1. `of` – Creates an Observable from arguments

typescript

Copy Edit

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

const observable$ = of(1, 2, 3, 4);
observable$.subscribe(value => console.log(value));
```

Output:

Copy Edit

```
1
2
3
4
```

2. `from` – Converts array, promise, or iterable into an Observable

typescript

Copy Edit

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

const observable$ = from([10, 20, 30]);
observable$.subscribe(value => console.log(value));
```

Output:

Copy Edit

```
10
20
30
```

3. `interval` – Emits an infinite sequence of ascending numbers in a given time interval

typescript

Copy Edit

```typescript
import { interval } from 'rxjs';

const observable$ = interval(1000); // emits every 1 second
observable$.subscribe(value => console.log(value));
```

Output:

python-repl

Copy Edit

```python-repl
0
1
2
3
...
```

4. `timer` – Emits after a delay, and optionally repeatedly

typescript

Copy Edit

```typescript
import { timer } from 'rxjs';

const observable$ = timer(2000); // emits once after 2 seconds
observable$.subscribe(value => console.log(value));
```

Output (after 2 seconds):

Copy Edit

```
0
```

5. `range` – Emits a sequence of numbers in a range

typescript

Copy Edit

```typescript
import { range } from 'rxjs';

const observable$ = range(5, 3); // starts at 5, emits 3 numbers
observable$.subscribe(value => console.log(value));
```

Output:

Copy Edit

```
5
6
7
```

6. `fromEvent` – Converts DOM events into Observables

typescript

Copy Edit

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

const clicks$ = fromEvent(document, 'click');
clicks$.subscribe(event => console.log('Clicked!', event));
```

✅ 1. `map` – Apply a function to each value emitted

Transforms each value using a projection function.

typescript

Copy Edit

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

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

Output:

Copy Edit

```
10
20
30
```

✅ 2. `mapTo` – Map each value to the same constant value

typescript

Copy Edit

```typescript
import { of } from 'rxjs';
import { mapTo } from 'rxjs/operators';

of('a', 'b', 'c')
  .pipe(mapTo('constant'))
  .subscribe(console.log);
```

Output:

pgsql

Copy Edit

```pgsql
constant
constant
constant
```

✅ 3. `pluck` – Extract a property from each emitted object

typescript

Copy Edit

```typescript
import { from } from 'rxjs';
import { pluck } from 'rxjs/operators';

const source$ = from([
  { name: 'Alice', age: 30 },
  { name: 'Bob', age: 25 }
]);

source$
  .pipe(pluck('name'))
  .subscribe(console.log);
```

Output:

nginx

Copy Edit

```nginx
Alice
Bob
```

✅ 4. `scan` – Like reduce, but emits the accumulated result over time

typescript

Copy Edit

```typescript
import { of } from 'rxjs';
import { scan } from 'rxjs/operators';

of(1, 2, 3, 4)
  .pipe(scan((acc, value) => acc + value, 0))
  .subscribe(console.log);
```

Output:

Copy Edit

```
1
3
6
10
```

✅ 5. `switchMap` – Map and switch to new inner observable (cancels previous)

typescript

Copy Edit

```typescript
import { fromEvent, interval } from 'rxjs';
import { switchMap } from 'rxjs/operators';

const clicks$ = fromEvent(document, 'click');

clicks$
  .pipe(switchMap(() => interval(1000)))
  .subscribe(console.log);
```

Explanation: Emits an increasing number every second, restarting on each click.

✅ 6. `concatMap` – Map and subscribe to inner observables one at a time, in order

typescript

Copy Edit

```typescript
import { of } from 'rxjs';
import { concatMap, delay } from 'rxjs/operators';

of(1, 2, 3)
  .pipe(
    concatMap(val => of(val * 10).pipe(delay(1000)))
  )
  .subscribe(console.log);
```

Output: Emits `10`, `20`, `30`, each 1 second apart, in order.

✅ 7. `mergeMap` – Map and flatten inner observables concurrently

typescript

Copy Edit

