Лекція №7. Відписка Unsibscribe

Видалення виконаних підписок

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().

Відписка в Angular

Якщо ми підпишемося на потік, він залишиться відкритим і буде викликатися щоразу, коли в нього передаються значення з будь-якої частини програми, доки він не буде закритий за допомогою виклику unsubscribe.

e:\Angular_App\Unsibscribe_Components\

```
import { Component, OnInit } from '@angular/core';
import { Subscription, interval} from 'rxjs';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit {
  subscription: Subscription
  ngOnInit () {
     const observable = interval(1000);
     this.subscription = observable.subscribe(x => console.log(x));
  }
}
```

В цій реалізації ми створюємо стрім, використовуючи метод інтервал для відправки значень кожну секунду, підписуємося на нього, щоб отримати відправлене значення, а наша callback-функція пише результат у консоль браузера.

Тепер, якщо AppComponent буде знищений, наприклад після виходу з компонента або за допомогою методу destroy(), ми все одно побачимо лог консолі в браузері. Це пов'язано з тим, що хоча AppComponent був знищений, підписка не була скасована.

Якщо підписка не закрита, callback-функція безперервно викликатиметься, що призведе до серйозного витоку пам'яті та проблем з продуктивністю. Щоб уникнути витоків необхідно «відписуватися» від Observable.

Чому і коли ми повинні відписуватися від Observable

Коли ми працюємо з RxJS, правильне забезпечення відписки має вирішальне значення для запобігання витоку пам’яті в застосунку. Проблема полягає в тому, що коли компонент руйнується, наявні підписки в ньому залишаються і все ще спрацьовують (навіть якщо компонент уже не існує). Підписки зберігають посилання на об’єкти і таким чином не дають «збирачу сміття» Javascript звільнити пам’ять, що й призводить до витоків пам’яті.

Деколи Angular здатний самостійно автоматично відписуватися, наприклад, якщо ми використовуємо AsyncPipe. Але це не завжди можливо. Як правило, нам необхідно відписуватися вручну, коли ми:

- маємо справу з будь-яким Observable з довгим (або нескінченним) строком життя (так звані long-lived Observables, які не завершуються самі собою);
- маємо потребу отримати доступ до значення в самому компоненті, тобто ми не можемо використати AsyncPipe (в іншому разі рекомендовано використовувати AsyncPipe).

Деякі приклади:

- Observables, які створюються на основі дій користувача (і таким чином є безтерміновими), наприклад, на основі click-подій за допомогою оператора fromEvent.
- Форми — наприклад, треба відписуватися від valueChanges.
- Subjects (у більшості випадків). Їх зазвичай використовують для стейт-менеджменту.
- NgRx store. Бібліотека NgRx реалізує принцип роботи Redux для додатків Angular. Головна мета NgRx — централізувати та зробити максимально зрозумілим керування всіма станами програми. Мета досягається завдяки закладеним у бібліотеці кількох фундаментальним принципам: Наявність

єдиного джерела даних про стан - сховища (store); Доступність стану лише читання; Зміна стану здійснюється тільки через дії (actions), які обробляються редюсерами (reducer), що є чистими функціями.

- Деякі оператори RxJS, які продукують нескінченний потік значень. Наприклад, оператор interval.
- Observables, які належать до API маршрутизатора, наприклад router.events.

У Angular маршрутизація представляє собою перехід від одного представлення (шаблону) до іншого залежно від заданого URL. Причому навігація може здійснюватись і всередині представлення.

Навігація в додатках Angular відбувається без перезавантаження сторінки.

Ключова роль у формуванні URL належить тегу <base>, що вказує шлях до додатку відносно розташування файлу index.html, то б то базову адресу додатку. Якщо index.html розташовується в директорії клієнтського додатка, тег повинен бути записаний наступним чином:

<base href="/" />

Якби клієнтська програма знаходилася в директорії example, а index.html на одному рівні ієрархії з нею, то було б так:

<base href="/example" />

За організацію маршрутизації до Angular відповідає модуль RouterModule бібліотеки @angular/router. URL-адреси організуються в спеціальні модулі та визначаються для кожного окремого модуля програми.

