Webalkalmazás architektúrák

BME AUT
Automatizálási és
Alkalmazott
Informatikai Tanszék

Uni-directional data flow

Flux, Redux

Motiváció

- SPA – komplex állapotok
  - UI állapot
  - Aktuális route állapot
  - Lokális objektumok (szerverre nincsenek perzisztálva)
  - Szervertől lekért adatok
- Kétirányú adatkötések alkalmazásával
  - Sok, komplex állapotváltozás
  - Módosítások követhetetlen egymásra hatásai
  - Debuggolhatóság?
  - Teljesítmény?

- Mutáció
- Aszinkronitás

Uni-directional data flow

- Működés:
  - Adott egy objektum, amely leírja az alkalmazás teljes állapotát
  - Ez alapján legeneráljuk a UI-t
  - Ha változik a központi objektum, újra generálunk mindent
  - Felügyelt adatmódosítás
- Kérdések:
  - Egy központi objektum
  - Hatékonyság
  - Biztonság (tényleg nem tudjuk változtatni?)

ImmutableJS

- Könyvtár
- **Immutable**: nem változik az értéke
- **Persistent**: függvényhívások hatására új objektumokat kapunk

```
var map1 = Immutable.Map({a:1, b:2, c:3});
var map2 = map1.set('b', 50);
map1.get('b'); // 2
map2.get('b'); // 50
```

- Adattípusok: objektum, List, Stack, Map, OrderedMap, Set, OrderedSet, …
- Beágyazott struktúrák

Redux

- Unidirectional data flow
  - szükség van egy mechaniumusra az állapot tárolására
  - Módosítására
  - Változásértesítésre

- Alapelvek:
  - Single source of truth (store)
  - A központi állapot objektuma legyen readonly
  - Változtatások csak mellékhatás mentes függvényekkel (pure functions)

Redux

- Állapot
  - > Egyszerű JS objektum
  - > Nincsenek „setterek”

```
{
        selectedSemester: '2016',
        semesters: {
                id : '2016',
                students: [
                        { id : 1, name: X, exam1: 5, ... },
                        { id : 2, name: Y, exam1: 5, ... },
                        ...
                        { id : 10, name: Z, exam1: 5, ... },
                ]
        },
        ...
}
```

Redux

- Módosítás (**action**)
- Egyszerű JS objektum

```
{ type: 'set selected semester', newSemesterId : '2016'}

{ type: 'add student to semester', student : { id : 1, name: X, exam1: 5, ... } }

{ type: 'delete current semester' }
```

