Лекція №6. Pure та Impure Pipes. AsyncPipe. Angular HttpClient. Посилальні змінні шаблону. Використання Observable для передачі значень.

Pipes бувають двох типів: pure (що не допускають змін) та impure (допускають зміни). Відмінність між цими двома типами полягає у реагуванні на зміну значень, що передаються в pipe.

За замовчуванням усі pipes є типом "pure". Такі об'єкти відстежують зміни у значеннях примітивних типів (String, Number, Boolean, Symbol). У інших об'єктах - типів Date, Array, Function, Object зміни відстежуються, коли змінюється посилання, але не значення за посиланням. Тобто, якщо в масив додали елемент, масив змінився, але посилання змінної, яка представляє даний масив, не змінилася. Тому подібну зміну pure pipes не відстежуватиме.

Impure pipes відстежують усі зміни. Можливо, постає питання, навіщо тоді потрібні pure pipes? Справа в тому, що відстеження змін позначається на продуктивності, тому pure pipes можуть показувати кращу продуктивність. До того ж не завжди необхідно відслідковувати зміни у складних об'єктах, іноді це не потрібно.

Тепер подивимося на прикладі. Створемо клас FormatPipe:

```
e:\Angular_App\Pipes_2\
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'format'
})
export class FormatPipe implements PipeTransform {
  transform(value: number, args?: any): string {

    return value.toString().replace(".", ",");
  }
}
```

За замовчуванням це pure pipe. А це означає, що він може відстежувати зміну значення, яке йому передається, оскільки воно є типом number.

У компоненті ми могли динамічно змінювати значення, для якого виконується форматування:

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

@Component({
    selector: 'my-app',
    template: `<input [(ngModel)]="num" name="fact">
```

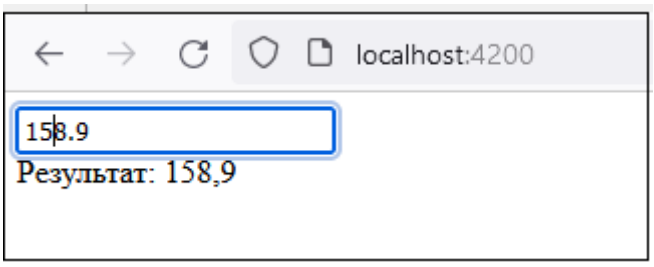
```
    <div>Результат: {{num | format}}</div>`
})
export class AppComponent {

    num: number = 15.45;
}
```

У файлі app.module.ts підключимо FormsModule, щоб використовувати двосторонню прив'язку:

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

Тут жодних проблем із введенням би не виникло - змінюємо число в текстовому полі, і відразу змінюється форматований результат:



Створимо інший pipe:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'join'
})
export class JoinPipe implements PipeTransform {
  transform(array: any, start?: any, end?: any): any {
    return array.join(", ");
  }
}
```

Цей pipe виконує операції над масивом. Відповідно, якщо в компоненті динамічно додавати нові елементи в масив, до якого застосовується JoinPipe, то ми не побачимо змін. Так як JoinPipe не відстежуватиме зміни над масивом.

