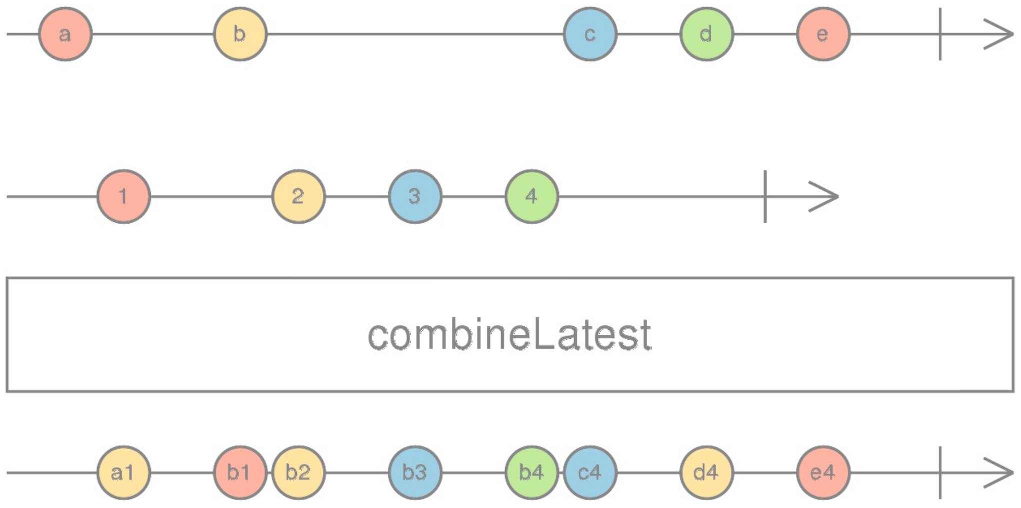
OPERATORS

Combination Operators

combineLatest

Combines multiple Observables to create an Observable whose values are calculated from the latest values of each of its input Observables.

```
public static combineLatest(observable1: ObservableInput, observable2:
ObservableInput, project: function, scheduler: Scheduler): Observable
```

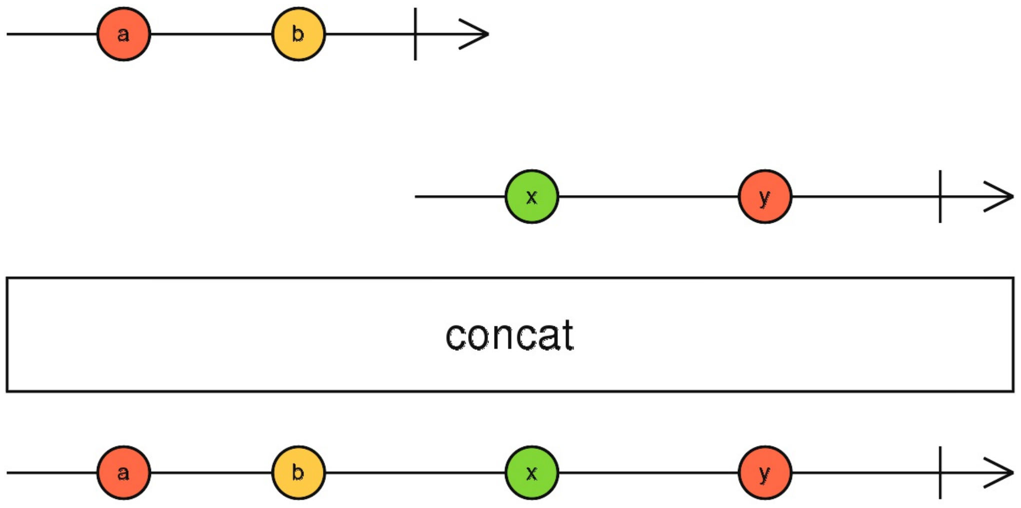


> Whenever any input Observable emits a value, it computes a formula using the latest values from all the inputs, then emits the output of that formula.

concat

Creates an output Observable which **sequentially** emits all values from given Observable and then moves on to the next.

```
public static concat(input1: ObservableInput, input2: ObservableInput,
scheduler: Scheduler): Observable
```

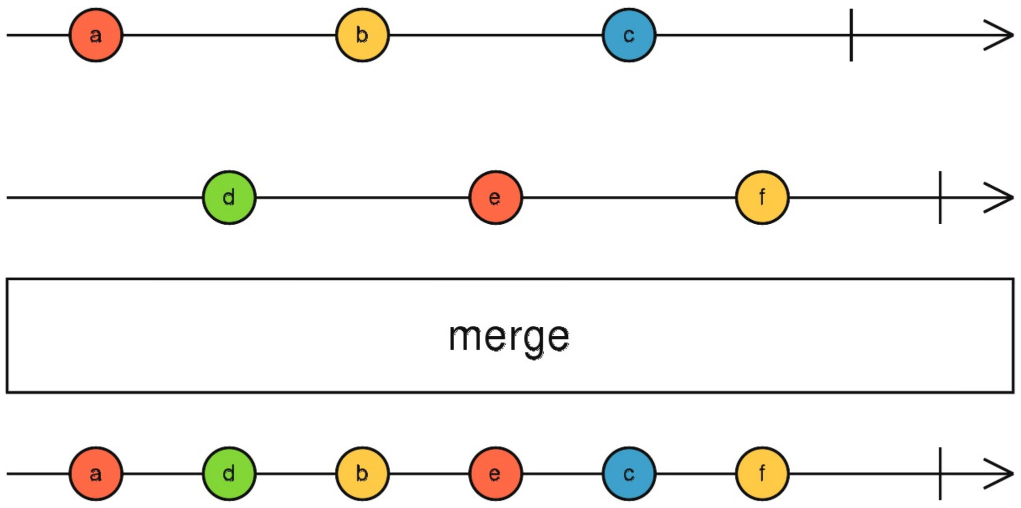


> Concatenates multiple Observables together by sequentially emitting their values, one Observable after the other.
>
> **note:** sequentially emits.