```typescript
import { of } from 'rxjs';
import { mergeMap, delay } from 'rxjs/operators';

of(1, 2, 3)
  .pipe(
    mergeMap(val => of(val * 10).pipe(delay(1000)))
  )
  .subscribe(console.log);
```

Output: Same values as `concatMap` but emitted **concurrently** (not guaranteed order).

✅ 8. `exhaustMap` – Ignores new emissions if one is already active

typescript

Copy Edit

```typescript
import { fromEvent, interval } from 'rxjs';
import { exhaustMap, take } from 'rxjs/operators';

fromEvent(document, 'click')
  .pipe(
    exhaustMap(() => interval(1000).pipe(take(3)))
  )
  .subscribe(console.log);
```

Explanation: On click, emits 0,1,2. Ignores further clicks until that completes.

✅ 1. `filter`

Emits only those items that pass the condition.

ts Copy Edit

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

of(1, 2, 3, 4, 5)
  .pipe(filter(x => x % 2 === 0))
  .subscribe(console.log);
```

Output: `2, 4`

✅ 2. `first`

Emits only the **first value** or the first that matches a condition.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { first } from 'rxjs/operators';

of(1, 2, 3)
  .pipe(first())
  .subscribe(console.log);
```

Output: `1`

✅ 3. `last`

Emits only the **last value** emitted by the source Observable.

ts Copy Edit

```ts
import { of } from 'rxjs';
import { last } from 'rxjs/operators';

of(1, 2, 3)
  .pipe(last())
  .subscribe(console.log);
```

Output: `3`

✅ 4. `take(n)`

Take the **first n values** emitted by the source.

ts

Copy Edit

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

of(1, 2, 3, 4, 5)
  .pipe(take(3))
  .subscribe(console.log);
```

Output: `1, 2, 3`

✅ 5. `takeLast(n)`

Take the **last n values** after the source completes.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { takeLast } from 'rxjs/operators';

of(1, 2, 3, 4, 5)
  .pipe(takeLast(2))
  .subscribe(console.log);
```

Output: `4, 5`

✅ 6. `skip(n)`

Skip the **first n values**.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { skip } from 'rxjs/operators';

of(1, 2, 3, 4)
  .pipe(skip(2))
  .subscribe(console.log);
```

Output: `3, 4`

✅ 7. `skipLast(n)`

Skip the **last n values.**

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { skipLast } from 'rxjs/operators';

of(1, 2, 3, 4)
  .pipe(skipLast(2))
  .subscribe(console.log);
```

Output: `1, 2`

✅ 8. `debounceTime(ms)`

Wait for the specified time before emitting the value (useful for type-ahead search).

ts

Copy Edit

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

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

✅ 9. `distinct`

Emits all items that are **distinct from the previous**.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { distinct } from 'rxjs/operators';

of(1, 2, 2, 3, 1)
  .pipe(distinct())
  .subscribe(console.log);
```

Output: `1, 2, 3`

✅ 10. `distinctUntilChanged`

Emits only when the **current value is different** from the last.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { distinctUntilChanged } from 'rxjs/operators';

of(1, 1, 2, 2, 3, 3)
  .pipe(distinctUntilChanged())
  .subscribe(console.log);
```

Output: `1, 2, 3`

✅ 11. `elementAt(index)`

Emits only the **value at a specific index.**

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { elementAt } from 'rxjs/operators';

of(10, 20, 30)
  .pipe(elementAt(1))
  .subscribe(console.log);
```

Output: `20`

✅ 12. `takeWhile(predicate)`

Take values while the condition is **true**.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { takeWhile } from 'rxjs/operators';

of(1, 2, 3, 4, 1)
  .pipe(takeWhile(x => x < 4))
  .subscribe(console.log);
```

Output: `1, 2, 3`

✅ 13. `skipWhile(predicate)`

Skip values **while the condition is true**, then take the rest.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { skipWhile } from 'rxjs/operators';

of(1, 2, 3, 4)
  .pipe(skipWhile(x => x < 3))
  .subscribe(console.log);
```

Output: `3, 4`

✅ 14. `auditTime(ms)`

Ignore values for a set time, then emit the **latest** one.

ts

Copy Edit

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

fromEvent(document, 'mousemove')
  .pipe(
    auditTime(1000),
    map((event: MouseEvent) => `X: ${event.clientX}, Y: ${event.clientY}`)
  )
  .subscribe(console.log);
```

🔗 1. `concat` – Sequentially emits all values from multiple Observables

ts

Copy Edit

