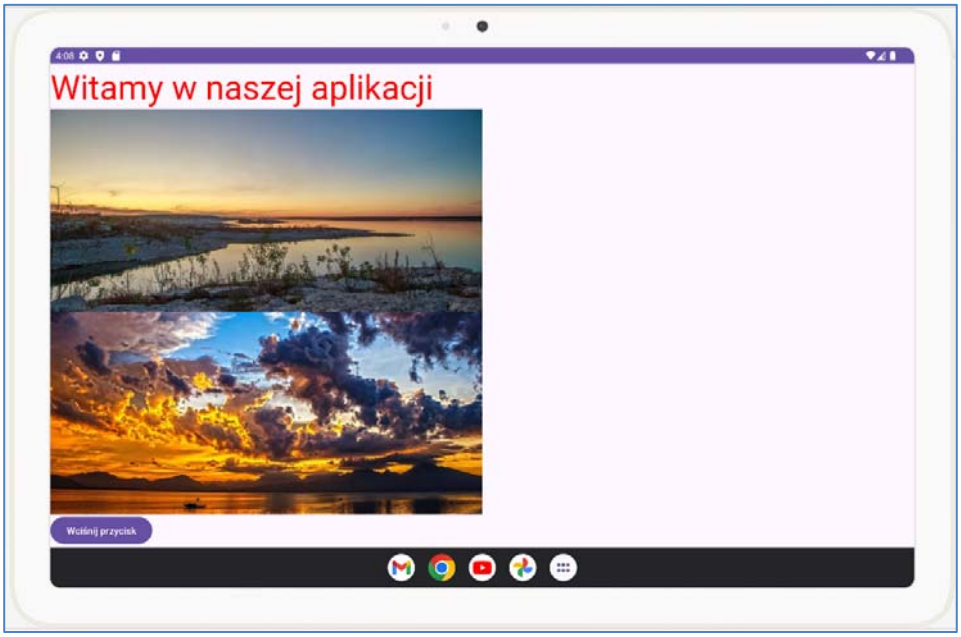
Po otwarciu pliku widzimy, że jest on taki sam jak poprzednio i po uruchomieniu go na tablecie nic się nie zmieni. Plik układu możemy swobodnie edytować i dostosować jego widok do dużego ekranu.

Gdy spojrzymy na zakładki w projekcie widzimy, który plik activity_main.xml definiuje układ na mniejszym ekranie, a który na większym.



Rysunek 6.152 Pliki activity_main.xml, oraz activity_main.xml z opcją large jako zakładki w projekcie

Na początek zmienimy wielkość czcionki w napisie na **50sp** i kolor tekstu na **czerwony**. Uruchomimy aplikację i sprawdzimy czy nastąpiły zmiany.



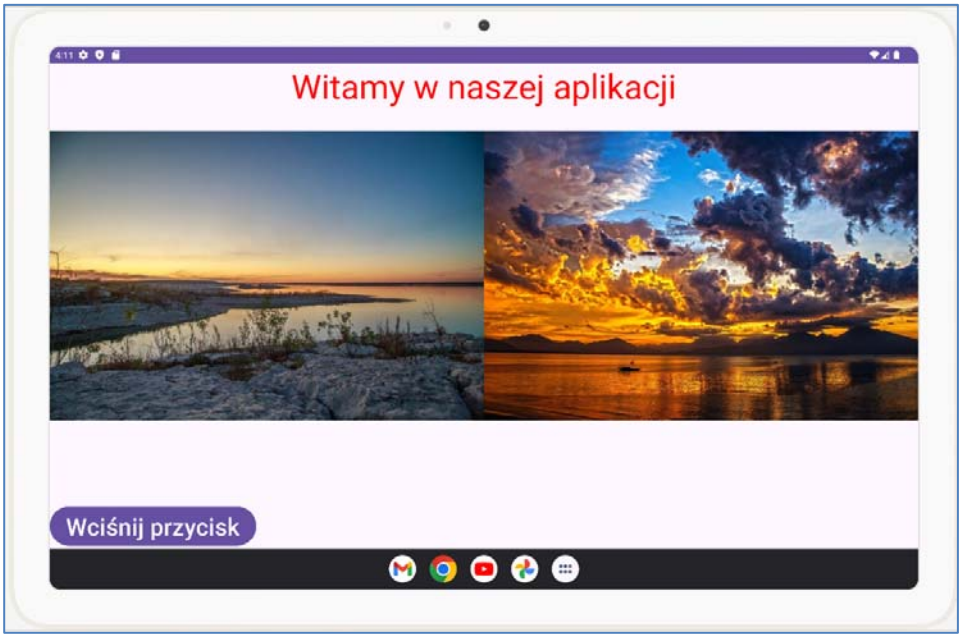
Rysunek 6.153 Aplikacja na dużym ekranie wyświetlona po zmianie parametrów w pliku main_activity.xml z opcją large

Widzimy na rysunku, że kolor i rozmiar czcionki zmieniły się. W aplikacji uruchomionej na smartfonie ustawienia widoku **TextView** są takie same jak poprzednio. Edytujemy plik **activity_main.xml** dla dużych ekranów. Kod układu będzie wyglądał następująco:

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

    <TextView
        android:id="@+id/tekst1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:text="@string/napis"
        android:layout_alignParentTop="true"
        android:textColor="@color/red"
        android:textSize="50sp"
        />
    <ImageView
        android:layout_marginTop="100dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/obrazek"
        android:src="@drawable/obrazek"
        android:layout_below="@id/tekst1"
        android:layout_alignParentTop="true"
        />
    <ImageView
        android:layout_marginTop="100dp"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_centerHorizontal="true"
        android:layout_toRightOf="@id/obrazek"
        android:id="@+id/obrazek2"
        android:src="@drawable/obrazek2"/>
    <Button
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/przycisk"
        android:textSize="35dp"
        android:layout_alignParentBottom="true"
        />
</RelativeLayout>
```

Listing 6.280 Kod układu activity_main.xml/large

Aplikacja po wprowadzeniu zmian będzie wyglądała następująco:



Rysunek 6.154 Aplikacja z modyfikowanym układem przeznaczonym dla urządzeń z większym ekranem

6.24 Przygotowanie aplikacji do publikacji w sklepie Google

Gdy nasza aplikacja jest już gotowa mamy możliwość podzielenia się nią z innymi. Jeżeli chodzi o aplikacje przeznaczoną dla telefonu z systemem operacyjnym Android odpowiednim miejsce będzie **Sklep Google Play**.

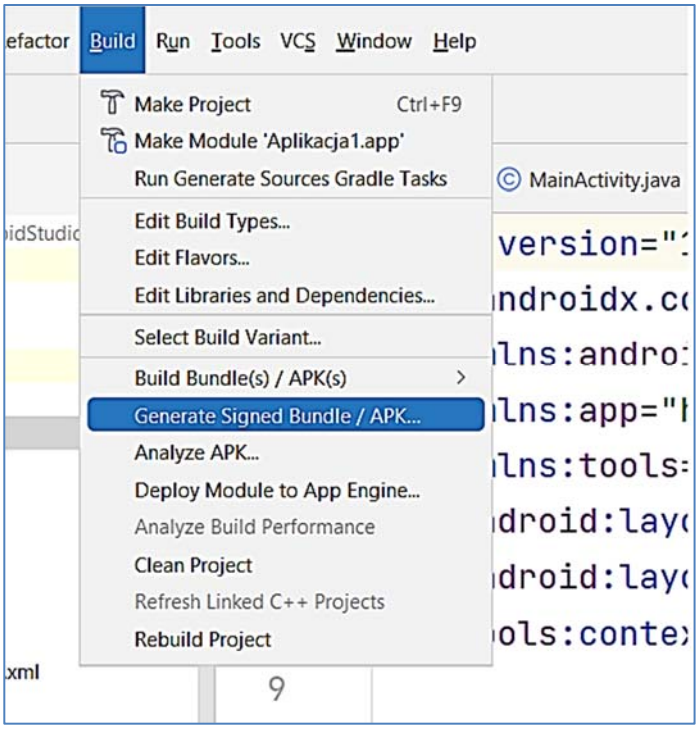
Zanim będziemy mogli przesłać aplikację do publikacji musimy ją odpowiednio przygotować. Aplikację należy odpowiednio zoptymalizować. Pozbywamy się z kodu nieużywanych **klas**, oraz **metod**. Dobrze, żeby kod był odpowiednio skompresowany. Pomocna w tym będą programy dostępne na rynku np. **R8** lub **ProGuard**. Ustawiamy odpowiednie uprawnienia dostępu tak, żeby użytkownik nie mógł np. zbyt modyfikować struktury bazy danych.

