DO NOWEJ PODSTAWY PROGRAMOWEJ

Część 3
Aplikacje webowe

Kwalifikacja INF.04
Projektowanie, programowanie i testowanie aplikacji

Podręcznik do nauki zawodu
technik programista

Łukasz Guziak

Helion

Redaktor prowadzący: Joanna Zaręba

Projekt okładki: Jan Paluch

Ilustracja na okładce została wykorzystana za zgodą Shutterstock.

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

Drogi Czytelniku!
Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres
http://helion.pl/user/opinie?inf043
Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

ISBN: 978-83-283-8102-5

Printed in Poland.

Spis treści

Wstęp

Niniejsza publikacja wchodzi w skład serii podręczników do nauki zawodu technik programista. Poruszono w niej zagadnienia związane z tworzeniem aplikacji webowych.

Przedstawiony w podręczniku materiał obejmuje podstawę programową wymienioną w dziale *INF.04.7. Programowanie aplikacji zaawansowanych webowych* kwalifikacji *INF.04. Projektowanie, programowanie i testowanie aplikacji* i pozwoli postawić następny krok w świecie technologii webowych.

Nieustanne zmiany zachodzące w otaczającym nas świecie nie ominęły zwłaszcza tej dziedziny. Jeszcze niedawno nie było smartfonów, Facebooka, aparatów cyfrowych, dostęp do internetu uzyskiwało się za pomocą modemu z szybkością 56 kb/s, a strony internetowe były tworzone jedynie za pomocą języka HTML i nie zapewniały żadnej interakcji z użytkownikiem. Technologie webowe nieustannie idą do przodu — pojawiają się nowe rozwiązania, a dotychczasowe są stopniowo wypierane przez nowsze. W tej branży spoczęcie na laurach oznacza krok w tył, dlatego programista stale musi się rozwijać i rozbudowywać swój warsztat pracy.

Podręcznik składa się z czterech rozdziałów. W trzech pierwszych omówiono technologie webowe, a w czwartym pokazano sposób ich użycia.

Rozdział 1. przedstawia bibliotekę jQuery, która usprawnia manipulowanie elementami dokumentu HTML i obsługę zdarzeń oraz pozwala tworzyć animacje. Zastosowanie jQuery upraszcza i przyśpiesza pracę programisty, a także sprawia, że napisany kod jest kompatybilny z wieloma przeglądarkami internetowymi. Użycie omówionego w tym rozdziale frameworka Bootstrap sprawi zaś, że strona wyświetli się poprawnie na każdym urządzeniu, niezależnie od wielkości ekranu.

Rozdział 2. został w całości poświęcony frameworkowi Angular, za pomocą którego stworzysz aplikację webową. Lwią część rozdziału zajmuje omówienie składni języka TypeScript, który dzięki swoim zaletom (takim jak m.in. sprawdzanie poprawności typów i lepsza współpraca z edytorami kodu) idealnie nadaje się do tworzenia dużych aplikacji internetowych.

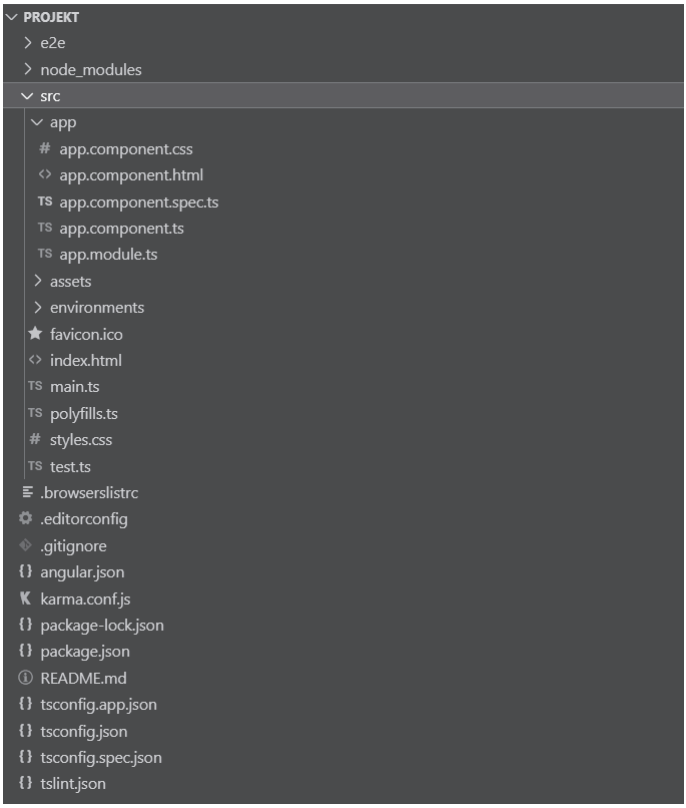
Rozdział 3. przedstawia platformę Node.js, która pozwala uruchomić kod JavaScript poza przeglądarką. Node.js w połączeniu z frameworkiem Express, również omówionym w tym rozdziale, otworzył nowe obszary zastosowań języka JavaScript, który może być wykorzystywany do tworzenia aplikacji działającej po stronie serwera. Ponadto w rozdziale znalazł się opis nierelacyjnej bazy danych MongoDB.

W ostatnim rozdziale wspólnie stworzymy projekt oparty na przedstawionych w podręczniku technologiach webowych.

Wszystkie liczące się witryny internetowe stawiają na interakcję. Użytkownik już nie jest widzem — jest uczestnikiem. Znajomość omówionych w podręczniku rozwiązań pozwoli tworzyć takie interaktywne strony.

2.3. Angular — pierwsze kroki

Po wygenerowaniu projektu struktura plików aplikacji jest następująca (rysunek 2.37):



Rysunek 2.37. Struktura plików aplikacji Angular

Wszystkie elementy użyte do zbudowania aplikacji Angular (szablony, style, obrazy, usługi) i niezbędne do jej działania znajdują się w katalogu *src*. Oto najważniejsze z nich:

- *app/app.component.(ts, html, css, spec.ts)* — są to cztery pliki o wspólnej nazwie *app.component* z rozszerzeniami podanymi w nawiasie. Pierwszy (*app.component.ts*) zawiera definicje głównego komponentu aplikacji, drugi jest szablonem HTML, trzeci arkuszem stylów CSS, a ostatni jest wykorzystywany do testów. Wraz z rozwojem aplikacji pojawią się tu pliki zagnieżdżonych komponentów.
- *app/app.module.ts* — ten plik określa konfigurację aplikacji.
- *assets/** — katalog z plikami wykorzystywanymi przez aplikację (obrazy, wideo, dźwięk).
- *index.html* — główna strona HTML aplikacji. Angular automatycznie dodaje do tego pliku wszystkie pliki *.js* i *.css* użyte podczas jej budowania.
- *node_modules/* — katalog utworzony przez Node.js. Zawiera wszystkie wymagane moduły używane przez aplikację.

- *angular.json* — konfiguracja dla Angular CLI.
- *package.json* — konfiguracja npm z listą pakietów użytych w aplikacji.
- *tsconfig.json* — konfiguracja kompilatora języka TypeScript.

Jednym z najważniejszych plików jest *app.component.ts* w katalogu *app*. Zawiera on definicję **komponentu** `AppComponent`, który jest „korzeniem" całej aplikacji.

Komponenty frameworka Angular to podstawowy budulec tworzonej aplikacji. Zarządzają szablonem i dostarczają do niego dane.

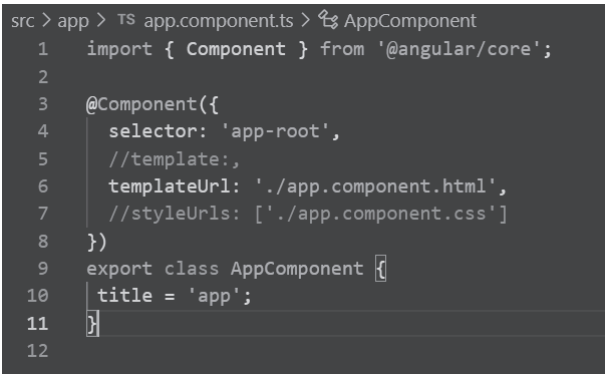
W pliku *app.component.ts* możemy wyróżnić trzy sekcje. Pierwsza zawiera polecenie `import` i odpowiada za wczytanie modułu `@angular/core`, który zarządza pracą komponentów. Druga, zaczynająca się od znaku `@`, to przykład tzw. **dekoratora**, dostarczającego informacji konfiguracyjnych za pomocą właściwości, którymi są: `selector`, `templateUrl` oraz `styleUrls`. Właściwość `selector`, której wartość odnajdziemy również w pliku *index.html* (umieszczoną w znacznikach <>), wskaże frameworkowi, że w tym miejscu za treść odpowiada komponent. To tutaj Angular wstrzyknie zawartość **szablonu komponentu**. Właściwości `templateUrl` oraz `styleUrls` wskazują frameworkowi Angular, gdzie znajduje się szablon powiązany z komponentem oraz formatujący go arkusz stylów.

Ostatnia sekcja to definicja klasy, która posłużyła do utworzenia komponentu (listing 2.60).