```ts
import { of, concat } from 'rxjs';

const obs1$ = of(1, 2, 3);
const obs2$ = of(4, 5);

concat(obs1$, obs2$).subscribe(console.log);
```

Output: `1, 2, 3, 4, 5`

(Waits for obs1$ to complete before starting obs2$)

🔗 2. `merge` – Emits values from multiple Observables as they arrive (interleaved)

ts

Copy Edit

```ts
import { of, merge } from 'rxjs';
import { delay } from 'rxjs/operators';

const obs1$ = of('A', 'B').pipe(delay(1000));
const obs2$ = of('1', '2');

merge(obs1$, obs2$).subscribe(console.log);
```

Output: Interleaved based on timing (e.g., `1`, `2`, `A`, `B`)

🔗 3. `combineLatest` – Emits the latest values from each observable whenever **any** emits

ts

Copy Edit

```ts
import { combineLatest, interval } from 'rxjs';
import { map, take } from 'rxjs/operators';

const obs1$ = interval(1000).pipe(take(3)); // 0, 1, 2
const obs2$ = interval(1500).pipe(take(2)); // 0, 1

combineLatest([obs1$, obs2$])
  .pipe(map(([x, y]) => `obs1: ${x}, obs2: ${y}`))
  .subscribe(console.log);
```

Output:

yaml

Copy Edit

```yaml
obs1: 1, obs2: 0
obs1: 2, obs2: 0
obs1: 2, obs2: 1
```

🔗 4. `forkJoin` – Waits for all Observables to complete, then emits the last values as an array

ts

Copy Edit

```ts
import { of, forkJoin } from 'rxjs';
import { delay } from 'rxjs/operators';

const obs1$ = of('A').pipe(delay(1000));
const obs2$ = of('B').pipe(delay(2000));

forkJoin([obs1$, obs2$]).subscribe(console.log);
```

Output: `['A', 'B']`

(*Only after both complete*)

5. `zip` – Pairs values by index and emits them together

ts

Copy Edit

```ts
import { of, zip } from 'rxjs';

const names$ = of('Alice', 'Bob');
const scores$ = of(90, 80);

zip(names$, scores$).subscribe(([name, score]) => {
  console.log(`${name}: ${score}`);
});
```

Output:

makefile

Copy Edit

```makefile
Alice: 90
Bob: 80
```

6. `startWith` – Prepends an initial value before the source emits

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { startWith } from 'rxjs/operators';

of(2, 3, 4)
  .pipe(startWith(1))
  .subscribe(console.log);
```

Output: `1, 2, 3, 4`

🔗 7. `withLatestFrom` – Combine source value with latest value from another observable

ts

Copy Edit

```ts
import { interval } from 'rxjs';
import { withLatestFrom, map } from 'rxjs/operators';

const fast$ = interval(1000);
const slow$ = interval(3000);

fast$
  .pipe(
    withLatestFrom(slow$),
    map(([fastVal, slowVal]) => `Fast: ${fastVal}, Slow: ${slowVal}`)
  )
  .subscribe(console.log);
```

Output: Emits only when `fast$` emits, using the latest `slow$` value.

🔗 8. `race` – Emits from the first observable to emit, and ignores the rest

ts

Copy Edit

```ts
import { of, race } from 'rxjs';
import { delay } from 'rxjs/operators';

const obs1$ = of('A').pipe(delay(1000));
const obs2$ = of('B').pipe(delay(500));

race(obs1$, obs2$).subscribe(console.log);
```

Output: `B`

(*obs2$ wins the race*)

🔗 9. `combineLatestWith` – Same as `combineLatest` but used as a pipeable operator

ts

Copy Edit

```ts
import { interval } from 'rxjs';
import { combineLatestWith, map, take } from 'rxjs/operators';

interval(1000)
  .pipe(
    take(3),
    combineLatestWith(interval(1500).pipe(take(2))),
    map(([x, y]) => `X: ${x}, Y: ${y}`)
  )
  .subscribe(console.log);
```

✅ 1. `tap`

Performs a **side effect** (like logging) without modifying the stream.

