

Reaktywne Programowanie

1. Wzorzec Obserwator

Abstraction

IObservable - nadajnik (strumień danych)

```
//  Nadajnik: emituje dane
public interface IObservable<T>
{
    IDisposable Subscribe(IObserver<T> observer);
}
```

IObserver - odbiorca (obserwator)

```
//  Obserwator: reaguje na dane
public interface IObserver<T>
{
    void OnNext(T value);           //  nowa wartość
    void OnError(Exception error);  //  błąd
    void OnCompleted();            //  koniec
}
```

Implementacja

IObservable - nadajnik (strumień danych)

```
public class MyObservable : IObservable<int>
{
    private IObserver<int>? _observer;

    public IDisposable Subscribe(IObserver<int> observer)
    {
        _observer = observer;
        return new Unsubscriber(() => _observer = null);
    }
}
```

```

    public void Emit(int value)
    {
        _observer?.OnNext(value);
    }

    public void Complete()
    {
        _observer?.OnCompleted();
    }

    private class Unsubscriber : IDisposable
    {
        private readonly Action _disposeAction;
        public Unsubscriber(Action disposeAction) => _disposeAction =
disposeAction;
        public void Dispose() => _disposeAction();
    }
}

```

IObserver - odbiorca (obserwator)

```

class ConsoleObserver : IObserver<int>
{
    public void OnNext(int value) => Console.WriteLine($"📦 {value}");
    public void OnError(Exception error) => Console.WriteLine($"❌ Błąd:
{error.Message}");
    public void OnCompleted() => Console.WriteLine("✅ Completed");
}

```

Przykład użycia

```

void Main()
{
    var stream = new SimpleStream();
    var observer = new ConsoleObserver();

    stream.Subscribe(observer);

    stream.Push(1);
    stream.Fail("Błędny odczyt");

    stream.Push(2);
}

```

```
stream.Complete();  
}
```

📖 Wersja „komiksowa” do zapamiętania

- 🦋 Obserwator ma trzy supermoce:
 - 📦 **OnNext** – „Zobaczyłem nową wartość!”
 - 💣 **OnError** – „O nie! Coś się popsło!”
 - ✅ **OnCompleted** – „Koniec transmisji.”

🔄 Porównanie IEnumerable vs IObservable

Cecha	<code>IEnumerable<T></code>	<code>IObservable<T></code>
Model	Pull	Push
Źródło	Kolekcja	Zdarzenia / dane w czasie
Konsumowanie danych	foreach	Subscribe
Flow kontroluje	Odbiorca	Źródło
Przykład	Lista liczb	Sensor temperatury

🧠 Pytanie:

„Co jeśli chciałbyś filtrować dane, transformować je, łączyć strumienie, opóźniać wysyłkę...
Z każdym razem pisać to ręcznie?”

💡 *To właśnie rozwiązuje Rx!*

⚡ 2. Reactive Extensions (Rx)

Reactive Extensions (Rx) to biblioteka, która:

- Rozszerza `IObservable<T>` o dziesiątki operatorów
- Pozwala pracować ze strumieniami danych jak z kolekcjami (`LINQ dla czasu`)
- Obsługuje czas, asynchroniczność, błędy i wiele źródeł

🔔 Events + 🔄 Linq = < Rx

🧠 Komentarz:

"Wyobraź sobie, że możesz filtrować zdarzenia tak samo, jak dane w kolekcji."

Na przykład:

- chcesz reagować tylko na ruchy myszki w prawo?
- albo interesują cię tylko kliknięcia co 5 sekund?

Zamiast analizować wszystko ręcznie — po prostu piszesz:

```
mouseMoves
    .Where(m => m.X > previousX)
    .Throttle(TimeSpan.FromSeconds(0.5))
    .Subscribe(m => Console.WriteLine($"👉 Ruch w prawo: {m.X}"));
```

💡 RX jest jak LINQ dla eventów, tylko z obsługą czasu, błędów i asynchroniczności 🚀.