Listing 2.60. Domyślna zawartość pliku app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'projekt';
}
```

Zawartość wyświetlanego **szablonu** znajduje się w pliku *./app.component.html*, na co wskazuje znacznik `templateUrl`. Szablon zawiera informacje wyświetlane przez aplikację. Do budowy widoku używa się składni języka HTML oraz dodatków frameworka Angular. W polu `title` znajduje się nazwa projektu (rysunek 2.38).



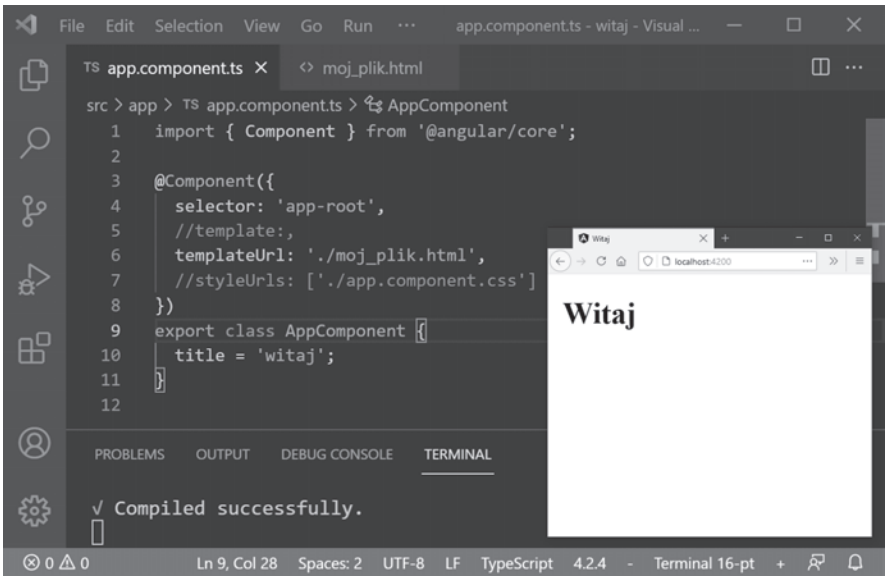
Rysunek 2.38. Zawartość pliku app.component.ts

2.3.1. Jednokierunkowe techniki wiązania danych

Interpolacja (string interpolation)

Utwórzmy nowy plik HTML o nazwie *moj_plik.html* i zmieńmy domyślną ścieżkę do pliku szablonu tak, aby wskazywała na niego.

W pliku pomiędzy znacznikami `<h1></h1>` został umieszczony tekst `Witaj`. Aby go wyświetlić, należy zmienić domyślną ścieżkę do szablonu na `templateUrl: './moj_plik.html'`, Po zapisaniu pliku następuje uaktualnienie okna przeglądarki (rysunek 2.39).



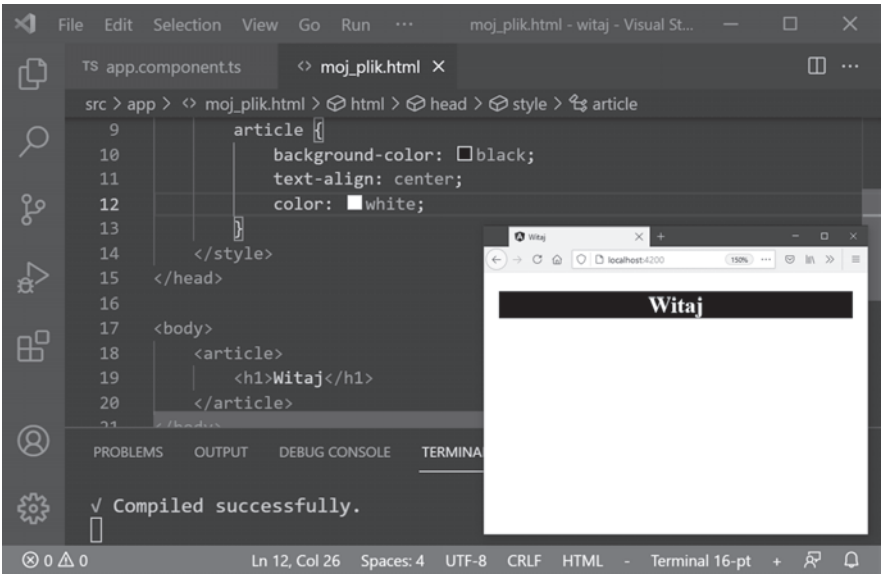
Rysunek 2.39. Zawartość pliku moj_plik.html

Plik ten oprócz znaczników języka HTML może zawierać style CSS.

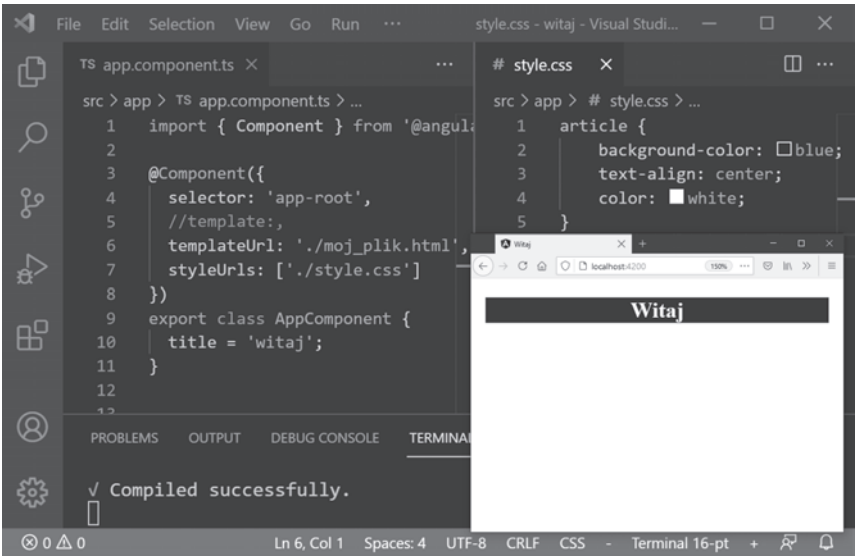
Nasz napis zostaje umieszczony pomiędzy znacznikami `<article></article>`, a zawartość jest formatowana za pomocą arkusza stylów, który zmienia kolor tła i tekstu oraz go wyśrodkowuje (rysunek 2.40).

Styl można zdefiniować w dowolnym pliku CSS. Aby to zademonstrować, w katalogu *app* utworzono plik o nazwie *style.css*. Podpięcie stylu, podobnie jak pliku HTML, następuje w pliku *app.component.ts* — po usunięciu znacznika komentarza z linijki zawierającej `styleUrls:` zmieniamy domyślną ścieżkę na: *./style.css* (rysunek 2.41).

Szablon może się znajdować także w pliku *app.component.ts*. W tym celu należy usunąć znacznik komentarza umieszczony przed `template` i pomiędzy znakami ``, `""` lub `''` zdefiniować zawartość.



Rysunek 2.40. Zawartość pliku moj_plik.html po zmianie — dodaniu stylu CSS



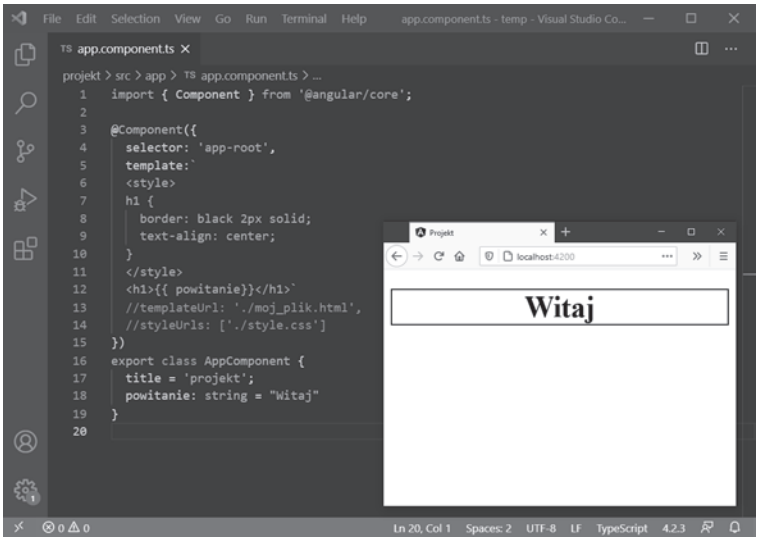
Rysunek 2.41. Definicja stylu CSS zapisanego we własnym pliku

WSKAZÓWKA

Pierwsze rozwiązanie, czyli umieszczenie zawartości szablonu pomiędzy znakami ``, jest najwygodniejsze, gdyż pozwala tworzyć wielowierszowe konstrukcje.

Dodajmy do naszego komponentu (jego definicja znajduje się w klasie `AppComponent`) pole `powitanie`, a jego wartość ustalmy na `Witaj`. Wartość zdefiniowanego w ten sposób pola można przekazać do szablonu — w tym celu należy umieścić **nazwę pola pomiędzy podwójnymi nawiasami klamrowymi (`{{}}`)**. Mechanizm ten nazywa się **interpolacją** lub **string interpolation**.

Pole to zostało umieszczone pomiędzy znacznikami `<h1></h1>`, a wygląd ustalono za pomocą stylu (obramowanie koloru czarnego o rozmiarze 2 px w postaci linii ciągłej i z wyśrodkowanym tekstem), którego definicja znajduje się w tym samym pliku (rysunek 2.42).



Rysunek 2.42. Umieszczenie wartości pola w szablonie — interpolacja

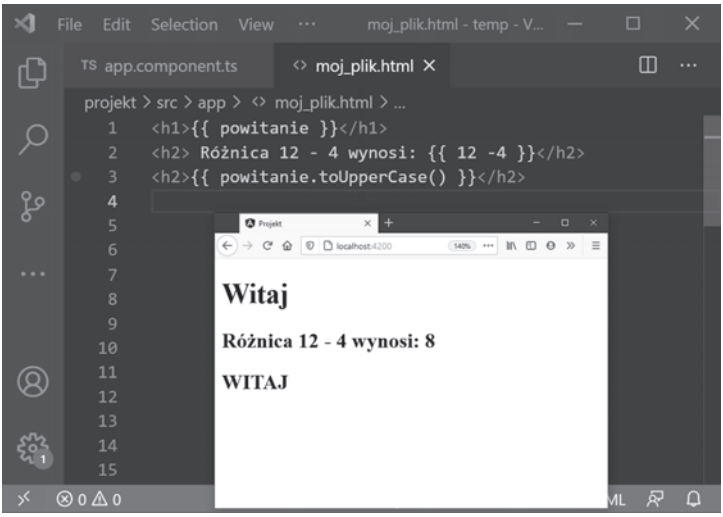
Oczywiście tego typu wstrzyknięcie zawartości komponentu zadziała także wtedy, gdy szablon znajduje się w zewnętrznym pliku HTML.

Wewnątrz podwójnych nawiasów klamrowych oprócz nazw pól komponentów można również umieścić **wyrażenia** (różnica liczb 12 i 4), a także wywoływać **metody** (wartość przypisana do pola `powitanie` jest wyświetlana dużymi literami w wyniku użycia metody `toUpperCase`) (rysunek 2.43).

Wiązanie właściwości (property binding)

Jak już wiesz, elementy w szablonie można umieszczać przez ich bezpośrednie definiowanie, a także z wykorzystaniem interpolacji. Jest jeszcze jeden sposób: **wiązanie właściwości** (ang. *property binding*).

Użycie wszystkich metod jest pokazane na listingu 2.61.



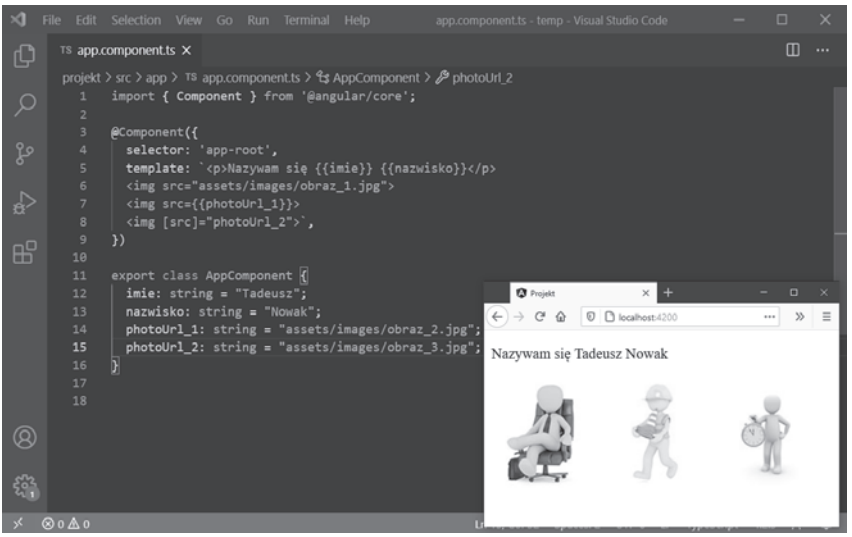
Rysunek 2.43. Zastosowanie interpolacji

Listing 2.61. Sposoby umieszczenia danych w szablonie

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<p>Nazywam się {{imie}} {{nazwisko}}</p>
    <img src="assets/images/obraz_1.jpg">
    <img src={{photoUrl_1}}>
    <img [src]="photoUrl_2">`,
})

export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  photoUrl_1: string = "assets/images/obraz_2.jpg";
  photoUrl_2: string = "assets/images/obraz_3.jpg";
}
```