ts

Copy Edit

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

of(1, 2, 3)
  .pipe(
    tap(value => console.log('Before map:', value)),
    map(value => value * 10),
    tap(value => console.log('After map:', value))
  )
  .subscribe();
```

Output:

arduino

Copy Edit

```arduino
Before map: 1
After map: 10
Before map: 2
After map: 20
Before map: 3
After map: 30
```

✅ 2. `finalize`

Executes code **when the observable completes or errors out.**

ts

Copy Edit

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

of('RxJS')
  .pipe(finalize(() => console.log('Observable completed')))
  .subscribe(console.log);
```

Output:

nginx

Copy Edit

```nginx
RxJS
Observable completed
```

✅ 3. `delay`

Delays each emission by the specified amount of time.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { delay } from 'rxjs/operators';

of('Hello')
  .pipe(delay(1000))
  .subscribe(console.log); // emits after 1 second
```

✅ 4. `delayWhen`

Delays each value based on another observable.

ts

Copy Edit

```ts
import { of, timer } from 'rxjs';
import { delayWhen } from 'rxjs/operators';

of('A', 'B', 'C')
  .pipe(delayWhen(() => timer(1000)))
  .subscribe(console.log); // All values delayed by 1 second
```

✅ 5. `timeout`

Throws an error if a value isn't emitted in the specified time.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { delay, timeout } from 'rxjs/operators';

of('Delayed value')
  .pipe(
    delay(2000),
    timeout(1000) // will throw an error because of 2s delay
  )
  .subscribe({
    next: val => console.log(val),
    error: err => console.error('Timeout error:', err.message)
  });
```

✅ 6. `repeat(n)`

Repeats the observable **n times**.

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { repeat } from 'rxjs/operators';

of('Hello')
  .pipe(repeat(3))
  .subscribe(console.log);
```

Output:

nginx

Copy Edit

```nginx
Hello
Hello
Hello
```

✅ 7. `retry(n)`

Retries the observable **n times on error.**

ts

Copy Edit

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

throwError(() => new Error('Failure'))
  .pipe(retry(2))
  .subscribe({
    error: err => console.log('Failed after retries:', err.message)
  });
```

✅ 8. `toPromise()` *(deprecated in RxJS 7, replaced by `firstValueFrom` or `lastValueFrom`)*

ts Copy Edit

```ts
import { of, firstValueFrom } from 'rxjs';

async function run() {
  const result = await firstValueFrom(of('RxJS'));
  console.log(result);
}
run();
```

✅ 9. `observeOn`

Changes the **scheduler** for downstream operators.

ts

Copy Edit

```ts
import { of, asyncScheduler } from 'rxjs';
import { observeOn } from 'rxjs/operators';

of('Hello')
  .pipe(observeOn(asyncScheduler))
  .subscribe(console.log); // executed asynchronously
```

✅ 1. `share()` – Share a single subscription among multiple subscribers

ts

Copy Edit

```ts
import { interval } from 'rxjs';
import { take, share } from 'rxjs/operators';

const source$ = interval(1000).pipe(
  take(3),
  share()
);

source$.subscribe(val => console.log('Subscriber A:', val));

setTimeout(() => {
  source$.subscribe(val => console.log('Subscriber B:', val));
}, 1500);
```

Output:

- Subscriber A: 0, 1, 2
- Subscriber B: Might get 1 and 2 depending on timing

✅ 2. `shareReplay()` – Shares the source and **replays** specified number of last values

ts

Copy Edit

```ts
import { of, delay } from 'rxjs';
import { shareReplay } from 'rxjs/operators';

const source$ = of(1, 2, 3).pipe(shareReplay(2));

source$.subscribe(val => console.log('First:', val));

setTimeout(() => {
  source$.subscribe(val => console.log('Second (replay):', val));
}, 1000);
```

Output:

sql

Copy Edit

```sql
First: 1
First: 2
First: 3
Second (replay): 2
Second (replay): 3
```

✅ 3. `publish()` + `connect()` – Manually control connection to source (older style)

ts

Copy Edit

```ts
import { interval } from 'rxjs';
import { publish, take } from 'rxjs/operators';

const source$ = interval(1000).pipe(take(3), publish()) as any;

source$.subscribe(val => console.log('A:', val));
source$.subscribe(val => console.log('B:', val));

source$.connect(); // starts emission manually
```

> ⚠️ `publish()` is deprecated in favor of `connectable()` or `share()` in modern RxJS.