Redux

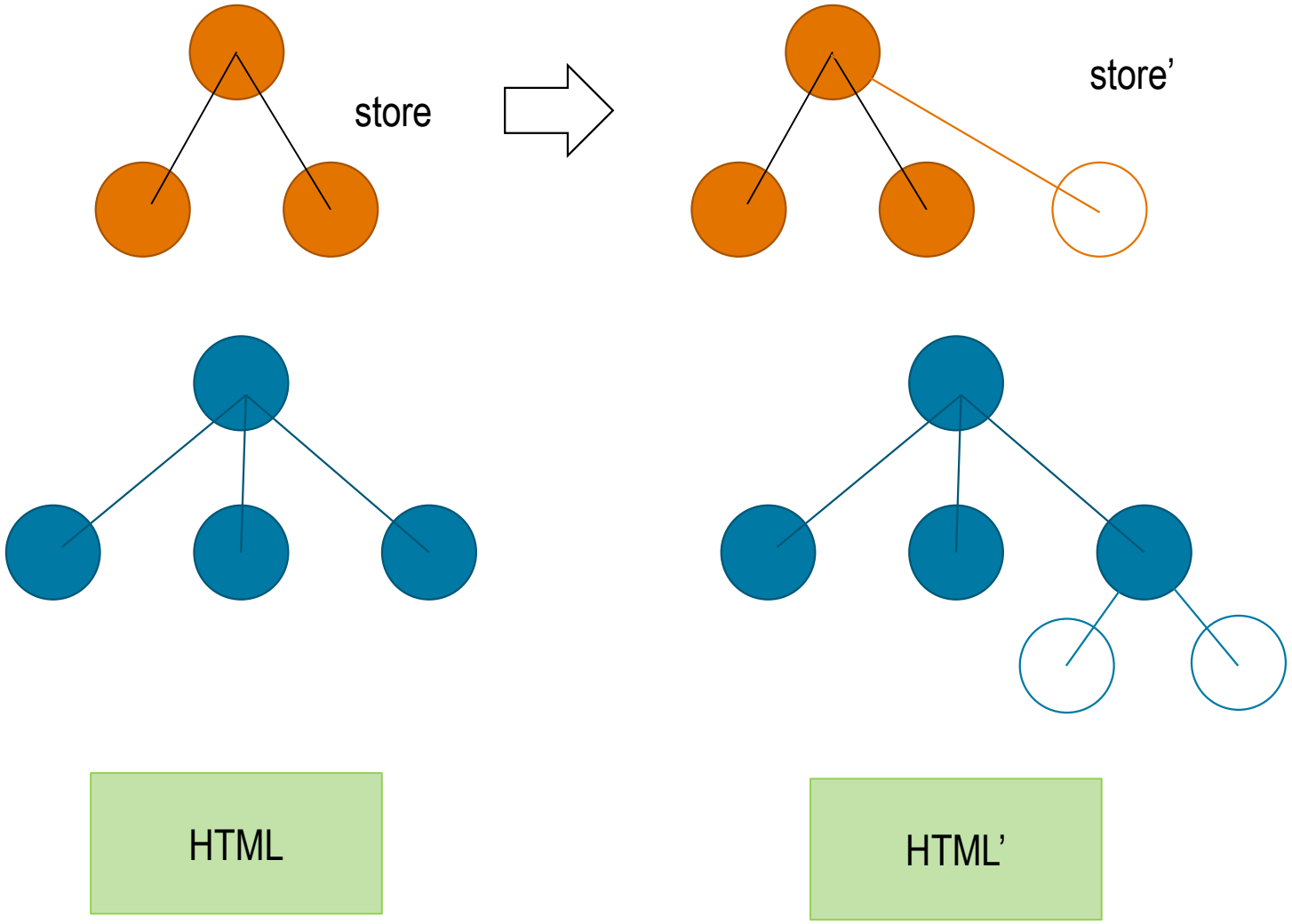
- Store frissítése az action-ök alapján: **reducer**
  - > Mindig új objektum-ot kell visszaadni (segít pl. az ImmutableJS)

```
function reducer(state, action) {
        switch (action.type) {
                case 'set selected semester':
                        return {...state, selectedSemester: action.newSemesterId };
                ...
        }
}
```

Redux

- Store:
  - Központi állapot tárolása
  - getState()
  - dispatch(action)
  - subscribe(listener)

React + Redux



Tudjuk, hogy mi változott

Tudjuk a különbséget az előző DOMhoz képest

Csak a tényleges változtatást (törlés, létrehozás, módosítás) kell emittálni.

Aszinkronitás

Eseményfeliratkozás

```
var div1 = selectDiv();

div1.addEventListener('load', function() {
  // inicializálás
});

div1.addEventListener('error', function() {
  // hibakezelés
});
```

- Események előbb bekövetkezhetnek, mint a feliratkozás

https://developers.google.com/web/fundamentals/getting-started/primers/promises

Eseményfeliratkozás

```
div1.callThisIfLoadedOrWhenLoaded(function() {
  // loaded
}).orIfFailedCallThis(function() {
  // failed
});

// and…
whenAllTheseHaveLoaded([div1, div2]).callThis(function() {
  // all loaded
}).orIfSomeFailedCallThis(function() {
  // one or more failed
});
```

- Valami ilyesmi kellene

Callback metódusok

```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });
      }
    })
  }
});
```

