FIT2102 Programming Paradigms Tutorial 4

Laziness and Observers

Faculty of Information Technology

Eager versus Lazy Evaluation

```
const eagerDate = Date();
const lazyDate = function() {
   return Date();
}
function printTime() {
   console.log("Eager: " + eagerDate)
   console.log("Lazy: " + lazyDate())
}
setTimeout(printTime, 1000)
```

TRIVIAL

But interesting!

```
> Eager: Sat Aug 12 2017 20:27:13 GMT+1000 (AUS Eastern Standard Time)
> Lazy: Sat Aug 12 2017 20:27:14 GMT+1000 (AUS Eastern Standard Time)
```

Eager Evaluation

JavaScript (and all imperative languages) evaluates expressions eagerly.

```
function sillyNaturalNumbers(initialValue:number): number {
    return sillyNaturalNumbers(initialValue + 1);
}
```

Lazy Evaluation

By wrapping an expression in a function, we delay its execution until we invoke the returned function:

```
function slightlyLessSillyNaturalNumbers(v) {
   return () => slightlyLessSillyNaturalNumbers(v+1)
}
```

But this is only slightly less silly because we have no way of getting the numbers out.

Infinite Sequence of Natural Numbers

```
interface LazySequence<T> {
   value: T;
   next():LazySequence<T>;
}

function naturalNumbers(): LazySequence<number> {
   return function _next(v:number): LazySequence<number> {
       return {
           value: v,
           next: ()=>_next(v+1)
       }
   }(1)
}
```

Infinite Sequence of Natural Numbers

```
interface LazySequence<T> {
   value: T;
   next():LazySequence<T>;
}

function initSequence<T>(transform: (value: T) => T):
   (initialValue: T) => LazySequence<T> {
   return function (val: T): LazySequence<T> {
       return {
           value: ???, // :T
           next: () => ??? // :LazySequence<T>
       }
   }
}
```

Observables

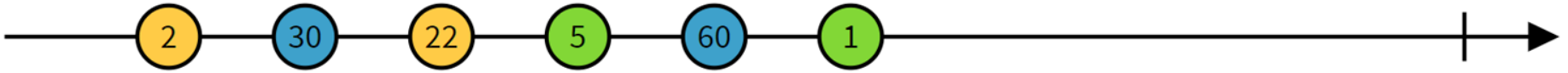
- Observables are lazy collections of multiple values over time.
- Can be used to model push-based data sources such as DOM events, timer intervals, and sockets.

```
import { fromEvent } from "rxjs"
import { map } from "rxjs/operators"
o = fromEvent<MouseEvent>(document, "mousemove")
            .pipe(map(({clientX, clientY})=> ({x: clientX, y: clientY})))
      .subscribe(({x,y})=> (console.log(x.ToString() + "," + y.ToString())));
```

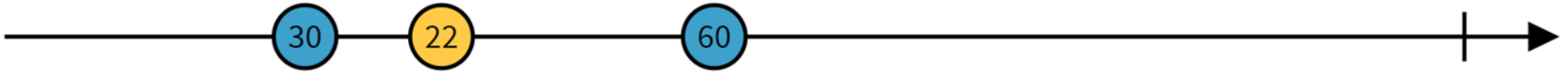
Our old friends!


```
map(x => 10 * x)
```

