Лабораторне завдання №2. Робота з компонентами. Взаємодія між компонентами. Прив'язка до подій дочірнього компоненту. Життєвий цикл компоненту.

Тема: Навчитися працювати з компонентами в Angular.

Завдання: Створити п'ять Angular-додатків під назвою:

I) Components1 (вправи 1-6). Виконати відповідні вправи;

II) Components2 (вправи 7-8). Виконати відповідні вправи;

III) Components3 (вправа 9). Виконати відповідні вправи;

IV) Components4 (вправа 10). Виконати відповідні вправи;

V) Components5 (вправа 11). Виконати відповідні вправи;

VI) Зробити звіт по роботі.

VII) Angular-додатки Components1 та Components5 розгорнути на платформі Firebase у проектах з ім'ям «ПрізвищеГрупаLaba2-1» та «ПрізвищеГрупаLaba2-5», наприклад «KovalenkoIP01Laba2-1» та «KovalenkoIP01Laba2-5»

I) Створіть Angular-додаток Components1.

Вправа 1: Створення додатку з назвою «Components1»

Створіть папку Components1. В ній створіть наступні файли:

1) Файл package.json

```
{
  "name": "Components1",
  "version": "1.0.0",
  "description": "Angular 16 Project",
  "author": "Juliy Polupan",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build"
  },
  "dependencies": {
    "@angular/common": "~16.0.0",
    "@angular/compiler": "~16.0.0",
    "@angular/core": "~16.0.0",
    "@angular/forms": "~16.0.0",
    "@angular/platform-browser": "~16.0.0",
    "@angular/platform-browser-dynamic": "~16.0.0",
    "@angular/router": "~16.0.0",
    "rxjs": "7.8.0",
    "zone.js": "~0.13.0"
```

```
  },
  "devDependencies": {
    "@angular-devkit/build-angular": "~16.0.0",
    "@angular/cli": "~16.0.0",
    "@angular/compiler-cli": "~16.0.0",
    "typescript": "~5.0.4"
  }
}
```

3) Файл tsconfig.gson

```
{
    "compileOnSave": false,
    "compilerOptions": {
      "baseUrl": "./",
      "sourceMap": true,
      "declaration": false,
      "downlevelIteration": true,
      "experimentalDecorators": true,
      "module": "esnext",
      "moduleResolution": "node",
      "target": "es2022",
      "typeRoots": [
        "node_modules/@types"
      ],
      "lib": [
        "es2022",
        "dom"
      ]
    },
      "files": [
          "src/main.ts",
          "src/polyfills.ts"
      ],
      "include": [
          "src/**/*.d.ts"
      ]
  }
```

3) Файл angular.json

```
{
    "version": 1,
    "projects": {
      "helloapp": {
        "projectType": "application",
        "root": "",
```

```
      "sourceRoot": "src",
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/helloapp",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.json",
            "aot": true
          }
        },
        "serve": {
          "builder": "@angular-devkit/build-angular:dev-server",
          "options": {
            "browserTarget": "helloapp:build"
          }
        }
      }
    }},
  "defaultProject": "helloapp"
}
```

4) Файл polyfills.ts

```
import 'zone.js/dist/zone';
```

5) Файл main.ts

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';
const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

6) Файл index.html

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Work with Components</title>
</head>
<body>
   <my-app>Завантаження...</my-app>
</body>
</html>
```

7) Файл app.module.ts

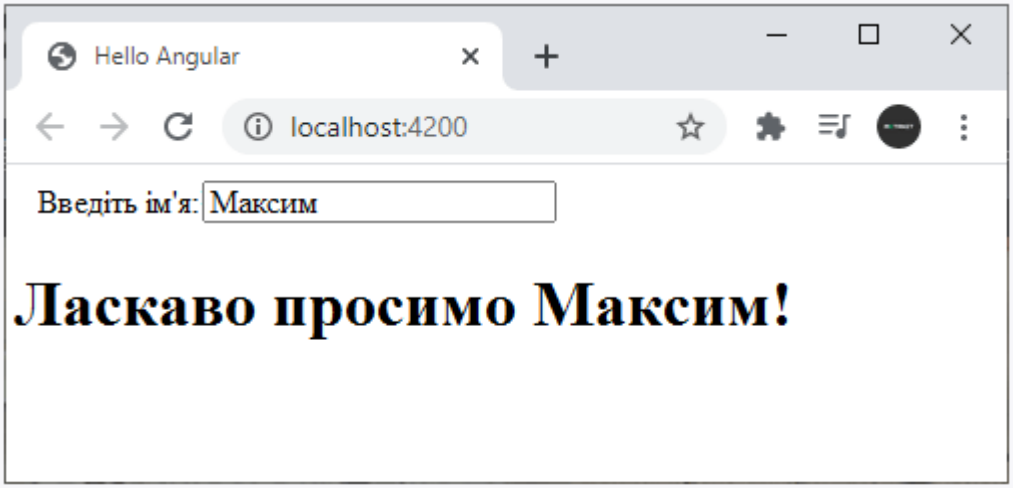
```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { AppComponent }   from './app.component';
@NgModule({
   imports:      [ BrowserModule, FormsModule ],
   declarations: [ AppComponent ],
   bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

8) Файл app.component.ts

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

@Component({
   selector: 'my-app',
   template: `<label>Введіть ім'я:</label>
              <input [(ngModel)]="name" placeholder="name">
              <h1>Ласкаво просимо {{name}}!</h1>`
})
export class AppComponent {
   name= '';
}
```

9) Командою команду npm install встановіть усі необхідні модулі.

10) У вікні «СЦЕНАРИИ NPM» запустіть команду start, або із папки проекту «Components1» запустіть команду ng serve. Результат буде наступний:



Компоненти

Одним із ключових елементів програми Angular є компоненти. Компонент керує відображенням представлення на екрані.

Так, в додатку «Components1» визначимо такий компонент:

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

@Component({
   selector: 'my-app',
   template: `<label>Введіть ім’я:</label>
            <input [(ngModel)]="name" placeholder="name">
            <h1>Ласкаво просимо {{name}}!</h1>`
})
export class AppComponent {
   name: "";
}
```

Сам клас компоненту тут відносно невеликий:

```
export class AppComponent {
   name: "";
}
```

Щоб клас міг бути використаний в інших модулях, він визначається з ключовим словом export. У самому ж класі визначено лише одну змінну, яка як значення зберігає певний рядок.

Для створення компонента необхідно імпортувати функцію декоратора @Component із бібліотеки @angular/core. Декоратор @Component дає змогу ідентифікувати клас як компонент.

Якби ми не застосували декоратор @Component до класу AppComponent, то клас AppComponent компонентом не вважався б.

Декоратор @Component як параметр приймає об'єкт із конфігурацією, яка вказує фреймворку, як працювати з компонентом та його представленням.

За допомогою властивості template ми можемо задавати шаблон представлення. Шаблон представляє шматок розмітки HTML із вкрапленнями коду Angular. Фактично шаблон це і є представлення, яке побачить користувач під час роботи з програмою.

Кожен компонент повинен мати один шаблон. Однак необов'язково визначати шаблон безпосередньо за допомогою властивості template. Можна винести шаблон у зовнішній файл із розміткою html, а для його підключення використовувати властивість templateUrl.

Шаблон може бути однорядковим або багаторядковим. Якщо шаблон багаторядковий, то він заключається в косі лапки (`), які варто відрізняти від стандартних простих лапок (').

Також у прикладі вище встановлюється властивість selector, що визначає селектор CSS. В елемент з цим селектором Angular додаватиме представлення компонента. Наприклад, у прикладі вище, селектор має значення my-app. Відповідно, якщо html-сторінка містить елемент <my-app></my-app>, наприклад:

```
<!DOCTYPE html>
<html>
<head>
   <meta charset="utf-8" />
   <title>Work with Components</title>
</head>
<body>
   <my-app>Завантаження...</my-app>
</body>
</html>
```

то саме цей елемент буде використовуватися для рендерингу представлення компонента.