merge

Creates an output Observable which **concurrently** emits all values from every given input Observable.

```
public static merge(observables: ...ObservableInput, concurrent: number,
scheduler: Scheduler): Observable
```

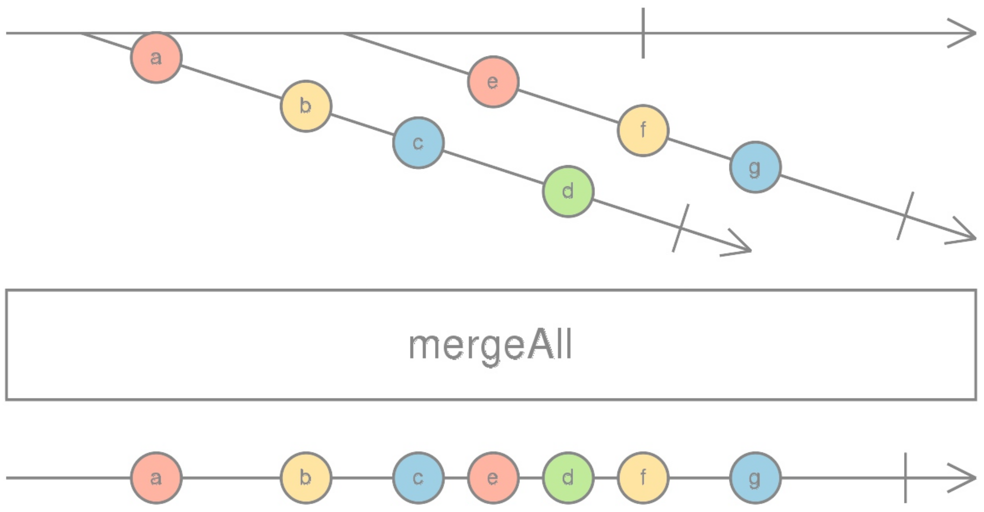


> Flattens multiple Observables together by blending their values into one Observable.
>
> **note:** concurrently emits

mergeAll

Converts a higher-order Observable into a first-order Observable which concurrently delivers all values that are emitted on the inner Observables.

```
public mergeAll(concurrent: number): Observable
```

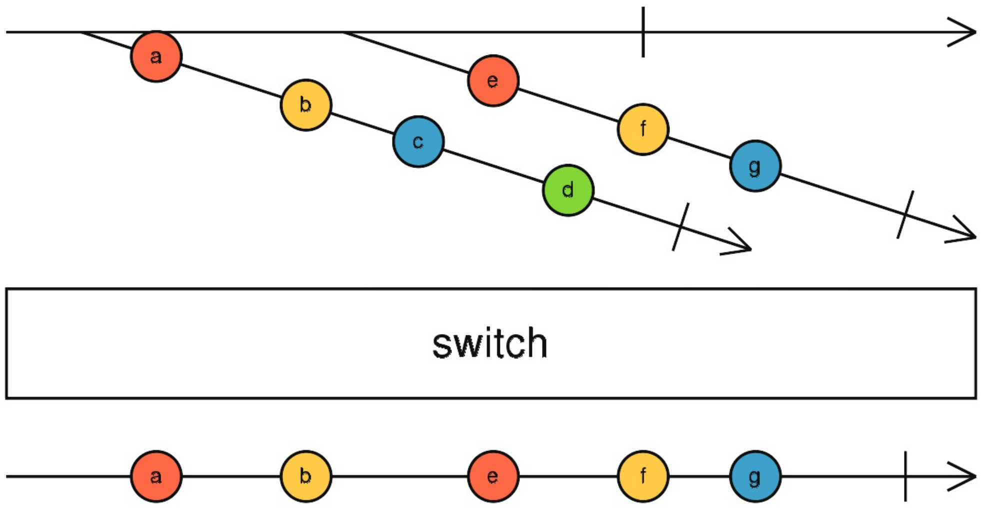


> Flattens an Observable-of-Observables.
>
> **notes** mergeAll merge and flatten Observables from the higher-order Observable; merge merge values from the Observable

switch

Converts a higher-order Observable into a first-order Observable by subscribing to only the most recently emitted of those inner Observables.

```
public switch(): Observable<T>
```