Zajmujemy się następnie wszystkimi plikami multimedialnymi. W przypadku obrazków trzeba zadbać o odpowiednie rozmiary. Warto przetestować działanie aplikacji pod różnymi ekranami.

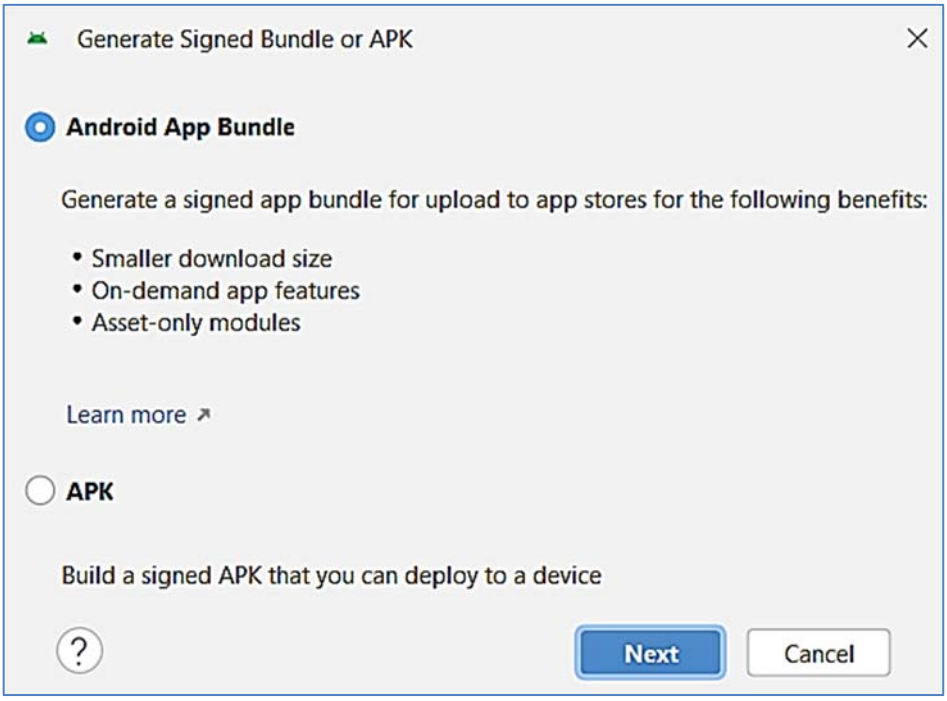
Aplikacja powinna wyglądać bezbłędnie na każdym uruchomionym urządzeniu. Warto również sprawdzić jej działanie na różnych poziomach API. Pliki muzyczne i filmowe powinny być odpowiednio skompresowane. Aplikacja powinna być jak najlżejsza tzn. **z zoptymalizowanym kodem**.

Podczas jej publikowania, trzeba umieścić jej opis. Warto być przygotowanym i w momencie publikacji mieć gotowy tekst do publikacji. Gdy wszystkie powyższe czynności są wykonane i nasza aplikacja jest gotowa to podzielenia się nią z użytkownikami Internetu, trzeba przygotować odpowiedni plik instalacyjny.

Android Studio umożliwia wygenerowanie podpisanego pliku **APK** bądź **bundle**. Co do wyboru sposobu zapisu to **APK** uznaje się za przestarzałe i najlepszym rozwiązaniem będzie utworzenie pliku **bundle**. Aby wygenerować odpowiedni plik przechodzimy do zakładki **Build** i wybieramy **Generate Signed Bundle / APK**.

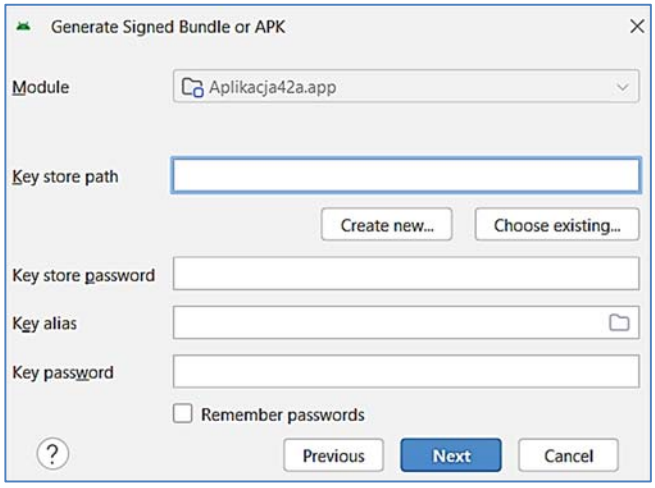


Rysunek 6.155 Generowanie pliku Bundle

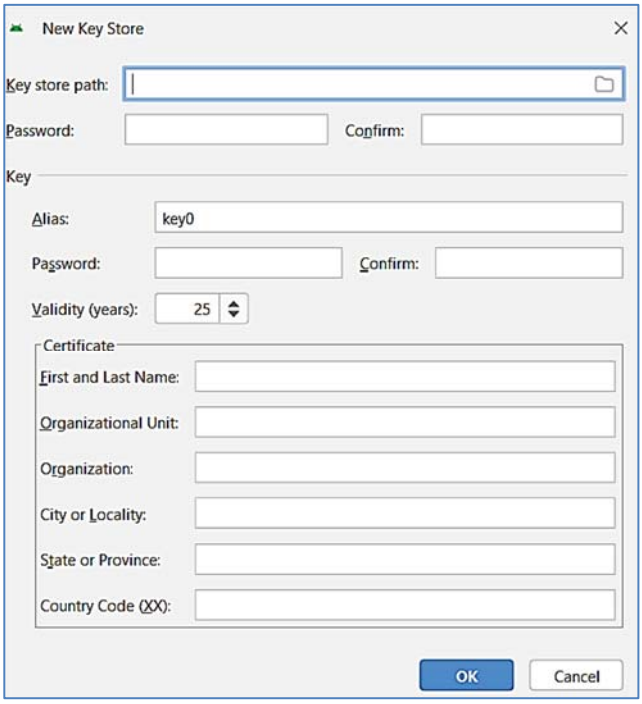


Rysunek 6.156 Generowanie podpisu do pliku Bundle

W okienku, które się pojawi wybieramy opcję **Android App Bundle** i klikamy **Next**. W następnym kroku trzeba przygotować tzw. **klucz**. Klikamy **Create New** i przechodzimy do następnego okienka.

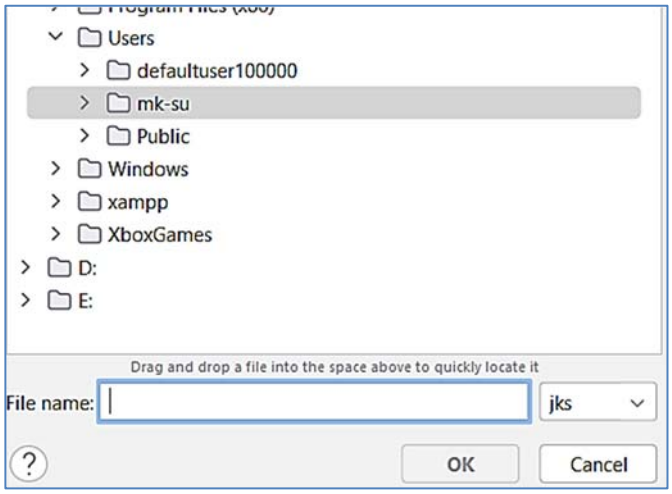


Rysunek 6.157 Generowanie podpisu do pliku Bundle – formularz



Rysunek 6.158 Tworzenie klucza

Wypełniamy po kolei wszystkie pola: **Key store path**: wybieramy katalog w którym będziemy przechowywać klucz. Klikamy w ikonę **Folderu**. Pojawi się kolejne okienko.

