Лекція №10. Event Loop.

Будь-яка операція реактивного програмування RxJS реалізується шляхом комбінування режиму спостерігача, ітеративного режиму та функціонального програмування. В основному цей режим має такі концепції:

- Observable (об'єкт, що спостерігається): колекція створюваних майбутніх значень або подій.
- Observer (Спостерігач): набір функцій зворотного виклику, який знає, як відстежувати значення, що надаються Observable.
- Підписка: представляє виконання Observable і в основному використовується для скасування виконання Observable.
- Оператори: чисті функції, що використовують стиль функціонального програмування, використовують такі оператори, як map, filter, concat, flatMap тощо. Використовуються для обробки колекцій.
- Subject: еквівалентна EventEmitter і є єдиним способом мультиплексування значень чи подій для декількох спостерігачів.
- Планувальники (Schedulers): використовуються для управління паралелізмом і представляють централізований планувальник, що дозволяє координувати виконання обчислень, наприклад setTimeout або requestAnimationFrame або інші.

Event Loop (Цикл подій)

До специфікації ES6 JavaScript, незважаючи на те, що він дозволяв виконувати асинхронні виклики (на зразок `setTimeout`), не містив вбудованих механізмів асинхронного програмування. JS-движки займалися лише однопоточним виконанням деяких фрагментів коду, по одному за раз.

Отже, хто повідомляє JS-движку (наприклад V8) про те, що він має виконати якийсь фрагмент програми? Насправді движок не працює в ізоляції - його власний код виконується всередині якогось оточення, яким, для більшості розробників, є або браузер, або Node.js. Насправді, в наші дні існують JS-движки для різних видів пристроїв - від роботів до розумних лампочок. Кожен такий пристрій представляє власний варіант оточення для JS-движка.

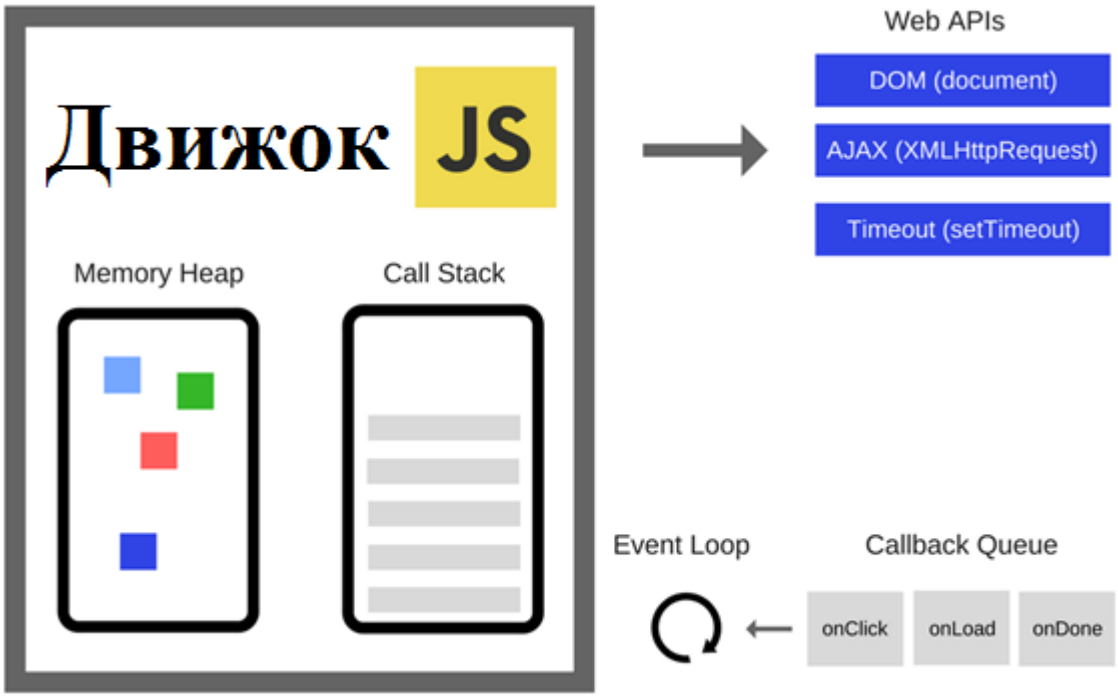
Загальною характеристикою всіх таких середовищ є вбудований механізм, який називається циклом подій (Event Loop). Він підтримує виконання фрагментів програми, викликаючи при цьому JS-движок.

Це означає, що движок можна вважати середовищем виконання будь-якого JS-коду, що викликається на вимогу. А плануванням подій (тобто сеансів виконання JS-коду) займаються механізми оточення, зовнішні по відношенню до движка.

Отже, наприклад, коли програма виконує Ajax-запит для завантаження якихось даних із сервера, то треба використати команду для запису цих даних у змінну `response` в середині коллбека, і JS-движок повідомляє оточенню: "Я збираюся призупинити виконання програми, але коли ти закінчиш виконувати цей мережевий запит і отримаєш якісь дані, будь ласка, виклич цей коллбек".

Потім браузер встановлює прослуховувач, що очікує відповідь від мережевої служби, і коли в нього є щось, що можна повернути в програму, що виконала запит, він планує виклик колбека, додаючи його до циклу подій.

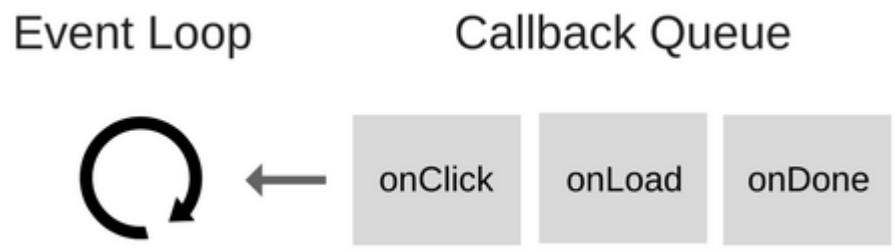
Коротко, це можні зобразити таким чином:



Web API - це потоки, до яких у розробника немає прямого доступу, але він може лише виконувати звернення до них. Вони вбудовані в браузер, де виконуються асинхронні дії.

Якщо розробка ведеться під Node.js, подібні API реалізовані засобами C++.

Цикл подій, у даному випадку, вирішує одне основне завдання: спостерігає за стеком викликів (Call Stack) та чергою коллбеків (Callback Queue). Якщо стек викликів порожній, цикл бере першу подію з черги і поміщає її в стек, що призводить до запуску цієї події на виконання.



Подібна ітерація називається тіком циклу подій. Кожна подія це просто коллбек.

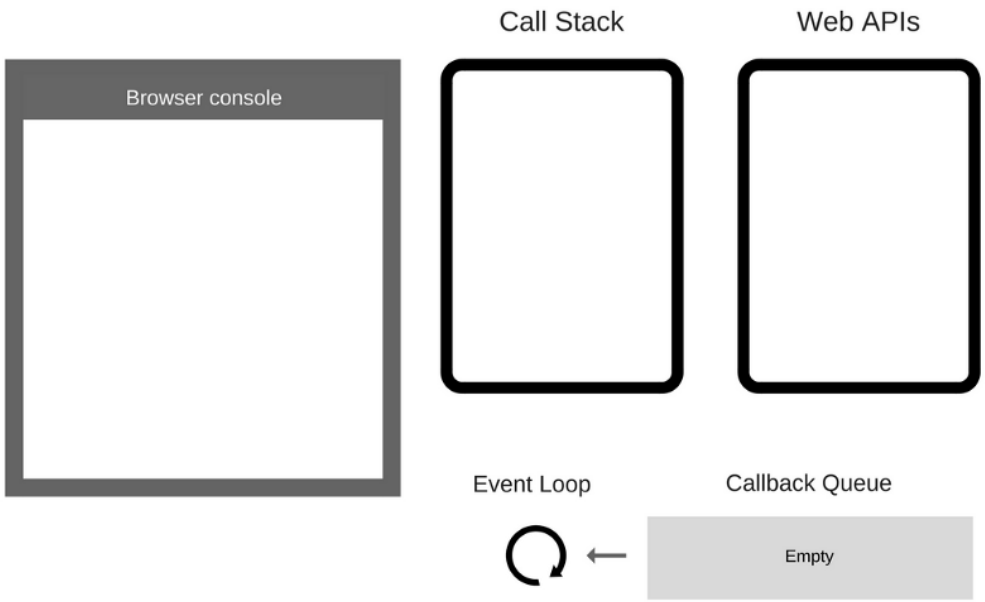
Розглянемо наступний приклад:

```
console.log('Hi');
setTimeout(function cb1() {
  console.log('cb1');
  }, 5000);
console.log('Bye');
```

Займемося покроковим виконанням цього коду і подивимося, що при цьому відбувається в системі.

