Traditional vs Reactive Programming

1. Overview

Aspect	Traditional Programming	Reactive Programming
Paradigm	Imperative / Procedural / Object-Oriented	Declarative / Functional / Event-driven
Flow of Data	Pull-based (request-response)	Push-based (event streams, observables)
Control Flow	Controlled by the programmer	Controlled by data/events
Concurrency	Typically synchronous or multithreaded	Asynchronous, non-blocking

2. Traditional Programming

Definition:
A programming model where you explicitly define the flow of the program using statements and control structures (e.g., if, for, while).

Characteristics:

- Executes instructions step-by-step.
- Data is fetched when needed (pull).
- Good for static or simple data interactions.
- Uses callbacks for asynchronous operations (can lead to callback hell).

Example (JavaScript):

```
const data = getData(); // Synchronous
console.log(data);
```

3. Reactive Programming

Definition:
A paradigm for writing code that reacts to changes over time, especially useful for asynchronous data streams like UI events, HTTP requests, and real-time updates.

Characteristics:

- Data is pushed to consumers when available.
- Emphasizes non-blocking, event-driven architecture.
- Uses **observables**, **streams**, or **signals**.
- Better handles dynamic or high-volume data scenarios.

Example (RxJS in Angular):

```
getData().subscribe(data => {
  console.log(data);
});
```

4. Key Concepts in Reactive Programming

- **Observable**: Represents a stream of data or events.
- **Observer/Subscriber**: Reacts to emitted data.
- **Operators**: Functions to transform, filter, or combine streams (e.g., `map`, `filter`, `merge`).
- **Backpressure**: Mechanism to control data flow rate in high-throughput systems.

5. When to Use Each

Use Case	Traditional	Reactive
Simple, synchronous logic	YES	NO
Handling user input in forms	YES	YES

Real-time updates (e.g., WebSocket)	NO	YES
Complex async data streams	NO	YES
Low resource / embedded systems	YES	NO

6. Benefits of Reactive Programming

- Better scalability for asynchronous apps.
- Cleaner and more maintainable async code.
- Improved performance for I/O-heavy applications.
- Naturally fits with event-driven architecture (e.g., UI, IoT, messaging apps).

7. Popular Libraries / Frameworks

- **JavaScript**: RxJS (Angular), MobX, Signals
- **Java**: Project Reactor, RxJava
- **Kotlin**: Flow, Coroutine Flow
- **C#**: Reactive Extensions (Rx.NET)

Push vs Pull Data Streams

1. Overview

Concept	Pull Stream	Push Stream
Who controls the data flow?	Consumer (pulls data when needed)	Producer (pushes data as it arrives)
Direction	Demand-driven	Data-driven
Timing	Consumer decides when to get data	Producer decides when to send data
Common Use Cases	Traditional I/O, APIs, iterators	Events, Observables, WebSockets

2. Pull-Based Streams

Definition:
In pull streams, the **consumer** initiates data retrieval. The flow only continues when the consumer explicitly asks for the next piece of data.

Key Traits:

- Synchronous or blocking in nature.
- Simple and predictable flow.
- Suitable for finite, sequential datasets.

Common Interfaces:

- **Iterator** (JavaScript, Java)
- **Streams API** (Node.js readable streams in paused mode)
- **REST API calls**

Example (JavaScript Iterator):

```
const iterator = [1, 2, 3][Symbol.iterator]();
console.log(iterator.next()); // { value: 1, done: false
}
```

3. Push-Based Streams

Definition:
In push streams, the **producer** controls the flow and sends data to the consumer as it becomes available.

Key Traits:

- Asynchronous, event-driven.
- Good for handling real-time or unbounded data.
- Often uses callbacks, listeners, or observers.