Тепер зробимо його impure pipe. Для цього додамо до декоратора Pipe параметр pure: false:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
    name: 'join',
    pure: false
})
export class JoinPipe implements PipeTransform {
  transform(array: any, start?: any, end?: any): any {
    return array.join(", ");
  }
}
```

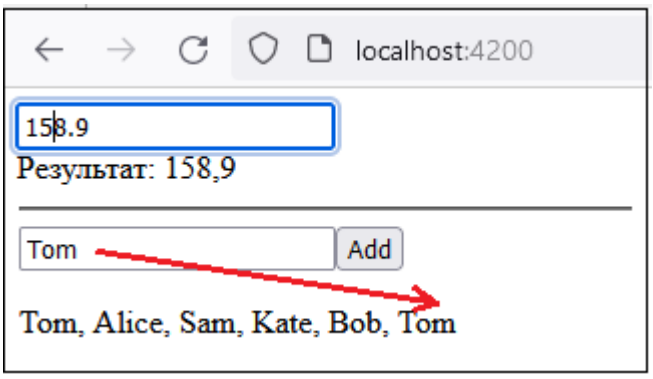
За замовчуванням параметр pure дорівнює true.

Тепер ми можемо додавати в компонент нові елементи в цей масив:

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

@Component({
    selector: 'my-app',
    template: `
    <input [(ngModel)]="num" name="fact">
    <div>Результат: {{num | format}}</div>
    <hr/>
                <input #user name="user" class="form-control">
                <button class="btn" (click)="users.push(user.value)">Add</button>
                <p>{{users | join}}</p>`
})
export class AppComponent {
    num: number = 15.45;
    users = ["Tom", "Alice", "Sam", "Kate", "Bob"];
}
```

І до всіх доданих елементів також застосовуватиметься JoinPipe:



Коли додається новий елемент, клас JoinPipe знову починає обробляти масив. Тому pipe застосовується до всіх елементів.

AsyncPipe

Одним із вбудованих класів, який на відміну від інших pipes вже за замовчуванням є тип impure. AsyncPipe дозволяє отримати результат асинхронної операції.

AsyncPipe відстежує об'єкти Observable та Promise та повертає отримане з цих об'єктів значення. При отриманні значення AsyncPipe сигналізує компонент про те, що треба перевірити зміни. Якщо компонент знищується, AsyncPipe автоматично відписується від об'єктів Observable і Promise, що унеможливлює можливі витоки пам'яті.

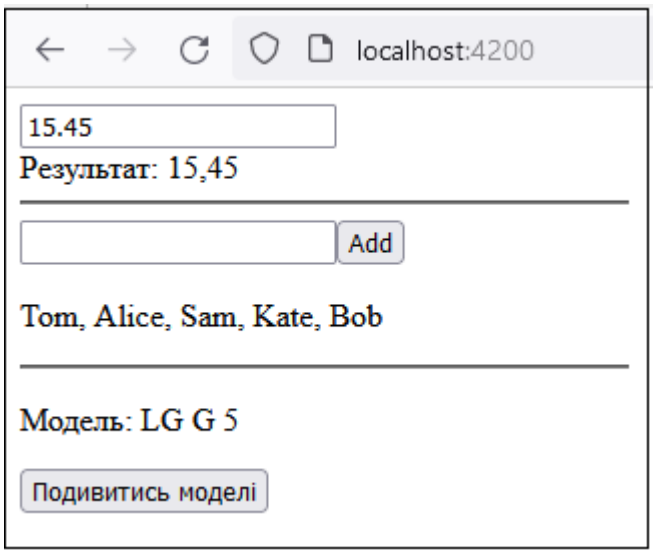
Використовуємо AsyncPipe:

```
e:\Angular_App\Pipes_2\
import { Component} from '@angular/core';
import { Observable, interval } from 'rxjs';
import { map } from 'rxjs/operators';

@Component({
    selector: 'my-app',
    template: `
<input [(ngModel)]="num" name="fact">
    <div>Результат: {{num | format}}</div>
   <hr/>
    <input #user name="user" class="form-control">
    <button class="btn" (click)="users.push(user.value)">Add</button>
    <p>{{users | join}}</p>
            <p>Модель: {{ phone| async }}</p>
            <button (click)="showPhones()">Подивитись моделі</button>`
})
export class AppComponent {
  num: number = 15.45;
  users = ["Tom", "Alice", "Sam", "Kate", "Bob"];
    phones = ["iPhone 7", "LG G 5", "Honor 9", "Idol S4", "Nexus 6P"];
    phone: Observable<string>|undefined;
```

```
  constructor() { this.showPhones(); }
  showPhones() {
     this.phone = interval(500).pipe(map((i:number)=> this.phones[i]));
  }
}
```

Тут з періодичністю 500 мілісекунд у шаблон компонента передається черговий елемент з масиву phones.