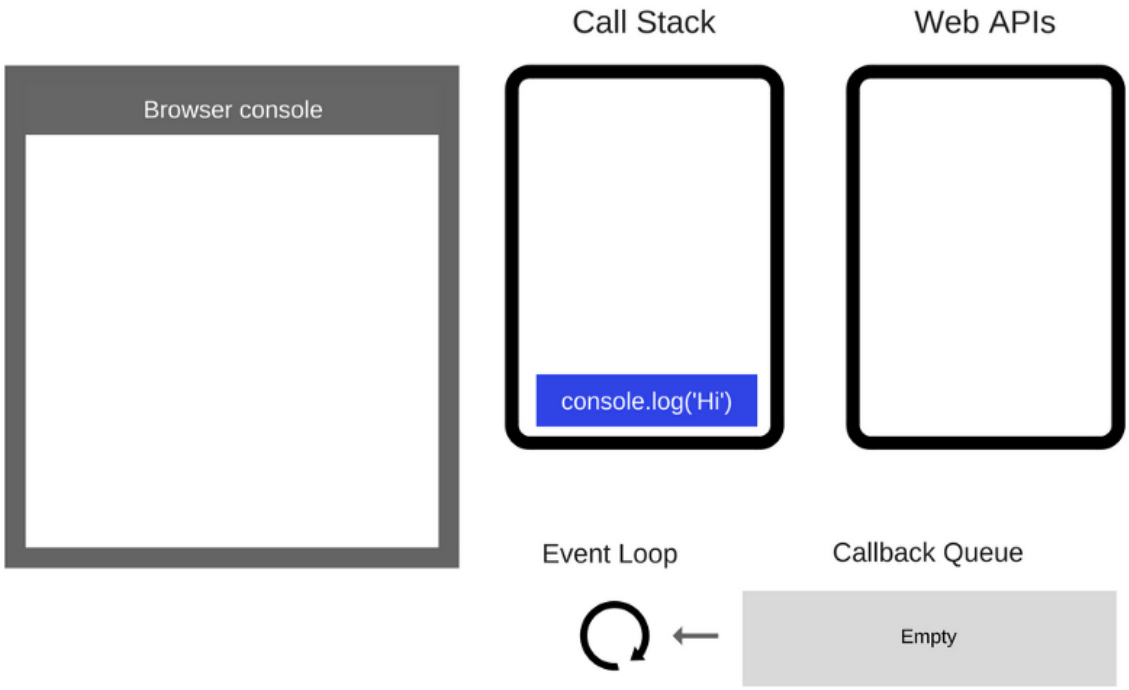
1. Поки що нічого не відбувається. Консоль браузера чиста, стек викликів порожній.

1 / 16



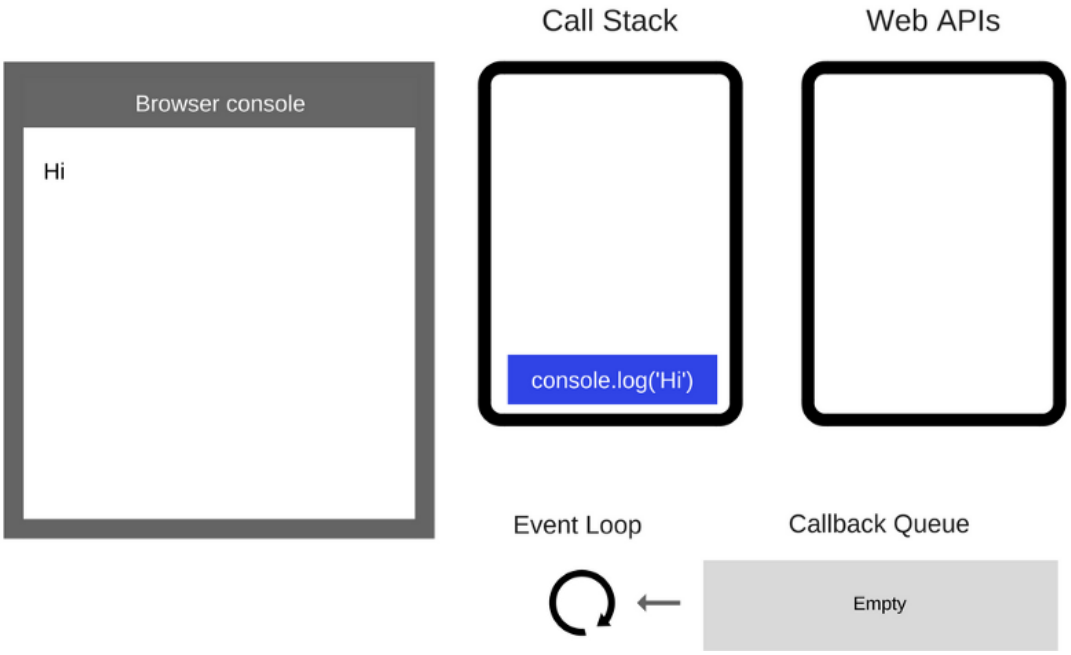
2. Команда console.log('Hi') додається до стека викликів.

2 / 16



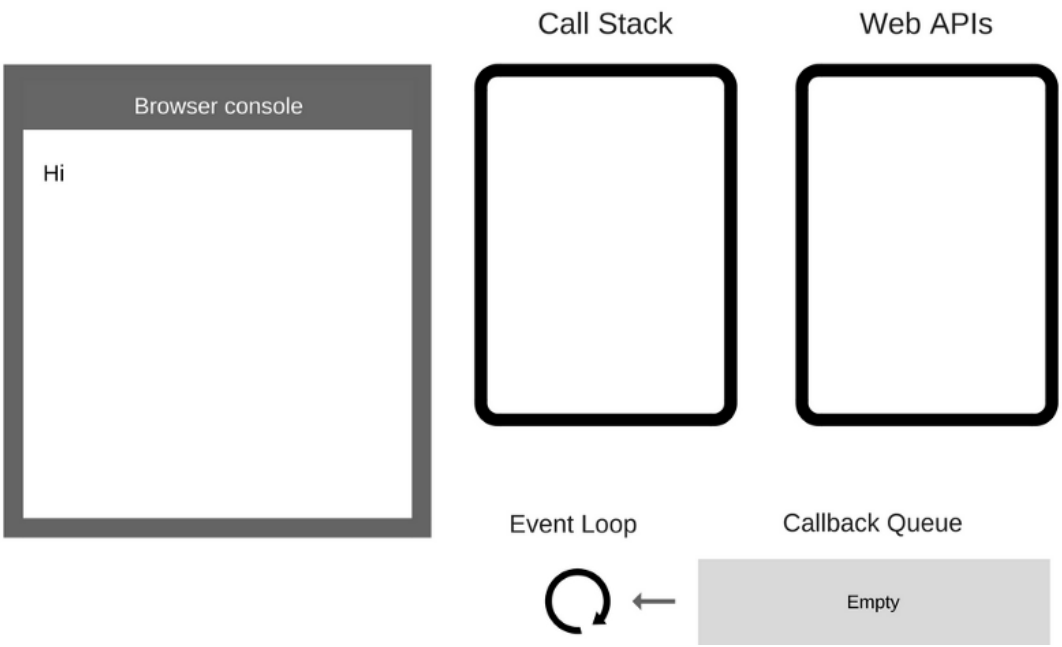
3. Команда console.log('Hi') виконується.

3 / 16



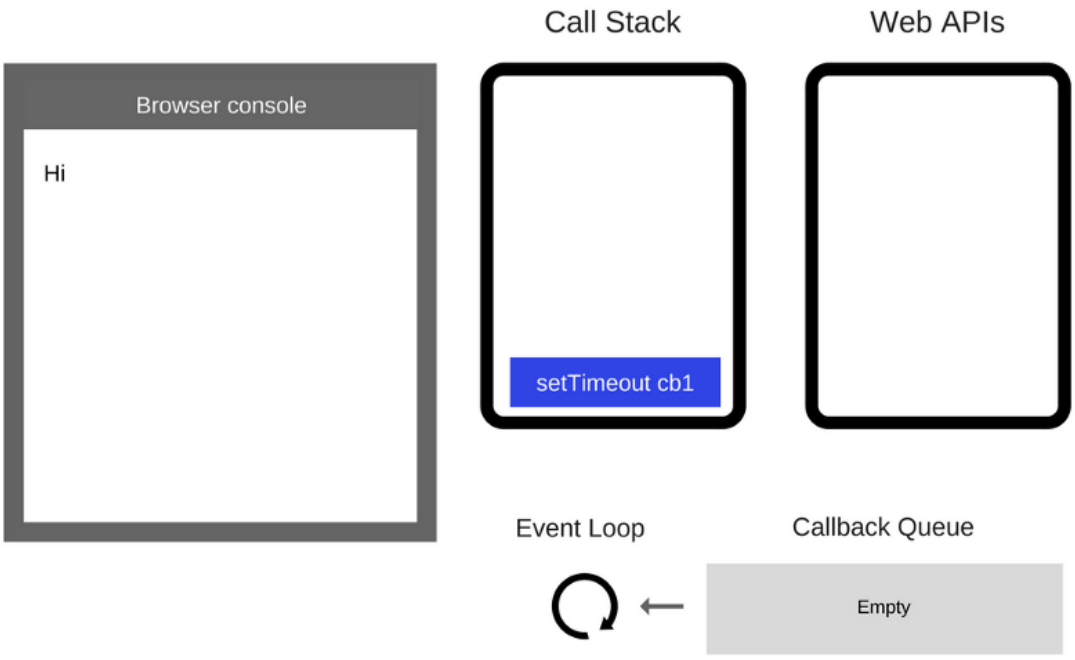
4. Команда console.log('Hi') видаляється зі стека викликів.