Użycie zapisu `<img [src]="photoUrl_2">` mówi frameworkowi Angular, że na elemencie `img` istnieje właściwość `src` (zapisujemy ją pomiędzy nawiasami kwadratowymi), której wartość powinna zostać pobrana z pola `photoUrl_2`.

Wynikiem wywołania kodu jest umieszczenie w widoku napisu `Nazywam się Tadeusz Nowak` oraz trzech zdjęć z katalogu *./assets/images* (rysunek 2.44).



Rysunek 2.44. Sposoby umieszczania danych w szablonie

Tych mechanizmów będziemy używali w zależności od typu przekazywanych wartości. **Interpolacja nie nadaje się do przekazywania argumentów np. do funkcji, które nie są typu `string`.**

Przykład z listingu 2.62 pokazuje zastosowanie metody wiązania właściwości.

Listing 2.62. Przykłady użycia mechanizmu wiązania właściwości

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <!--przykład pierwszy-->
    <p [style.background-color]="kolor" [style.text-align]="jak"> Nazywam się
{{imie}} {{nazwisko}}</p>
    <!--przykład drugi-->
    <img src="assets/images/obraz_1.jpg" [class.zdjecie]="aktywna">
  `,
  styles: ['.zdjecie {border: 2px solid #000; display: block; margin: auto;}']
})

export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
}
```

Pierwszy przykład pokazuje, jak za pomocą wiązania właściwości zmienić styl elementu. W tym celu zdefiniowano dwie właściwości: `kolor` o wartości `yellow` oraz `jak` o wartości `center`. Właściwości te zostały użyte do zmiany wyglądu akapitu, w którym znajduje się napis `Nazywam się Tadeusz Nowak` — kolor tła zmienił się na żółty, a tekst jest teraz wycentrowany. Zapis `<p [style.background-color]="kolor" [style.text-align]="jak">` powoduje zmianę stylu.

Drugi przykład dodaje nową klasę, czego efektem jest zmiana wyglądu zdjęcia. Do znacznika `img` zostaje dodany kod `[class.zdjecie]="aktywna"`, który inicjuje klasę (przekazywana jest wartość `true`). Definicja stylu klasy powinna być umieszczona w selektorze `styles`. Pole to uaktywnia lokalny arkusz stylu, który jest definiowany bezpośrednio w pliku komponentu. Użyte znaczniki CSS powodują dodanie czarnego obramowania zdjęcia o szerokości 2 px oraz jego wyśrodkowanie.

Wiązanie zdarzeń (event binding)

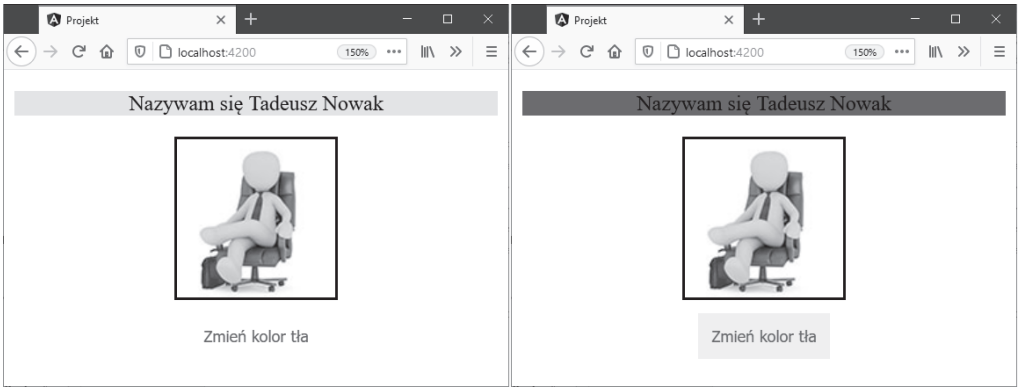
W Angularze **wiązanie zdarzeń** (ang. *event binding*) jest wykorzystywane do obsługi zdarzeń. Zaliczamy do nich np. kliknięcie przycisku myszy, ruch wskaźnika myszy i zmianę wartości tekstowej. Taka interakcja powoduje wywołanie w komponencie określonej metody.

Przykład zdarzenia jest pokazany na listingu 2.63. Za pomocą znacznika `<button>` zostaje dodany przycisk, którego kliknięcie powoduje (typ zdarzenia umieszczamy w nawiasie okrągłym) wykonanie metody `zmienKolor()` — zmianę koloru tła pod napisem. Wraz z przyciskiem zostają zdefiniowane dwie klasy: pierwsza określa jego wygląd, a druga — zachowanie po nakierowaniu na niego wskaźnika myszy (pseudoklasa `:hover`). Metoda `zmienKolor()` zostaje zdefiniowana w komponencie — odpowiada ona za zmianę koloru tła akapitu (rysunek 2.45).

Listing 2.63. Wiązanie zdarzeń

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{imie}} {{nazwisko}}</p>
    <img src="assets/images/obraz_1.jpg" [class.zdjecie]="aktywna">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>`,
  styles: [`
    .zdjecie {border: 2px solid #000; display: block; margin: auto;}
    .przycisk {border: none; color: green; background-color: inherit; padding:
10px; font-size: 12px; cursor: pointer; display: block; margin: 10px auto;}
    .przycisk:hover {background: #eee;}
  `]
})
```

```
export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```



Rysunek 2.45. Użycie wiązania zdarzeń

Zastosowany w metodzie `zmienKolor()` zapis `this.kolor = this.kolor === "yellow" ? "green" : "yellow";` to tzw. **wyrażenie warunkowe**. Oznacza ono: jeżeli warunek jest prawdziwy, przypisz do `this.kolor` wartość `green`, w przeciwnym razie użyj `yellow`. Należy je traktować jako skróconą wersję instrukcji warunkowej `if`.

Podane przykłady to jednokierunkowe techniki wiązania danych, wykorzystywane do umieszczania w szablonie (widoku) danych pochodzących z kodu TypeScript.

UWAGA

Zazwyczaj w definicji zdarzenia podaje się nazwę metody, która ma zostać użyta po zaistnieniu zdarzenia, ale nic nie stoi na przeszkodzie, aby zamiast nazwy metody umieścić tu jej definicję. Oznacza to, że kod `this.kolor = this.kolor === "yellow" ? "green" : "yellow";` może się znajdować bezpośrednio w szablonie. Jednak to, co dozwolone, nie zawsze jest zalecane — podczas analizy kodu (bądź gdy trzeba do niego wrócić po jakimś czasie) pierwszy wariant wydaje się lepszym podejściem.

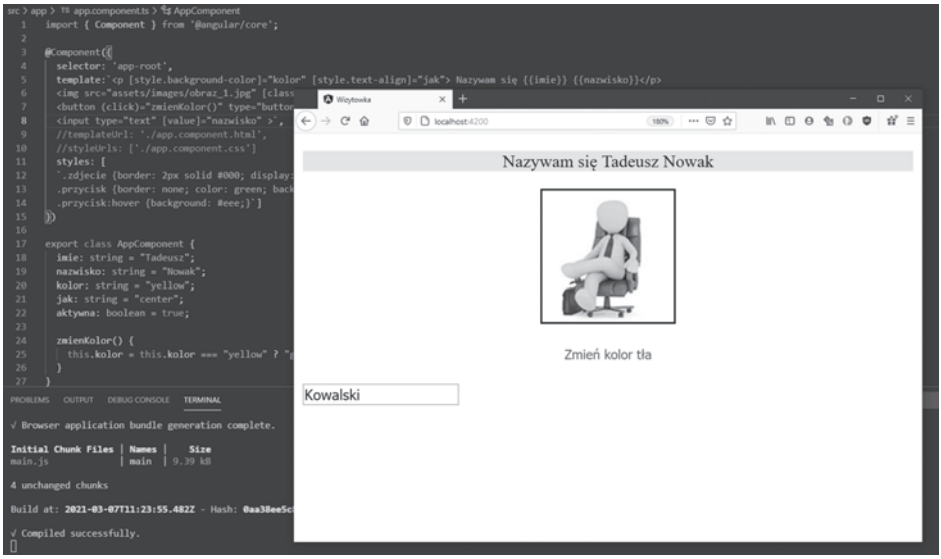
Zadanie 2.20.

Przygotuj trzy pliki: prosty szablon, arkusz stylów, który go formatuje, i swoje zdjęcie. Za pomocą wybranych technik wiązań jednokierunkowych stwórz prostą wizytówkę.

2.3.2. Wiązanie dwukierunkowe (two-way binding)

Wiązanie jednokierunkowe nie pozwala na odzwierciedlenie zmian szablonu w kodzie TypeScript. Dlatego też framework Angular został wyposażony w tzw. **mechanizm wiązania dwukierunkowego** (ang. *two-way binding*), którego zadaniem jest przekazywanie danych pochodzących z komponentu do widoku i na odwrót. Synchronizacja ta następuje automatycznie.

Na przykład dodanie do szablonu linijki `<input type="text" [value]="nazwisko">` spowoduje wyświetlenie pola formularza z wartością `Nowak`, lecz jakakolwiek jego zmiana nie doprowadzi do aktualizacji właściwości `nazwisko` (rysunek 2.46).



Rysunek 2.46. Dodanie pola formularza

Aby to zmienić, należy utworzyć **wiązanie dwukierunkowe** według wzoru:

```
[(ngModel)] = "właściwość zdefiniowana w komponencie"
```

Najpierw używamy słowa `ngModel`, umieszczonego pomiędzy nawiasami okrągłymi i kwadratowymi, a następnie, po znaku `=`, wpisujemy nazwę właściwości zdefiniowanej w komponencie. Ponieważ wiązanie dwukierunkowe jest **połączeniem wiązania właściwości oraz wiązania zdarzeń**, jego składnia zawiera parę nawiasów kwadratowych oraz okrągłych (listing 2.64).

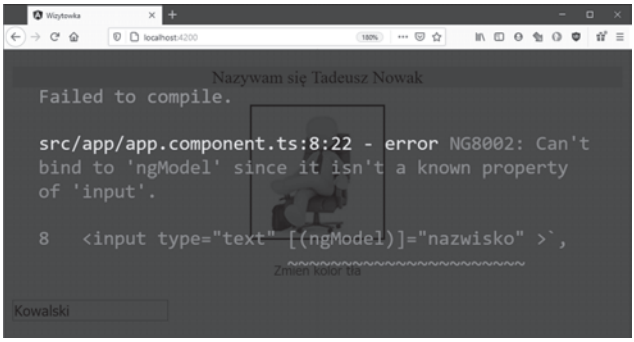
WSKAZÓWKA

Aby zapamiętać kolejność stosowania nawiasów, wyobraź sobie banany w skrzynce.

Listing 2.64. Użycie wiązania dwustronnego — plik app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{imie}} {{nazwisko}}</p>
    <img src="assets/images/obraz_1.jpg" [class.zdjecie]="aktywna">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <input type="text" [(ngModel)]="nazwisko" >`,
  styles: [`
    .zdjecie {border: 2px solid #000; display: block; margin: auto;}
    .przycisk {border: none; color: green; background-color: inherit; padding:
10px; font-size: 12px; cursor: pointer; display: block; margin: 10px auto;}
    .przycisk:hover {background: #eee;}
  `]
})

export class AppComponent {
  imie: string = "Tadeusz";
  nazwisko: string = "Nowak";
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Zapisanie projektu wywoła błąd kompilacji. Angular nie umie rozpoznać, czym jest `ngModel` (rysunek 2.47).



Rysunek 2.47. Błąd — nieznana właściwość ngModel

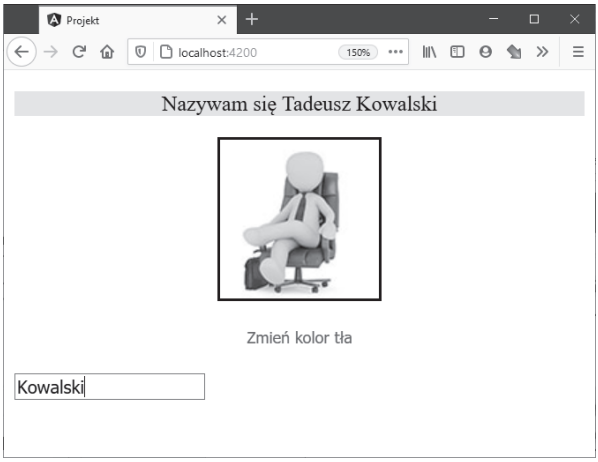
Użyty moduł odpowiada za obsługę formularzy (powrócimy do niego w podrozdziale 2.7) i w domyślnej konfiguracji nie jest importowany, tak więc aby móc skorzystać z tego typu wiązań, musimy **włączyć dyrektywę ngModel**. Jest ona zależna od pakietu `FormsModule`, którego definicja musi się znaleźć w pliku modułu głównego *app.module.ts* (listing 2.65).

W pliku *app.module.ts* importujemy identyfikator `FormsModule` z biblioteki *angular/forms*, a następnie w tablicy `imports` dopisujemy `FormsModule`.

Listing 2.65. Zawartość pliku app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Po zapisaniu zmian i skompilowaniu kodu wiązanie dwustronne działa. Zmiana zawartości pola `input` powoduje aktualizację właściwości `nazwisko` (rysunek 2.48).



Rysunek 2.48. Wiązanie dwustronne

Ponieważ rozwijanego kodu użyjemy w następnym rozdziale, wprowadźmy w nim pewne modyfikacje.

Dwa przedstawione poniżej listingi, 2.66 oraz 2.67, nie zmieniają w żaden sposób funkcjonalności przykładu, lecz jedynie go porządkują — część właściwości zostaje zgrupowana w jeden obiekt, którego kształt jest określany przez interfejs zdefiniowany w pliku *interfejs.ts*.

Listing 2.66. Zawartość pliku app.component.ts

```
import { Component } from '@angular/core';
import { Osoba } from './interfejs';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <img src={{osoba.zdjecie}} [class.zdjecie]="aktywna">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <input type="text" [(ngModel)]="osoba.nazwisko">`,
  styleUrls: ['./app.component.css']
 })

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg"
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Listing 2.67. Zawartość pliku interfejs.ts

```
export interface Osoba {
    imie: string;
    nazwisko: string;
    zdjecie: string;
}
```

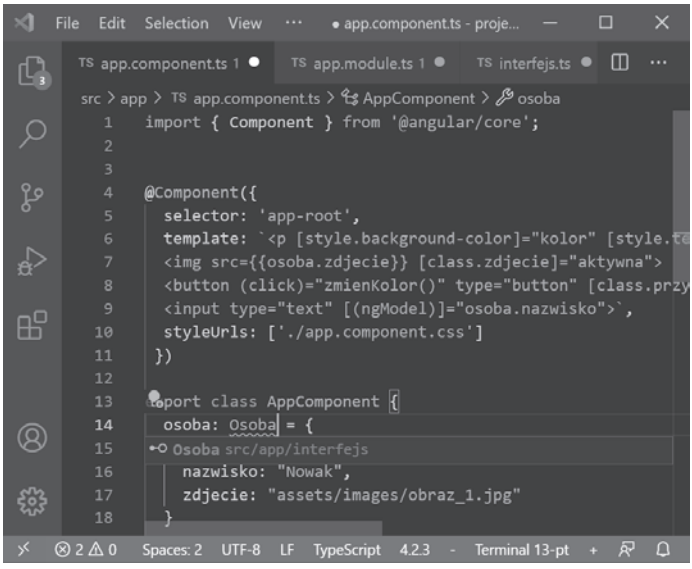
W pliku *interfejs.ts* został utworzony interfejs o nazwie `Osoba` (zgodnie z opisem zamieszczonym w punkcie 2.2.6). Plik ten ma trzy pola: `imie`, `nazwisko` oraz `zdjecie`, wszystkie typu `string`. Ponieważ pola tego interfejsu mają być dostępne w pliku *app.component.ts*, przed jego definicją zostało umieszczone słowo `export`.

Definicja interfejsu `Osoba` zostanie wykorzystana w pliku *app.component.ts*, dlatego interfejs musi zostać do niego zaimportowany. Aby to zrobić, należy dodać w kodzie linijkę `import { Osoba } from './interfejs';`. W nawiasach klamrowych określamy nazwy importowanych elementów, a po słowie `from` — plik, z którego pochodzą. Oba pliki znajdują się w tym samym katalogu, dlatego nazwa pliku rozpoczyna się od znaków `./`, po których następuje nazwa pliku (opuszczamy rozszerzenie).