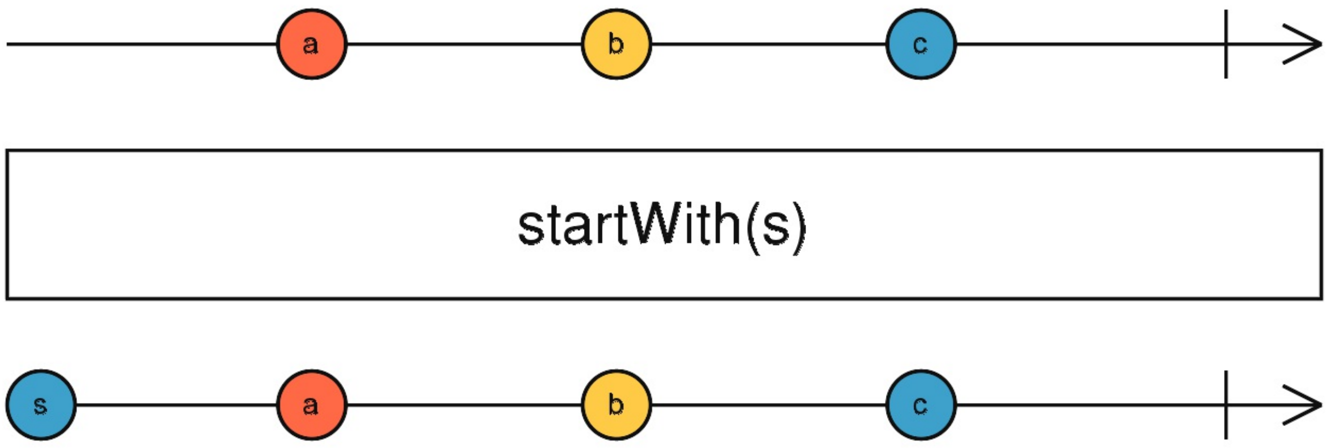


> Flattens an Observable-of-Observables by dropping the previous inner Observable once a new one appears.

startWith

Returns an Observable that emits the items you specify as arguments before it begins to emit items emitted by the source Observable.

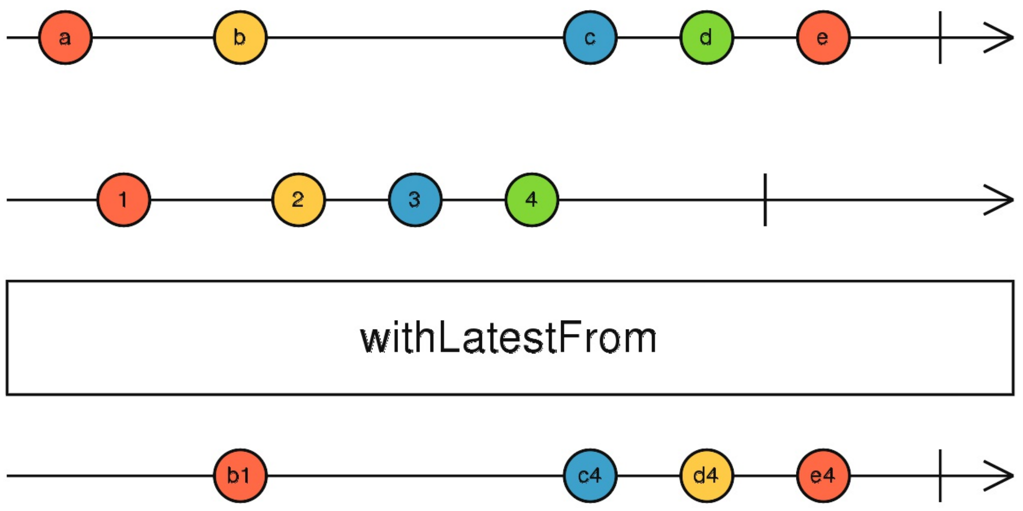
```
public startWith(values: ...T, scheduler: Scheduler): Observable
```



withLatestFrom

Combines the source Observable with other Observables to create an Observable whose values are calculated from the latest values of each, only when the source emits.

```
public withLatestFrom(other: ObservableInput, project: Function): Observable
```



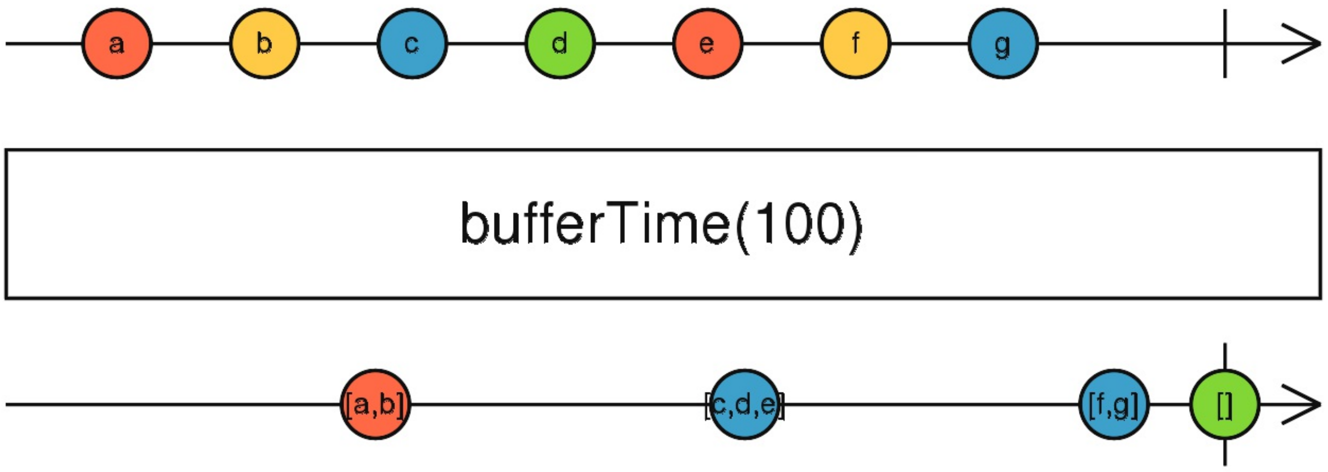
> Whenever the source Observable emits a value, it computes a formula using that value plus the latest values from other input Observables, then emits the output of that formula.