Common Interfaces:

- **EventEmitter**
- **Observable** (RxJS, RxJava)
- **WebSockets / SSE (Server-Sent Events)**
- **Streams in flowing mode (Node.js)**

Example (RxJS Observable):

```
const observable = of(1, 2, 3);
observable.subscribe(value => console.log(value));
```

4. Comparison Table

Feature	Pull Stream	Push Stream

Control	Consumer	Producer
Sync/Async	Usually synchronous	Usually asynchronous
Error Handling	Try/catch block	Subscription error callbacks
Examples	Array iteration, REST	Event listeners, WebSockets
Backpressure Handling	Requires manual implementation	Built-in in some reactive libs

5. Real-World Analogies

- **Pull**: Reading a book page-by-page at your own pace.
- **Push**: Subscribing to a newsletter – new content is sent to you whenever it's available.

---

6. When to Use

Scenario	Use Pull	Use Push
Data is available on demand	Yes	No
Continuous, unpredictable data flow	No	Yes
Real-time app (chat, stock ticker)	No	Yes
Static report from a database	Yes	No

What is RxJS?

1. Definition

RxJS (Reactive Extensions for JavaScript) is a **library for reactive programming** using **Observables** to handle **asynchronous data streams**.
It helps you work with events, AJAX, timers, and WebSockets in a **declarative**, **composable**, and **concise** way.

2. Core Concept: Observable

- An **Observable** is like a stream of data that you can **subscribe** to.
- It emits values **over time**, not just once.
- Think of it as an advanced version of Promises that supports **multiple values**, **cancellation**, and **operators**.

Example:

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

const stream$ = of(1, 2, 3);
stream$.subscribe(value => console.log(value)); // Logs: 1, 2, 3
```

Why Use RxJS?

1. Handle Async Data Easily

RxJS provides a unified way to work with:

- Timers
- Events (e.g. button clicks)
- HTTP responses
- WebSockets
- User input streams

2. Powerful Operators

Over **100+ operators** to:

- **Transform** (e.g. `map`, `filter`)
- **Combine** (e.g. `merge`, `combineLatest`)
- **Retry** or **Debounce**
- **Switch** between streams (`switchMap`, `concatMap`, `exhaustMap`)

3. Better Composition

You can **chain multiple async operations** in a clean and readable way.

Example:

```
import { fromEvent } from 'rxjs';
import { map, filter } from 'rxjs/operators';

fromEvent(document, 'click')
  .pipe(
    map(event => event.clientX),
    filter(x => x > 100)
  )
  .subscribe(x => console.log(x));
```

4. Cancel and Clean Up

Easily **unsubscribe** or cancel operations (something Promises can’t do).

```
const sub = observable.subscribe(...);
sub.unsubscribe(); // Stops listening to stream
```

5. Declarative and Readable

- Instead of saying *how* to do things (imperative), you describe *what* you want (declarative).

- Reduces **callback hell** and complex async chains.

6. Framework Integration

- **Angular** has built-in RxJS support (e.g., `HttpClient` returns `Observable`).

- Works well with **React**, **Vue**, **Node.js**, and vanilla JavaScript.

Summary

Feature	Benefit
Unified async handling	Events, AJAX, WebSockets, timers
Operators	Chain, transform, combine streams
Observable model	Push-based data, cancelable, reusable
Declarative syntax	Clean, readable code
Integration	Especially powerful in Angular

Observable vs Observer vs Subscription

These are **three fundamental building blocks** in RxJS used to handle **asynchronous streams**.

1. Observable

Definition:

An **Observable** is a data producer that **emits a sequence of values over time**.
It does **not** start emitting until it is **subscribed** to.

Key Points:

- Represents a stream (could be finite or infinite).
- Can emit multiple values (unlike Promises).
- Can emit `next`, `error`, or `complete` notifications.
- Created using functions like `of()`, `from()`, `interval()`, etc.

Example:

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

const numbers$ = of(1, 2, 3); // Observable
```

2. Observer

📘 **Definition:**

An **Observer** is a **consumer** of data emitted by an Observable.
It is an object that defines **handlers** for the three types of notifications:

Method	Purpose
`next()`	Handles each emitted value

`error()`	Handles an error if one occurs
`complete()`	Handles when Observable completes

📦 **Example:**

```
const observer = {
  next: val => console.log('Received:', val),
  error: err => console.error('Error:', err),
  complete: () => console.log('Done!')
};
```

3. Subscription

Definition:

A **Subscription** represents the **execution** of an Observable and its link to an Observer.