Zaimportowany interfejs `Osoba` posłużył do utworzenia obiektu `osoba`. Ponieważ interfejs określa **kształt obiektu**, wszystkie pola obecne w interfejsie muszą mieć odzwierciedlenie w obiekcie, do którego interfejs został przypisany. Ze względu na to, że już nie odwołujemy się do zmiennej zadeklarowanej w komponencie, lecz do obiektu, wszystkie wywołania należy poprzedzić **nazwą obiektu.** Dlatego np. musiał się zmienić zapis — z `imie` na `osoba.imie` itd.

Jeśli nie wykonamy importu, podczas kompilacji nastąpi błąd. Visual Studio Code wspomaga nas w tym zadaniu — wystarczy, że klikniemy element, o którym wiemy, że pochodzi z innego pliku, i wybierzemy skrót *Ctrl+I*, a wtedy edytor podpowie, w jakim pliku znajduje się jego definicja (rysunek 2.49). Innym sposobem jest użycie **szybkiego rozwiązania**, tzw. *Quick Fix*, dostępnego w opisie błędu. Kliknięcie go wprowadzi w nagłówku pliku stosowne instrukcje.

Dodatkowo definicja lokalnego stylu została przeniesiona do pliku *./app.component.css*.



Rysunek 2.49. Użycie skrótu Ctrl+I

Pytania kontrolne

1. W jaki sposób przekazać dane do szablonu?
2. Jakie są ograniczenia jednokierunkowych technik wiązania danych?
3. Czy wiązania dwukierunkowe pozwalają pokonać te ograniczenia?

2.4. Dyrektywy

Dyrektywy wbudowane we framework Angular umożliwiają dostęp do struktury dokumentu. Z wykorzystaniem dyrektyw możemy **zmieniać zarówno wygląd, jak i zachowanie danego elementu**. Angular zapewnia wiele wbudowanych dyrektyw (rozpoczynają się one od prefiksu `ng`), ale także pozwala tworzyć własne.

Dyrektywy w Angularze dzielimy na trzy kategorie, w zależności od ich zachowania:

- **komponenty** (ang. *Component Directives*),
- **dyrektywy strukturalne** (ang. *Structural Directives*),
- **dyrektywy atrybutowe** (ang. *Attribute Directives*).

Definiowanie komponentu rozpocząłeś (nieświadomie) w poprzednim rozdziale; w następnym powrócimy do tego tematu.

2.4.1. Dyrektywy strukturalne

Dyrektywy strukturalne zmieniają strukturę drzewa dokumentów przez dodawanie, usuwanie lub modyfikowanie jego elementów, a co za tym idzie, mają **wpływ na strukturę kodu HTML oraz jego układ**.

Dyrektywa ta zostaje przypisana do konkretnego elementu i łatwo ją rozpoznać, ponieważ poprzedza się ją znakiem gwiazdki (`*`).

Dyrektywa ngIf

Dyrektywa `ngIf` jest używana do **dodawania lub usuwania elementów HTML**. Jeżeli wartość to **false**, element jest usuwany z drzewa dokumentów, a jeśli **true** — jest w nim uwzględniany.

UWAGA

Dyrektywa `ngIf` nie ukrywa elementu, ale nie uwzględnia go w drzewie dokumentów.

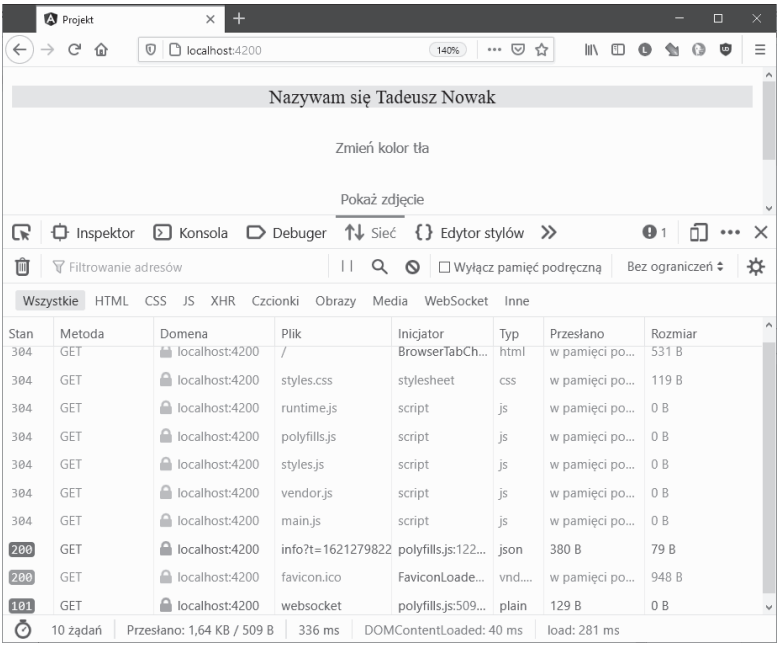
Abyś zobaczył, jak działa dyrektywa `ngIf`, zmodyfikujmy listing 2.67 i ukryjmy zdjęcie. Będzie można je wyświetlić przez wciśnięcie przycisku (listing 2.68).

Został dodany nowy przycisk z etykietą `Pokaż zdjęcie`. Kliknięcie go uruchamia zdarzenie — zmianę wartości zmiennej `pokazZdjecie` z `false` na `true` bądź odwrotnie. Domyślna wartość zmiennej `pokazZdjecie` jest ustalona na `false`. Dyrektywa `ngIf` zostaje zdefiniowana w znaczniku `<img>` i powiązana z `pokazZdjecie` (deklarację rozpoczynamy od gwiazdki). Dodatkowo został dodany akapit, który wyświetla bieżącą wartość zmiennej `pokazZdjecie`. Po skompilowaniu kodu zdjęcie nie zostaje załadowane.

Listing 2.68. Dyrektywa ngIf

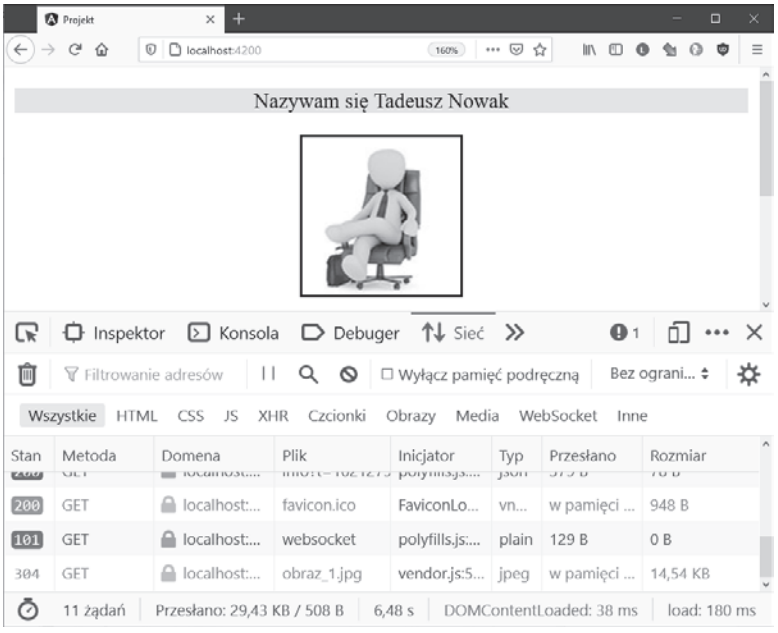
```
import { Component } from '@angular/core';
import { Osoba } from './interfejs';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie">
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <button (click)="pokazZdjecie=!pokazZdjecie" type="button" [class.
przycisk]="aktywna">Pokaż zdjęcie </button>
    <p> Obecny status właściwości 'pokazZdjecie' = {{pokazZdjecie}} </p>`,
    styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg"
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Aby zweryfikować, czy obraz **nie jest uwzględniany** w drzewie dokumentu, należy uruchomić narzędzia deweloperskie przeglądarki (klawiszem *F12*) i wybrać zakładkę *Sieć*. Jak można się przekonać, na liście żądań *GET* nie ma tego, które nakazuje pobranie obrazka (rysunek 2.50).



Rysunek 2.50. Sposób działania dyrektywy nglf

Żądanie wczytania zdjęcia (metoda *GET*) pojawia się w chwili kliknięcia przycisku *Pokaż zdjęcie* (rysunek 2.51).



Rysunek 2.51. Sposób działania dyrektywy nglf

Taki sposób działania dyrektywy powoduje, że jej umiejętne wykorzystanie znacznie wpływa na szybkość działania aplikacji — **jej poszczególne elementy są wczytywane, w miarę jak występują określone zdarzenia**.

Możliwość ukrywania i wyświetlania zdjęcia możemy zapewnić również za pomocą pola typu **checkbox**. Zaznaczenie pola (przekazanie wartości `true`) skutkuje pojawieniem się zdjęcia, natomiast jego brak (przekazanie wartości `false`) — ukryciem. Zmodyfikowany program jest pokazany na listingu 2.69.

Do powiązania pola `checkbox` ze zmienną `pokazZdjecie` została użyta dyrektywa `ngModel`.

Listing 2.69. Użycie dyrektywy ngIf w połączeniu z polem typu checkbox

```
import { Component } from '@angular/core';
import { Osoba } from './interfejs';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna" *ngIf="pokazZdjecie"
>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <button (click)="pokazZdjecie=!pokazZdjecie" type="button" [class.
przycisk]="aktywna">Pokaż zdjęcie </button>
    <label><input type="checkbox" [(ngModel)]=pokazZdjecie>Pokaż zdjęcie </
label>
    <p> Obecny status właściwości 'pokazZdjecie' = {{pokazZdjecie}} </p>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg"
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Dyrektywa `ngIf` może zostać połączona z klauzulą `else` — określa, co ma się wydarzyć w przypadku, gdy stan będzie miał wartość `false`. Jest to zobrazowane w przykładzie poniżej.