Transformation Operators

bufferTime

Buffers the source Observable values for a specific time period.

```
public bufferTime(bufferTimeSpan: number, bufferCreationInterval: number,
maxBufferSize: number, scheduler: Scheduler): Observable<T[]>
```

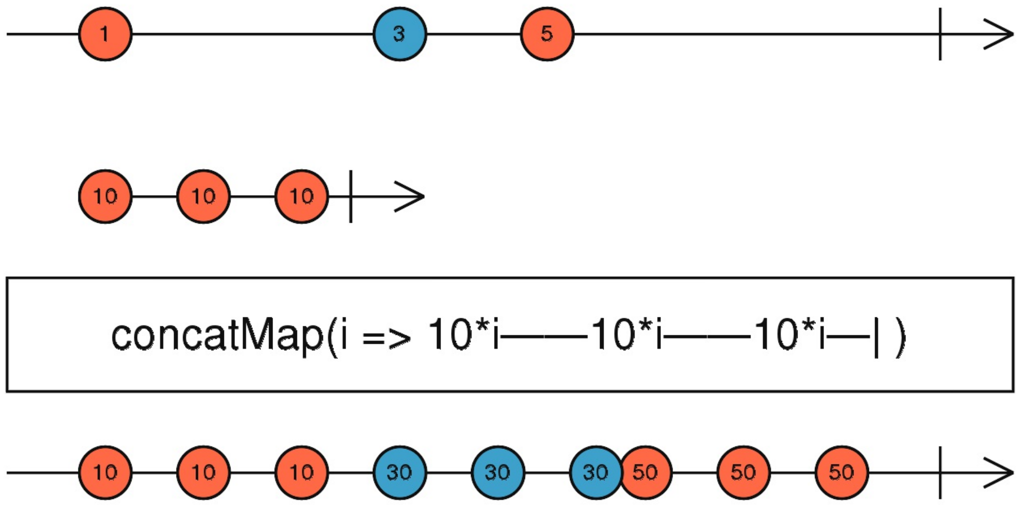


> Collects values from the past as an array, and emits those arrays periodically in time.

concatMap

Projects each source value to an Observable which is merged in the output Observable, in a serialized fashion waiting for each one to complete before merging the next.

```
public concatMap(project: function(value: T, ?index: number): ObservableInput,
resultSelector: function(outerValue: T, innerValue: I, outerIndex: number,
innerIndex: number): any): Observable
```

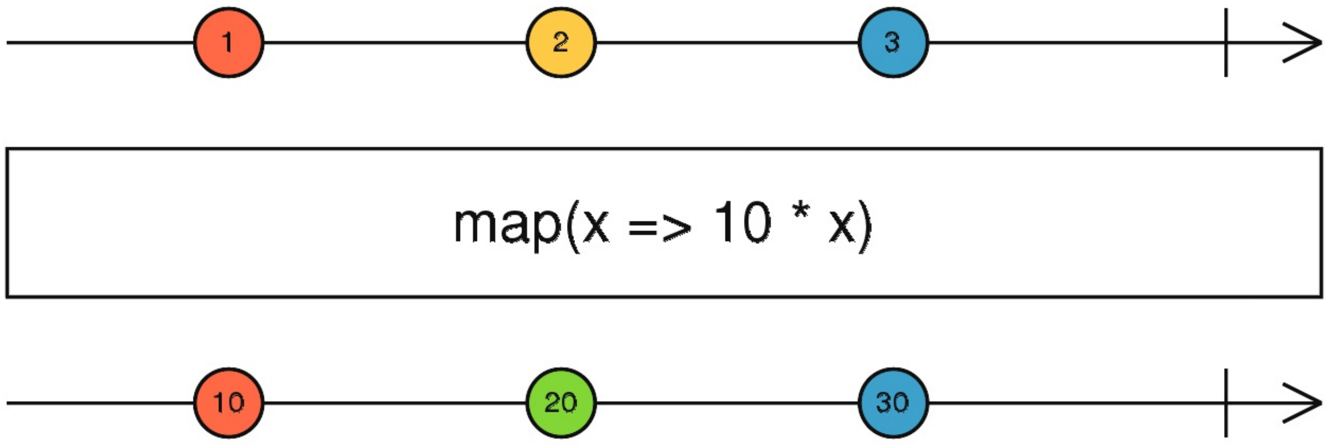


> Maps each value to an Observable, then flattens all of these inner Observables using concatAll.

map

Applies a given project function to each value emitted by the source Observable, and emits the resulting values as an Observable.

```
public map(project: function(value: T, index: number): R, thisArg: any):
Observable<R>
```