Компонент не повинен підписуватись на асинхронне отримання даних, обробляти їх, а при знищенні відписуватись від отримання даних. Всю цю роботу робить AsyncPipe.

Можно використовувати інструкцію user$ | async as user у зв’язці зі структурними директивами, такими як *ngIf и *ngFor, щоб не підписуватись два рази на одні й ті ж дані:

```
e:\Angular_App\Pipes3\
import { Component } from '@angular/core';
import { HttpClient } from '@angular/common/http';

@Component({
  selector: 'app-root',
  template: `
  <div *ngIf="user$ | async as user">
  name: {{ user.name }}, id: {{user.id }}
  </div>
`,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'Pipes3';
  readonly user$ = this.http.get<{name: string, id: string}>(
    'https://jsonplaceholder.typicode.com/users/2'
```

```
  );

  constructor(private readonly http: HttpClient){}
}
```

У даному випадку, ми підписуємось на user$ при допомозі async в шаблоні та поміщуємо отримане значення з user$ в змінну user, яку можемо використати всередиі тега .

Можна одразу звертатися до властивості асинхронного об'єкта, якщо обернути вираз user$ | async в круглі скобки:

```
*ngIf="(user$ | async).name as name"
```

Однак у рантаймі буде видано помилку в консоль про те, що неможливо прочитати властивість name. У пайпі async перший виклик повертає null, саме тому виникає проблема зі зверненням до вкладених властивостей. А щоб уникнути цього, треба скористатися Optional Chaining і додати? перед зверненням до властивості:

```
*ngIf="(user$ | async)?.name as name"
```

Angular. async pipe в порівнянні з ngIf

Звичайний *ngIf else

```
e:\Angular_App\Pipes3\
<ng-container
  *ngIf="loading; then thenBlock else elseBlock">
  </ng-container>

  <ng-template #thenBlock>
  Then template <br/>
  </ng-template>

  <ng-template #elseBlock>
  <img src="assets/Picture.png" width="250" height="250" alt="Вивчаємо pipes">
  <br/>
  </ng-template>
```

ngIf async pipe

```
e:\Angular_App\Pipes3\
<ng-container *ngIf="(user$ | async) as user; else elseBlock">
  <h1>name: {{ user.name }}, id: {{user.id }}</h1>
  </ng-container>
```

```
  <ng-template #elseBlock>
  <img src="assets/Picture.png" width="250" height="250" alt="Вивчаємо pipes">
  <br/>
  </ng-template>
```

У прикладі user$ - це Observable:

```
user$ : Observable<{name: string, id: string}>| undefined;
```

Зверніть увагу на "as user" в кінці виразу. Змінній user буде призначено значення (user$ | async), коли воно буде доступне. Змінна user повинна використовуватися всередині структурної директиви *ngIf.

Angular HttpClient

У Angular є власний модуль HTTP, який працює з додатками Angular. Він використовує бібліотеку RxJS для обробки асинхронних запитів і надає багато варіантів для виконання запитів HTTP.

Здійснення запиту на сервер за допомогою Angular HttpClient

Перше, що нам потрібно зробити — імпортувати HttpClientModule до app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {HttpClientModule} from '@angular/common/http';
import { AppComponent } from './app.component';

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

Потім ми повинні створити сервіс для обробки запитів. Ви можете легко створити сервіс за допомогою Angular CLI:

```
ng g service  FetchdataService
```

Далі нам потрібно імпортувати HttpClient до сервісу fetchdataService.ts і вставити його всередині конструктора:

```
import { Injectable } from '@angular/core';
import {HttpClient} from '@angular/common/http';

@Injectable()
export class FetchdataService {

  constructor(private http:HttpClient) { }

  getData(url:string){
    return this.http.get(url)
  }
}
```