4 / 16



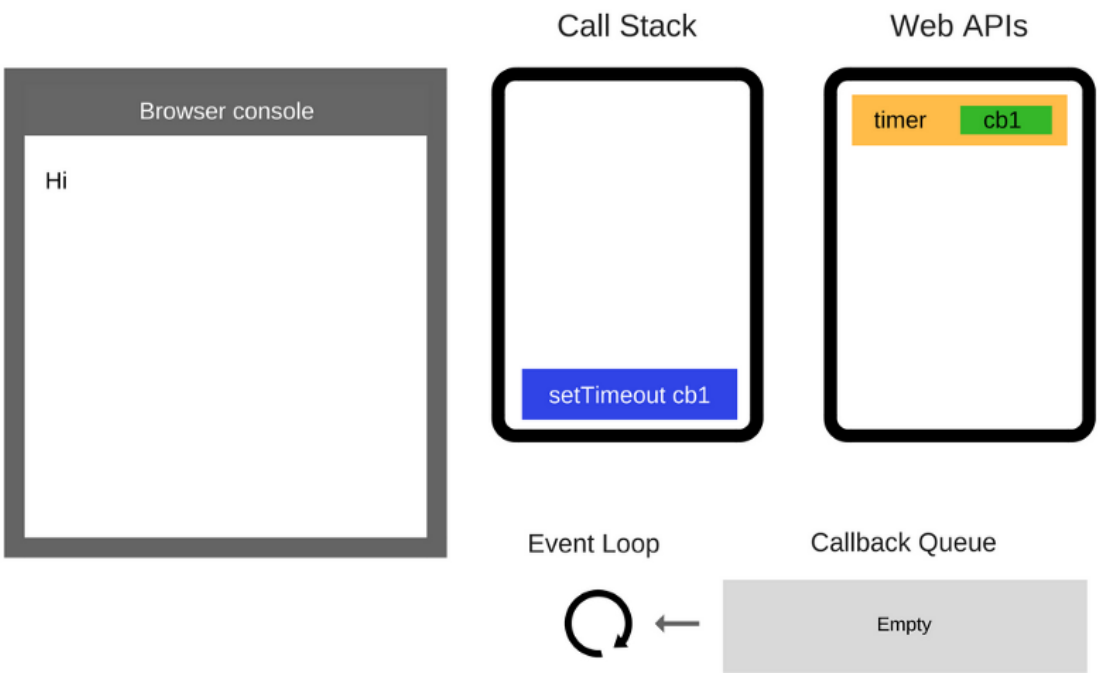
5. Команда setTimeout(function cb1() {...}) додається до стека викликів.

5 / 16



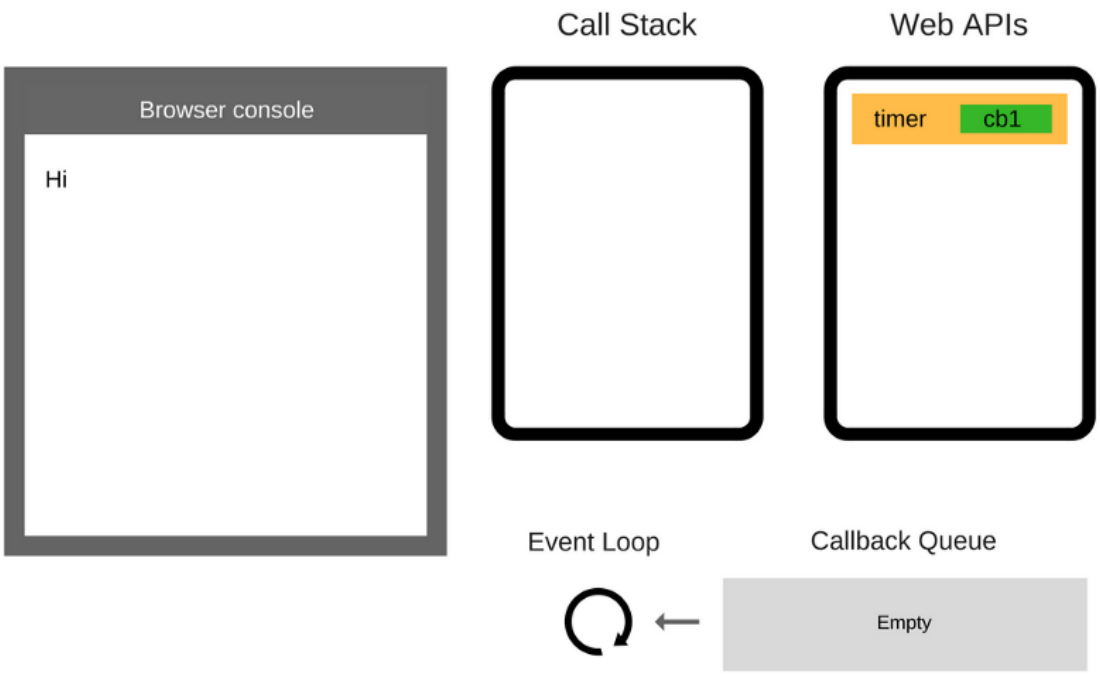
6. Команда setTimeout(function cb1() {...}) виконується. Браузер створює таймер, який є частиною Web API. Він виконуватиме зворотній відлік часу. Тобто оточення реєструє таймер.

6 / 16



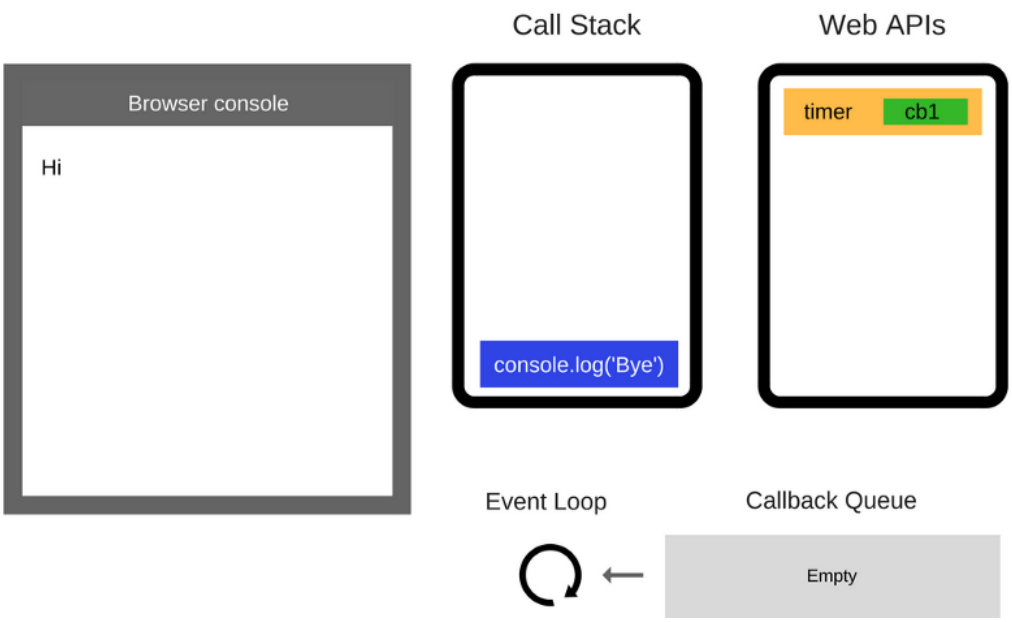
7. Команда setTimeout(function cb1() {...}) завершила роботу та видаляється зі стека викликів.

7 / 16



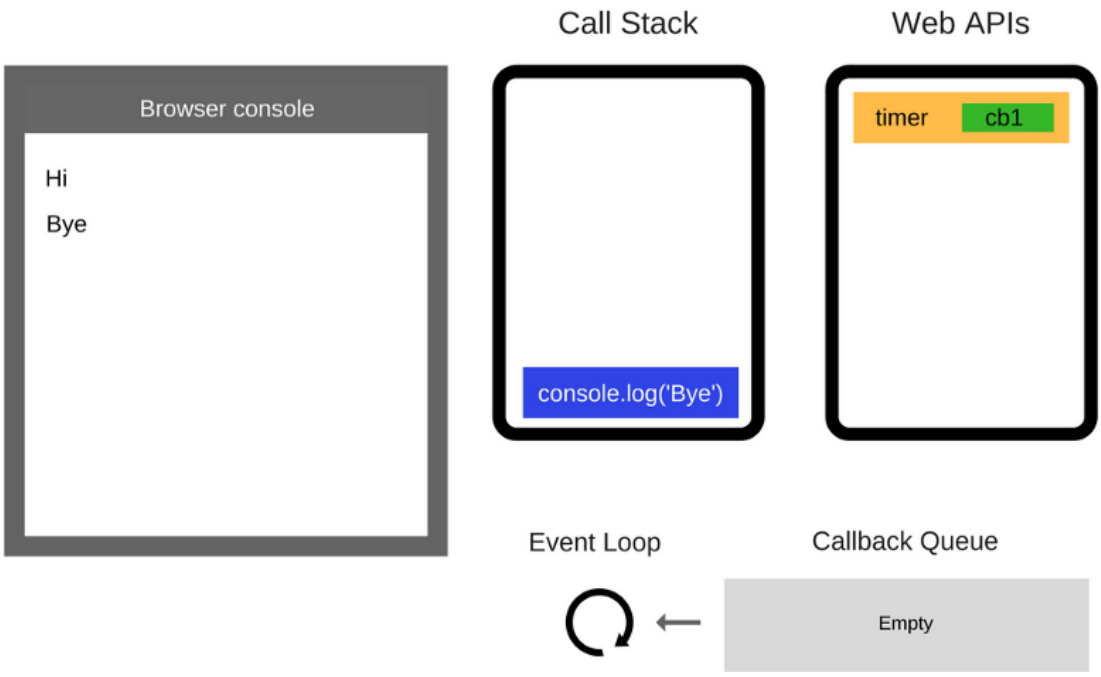
8. Команда console.log('Bye') додається до стека викликів.