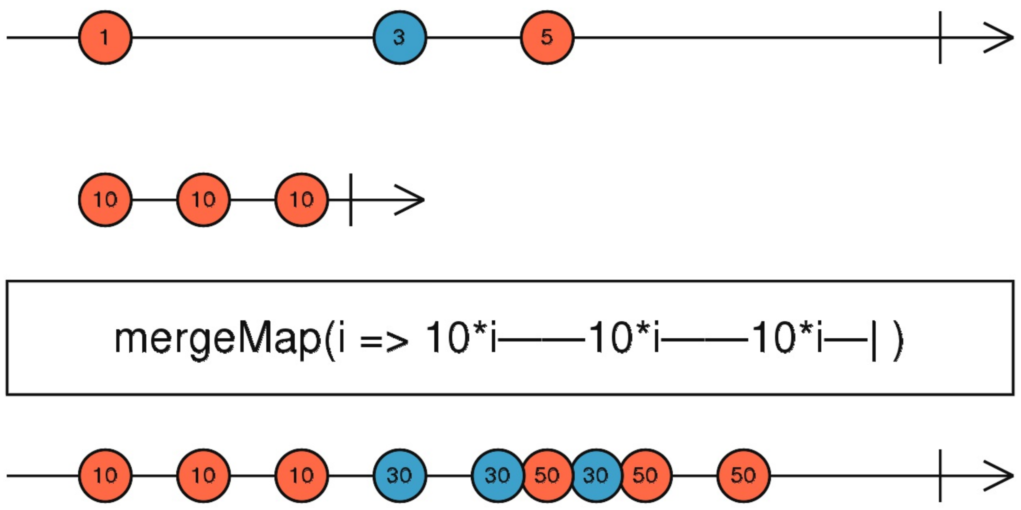


> Like Array.prototype.map(), it passes each source value through a transformation function to get corresponding output values.

mergeMap

Projects each source value to an Observable which is merged in the output Observable.

```
public mergeMap(project: function(value: T, ?index: number): ObservableInput,
resultSelector: function(outerValue: T, innerValue: I, outerIndex: number,
innerIndex: number): any, concurrent: number): Observable
```

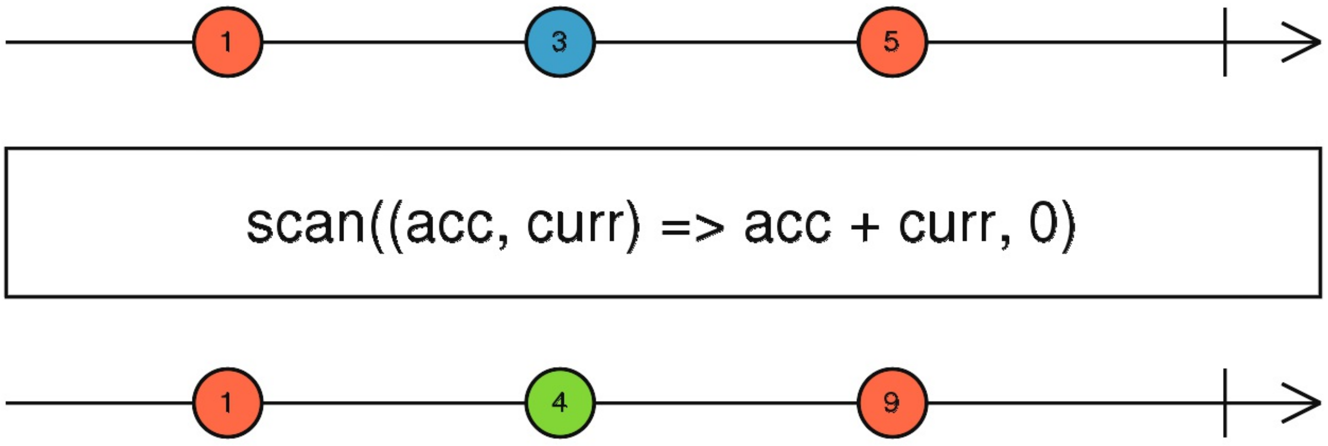


> Maps each value to an Observable, then flattens all of these inner Observables using mergeAll.

scan

Applies an accumulator function over the source Observable, and returns each intermediate result, with an optional seed value.

```
public scan(accumulator: function(acc: R, value: T, index: number): R, seed: T
| R): Observable<R>
```

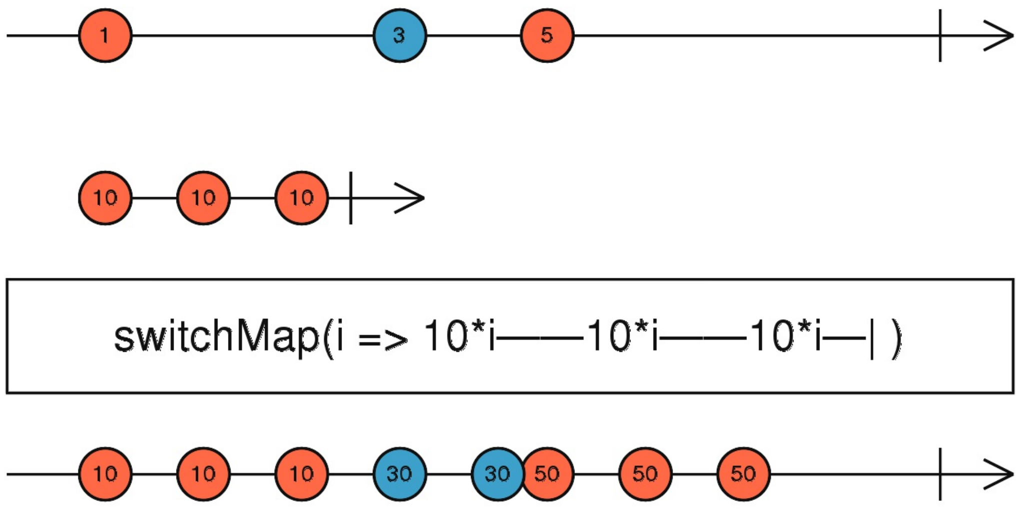


> It's like reduce, but emits the current accumulation whenever the source emits a value.

switchMap

Projects each source value to an Observable which is merged in the output Observable, emitting values only from the most recently projected Observable.

```
public switchMap(project: function(value: T, ?index: number): ObservableInput,
resultSelector: function(outerValue: T, innerValue: I, outerIndex: number,
innerIndex: number): any): Observable
```