Щоб застосувати маршрути, вони передаються в метод RouterModule.forRoot(appRoutes).

Метод RouterModule.forRoot() повертає модуль, який містить сконфігурований сервіс Router. Коли додаток завантажується, Router виконує початкову навігацію за поточною URL-адресою, яка знаходиться в адресному рядку браузера.

Щоб можна було впровадити в AppComponent той компонент, який обробляє запит, необхідно використовувати елемент RouterOutlet. На місце елемента <router-outlet> буде рендеритись компонент, вибраний для обробки запиту.

Відповідно до офіційної документації Angular повинен сам відписуватись від подій роутера, але це не відбувається, тому відписуватись необхідно самостійно.

Щоразу, коли в Angular програмі здійснюється навігація, Router сервіс ініціює ряд подій:

- NavigationStart - початок навігації;

- RoutesRecognized — завершення процесу парсингу URL та розпізнавання маршрутів;
- RouteConfigLoadStart - ініціюється безпосередньо перед асинхронним завантаженням маршрутів;
- RouteConfigLoadEnd - ініціюється безпосередньо після асинхронного завантаження маршрутів;
- NavigationEnd - завершення навігації;
- NavigationCancel - Навігація відхилена, виникає, коли guard повертає false;
- NavigationError - виникнення непередбаченої помилки в процесі здійснення навігації.

Наведені вище події можуть бути оброблені в будь-якому компоненті або сервісі програми. Щоб визначити для них обробники, необхідно підписатися на властивість events сервісу Router.

Властивість events об'єкта router повертає об'єкт типу Observable<Event>, який можна використовувати для відстеження навігаційних змін.

Наприклад, визначимо обробники в компоненті AppComponent

e:\Angular_App\Unsibscribe_Components_2\

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subscription} from 'rxjs';
import { Router, NavigationEnd, NavigationStart } from '@angular/router';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss']
})
export class AppComponent implements OnInit, OnDestroy {
  currentRouteUrl: string = '';
  routerSub: Subscription;
  ngOnInit () {
    this.routerSub=this.router.events.subscribe((event: any) => {
      if (event instanceof NavigationStart) {
        if (event.url !== this.currentRouteUrl) {
          this.trackEvent(event.url);
        }
```

```
      }
      if (event instanceof NavigationEnd) {
        this.currentRouteUrl = event.url;
        console.log(event);
      }
    });
  }
  ngOnDestroy() {
  this.routerSub.unsubscribe();
}
constructor(private router: Router) {}
trackEvent(event_url:string) {
  alert("Завантажується сторінка: " + event_url);
}
}
```

Наприклад, наведене вище можна використовувати для відображення/приховування індикатора завантаження сторінки.

Зауважте, що класи подій повинні бути імпортовані з бібліотеки @angular/router.

```
import { Router, NavigationEnd, NavigationStart } from '@angular/router';
```

Отже, проблема була, але довгий час не існувало загальноприйнятого методу її вирішення (існувало декілька способів, які використовувалися, для вирішення цієї задачі). І ось, в Angular 16 з'явилася довгоочікувана функція — «офіційний» та зручний метод відписки від Observables, який, безумовно, поступово замінить усі інші способи. В старому коді часто ще зустрічатимуться ці методи, тому корисно знати, як вони працюють.

Використання методу unsubscribe

Будь-який Subscription має функцію unsubscribe() для звільнення ресурсів та скасування виконання Observable. Щоб запобігти витоку пам'яті, необхідно скасувати підписки за допомогою методу unsubscribe в Observable.

У Angular потрібно відписатися від Observable, коли компонент знищується. У Angular є хук ngOnDestroy, який викликається перед знищенням компонента, що дозволяє

розробникам забезпечити очищення пам'яті, уникнути зависання підписок, відкритих портів і т. д.