8 / 16



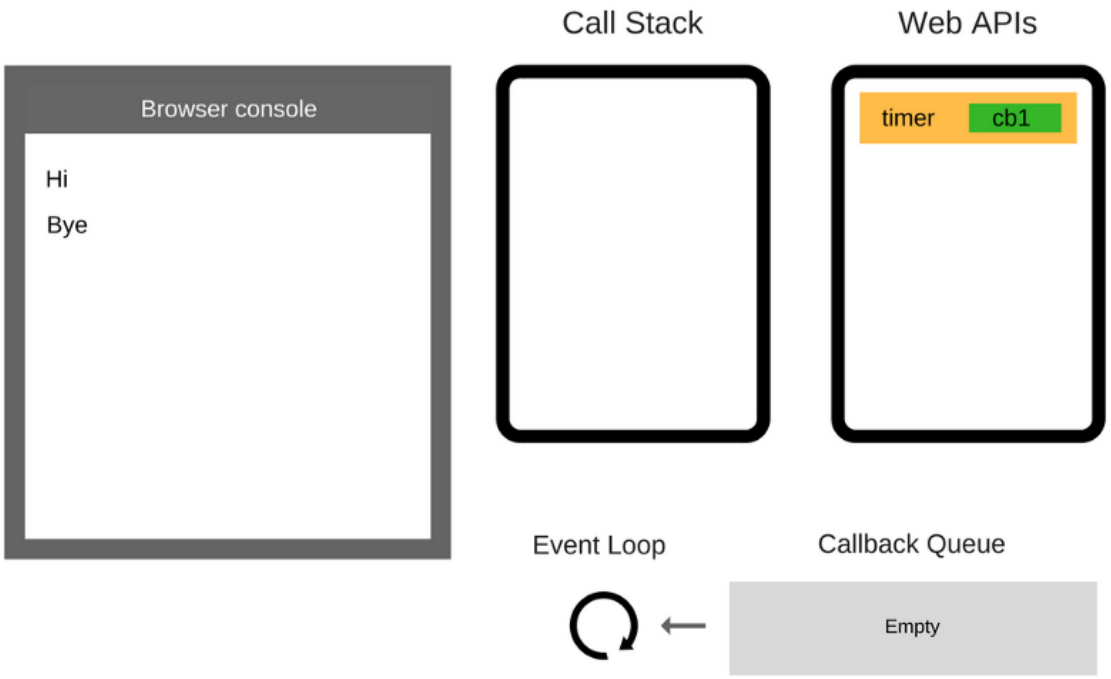
9. Команда console.log('Bye') виконується.

9 / 16



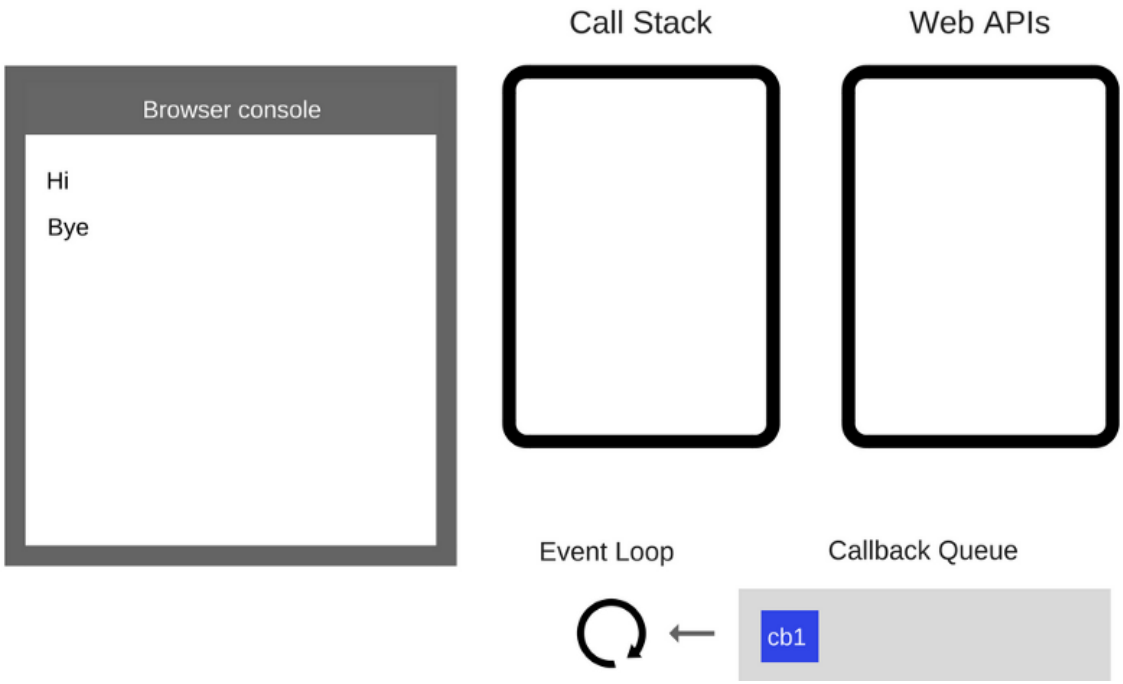
10. Команда console.log('Bye') видаляється зі стека викликів.

10 / 16



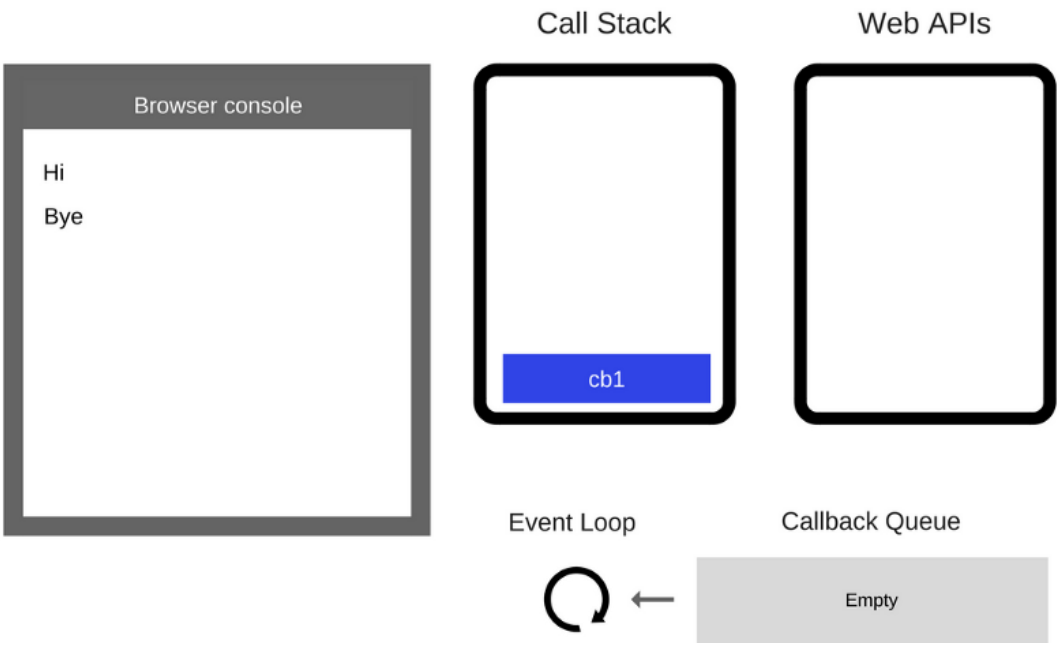
11. Після того, як пройдуть як мінімум 5000 мс., таймер завершує роботу і поміщає коллбек cb1 у чергу коллбеків.

11 / 16



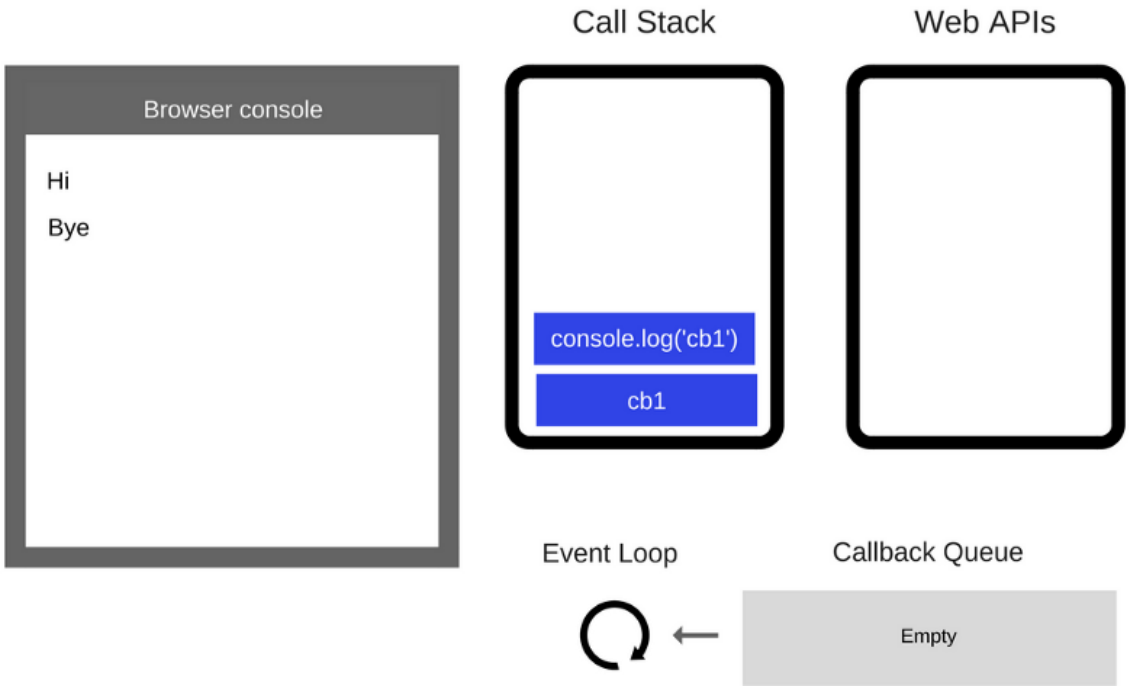
12. Цикл подій бере функцію cb1 з черги коллбеків і поміщає її у стек викликів.