- Nehézkes a többszörös egymásbaágyazás

Promise pattern

- Promise: egy aszinkron hívás eredménye
- Állapotok:
  - > pending (várakozik)
  - > fulfilled (sikeres)
  - > Rejected (sikertelen
- Feliratkozhatunk a sikerre, vagy a hibára
- „Szabványok”
  - > Többféle implementáció (Q, jQuery (Deferred), when, WinJS, …)
  - > ES6 része

Promise API

- Létrehozás:
  - Konstans objektumból
  - Sok beépített aszinkron függvény ilyennel tér vissza (pl. fetch)

```
var promise = new Promise(function(resolve, reject) {
        if (...) resolve(/*value*/);
        else reject(/*reason*/);
});

var promise2 = Promise.resolve(5);
```

Promise API

- Feliratkozás
  - then pattern: sikerre, hibára külön callback
  - catch: csak a hibára callback

```
promise.then(function(value) {...});
promise.then(function(value) {...}, function (error) {...});
promise.catch(function (error) {...});
```

Promise API

- Várakozás több promise-ra:
  - all: mindegyikre
  - race: elsőre, ami visszatér

```
Promise.all(p1, p2, p3);
Promise.race(p1, p2, p3);
```

Promise API

- Promise chaining

```
var promise = Promise.resolve(2);
Promise
.then(value => {
        console.log(value); // 4
        return value * 2;
} .then(value => {
        console.log(value); // 8
        return value * 2;
} .then(value => {
        console.log(value); // 16
        return value * 2;
}
```

Async/Await (ES6, TS)

- Speciális nyelvi elemek, amik a promise mintával együttműködnek.
- async-kal megjelölt függvények:
  - Promise-szal térnek vissza

```
async function f() {
  return 1;
}

f().then(alert); // 1

async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

Async/Await (ES6, TS)

- await:
  - > Csak async függvényen belül
  - > Utána egy promise áll
  - > Az **await promise** kifejezés eredménye a promise eredménye
- Mikor tér vissza?

```
async funcion f() {
  let promise = fetch("http://getjson.com/");
  let json = await promise;
}

async function f2() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise;
  alert(result); // "done!"
}

f2();
```

Async/Await (ES6, TS)

- Mi történik hiba esetén?

```
async function f() {
  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

Reaktív programozás

- Aszinkron esemény alapú programozási minta
- Observable minta alapú megvalósítás
- Olvashatóbb kód, mert kevesebb callback hívás lesz

- Sok programozási nyelvre: RxJS, RxJava, Rx.NET, RxScala stb.

Observable

- *Lazy Push collection of multiple values*
- Adatok sorozata,
- Nem tudjuk mikor jön a következő
- Feliratkozhatunk, leiratkozhatunk
- Nincs history

	SINGLE	MULTIPLE
Pull	`Function`	`Iterator`
Push	`Promise`	`Observable`

Observable

- = Stream (=$): értékek időben egymás utáni sorozata
- Lehetséges események:
  - > Új elem érkezett
  - > Véget ért a stream
  - > Hiba
- Mindegyik eseményre feliratkozhatunk (subscribe)

- Reaktív programozás: a lehetséges események sorozatára reagál az alkalmazásunk

Observable

- Observable stream előállítása:
  - Konstans értékekből

```
var observable = Observable.from([1, 2, 3, 4]);
```

Observable

- Observable stream előállítása:
  - Generátor függvénnyel

```
const observable = Observable.create(function (observer) {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  setTimeout(() => {
    observer.next(4);
    observer.complete();
  }, 1000);
});
```

Observable

- Observble stream előállítása:
  - > Subject/EventEmitter

```
const s = new Subject();
```

```
s.next(1);
```

```
s.next(2);
```

Observable

- Observable stream előállítása:
  - > Promise-ból

```
const observable = Observable.fromPromise(promise);
```