В додатку Components1 виконайте вправи 2-6.

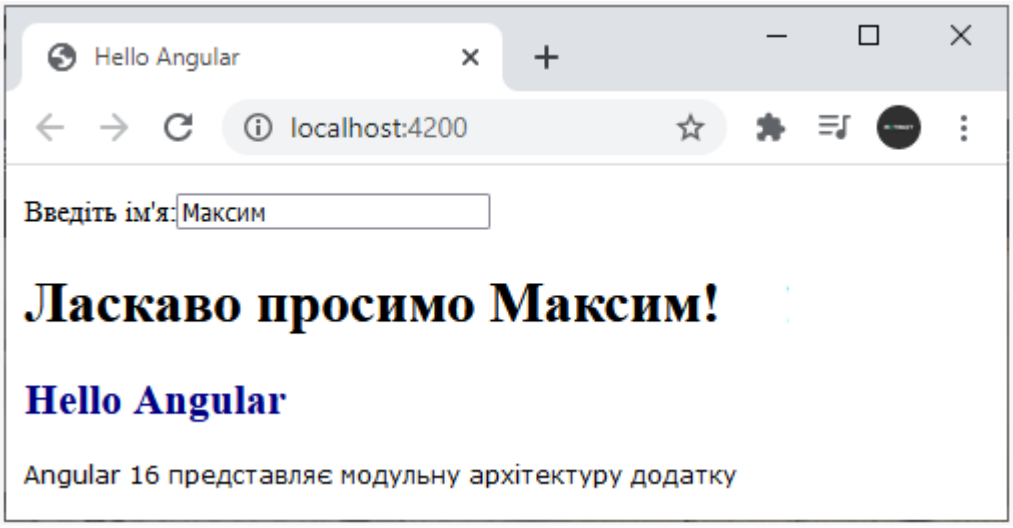
Вправа 2: Стилі та шаблони компонента

Стилізація компонента може виконуватися як за допомогою установки стилів у самому компоненті, так і за допомогою підключення зовнішніх CSS-файлів.

Для встановлення стилів у директиві @Component визначено властивість styles:

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

@Component({
   selector: 'my-app',
   template: `<h2>Hello Angular</h2>
         <p>Angular 16 представляє модульну архітектуру додатку</p>`,
   styles: [`
         h2,h3{color:navy;}
         p{font-size:13px; font-family:Verdana;}
   `]
})
export class AppComponent { }
```

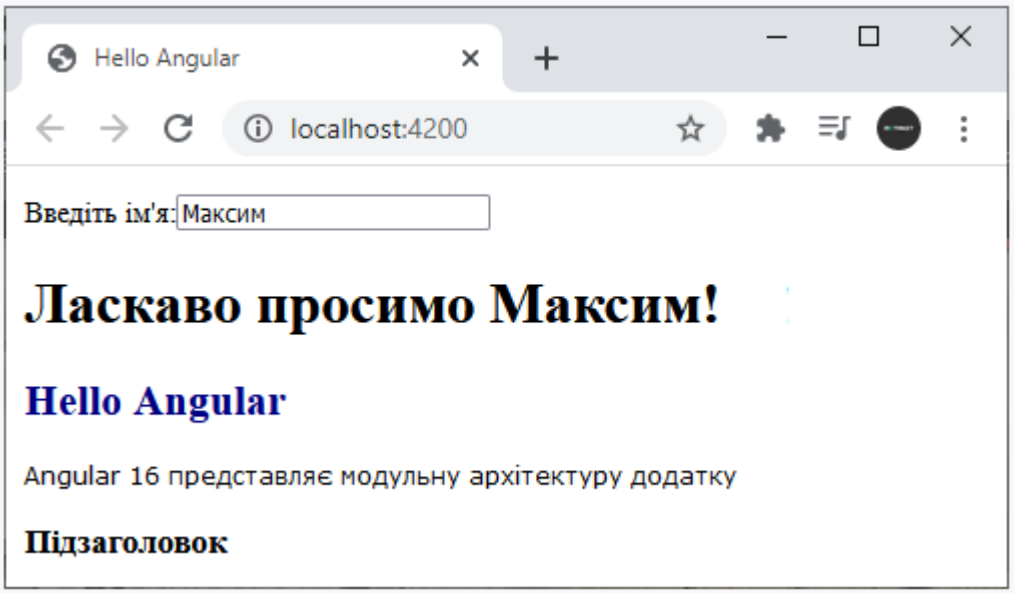
Параметр styles містить набір стилів, які використовуватимуться компонентом. Додайте в додаток «Components1» нові стилі, щоб результат був наступний:



При використанні стилів слід враховувати, що вони застосовуються локально лише до розмітки, керованої компонентом. Наприклад, якщо на сторінці будуть елементи поза областю керування компонентом, то до них вже не застосовуватимуться стилі. Наприклад:

```
<body>
   <my-app>Loading...</my-app>
   <h3>Підзаголовок</h3>
</body>
```

Якби заголовок h3 тут розташовувався в шаблоні компонента, то до нього застосовувався б стиль. А так він не буде стилізований:



Селектор: host

Селектор :host посилається на елемент, у якому хоститься компонент. Тобто в цьому випадку це елемент <my-app></my-app>. І селектор :host дає змогу застосувати стилі до цього елементу:

```
styles: [`
   h1, h2{color:navy;}
   p{font-size:13px;}
   :host {
      font-family: Verdana;
      color: #555;
   }
`]
```

Вправа 3: Підключення зовнішніх файлів

Якщо стилів багато, код компонента може бути занадто роздутий, і в цьому випадку їх виносять в окремий файл css. Так, створимо в одній папці з класом компонента (за замочуванням розташовується в папці app) новий файл app.component.css з таким вмістом:

```
h1, h2{color:navy;}
p{font-size:13px;}
:host {
   font-family: Verdana;
   color: #555;
}
```

Потім змінимо код компонента:

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

@Component({
   selector: 'my-app',
   template: `<label>Введіть ім’я:</label>
            <input [(ngModel)]="name" placeholder="name">
            <h1>Ласкаво просимо {{name}}!</h1>
            <h2>Hello Angular</h2>
         <p>Angular представляє модульну архітектуру додатку</p>`,
   styleUrls: ['./app.component.css']
})
export class AppComponent { }
```

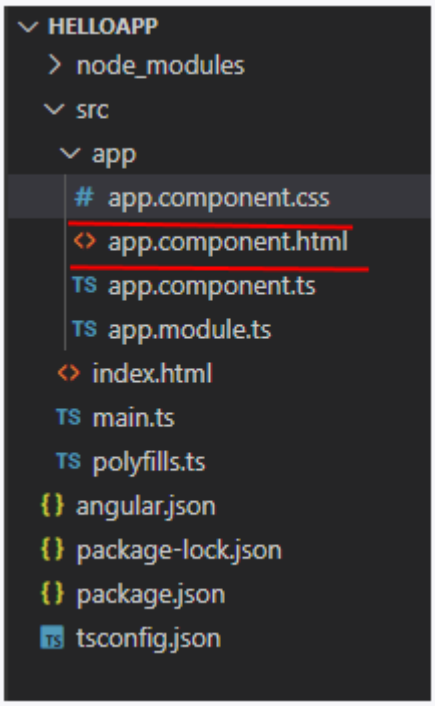
Параметр styleUrls дозволяє вказати набір файлів CSS, які застосовуються для стилізації. В даному випадку передбачається, що файл CSS розташовується в проекті в папці app.