А в app.component.ts потрібно імпортувати fetchdataService :

```
import { FetchdataService } from './fetchdata.service';
```

Також треба створити файл post.ts

```
export class Post{
constructor(public userId:number, public id:number,public title:string, public body:string){}
}
```

Зрештою, потрібно викликати сервіс і запустити його.

app.component.ts:

```
import { Component ,OnInit} from '@angular/core';
import { FetchdataService } from './fetchdata.service';
import { Observable } from 'rxjs/Observable';
import {Post} from './post';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers:[FetchdataService]
})
export class AppComponent implements OnInit{
posts!: Post[];
```

```
  title='Angular HttpClient';
  Url = "https://jsonplaceholder.typicode.com/posts"

  // inject FetchdataService service
  constructor(private srv: FetchdataService) { }

  getPosts() : void {
    this.srv.getData(this.Url)
      .subscribe(
        (data:any)  => this.posts.push(data),
        error=> console.log(error)
      )
  }
  ngOnInit(){
    this.getPosts()
  }
}
```

Шаблон app.component.html

```
<div class="container">
  <div class="item" *ngFor="let post of posts">
    <h3 class="title">{{post.title}}</h3>
    <div class="item-body">
      {{post.body}}
    </div>
  </div>
</div>
<router-outlet></router-outlet>
```

Посилальні змінні шаблону

У даних прикладах використовуються посилальні змінні шаблону. Шаблонні змінні дозволяють визначити деякі змінні всередині шаблону компонента, а потім посилатися до цих змінних з цього ж шаблону. Для визначення подібних змінних застосовується знак (#). Наприклад, визначимо шаблонну змінну userName у компоненті:

```
e:\Angular_App\Pipes3\
import { Component } from '@angular/core';
```

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

Визначення змінної виглядає так:

```
<p #userName>{{name}}</p>
```

Визначення змінної userName в елементі параграфа означає, що вона представлятиме цей параграф, тобто елемент <p> розмітки html. І далі ми можемо звертатися до цього параграфу через цю змінну. Наприклад, за допомогою userName.textContent можна отримати текстовий вміст параграфа. При цьому якщо прив'язане до параграфа значення змінної name зміниться, то відповідно зміниться і значення userName.textContent.

При цьому змінну ми можемо використовувати тільки всередині шаблону.

Змінні шаблону допомагають використовувати дані з однієї частини шаблону в іншій частині. Посилальні змінні шаблону використовуються для виконання таких завдань, як реагування на введення даних користувачем або для тонкого настроювання форм програми.

Змінна шаблону може стосуватися наступного:

- элементу DOM в шаблоні
- директиви або компоненту
- TemplateRef з ng-шаблону
- веб-компоненту

Як Angular надає значення змінним шаблону¶

Angular надає змінній шаблону значення залежно від того, де ця змінна була оголошена:

• Якщо ви оголосили змінну в компоненті, змінна посилається на екземпляр компонента.

• Якщо ви оголосите змінну в стандартному HTML-тегу, змінна посилатиметься на елемент.

• Якщо ви оголошуєте змінну в елементі <ng-template>, змінна посилається на екземпляр TemplateRef, який представляє шаблон.

Область видимості шаблонних змінних

Подібно до змінних у коді JavaScript або TypeScript, змінні шаблону прив'язані до шаблону, який їх оголошує. Аналогічно, структурні директиви, такі як *ngIf та *ngFor, або оголошення <ng-template> створюють нову вкладену область видимості шаблону, подібно до того, як оператори потоку управління JavaScript, такі як if і for, створюють нові лексичні області видимості. Ви не можете отримати доступ до змінних шаблону всередині однієї з цих структурних директив ззовні її границь.