🔄 Porównanie Kod imperatywny vs Kod reaktywny

📦 Kod imperatywny:

```
foreach (var x in data) { ... }
```

⚡ Kod reaktywny:

```
observable.Subscribe(x => { ... })
```

🔄 My pytamy vs 🔔 My reagujemy

Subject

```
public class SimpleObservable : IObservable<string>, IObservable<string>
{
    private readonly List<IObserver<string>> _observers = new();

    // IObservable<T>
    public IDisposable Subscribe(IObserver<string> observer)
    {
        _observers.Add(observer);
        return new Unsubscriber(_observers, observer);
    }

    // IObserver<T>
    public void OnNext(string value)
```

```

{
    foreach (var o in _observers)
        o.OnNext(value);
}

public void OnError(Exception error)
{
    foreach (var o in _observers)
        o.OnError(error);
}

public void OnCompleted()
{
    foreach (var o in _observers)
        o.OnCompleted();
}

// pomocnicza klasa do usuwania subskrybenta
private class Unsubscriber : IDisposable
{
    private readonly List<IObserver<string>> _observers;
    private readonly IObserver<string> _observer;

    public Unsubscriber(List<IObserver<string>> observers,
        IObserver<string> observer)
    {
        _observers = observers;
        _observer = observer;
    }

    public void Dispose() => _observers.Remove(_observer);
}
}

```

Subject<T>

Subject<T> to podstawowy typ spośród dostępnych subjectów i zapewnia łatwy sposób pracy z biblioteką, bez konieczności samodzielnego implementowania interfejsów IObservable<T> i IObservable<T>.

```

var subject = new Subject<int>();
subject.Subscribe(Console.Write);
subject.OnNext(1);
subject.OnNext(2);

```

```
subject.OnNext(3);
subject.OnNext(4);
```

Source: —1—2—3—4—|
Observer: —1—2—3—4—|

```
var subject = new Subject<int>();
subject.OnNext(1);
subject.OnNext(2);
subject.Subscribe(Console.Write);
subject.OnNext(3);
subject.OnNext(4);
```

Objaśnienie:

- `subject.OnNext(1)` i `subject.OnNext(2)` są wyemitowane przed subskrypcją, więc nie są odebrane przez subskrybenta.
- Subskrypcja (`Subscribe(...)`) następuje tuż przed `OnNext(3)` .
- `Subject<T>` jest gorącym źródłem – nie buforuje ani nie ponawia danych sprzed momentu subskrypcji.



ReplaySubject<T>

```
var subject = new ReplaySubject<int>();
subject.OnNext(1);
subject.OnNext(2);
subject.Subscribe(Console.Write);
subject.OnNext(3);
subject.OnNext(4);
```

Source: —1—2—3—4—|
 |
Observer: 1—2—3—4—|

Objaśnienie:

- `ReplaySubject<T>` zapamiętuje wszystkie wcześniejsze wartości (domyślnie, jeśli nie podano bufora limitującego).
- Gdy subskrybent dołącza (przy `Subscribe(...)`), otrzymuje natychmiast wszystkie wartości wysłane wcześniej (1 i 2), a potem również bieżące (3 , 4).

- Wartości są przekazywane w kolejności: 1, 2, 3, 4.



BACK

ReplaySubject<T>(bufferSize:N)

```
var subject = new ReplaySubject<int>(bufferSize: 1);
subject.OnNext(1);
subject.OnNext(2);
subject.Subscribe(Console.WriteLine);
subject.OnNext(3);
subject.OnNext(4);
```

```
Source:      —1—2—3—4—|
              |
Observer:    2—3—4—|
```

Objaśnienie:

- Subskrybent otrzymuje tylko ostatnią wartość przed subskrypcją (2), a potem bieżące (3, 4).

ReplaySubject<T>(TimeSpan)

```
var subject = new ReplaySubject<int>(TimeSpan.FromMilliseconds(1000));
subject.OnNext(1);
Thread.Sleep(500);
subject.OnNext(2);
Thread.Sleep(200);
subject.OnNext(3);
Thread.Sleep(500);
subject.Subscribe(Console.WriteLine);
subject.OnNext(4);
Thread.Sleep(500);
```

BehaviorSubject<T>

```
var subject = new BehaviorSubject<int>(0);
subject.OnNext(1);
subject.OnNext(2);
subject.Subscribe(Console.WriteLine);
subject.OnNext(3);
subject.OnNext(4);
```

```
Source:  —1—2—3—4—|
           |
Observer:      2—3—4—|
```

Objasnienie:

- Przechowuje ostatnią wyemitowaną wartość (albo wartość początkową).
- Gdy subskrybent się podłącza, otrzymuje od razu ostatnią znaną wartość, nawet jeśli została wyemitowana przed jego subskrypcją.

Opis działania

◆ Subject<T>

- Nie przechowuje historii.
- Subskrybent otrzymuje **tylko wartości od momentu subskrypcji**.