Observable

- Observable stream előállítása:
  - DOM eseményekből

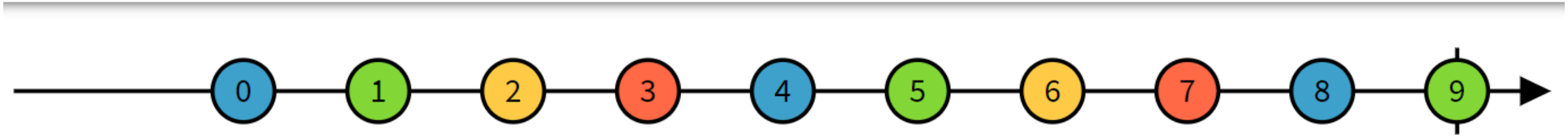
```
var observable1 = fromEvent(document, 'click');

var observable2 = Rx.Observable.fromEvent(document.getElementById("myButton"), 'click');
```

Observable

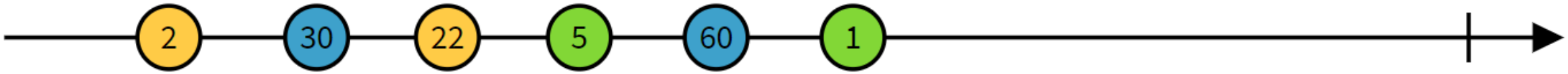
- Observable stream előállítása:
  - Más observable-ökből komponáló operátorokkal