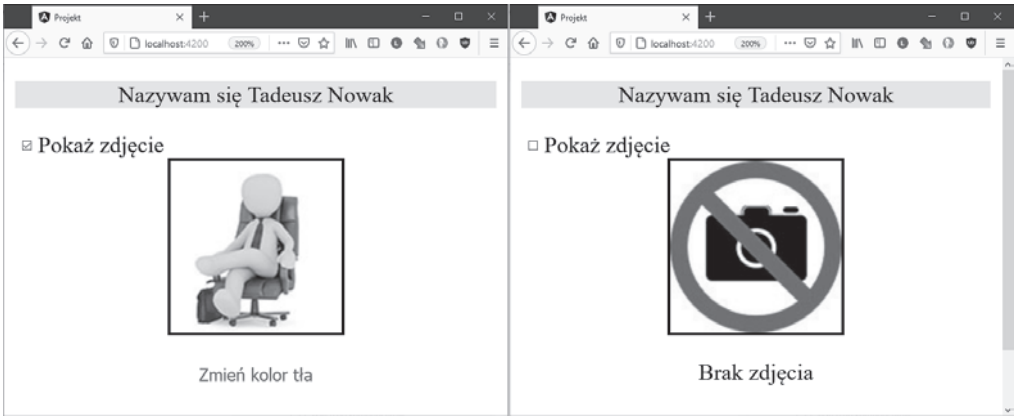
Zapis `*ngIf="pokazZdjecie else brakZdjecia"` oznacza, że jeśli warunek nie będzie spełniony, ma zostać wykonany kod zawarty w **zmiennej szablonowej** `brakZdjecia`. Zmienna ta została zdefiniowana wraz ze znacznikami `<ng-template></ng-template>`, pomiędzy którymi jest zawarta instrukcja — wyświetlenie zdjęcia wraz z podpisem `Brak zdjęcia`. Znaczniki te należy traktować jako **kontener** z kodem, który zostanie wykonany, jeśli nastąpi zdefiniowane zdarzenie. **Gdyby go nie było, kod zostałby zinterpretowany i wyświetlony natychmiast.** Nazwa zmiennej szablonowej musi być poprzedzona znakiem *hash* (`#`). Do określenia ścieżki zdjęcia użyto właściwości `bezZdjecia`. Właściwość znajduje się w obiekcie `osoba`, a jak pamiętasz, obiekt ten został utworzony za pomocą interfejsu, tak więc definicja właściwości również musi się znaleźć w opisie interfejsu. Dodajemy zatem linijkę: `bezZdjecia: string;`. Do pliku *app.component.css* zostaje dodana nowa klasa, `.tekst_srodek`, która wyrównuje tekst. Zmiany w kodzie pokazuje listing 2.70.

Listing 2.70. Dyrektywa ngIf w połączeniu z else

```
import { Component } from '@angular/core';
import { Osoba } from './interfejs';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <label><input type="checkbox" [(ngModel)]="pokazZdjecie">Pokaż zdjęcie </
label>
    <div id="ramka">
        <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie else brakZdjecia">
    </div>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <ng-template #brakZdjecia>
        <img src="{{osoba.bezZdjecia}}" class="zdjecie">
        <p class="tekst_srodek">Brak zdjęcia</p>
    </ng-template>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg",
    bezZdjecia: "assets/images/nofoto.jpg"
  }
```

```
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Wynik działania kodu jest pokazany na rysunku 2.52.



Rysunek 2.52. Użycie dyrektywy nglf w połączeniu z else. Po lewej — pole checkbox zaznaczone, po prawej — zaznaczenie pola checkbox usunięte

Zadanie 2.21.

Przygotuj krótki opis miejscowości, w której mieszkasz, oraz dołącz do niego dwa zdjęcia. Opis i zdjęcia wykorzystaj do stworzenia szablonu. Dodatkowo pod opisem umieść dwa przyciski i z użyciem dyrektywy `ngIf` spraw, aby kliknięcie każdego z nich powodowało wyświetlenie innego zdjęcia.

Dyrektywa ngFor

Dyrektywa `*ngFor` pozwala na powtarzanie części szablonu HTML dla iterowanego obiektu.

Podstawowa składnia jest następująca:

```
*ngFor="let element of kolekcja;"
```

Przykład użycia tej dyrektywy jest pokazany na listingu 2.71.

Obiekt `osoba` został wzbogacony o tabelę zawierającą adres. Pod pierwszym indeksem jest zapisana `ulica`, pod drugim `kod pocztowy`, pod trzecim zaś `miasto`. Aby można było umieścić te dane w pliku *app.component.ts*, należy zmodyfikować definicję interfejsu, który znajduje

się w pliku *interfejs.ts*, tzn. dodać linijkę: `adres: string[];` (dane będą przechowywane w tablicy). Za wyświetlenie tych informacji odpowiada dyrektywa `ngFor`, zadeklarowana w znaczniku `<li>`. Nakazuje ona pobranie wartości z tablicy `osoba.adres` i przypisanie jej do zmiennej `value`. Wartość zmiennej jest następnie wypisywana za pomocą interpolacji. Elementy tabeli zawierające adres są wyświetlane jako lista punktowana.

Listing 2.71. Użycie dyrektywy ngFor

```
import { Component } from '@angular/core';
import { Osoba } from './interfejs';

@Component({
  selector: 'app-root',
  template: `<p [style.background-color]="kolor" [style.text-align]="jak">
Nazywam się {{osoba.imie}} {{osoba.nazwisko}}</p>
    <label><input type="checkbox" [(ngModel)]="pokazZdjecie">Pokaż zdjęcie </
label>
    <div id="ramka">
        <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie else brakZdjecia">
    </div>
    <div id="kontakt" class="tekst">
        <p>Kontakt: </p>
        <ul>
            <li *ngFor="let value of osoba.adres"> {{value}} </li>
        </ul>
      </div>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <ng-template #brakZdjecia>
        <img src="{{osoba.bezZdjecia}}" class="zdjecie">
        <p class="tekst_srodek">Brak zdjęcia</p>
    </ng-template>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg",
    bezZdjecia: "assets/images/nofoto.jpg",
    adres: [
      "Piękna 16",
      "40-003",
      "Katowice"
    ]
  }
```

```
  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

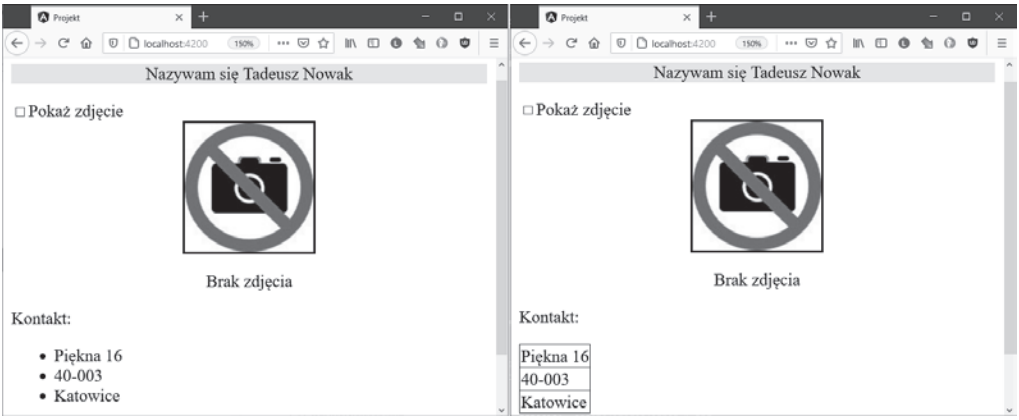
Zamiana kodu objętego znacznikiem `div` o identyfikatorze `kontakt` na:

```
<table>
    <tr *ngFor="let item of osoba.adres">
        <td>{{item}}</td>
    </tr>
</table>
```

spowoduje wyświetlenie adresu w tabeli. Oczywiście za wyświetlenie obramowania tabeli odpowiada styl CSS.

```
table, td {
    border: 1px solid black;
    border-collapse: collapse;
}
```

Efekt zastosowania obu wersji kodu jest pokazany na rysunku 2.53.



Rysunek 2.53. Użycie dyrektywy ngFor. Po lewej — lista wypunktowana, po prawej — tabela

Działaniem dyrektywy `ngFor` możemy sterować za pomocą zmiennych pokazanych w tabeli 2.2.

Tabela 2.2. Zmienne dyrektywy ngFor

Nazwa zmiennej	Opis
`index`	Wartość typu `number` określająca położenie bieżącego obiektu
`odd`	Wartość logiczna; `true` oznacza, że położenie bieżącego obiektu jest określone liczbą nieparzystą
`even`	Wartość logiczna; `true` oznacza, że położenie bieżącego obiektu jest określone liczbą parzystą
`first`	Wartość logiczna; `true` oznacza, że bieżący obiekt jest pierwszym
`last`	Wartość logiczna; `true` oznacza, że bieżący obiekt jest ostatnim

Użycie wszystkich zmiennych dyrektywy jest przedstawione w kodzie poniżej (między znacznikami `div` o identyfikatorze `kontakt`). Został utworzony szereg zmiennych: zmienna `i` to `index`, zmienna `parzysta` przyjmuje wartość `true`, gdy w źródle danych położenie zostało określone liczbą parzystą (w omawianym scenariuszu jest to parzysty wiersz tabeli), a zmienne `pierwszy` i `ostatni` określają, odpowiednio, pierwszy i ostatni element tabeli. Zależnie od wartości zmiennych do wiersza tabeli zostaje przypisana odpowiednia klasa (listing 2.72).