- Created by calling `.subscribe(observer)` on an Observable.
- Can be used to **unsubscribe** and stop receiving further values.
- Manages resource cleanup.

Example:

```
const subscription = numbers$.subscribe(observer);

// Later, to stop:
subscription.unsubscribe();
```

🧩 How They Work Together

```
// 1. Create Observable
```

```
const obs$ = of(1, 2, 3);

// 2. Define Observer
const observer = {
  next: x => console.log('Next:', x),
  error: err => console.error('Error:', err),
  complete: () => console.log('Complete')
};

// 3. Subscribe
const sub = obs$.subscribe(observer);
```

Flow: Observable → emits data → Observer receives → Subscription manages

Summary Table

Concept	Role	Key Methods / Usage
Observable	Emits data	`of()`, `from()`, `interval()`
Observer	Handles data	`next()`, `error()`, `complete()`
Subscription	Connects observer to observable	`.subscribe()`, `.unsubscribe()`

Create a Basic Observable (RxJS)

```
// Import from RxJS
import { Observable } from 'rxjs';

// Step 1: Create an Observable
const myObservable = new Observable(subscriber => {
  subscriber.next('Hello');
  subscriber.next('from');
  subscriber.next('RxJS!');
  subscriber.complete(); // Signals completion
});

// Step 2: Create an Observer
const myObserver = {
  next: value => console.log('Received:', value),
  error: err => console.error('Error:', err),
  complete: () => console.log('Stream complete.')
};

// Step 3: Subscribe
myObservable.subscribe(myObserver);
```

Output

```
Received: Hello
Received: from
Received: RxJS!
Stream complete
```

Explanation

- **Observable**: Emits a sequence of values.
- **Observer**: Defines how to handle each value, error, and completion.
- **Subscription**: Triggers the observable and links it to the observer.

What Are Operators in RxJS?

Definition:

Operators are **functions** in RxJS that **transform, filter, combine, or manage** data emitted by Observables.
They allow you to build **powerful data pipelines** in a **declarative and composable** way.

> Think of operators as the **middleware** between the source (Observable) and the output (Observer).

Types of Operators

Category	Purpose	Examples
Creation	Create new Observables	`of`, `from`, `interval`, `timer`
Transformation	Change emitted values	`map`, `pluck`, `scan`
Filtering	Allow only certain values	`filter`, `take`, `first`, `skip`
Combination	Merge or join multiple Observables	`merge`, `combineLatest`, `concat`
Utility	Debug, delay, or finalize streams	`tap`, `delay`, `finalize`
Error Handling	Handle or recover from errors	`catchError`, `retry`, `retryWhen`
Multicasting	Share a single Observable among many subscribers	`share`, `shareReplay`

Example: Transforming Data with `map`

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

of(1, 2, 3)
  .pipe(
    map(x => x * 10)
  )
  .subscribe(result => console.log(result));
```

Output:

```
10
20
30
```

The `pipe()` Function

- Used to **chain** multiple operators.
- Keeps code **clean** and **readable**.