12 / 16



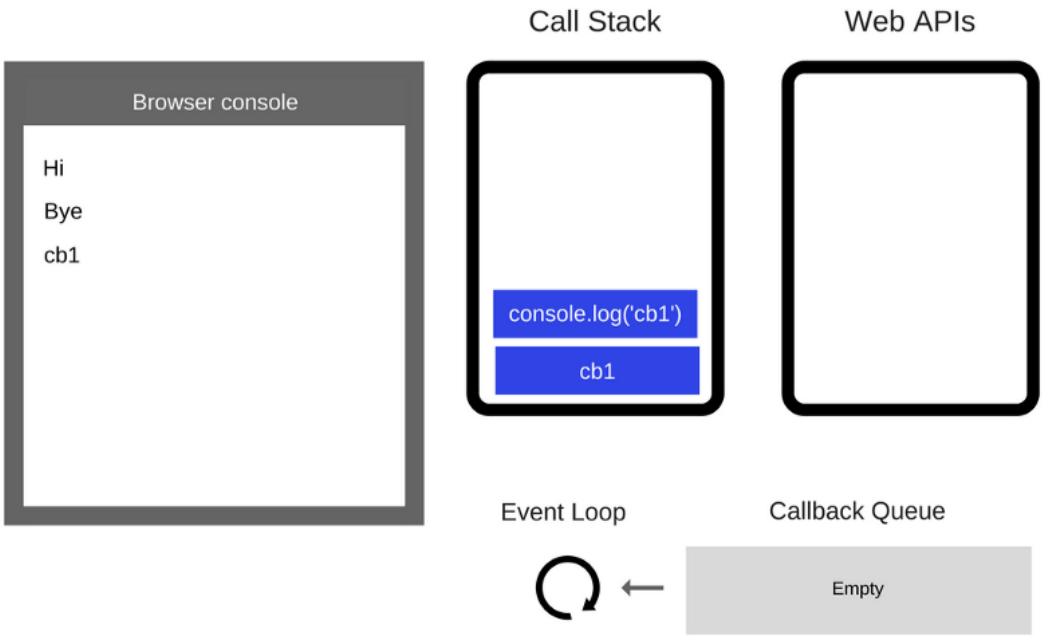
13. Функція cb1 виконується та додає console.log('cb1') у стек викликів.

13 / 16



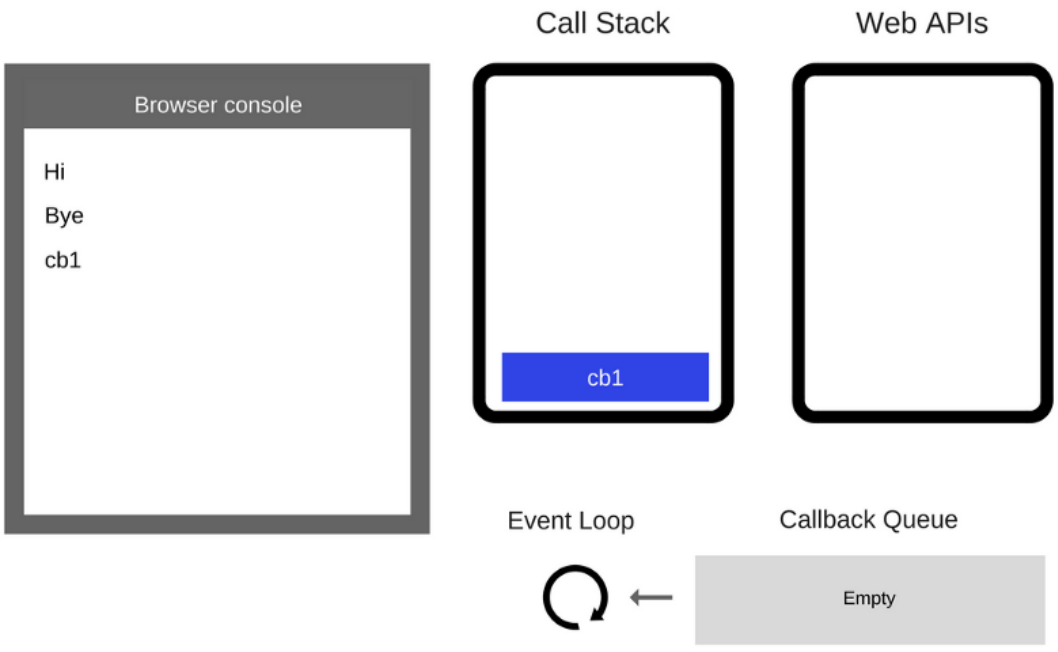
14. Команда console.log('cb1') виконується.

14 / 16



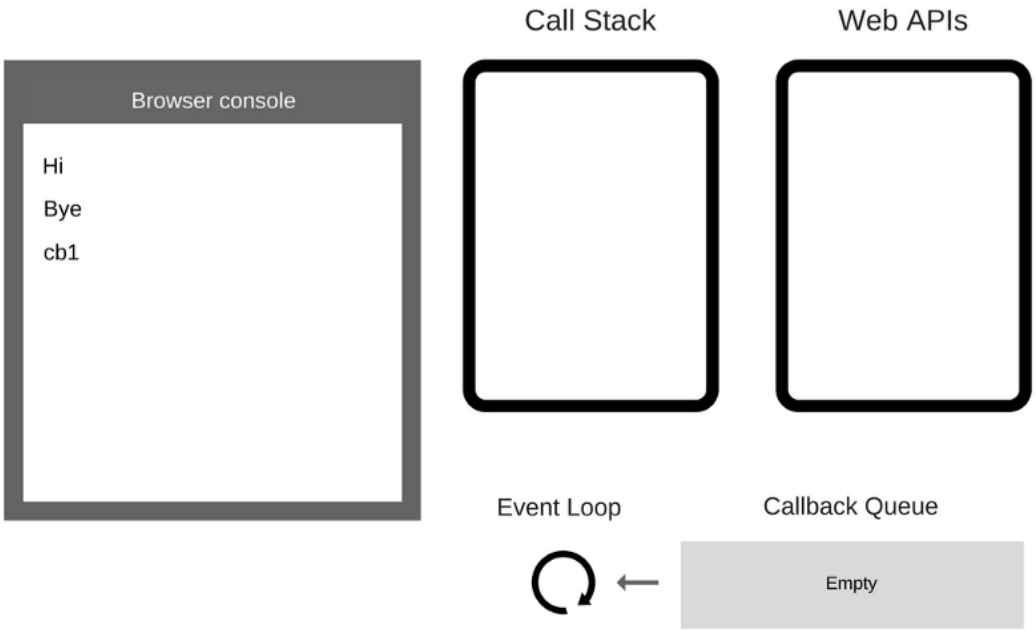
15. Команда console.log('cb1') видаляється зі стека викликів.

15 / 16



16. Функція cb1 видаляється зі стека викликів.

16 / 16



Цікаво помітити, що специфікація ES6 визначає те, як повинен працювати цикл подій, а саме, вказує на те, що технічно він знаходиться в межах відповідальності JS-движка, який починає відігравати важливішу роль в екосистемі JS. Основна причина подібного полягає в тому, що ES6 з'явилися проміси і їм потрібний надійний механізм планування операцій у черзі циклу подій.

Лекція №11. Event Loop. Schedulers.

Теми дипломних робіт:

1) Розробка системи психологічного тестування респондентів для виявлення їхніх особистісних якостей;
2) Створення аукціонного майданчику з продажу мультимедійного контенту (для агентства нерухомості);
3) Створення багатопотокового (асинхронного) парсеру для збору інформації з електронних торгівельних майданчиків;
4) Створення системи для ведення кулінарного блогу з використанням SPA технологій.

Функція setTimeout(…)

Виклик setTimeout(…) не призводить до автоматичного розміщення коллбека у черзі циклу подій. Ця команда запускає таймер. Коли таймер спрацьовує, оточення поміщає коллбек у цикл подій, в результаті, в ході якогось із майбутніх тіків, цей коллбек буде взятий у роботу та виконаний. Розглянемо фрагмент коду:

```
setTimeout(myCallback, 1000);
```

Виконання цієї команди не означає, що myCallback буде виконаний через 1000 мс, правильніше буде сказати, що через 1000 мс. myCallback буде додано до черги макрозадач (макротасків). У черзі, однак, можуть бути інші події, додані туди раніше, в результаті нашому коллбеку доведеться почекати.

Враховуючи все вищесказане, можна сказати те, що виклик setTimeout з другим аргументом, який дорівнює 0, просто відкладає виклик коллбека до моменту очищення стека викликів (Call Stack).

Розглянемо приклад:

```
console.log('Hi');
setTimeout(function() {
  console.log('callback');
  }, 0);
console.log('Bye');
```

Хоча час, на який встановлено таймер, становить 0 мс., в консоль буде виведено наступне:

Hi

Bye

callback

Подієвий цикл: мікрозадачі та макрозадачі

Потік виконання в браузері, як і в Node.js, заснований на подієвому циклі.

Розуміння роботи подієвого циклу важливе для оптимізації, для правильної архітектури, для розуміння роботи планувальників (Schedulers).

Ідея подієвого циклу дуже проста. Є нескінченний цикл, в якому движок JavaScript чекає завдання у Call Stack, виконує їх і знову чекає на появу нових завдань.

Загальний алгоритм движка JS, який відповідає за стек викликів (Call Stack):

1) Поки що є завдання:
    а) виконати їх, починаючи з найстарішого;
2) Не діяти до появи нового завдання, а потім перейти до пункту 1.

Це формалізація того, що ми спостерігаємо, переглядаючи веб-сторінку. JavaScript багато часу нічого не робить і працює, тільки якщо потрібно виконати скрипт/обробник або обробити подію.

Приклади завдань:

- Коли завантажується зовнішній скрипт <script src="..."></script>, то завдання – це виконання цього скрипту.
- Коли користувач рухає мишу, завдання полягає в тому, щоб згенерувати подію `mousemove` і виконувати її обробники.
- Коли таймер, встановлений за допомогою setTimeout(func, ...), закінчиться, завдання – це виконання функції колбеку func.
- І так далі.