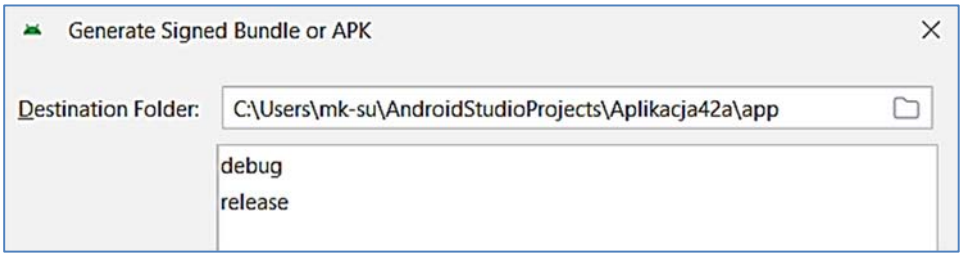


Rysunek 6.159 Okno wyboru katalogu przechowującego wygenerowany klucz

Wybieramy **katalog** w którym umieścimy **klucz**. Na dole formularza wpisujemy nazwę klucza np. **klucz1**. Po kliknięciu **OK** powracamy do poprzedniego okienka. Uzupełniamy je dalej. **Password**: tworzymy hasło, w polu **Confirm** powtarzamy je. To samo wpisujemy w polach w obszarze **Key**.

Dalej musimy uzupełnić dane dotyczące twórcy. **First** and **Last Name**: **Imię** i **Nazwisko**. **Organizational Unit**: **Jednostka organizacyjna**. **Organization**: **Organizacja**, firma. **City of Localty**: **Miasto**. **State or Province**: **Region** np. województwo. **Country Code**: **Kod kraju**.

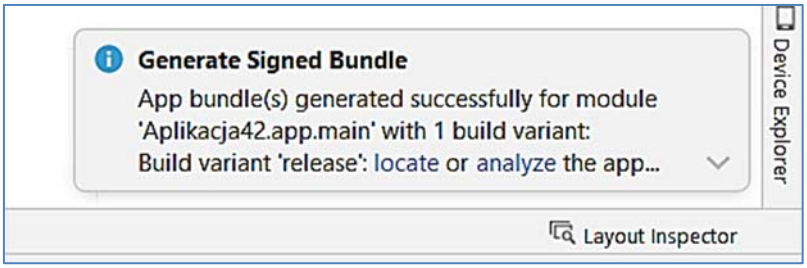
Po wypełnieniu wszystkich pól klikamy **OK**. Powracamy do pierwszego z okienek i tam klikamy **Next**. Wybieramy **Folder** do którego ma zostać zapisany plik aplikacji. Wybieramy wariant pomiędzy **debug** a **relase**.



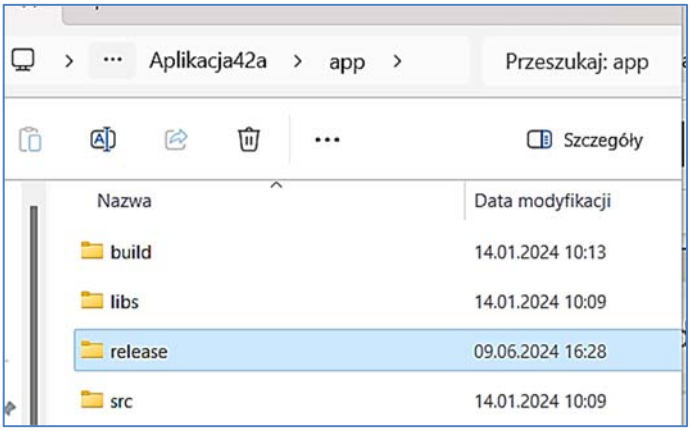
Rysunek 6.160 Okno wyboru Build Variants

Jeżeli chcemy, aby aplikacja była opublikowana w **sklepie Play** wybieramy opcję **release**. Klikamy przycisk **Create**.

Gradle buduje plik. Czekamy aż zakończy się proces: **Gradle Build**. Na ekranie otrzymamy odpowiedni komunikat. W pojawiającym się okienku klikamy **locate**. Wówczas otworzy się katalog aplikacji w którym znajdziemy plik instalacyjny.

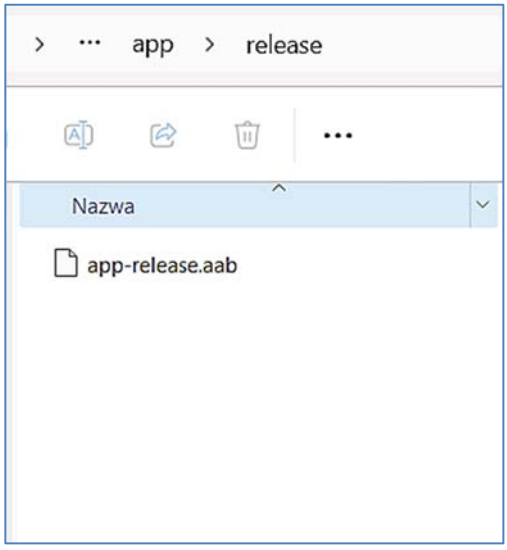


Rysunek 6.161 Potwierdzenie utworzenia pliku Bundle



Rysunek 6.162 Katalog relase na dysku komputera

Plik znajduje się w katalogu **release**.



Rysunek 6.163 Plik Bundle na dysku komputera

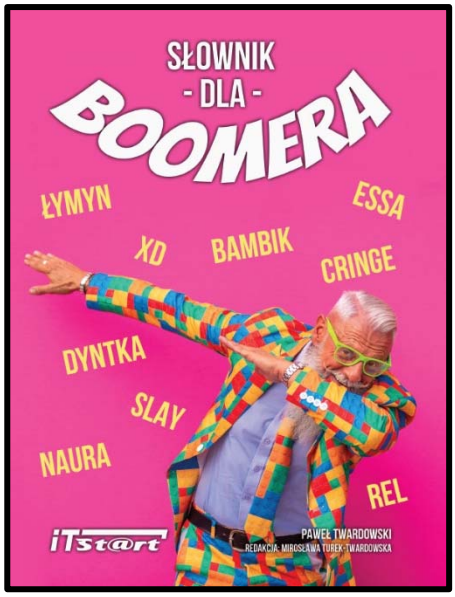
Aby opublikować aplikację trzeba jeszcze założyć konto programisty w **Google Play**. Koszt założenia konta to **25$**. Jest to opłata jednorazowa za założenie konta w ramach którego można publikować nieograniczoną liczbę aplikacji.

NOWOŚCI WYDAWNICZE:

Dogadaj się ze swoim wnukiem, to ilustrowany przewodnik prowadzący do zrozumienia młodzieżowej mowy. Każda babcia i dziadek często mają problem z przyswojeniem młodzieżowego slangu swojego wnuka, dlatego książka ta pomoże otworzyć szerzej oczy na młode pokolenie.

Znajomość tzw. „nowomowy" nie tylko pozwala zrozumieć lepiej młodzieżowy język, ale i umacnia wielopokoleniowe więzi, które bez dwustronnej komunikacji bywają zachwiane. Do każdego pojęcia zostały dodane elementy graficzne, powodujące, że książka staje się częściowo komiksem z pełnym wsparciem dialogów zapadających w pamięci każdego seniora.