```
observable.pipe(
  operator1(),
  operator2(),
  ...
);
```

Why Use Operators?

- They simplify complex async tasks.
- Promote **functional programming** style.
- Avoids deeply nested callbacks.
- Makes stream handling **declarative**, **flexible**, and **clean**.

Popular Operators to Know

Operator	Description
`map`	Transforms values

`filter`	Emits values that match a condition
`mergeMap`	Flattens inner Observables (async)
`switchMap`	Switches to a new inner Observable
`take`	Takes the first `n` values
`catchError`	Handles errors

RxJS Operator Categories

RxJS operators are grouped into several categories based on their function. Let's explore the four most essential ones: **Creation, Transformation, Filtering, and Combination**.

1. Creation Operators

Purpose:
Create new Observables from various sources such as values, arrays, events, or timers.

Operator	Description	Example
`of`	Emits the provided values in sequence	`of(1, 2, 3)`
`from`	Converts arrays, promises, or iterables	`from([10, 20, 30])`
`interval`	Emits numbers at a fixed time interval	`interval(1000)`
`timer`	Emits after a delay or periodically	`timer(2000, 1000)`
`fromEvent`	Converts DOM events into observable streams	`fromEvent(button, 'click')`

2. Transformation Operators

Purpose:
Modify the data emitted by an Observable before passing it to the Observer.

Operator	Description	Example
`map`	Applies a function to each value	`map(x => x * 2)`
`pluck`	Extracts a property from emitted objects	`pluck('name')`
`scan`	Accumulates values over time (like reduce)	`scan((acc, x) => acc + x, 0)`
`concatMap`	Maps and flattens in order (inner observable)	`concatMap(val => someObs(val))`

3. Filtering Operators

Purpose:
Filter or limit the values emitted based on specific conditions.

Operator	Description	Example
`filter`	Only passes values that meet a condition	`filter(x => x > 10)`
`take`	Emits only the first N values	`take(3)`
`first`	Emits only the first value	`first()`
`skip`	Skips the first N values	`skip(2)`
`debounceTime`	Emits after specified silence time	`debounceTime(300)`

---

4. Combination Operators

Purpose:
Combine multiple Observables into one stream.

Operator	Description	Example
`merge`	Combines and emits as they come	`merge(obs1, obs2)`
`concat`	Emits from one observable after the other	`concat(obs1, obs2)`
`combineLatest`	Emits latest values from multiple observables	`combineLatest([obs1, obs2])`
`withLatestFrom`	Combines latest from another observable on event	`source.withLatestFrom(other)`
`zip`	Combines values by index	`zip(obs1, obs2)`

Summary Table

Category	Use Case	Example Operator

Creation	Generate streams from scratch	`of`, `from`
Transformation	Modify values in a stream	`map`, `scan`
Filtering	Control what values get through	`filter`, `take`
Combination	Merge or coordinate observables	`merge`, `zip`

Key RxJS Operators

1. `map`

What It Does:

Transforms each emitted value by applying a function.

Use Case:

Double each number in a stream.

Example:

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';

of(1, 2, 3)
  .pipe(map(x => x * 2))
  .subscribe(console.log);
// Output: 2, 4, 6
```

2. `filter`

What It Does:

Only emits values that satisfy a given condition.

Use Case:

Allow only even numbers through.

Example:

```
import { of } from 'rxjs';
import { filter } from 'rxjs/operators';

of(1, 2, 3, 4)
  .pipe(filter(x => x % 2 === 0))
  .subscribe(console.log);
// Output: 2, 4
```

3. take

What It Does:

Emits only the first **N** values, then completes.

Use Case:

Limit the output to the first few values.

Example:

```
import { of } from 'rxjs';
import { take } from 'rxjs/operators';

of(10, 20, 30, 40)
  .pipe(take(2))
  .subscribe(console.log);
// Output: 10, 20
```

4. debounceTime

What It Does:

Emits a value from the source Observable **only after a specified time has passed** without another source emission.

Use Case:

Useful in search fields to avoid sending too many requests while the user types.

Example:

```
import { fromEvent } from 'rxjs';
import { debounceTime, map } from 'rxjs/operators';

fromEvent(document, 'keyup')
  .pipe(
    debounceTime(300),
    map(event => event.target.value)
  )
  .subscribe(console.log);
```

5. `mergeMap`

What It Does:

Maps each value to an inner Observable and **merges** all inner Observables into a single output stream.

Use Case:

Perform async operations like API calls for each item.