```
@Component({…})
export class AppComponent implements OnInit, OnDestroy {
  subscription: Subscription
  ngOnInit () {
    const observable = interval(1000);
    this.subscription = observable.subscribe(x => console.log(x));
  }
  ngOnDestroy() {
    this.subscription.unsubscribe()
  }
}
```

Ми додали ngOnDestroy до нашого AppComponent і викликали метод unsubscribe на Observable this.subscription. Коли AppComponent буде знищено (за допомогою переходу за посиланням, методу destroy() тощо), підписка не зависатиме, інтервал буде зупинено, а в браузері більше не буде логів консолі.

А якщо у нас є декілька підписок?

```
@Component({…})
export class AppComponent implements OnInit, OnDestroy {
  subscription1$: Subscription;
  subscription2$: Subscription;
  ngOnInit () {
    const observable1$ = Rx.Observable.interval(1000);
    const observable2$ = Rx.Observable.interval(400);
    this.subscription1$ = observable.subscribe(x => console.log("From interval 1000" x));
    this.subscription2$ = observable.subscribe(x => console.log("From interval 400" x));
  }
  ngOnDestroy() {
    this.subscription1$.unsubscribe();
    this.subscription2$.unsubscribe();
  }
```

```
}
```

У AppComponent дві підписки та обидві відписалися в хуку ngOnDestroy, запобігаючи витоку пам'яті.

Також можна зібрати всі підписки в масив і відписатися від них у циклі:

```
@Component({…})
export class AppComponent implements OnInit, OnDestroy {
  subscription1$: Subscription;
  subscription2$: Subscription;
  subscriptions: Subscription[] = [];
  ngOnInit () {
    const observable1$ = Rx.Observable.interval(1000);
    const observable2$ = Rx.Observable.interval(400);
    this.subscription1$ = observable.subscribe(x => console.log("From interval 1000" x));
    this.subscription2$ = observable.subscribe(x => console.log("From interval 400" x));
    this.subscriptions.push(this.subscription1$);
    this.subscriptions.push(this.subscription2$);
  }
  ngOnDestroy() {
    this.subscriptions.forEach((subscription) => subscription.unsubscribe());
  }
}
```

Метод subscribe повертає об'єкт RxJS типу Subscription. Він є одноразовим ресурсом. Підписки можуть бути згруповані за допомогою методу add, який прикріпить дочірню пыдписку до поточної. Коли підписка скасовується, всі її дочірні елементи також відписуються. Спробуємо переписати наш AppComponent:

```
@Component({…})
export class AppComponent implements OnInit, OnDestroy {
  subscription: Subscription;
  ngOnInit () {
    const observable1$ = Rx.Observable.interval(1000);
    const observable2$ = Rx.Observable.interval(400);
    const subscription1$ = observable.subscribe(x => console.log("From interval 1000" x));
```

```
    const subscription2$ = observable.subscribe(x => console.log("From interval 400" x));
    this.subscription.add(subscription1$);
    this.subscription.add(subscription2$);
  }
  ngOnDestroy() {
    this.subscription.unsubscribe()
  }
}
```

Так ми відпишемо this.subscripton1$ та this.subscripton2$ у момент знищення компонента.

Використання Async | Pipe

Pipe async підписується на Observable або Promise та повертає останнє передане значення. Коли нове значення відправляється, pipe async перевыряє цей компонент на відстеження змін. Якщо компонент знищується, pipe async автоматично відписується.