Czy jesteś gotów na głęboką podróż w świat słów i zwrotów, które rządzą dzisiejszą młodzieżą? Ta fascynująca książka, stworzona specjalnie dla Ciebie, to klucz do zrozumienia tajemniczego języka młodszych pokoleń. Od dziwnych skrótów po szokujące akronimy – ten słownik odkryje przed Tobą tajniki młodzieżowej mowy.



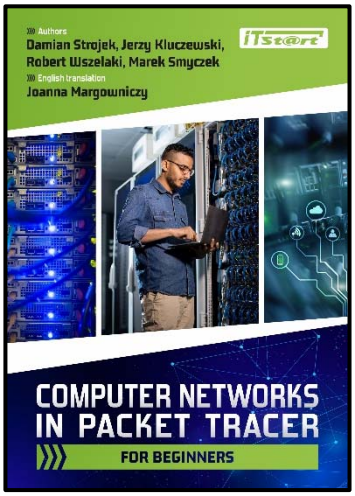
Wniknij w głowę nastolatków i poznaj zwroty, które kształtują ich świat. Dzięki temu słownikowi nie tylko poznasz znaczenie słów, ale także zrozumiesz, jakie emocje, idee i trendy kryją się za nimi. W trakcie tej ekscytującej podróży odkryjesz, dlaczego młodzież używa określeń takich jak , "YOLO", "LOL" i wiele innych.

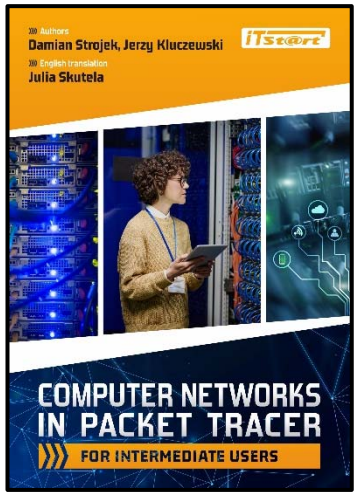
Słownik dla BOOMERA to nie tylko narzędzie do nauki, to przewodnik po współczesnej kulturze, który pomaga zrozumieć młodsze pokolenia i budować mosty między nimi a światem dorosłych. Czy jesteś gotów na przygodę? Wsiądź do kapsuły czasu i przenieś się w świat dzisiejszej młodzieży.

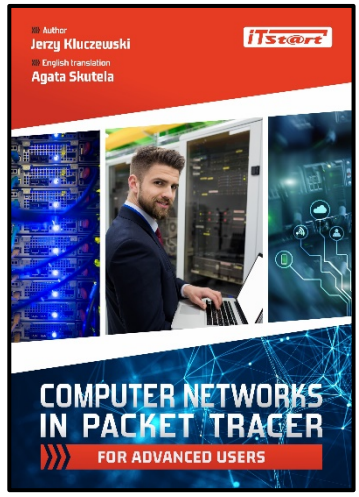
NOWOŚCI I ZAPOWIEDZI PROGRAMISTYCZNE:

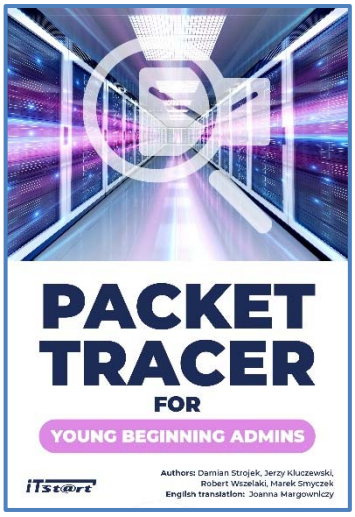
ENGLISH VERSION:

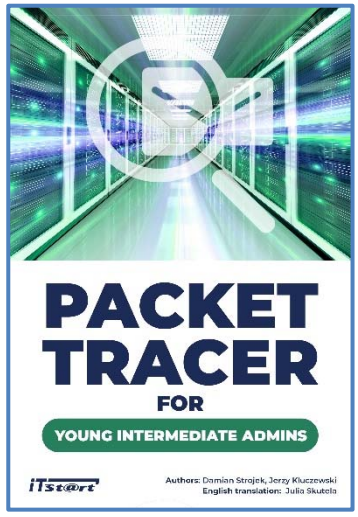
4,5

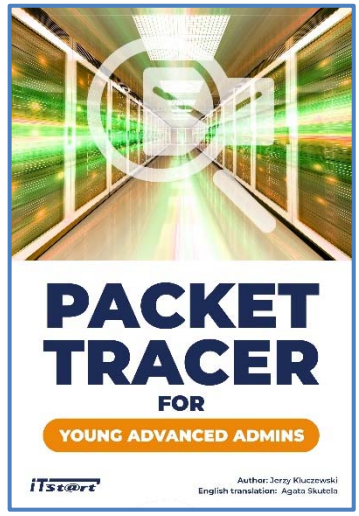




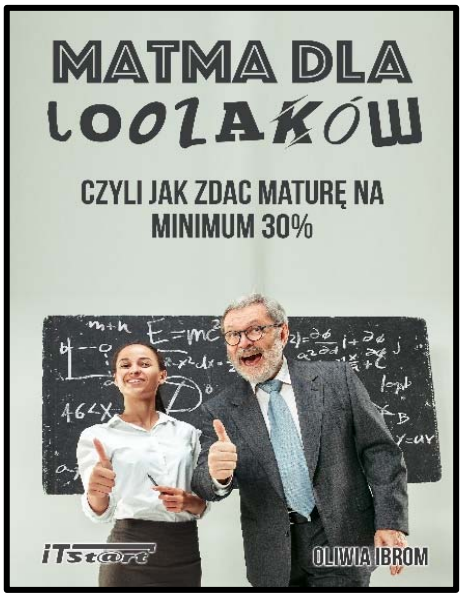








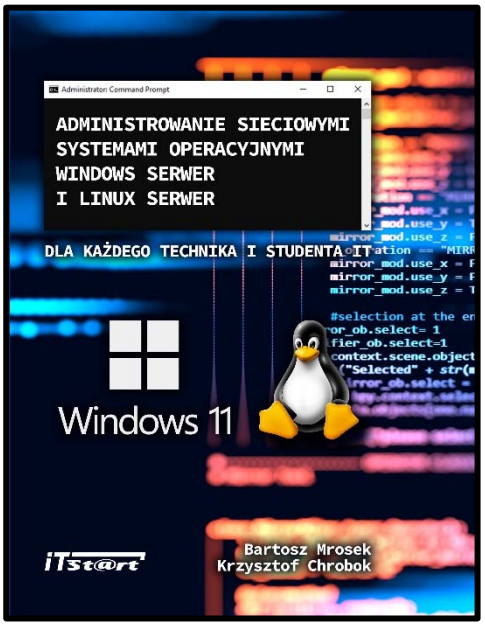
NASZE BESTSELLERY:



Nie każdy musi być orłem z matematyki. Wielu z nas jest mistrzem w innych dziedzinach i jest to całkowicie normalne. Jednak maturę wypadałoby zdać, tylko pytanie brzmi jak? Odpowiedzią są treści zawarte w książce, którą trzymasz w ręce „**Matma dla Loozaków, czyli jak zdać Maturę na minimum 30%**”.

Nie trzeba siedzieć po nocach żeby zdać maturę. Lepiej jest zrozumieć i wypracować pewne schematy, dzięki którym da się rozwiązać nawet najtrudniejsze zadania. Zagadnienia te, są rdzeniem tej książki, obejmującej cały zakres matematyki, obowiązujący na maturze. Bogactwo przykładów, ćwiczeń oraz zadań pomoże opanować i zrozumieć wszystkie niezbędne treści i nauczy Cię prostych rozwiązań. Cenną wartością dodaną do książki są quizy online, dostępne w łatwy sposób dzięki linkom zapisanych w postaci kodów QR.