Example:

```
import { from } from 'rxjs';
import { mergeMap } from 'rxjs/operators';

const mockApiCall = val => from([`${val}-result`]);

from(['A', 'B'])
  .pipe(
    mergeMap(val => mockApiCall(val))
  )
  .subscribe(console.log);
// Output: A-result, B-result
```

Summary Table

Operator	Function	Common Use Case
`map`	Transforms each value	Modify or format stream data
`filter`	Allows values that meet a condition	Screen values (e.g., only even numbers)
`take`	Takes the first N values	Limit emissions

`debounceTime`	Waits for silence before emitting	Debounce form inputs
`mergeMap`	Flattens and merges async streams	Run HTTP requests or async tasks

RxJS: Error Handling and Cleanup

Goal

Learn how to gracefully handle errors in RxJS streams and avoid memory leaks by properly managing subscriptions.

1. Error Handling in RxJS

In RxJS, **errors propagate through the observable chain** and are handled in the `error` callback or with error-handling operators.

Key Operators

`catchError()`

- Catches errors in an observable stream and allows you to recover.
- Returns a new observable (e.g., fallback value or empty stream).

```
import { of } from 'rxjs';
import { catchError } from 'rxjs/operators';

source$.pipe(
  catchError(err => {
    console.error('Caught error:', err);
    return of('Fallback Value');
  })
)
```

`retry()`

- Retries a failed observable stream a specified number of times.

```
import { retry } from 'rxjs/operators';

source$.pipe(
  retry(3) // try 3 more times before failing
)
```

`finalize()`

- Executes a callback when the observable completes or errors out.
- Useful for cleanup (e.g., stop loading spinner).

```
import { finalize } from 'rxjs/operators';

source$.pipe(
  finalize(() => console.log('Stream ended or errored'))
)
```

2. Unsubscribing from Observables

Why?

- If you don't unsubscribe, long-lived observables (e.g., `interval`, `fromEvent`) can cause **memory leaks**.

Manual Unsubscription

```
const subscription = source$.subscribe(data => console.log(data));

// Later, cleanup:
subscription.unsubscribe();
```

Use in Angular

```
ngOnDestroy() {
  this.subscription.unsubscribe();
}
```

3. Memory Leaks Warning

When it happens:

- You subscribe but **don't unsubscribe**.

- Components are destroyed, but the subscription continues.

Solutions:

- **Use `takeUntil()` with a `Subject`** for component cleanup.
- **Use `async` pipe** in Angular templates (auto-unsubscribes).
- **Use operators like `take(n)`, `first()`** for auto-complete streams.

Best Practices

- Always unsubscribe from manual subscriptions.
- Use `catchError` to handle errors gracefully.
- Use `finalize` for side-effects like hiding loaders.
- Avoid leaks using `takeUntil`, `async`, or complete signals.

RxJS with Events and AJAX (30 mins)

This session introduces real-world use cases for RxJS with DOM events and HTTP requests using built-in RxJS features like `fromEvent`, `debounceTime`, and `ajax`.

1. Using `fromEvent` for Event Streams

What It Does:

`fromEvent` creates an observable stream from DOM events (like `click`, `keyup`, `scroll`, etc).

Use Case:

Listen to button clicks or input typing without manual event listeners.

Example: Button Click

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

const button = document.getElementById('myButton');

fromEvent(button, 'click').subscribe(() => {
  console.log('Button clicked!');
});
```

2. Debouncing Input Search

What It Does:

Waits until the user stops typing before sending the value — avoids flooding backend with every keystroke.

Operators Used:

- `fromEvent` – listens to `keyup`
- `debounceTime` – waits for typing to stop
- `map` – extracts input value
- `filter` – optional: only search if input has length

Example:

```
import { fromEvent } from 'rxjs';
import { debounceTime, map, filter } from 'rxjs/operators';

const input = document.getElementById('searchBox');

fromEvent(input, 'keyup').pipe(
  debounceTime(300),
  map((e) => e.target.value),
  filter(text => text.length > 2)
).subscribe(searchText => {
  console.log('Search:', searchText);
});
```

3. AJAX Requests with RxJS

What It Does:

Use RxJS's built-in `ajax` operator to perform HTTP requests.

`ajax` Features:

- Returns an Observable of the response
- Supports GET, POST, PUT, DELETE
- Handles success, error, and progress events

Example: Basic GET Request