Завдання надходять на виконання - движок виконує їх - потім чекає на нові завдання (під час очікування практично не навантажуючи процесор комп'ютера).

Може так статися, що завдання надходить, коли движок зайнятий чимось іншим, тоді воно ставиться у чергу.

Чергу, яку формують такі завдання, називають чергою макрозадач (macrotask queue, термін v8).

Наприклад, коли движок зайнятий виконанням скрипта, користувач може пересунути мишу, тим самим викликавши появу події `mousemove`, або може закінчитися

таймер, встановлений setTimeout, і т.п. Ці завдання формують чергу макрозадач, як показано на ілюстрації.

Завдання із черги виконуються за правилом «першим прийшов – першим пішов». Коли браузер закінчує виконання скрипта, він обробляє подію mousemove, потім виконує обробник, заданий setTimeout, і так далі.

Зазначимо дві деталі:

1) Рендеринг (малювання сторінки) ніколи не відбувається під час виконання завдання движком. Не має значення, наскільки довго виконується завдання. Зміни DOM відмальовуються тільки після того, як завдання виконано.
2) Якщо завдання виконується дуже довго, то браузер не може виконувати інші завдання, обробляти події користувача, тому через деякий час браузер пропонує «вбити» завдання, що довго виконується. Таке можливо, коли у скрипті багато складних обчислень чи помилка, що веде до нескінченного циклу.

Макрозадачі та Мікрозадачі

Крім макрозадач існують мікрозадачі. Мікрозадачі надходять лише з коду. Зазвичай вони створюються промісами: виконання оброблювача .then/catch/finally це мікрозадачі. Мікрозадачі також використовуються «під капотом» await, так як це форма обробки промісу.

Також є спеціальна функція queueMicrotask(func), яка поміщає func у чергу мікрозадач.

Відразу після кожної макрозадачі движок виконує всі задачі з черги мікрозадач перед тим, як виконати наступну макрозадачу або відобразити зміни на сторінці (rerender), або зробити щось ще.

Наприклад:

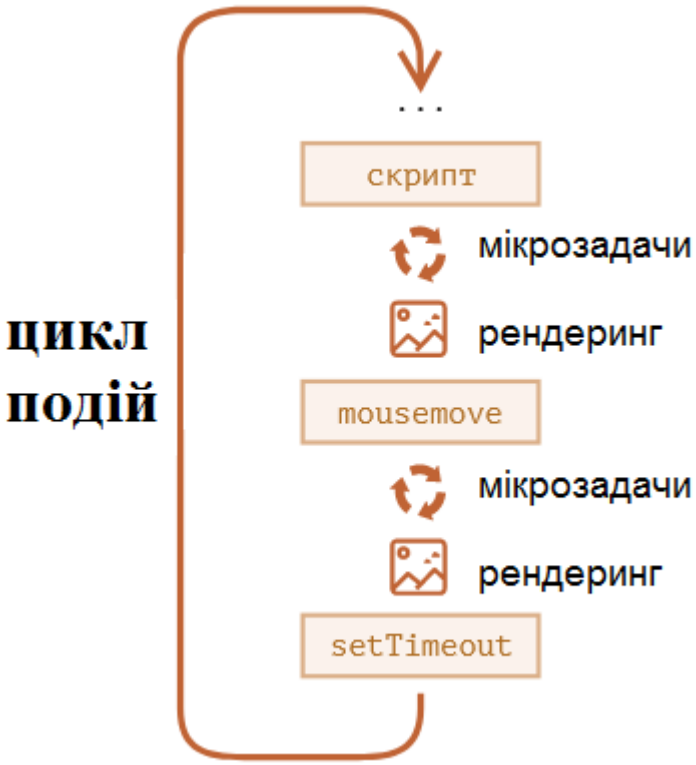
```
setTimeout(() => alert("timeout"));
```

```
Promise.resolve()
    .then(() => alert("promise"));
alert("code");
```

Який тут буде порядок?

1) code з’явиться першим, т.я. це звичайний синхронний виклик.
2) promise з'явиться другим, тому що .then проходить через чергу мікрозадач та виконується після поточного синхронного коду.
3) timeout з'явиться останнім, оскільки це макрозадача.

Докладніше зображення циклу подій виглядає так:



Усі мікрозадачи завершуються до обробки будь-яких подій або рендерингу, або переходу до іншої макрозадачі.

Це важливо, оскільки гарантує, що загальне оточення залишається одним і тим самим між мікрозадачами – не змінено координати миші, не отримано нових даних по мережі, тощо.

Якщо ми хочемо запустити функцію асинхронно (після поточного коду), але до відображення змін і нових подій, то можемо запланувати це через queueMicrotask.

Висновок

Більш докладний алгоритм подієвого циклу (хоч і спрощений порівняно зі специфікацією):

1) Вибрати та виконати найстаріше завдання із черги макрозадач (наприклад, «script»).
2) Виконати всі мікрозавдання:
    - Поки черга мікрозадач не порожня: - Вибрати з черги і виконати найстарішу мікрозадачу;
3) Відобразити зміни сторінки, якщо вони є (рендеринг).
4) Якщо черга макрозадач порожня – почекати, доки з'явиться макрозадача;
5) Перейти до кроку 1.

Щоб додати до черги нову макрозадачу:

- Використовуйте setTimeout(f) із нульовою затримкою.

Цей спосіб можна використовувати для розбиття великих обчислювальних завдань на частини, щоб браузер міг реагувати на події користувача і показувати прогрес виконання цих частин.

Також це використовується в обробниках подій для відкладеного виконання дії після того, як подія повністю оброблена (спливання завершено).

Для додавання нової мікрозадачі в чергу:

- Використовуйте queueMicrotask(f).
- Використовуйте проміси. Обробники промісів виконуються в рамках черги мікрозадач.

Події інтерфейсу користувача та мережні події в проміжках між мікрозадачами не обробляються: мікрозадачі виконуються безперервно одна за одною.

Тому queueMicrotask можна використовувати для асинхронного виконання функції у тому ж стані оточення.

Як працює Event Loop з мікротасками та макротасками можна подивитись тут:

https://www.jsv9000.app/

http://latentflip.com/loupe/?code=JC5vbignYnV0dG9uJywgJ2NsaWNrJywgZnVuY3Rpb24gb25DbGljaygpIHsKICAgIHNldFRpbWVvdXQoZnVuY3Rpb24gdGltZXIoKSB7CiAgICAgICAgY29uc29sZS5sb2coJ1lvdSBjbGlja2VkIHRoZSBidXR0b24hJyk7ICAgIAogICAgfSwgMjAwMCk7Cn0pOwoKY29uc29sZS5sb2coIkhpISIpOwoKc2V0VGltZW91dChmdW5jdGlvbiB0aW1lb3V0KCkgewogICAgY29uc29sZS5sb2coIkNsaWNrIHRoZSBidXR0b24hIik7Cn0sIDUwMDApOwoKY29uc29sZS5sb2coIldlbGNvbWUgdG8gbG91cGUuIik7!!!PGJ1dHRvbj5DbGljayBtZSE8L2J1dHRvbj4%3D

RxJS планувальники (Schedulers)

Згадаємо про порядок виконання коду Event Loop:

1) Черга синхронного коду (call stack)
2) Черга мікрозадач (Promise)
3) Черга макрозадач (setTimeout() та setInterval() або AJAX-запити
4) Окрема черга браузера для відображення вмісту (requestAnimationFrame)

Планувальники (Schedulers) контролюють коли буде здійснено підписку або коли буде доставлено нове значення підписнику. Планувальники складаються з трьох основ:

- Планувальник це структура даних, він знає як зберігати або чередувати задачі на підставі пріоритетів або іншого критерію.
- Планувальник це контекст виконання, він вирішує коли і де буде виконана та чи інша задача (наприклад в setTimeout, process.nextTick або в animation frame).
- У планувальника є годинник (віртуальний), він надає дані про "час" через метод now().

Планувальник визначає у якому контексті виконання Observable буде надсилати дані підписнику.

Іншими словами, Scheduler управляє черговістю та часом виконання операцій у Observable. Розглянемо приклад:

```
import { of } від "rxjs";
console.log("Start");
of("Observable").subscribe(console.log);
console.log("End");
// Logs:
// Start
// Observable
// End
```

Перед нами два console.log та Observable між ними. Код виконується синхронно. Якщо ми хочемо, щоб наш Observable виконувався асинхронно, потрібно додати оператор observeOn і всередину нього прокинути потрібний Scheduler.

```
import {asyncScheduler, of} від "rxjs";
import {observeOn} від "rxjs/operators";

console.log("Start");
```

```
of("Observable")
    .pipe(observeOn(asyncScheduler))
    .subscribe(console.log);
console.log("End");

// Logs:
// Start
// End
// Observable
```

Тепер Observable надає дані асинхронно. Це схоже, якби ми обернули його в setTimeout(…, 0), тобто з нульовою затримкою.

В даному коді використовується asyncScheduler. Це один із Schedulers. Але їх набагато більше.