Administrowanie sieciowymi systemami operacyjnymi Windows Serwer i Linux Serwer dla każdego Technika i studenta IT to podręcznik przygotowany z myślą o każdym uczniu oraz studencie związanym z kierunkami informatycznymi. Główną tematyką jest konfiguracja i zarządzanie sieciowymi systemami operacyjnymi Windows i Linux, dlatego książka będzie niezastąpioną pomocą podczas przygotowywania się do egzaminów zawodowych, jak i zaliczenia semestru.



Zagadnienia opisane w książce zostały odpowiednio ułożone i podzielone na odrębne tematy, gdzie każda usługa to jeden, odrębny rozdział, kolejno omawiający krok po kroku konfigurację usług zarówno w środowiskach Linux jak i Windows.

Podręcznik przygotuje użytkownika do instalacji systemu i konfiguracji kluczowych usług, czyli: SSH, Active Directory, Samba SMB, FTP, RDP, e-mail, HTTP oraz NAT.

PRACOWNIA URZĄDZEŃ TECHNIKI KOMPUTEROWEJ DLA UCZNIÓW I STUDENTÓW – Część 1 to zbiór przykładów i zadań opracowany pod kątem praktycznych zagadnień laboratoryjnych. Znajdziemy tu symulacje obwodów pomagających zrozumieć świat elektroniki analogowej i cyfrowej. Przedstawiono sposoby badań układów i parametrów podzespołów komputerowych jak i całych urządzeń. Napisanie idealnego zbioru ćwiczeń laboratoryjnych jest bardzo trudne. Dlatego autor nie zawarł wszystkich możliwych zagadnień związanych z zasadą działania komputera i jego budowy, ale zostawił sobie furtkę do kontynuowania tematyki, co ma zamiar przedstawić w kolejnych częściach książki.

PRACOWNIA URZĄDZEŃ TECHNIKI KOMPUTEROWEJ DLA UCZNIÓW I STUDENTÓW – Część 2 to zbiór przykładów, ćwiczeń i zadań opracowany pod kątem praktycznych zagadnień laboratoryjnych bazujących na symulacjach. Znajdziemy tu symulacje obwodów pomagających zrozumieć świat elektroniki cyfrowej i zasad na podstawie których działają komputery. Przedstawiono zasady projektowania, badania i programowania układów cyfrowych.

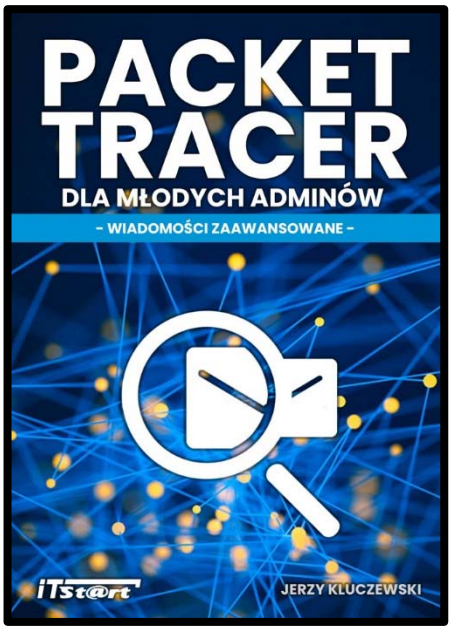
Książka dedykowana jest uczniom, studentom i nauczycielom. Zawiera materiały pomocnicze dla przedmiotów związanych z elektroniką cyfrową i urządzeniami techniki komputerowej. Może być wykorzystana w laboratoriach studenckich jak i na kursach branżowych. Omawia podstawy algebry Boole'a, minimalizację funkcji logicznych i zagadnienia związane z programowaniem. Omówione zostały układy kombinacyjne, sekwencyjne i pamięci w postaci konkretnych układów scalonych. Cennym dodatkiem do książki jest możliwość pobrania plików ćwiczeń, przykładów i prezentacji przygotowanych przez autora.

Linux dla każdego Technika IT to książka w głównej myśli kierowana do każdego ucznia oraz studenta uczącego się na kierunku informatycznym lub pokrewnym. Krótko mówiąc dla wszystkich, którzy na co dzień nie pracują z systemem Linux, a chcą się nauczyć jego obsługi zarówno za pośrednictwem interfejsu graficznego jak i tekstowego.

Tematyka książki jest bardzo szeroka. Przygotowuje użytkownika do instalacji systemu i konfiguracji: sprzętu, sieci, środowiska pracy, użytkowników. Pokazuje jak pracować z plikami, stosować uprawnienia, a także tworzyć skrypty systemowe. W treści poruszone też zostały kwestie związane z bezpieczeństwem sytemu oraz danych.

Na końcu książki znajduje się próbny egzamin zawodowy, w którym każde pytanie poddane zostaje szczegółowej analizie w wyniku której, prezentowana jest poprawna odpowiedź, co stanowi wartość dodaną zarówno dla ucznia jak i nauczyciela. Znajdziemy tutaj również wykaz najważniejszych poleceń systemowych – niezbędnych do codziennej pracy.

PACKET TRACER DLA MŁODYCH ADMINÓW – Wiadomości zaawansowane,



to zbiór scenariuszy oraz symulacji sieciowych dla użytkowników posiadających już co najmniej podstawową wiedzę z sieci komputerowych i doświadczenie w pracy związanej z administracją podstawowymi urządzeniami tworzącymi architekturę sieci. Symulacje zostały przygotowane w taki sposób, aby rozwijać wiedzę przedstawioną już w poprzednich książkach Naszego Wydawnictwa przez Jerzego Kluczewskiego.

Tematyka książki jest bardzo rozbudowana i różnorodna. Tryb Multiuser, wirtualizacja, połączenia Bluetooth, sieci komórkowe, kontrolery sieci WLAN, routery przemysłowe, protokół IPV6, to tylko kilka wybranych zagadnień do których Autor przygotował scenariusze oraz gotowe pliki symulacyjne, które można pobrać z witryny wydawnictwa.

Seria książek Senior pracuje(…) to ciekawe kompendium wiedzy opisujące zagadnienia jak zacząć pracę z Windowsem 10, z Internetem, z Edytorem tekstów czy też jak zapanować nad arkuszami kalkulacyjnymi. Z myślą o osobach starszych zastosowano większą czcionkę w każdej pozycji

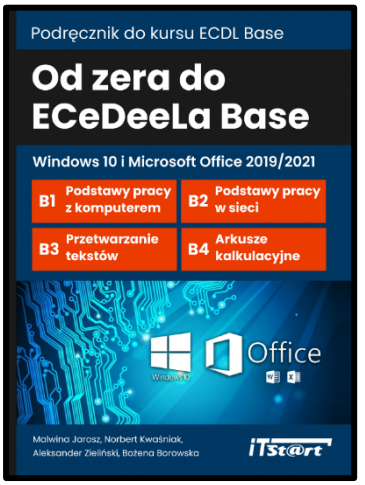
Część pierwsza skupia się wokół obsługi systemu Windows 10 zainstalowanym na komputerze stacjonarnym lub przenośnym. Książka została przygotowana w taki sposób aby użytkownik z różnym stopniem zaawansowania odnalazł coś dla siebie. Przedstawione zostały podstawowe pojęcia, nazwy związane ze sprzętem, systemem Windows i oprogramowaniem użytkowym.

Część druga opisuje zagadnienia wykorzystania sieci komputerowej i Internetu. Książka została przygotowana dla czytelnika z różnym stopniem zaawansowania. Przedstawione zostały podstawowe pojęcia i nazwy związane z przeglądaniem Internetu i wyszukiwaniu w nim informacji, komunikacji i korzystania z poczty elektronicznej.

Część trzecia wyjaśnia tematy związane z ogólnie pojętym tematem pisania tekstów. Książka została dostosowana do czytelnika z różnym stopniem zaawansowania umiejętności związanych z edycją tekstów. Przedstawione zostały tutaj podstawowe pojęcia i nazwy związane z pracą podczas tworzenia jak i edycji tekstu za pomocą komputera.

Część czwarta zawiera wiele zagadnień związanych z obliczeniami z wykorzystaniem komputera. Książka napisana z myślą o czytelnikach z różnymi stopniami zaawansowania. Przedstawione zostały tutaj podstawowe pojęcia i nazwy związane z obliczeniami i ich wizualizacją za pośrednictwem wykresów.