Доступ у вкладеному шаблоні

Внутрішній шаблон може звертатися до змінних шаблону, який визначає зовнішній шаблон. У наступному прикладі зміна тексту в <input> змінює значення в , оскільки Angular негайно оновлює зміни через змінну шаблону, ref1.

```
<input #ref1 type="text" [(ngModel)]="firstExample" />
<span *ngIf="true">Value: {{ ref1.value }}</span>
```

У цьому випадку *ngIf на створює нову область видимості шаблону, яка включає змінну ref1 з її батьківської області видимості.

Однак, звернення до змінної шаблону з дочірньої області видимості в батьківському шаблоні не працює:

```
<input
  *ngIf="true"
  #ref2
  type="text"
  [(ngModel)]="secondExample"
/>
<span>Value: {{ ref2?.value }}</span>
<!-- doesn't work -->
```

Тут ref2 оголошена в дочірній області видимості, що створена при допомозі *ngIf, і недоступна з батьківського шаблону.

Використання Observable для передачі значень

Observables забезпечують підтримку передачі повідомлень між частинами програми. Вони часто використовуються в Angular і служать для обробки подій, асинхронного програмування та роботи з кількома значеннями.

Observables є декларативними — тобто ви визначаєте функцію для опублікування значень, але вона не виконується доти, доки на неї не підпишеться споживач. Підписаний споживач отримує повідомлення до тих пір, поки функція не завершиться, або доки він не відмовиться від підписки.

Observables може передавати кілька значень будь-якого типу - літерали, повідомлення чи події, залежно від контексту. API для отримання значень однакові незалежно від того, синхронно або асинхронно вони передаються. Оскільки логіка установки та видалення обробляється observables, код програми повинен бути стурбований лише підпискою на споживання значень та скасуванням підписки. Будь то потік натискання клавіш, HTTP-відповідь або інтервальний таймер, інтерфейс для прослуховування значень і припинення прослуховування однаковий.

Завдяки цим перевагам observables широко використовуються в Angular, а також при розробці додатків.

Основні способи використання та терміни¶

Як видавець ви створюєте екземпляр Observable, який визначає функцію subscriber. Саме це функція виконується, коли споживач викликає метод subscribe(). Функція підписника визначає, як отримувати чи генерувати значення чи повідомлення для публікації.

Щоб запустити створені observables і почати отримувати повідомлення, необхідно викликати його метод subscribe(), передавши йому observer. Це об'єкт JavaScript, який визначає обробників отриманих повідомлень. Виклик subscribe() повертає об'єкт Subscription, що має метод unsubscribe(), який викликається для припинення отримання повідомлень.

e:\Project(RxJS)\Project3\index4.js

```
const observable = new rxjs.Observable((subscriber) => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);
  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();
```

```
  }, 1000);
 });

 console.log('just before subscribe');
 observable.subscribe({
  next(x) {
   console.log('got value ' + x);
  },
  error(err) {
   console.error('something wrong occurred: ' + err);
  },
  complete() {
   console.log('done');
  },
 });
 console.log('just after subscribe');
```

Результат:

```
just before subscribe
got value 1
got value 2
got value 3
just after subscribe
got value 4
done
```

Pull проти Push

Pull і Push – це два різні протоколи, які описують, як видавець даних може взаємодіяти зі споживачем даних.

У системах Pull підписник визначає коли він отримає дані від видавця даних. Сам видавець не знає, коли дані буде доставлено підписникам.

Кожна функція JavaScript є системою Pull. Функція є джерелом даних, а код, що викликає функцію, споживає їх, «витягуючи» одне значення, що повертається зі свого виклику.

В ES2015 представлені функції-генератори та ітератори (function *), ще один тип системи Pull. Код, який викликає iterator.next(), є споживачем, який «витягує» кілька значень з ітератора (Видавця).

	Видавець	Споживач (Підписник)
Pull	**Пасивний:** створює дані на запит.	**Активний:** вирішує, коли запитуються дані.
Push	**Активний:** виробляє дані у своєму власному темпі.	**Пасивний:** реагує на отримані дані.