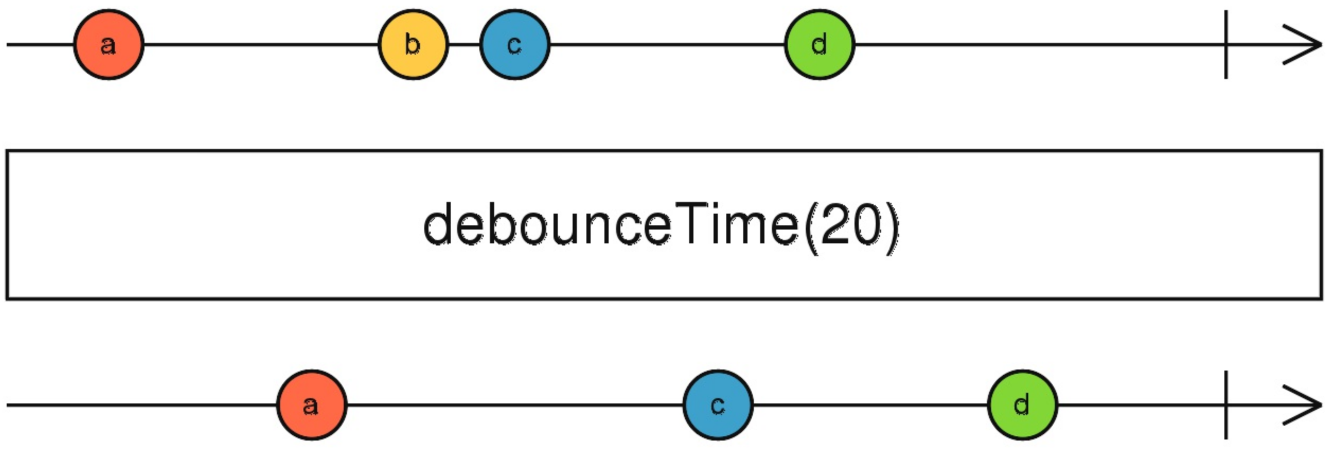
> Maps each value to an Observable, then flattens all of these inner Observables using switch.

Filtering Operator

debounceTime

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

```
public debounceTime(dueTime: number, scheduler: Scheduler): Observable
```



> It's like delay, but passes only the most recent value from each burst of emissions.

distinctUntilChanged

Returns an Observable that emits all items emitted by the source Observable that are distinct by comparison from the previous item.

If a comparator function is provided, then it will be called for each item to test for whether or not that value should be emitted.

If a comparator function is not provided, an equality check is used by default.

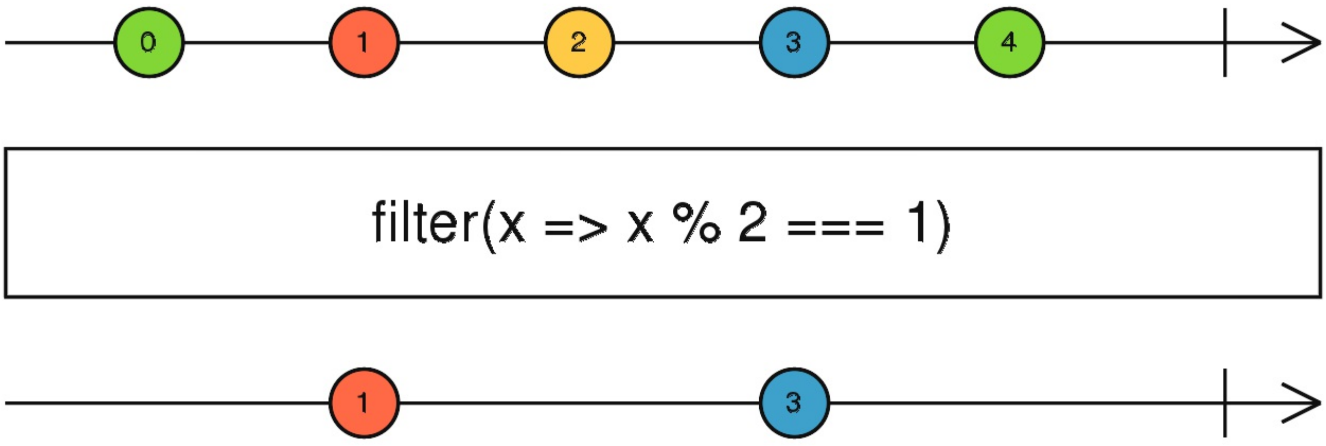
```
public distinctUntilChanged(compare: function): Observable
```

```
interface Person {
   age: number,
   name: string
}

Observable.of<Person>(
    { age: 4, name: 'Foo'},
    { age: 7, name: 'Bar'},
    { age: 5, name: 'Foo'})
    { age: 6, name: 'Foo'})
    .distinctUntilChanged((p: Person, q: Person) => p.name === q.name)
    .subscribe(x => console.log(x));

// displays:
// { age: 4, name: 'Foo' }
// { age: 7, name: 'Bar' }
// { age: 5, name: 'Foo' }
```

filter

Filter items emitted by the source Observable by only emitting those that satisfy a specified predicate.

```
public filter(predicate: function(value: T, index: number): boolean, thisArg:
any): Observable
```