```
@Component({
  ...,
  template: `
    <div>
    Interval: {{observable$ | async}}
    </div>
  `
})
export class AppComponent implements OnInit {
  observable$;
  ngOnInit () {
    this.observable$ = interval(1000);
  }
}
```

При ініціалізації AppComponent створить Observable із методу інтервалу. У шаблоні потік observable$ передається як async. Він підпишеться на observable$ і відобразить його значення в DOM. Також він скасує підписку, коли AppComponent буде

знищений. Pipe async у своєму класі містить хук ngOnDestroy, тому той буде викликаний, коли його view буде знищено.

Pipe async дуже зручно використовувати, тому що він сам підписується на Observable і відписується від них. І можно не турбуватися, якщо розробник забуде відписатися в ngOnDestroy.

Використання операторів RxJS take*

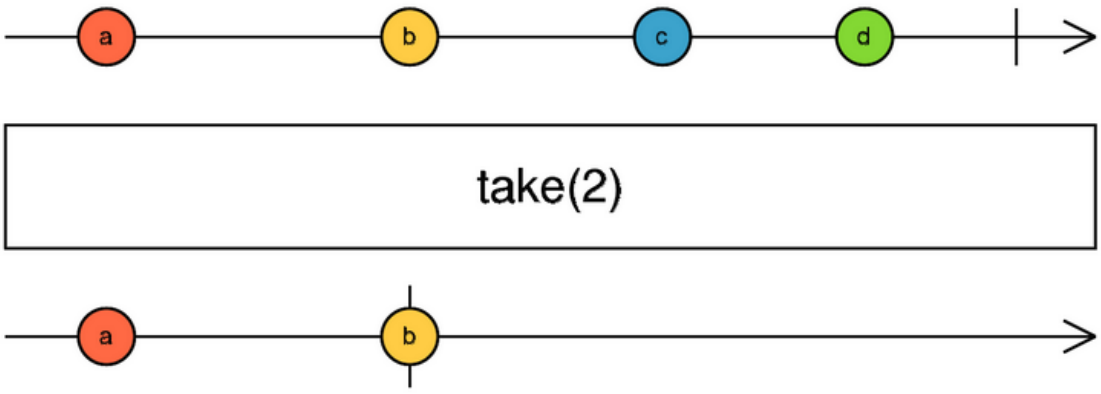
RxJS містить корисні оператори, які можна використовувати у декларативний спосіб, щоб скасовувати підписки в нашому Angular-проекті. Один із них — оператори сімейства *take**:

- **take(n)**
- **takeUntil(notifier)**
- **takeWhile(predicate)**

take(n)

Видає лише перші значення потоку, випущені джерелом Observable. То б то цей оператор emit-ить вихідну підписку зазначену кількість разів і завершується. Найчастіше в take передається одиниця (1) для підписки та виходу. Якщо джерело видає менше значень, ніж зазначена кількість в take, тоді видаються всі його значення. Після цього потік завершується, незалежно від того, чи завершується джерело.

Цей оператор корисно використовувати, якщо ми хочемо, щоб Observable передав значення один раз, а потім відписався від потоку:



Приклад роботи з take(n)

```
import { interval, take } from 'rxjs';
const intervalCount = interval(1000);
const takeFive = intervalCount.pipe(take(5));
```

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

// Logs:
// 0
// 1
// 2
// 3
// 4
```

Приклад роботи з take(n)

```
@Component({...})
export class AppComponent implements OnInit {
  subscription$;
  ngOnInit () {
    const observable$ = interval(1000);
    this.subscription$ = observable$.pipe(take(1)).subscribe(x => console.log(x));
  }
}
```

У даному випадку subscription$ скасує передплату, коли інтервал передасть перше значення.

Зверніть увагу: навіть якщо AppComponent буде знищено, subscription$ не скасує підписку, доки інтервал не передасть значення. Тому все одно краще переконатися, що все відписано в хуку ngOnDestroy:

```
@Component({…})
export class AppComponent implements OnInit, OnDestroy {
  subscription$;
  ngOnInit () {
    var observable$ = interval(1000);
    this.subscription$ = observable$.pipe(take(1)).subscribe(x => console.log(x));
  }
  ngOnDestroy() {
    this.subscription$.unsubscribe();
  }
}
```

takeUntil(notifier)

Нам потрібен спеціальний subject destroy$ і оператор takeUntil. Знову ж таки, треба імплементувати OnDestroy.

Оператор takeUntil виконує відписку, коли subject destroy$ видає значення (а це відбувається в хуку ngOnDestroy, тобто коли компонент знищується). При цьому, оператор takeUntil повинен бути останнім оператором у ланцюгу, інакше стає можливим витік пам'яті.