✅ 4. `multicast()` – Share a subject among subscribers

ts

Copy Edit

```ts
import { interval, Subject } from 'rxjs';
import { multicast, take } from 'rxjs/operators';

const source$ = interval(1000).pipe(
  take(3),
  multicast(() => new Subject())
) as any;

source$.subscribe(val => console.log('Subscriber A:', val));
source$.subscribe(val => console.log('Subscriber B:', val));

source$.connect();
```

> ⚠️ Deprecated in RxJS 7+. Use `connectable()` instead.

✅ 5. `connectable()` (RxJS 7+) – New alternative to `publish()` and `multicast()`

ts

Copy Edit

```ts
import { connectable, interval } from 'rxjs';
import { take } from 'rxjs/operators';

const source$ = connectable(interval(1000).pipe(take(3)));

source$.subscribe(val => console.log('A:', val));
source$.subscribe(val => console.log('B:', val));

source$.connect();
```

✅ 1. `catchError` – Catch errors and return a fallback Observable

ts

Copy Edit

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

throwError(() => new Error('Oops!'))
  .pipe(
    catchError(err => {
      console.error('Caught error:', err.message);
      return of('Fallback value');
    })
  )
  .subscribe(console.log);
```

Output:

nginx

Copy Edit

```nginx
Caught error: Oops!
Fallback value
```

✅ 2. `retry(n)` – Retry the source Observable on error `n` times

ts

Copy Edit

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

throwError(() => new Error('Error!'))
  .pipe(retry(2)) // retries 2 times, then throws
  .subscribe({
    next: val => console.log(val),
    error: err => console.log('Failed after retries:', err.message)
  });
```

Output:

javascript

Copy Edit

```javascript
Failed after retries: Error!
```

✅ 3. `retryWhen` – Retry based on a custom condition or delay strategy

ts

Copy Edit

```ts
import { throwError, timer } from 'rxjs';
import { retryWhen, delayWhen } from 'rxjs/operators';

throwError(() => new Error('Oops'))
  .pipe(
    retryWhen(errors =>
      errors.pipe(delayWhen(() => timer(2000))) // retry after 2 seconds
    )
  )
  .subscribe({
    error: err => console.log('Still failed:', err.message)
  });
```

Behavior: Keeps retrying every 2 seconds until unsubscribed or an error is thrown that stops the stream.

✅ 4. `onErrorResumeNext` – Continue with another Observable on error (silently skips)

ts

Copy Edit

```ts
import { of, throwError, onErrorResumeNext } from 'rxjs';

onErrorResumeNext(
  throwError(() => new Error('fail')),
  of('Recovered', 'Continued')
).subscribe(console.log);
```

Output:

nginx

Copy Edit

```nginx
Recovered
Continued
```

✅ 5. `throwError` – Creates an Observable that immediately throws an error

ts

Copy Edit

```ts
import { throwError } from 'rxjs';

throwError(() => new Error('Custom error')).subscribe({
  error: err => console.log('Error:', err.message)
});
```

Output:

vbnet

Copy Edit

```vbnet
Error: Custom error
```

✅ 6. `defaultIfEmpty` – Emit a default value if the source completes without emitting

ts

Copy Edit

```ts
import { EMPTY } from 'rxjs';
import { defaultIfEmpty } from 'rxjs/operators';

EMPTY.pipe(defaultIfEmpty('Nothing emitted')).subscribe(console.log);
```

Output:

kotlin

Copy Edit

```kotlin
Nothing emitted
```

✅ 7. `timeout` – Throws an error if no value is emitted in a given time

ts

Copy Edit

```ts
import { of } from 'rxjs';
import { delay, timeout } from 'rxjs/operators';

of('Too late')
  .pipe(
    delay(2000),
    timeout(1000) // throws timeout error
  )
  .subscribe({
    next: val => console.log(val),
    error: err => console.log('Timeout:', err.message)
  });
```