```
import { ajax } from 'rxjs/ajax';

ajax('https://jsonplaceholder.typicode.com/posts/1')
  .subscribe({
    next: response => console.log('Data:', response.response),
    error: err => console.error('Error:', err),
    complete: () => console.log('Request complete.')
  });
```

Example: Combine with Input

```
import { fromEvent } from 'rxjs';
import { debounceTime, map, switchMap } from 'rxjs/operators';
import { ajax } from 'rxjs/ajax';

const input = document.getElementById('searchBox');

fromEvent(input, 'keyup').pipe(
  debounceTime(500),
  map(e => e.target.value),
  switchMap(searchTerm =>
    ajax.getJSON(`https://api.example.com/search?q=${searchTerm}`)
  )
).subscribe(results => {
  console.log('Results:', results);
});
```

Key Takeaways

Concept	Summary
`fromEvent()`	Converts DOM events to observable streams
`debounceTime()`	Prevents over-triggering (e.g., on user typing)
`ajax()`	Makes HTTP calls that return observable streams
`switchMap()`	Cancels previous requests for fast changing input

RxJS Integration with Angular

Angular is built with RxJS at its core. This section focuses on how Angular uses **Observables**, especially in **HttpClient**, and how RxJS integrates seamlessly in services and components.

1. Angular `HttpClient` Returns Observables

What It Does:

Angular’s `HttpClient` automatically returns **Observables** for HTTP operations like `GET`, `POST`, `PUT`, `DELETE`.

Why It Matters:

- Non-blocking HTTP requests.
- Composable with RxJS operators.
- Easy to cancel or retry requests.
- Supports async handling via `async` pipe.

Example:

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

constructor(private http: HttpClient) {}

getPosts() {
  return this.http.get('https://jsonplaceholder.typicode.com/posts');
}
```

2. Using RxJS in Angular Services and Components

Best Practice:

Use **services** to make HTTP calls, and **components** to subscribe or bind via `async`.

Service Example:

```
@Injectable({ providedIn: 'root' })
```

```
export class PostService {
  constructor(private http: HttpClient) {}

  getPosts() {
    return this.http.get('https://jsonplaceholder.typicode.com/posts');
  }
}
```

Component Example (manual subscribe):

```
export class PostListComponent implements OnInit {
  posts: any[] = [];

  constructor(private postService: PostService) {}

  ngOnInit() {
    this.postService.getPosts().subscribe(data => {
      this.posts = data;
    });
  }
}
```

3. The async Pipe (Template Binding)

What It Does:

Automatically subscribes to and unsubscribes from Observables in templates.
Great for one-way data binding.

Benefits:

- No need to manually `subscribe()` or `unsubscribe()`
- Clean and concise templates
- Handles memory management

Component Example (returns observable directly):

```
posts$ = this.postService.getPosts();
```

Template Example:

```
<ul *ngIf="posts$ | async as posts">
  <li *ngFor="let post of posts">{{ post.title }}</li>
</ul>
```

Summary Table

Feature	Role in Angular
`HttpClient`	Returns Observables for HTTP
RxJS in Services	Centralize async logic
RxJS in Components	Consume observables or use `async` pipe
`async` Pipe	Template-friendly, handles subscriptions

Explore Subjects in RxJS

What is a Subject?

A **Subject** in RxJS is **both an Observable and an Observer**:

- It can **emit values** to subscribers (like an Observable).
- It can **receive values** using `.next()` (like an Observer).

Subjects are **multicast**: all subscribers share the same execution and receive the same values.

1. `Subject`

Basic Subject

- Does **not** hold a current value.
- Subscribers only receive **future** values (emitted **after** subscription).
- Does **not replay** or remember old values.