Подібним чином ми можемо винести шаблон в окремий файл html. Також у папці app створимо новий файл app.component.html з наступним кодом:

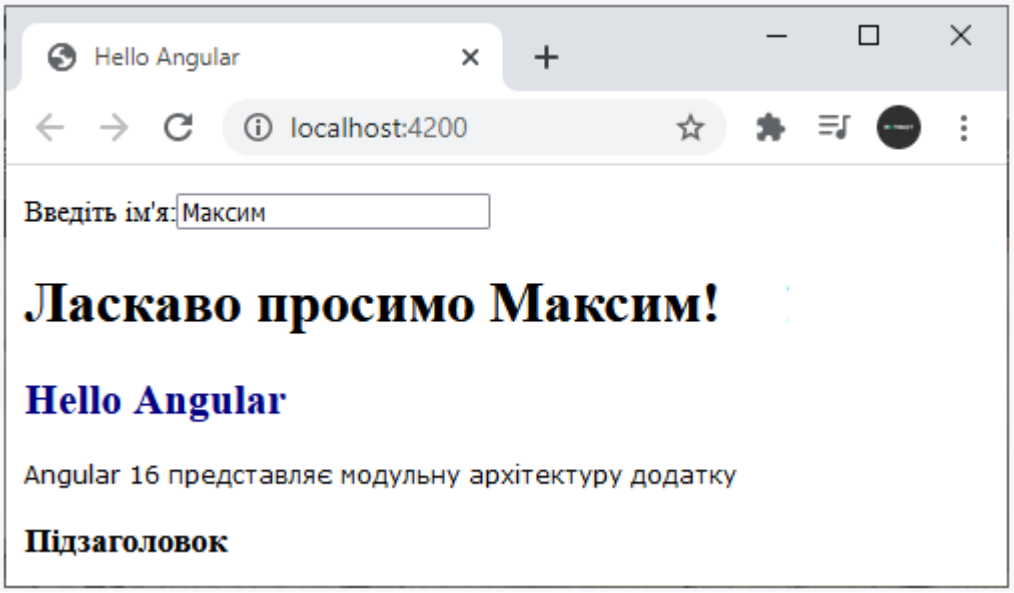
```
<label>Введіть ім’я:</label>
<input [(ngModel)]="name" placeholder="name">
<h1>Ласкаво просимо {{name}}!</h1>
<h2>Hello Angular</h2>
<p>Angular представляє модульну архітектуру додатку</p>
```

Тобто тут визначено той самий код, що раніше був у шаблоні компонента. І тепер змінимо сам компонент:

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

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



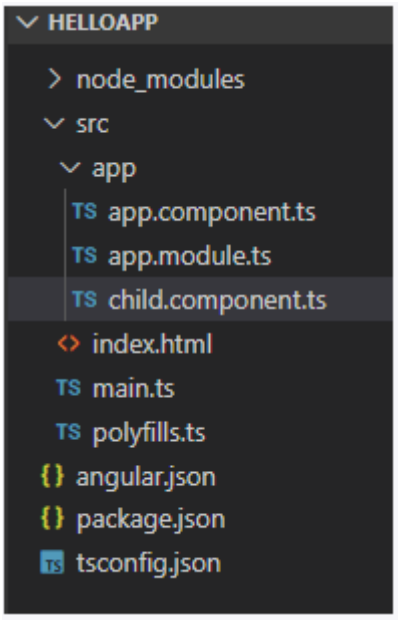
Таким чином, за рахунок винесення коду css і html код самого компонента став простіше і чистіше. Результат роботи додатку буде той самий:



Вправа 4: Робота з компонентами

Крім основних компонентів у додатку, ми також можемо визначати якісь допоміжні компоненти, які керують якоюсь ділянкою розмітки html. Більше того, у додатку на сторінці може бути ряд різних блоків з певним завданням. І для кожного такого блоку можна створити окремий компонент, щоб спростити керування блоками на сторінці.

Додамо до проекту другий компонент. Для цього додамо до папки src/app новий файл child.component.ts. У результаті весь проект виглядатиме так:



Визначимо у файлі child.component.ts наступний код:

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

```
@Component({
   selector: 'child-comp',
   template: `<h2>Ласкаво просимо {{name}}!</h2>`,
   styles: [`h2, p {color:red;}`]
})
export class ChildComponent {
   name= "Тарас";
}
```

Тут визначено клас ChildComponent. Знову ж таки, щоб зробити цей клас компонентом, необхідно застосувати декоратор @Component.

Компонент керуватиме розміткою html, яка вставлятиметься в елемент child-comp.

Шаблон представлення буде просто виводити заголовок. У заголовку виводиться просто ім'я, задане через змінну name.

Крім того, тут визначені стилі для елементів h2 і p.

Тепер змінимо код компонента AppComponent у файлі app.component.ts:

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

@Component({
   selector: 'my-app',
   template: `
<label>Введіть ім'я:</label>
              <input [(ngModel)]="name" placeholder="name">
              <h1>Ласкаво просимо {{name}}!</h1>
              <h2>Hello Angular</h2>
          <p>Angular 16 представляє модульну архітектуру додатку</p>
            <child-comp></child-comp>
             <p>Hello {{name}}</p>`,
   styles: [`h2, p {color:#333;}`]
})
export class AppComponent {
   name = 'Петро';
}
```

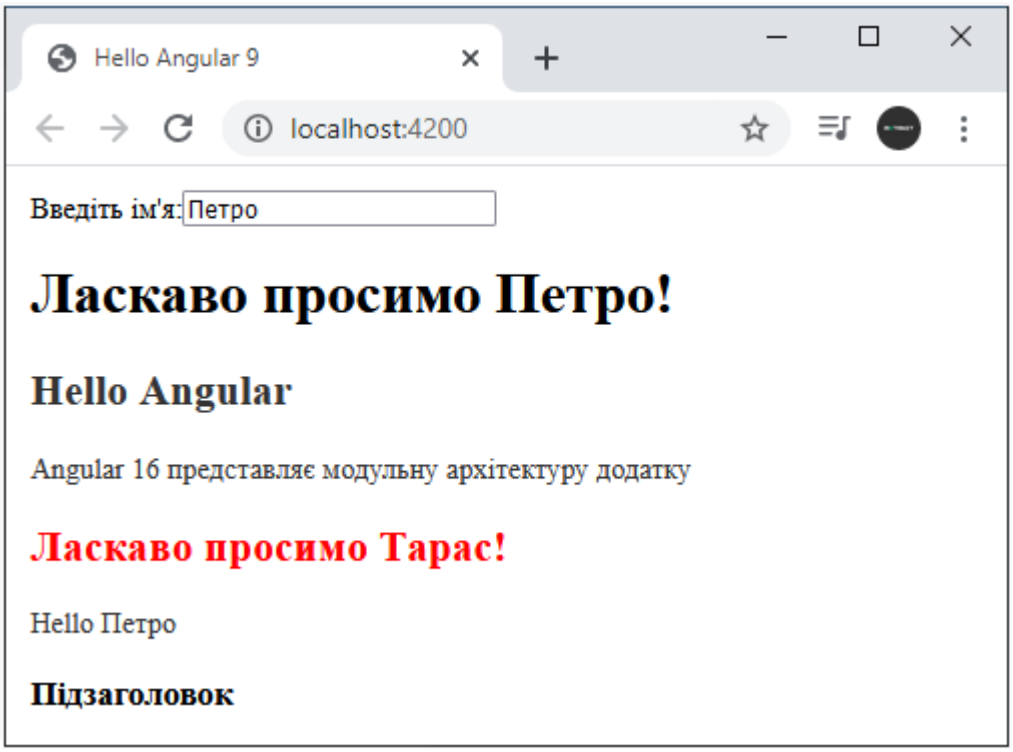
Це основний компонент, який запускатиметься при запуску програми, і через нього ми будемо використовувати інші компоненти. Так, компонент ChildComponent завантажуватиметься в елемент child-comp. І в шаблоні компонента AppComponent якраз визначено такий елемент.

Крім того, компонент визначає стилі для тих же елементів на сторінці, а також, як і ChildComponent, визначає властивість name, тільки з іншим значенням.

Щоб використовувати всі компоненти, визначені в проекті, вони повинні бути вказані в головному модулі програми. Визначимо у файлі app.module.ts наступний модуль:

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule }   from '@angular/forms';
import { AppComponent }   from './app.component';
import { ChildComponent }   from './child.component';
@NgModule({
   imports:      [ BrowserModule, FormsModule ],
   declarations: [ AppComponent, ChildComponent ],
   bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Запустимо проект:



Результат показує, що незважаючи на те, що один компонент як би включений в інший за допомогою тега <child-comp>, проте стилі одного компонента не застосовуються до іншого. Кожен компонент окреслює свою область за допомогою шаблону, своє представлення, яким він управляє.

Також кожен компонент використовує значення властивості name. Тобто, компоненти фактично існують відносно незалежно.

Вправа 5: ng-content

Елемент ng-content дозволяє батьківським компонентам впроваджувати код HTML у дочірні компоненти. Так, змінимо компонент ChildComponent наступним чином:

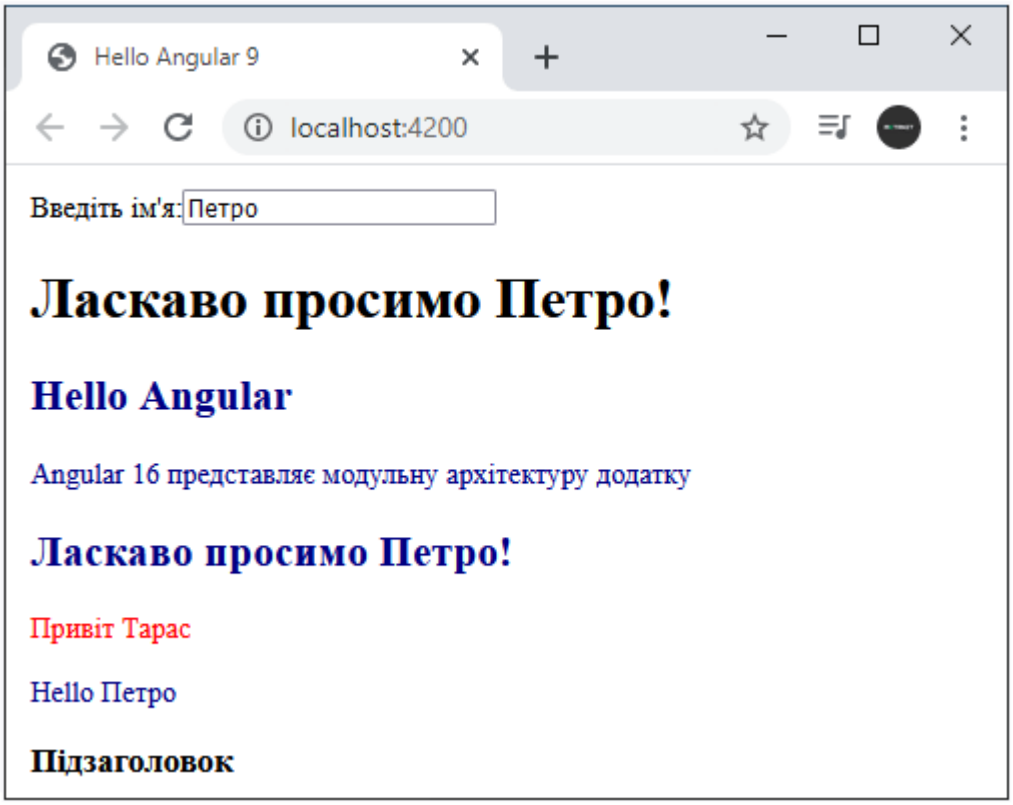
```
import { Component } from '@angular/core';

@Component({
 selector: 'child-comp',
   template: `<ng-content></ng-content>
            <p>Привіт {{name}}</p>`,
   styles: [`h2, p {color:red;}`]
})
export class ChildComponent {
   name= "Тарас";
}
```

Замість елемента <ng-content> ззовні можна буде передати будь-який вміст.

І змінимо код головного компонента AppComponent:

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

@Component({
   selector: 'my-app',
   template: <label>Введіть ім'я:</label>
              <input [(ngModel)]="name" placeholder="name">
              <h1>Ласкаво просимо {{name}}!</h1>
              <h2>Hello Angular</h2>
          <p>Angular 16 представляє модульну архітектуру додатку</p>
          <child-comp><h2>Ласкаво просимо {{name}}!</h2></child-comp>
             <p>Hello {{name}}</p>
`,
   styles: [` h3{color:navy;}
         h2, p {color:navy;}`]
})
export class AppComponent {
   name = 'Петро';
}
```

В елемент <child-comp> тут передається заголовок <h2>Ласкаво просимо {{name}}!</h2>. Потім цей заголовок буде вставлятись у дочірній компонент ChildComponent на місце <ng-content>:



Причому керувати розміткою, яка вставляється у <ng-content>, буде AppComponent. Тому саме цей компонент задає стилі і вирази прив'язки для шматка html, що вставляється.

Вправа 6: Взаємодія між компонентами. Передача даних у дочірній компонент

У минулій темі було розглянуто, як викликати компонент із головного компонента. Однак за замовчуванням ці компоненти не взаємодіють, вони незалежні. Кожен компонент визначає свої вирази прив'язки. Однак що, якщо ми хочемо прив'язати властивості дочірнього компонента до властивостей головного компонента? Для цього визначимо наступний дочірній компонент:

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

@Component({
   selector: 'child-comp',
   template: ` <ng-content></ng-content>
                <p>Привіт {{name}}</p>
           <p>Ім’я користувача: {{userName}}</p>
           <p>Вік користувача: {{userAge}}</p>`
})
export class ChildComponent{
```

```
    @Input() userName: string = "";
    @Input() userAge: number = 0;
}
```

Ключовим моментом є визначення вхідних властивостей з допомогою декоратора @Input(). І природно, щоб використовувати декоратор, його треба імпортувати:

```
import { Input } від '@angular/core';
```

Ключовою особливістю таких вхідних властивостей є те, що вони можуть встановлюватися ззовні, тобто ззовні отримувати значення, наприклад, з головного компонента.

Тепер змінимо код головного компонента:

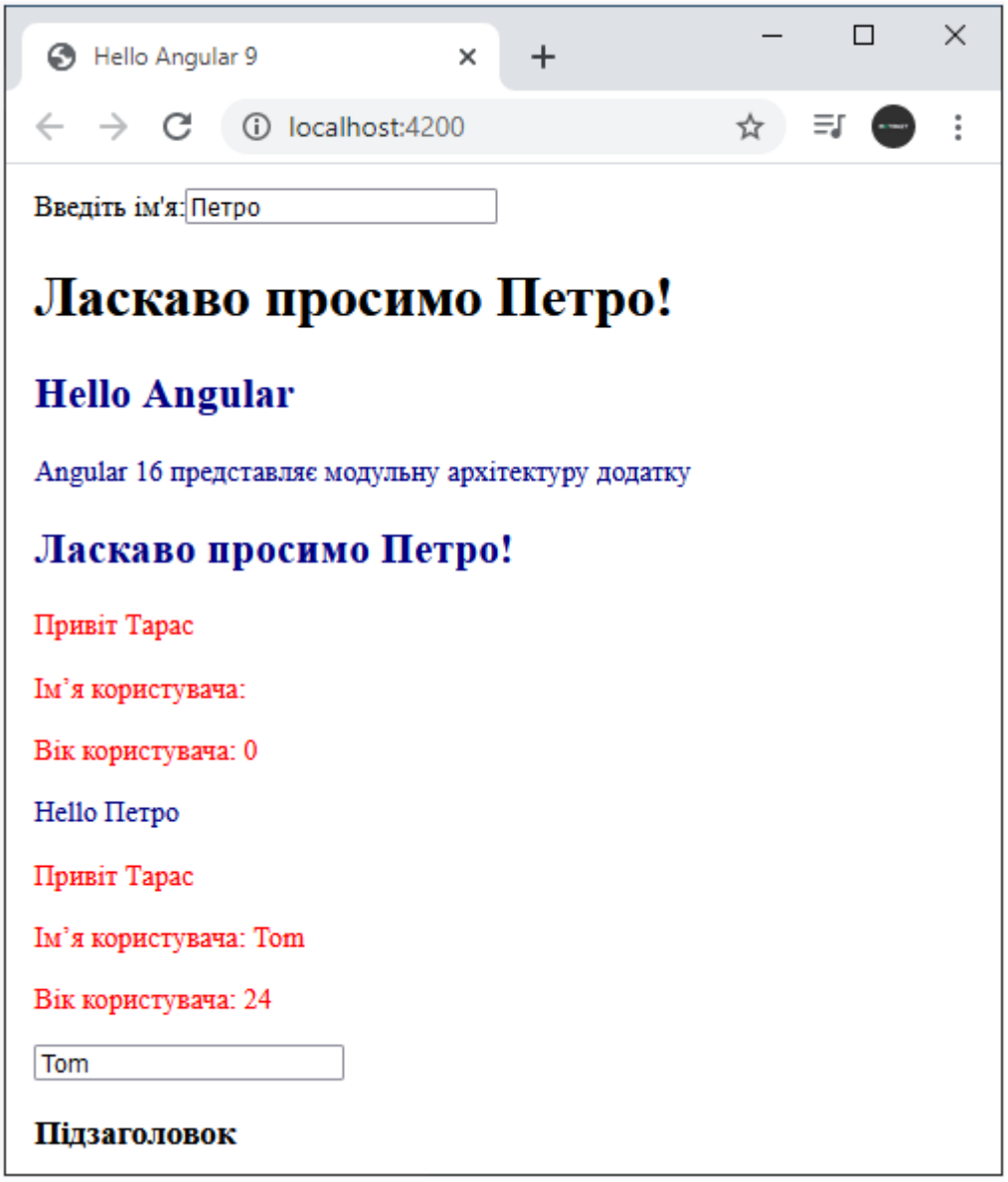
```
import { Component } from '@angular/core';

@Component({
    selector: 'my-app',
    template: ` <label>Введіть ім'я:</label>
                <input [(ngModel)]="name" placeholder="name">
                <h1>Ласкаво просимо {{name}}!</h1>
                <h2>Hello Angular</h2>
            <p>Angular 16 представляє модульну архітектуру додатку</p>
            <child-comp><h2>Ласкаво просимо {{name}}!</h2></child-comp>
               <p>Hello {{name}}</p>
                <child-comp [userName]="name2" [userAge]="age"></child-comp>
                <input type="text" [(ngModel)]="name2" />
`
})
export class AppComponent {
    name = 'Петро';
    name2:string="Tom";
    age:number = 24;
}
```

Оскільки властивість userName у дочірньому компоненті визначена як вхідна з декоратором Input, то в головному компоненті ми можемо її використовувати як атрибут і фактично застосувати прив'язку властивостей:

```
<child-comp [userName]="name" [userAge]="age"></child-comp>
```

Це ж стосується і властивості userAge.

В результаті характеристики userAge і userName будуть прив'язані до значень з основного компонента:



II) Створіть новий додаток з назвою «Components2». Виконайте в цьому додатку вправи 7 – 8.

Вправа 7: Прив'язка до сетера

Крім прив'язки до властивості, ми можемо встановити прив'язку до сеттера дочірнього компонента. Це може бути необхідно, коли у дочірньому компоненті треба здійснювати перевірку або навіть модифікацію значення, що отримується від головного компонента.

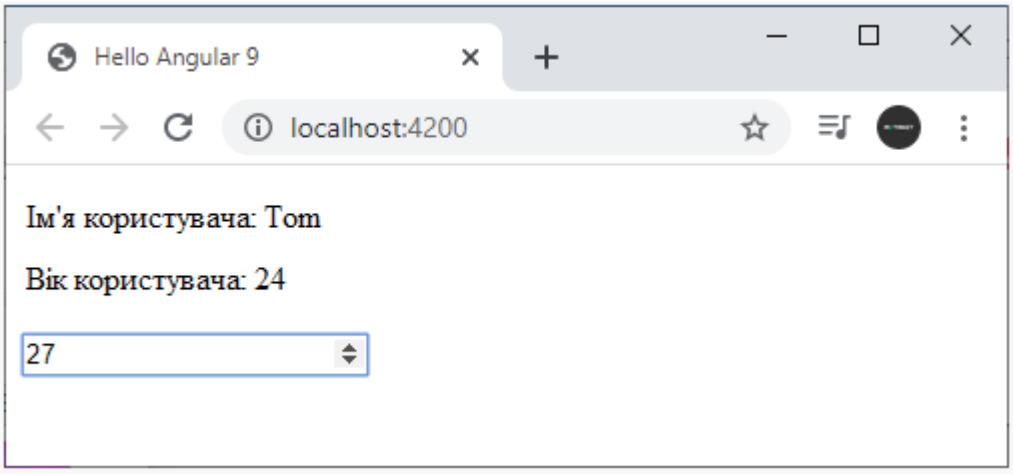
Наприклад, нехай у головному компоненті встановлюється вік користувача:

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

@Component({
    selector: 'my-app',
    template: `<child-comp [userName]="name" [userAge]="age"></child-comp>
                <input type="number" [(ngModel)]="age" />`
})
export class AppComponent {
    name:string="Tom";
    age:number = 24;
}
```

А в дочірньому компоненті отримуватимемо переданий вік через сеттер:

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

@Component({
    selector: 'child-comp',
    template: `<p>Ім’я користувача: {{userName}}</p>
                    <p>Вік користувача: {{userAge}}</p>`
})
export class ChildComponent{
    @Input() userName: string = "";
     _userAge: number = 0;

    @Input()
    set userAge(age:number) {
        if(age<0)
            this._userAge=0;
        else if(age>100)
            this._userAge=100;
        else
            this._userAge = age;
  }
  get userAge() { return this._userAge; }
}
```

У головному компоненті ми можемо запровадити будь-яке значення у полі введення, зокрема і негативні числа. У дочірньому компоненті через сеттер перевіряємо введене значення та за необхідності коригуємо його.



Вправа 8: Прив'язка до подій дочірнього компонента

Ще однією формою взаємодії є прив'язка до подій дочірнього компонента. Так, визначимо наступний дочірній компонент:

```
import { Component, EventEmitter, Input, Output} from '@angular/core';

@Component({
   selector: 'child-comp',
   template: `
            <p>Ім’я користувача: {{userName}}</p>
            <p>Вік користувача: {{userAge}}</p>
            <button (click)="change(true)">+</button>
            <button (click)="change(false)">-</button>`
})
export class ChildComponent{
        @Input() userName: string = "";
    _userAge: number = 0;