Książka **Od Zera do ECeDeeLa Base** w pierwszej kolejności została napisana z myślą o każdym Europejczyku biorącym udział w kursach kompetencji informatycznych. Jest podręcznikiem, który stanowi podstawowe narzędzie do zdobycia kluczowych informacji w najpopularniejszym z modułów szkoleń, wspomagający uzyskanie międzynarodowego certyfikatu ECDL Base. Jednocześnie jest to książka którą może wziąć do ręki każdy początkujący użytkownik komputera, chcący samodzielnie poszerzyć swoją wiedzę.



Wszelkie zagadnienia zostały przedstawione w prostej i przejrzystej formie, tak aby każdy mógł samodzielnie się uczyć obsługi komputera wyposażonego w system Windows 10 oraz pakiet biurowy MS Office 2019. Treść podręcznika została podzielone na cztery funkcjonalne części. W pierwszej znajdziemy wszystko, co związane jest z podstawowymi czynnościami pracy w systemie Windows, czyli zarządzaniem folderami i plikami, konfiguracją i dostosowaniem pulpitu oraz drukowaniem. W części drugiej opisane zostały zasady pracy w sieci oraz działania, z którymi każdy użytkownik komputera spotyka się podczas codziennej pracy w wykorzystaniem Internetu. Część trzecia porusza zagadnienia tworzenia dokumentów tekstowych, a ostatnia skupia się na arkuszach kalkulacyjnych i wizualizacji danych za pośrednictwem wykresów.

Książka „**Podstawy sieci dla technika i studenta - Część 1**” to kompletny zasób wiedzy w ujęciu teoretycznym jak i praktycznym, poświęcony sieciom komputerowym. Autor przystępnym językiem oprócz zarysu historycznego przedstawił media i sygnały transmisyjne, począwszy od teorii, aż do zastosowań praktycznych, demonstrując również techniki ich montażu. W książce dokładnie opisano podstawy komunikacji sieciowej, zasadę działania urządzeń i usług działających w współczesnym Internecie.

Książka kierowana jest dla każdego czytelnika, nawet ze znikomą wiedzą informatyczną, chcącego zrozumieć jak działają sieci komputerowe. Przede wszystkim adresowana jest do uczniów techników informatycznych, teleinformatycznych oraz studentów IT, ponieważ zawiera odnośniki do podstaw programowych i zagadnień jakie należy opanować w trakcie całego cyklu nauki. Dodatkową pomocą w zdobywaniu wiedzy są przykłady, ćwiczenia, pytania diagnozujące wiedzę i linki w postaci kodów QR do treści źródłowych, a także zestaw samo oceniających plików *.pka symulatora sieciowego Packet Tracer.

ZBIÓR ZADAŃ Z SIECI KOMPUTEROWYCH, to bogaty zasób zadań oraz ćwiczeń laboratoryjnych. Dzięki niemu można samodzielnie lub pod kontrolą nauczyciela/wykładowcy, wykonać i zasymulować wiele zagadnień sieciowych na podstawie gotowych scenariuszy, kart laboratoryjnych oraz plików przygotowanych w programie Packet Tracer *(wszystkie niezbędne pliki są dostępne do pobrania dla każdego posiadacza książki)*. Większość zadań sieciowych opisanych w podręczniku jest zgodna z wymaganiami IT na kierunku TECHNIK INFORMATYK w szkole średniej oraz z zagadnieniami poznawanymi przez studentów informatyki o specjalizacji SIECI KOMPUTEROWE.

ROZWIĄZANIA ZADAŃ Z SIECI KOMPUTEROWYCH jest uzupełnieniem książki pt. **ZBIÓR ZADAŃ Z SIECI KOMPUTEROWYCH**. Stanowi ona kompletny zestaw rozwiązań zagadnień problemowych przedstawionych w powiązanej pozycji. Wśród rozwiązań możemy znaleźć, wypełnione karty pracy ucznia/studenta wraz z przykładowymi wnioskami. Książka przeznaczona jest dla nauczycieli szkół średnich na kierunku: TECHNIK INFORMATYK oraz wykładowców uczelni kierunków IT o specjalizacji SIECI KOMPUTEROWE.

Wszystkie proponowane rozwiązania w postaci gotowych kart pracy oraz rozwiązanych problemów symulacyjnych w programie Packet Tracer stanowią dodatek, dla każdego posiadacza książki, które można pobrać wprost z serwera naszego wydawnictwa.

„**Prawdziwa historia Internetu na świecie**” to już czwarte wydanie obszernego źródła wiedzy o Internecie i jego ewolucji na całym świecie. To jedyny i zarazem unikatowy zbiór najważniejszych wydarzeń oraz postaci, które miały wpływ na rozwój i ewolucję sieci. Znajdziemy tu szczegółowo opisane najważniejsze usługi, protokoły i programy dzięki którym łatwo i sprawnie wymieniamy pliki, rozmawiamy, dzielimy się wiedzą i przeżyciami, czyli po prostu surfujemy po sieci.

Książka kierowana jest dla ludzi, którym nie wystarcza samo surfowanie po Internecie, lecz pociąga ich wiedza i ciekawość. To także niesamowita gratka dla nauczycieli-pasjonatów historii współczesnej z zacięciem do IT, którzy wiedzę zawartą w książce mogą swobodnie wykorzystać podczas ciekawej, innowacyjnej lekcji historii z uczniami np. omawiając światowe wojny przeglądarek internetowych lub batalie wyszukiwarek informacji.

„**Prawdziwa historia Internetu w Polsce**” to rozszerzenie popularnej książki naszego wydawnictwa o tej samej nazwie, lecz opisującej wydarzenia związane z rozwojem Internetu na całym świecie. Jest unikatową, obszerną na skalę kraju pozycją książkową, będącą źródłem wiedzy o najważniejszych wydarzeniach i postaciach, które wpłynęły na rozwój sieci w naszym kraju.

Książka opisuje rozwój rodzimego rynku dostawców internetowych, omawia w jaki sposób pobierało się pierwsze programy komputerowe wprost z radia, jak i gdzie włamywali się pierwsi polscy hakerzy, na czym polegała wojna polsko-turecka, oraz jakie znaczenie w sieci miał polski hydraulik we Francji.

Czytając książkę rozdział po rozdziale, dowiemy się jak zmieniły się z biegiem czasu oczekiwania polskich internautów, jak ewoluowały pierwsze serwisy aukcyjne, kiedy i z jakiego powodu pękły pierwsze bańki internetowe. To tylko fragment, tego co czytelnik znajdzie w całej treści. Pozycja ta skierowana jest do ludzi, którym nie wystarcza samo surfowanie po Internecie, lecz pociąga ich wiedza i ciekawość. Jest to książka popularnonaukowa, w której autor postawił sobie bardzo trudne zadanie: w jasny i przejrzysty sposób pokazać pełną historię Internetu w naszym kraju.

Książka **„Tworzenie gier dla początkujących”** dedykowana jest wszystkim osobom chcącym wgłębić się w świat programowania gier 3D oraz zaznajomić się i wykorzystać możliwości środowiska Unity. Celem książki jest przekazanie początkującym twórcom wiedzy z zakresu: obsługi podstawowych funkcjonalności, pisania i implementacji skryptów, tworzenia obiektów fizycznych, animacji, teksturowania i optymalizacji.

Książka adresowana jest również dla każdego, już nieco bardziej doświadczonego programisty, który szuka nowych, rozbudowanych narzędzi, pozwalających poszerzyć swoją dotychczas zdobytą wiedzę o coś zupełnie odmiennego i niezwykle kreatywnego.

Stwórz swój własny trójwymiarowy świat i zbuduj model rozgrywki tak, jak sobie tylko wymarzysz. Bo jedyne co ogranicza Cię w tworzeniu gier, to Twoja wyobraźnia!