У прикладі нижче, ми створюємо звичайний Observable, який відправляє значення 1, 2, 3 синхронно і використовуючи оператор observeOn з планувальником async:

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

const observable = new Observable((observer) => {
  observer.next(1);
  observer.next(2);
  observer.next(3);
  observer.complete();
}).pipe(
  observeOn(asyncScheduler)
);

console.log('just before subscribe');
observable.subscribe({
  next(x) {
    console.log('got value ' + x)
  },
  error(err) {
```

```
    console.error('something wrong occurred: ' + err);
  },
  complete() {
    console.log('done');
  }
});
console.log('just after subscribe');
```

Після використання, отримаємо:

```
just before subscribe
just after subscribe
got value 1
got value 2
got value 3
done
```

Треба зауважити, що повідомлення got value ... були отримані після повідомлення just after subscribe, це трохи відрізняється від стандартної синхронної поведінки. Все через оператор observeOn, який реалізує між вихідним Observable і підписником, ще один підписник. Перейменуємо назви деяких змінних, щоб стало зрозуміліше:

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

var observable = new Observable((proxyObserver) => {
  proxyObserver.next(1);
  proxyObserver.next(2);
  proxyObserver.next(3);
  proxyObserver.complete();
}).pipe(
  observeOn(asyncScheduler)
);

var finalObserver = {
  next(x) {
    console.log('got value ' + x)
```

```
  },
  error(err) {
    console.error('something wrong occurred: ' + err);
  },
  complete() {
    console.log('done');
  }
};

console.log('just before subscribe');
observable.subscribe(finalObserver);
console.log('just after subscribe');
```

Після виконання, отримаємо (результат такий же):

```
just before subscribe
just after subscribe
got value 1
got value 2
got value 3
done
```

Планувальник async працює на основі функцій setTimeout або setInterval. Часто в Javascript для того щоб відкласти якусь операцію на наступний такт циклу подій використовують таку конструкцію як setTimeout(fn, 0). Завдяки цьому, значення got value 1 було отримано finalObserver після того як вивелося повідомлення just after subscribe.

Типи Schedulers

Для розуміння роботи з Schedulers потрібно знати, як працює Event Loop JavaScript. Якщо коротко, то у браузері свій порядок виконання коду:

1) Спочатку виконується синхронний код (callstack)
2) Далі черга мікрозадач (Promise)
3) Потім черга макрозадач (setTimeout, setInterval, XMLHttpRequest і т.д.).
4) Окремо стоїть черга для задач, які виконуються одразу перед наступним циклом перемальовування контенту (requestAnimationFrame).

У RxJS є Scheduler на кожен із цих пунктів:

queueScheduler	планування синхронного коду
asapScheduler	планування коду в чергу мікрозадач
asyncScheduler	планування коду в чергу макрозадач
animationFrameScheduler	планування коду в чергу перед перемальовуванням контенту

Ще існують VirtualTimeScheduler та TestScheduler, які використовуються для тестів.

Розглянемо приклад:

```
import { of, merge, asapScheduler, asyncScheduler, queueScheduler, animationFrameScheduler } from "rxjs";
import {observeOn} від "rxjs/operators";

const async$ = of("asyncScheduler").pipe(observeOn(asyncScheduler));
const asap$ = of("asapScheduler").pipe(observeOn(asapScheduler));
const queue$ = of("queueScheduler").pipe(observeOn(queueScheduler));
const animationFrame$ = of("animationFrameScheduler").pipe(
observeOn(animationFrameScheduler)
);
merge(async$, asap$, queue$, animationFrame$).subscribe(console.log);

console.log("synchronous code");

// Logs:
// queueScheduler
// synchronous code
// asapScheduler
// animationFrameScheduler
// asyncScheduler
```

Як можно побачити, "queueScheduler" відпрацював синхронно, тому що він перед "synchronous code". А "asapScheduler" раніше "asyncScheduler", тому що для подій цього потоку використовується черга мікрозадач.

Як використовувати Schedulers

Як ми вже бачили, Scheduler використовується з операторами observeOn і subscribeOn. Обидва приймають у собі першим аргументом Scheduler, а другим аргументом delay, який за умовчанням дорівнює нулю.

```
import { of, asyncScheduler } від "rxjs";
import {observeOn, subscribeOn} від "rxjs/operators";

of("observeOn")
.pipe(observeOn(asyncScheduler, 1000))
.subscribe(console.log);

of("subscribeOn")
.pipe(subscribeOn(asyncScheduler, 500))
.subscribe(console.log);

// Logs:
// subscribeOn
// observeOn
```

Відмінність їх у тому, що observeOn планує у якому контексті виконуватимуться методи observer — next, error і complete будуть виконуватися у відповідному до Scheduler контексті. А subscribeOn впливає на subscriber - метод subscribe буде виконуватись в відповідному до Scheduler контексті.

Цікавий факт, що якщо задати delay не рівний нулю в observeOn/subscribeOn, то незалежно від того, який використовується Scheduler, буде використовуватися asyncScheduler. Нема сенсу у наступному коді - observeOn(animationFrameScheduler, 100).

До версії RxJS 6.5.0 можна було додати Scheduler другим аргументом для of, from, merge, range і т.д. У нових версіях RxJS це поведінка застаріла (deprecated), і необхідно використовувати функцію scheduled для цього.

```
import { of, scheduled, asapScheduler} from 'rxjs';

// DEPRECATED
// of(2, asapScheduler).subscribe(console.log);
```

```
scheduled(of('scheduled'), asapScheduler).subscribe(console.log);
```

Приклад використання Scheduler

Ми не замислюємося про Schedulers під час роботи з RxJS, оскільки автори бібліотеки провели чудову роботу з абстрагування цієї логіки. Але буває, коли використання Scheduler буде органічним. Наприклад, необхідно реалізувати кешування запитів, що відрізняються за ідентифікатором. Це можна зробити наступним чином:

```
const cache = new Map<number, any>();
function get(id: number): Observable<any> {
 if (cache.has(id)) {
  return of(cache.get(id));
 }
 return http.get('some-url\' + id).pipe(
  tap(data => {
   cache.set(id, data);
  }),
 );
}
```

У прикладі при першому використанні функції, ми надсилаємо запит до backend-у і кешуємо id. Вдруге, коли звертаємось до функції, то створюємо потік і повертаємо закешоване значення id.

Код працює. Але є один нюанс. Вперше, коли використовується функція, вона повертає значення асинхронно, а вдруге – синхронно. Коли функція поводиться так непередбачувано, це неправильно.

Зазвичай це вважається поганою практикою, оскільки:

- Це може призвести до несподіваних умов перегонів.
- Це змінює те, у який стек викликів кидаються винятки.
- Загалом це робить функцію менш легкою для розуміння та використання.

Але, якщо додати scheduled з asyncScheduler у 4 рядок, то ситуація виправляється і код працює передбачувано.

```
return scheduled(of(cache.get(id)), asyncScheduler);
```

Тепер все працює передбачувано.

При роботі з RxJS часто використовуються планувальники, тому що будь-який оператор використовує якийсь планувальник за замовчуванням, якщо ви явно не вказали планувальник, RxJS вибере його за вас намагаючись використовувати такий планувальник який приведе виконання Observable максимально до синхронного вигляду. Наприклад для Observable, які відправляють кінцеву або малу кількість значень не використовується планувальник, для Observable, які відправляють потенційно велику кількість або нескінченну кількість значень використовується планувальник queue, для операторів працюючих з часом використовується async.

Можна вибрати і інший планувальник, для підвищення продуктивності програми, використовуючи оператори які приймають як аргумент тип планувальника, наприклад: from([10, 20, 30], asyncScheduler).

Нижче наведено список операторів створення, в аргументах яких можна передати планувальник:

- `bindCallback`
- `bindNodeCallback`
- `combineLatest`
- `concat`
- `empty`
- `from`
- `fromPromise`
- `interval`
- `merge`
- `of`
- `range`
- `throw`
- `timer`

Висновок

1) Використовуйте subscribeOn коли потрібно відкласти виконання методу subscribe.
2) Використовуйте observeOn коли потрібно відкласти виконання наступного оператора.
3) Оператори, що працюють з часом, такі як bufferTime, debounceTime, delay, auditTime, sampleTime, throttleTime, timeInterval, timeout, timeoutWith,

windowTime, в якості останнього аргументу приймають планувальник. За замовчуванням використовуються asyncScheduler.

4) Schedulers впливають на час та порядок виконання завдань. Чверть операторів RxJS використовує під капотом планувальник. З великою ймовірністю поінформованість про них не потрібна. Але ніколи не знаєш, коли вони можуть стати у нагоді.

Оператори в RxJS

Оператори в RxJS — це функції, які дозволяють здійснювати різноманітні операції з потоками даних, такі як фільтрування, перетворення, злиття та багато інших. Використання операторів дозволяє зменшити кількість коду та спростити роботу з потоками даних.

Ось декілька причин, навіщо варто використовувати оператори в RxJS:

1) Зручне та ефективне управління потоками даних. Оператори дозволяють здійснювати широкий спектр операцій з потоками даних, що дозволяє ефективно управляти цими даними, зокрема зменшувати їх кількість та робити розумні трансформації.
2) Краща читабельність коду. Використання операторів дозволяє зменшити кількість коду та зробити його більш зрозумілим. Наприклад, замість декількох вкладених циклів можна скористатися оператором “mergeMap” (колишній flatMap), щоб спростити код та підвищити його читабельність.
3) Висока перевикористовуваність. Оператори дозволяють створювати багаторазовий код, який можна використовувати в різних проектах та ситуаціях.
4) Підтримка асинхронного програмування. Оператори в RxJS дозволяють працювати з асинхронними операціями та подіями, що дозволяє зберігати консистентність даних в проекті.
5) Розширення функціональності. Оператори в RxJS можуть розширити функціональність вашого коду та дозволити зробити складні операції більш простими. Наприклад, оператор “reduce” може зменшити потік даних до одного значення, що дозволяє виконувати складні операції з потоками даних, такі як обчислення середнього значення або знаходження максимального значення.

Оператори в RxJS можуть також допомогти зробити ваш код більш безпечним та масштабовним. Наприклад, оператор “catchError” дозволяє перехоплювати та обробляти

помилки в потоках даних, що допомагає зберегти консистентність даних та забезпечити більш безпечну роботу з потоками.