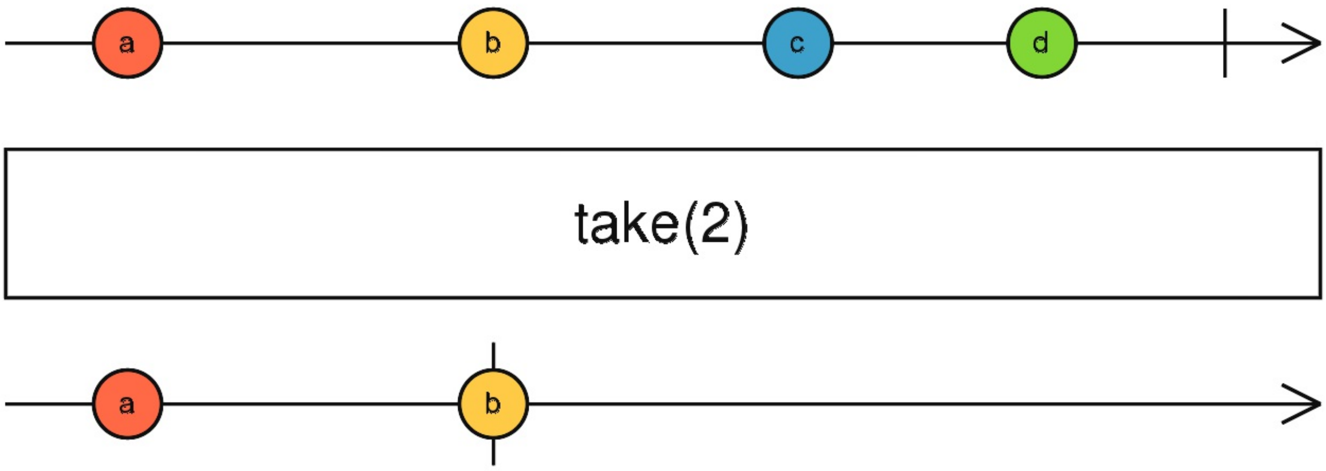


> Like Array.prototype.filter(), it only emits a value from the source if it passes a criterion function.

take

Emits only the first count values emitted by the source Observable.

```
public take(count: number): Observable<T>
```

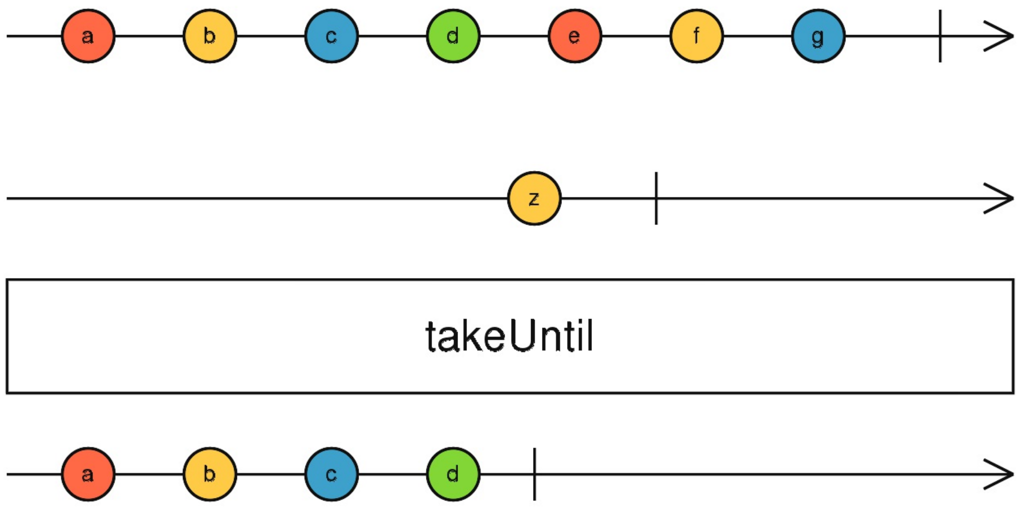


> Takes the first count values from the source, then completes.

takeUntil

Emits the values emitted by the source Observable until a notifier Observable emits a value.

```
public takeUntil(notifier: Observable): Observable<T>
```



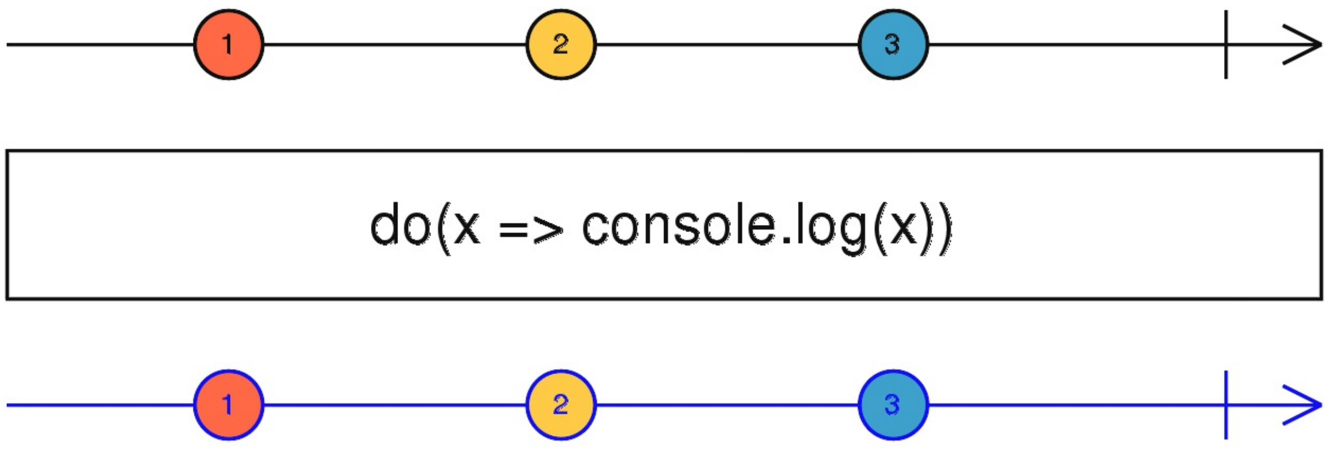
> Lets values pass until a second Observable, notifier, emits something. Then, it completes.

