**const, let, var**

1. **var** – змінна яка вспливає із контексту, що дозволяє нам:
   1. Викликати \ змінити змінну, ще до її об’явлення в коді
   2. Переопреділити \ звернутись до змінної об’явеної в іншому контексті
2. **let** – змінна, що не вспливає із контексту, що дозволяє нам:
   1. Звертатись до змінної лише в контексті, в якому її було об’явленно
   2. Унеможливити виклик чи переопреділення змінної до її об’явлення, окрім ситуацій, коли змінна переопреділяється в функції (лайфхак)
3. **const** – володіє усіма рисами let і на додачу:
   1. Унеможливлює переопреділення примітивних типів
   2. Дозволяє міняти вміст складних типів даних, але забороняє робити зміну на інший тип данних (arr<!=>obj…)

**Arrow function**

Окрім синтаксису основною відмінністю від звичайних функцій є відсутність власного контексту.

This в звичайних функціях вказують на контекст в якому було створено дану функцію. В стрілочних фукнціях this завдяки відсутності власного контексту в даних функцій вказує на контекст контексту в якому була об’явлена функція. Тобто перепригує через контекс.

Щоб скористатись даною особливістю правильно варто використовувати стрілочну фукнцію в ситуаціях, коли потрібно переригрути контекст, наприклад для доступу до полів об’єкта із функції об’явленої в методі об’єкта із ключовим словом function.

**Default functions params**

В якості додаткового параметра можна задавати фукнції

**Function** (**a** = 10, **b** = **function**()). Якщо параметр b не переданий, то замість нього в функція потрапить результат виконання фукнції заданої як дефолне значення.

Функцї, яка задається як дефолтне значення можна передавати параметри, і параметром можна вказати параметри базової функції

**Function**(**a** = 10, **b** = **function**(**a**))

**rest /scpread**

Тема відноситься для **способу задання параметрів функції**.

Використання rest оператора при дистуктуризації складних типів даних і при об’явлені функції з довільною к-стю параметрів.

**rest**

Якщо ми хочемо зробити функцію, яка приймає різну к-сть параметрів, з якими ми б хотіли взаємодіяти, як із масивом данних. То можна використовувати **оператор rest (…argum)**, 3 точки перед змінною говорить коду, що всі змінні введені в функцію в якості параметра ми закинемо в масив.

Якщо нам потрібно, щоб окрім параметру rest функція приймала додаткові окремі параметри, які не мали б стати частиною масиву параметрів, то ми прописуємо структуру

Function(a, b, …arg) -> function(10, 12, 34, 32) -> a=10, b=12, arg[34,32]

**Параметри, що залишаться перемістяться в масив**

**Раніше** це робилось через оператор **arguments**, який є псевдомасивом, який ми приводили до масива з допомогою **Array.from(arguments)**

**spread**

Даний оператор необхідний для розгортання структури складних типів даних (масиви, об`экти).

Для розгортання масива достатньо прописати перед ним (…) –> …array, для розгортання об’єкта ми повинні його помістити в інший об’єкт -> { …obj }

Spread зручно використовувати для **копіювання масивів, об’єктів, для конвертування псевдомасивів(масивоподібних об’єктів) в масив, для передачі набору простих типів данних в методи**, які не працюють із складними типами даних (**max, min**)

1. **Приклад передача масива в метод, що не розуміє масиви**

const arr = [12,25,34,54,32]

Math.max() – метод приймає лише набір чисел, і не може взаємодіяти із масивами, для того, щоб вирішити дану ситуацію існує **спосіб розгортання spread оператор** –

arr -> **…arr =** [12, 25, 34, 54, 32] -> **12, 25, 34, 54, 32**

**Раніше** використовували для даної задачі **apply**, який приймає масив чисел в якості аргумента в контексті даної функцї – **Math.max().apply(null, [12, 25, 34, 54, 32])**

1. **Приклад копіювання масива**

Так як оператор масив переводить в набір елементів, то для копіювання достатньо його об’явити в [] нового масиву

arr1 = [1,2,3,4]

arr3 = [7,8,9,0]

arr2 = […arr1, …arr3] => [1,2,3,4,7,8,9,0]

1. **Приклад диструктуризації масиву з вкористанням spread оператора**

**const [a, b, …c] = arr** – ми отримаємо перші 2 значення в змінні і решта в вигляді масиву передати в третій параметр

1. **Приклад використання spread оператора (…) при копіюванні об’єктів**

**const newObj = {…obj } –** розгорнутий об’єкт потрібно загорнути в новий об’єкт.

Якщо ми хочемо в новий об’єкт розгорнути 2 об’єкта, то робимо це через кому, якщо в цих 2-х об’єктах є поле із однаковим ключем, то в новий об’єкт запишеться поле із останньо доданого об’єкта

1. **Приклади копіювання об’єкта \ масива з допомогою spread (…) оператора**

const […**newArr**] = arr – **newArr** буде точною копією масива

const {…**newObj**} = obj – **newObj** буде точною копією об’єкта

Якщо при копіюванні об’єкта через дистукруризацію і оператор spread(…)