В документації сказано: takeUntil емітить значення, видані джерелом Observable доти, доки notifier Observable не випустить значення.

takeUntil підписується та починає віддзеркалювати джерело Observable. Він також відстежує другий Observable, який ви надаєте. Якщо notifier видає значення, вихідний Observable припиняє віддзеркалювати джерело Observable і завершує роботу. Якщо notifier не видає жодного значення та завершується, takeUntil передасть усі значення.

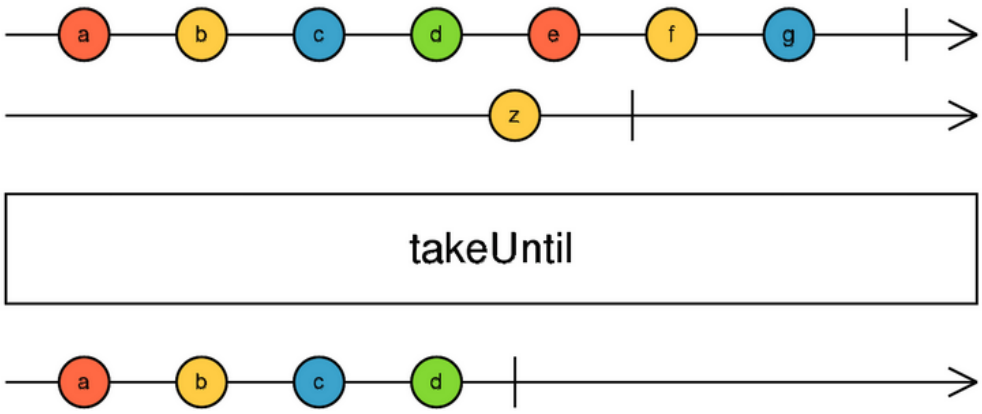


Рис. takeUntil дозволяє передавати значення, доки другий Observable, notifier (сповіщувач), не видасть значення. Потім він завершується.

Приклад: Створюємо два потоки. Перший потік емітить подію кожну секунду. Другий потік прослуховує клік по документу. Як тільки клік відбувся (потік clicks$ видав значення), потік source$ завершує свою роботу.

```
import { interval, fromEvent, takeUntil } from 'rxjs';
const source$ = interval(1000);
const clicks$ = fromEvent(document, 'click');
const result = source$.pipe(takeUntil(clicks$));
result.subscribe(x => console.log(x));
```

В Angular при використанні оператора takeUntil розробник також повинен подбати про те, щоб завершити сам subject destroy$.

```
import { Component, OnDestroy, OnInit } from '@angular/core';
import { Subject, interval, takeUntil } from 'rxjs';
@Component({
  standalone: true,
  selector: 'app-take-until-example',
  template: `<p>take-until-example works!</p>`,
})
export class TakeUntilComponent implements OnInit, OnDestroy {
  private destroy$ = new Subject<void>();
  ngOnInit() {
    interval(1000)
      .pipe(takeUntil(this.destroy$))
      .subscribe((value) => {
        console.log(value);
      });
  }
  ngOnDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
  }
}
```

В цьому прикладі, ми маємо додатковий Subject для повідомлень, який відправить команду, щоб відписати this.subscription. Ми пайпимо Observable в takeUntil до тих пір, поки ми підписані. TakeUntil буде емітить повідомлення інтервалу, поки destroy$ не скасує підписку observable$. Найзручніше поміщати notifier в хук ngOnDestroy.

Коротко про Subject

RxJS Subject є різновидністю об’єктів Observable. Особливість Subject в тому, що він може надсилати дані одночасно безлічі "споживачів", які можуть бути зареєстровані вже в процесі виконання Subject, у той час як виконання стандартного Observable здійснюється унікально для кожного його виклику.

Об'єкти RxJS Subject реалізують принцип роботи подій, підтримуючи можливість реєструвати необмежену кількість обробників, відправляємих ними даних.

Subject дозволяє не тільки посилати щось зі свого потоку, але й приймати до себе.

- На Subject можна підписуватися як на Observable
- Subject сам може підписуватися на інші Observable.

Є важлива особливість: коли Subject щось посилає, всі підписники отримують одні й ті ж дані.