Listing 2.72. Zastosowanie zmiennych dyrektywy ngFor

```
<div id="kontakt" class="tekst">
    <table>
        <tr *ngFor="let item of osoba.adres; let i = index; let parzysta =
even; let pierwszy = first; let ostatni = last"
            [class.parzysta]="parzysta" [class.nieparzysta]="!parzysta" [class.
wyroznienie]="pierwszy || ostatni">
            <td>{{i+1}}</td>
            <td>{{item}}</td>
        </tr>
    </table>
</div>
```

Efektem jest zastosowanie różnego formatowania wierszy tabeli w zależności od tego, czy dany wiersz jest parzysty (kolor żółty — `.parzysta { background-color: yellow; }`), czy nieparzysty (kolor niebieski — `.nieparzysta { background-color: blue; }`), ale także od tego, czy jest wierszem pierwszym, czy ostatnim (efekt negatywu — `.wyroznienie {background-color: black; color: white; }`). Zmienna `i` (zwiększana o 1) numeruje kolejne wiersze tabeli (rysunek 2.54).

Rysunek 2.54. Użycie zmiennych dyrektywy ngFor

UWAGA

Pamiętaj, że indeksowanie tabeli rozpoczynamy od 0, dlatego wyświetlony numer wiersza nie odpowiada jego faktycznemu indeksowi. Można by sądzić, że wiersz wyświetlający nazwę miasta powinien być koloru niebieskiego, ponieważ jak wskazuje wyświetlona wartość, jest on nieparzysty. W rzeczywistości wartość `Katowice` jest przechowywana w tabeli pod indeksem `2`.

Dyrektywa ngSwitch

W Angularze **ngSwitch** jest dyrektywą strukturalną, która jest używana do dodawania/usuwania elementów w drzewie dokumentów. Sposobem działania przypomina instrukcję `switch` dostępną w innych językach programowania, takich jak C++ i PHP. Przyłączanie odbywa się na podstawie zdefiniowanego wyrażenia i dopasowania warunków.

Ogólna składnia dyrektywy jest następująca:

```
<znacznik [ngSwitch]="wyrażenie">
    <znacznik_wewnętrzny *ngSwitchCase="wartość_1">instrukcja_1</ znacznik_
wewnętrzny>
    <znacznik_wewnętrzny *ngSwitchCase="wartość_2">instrukcja_2</znacznik_
wewnętrzny >
     .
     .
```

```
    <znacznik_wewnętrzny *ngSwitchCase="wartość_n">instrukcja_n</znacznik_
wewnętrzny>
    <znacznik_wewnętrzny *ngSwitchDefault>instrukcja_domyślna</znacznik_
wewnętrzny>
</znacznik>
```

Kod z listingu 2.73 przedstawia sposób użycia dyrektywy `ngSwitch`. Jej zadaniem jest wyświetlenie znaku obok nazwiska w zależności od płci: kobieta — znak Wenus, mężczyzna — znak Marsa, płeć nieokreślona — gwiazdka. Dyrektywa `ngSwitch` została połączona z wyliczeniem zdefiniowanym w pliku *interfejs.ts* (musimy pamiętać o wyeksportowaniu go i zaimportowaniu do pliku, w którym znajduje się szablon). Wyliczenie wraz z modyfikacją interfejsu `Osoba` jest pokazane na listingu 2.74. Sprawdzane wyrażenie umieszczono pomiędzy znacznikami `<ng-container></ng-container>`. Stosujemy je, gdy do szablonu nie chcemy wprowadzać żadnych dodatkowych obiektów języka HTML. Ponieważ wyliczenie (zobacz punkt 2.2.3) zdefiniowanym nazwom przypisuje wartość, wykorzystano to do sterowania dyrektywą `ngSwitch`. Za pomocą instrukcji `*ngSwitchCase` określamy, co ma się stać, gdy warunek zostanie spełniony. Wartość `1` spowoduje wyświetlenie piktogramu *wenus.png*, a wartość `2` — *mars.png*. Domyślnie numerowanie nazw wyliczenia rozpoczyna się od `0`, dlatego w jego definicji przy pierwszym elemencie dopisano `=1`. Ponowna zmiana definicji interfejsu wymusiła zmianę kształtu obiektu `osoba`. Obiekt poszerzono o właściwość `type`, która określa (lub nie) płeć.

Listing 2.73. Użycie dyrektywy ngSwitch

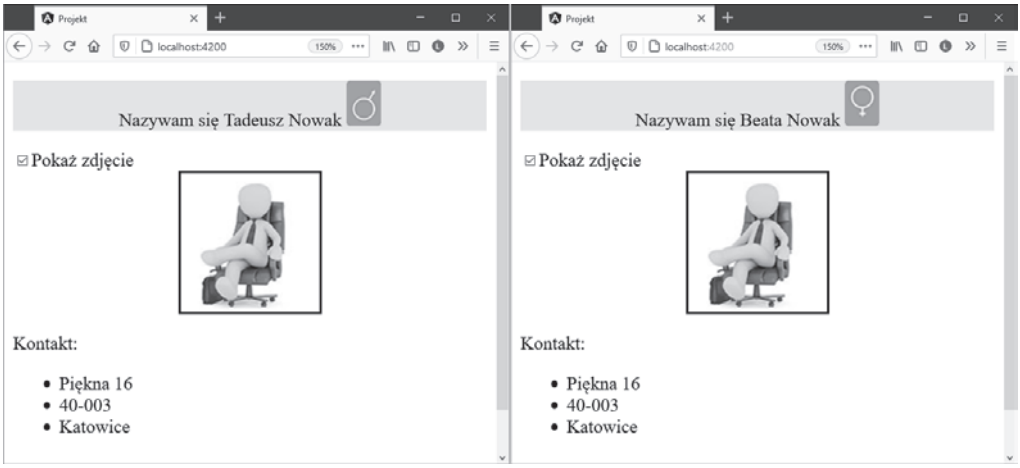
```
import { Component } from '@angular/core';
import { Osoba, Plec } from './interfejs';

@Component({
  selector: 'app-root',
  template: `
    <p [style.background-color]="kolor" [style.text-align]="jak"> Nazywam się
{{osoba.imie}} {{osoba.nazwisko}}
        <ng-container [ngSwitch]="osoba.type">
            <img src="assets/images/wenus.png" *ngSwitchCase=1>
            <img src="assets/images/mars.png" *ngSwitchCase=2>
            <img src="assets/images/unnamed.png" *ngSwitchDefault>
        </ng-container>
    </p>
    <label><input type="checkbox" [(ngModel)]="pokazZdjecie">Pokaż zdjęcie </
label>
    <div id="ramka">
        <img src="{{osoba.zdjecie}}" [class.zdjecie]="aktywna"
*ngIf="pokazZdjecie else brakZdjecia">
    </div>
    <div id="kontakt" class="tekst">
        <p>Kontakt: </p>
        <ul>
            <li *ngFor="let value of osoba.adres"> {{value}} </li>
        </ul>
```

```
    </div>
    <button (click)="zmienKolor()" type="button" [class.
przycisk]="aktywna">Zmień kolor tła</button>
    <ng-template #brakZdjecia>
        <img src="{{osoba.bezZdjecia}}" class="zdjecie">
        <p class="tekst_srodek">Brak zdjęcia</p>
    </ng-template>`,
  styleUrls: ['./app.component.css']
})

export class AppComponent {
  osoba: Osoba = {
    imie: "Tadeusz",
    nazwisko: "Nowak",
    zdjecie: "assets/images/obraz_1.jpg",
    bezZdjecia: "assets/images/nofoto.jpg",
    adres: [
      "Piękna 16",
      "40-003",
      "Katowice"
    ],
    type: Plec.niezdefiniowana
  }

  kolor: string = "yellow";
  jak: string = "center";
  aktywna: boolean = true;
  pokazZdjecie: boolean = false;

  zmienKolor() {
    this.kolor = this.kolor === "yellow" ? "green" : "yellow";
  }
}
```

Listing 2.74. Wyliczenie (enum) Plec

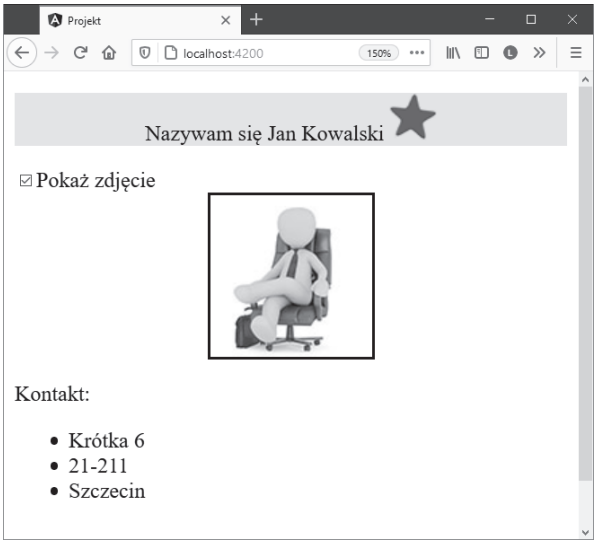
```
export interface Osoba {
    imie: string;
    nazwisko: string;
    zdjecie: string;
    bezZdjecia: string;
    adres: string[];
    type: Plec;
}

export enum Plec {
    kobieta = 1,
    mezczyzna,
    niezdefiniowana
}
```

Rysunek 2.55 pokazuje, jak zmienia się okno przeglądarki w zależności od definicji płci.