У системах Push видавець визначає, коли надсилати дані споживачеві. Споживач (Підписник) не знає, коли він отримає ці дані.

Проміси на сьогоднішній день є найбільш поширеним типом системи Push в JavaScript. Promise (Видавець) надає обчислене значення зареєстрованим зворотним викликам (Споживачам), але на відміну від функцій саме Promise відповідає за точне визначення того, коли це значення передається у зворотні виклики.

RxJS представляє Observables, нову систему Push для JavaScript. Observable - це виробник кількох значень, що «пересилає» їх споживачам (підписникам).

- Функція — це ліниво обчислюване обчислення, яке синхронно повертає одне значення під час виклику.
- Генератор - це обчислення, що ліниво обчислюється, яке синхронно повертає від нуля до (потенційно) нескінченної кількості значень.
- Promise - це обчислення, яке може (або не може) зрештою повернути одне значення.
- Observable - це ліниво обчислюване обчислення, яке може синхронно або асинхронно повертати від нуля до (потенційно) нескінченної кількості значень з моменту його виклику.

В RxJS є методи, які призначені для перетворення Observable на Promises.

Спостерігаючі як узагальнення функцій

Всупереч поширеним твердженням, Observables не схожі на EventEmitters і не на Promises для декількох значень. У деяких випадках Observables можуть діяти як EventEmitters, а саме коли вони використовуються в multicasted Observable разом з Subjects RxJS, але зазвичай вони не діють як EventEmitters.

Observables схожі на функції з нульовою кількістю аргументів, але узагальнюють їх, допускаючі кілька значень.

e:\Project(RxJS)\Project3\index4.js

```
function foo() {
console.log('Hello');
return 42;
}

const x = foo();
console.log(x);
const y = foo();
console.log(y);
```

Результат:

```
Hello
42
Hello
42
```

Ви можете написати ту ж поведінку, що й вище, але з Observables:

```
import {Observable} from 'rxjs';
const foo$=new rxjs.Observable((subscriber) => {
console.log('Hello');
subscriber.next(42);
});

foo$.subscribe((x) => {
console.log(x);
});
foo$.subscribe((y) => {
console.log(y);
});
```

Результат буде той самий:

```
Hello
42
Hello
42
```

Це відбувається тому, що і функції, і Observables є лінивими обчисленнями. Якщо ви не викличете функцію, то виведення console.log('Hello') не станеться. Також із Observables, якщо ви не «викличете» його (за допомогою subscribe), то console.log('Hello') не станеться. Крім того, "виклик" або "підписка" - це ізольована операція: два виклики функції призводять до отримання двох окремих результатів, а дві окремі підписки Observable також призводять до двох окремих результатів. На відміну від EventEmitters, які мають загальні результати та мають швидке виконання незалежно від наявності підписників, Observables не мають спільного виконання та є лінивими.

Підписка Observable аналогічна виклику функції.

Є ствердження, що Observables асинхронні. Але Observables можуть працювати як в синхронному так і в асинхронному режимі. Наприклад:

```
console.log('before');
console.log(foo.call());
console.log('after');
```

Результат буде наступний:

```
before
Hello
42
after
```

І та ж сама поведінка з Observables:

```
console.log('before');
foo$.subscribe((x) => {
console.log(x);
});
console.log('after');
```

І результат:

```
before
Hello
42
after
```

Це доводить, що підписка foo$ була повністю синхронною, як функція.

Observables можуть доставляти значення синхронно чи асинхронно.