Rx Operátorok

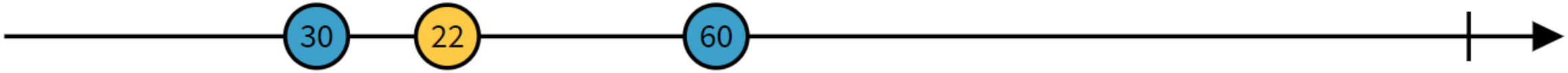
```
Observable.interval(10)
```



Rx Operátorok

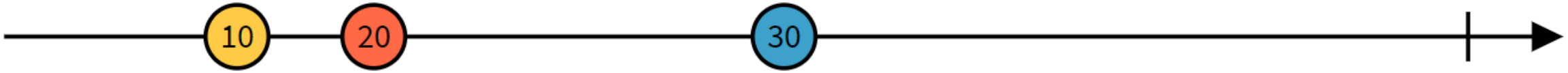



```
filter(x => x > 10)
```

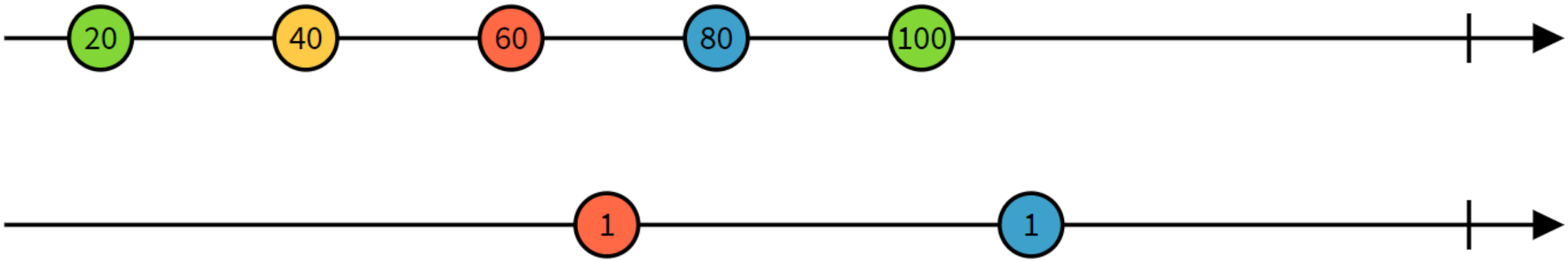


Rx Operátorok

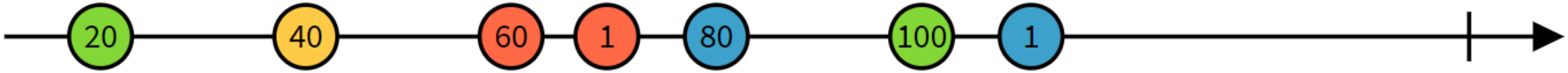

```
map(x => 10 * x)
```



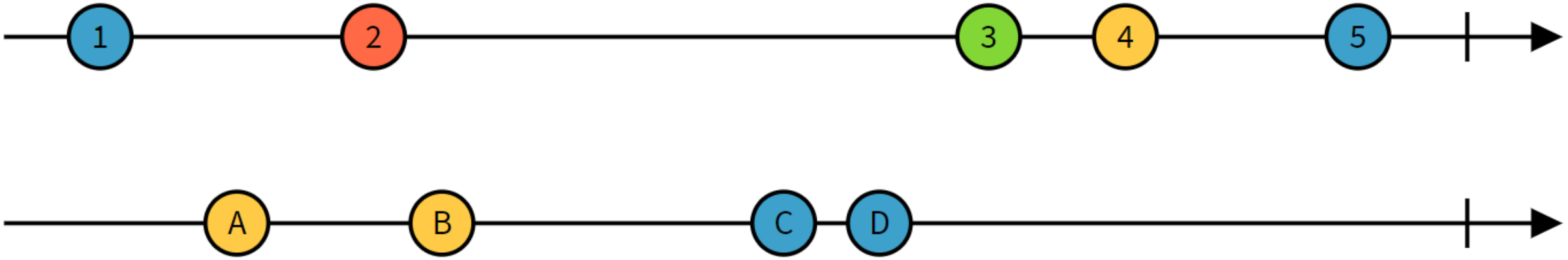
Rx Operátorok



merge



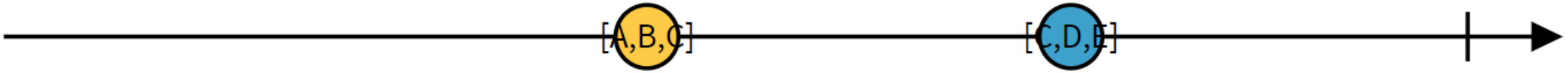
Rx Operátorok



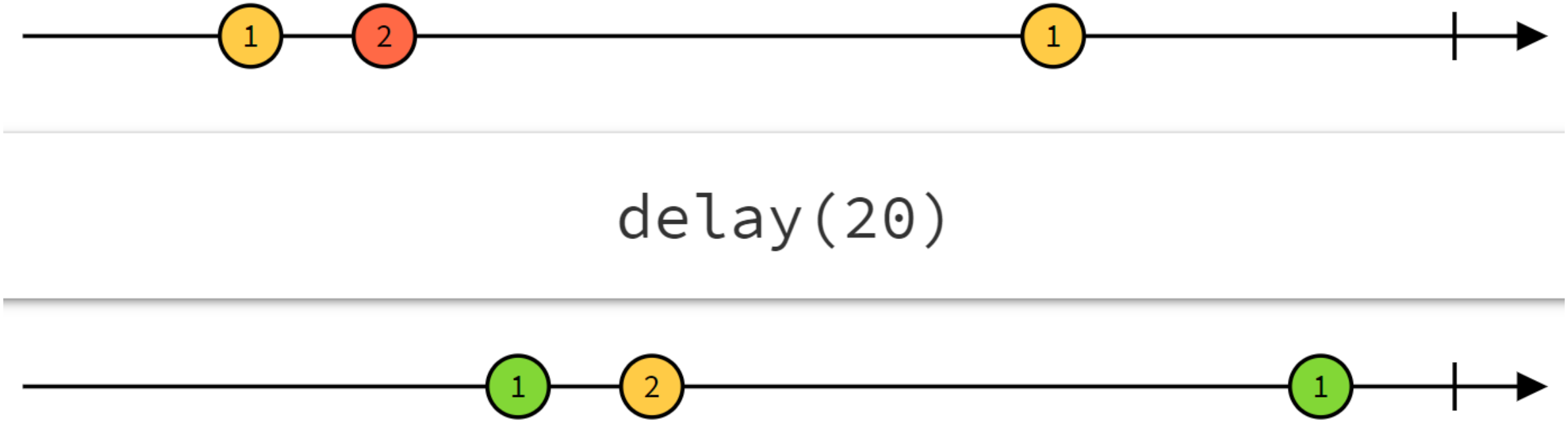

```
combineLatest((x, y) => "" + x + y)
```