Rysunek 2.55. Działanie dyrektywy ngSwitch — dopasowanie. Po lewej — mężczyzna, po prawej — kobieta

W Angularze dozwolone jest posługiwanie się instrukcją `ngSwitchDefault`. Taki domyślny element zostanie wyświetlony, jeśli **nie zostanie znalezione żadne dopasowanie — wartość domyślna**. W naszym przykładzie, gdy właściwość `type` przyjmie wartość `Plec.niezdefiniowana`, zostanie wyświetlona gwiazdka (w wyliczeniu pole `niezdefiniowana` znajduje się na trzeciej pozycji, dlatego jego wartość to `3`, co oznacza brak dopasowania z instrukcją `ngSwitchCase` (rysunek 2.56).



Rysunek 2.56. Działanie dyrektywy ngSwitch — brak dopasowania (wartość domyślna)

WSKAZÓWKA

Aby wraz z instrukcją `ngSwitchCase` zamiast wartości numerycznych móc zastosować bardziej czytelny zapis, należy w komponencie dodać nowe pole `Plec`, które następnie trzeba powiązać z wyliczeniem. Wpis `Plec=Plec;` pozwoli zastąpić wartości liczbowe. Wartość `1` będzie można zamienić na `Plec.kobieta`, a wartość `2` — na `Plec.mezczyzna`.

2.4.2. Dyrektywy atrybutowe

Dyrektywy tego typu są używane do **zmiany wyglądu i zachowania elementów drzewa dokumentów**.

Dyrektywa ngStyle

Dyrektywa `ngStyle` służy do ustawienia stylu formatowanego elementu.

Listing 2.75 przedstawia kod, który zmienia rozmiar tekstu nagłówka `h1` oraz kolor tła po każdym wciśnięciu przycisku `Zmień styl` (wywołanie zdarzenia `click()`).

Listing 2.75. Dyrektywa ngStyle — zawartość pliku app.component.html

```
<button type="button" (click)="zmiana()">Zmień styl</button>
<h1 [ngStyle]="{'font-size.%': wybor*20+100,
'color': 'white',
'background-color': kolory[wybor] }"> Treść nagłówka</h1>
```

Oczywiście aby zdarzenie mogło zaistnieć, jego definicja musi zostać zapisana w pliku *app.component.ts*.

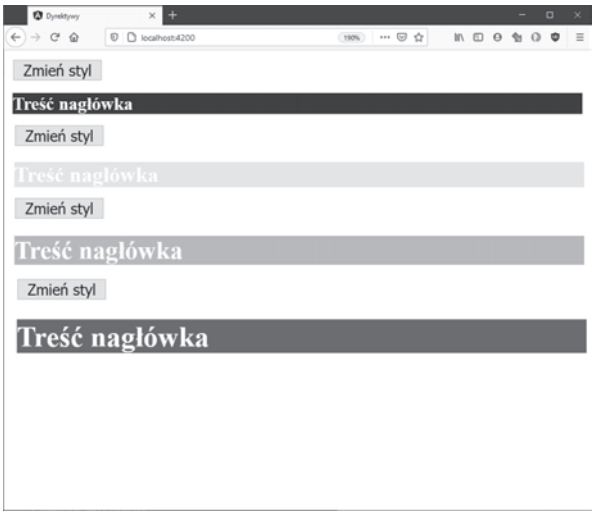
Kolor tła jest pobierany z tablicy `kolory`, a za jego wybór odpowiada zmienna `wybor`. Zmienna ta wpływa również na rozmiar wyświetlanej czcionki (listing 2.76).

Listing 2.76. Dyrektywa ngStyle — zawartość pliku app.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'dyrektywy';

  kolory: string[] = ["blue", "yellow", "orange", "red"];
  wybor: number = 0;
```

```
  zmiana(): void {
    this.wybor++;
    if (this.wybor >= 4) this.wybor = 0;
  }
}
```

Wynik działania kodu jest pokazany na rysunku 2.57 — każdorazowe wciśnięcie przycisku powoduje zmianę selektorów `font-size` i `background-color` klasy.



Rysunek 2.57. Działanie dyrektywy ngStyle

Dyrektywa ngClass

Dyrektywa `ngClass` odpowiada za ustawienie klasy elementu.

Aby poznać sposób jej działania, prześledźmy poniższy przykład. Rozpoczynamy od zdefiniowania pliku stylu.

Plik zawiera definicję trzech stylów: `a`, `b` oraz `c` (listing 2.77).

Listing 2.77. Definicja pliku stylów CSS

```
.a {
    color: blue;
}
.b {
    font-size: 20px;
    text-decoration: underline;
}
.c {
    background-color: black;
}
```

W pliku *app.component.ts* zdefiniowano tablicę `osoby`.

```
osoby = ['Jan Kowalski', 'Tadeusz Nowak', 'Beata Tryla']
```

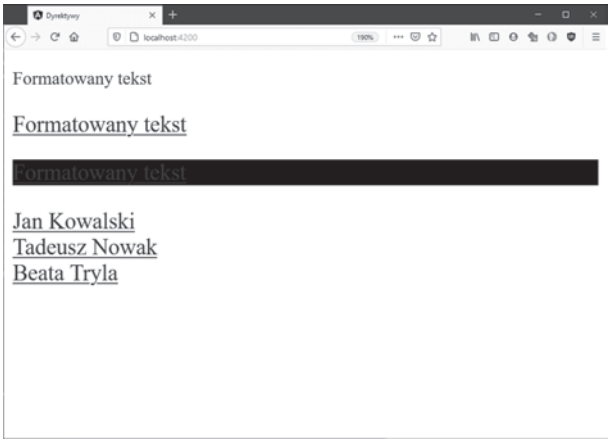
Za wyświetlenie sformatowanego tekstu odpowiada kod znajdujący się w pliku HTML (listing 2.78). Poszczególnym akapitom zostają przypisane klasy.

Przedstawiona w poprzednim rozdziale dyrektywa `ngFor` dodatkowo wyświetla i formatuje wszystkie elementy tablicy `osoby`.

Listing 2.78. Działanie dyrektywy ngClass

```
<p [ngClass]="'a'">Formatowany tekst</p>
<p [ngClass]="'a b'">Formatowany tekst</p>
<p [ngClass]="'a b c'">Formatowany tekst</p>

<div *ngFor="let osoba of osoby" [ngClass]="'a b'">{{osoba}}</div>
```

Wynik działania kodu jest pokazany na rysunku 2.58.



Rysunek 2.58. Działanie dyrektywy ngClass

Pytania kontrolne

1. Jaką rolę odgrywa dyrektywa?
2. Wskaż różnice i podobieństwa między omówionymi dyrektywami strukturalnymi a ich pierwowzorami: funkcją warunkową, pętlą oraz przełącznikiem. Czy różnią się w sposobie działania?
3. Podaj sposoby wykorzystania dyrektyw atrybutowych.

Kwalifikacja INF.04

Projektowanie, programowanie i testowanie aplikacji

Podręcznik do nauki zawodu
technik programista

Informatycy i programiści należą obecnie do najbardziej poszukiwanych specjalistów. Stąd tytuł, który uzyskuje się po szkole średniej, nie stanowi jedynie świadectwa ukończenia pewnego etapu edukacji. Technik programista to zawód o wymiernej wartości rynkowej. Absolwenci tego kierunku kształcenia nie mają większych problemów ze znalezieniem intratnego i rozwojowego zajęcia, a pracodawcy chętnie inwestują w ich szkolenia, by mogli zdobywać coraz wyższe kwalifikacje. Wśród umiejętności, które powinien posiadać specjalista w tej dziedzinie, dziś kluczowe znaczenie mają te związane z aplikacjami webowymi — błyskawiczny rozwój technologii internetowych wręcz wymusza na programistach stałą pracę nad warsztatem.

Podręcznik, oparty na podstawie programowej zawartej w kwalifikacji INF.04, w dziale *INF.04.7. Programowanie aplikacji zaawansowanych webowych*, składa się z czterech rozdziałów. W trzech zostały szczegółowo omówione technologie webowe:

- biblioteka jQuery (wraz z frameworkiem Bootstrap), umożliwiająca tworzenie animacji, a także usprawniająca manipulowanie elementami dokumentu HTML i tworzenie zdarzeń
- framework Angular, napisany w języku TypeScript i służący do budowania aplikacji webowych
- platforma Node.js (wraz z frameworkiem Express), pozwalająca uruchomić kod JavaScript poza przeglądarką

Czwarty rozdział podręcznika natomiast ma na celu zapoznać uczniów z opisanymi technologiami w praktyce, jest bowiem poświęcony tworzeniu za ich pomocą projektu aplikacji internetowej.

Podręcznik do nauki zawodu technik programista to charakteryzujący się wysoką jakością kompletny zestaw edukacyjny przygotowany przez dysponującego ogromnym doświadczeniem lidera na rynku książek informatycznych — wydawnictwo Helion.

W skład zestawu podręczników do kwalifikacji INF.04 wchodzą także:

- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 1. Inżynieria programowania — projektowanie oprogramowania, testowanie i dokumentowanie aplikacji. Podręcznik do nauki zawodu technik programista*
- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 2. Programowanie obiektowe. Podręcznik do nauki zawodu technik programista*
- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 4. Aplikacje mobilne. Podręcznik do nauki zawodu technik programista*
- *Kwalifikacja INF.04. Projektowanie, programowanie i testowanie aplikacji. Część 5. Aplikacje desktopowe. Podręcznik do nauki zawodu technik programista*

Podręczniki należące do tej serii przygotowano z myślą o wykształceniu kompetentnych techników, którzy bez trudu poradzą sobie z wyzwaniami, jakie stawia przed nimi współczesna informatyka. Wiedza zawarta w serii pomoże zdać egzamin zawodowy i uzyskać umiejętności praktyczne, przydatne w przyszłej pracy.

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

ISBN 978-83-283-8102-5
9 788328 381025