У чому різниця між Observable та функцією? Observables можуть "повертати" кілька значень з часом, чого не можуть зробити функції. Ви не можете зробити це:

```
function foo() {
console.log('Hello');
return 42;
return 100; // Помилка
}
```

Функції можуть повертати лише одне значення. Observables можуть повертати декілька значень:

```
import {Observable} from 'rxjs';

const foo=new rxjs.Observable((subscriber) => {
console.log('Hello');
subscriber.next(42);
subscriber.next(100);
subscriber.next(200);
});

console.log('before');
foo.subscribe((x) => {
console.log(x);
});
console.log('after');
```

З синхронним виходом:

```
before
Hello
42
100
200
after
```

Але також можна "повертати" значення асинхронно:

```
import {Observable} from 'rxjs';
```

```
const foo=new rxjs.Observable((Subscriber) => {
console.log('Hello');
subscriber.next(42);
subscriber.next(100);
subscriber.next(200);
setTimeout(() => {
subscriber.next(300);
}, 1000);
});
console.log('before');
foo.subscribe((x) => {
console.log(x);
});
console.log('after');
```

Результат:

```
before
Hello
42
100
200
after
300
```

Висновок:

- func.call() означає "дайте мені одне значення синхронно "
- observable.subscribe() означає " дайте мені будь-яку кількість значень, синхронно чи асинхронно".

Анатомія об’єкта Observable

Observables створюються за допомогою new Observable чи при допомозі операторів, підписуються з використанням Observer, виконуються для доставки повідомлень при допомозі нотифікацій next/error/complete об’єкта Observer і їх виконання може бути видалено . Всі ці чотири аспекти закодовані в екземплярі Observable, але деякі з цих аспектів пов'язані з іншими типами, такими як Observer та Subscription.

Основні моменти роботи з Observable:

- **створення** Observables
- **підписка н**а Observables
- **виконання** Observable
- **видалення** Observables.

Створення Observables

Конструктор Observable приймає один аргумент: subscribe функцію.

У наступному прикладі створюється Observable, який кожну секунду передає рядок 'hi' підписнику.

```
import { Observable } from 'rxjs';
const observable = new rxjs.Observable(function subscribe(subscriber) {
  const id = setInterval(() => {
    subscriber.next('hi');
  }, 1000);
});
```

Observables можна створювати за допомогою new Observable. Але найчастіше Observables створюються з використанням функцій створення, таких як of, from, interval і т.д.

У наведеному вище прикладі subscribe функція є найважливішою частиною опису Observable.

Підписка Observables

На створюваний об’єкт observable у прикладі вище можна підписатися так:

```
observable.subscribe((x) => console.log(x));
```

Не випадково observable.subscribe і функція subscribe в new Observable(function subscribe(subscriber) {...}) носять одне й те саме ім'я. У бібліотеці вони різні, але для практичних цілей їх можна вважати концептуально рівними.

Це показує, як subscribe виклики не розподіляються між кількома Observers одного і того ж Observable. При виклику observable.subscribe за допомогою Observer функція subscribe в new Observable(function subscribe(subscriber) {...}) запускається для даного підписника. Кожен виклик observable.subscribe викликає власне незалежне налаштування для цього підписника.

Підписка на Observable схожа на виклик функції, що забезпечує зворотні виклики, коли будуть доставлені дані.

Це кардинально відрізняється від API-інтерфейсів обробників подій, таких як addEventListener/removeEventListener. При використанні observable.subscribe кожен Observer не реєструється як слухач у Observable. Observable навіть не підтримує перелік прикріплених спостерігачів.

Виклик subscribe - це просто спосіб запустити "Спостережуване виконання" і доставити значення або події спостерігачеві (Observer) цього виконання.

Таким чином, об'єкт Observable виступає як постачальник даних, який має обробники даних. Обробники виконують роль користувачів, які реагують на надсилання постачальником даних.

При створенні Observable конструктор класу приймає функцію з набором callback-функцій як аргумент. У переданій функції описується логіка обробки та повернення значень.

Об'єкт, що приймається функцією, реалізує інтерфейс із трьома методами:

- next()- приймає значення, яке буде повернуто функції-обробнику;
- error()- приймає значення, що повертається функції-обробнику помилок;
- complete()- викликається для повідомлення "підписників" про закінчення розсилки.