Książka **„Bezpieczeństwo w sieci – Jak skutecznie chronić się przed atakami”** dedykowana jest każdemu kto codziennie korzysta z komputera i Internetu. Głównym celem autorów było przygotowanie prostego poradnika, który pomoże każdemu, bez względu na poziom wtajemniczenia informatycznego, wiek czy płeć, przemyśleć sposób w jaki korzysta z komputera i sieci.

W książce zgromadziliśmy wyselekcjonowany zasób podstawowych pojęć informatycznych. Przedstawiliśmy zagrożenia jakie czyhają w sieci, oraz tajemniczego wroga sieciowego, bo zrozumienie jego postępowania to klucz do naszego bezpieczeństwa.

Omawiając zasady bezpiecznego korzystania z systemu Windows i domowej lub firmowej sieci komputerowej staraliśmy się dać przepis jak zabezpieczyć nasze cyfrowe życie. Praca z przeglądarką Internetu, korzystanie z chmury, bezpieczne zakupy czy ochrona dzieci to kluczowe pojęcia jakimi się zajęliśmy.

Książki: **„Kwalifikacja INF.02 i INF.07 Administrowanie sieciami komputerowymi w programie Packet Tracer” Część1** i **Część2** oraz **„Zarządzanie Sieciami Komputerowymi w Programie Packet Tracer” - Wiadomości podstawowe i wiadomości zaawansowane** książki opisujące bardzo podobne zagadnienia.

Pierwsze z nich dedykowane są uczniom szkół średnich uczących się w zawodach: **Technik Informatyk i Teleinformatyk**, drugie natomiast zwykłym czytelnikom chcącym samodzielnie opanować zagadnienia sieciowe.

Książki zawierają porady dla osób chcących poznać działanie sieci oraz większości sprzętu, które tworzą współczesną architekturę sieciową.

Czytelnik znajdzie tutaj także podstawy konfigurowania usług sieciowych takich jak TELNET, SSH, FTP, EMAIL, DHCP, DNS oraz protokołów RIP, EIGRP, OSPF, eBGP, routing statyczny, listy kontroli dostępu, VoIP, STP, RSTP, VTP, FRAME RELAY, PPP, uwierzytelnianie PAP i CHAP, RADIUS, NETFLOW, NAT, L2NAT, VPN, tunelowanie. Ta część obejmuje również konfigurowanie przełączników wielowarstwowych 3560-24PS oraz 3650-24PS

Autorzy opisując zagadnienia administrowania sieciami komputerowymi, posługują się wieloma przykładami i ćwiczeniami. przyjęli zasadę minimum teorii maksimum przykładów praktycznych”, co umożliwia Czytelnikowi naukę administrowania
sieciami i urządzeniami bez potrzeby zakupu drogiego sprzętu Cisco

Książki są kompilacją poprzednich publikacji naszego Wydawnictwa z serii Packet Tracer, posiadają zaktualizowany interfejs obecnie najnowszego oprogramowania Packet Tracer oraz zawierają zestaw nowych przykładów i ćwiczeń.

Seria: „Również dla Seniora”

To seria trzech książek dedykowanych nie tylko seniorom, ale i wszystkim osobom chcącym w pełni wykorzystywać współczesne technologie przy użyciu: **Laptopa, Smartfonu i Internetu**.

Książki opisują codzienne wykorzystanie współczesnego sprzętu, który już od dość dawna zagościł w naszych domach, jednak bardzo często w minimalnym stopniu wykorzystujemy współczesne narzędzia działające w sieci, ograniczając się do podstawowych czynności, takich jak szukanie wiadomości czy zaglądanie do portalu społecznościowego. Czasami nie zdajemy sobie nawet sprawy z tego, że przez Internet możemy załatwić większość spraw urzędowych, wymienić walutę, czy nawet kupić produkty spożywcze w markecie bez wychodzenia z domu.

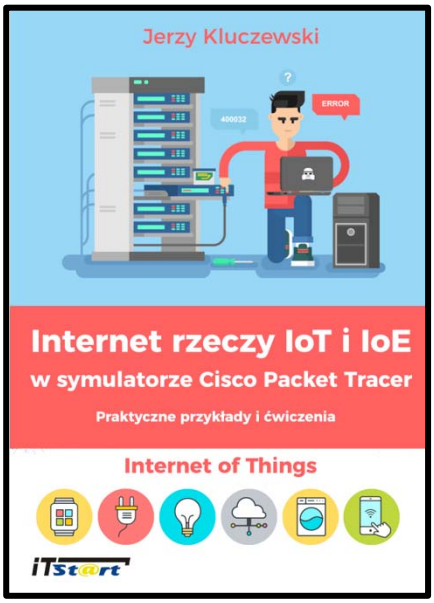
Książki adresowane są również dla seniorów, którzy chcą zdalnie dokonywać rezerwacji swojego wypoczynku, załatwiać sprawy w Urzędach miejskich, czy też korzystać z bankowości elektronicznej. Autorzy zwrócili szczególną uwagę nie tylko na bezpieczeństwo sprzętu i systemu operacyjnego, ale również przedstawili typowy model profilaktyki antywirusowej.

Treść wszystkich publikacji pisana jest bardzo przystępnym językiem, co pozwala nie tylko na łatwe przyswajanie wiedzy, ale przysłowiowo potrafi „wciągnąć” czytelnika. Książki, dedykowane są również wszystkim osobom chcącym w pełni poznać i wykorzystać możliwości współczesnych smartfonów.

Zdarza się, że nowoczesne telefony użytkujemy już od dawna, ale bardzo często ograniczamy się tylko do podstawowych czynności jakimi są dzwonienie czy pisanie SMSów. Czasami nie wiemy o tym, że smartfonem możemy sterować naszym telewizorem lub użyć go jako urządzenie udostępniające Internet naszemu laptopowi lub innemu urządzeniu mobilnemu.

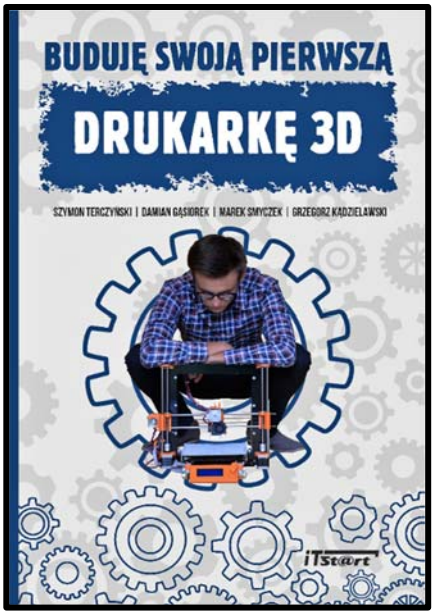
Autorzy, skupili się również na tym jak tworzyć, a następnie drukować proste dokumenty oraz jak posługiwać się skanerem. Nie brakło też miejsca na przedstawienie zagadnień jak przenieść zdjęcia z aparatu do komputera, a także jak posługiwać się pamięciami przenośnymi typu Pendrive oraz dyski USB.

Książka **Bezpieczeństwo sieci komputerowych - Praktyczne przykłady i ćwiczenia w symulatorze Cisco Packet Tracer**, kierowana jest do szerokiego grona osób chcących poszerzyć swoją wiedzę z zakresu bezpiecznego wykorzystania sieci komputerowych w codziennym życiu. Nauczyć się od podstaw projektować i wdrażać zasady bezpieczeństwa. Nie jest to jednak materiał uczący bezpiecznego korzystania z usług sieciowych od strony standardowego użytkownika Internetu.
Pozycja ta, to również idealny podręcznik dopełniający wiedzę praktyczną podczas nauki w techniku informatycznym, technikum teleinformatycznym i Akademiach CISCO CCNA. To materiał uzupełniający, dzięki któremu w prosty sposób można poszerzyć i uzupełnić swoją wiedzę i przygotować się do uzyskania kwalifikacji potwierdzających kompetencje zawodowe.