Example:

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

const subject = new Subject();

subject.subscribe(value => console.log('Sub1:', value));
subject.next('Hello');  // Sub1 gets this

subject.subscribe(value => console.log('Sub2:', value));
subject.next('World');  // Both Sub1 and Sub2 get this
```

Output:

```
Sub1: Hello
Sub1: World
Sub2: World
```

2. `BehaviorSubject`

Key Features:

- **Holds the latest value** (initial value is required).
- New subscribers **immediately receive the latest value**.
- Useful for storing state (e.g., current user, auth status).

Example:

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

const behaviorSubject = new BehaviorSubject('Initial');

behaviorSubject.subscribe(val => console.log('Sub1:', val));
behaviorSubject.next('Update 1');

behaviorSubject.subscribe(val => console.log('Sub2:', val));
behaviorSubject.next('Update 2');
```

Output:

```
Sub1: Initial
Sub1: Update 1
Sub2: Update 1
Sub1: Update 2
Sub2: Update 2
```

3. `ReplaySubject`

Key Features:

- Replays a **specified number of past values** to new subscribers.

- Useful for **caching** or **logging** scenarios.

Example:

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

const replaySubject = new ReplaySubject(2); // Buffer last 2 values

replaySubject.next('A');
replaySubject.next('B');
replaySubject.next('C');

replaySubject.subscribe(val => console.log('Sub:', val));
```

Output:

```
Sub: B
Sub: C
```

Summary Table

Feature	`Subject`	`BehaviorSubject`	`ReplaySubject`
Remembers latest value	NO	YES	YES (multiple values)
Sends previous values	NO	YES(last only)	YES(as many as configured)
Requires initial value	No	Yes	Yes
Use case	Event stream	App state, current value	Caching, history

1. `Subject` — Refresh Trigger Example

Use case:

Trigger data reload across components.

`refresh.service.ts`

```
import { Injectable } from '@angular/core';
import { Subject } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class RefreshService {
  private refreshSubject = new Subject<void>();
  refresh$ = this.refreshSubject.asObservable();

  triggerRefresh() {
    this.refreshSubject.next();
  }
}
```

`data.component.ts`

```
@Component({ ... })
export class DataComponent implements OnInit {
  constructor(private refreshService: RefreshService) {}

  ngOnInit() {
    this.refreshService.refresh$.subscribe(() => {
      this.loadData();
    });
  }

  loadData() {
    console.log('Fetching data...');
    // e.g. this.http.get(...)
  }
}
```

`button.component.ts`

```
@Component({ ... })
export class ButtonComponent {
  constructor(private refreshService: RefreshService) {}

  refresh() {
    this.refreshService.triggerRefresh();
  }
}
```

2. `BehaviorSubject` — Shared Auth State

Use case:

Track & share the currently logged-in user across components.

`auth.service.ts`

```
@Injectable({ providedIn: 'root' })
export class AuthService {
  private userSubject = new BehaviorSubject<User | null>(null);
  user$ = this.userSubject.asObservable();

  login(user: User) {
    this.userSubject.next(user);
  }

  logout() {
    this.userSubject.next(null);
  }

  get currentUser(): User | null {
    return this.userSubject.value;
  }
}
```

`navbar.component.ts`

```
@Component({ ... })
export class NavbarComponent implements OnInit {
  user: User | null = null;

  constructor(private authService: AuthService) {}

  ngOnInit() {
    this.authService.user$.subscribe(user => this.user = user);
  }
}
```

3. `ReplaySubject` — Message History

Use case:

Chat messages that should be shown to new subscribers.

`chat.service.ts`

```
@Injectable({ providedIn: 'root' })
export class ChatService {
  private messagesSubject = new ReplaySubject<string>(5);
  messages$ = this.messagesSubject.asObservable();

  sendMessage(msg: string) {
    this.messagesSubject.next(msg);
  }
}
```

`chat-log.component.ts`

```
@Component({ ... })
export class ChatLogComponent implements OnInit {
  messages: string[] = [];
  constructor(private chatService: ChatService) {}

  ngOnInit() {
    this.chatService.messages$.subscribe(msg => this.messages.push(msg));
  }
}
```