Для обробки даних використовується метод subscribe(), який також приймає три функції: next, error і complete — для кожного з методів об'єкта конструктора.

Повідомлення «next» є найбільш важливим і поширеним типом: вони є фактичними даними, що подаються підписнику. Повідомлення «Error» та «Complete» можуть з'являтися лише один раз під час Observable виконання, і може бути лише одне з них.

Ці обмеження найкраще виражені у так званій Observable граматиці або Contract, записаній у вигляді регулярного виразу:

```
next*(error|complete)?
```

При Observable виконанні може бути доставлено від нуля до нескінченності повідомлень Next. Якщо надіслано повідомлення «Error» або «Complete», то більше нічого не може бути доставлено згодом.

Нижче наведено приклад виконання Observable, який доставляє три повідомлення Next, а потім завершується:

```
import {Observable} from 'rxjs';
```

```
constobservable= new rxjs.Observable(function subscribe(subscriber) {
subscriber.next(1);
subscriber.next(2);
subscriber.next(3);
subscriber.complete();
});
```

Observables суворо дотримуються Observable Contract, тому наступний код не доставлятиме повідомлення Next 4:

```
import {Observable} from 'rxjs';
constobservable= newObservable(function subscribe(subscriber) {
subscriber.next(1);
subscriber.next(2);
subscriber.next(3);
subscriber.complete();
subscriber.next(4);
});
```

Хорошою ідеєю буде обернути будь-який код з subscribe за допомогою try/catch block, який доставлятиме повідомлення про помилку, якщо він перехопить виняток:

```
import { Observable } from 'rxjs';
const observable = new rxjs.Observable(function subscribe(subscriber) {
  try {
    subscriber.next(1);
    subscriber.next(2);
    subscriber.next(3);
    subscriber.complete();
  } catch (err) {
    subscriber.error(err); // delivers an error if it caught one
  }
});
```

Видалення виконаних виконань

Оскільки Observable виконання можуть бути нескінченними, а спостерігач часто хоче перервати виконання за кінцевий час, нам потрібен API для скасування виконання. Оскільки кожне виконання є ексклюзивним тільки для одного Observer, як тільки Observer завершує отримання значень, він повинен мати спосіб зупинити виконання, щоб уникнути втрати обчислювальної потужності або ресурсів пам'яті.

При виклику observable.subscribe Observer приєднується до новоствореного виконання Observable. Цей виклик також повертає об'єкт Subscription:

```
const subscription = observable.subscribe((x) => console.log(x));
```

Підписка є поточним виконанням і має мінімальний API, який дозволяє скасувати це виконання. За допомогою subscription.unsubscribe() можна скасувати поточне виконання:

Наприклад, скасовуємо підписку за натисканням на кнопку. Виведення в консоль чисел з інтервалом у секунду:

e:\Project(RxJS)\Project5\index.js

```
const rxjsBtn = document.getElementById('rxjs')
const intervalStream$=interval(1000)
sub = intervalStream$.subscribe((value)=>{
console.log(value)
})

rxjsBtn.addEventListener('click',()=>{
sub.unsubscribe();
})
```

Приклад: Виведення координат миші x і y заголовок h1. Скасуємо підписку при натискання на кнопку.

e:\Project(RxJS)\Project5\index.js

```
const rxjsBtn=document.getElementById('rxjs')
sub$=fromEvent(document,'mousemove')
.subscribe(e=>{
 document.querySelector('h1').innerHTML=`X:=${e.clientX}, Y:${e.clientY}`;
})
```

```
rxjsBtn.addEventListener('click',()=>{
 sub$.unsubscribe();
})
```

Коли ви підписуєтеся, ви отримуєте об’єкт Subscription, який є поточним виконанням. Щоб скасувати виконання треба викликати unsubscribe().