   @Input()
   set userAge(age:number) {
      if(age<0)
         this._userAge=0;
      else if(age>100)
         this._userAge=100;
      else
         this._userAge = age;
  }
  get userAge() { return this._userAge; }
   @Output() onChanged = new EventEmitter<boolean>();
   change(increased:any) {
      this.onChanged.emit(increased);
   }
}
```

У цьому компоненті у кнопки використовується подія click, яка викликає метод change, передаючи йому значення true або false. Тут же в дочірньому компоненті ми можемо обробити події. Але якщо ми повинні передавати його батьківському компоненту, то для цього нам треба використовувати властивість типу EventEmitter, яким тут є властивість onChanged. Оскільки ми передаватимемо значення типу true або false, то дана властивість типизується типом boolean. При цьому властивість onChanged повинна бути вихідною, тому вона позначається за допомогою декоратора @Output.

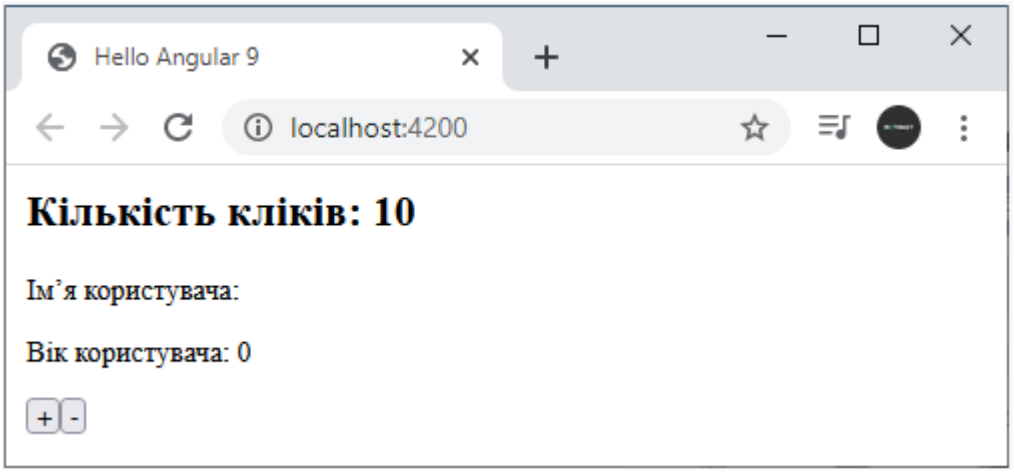
Фактично властивість onChanged буде представляти собою подію, яка викликається в методі change() при кліку на кнопку і передається головному компоненту.

Далі визначимо код головного компонента:

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

@Component({
    selector: 'my-app',
    template: `
                <h2>Кількість кліків: {{clicks}}</h2>
                <child-comp (onChanged)="onChanged($event)"></child-comp>
`
})
export class AppComponent {
    name:string="Tom";
    age:number = 24;
    clicks:number = 0;
    onChanged(increased:any){
        increased==true?this.clicks++:this.clicks--;
    }
}
```

За допомогою виразу (onChanged)="onChanged($event)" прив'язуємо метод onChanged до події onChanged(), що викликається в дочірньому компоненті. Параметр $event інкапсулює всі дані, що передаються із дочірнього компонента.

У результаті при натисканні на кнопки в дочірньому компоненті подія натискання транслюватиметься головному копоненту, який в залежності від переданого значення збільшуватиме або зменшуватиме лічильник.



III) Створіть новий додаток з назвою «Components3». Виконайте в цьому додатку вправу 9.

Вправа 9: Двостороння прив'язка

У прикладі вище ми визначали прив'язку до події дочірнього компонента: у разі виникнення події у дочірньому компоненті ми обробляли її у головному компоненті за допомогою методу. Але ми можемо використовувати двосторонню прив'язку між властивостями головного і дочірнього компонента. Наприклад, нехай дочірній компонент виглядатиме так:

```
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
   selector: 'child-comp',
   template: `
       <input [ngModel]="userName" (ngModelChange)="onNameChange($event)" />`
})
export class ChildComponent{

   @Input() userName:string = "";
   @Output() userNameChange = new EventEmitter<string>();
   onNameChange(model: string){

      this.userName = model;
      this.userNameChange.emit(model);
   }
}
```

Тут визначено вхідну властивість userName, до якого прив'язане текстове поле input. Для зв'язку використовується атрибут [ngModel], який пов'язує значення атрибута value у текстового полі з властивістю userName.

Для відстеження зміни моделі цього поля за допомогою атрибуту (ngModelChange) прив'язуємо метод, який спрацьовує при зміні значення. Тобто ngModelChange – це фактично подія зміни введеного значення, тому тут діє прив'язка до події.

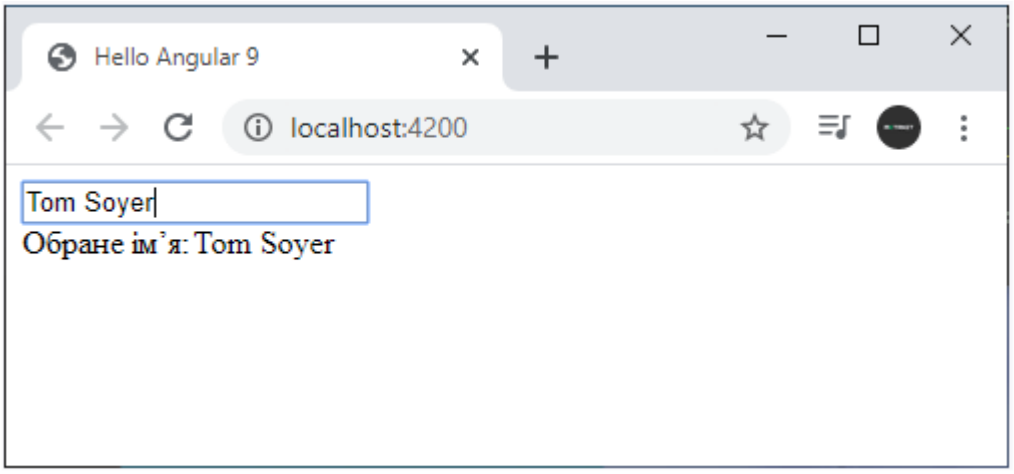
Так як у нас тут одностороння прив'язка, то в методі-обробнику отримуємо введене значення та змінюємо властивість userName і генеруємо подію userNameChange, яку визначено як вихідний параметр.

Тобто тут ззовні ми отримуємо значення властивості userName і встановлюємо його для текстового поля. При введенні користувача в це поле генеруємо зовні подію userNameChange.

Тепер визначимо код головного компонента:

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

@Component({
    selector: 'my-app',
    template: `<child-comp [(userName)]="name"></child-comp>
                <div>Обране ім’я: {{name}}</div>`
})
export class AppComponent {