Можна переопреділити скопійовані параметри нового об’єкта-копії

const {…**newObj, code: 123**} = obj

**Modules (exports/imports)**

Найкраще використовувати звичайні **export і** диструктуровий імпорт

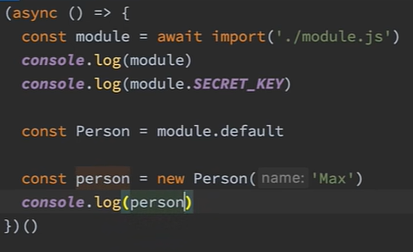
**В модулі З якого ми експортуємо** допустимі такі конструкції експорту:

1. **export** + variable / function
2. **export** **default** + {}

**В модулі В який імпортуються** файли:

1. **import** + … + **from** ‘./…’
   1. **varName** – якщо імпортуємо конструкцію передану через default
   2. **destructions** – {currentName, currentName} – змінні\методи які передані через звичайний export ми імпортуємо через диструктуризацію
   3. **\*** as **varName – імпортуємо всі експорти в дану змінну**
      1. Через крапку використовуємо змінні експортовані звичайним способом
      2. Конструкції експортовані через **export default** імортуємо звернувшись до поля default нашої змінної

**Динамічні імпорти**



За допомогою асинхронної самовикликаючої функції ми імпортуємо на клієнт модулі, які приходять у вигляді об’єкта і взаємодіємо із його експортами, як із ключами. Дефолтний експорт отримуємо через ключ default

**OOP**

**Переписування методів бальківського класу**

Для переписування методів батьківського класу ми створюємо метод із такою ж назвою в класі насліднику і переписуємо його функціонал.

Для доповнення методу батьківського класу, використовуємо ключове слово **supper.parentFuntionName()**, в методі із таким же ім’ям і добавляємо новий функціонал

**Static властивості**

Властивості із ключовим словом **static** доступні лише із самого класу, навідміну від усіх інших змінних

**Symbol**

7-мий тип даних в JS, призначений для утворення унікальних значень

**const** demo = Symbol**(**str**)** – задаємо параметр для змінної типу symbol

Особливістю даної конструкції, що при створені нової змінної із таким же параметром, при будь-якому порівнянні (==, ===), результат буде false. Ці дві змінн абсолютно унікальні, навіть із однаково переданий параметром при ініціалізації

Ключ об’єкта заданого через **symbol** не видимий в циклі **for in**

**Promises(async/await)**

Проміси це обгортка для асинхронного коду, яка допомагає легше із ним взаємодіяти і прибирає вложеності колбеків(callback hel)

В проміса, який нам повернувся, як результат, ми можемо вивести ще проміси

async/await – простіша форма роботи із промісами. Робить асинхронний код синхронним

аналогом then є присвоювання результату роботи асинхронної функції змінній, а аналогом reject – є конструкція try catch в якій пишиться логіка всього asinc/await

**Map/Set/WeakMap/WeakSet**

**Map / Set**

**Map**

Метод, який приймає в якості параметра об’єкт з якого створили карту методом Object.entries – масив масивів двох значень, ключ –> значення. Даний конструктор використовується для зручної взаємодії із об’єктом.

Навідміну від об’єктів ключем в map може бути будь-який тип даних, включно з об’єктами

Створюємо екземпляр класу Mep і отримуємо доступ до методів даного об’єкта:

1. **.get(**key**)** – отримуємо значення по ключу
2. **.set(**key, value**)** – задаємо значення (key – любого типу даних)
3. **.clear() –** очистити весь Map
4. **.has(**key**)** – перевірка наявності ключа -> **повертає Boolean**
5. **.delete(**key**) –** видалення пари за ключем -> **повертає Boolean**
6. **.size() –** повертає к-сть пар ключ -> значення
7. **.keys()** – повертає об’єкт ключів, ітерується for of
8. **.values()** – повертає об’єкт значень, ітерується for of
9. **.entries()** – повертає нам звичний масив типу entries ключ-значення (
   1. найчастіше використовується для роботи для ітерації в масиві **for of ,** хоча даний метод викликається по замовчуванню
   2. Для передачі в fromEntries map в якості параметра )

**Всі методи повертають** об’єкт екземпляр map, що дозволяє зробити запис ланцюгом (chaning)

Ключем в map можуть бути і об’єкти. Але при використанні метода **fromEntries** ключ-об’єкт не зможе бути ключем об’єкта сформованого із карти і буде приведений до строки

Розгортання структури Map оператором spead – поверне нам масив масивів типу entries

**Set**

Приймає масив значень і видаляє усі дублікати із нього, залишаючи в собі лише оригінальні значення

Методи:

1. **.size() –** розмір сета
2. **.add()** – додати елемент
3. **.delete() –** чистить set
4. **.keys()** – значення set, так як ключів немає, ітерується for of
5. **.entries()** – повертає об’єкт масивів пар ключ –значення, які в set однакові
6. **.values()** – значення сета, ітерується

**WeakMap**

Ключем в карті weakmap можуть бути лише об’єкти. При видалені об’єкта, що являвся кючем в структурі weakmap, зборщиком мусору – видалиться і пара ключ-значення, ключем якої був даний об’єкт, навідміну від map.