Książka zawiera materiał poświęcony klasyfikowaniu zagrożeń w sieciach, zasadom pozwalającym na unikanie cyber-ataków, założeniom polityki bezpieczeństwa, protokołom SSH, NTP, SYSLOG, RADIUS, TACACS+, GRE, Piec, usługom AAA a także problemom związanym z prawem ogólnego rozporządzenia o ochronie danych osobowych RODO (GDRP).



Internet rzeczy IoT i IoE w symulatorze Cisco Packet Tracer - Praktyczne przykłady i ćwiczenia. Czasy w których żyjemy, przyzwyczaiły nas do korzystania z mobilnego dostępu do Internetu. Jednak współczesna sieć to nie tylko treści audio i wideo, oraz dostęp do społeczności. To coraz bardziej rozbudowana możliwość kontroli lub sterowania elementami naszego otoczenia. Nikogo już nie dziwi zdalne sprawdzenie systemu alarmowego, automatyczne sterowanie oświetleniem czy ogrzewaniem. Coraz powszechniejsze są urządzenia AGD i RTV z adresem sieciowym. Za pomocą telefonu sterujemy między innymi pralką, zmywarką i ekspresem do kawy, a lodówki potrafią już samodzielnie dokonywać zakupów w sklepach internetowych. Jednak możliwości tej technologii są jeszcze bardziej rozbudowane i wciąż się rozwijają.
Książka dedykowana jest osobom chcącym jeszcze dokładniej zapoznać się z technologią „inteligentnych urządzeń", które kryją się pod anglojęzycznym pojęciem „Internet of Things". Jest to podręcznik, który ma nauczyć projektować, programować i tworzyć sieci komputerowe składające się z czujników wraz z urządzeniami wykonawczymi powszechnego użytku.

Buduję swoją pierwszą drukarkę 3D



Książka Buduję swoją pierwszą drukarkę 3D skierowana jest dla osób, które chcą poglądowo zapoznać się z technologiami przyrostowymi, a w szczególności technologią FDM (Fused Deposition Modeling), poznać historię technologii przyrostowych na świecie, a także dowiedzieć się jak można zbudować własną i tanią drukarkę 3D.

Nie jest to podręcznik teoretyczny, skupiający się na szczegółowym opisie każdej technologii czy też współczesnej drukarki. Stanowi natomiast pełny, praktyczny przewodnik, opisujący proces budowy i użytkowania drukarek 3D na przykładzie modelu Prusa i3. Zawiera także instrukcję montażu niecodziennego projektu drukarki 3D z odzyskanych, starych napędów CD ROM.

Książka ta, to również dowód i przykład jak obecnie wygląda współpraca uczelni wyższej z wybranymi szkołami średnimi (Zespół Szkół Nr1 z Piekar Śląskich oraz Śląskie Techniczne Zakłady Naukowe z Katowic). Współpraca szkół średnich z pracownikami Politechniki Śląskiej to dowód na wspieranie młodych talentów, a owocem tego jest między innymi budowa własnych drukarek, czyli urządzeń mechatronicznych, łączących konstrukcję, napędy i sterowanie. Tematyka związana z drukiem 3D jest jednym z filarów przemysłu 4.0 i coraz więcej firm poszukuje osób, które potrafią drukować modele przestrzenne.

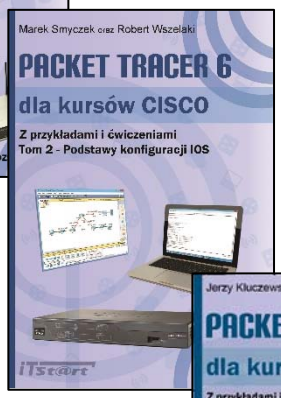
Platforma Moodle dla każdego nauczyciela, to podręcznik, który w zasadzie dedykowany jest dla wszystkich osób, które chcą wspomagać codzienny proces nauczania sprawdzonym i przetestowanym narzędziem do e-learningu, wykorzystywanym niemalże na całym świecie.

Platforma Moodle nie zastąpi z pewnością nauczyciela, a proces dydaktyczny nie będzie mógł przebiegać bez jego obecności, jednak zdecydowanie jest w stanie ułatwić pracę każdego pedagoga. W książce można znaleźć takie informacje jak: Tworzenie nowych kursów i zarządzanie nimi, przypisywanie do nich nowych uczestników, a także udostępnianie wcześniej przygotowanych materiałów dydaktycznych, testów lub zadań. Niezastąpioną pomocą jest przygotowana przez nas platforma testowa **http://demo.itstart.pl**, dzięki której można łatwo sprawdzić opisywane zagadnienia.

Osoby, które oprócz kształcenia pełnią również rolę administratorów platformy, albo stoją przed zadaniem zainstalowania jej na dowolnym systemie operacyjnym (Windows, Linux, Mac-os) również znajdą coś dla siebie, bo jeden z rozdziałów książki, szczegółowo opisuje właśnie te zagadnienia.

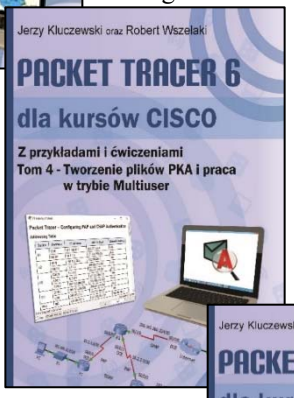
Seria książek: Packet Tracer 6 dla kursów CISCO (TOMY 1 – 5)

Książka **Packet Tracer 6 dla kursów CISCO Tom 1** dedykowana jest wszystkim osobom chcącym nauczyć się tworzyć sieci, zarządzać nimi, diagnozować uszkodzenia, a nawet je projektować, poświęcona jest podstawowym zagadnieniom spotykanym w sieciach LAN, dlatego idealnie nadaje się dla osób, które dopiero rozpoczynają przygodę z sieciami.



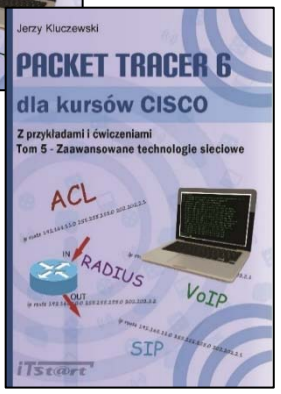
Książka **Packet Tracer 6 dla kursów CISCO Tom 2** dedykowana jest wszystkim osobom chcącym nauczyć się projektować i tworzyć nie tylko sieci lokalne, ale również zarządzać sprzętem spotykanym w sieciach rozległych. Tom 2 poświęcony jest routerom i przełącznikom CISCO wyposażonym w system IOS, ale również pokazuje możliwości programu, pozwalające na ich podstawową konfigurację bez konieczności zagłębiania się w arkana IOS.

Książka **Packet Tracer 6 dla kursów CISCO Tom 3** W odróżnieniu od dwóch poprzednich części, tom 3 opisuje działanie usług sieciowych, a nie tylko pokazuje jak nauczyć się konfigurowania urządzeń w programie Packet Tracer. Autor opisując dynamiczne protokoły routingu takie jak RIP, EIGRP, OSPF i BGP stara się przybliżyć zrozumieć czytelnikowi te zagadnienia i pomóc zasadę ich działania.



Packet Tracer 6 dla kursów CISCO Tom 4

napisana została z myślą o instruktorach programu Cisco Networking Academy, jako pomoc w tworzeniu interaktywnych zadań sprawdzających umiejętności uczestników kursów.



Packet Tracer 6 dla kursów CISCO Tom 5

Autor w tomie piątym szczegółowo opisuje zagadnienie routingu statycznego oraz działanie i konfigurację wielu protokołów, między innymi STP, VTP, PPP, czy Frame Relay. Znalazło się tu również miejsce na opisanie zagadnień kontroli dostępu do sieci za pomocą list ACL, urządzeń typu chmura i technologii VOIP. Wspomniano również o metodach ułatwiających zarządzanie siecią poprzez protokół Radius i oprogramowanie Netflow, analizujące ruch sieciowy.