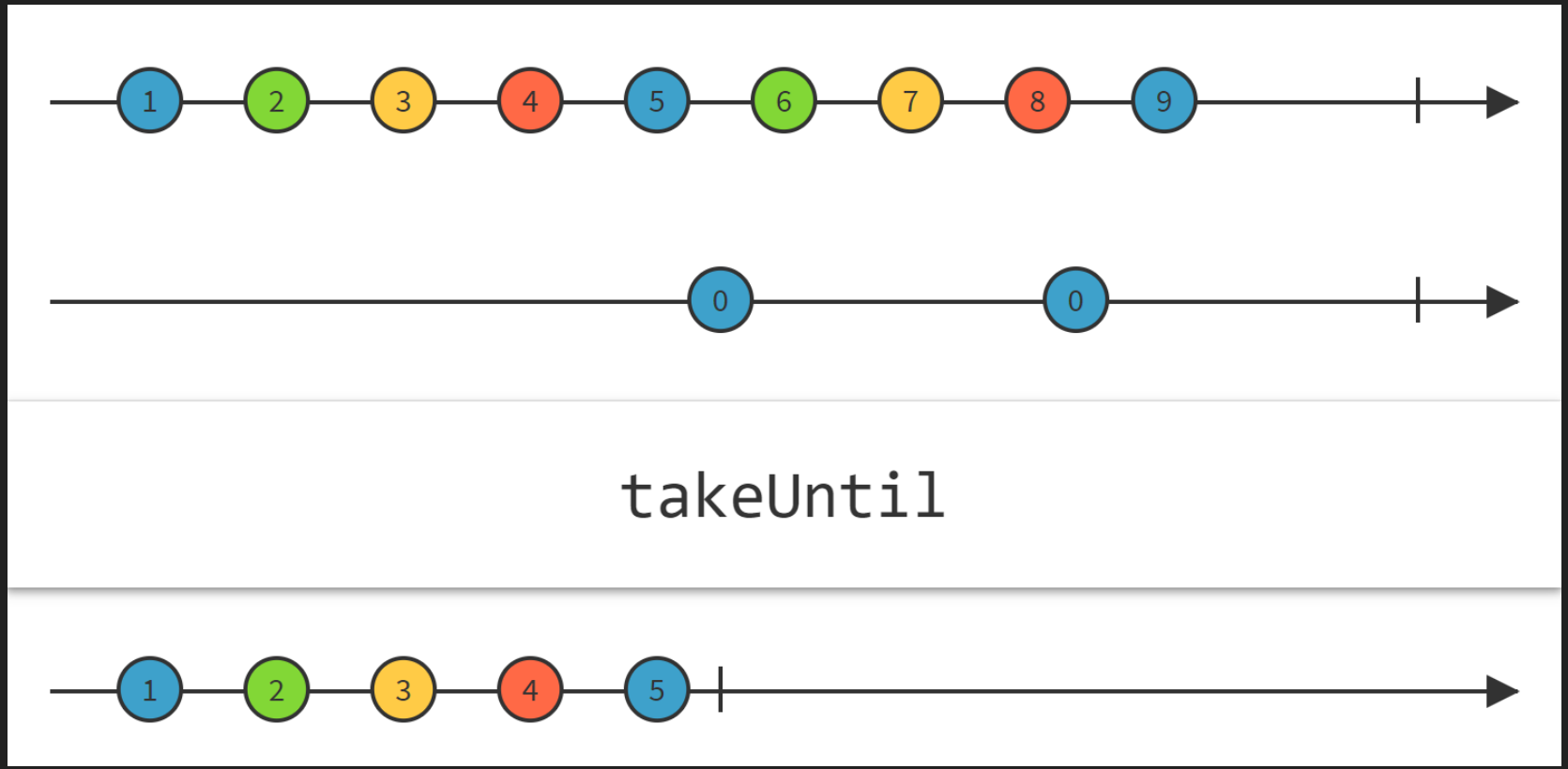
Our old friends!




```
filter(x => x > 10)
```



New Friend!



The mergeMap Operator

```
import { from } from "rxjs"
import { map, mergeMap } from "rxjs/operators"
from([1,2,3])
  .pipe(mergeMap(x=>from([4,5,6])
    .pipe(map(y=>[x,y]))))
  .subscribe(console.log)
```

```
[1, 4]
[1, 5]
[1, 6]
[2, 4]
[2, 5]
[2, 6]
[3, 4]
[3, 5]
[3, 6]
```