    name: string = "Tom";
}
```

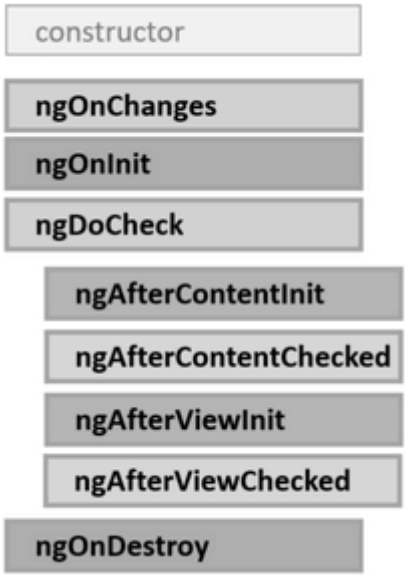
Тут встановлюється двостороння прив'язка властивостей userName дочірнього компонента та властивості name головного компонента. При цьому не потрібно вже вручну обробляти подію userNameChange, все робитиметься автоматично.



IV) Створіть новий додаток з назвою «Components4». Виконайте в цьому додатку вправу 10.

Вправа 10: Життєвий цикл компоненту

Після створення компонента фреймворк Angular викликає у цього компоненту ряд методів, які представляють різні етапи життєвого циклу:



- ngOnChanges: викликається до методу ngOnInit() при початковій установці властивостей, які пов'язані механізмом прив'язки, а також при будь-якій їхній переустановці або зміні їх значень. Даний метод як параметр приймає об'єкт класу SimpleChanges, який містить попередні та поточні значення властивості.
- ngOnInit: викликається один раз після встановлення властивостей компонента, що беруть участь у прив'язці. Виконує ініціалізацію компонента.
- ngDoCheck: викликається при кожній перевірці змін властивостей компонента відразу після методів ngOnChanges та ngOnInit.
- ngAfterContentInit: викликається один раз після методу ngDoCheck() після вставки вмісту у представлення компонента.
- ngAfterContentChecked: викликається фреймворком Angular при перевірці змін вмісту, який додається до представлення компонента. Викликається після методу ngAfterContentInit() та після кожного наступного виклику методу ngDoCheck().
- ngAfterViewInit: викликається фреймворком Angular після ініціалізації представлення компонента, а також представленнь дочірніх компонентів. Викликається лише один раз відразу після першого виклику методу ngAfterContentChecked().
- ngAfterViewChecked: викликається фреймворком Angular після перевірки на зміни у представленні компонента, а також перевірки представленнь дочірніх

компонентів. Викликається після першого виклику методу ngAfterViewInit() та після кожного наступного виклику ngAfterContentChecked().

- ngOnDestroy: викликається перед тим, як фреймворк Angular видалить компонент.

Кожен такий метод визначено в окремому інтерфейсі, який називається ім'ям методу без префікса "ng". Наприклад, метод ngOnInit визначено в інтерфейсі OnInit. Тому, якщо ми хочемо відстежувати якісь етапи життєвого циклу компонента, то клас компонента має застосовувати відповідні інтерфейси:

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

@Component({
   selector: 'my-app',
   template: `<p>Hello Angular 2</p>`
})
export class AppComponent implements OnInit, OnDestroy {
   name:string="Tom";

   constructor(){ this.log(`constructor`); }
   ngOnInit() { this.log(`onInit`); }

   ngOnDestroy() { this.log(`onDestroy`); }

   private log(msg: string) {
      console.log(msg);
   }
}
```

ngOnInit

Метод ngOnInit() застосовується для комплексної ініціалізації компонента. Тут можна виконувати завантаження даних із сервера або інших джерел даних.

ngOnInit() не аналогічний конструктору. Конструктор також може виконувати деяку ініціалізацію об'єкта, водночас щось складне у конструкторі робити не рекомендується. Конструктор має бути по можливості простим та виконувати саму базову ініціалізацію. Щось складніше, наприклад, завантаження даних із сервера, яке може зайняти тривалий час, краще робити в методі ngOnInit.

ngOnDestroy

Метод ngOnDestroy() викликається перед видаленням компонента. І в цьому методі можна звільняти ті ресурси, які не видаляються автоматично збирачем сміття. Тут також можна видаляти підписку на якісь події елементів DOM, зупиняти таймери тощо.

ngOnChanges

Метод ngOnChanges() викликається перед методом ngOnInit() і при зміні властивостей в прив'язці. За допомогою параметра SimpleChanges у методі можна отримати поточне та попереднє значення зміненої властивості. Наприклад, нехай у нас буде наступний дочірній компонент:

```
import { Component, Input, OnInit, OnChanges, SimpleChanges } from '@angular/core';

@Component({
  selector: 'child-comp',
  template: `<p>Привіт {{name}}</p>`
})
export class ChildComponent implements OnInit, OnChanges {
  @Input() name: string = "";

  constructor(){ this.log(`constructor`); }
  ngOnInit() { this.log(`onInit`); }

  ngOnChanges(changes: SimpleChanges) {
    for (let propName in changes) {
      let chng = changes[propName];
      let cur  = JSON.stringify(chng.currentValue);
      let prev = JSON.stringify(chng.previousValue);
      this.log(`${propName}: currentValue = ${cur}, previousValue = ${prev}`);
    }
  }
  private log(msg: string) {
    console.log(msg);
  }
}
```

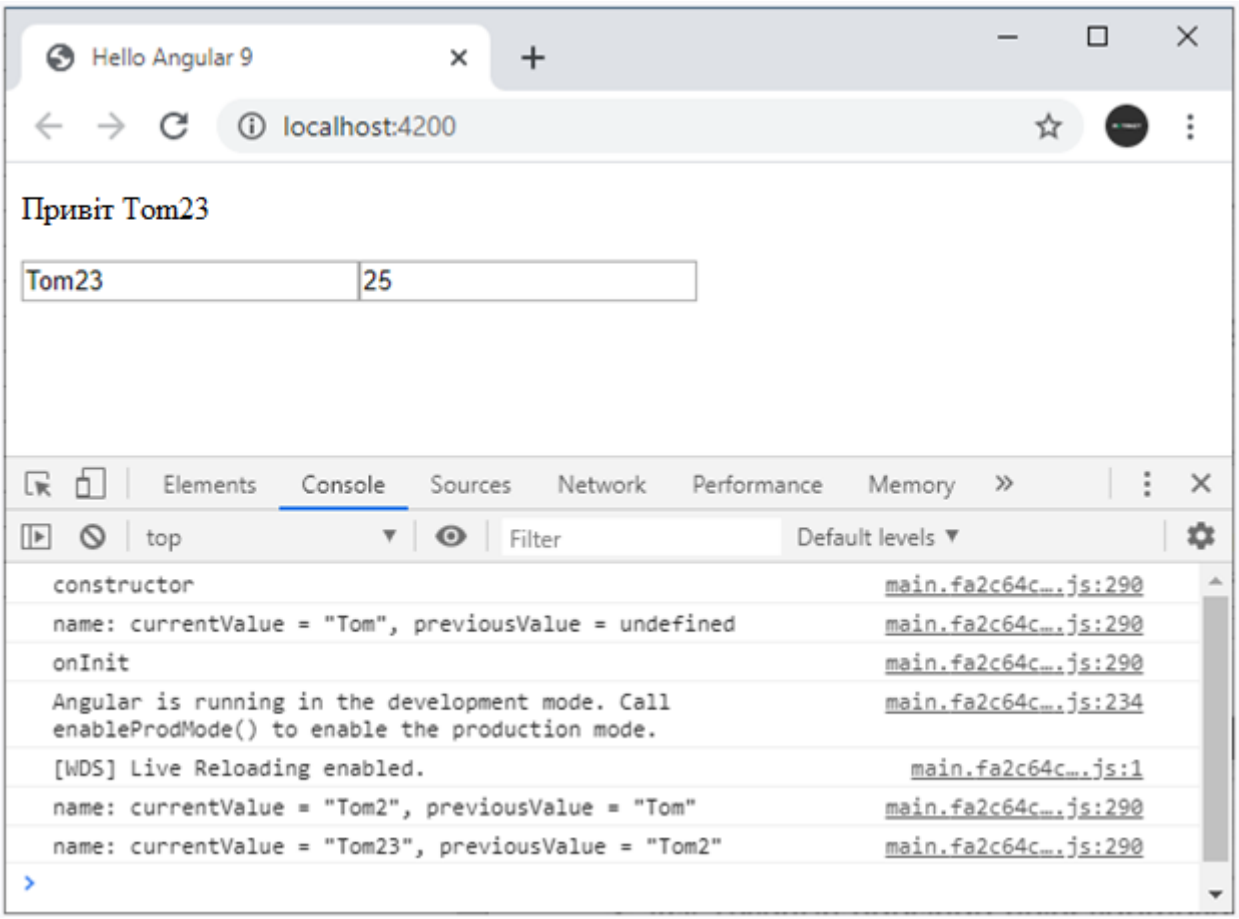
І нехай цей компонент використовується у головному компоненті:

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

@Component({
  selector: 'my-app',
  template: `<child-comp [name]="name"></child-comp>
        <input type="text" [(ngModel)]="name" />
        <input type="number" [(ngModel)]="age" />`
```