Створення відбувається за допомогою new Subject(). Далі реєструються обробники викликом методу subscribe(), який приймає подібно до звичайного Observable три функції: next(), error() і complete().

Але тут обробники виконуються не відразу в момент виклику subscribe() (на відміну від Observable), а після звернення до методів next(), error() або complete() самого об’єкта. При чому реєстрація нових "споживачів" може відбуватися в будь-який момент часу. Но дані отримувати вони будуть вже починаючи з наступної розсилки.

Розглянемо приклад 1.

e:\Project(RxJS)\Project8\src\index.js

```
 var observable = rxjs.Observable.create(function(source) {
   source.next(Math.random());
 });

 observable.subscribe(v => console.log('consumer A: ' + v));
 observable.subscribe(v => console.log('consumer B: ' + v));
```

Результат:

```
consumer A: 0.5583628356583453
consumer B: 0.07767948079980302
```

Розглянемо приклад 2.

e:\Project(RxJS)\Project8\src\index.js

```
const sbj = new rxjs.Subject();
sbj.subscribe((vl) => console.log(`1st: ${vl}`));
sbj.next(Math.random());
sbj.subscribe((vl) => console.log(`2nd: ${vl}`));
sbj.next(Math.random());
```

Результат:

```
1st: 0.25560655366966223
1st: 0.9791443107699315
2nd: 0.9791443107699315
```

Розглянемо приклад 3. Отримання однакових даних підписниками Subject.

e:\Project(RxJS)\Project8\src\index.js

```
var observable = rxjs.Observable.create(function(source) {
   source.next(Math.random());
  });
  var subject = new rxjs.Subject();
subject.subscribe(v => console.log('consumer A: ' + v));
subject.subscribe(v => console.log('consumer B: ' + v));
observable.subscribe(subject);
```

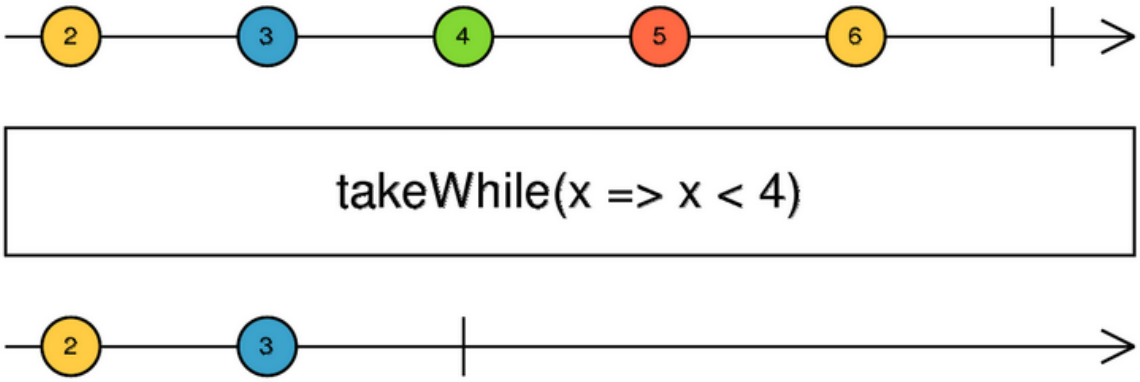
Результат:

```
consumer A: 0.9052747613114129
consumer B: 0.9052747613114129
```

takeWhile(predicate)

Цей оператор буде емітити значення Observable, поки вони відповідають умові предикату.

takeWhile підписується та починає віддзеркалювати джерело Observable. Кожне значення, видане джерелом, передається функції предикату, яка повертає логічне значення, що представляє умову, якій мають задовольняти вихідні значення. Вихідний Observable випромінює вихідні значення до тих пір, поки предикат не поверне false, після чого takeWhile припиняє віддзеркалювати вихідний Observable і завершує вихідний Observable.