◆ ReplaySubject<T>

- **Przechowuje wszystkie wartości** (domyślnie).
- Subskrybent otrzymuje **pełną historię**.
- Można jednak ograniczyć historię:
 - liczbą elementów: `new ReplaySubject<int>(bufferSize: 2)`
 - lub czasem trwania: `new ReplaySubject<int>(window: TimeSpan.FromSeconds(30))`

◆ BehaviorSubject<T>

- **Przechowuje ostatnią wartość**.
- Subskrybent dostaje **ostatnią wartość od razu**, a potem bieżące.

◆ AsyncSubject<T>

- Przechowuje **tylko ostatnią wartość**, ale...
- **...emituje ją tylko po** `OnCompleted()`.
- Subskrybent widzi tylko ostatnią wartość **i tylko jeśli strumień się zakończy**.

Przykład użycia `Subject<T>` :

```
public class EventHub {
    private readonly Subject<string> _eventStream = new Subject<string>();

    public IObservable<string> Events => _eventStream.AsObservable();

    public void Publish(string evt) => _eventStream.OnNext(evt);
}
```

➡ Zastosowanie: system powiadomień lub komunikacja komponentów, gdzie tylko aktywni subskrybenci mają znaczenie.

Przykład użycia `BehaviorSubject` :

```
public class ConnectionStateService {
    private BehaviorSubject<bool> _isConnected = new BehaviorSubject<bool>
(false);

    public IObservable<bool> ConnectionState => _isConnected.AsObservable();

    public void SetConnected(bool value) => _isConnected.OnNext(value);
}
```

➡ Subskrybenci zawsze dostaną aktualny stan (true lub false), nawet jeśli dołączą później.

Przykład użycia `ReplaySubject<T>`

```
public class LogBuffer {
    private readonly ReplaySubject<string> _logStream = new
ReplaySubject<string>(bufferSize: 100);

    public IObservable<string> Logs => _logStream.AsObservable();

    public void Log(string entry) => _logStream.OnNext(entry);
}
```

➡ Zastosowanie: system logowania, w którym nowe komponenty mogą „dogonić” ostatnie 100 wpisów

📌 Przykład użycia AsyncSubject<T>

```
public class BatteryLevelSensor
{
    private readonly AsyncSubject<int> _batteryLevel = new AsyncSubject<int>
    ();

    public void SetBatteryLevel(int value)
    {
        _batteryLevel.OnNext(value);
        _batteryLevel.OnCompleted(); // bez tego nic się nie wyemituje!
    }

    public IObservable<int> ReadBatteryLevel() => _batteryLevel;
}
```

➡ Zastosowanie: jednorazowy odczyt poziomu baterii z urządzenia (wartość trafia do wszystkich dopiero po zakończeniu pomiaru)

🌀 Typy źródeł

❄️ Zimne

Zimne źródła emitują dane **dopiero po subskrypcji** — każdy subskrybent dostaje "pełną historię" od początku.

- Emitują dane na żądanie
- Każdy subskrybent otrzymuje wszystko od początku
- Przykłady: `Observable.Range`, `Observable.Generate`

👤 *Przykład dydaktyczny:* Student spóźnił się na zajęcia, więc nie usłyszał pierwszych informacji od wykładowcy, ale wziął notatki od kolegi — dzięki temu nic nie stracił. Kolega jest źródłem zimnym. Tak samo działa zimne źródło — każdy subskrybent dostaje dane od początku.

🔥 Gorące

Gorące źródła **emitują dane niezależnie od subskrypcji** — jeśli nie jesteś podłączony w momencie emisji, tracisz dane.

- Emitują dane w czasie rzeczywistym
- Subskrybenci dostają tylko aktualne/nowe dane
- Przykłady: `Subject`, `Observable.Interval`, `Observable.Timer`

👤 **Przykład dydaktyczny:** Student spóźnił się na zajęcia, więc nie usłyszał pierwszych informacji od wykładowcy. Wykładowca jest źródłem gorącym. Tak samo działa gorące źródło — jeśli subskrybujesz za późno, tracisz wcześniejsze dane.

∞ Notacja marble diagram (diagram perełkowy)

Marble diagram (czyli *diagram perełkowy*) to sposób graficznego przedstawienia **strumienia danych w czasie**, używany w programowaniu reaktywnym.

🕒	—	→ upływ czasu
📦	1, 2, 3...	→ dane (<code>OnNext</code>)
✅		→ zakończenie (<code>OnCompleted</code>)
💣	#	→ błąd (<code>OnError</code>)
📦	[]	→ bufor / lista
🔗	()	→ para (np. <code>CombineLatest</code> , <code>Zip</code>)
∞	...	→ nieskończony strumień

✅ Źródło z zakończeniem:

Source: —1—2—3—4—5—6—7—8—9—| (`OnCompleted`)

🔄 Źródło nieskończone

Source: —1—2—3—4—5—6—7—8—9—...