Rx Operátorok

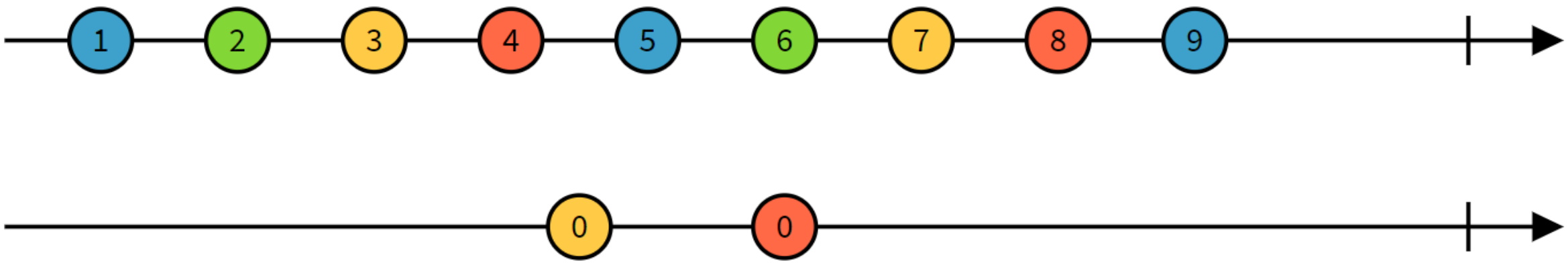

```
bufferCount(3, 2)
```



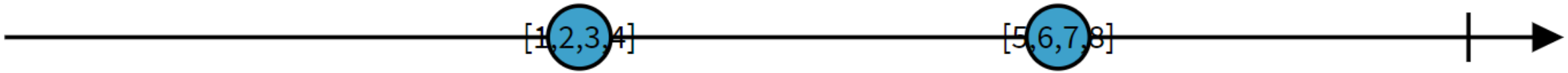
Rx Operátorok



Rx Operátorok



bufferWhen



Observable feliratkozás

- subscribe(
  successCallback,
  errorCallback?,
  completedCallback?)

```
var clicks = Rx.Observable.fromEvent(document, 'click');
clicks.subscribe(x => console.log(x));
```

RxJS példa

```
Rx.Observable
  .fromEvent(button, 'click')
  .bufferCount(3)
  .subscribe(() => {
    //a gomb minden 3. kattintására fog lefutni
  });
```

RxJS példa

```
const click$ = Rx.Observable.fromEvent(button, 'click');
click$
  .bufferWhen(() => click$.delay(500))
  .filter(events => events.length >= 3)
  .subscribe((res) => {
        // ha 500 ezred másodpercen belül legalább 3-szor kattintott
});
```

RxJS példa

```
let txtBox1 = document.getElementById('txtBox1');
let txtBox2 = document.getElementById('txtBox2');

const txtBox1Input$ = Rx.Observable.fromEvent(txtBox1, 'input').map(e => e.target.value);
const txtBox2Input$ = Rx.Observable.fromEvent(txtBox2, 'input').map(e => e.target.value);

Observable
        .combineLatest(txtBox1Input$, txtBox2Input$)
        .subscribe(values => {
                let txtBox1Value = values[0];
                let txtBox2Value = values[1];
                //A két szövegdoboz közül valamelyiknek változott a tartalma
        });
```

Redux + Observables + Angular

- DEMO