```
})
export class AppComponent implements OnChanges {
    name:string="Tom";
    age:number = 25;
    ngOnChanges(changes: SimpleChanges) {
      for (let propName in changes) {
        let chng = changes[propName];
        let cur  = JSON.stringify(chng.currentValue);
        let prev = JSON.stringify(chng.previousValue);
        this.log(`${propName}: currentValue = ${cur}, previousValue = ${prev}`);
      }
    }

    private log(msg: string) {
        console.log(msg);
    }
}
```

Тобто значення властивості name передається в дочірній компонент ChildComponent з головного - AppComponent. Причому в головному компоненті також реалізований метод ngOnChanges().

І якщо ми запустимо додаток, то зможемо помітити, що при кожній зміні властивості name у головному компоненті викликається метод ngOnChanges:



У той самий час слід зазначити, що цей метод викликається лише при зміні вхідних властивостей з декоратором @Input. Тому зміна властивості age у AppComponent тут не буде відстежуватися.

V) Створіть новий додаток з назвою «Components5». Виконайте в цьому додатку вправу 11.

Вправа 11: Реалізація всіх методів

Визначимо наступний дочірній компонент:

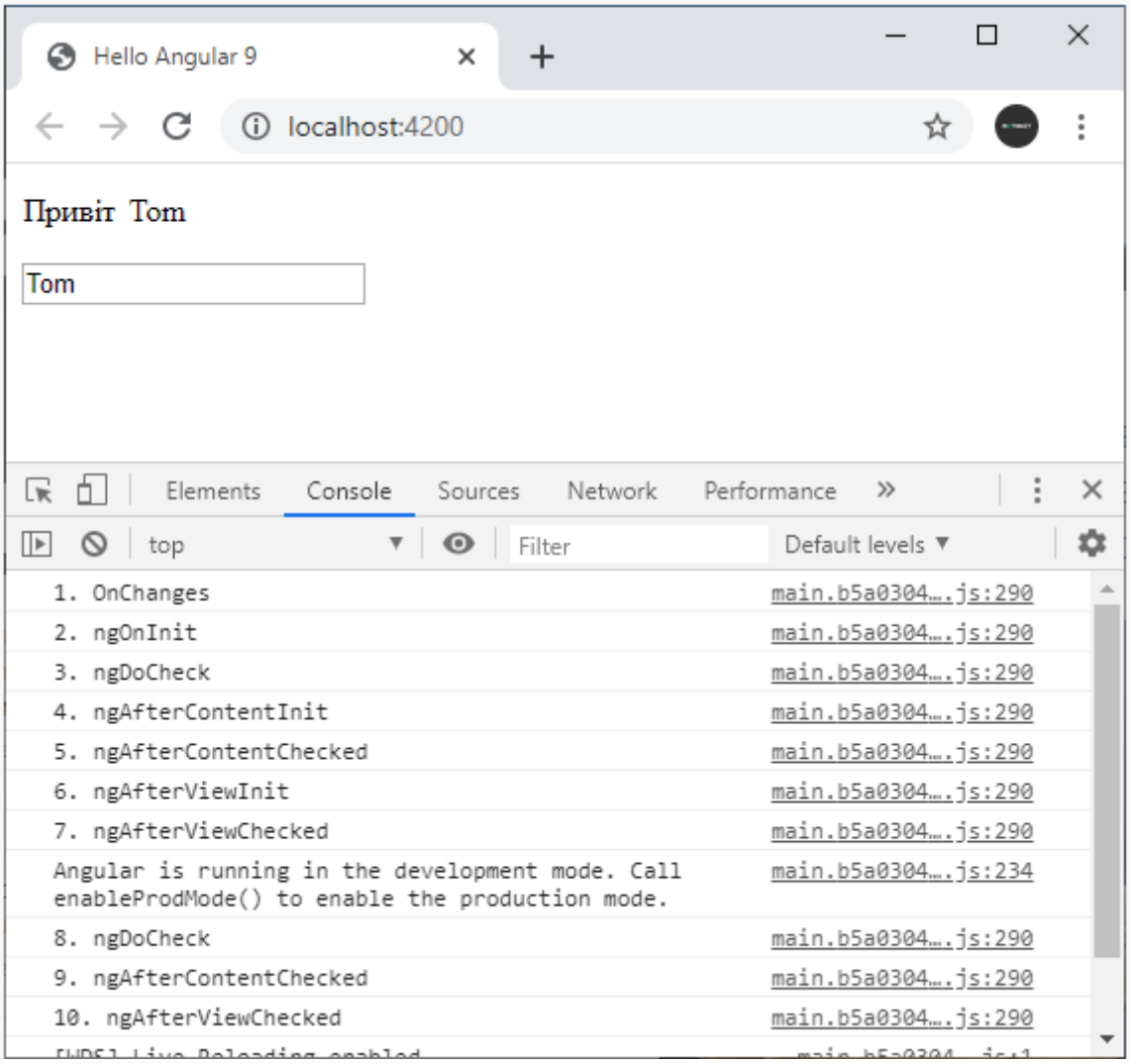
```
import { Component,
         Input,
         OnInit,
         DoCheck,
         OnChanges,
        AfterContentInit,
        AfterContentChecked,
        AfterViewChecked,
        AfterViewInit} from '@angular/core';

@Component({
   selector: 'child-comp',
   template: `<p>Привіт {{name}}</p>`
```

```
})
export class ChildComponent implements OnInit,
        DoCheck,
        OnChanges,
       AfterContentInit,
       AfterContentChecked,
       AfterViewChecked,
       AfterViewInit  {
   @Input() name: string = "";
   count:number = 1;

   ngOnInit() {

     this.log(`ngOnInit`);
   }
   ngOnChanges() {

     this.log(`OnChanges`);
   }
   ngDoCheck() {

     this.log(`ngDoCheck`);
   }
   ngAfterViewInit() {

     this.log(`ngAfterViewInit`);
   }
   ngAfterViewChecked() {

     this.log(`ngAfterViewChecked`);
   }
   ngAfterContentInit() {

     this.log(`ngAfterContentInit`);
   }
   ngAfterContentChecked() {

     this.log(`ngAfterContentChecked`);
   }

   private log(msg: string) {
      console.log(this.count + ". " + msg);
      this.count++;
   }
}
```

І використаємо цей компонент у головному компоненті:

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

@Component({
   selector: 'my-app',
   template: `<child-comp [name]="name"></child-comp>
            <input type="text" [(ngModel)]="name" />`
})
export class AppComponent{
    name:string="Tom";
}
```

І при зверненні до програми ми отримаємо наступний ланцюжок викликів:



VI) Зробити звіт по роботі. Звіт повинен бути не менше 8 сторінок без титульного аркуша (шрифт Times New Roman, 14, полуторний інтервал). Титульний аркуш приводиться у додатку. Звіт повинен містити наступні розділи:

a) Стилі та шаблони компонента.

b) Селектор: host. Призначення та використання.

c) Підключення зовнішніх файлів стилів та шаблонів.

d) ng-content: призначення та використання.

e) Взаємодія між компонентами. Передача даних у дочірній компонент.

f) Прив'язка до сетера.

g) Прив'язка до подій дочірнього компонента.

h) Двостороння прив'язка.

i) Життєвий цикл компоненту.

VII) Angular-додатки Components1 та Components5 розгорнути на платформі Firebase у проектах з ім’ям «ПрізвищеГрупаLaba2-1» та «ПрізвищеГрупаLaba2-5», наприклад «KovalenkoIP01Laba2-1» та «KovalenkoIP01Laba2-5».

ДОДАТОК

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ «КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки

Кафедра інформатики та програмної інженерії

Звіт по лабораторній роботі №____

_____

назва лабораторної роботи

з дисципліни: «Реактивне програмування»

Студент: _____

Група: _____

Дата захисту роботи: _____

Викладач: доц. Полупан Юлія Вікторівна

Захищено з оцінкою: _____

Київ, 2023