Приклад:

Емітимо події кліків миші лише тоді, коли властивість clientX перевищує 200.

e:\Project(RxJS)\Project8\src\index.js

```
fromEvent(document, 'click')
.pipe(takeWhile(ev => ev.clientX > 200))
.subscribe(x => console.log(x));
```

Приклад. Створити observable$ з оператором takeWhile, який буде відправляти значення до тих пір, поки вони менше 10. Якщо прийде значення більше або дорівнює 10, оператор повинен скасувати підписку. Важливо розуміти, що підписка observable$ буде відкрита, доки інтервал не видасть 10. Тому для безпеки бажано додати хук ngOnDestroy, щоб відписатися від observable$, коли компонент знищений.

```
@Component({…})
export class AppComponent implements OnInit, OnDestroy {
  subscription$;
  ngOnInit () {
    var observable$ = Rx.Observable.interval(1000);
    this.subscription$ = observable$.pipe(takeWhile(value => value < 10)).subscribe(x => console.log(x));
  }
  ngOnDestroy() {
    this.subscription$.unsubscribe();
  }
}
```

Використання оператора RxJS first

Цей оператор схожий на об'єднаний take(1) та takeWhile.

Якщо він викликається без параметра, то emit-іт перше значення Observable і завершується. Якщо він викликається з функцією предикату, то emit-ит перше значення вихідного Observable, яке відповідає умові функції предикату, і завершується.

```
@Component({...})
export class AppComponent implements OnInit {
  observable$;
  ngOnInit () {
    this.observable = Rx.Observable.interval(1000);
    this.observable$.pipe(first()).subscribe(x => console.log(x));
  }
}
```

Observable$ завершиться, якщо інтервал передасть перше значення. Це означає, що в консолі ми побачимо лише одне повідомлення лога.

```
@Component({...})
export class AppComponent implements OnInit {
  observable$;
  ngOnInit () {
     this.observable$ = Rx.Observable.interval(1000);
    this.observable$.pipe(first(val => val === 10)).subscribe(x => console.log(x));
  }
}
```

Тут first не буде емітити значення, поки оператор interval не передасть десятку, а потім завершить observable$. У консолі побачимо лише одне повідомлення 10.

У першому прикладі, якщо AppComponent знищено до того, як first отримає значення з observable$, підписка буде, як і раніше, відкрита до отримання першого повідомлення. Також і у другому прикладі підписка буде, як і раніше, відкрита до тих пір, поки інтервал не віддасть 10. Тому, щоб забезпечити безпеку, ми повинні явно скасовувати підписки в хуку ngOnDestroy.

ngneat/until-destroy package

Попередні підходи є багатослівними і ускладнюють логіку компонентів. Саме тому npm-пакет ngneat/until-destroy набув популярності. Він додає синтаксичний цукор до шаблону untilDestroy за допомогою декоратора `UntilDestroy`.

Крім того, замість властивостей класів можна використовувати спеціальний оператор для RxJS під назвою untilDestroyed із цієї бібліотеки. Для цього потрібно застосувати декоратор UntilDestroy до нашого компоненту та додати оператор untilDestroyed у метод pipe() об’єкта Observable:

```
import { UntilDestroy, untilDestroyed } from '@ngneat/until-destroy';

@UntilDestroy()
@Component({})
export class InboxComponent {
 ngOnInit() {
  interval(1000).pipe(untilDestroyed(this)).subscribe();
 }
```

```
}
```

Якщо передати декоратору @UntilDestroy опцію { checkProperties: true }, то він автоматично перевірить поля класу, коли компонент буде знищено, і відпишеться від усіх підписок, які зберігаються як поля класу:

```
@UntilDestroy({ checkProperties: true })
@Component({})
export class HomeComponent {
  // We'll dispose it on destroy
  subscription = fromEvent(document, 'mousemove').subscribe();
}
```

Останні два підходи можуть бути скомбіновані:

```
@UntilDestroy({ checkProperties: true })
@Component({
  standalone: true,
  selector: 'app-auto-until-destroy-example',
  template: `
    <p>auto-until-destroy-example works!</p>
  `,
})
export class AutoUntilDestroyComponent implements OnInit {
  subscription = fromEvent(document, 'mousemove').subscribe();
  ngOnInit() {
    this.subscription.add(
      interval(1000)
        .pipe(untilDestroyed(this))
        .subscribe((value) => {
          console.log(value);
        })
    );
  }
}
```

Також можна створювати та використовувати власні Декоратори в Angular-проектах, щоб автоматично відписатися від усіх підписок у компоненті.