В цілому, використання операторів в RxJS дозволяє здійснювати більш ефективну та гнучку роботу з потоками даних в JavaScript. Це може збільшити продуктивність нашого коду та зробити його більш зрозумілим та безпечним для розробників.

RxJS надає безліч операторів, але лише деякі з них використовуються часто.

Область	**Оператори**
Створення	from, fromEvent, of
Комбінування	combineLatest, concat, merge, startWith , withLatestFrom, zip
Фільтрація	debounceTime, distinctUntilChanged, filter, take, takeUntil
Трансформація	bufferTime, concatMap, map, mergeMap, scan, switchMap
Утиліта	tap
Мультикастинг	share

Одним зі способів досить легко зрозуміти оператори в RxJS є порівняння їх з методами масивів в нативному JavaScript. Багато з операторів в RxJS мають схожі функції з методами масивів, які зазвичай використовуються для маніпулювання масивами даних.

Ось декілька прикладів схожості між операторами в RxJS та методами масивів:

1) Map: як і метод масиву, оператор “map” в RxJS дозволяє перетворити кожен елемент потоку даних в нове значення. Наприклад, метод масиву “map” може використовуватися для створення нового масиву, в якому кожен елемент масиву був би перетворений згідно з певною функцією. Аналогічно, оператор “map” може використовуватися для створення нового потоку даних, в якому кожен елемент був би перетворений відповідно до певної функції.

Приклад: Визначення довжини рядка для кожного елемента масиву рядків:

```
const strings = ['apple', 'banana', 'orange', 'mango'];
const lengths = strings.map(function(str) {
  return str.length;
});
console.log(lengths); // [5, 6, 6, 5]
```

У цьому прикладі ми використовуємо метод масиву map(), щоб створити новий масив з тих же кількостей елементів, що й вхідний масив. Кожен елемент нового масиву формується шляхом застосування функції-аргумента до відповідного елемента вхідного масиву.

А ось приклад використання оператора map() в RxJs. Визначення довжини рядка для кожного елемента масиву рядків:

```
import { of } from 'rxjs';
import { map } from 'rxjs/operators';
const strings = of('apple', 'banana', 'orange', 'mango');
const lengths = strings.pipe(map(str => str.length));
lengths.subscribe(console.log); // 5, 6, 6, 5
```

В цьому прикладі ми використовуємо функцію pipe() для застосування оператора map() до нашого вхідного потоку. Кожен елемент вхідного потоку проходить через оператор map(), де до нього застосовується функція-аргумент, а результат стає елементом вихідного потоку.

Крім того, у наших прикладах ми використовуємо оператори of() та subscribe(), які допомагають створити вхідний потік та підписатися на вихідний потік відповідно. Оператор of() створює потік, що відправляє задані значення, а оператор subscribe() підписується на вихідний потік та викликає зазначену функцію-аргумент для кожного елемента, який проходить через потік.

2. Filter: як і метод масиву, оператор “filter” в RxJS дозволяє відфільтрувати потік даних, щоб включити тільки ті значення, які задовольняють певні умови. Наприклад, метод масиву “filter” може використовуватися для створення нового масиву, в якому відфільтровані елементи масиву відповідають заданій умові. Аналогічно, оператор “filter” може використовуватися для створення нового потоку даних, в якому включені тільки ті значення, які відповідають заданій умові.

Ось приклад використання методу filter() в JavaScript:

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const filteredNumbers = numbers.filter((number) => number % 2 === 0);
console.log(filteredNumbers); // [2, 4, 6, 8, 10]
```

У цьому прикладі filter() фільтрує масив numbers і повертає новий масив filteredNumbers, який містить тільки парні числа.

А ось приклад використання оператора filter() для здійснення такої ж операції (фільтрації парних чисел) в RxJs:

```
import { from } from 'rxjs';
import { filter } from 'rxjs/operators';
const numbers = from([1, 2, 3, 4, 5]);
const filteredNumbers = numbers.pipe(filter(number => number % 2 === 0));
filteredNumbers.subscribe(x => console.log(x)); // 2, 4
```

У цьому прикладі ми створюємо потік numbers, який відправляє послідовність чисел від 1 до 5. Ми застосовуємо до потоку оператор filter(), який фільтрує тільки парні числа та передає їх далі до підписника через subscribe(). Це призводить до виводу чисел 2 та 4 у консоль.

Отже, метод filter() в JavaScript та оператор filter() в RxJs дозволяють фільтрувати дані з масивів та потоків подій відповідно.

3. Оператор reduce в бібліотеці RxJS та метод reduce в масиві мають схожість у тому, що обидва вони зменшують список значень до одного значення. Однак, вони виконують цю операцію з дещо різним підходом.

Метод reduce масиву приймає функцію з двома аргументами: акумулятор та поточне значення масиву. Функція виконується для кожного елемента масиву, і результат зберігається в акумуляторі. По закінченню перебору всіх елементів масиву, результат є кінцевим результатом.

Оператор reduce в RxJS працює з потоками даних і дозволяє зменшити потік до одного значення. Він приймає функцію з двома аргументами: аккумулятор та поточний елемент потоку даних. Функція виконується для кожного елемента потоку, і результат зберігається в аккумуляторі. По закінченню перебору всіх елементів потоку, результат є кінцевим результатом.

Хоча обидва підходи зменшують список до одного значення, метод reduce масиву зупиняється після обробки всіх елементів масиву та повертає кінцевий результат. У випадку з оператором reduce в RxJS потік даних може продовжувати надходити після того, як був знайдений кінцевий результат. Тому результат reduce в RxJS може бути збережений відразу ж після того, як було оброблено всі елементи потоку, або при надходженні певної події в потоці.

Інша різниця між методом reduce масиву та оператором reduce в RxJS полягає в тому, що метод reduce масиву доступний безпосередньо на об'єкті масиву, тоді як оператор reduce в RxJS має бути імпортований з бібліотеки RxJS.

Приклад використання методу reduce() в JavaScript:

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => accumulator + currentValue);
console.log(sum); // 15
```

У цьому прикладі ми використовуємо метод reduce() для зведення масиву numbers до одного значення - суми всіх елементів масиву. Перший аргумент методу reduce() є функцією зведення, яка приймає два аргументи: accumulator та currentValue. Вона обчислює нове значення аккумулятора, використовуючи поточне значення масиву та попередній результат зведення. Після завершення пыдрахунку метод reduce() повертає остаточне значення аккумулятора.

Тепер давайте подивимося на приклад використання оператора reduce() в RxJs:

```
import { from } from 'rxjs';
import { reduce } from 'rxjs/operators';
const numbers = from([1, 2, 3, 4, 5]);
const sum = numbers.pipe(reduce((accumulator, currentValue) => accumulator + currentValue));
sum.subscribe(x => console.log(x)); // 15
```

У цьому прикладі ми використовуємо оператор reduce() для зведення потоку numbers до одного значення - суми всіх елементів потоку. Перший аргумент оператора reduce() є функцією зведення, яка працює так само, як і в методі reduce(). Після завершення пыдрахунку оператор reduce() повертає новий потік, який містить остаточне значення аккумулятора.

Одна з основних відмінностей між методом reduce() в JavaScript та оператором reduce() в RxJs полягає в тому, що метод reduce() в JavaScript приймає тільки два аргументи — функцію звуження (**reducer function**) та початкове значення (**initial value**), тоді як оператор reduce() в RxJs може бути використаний для зменшення потоку даних (**stream**) за допомогою функції звуження, а також для комбінування потоків даних з використанням різних стратегій злиття (**merge strategies**). Крім того, оператор reduce() в

RxJs може працювати з асинхронними операціями та дозволяє використовувати функції звуження, які повертають **Observables**, що дозволяє створювати комплексні потоки даних.

Функції of, from, fromEvent.

Ці оператори відносяться до творчих і викликаються як самостійні функції створення нових Observable-объектів з переданих аргументів.

Наступний приклад демонструє створення Observable-об'єкта зі списку за допомогою функції from с наступною підпискою та відпискою.

```
import { from } from 'rxjs';
const observable = from([10, 20, 30]);
const subscription = observable.subscribe(x => console.log(x));
subscription.unsubscribe()
```

На відміну від функції of(), функція from() приймає аргументи різних видів (проміси, списки тощо), перетворюючи їх (якщо необхідно) на об'єкти Observable, на які можна підписатись. Функція of() приймає значення та повертає їх потоком без перетворення. Ще одна важлива відмінність у тому, як ці аргументи обробляються.

```
of([1,2,3]).subscribe( x => {
console.log(x);
});
from([1,2,3]).subscribe( x => {
console.log(x);
})
```

У наведеному вище прикладі, в першому випадку в потік зайде весь масив, а в другому будуть послідовно згенеровані його елементи. За допомогою функції from ми можемо згенерувати Observable з промісу.

```
const promiseSource = from(new Promise(resolve => resolve('Hello World!')));
const subscribe = promiseSource.subscribe(val => console.log(val))
```

Функція fromEvent дозволяє згенерувати Observable з події за переданим джерелом подій, наприклад посиланням, кнопкою або об'єктом типу EventEmitter.

У прикладі нижче ми створюємо в шаблоні кнопку, позначаючи її якорем #mybutton, щоб мати можливість вибрати її в компоненті за допомогою декоратора ViewChild.

```
<button #mybutton>Click me</button>
```

Потім створюємо потік Observable за переданим елементом DOM.

```
import { Component, ViewChild, OnInit} from '@angular/core';
import { fromEvent } from 'rxjs';
@Component({
...
})
export class AppComponent implements OnInit{
@ViewChild('mybutton') button;
ngOnInit(){
fromEvent(this.button.nativeElement, 'click').subscribe(evt => {
console.log(evt);
}
);
```