Utility Operators

do

Perform a side effect for every emission on the source Observable, but return an Observable that is identical to the source.

```
public do(nextOrObserver: Observer | function, error: function, complete:
function): Observable
```



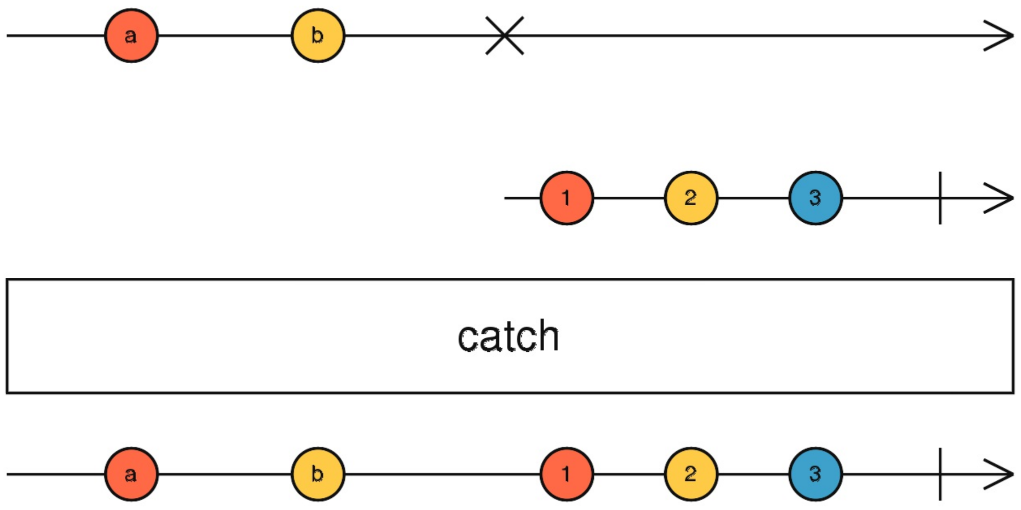
> Intercepts each emission on the source and runs a function, but returns an output which is identical to the source as long as errors don't occur.

Error Handling Operators

catch

Catches errors on the observable to be handed by returning a new observable or throwing an error.

```
public catch(selector: function): Observable
```



Multicasting Operators

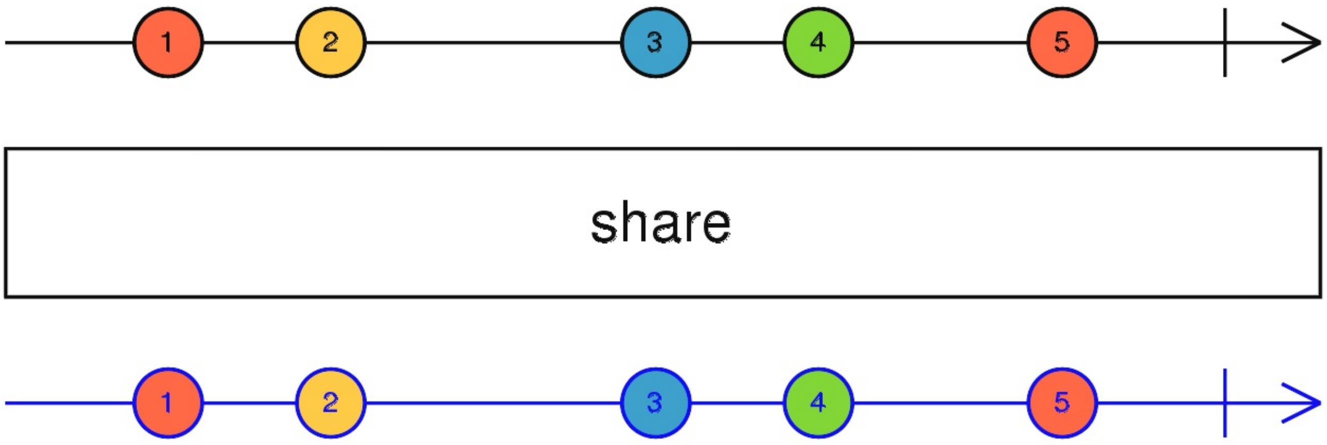
share

Returns a new Observable that multicasts (shares) the original Observable. As long as there is at least one

Subscriber this Observable will be subscribed and emitting data. When all subscribers have unsubscribed it will unsubscribe from the source Observable. Because the Observable is multicasting it makes the stream hot.

This behaves similarly to .publish().refCount(), with a behavior difference when the source observable emits complete. .publish().refCount() will not resubscribe to the original source, however .share() will resubscribe to the original source. Observable.of("test").publish().refCount() will not re-emit "test" on new subscriptions, Observable.of("test").share() will re-emit "test" to new subscriptions.

```
public share(): Observable<T>
```


1
2
3
4
5
share
1
2
3
4
5