Оператор takeUntilDestroyed

Усі попередні підходи є правильними. Однак проблема полягала в тому, що різні розробники надавали перевагу різним підходам, що призводило до потенційного змішування цих підходів в межах одного проєкту.

Починаючи з 16 версії, у Angular з'явився оператор takeUntilDestroyed, який надає елегантний і простий підхід до відписки. Цей оператор усуває необхідність створення subject, реалізації хука OnDestroy і застосування декораторів — він уже має все, що нам потрібно. Для цього необхідно імпортувати його та додати до pipe перед .subscribe():

```
import { Component } from '@angular/core';
import { interval } from 'rxjs';
import { takeUntilDestroyed } from '@angular/core/rxjs-interop';
@Component({
 standalone: true,
 selector: 'app-take-until-destroyed-example',
 template: `
  <p>take-until-destroyed-example works!</p>
 `,
})
export class TakeUntilDestroyedComponent {
 constructor() {
  interval(1000)
   .pipe(takeUntilDestroyed())
   .subscribe((value) => {
    console.log(value);
   });
 }
}
```

Але треба пам'ятати кілька важливих моментів, коли ви використовуєте takeUntilDestroyed:

- як і з іншими способами відписки за допомогою операторів, цей оператор повинен бути останнім у ланцюгу;

- ви не зможете використати цей оператор, наприклад, у хуку ngOnInit. Тобто він буде працювати в контексті конструктора або ініціалізації поля класу, але ви отримаєте помилку у рантаймі: takeUntilDestroyed() can only be used within an injection context, якщо ви спробуєте використати його ще десь;
- якщо ж вам все ж-таки треба це зробити, ви повинні передати destroyRef як аргумент в оператор.

```
@Component({
  standalone: true,
  selector: 'app-take-until-destroyed-example',
  template: `
    <p>take-until-destroyed-example works!</p>
  `,
})
export class TakeUntilDestroyedOnInitComponent implements OnInit {
  constructor(private destroyRef: DestroyRef) {}
  ngOnInit() {
    interval(1000)
      .pipe(takeUntilDestroyed(this.destroyRef))
      .subscribe((value) => {
        console.log(value);
      });
  }
}
```

У підсумку скажу: не зволікайте з оновленням до Angular 16. Серед інших цікавих можливостей, він дозволить рефакторити ваш спосіб обробки відписок і зробить ваш код більш чистим і лаконічним.

Коли відписуватись не потрібно

1) Async pipe

Async pipe виконує роботу по відписці самостійно:

```
@Component({
selector:’test’,
```

```
template: `<todos [todos]=”todos$|async”></todos>`
})
export class TestComponent{
constructor(private store: Store){}
ngOnInit(){
this.todos$=this.store.select(‘todos’);
    }
}
```

2) HostListener

Також нам не треба відписуватись, коли ми створюємо прослуховувала подій при допомозі HostListener:

```
export class TestDirective {
@HostListener(‘click’)
onClick(){
…
    }
}
```

3) Кінцеві послідовності

Бувають послідовності, які самі завершуються, такі як HTTP та timer:

```
export class TestComponent {
    constructor (private http: Http) {}
    ngOnInit(){
        Observable.timer(1000).subscribe(console.log);
        this.http.get(‘http://api.com’).subscribe(console.log);
    }
}
```

Висновок

Підписка, що повисла або відкрита, може призвести до витоків пам'яті, помилок, небажаної поведінки або зниження продуктивності додатків. Щоб уникнути цього, можна використовувати різні способи відписки, але бажаним способом є відписка при допомозі оператор takeUntilDestroyed. Який спосіб використовувати у конкретній ситуації – вирішувати розробнику.