💣 Źródło z błędem:

Source: —1—2—# (`OnError` przerwał strumień)

Źródło danych

Synchroniczne (natychmiastowe)

```
var observable = Observable.Range(1, 10); // dane: 1..10
```

Zachowuje się jak pętla: `// for (int i = 1; i <= 10; i++) yield return i;`

Source: —1—2—3—4—5—6—7—8—9—10—|

Aktywne (Subject<T>)

```
var subject = new Subject<int>();

subject.Subscribe(x => Console.WriteLine($"[Subject]: {x}"));

subject.OnNext(1); // ręczna emisja danych
subject.OnNext(2);
subject.OnCompleted();
```

Czasowe (interwały)


```
Observable.Interval(TimeSpan.FromSeconds(1))
```

Source: —1—2—3—4—5—6—7—8—9—10—...

Podstawowe operatory

Select - transformacja wartości (mapowanie)

Przykład 1: `Select(x => x * 2)`

```
observable
    .Select(x => x * 2)
    .Subscribe(x => Console.WriteLine($" {x}"));
```

🧠 Analogia do funkcji $x \Rightarrow f(x)$

```
Input:      —1—2—3—4—5—|  
Select(x*2): —2—4—6—8—10—|
```

🔍 Where – filtrowanie danych

◆ Przykład 1: `Where(x => x % 2 == 0)`

```
observable  
  .Where(x => x % 2 == 0)  
  .Subscribe(x => Console.WriteLine($"📦 {x}"));
```

```
Input:      —1—2—3—4—5—|  
Where(%2==0): — 2 — 4 — |
```

🧠 **predykat** $f: T \rightarrow \text{bool}$ — mówi: "czy wartość spełnia warunek?"

🪣 Buffer – grupowanie po n elementów

◆ Przykład 1: `Buffer(3,3)`

```
observable  
  .Buffer(3, 3)  
  .Subscribe(buffer => Console.WriteLine($"[Fixed] {string.Join(", ",  
buffer)}"));
```

```
Buffers(3):  —[1 2 3]—[4 5 6]—[7 8 9]—[10]—|
```

◆ Przykład 2: `Buffer(3,1)`

```
observable  
  .Buffer(3, 1)
```

```
.Subscribe(buffer =>
    Console.WriteLine($"[Sliding] {string.Join(", ", buffer)}"));
```

```
Buffers(3,1):  —[1 2 3]
               [2 3 4]
               [3 4 5]
               [4 5 6]
               [5 6 7]
               [6 7 8]
               [7 8 9]
               [8 9 10]
               [9 10]
               [10]
```

🧠 Wniosek dydaktyczny

- **Buffer(3,3)** = okna **niezależne** → jak *wiadra*
- **Buffer(3,1)** = okna **nachodzące** → jak *okno przesuwane po strumieniu*

💡 Wskazówka

Buffer(3,3) = Buffer(3) → są równoważne — oba tworzą **niezachodzące** okna po 3 elementy.

🕒 Throttle/Debounce – wygaszanie "szumu"

```
observable
    .Throttle(TimeSpan.FromSeconds(1))
```

pomija zbyt częste zdarzenia — reaguje **dopiero wtedy**, gdy przez pewien czas nic nowego się nie pojawiło.

```
Input:  —1-2-3———4——5—|
Output:           3       5
```

🔗 CombineLatest – łącz ostatnie dane z różnych źródeł

```
Observable.CombineLatest(temp, humidity, (t, h) => $"{t}°C, {h}%")
```

```
temp:      —20——21——22——  
humidity: -70———75———  
Output:    —"20,70" "21,70" "21,75" "22,75"
```

👉 Każda wartość czeka na parę, to jak polonez — zanim ruszysz, musisz mieć parę.

+ Merge – połącz wiele strumieni w jeden

```
Observable.Merge(sensorA, sensorB)
```

```
A:    -1——3———5—  
B:      2——4——6  
Result:-1-2-3-4——5-6
```

👉 Każda wartość przepływa osobno, **bez parowania**.

🧠 Merge = **suma (u) strumieni zdarzeń**

🧊 DistinctUntilChanged – ignoruj powtórzenia

```
observable.DistinctUntilChanged()
```

```
Input:   -1—1—2—2—3—3—1  
Output:  -1—2—3—1
```

🧠 Analogia do funkcji $x, y \Rightarrow f(x, y)$