Методи, що є в weakmap – **get, set, has, delete**

**WeakSet**

Приймає лише масив об’єктів, при видалені об’єкта десь в коді зборщиком мусору, даний об’єкт видаляється і з **weakmap.** Методи в якості параметра приймають лише масив об’єктів

Методи, що є в weakmap – **add, set, has, delete**

**Reflect**

**-> Reflect**

Створення екземпляру класу може здійснюватись через **Reflect**.**construct**(CurrentClass, array, ProtoClass) -> Сучасна версія **Object.create()**

**CurrentClass** – поточний клас від якого утворється екземпляр

**Array** – масив параметрів, які ми маємо передати в конструктор

**ProtoClass** – клас який буде записаний в \_\_proto\_\_ нашого екземпляру

**Reflect**.**apply**(obj.function, contextObject, paramsForFunction) – виклик методів з конкретним контекстом

**obj.function** – метод який ми хочемо використати, без виклику

**contextObject** – об’єкт в контексті якого ми хочемо його викликати

**paramsForFunctiono** – параметри, які необхідно передати в метод, який викликаємо

**Reflect**.**ownKeys**(object) – отримуємо ключі екземпляру класу(окрім методів)

**Reflect**.**reventExtensions**(object) – блокуємо модифікацію об’єкта

**Reflect**.**isExtensible**.student – перевірка на заблокованість об’єкта

**WeakRef**

<https://github.com/tc39/proposal-weakrefs>

Спеціальний конструктор в який ми передаємо параметром об’єкт. Для звернення до полів такого об’єкта використовується властивість **deref**. В разі видалення об’єкта зборщиком мусору, ми не зможемо звернутись до його полів, навідміну від даної конструкції

Найкраще даний конструктор працює в поєднанні із конструктором FinalizationRegistry(cb(value)) – якщо в даного конструктора створити екземпляр класу і передати в нього екземпляр класу weakref – то при очищені даного об’єкта зборщикм мусору, ми отримаємо повідомлення яка виведеться в колбеку нашого finalizationregistry

**Оператор для роботи із функціями вищого порядку**

<https://github.com/tc39/proposal-pipeline-operator>

**=================================================**

**Prototype**

**[[ prototype ]]** – властивість притаманна усім об’єктам в js і посилається на prototype (на батьківський об’єкт)

**Prototype** – об’єкт, яка належить усім об’єктам і кожен дочірній об’єкт містить в собі властивість **[[ prototype ]],** яке зсилається на даний об’єкт батьківського об’єкта. При пошуку властивості чи метода, спершу js дивиться в поточному об’єкті, а при його відсутності шукає вглибину прототипів усіх батьківський об’єктів.

Даний об’єкт-властивість слугує для розширення функціоналу все існуючих об’єктів в JS і для наслідування

**Object** – головний батьківський об’єкт і запис будь-яких методів чи властивостей в його prototype дозволить усім сутносям мати до нього доступ

**Object.create**(obj) – поверне об’єкт, в властивість [[ prototype ]] (\_\_proto\_\_) якого вкаже prototype переданого об’єкта, тобто унаслідує властивості переданого об’єкта

**This / call, apply, bind**

**This** – ключове слово, яке вказує на контекст в якому його було викликано

**Call/apply/bind** – методи для переопреділення контексту

**Bind** – повертає метод, який ми перевизначаємо, з уже переопреділеним контекстом і ми можемо передати параметри одразу, або змінній, в яку передати результат виконання даного методу

**Call** – також перевизначає контекст, але не повертає методу

**Apply** – робить те ж, що і вище – міняє контекст, але приймає параметри в вигляді масива

Єдина різниця між методами в способі передачі параметрів

**Замикання**

Замикання – це функція в середині іншої функції

**Асинхронність**

**Event loop**

Є декілька місць виконання коду js в браузерному середовищі. **Call stack, web apis, event loop**

В **call stack** – місце в якому виконуються функції в тій послідовності, в якій вони прописані в коді. Проте, якщо це асинхронні операції(події, взаємодії із сервером, таймаути ), то вони реєструються в **call stack**, як і будь-які функції, але, щоб не затримувати послідовне виконання синхронного коду, потрапляють із **call stack** в **web apis**, де очікують свого виконання, після чого потрапляють в **event loop** (callback queue), це цикл перевіряє наявність нових функцій, що потрапляють в нього після асинхронного виконання, і добавляє їх в основний стек після виконання усіх функцій в **call stack** (проте закидаються в стек вони по одному і в послідовності, в якій вони були передані в **event loop**)

Жодна асинхронна операція не може бути виконана раніше, ніж код в основному стеку, навіть setTimeout з 0 часом очікування

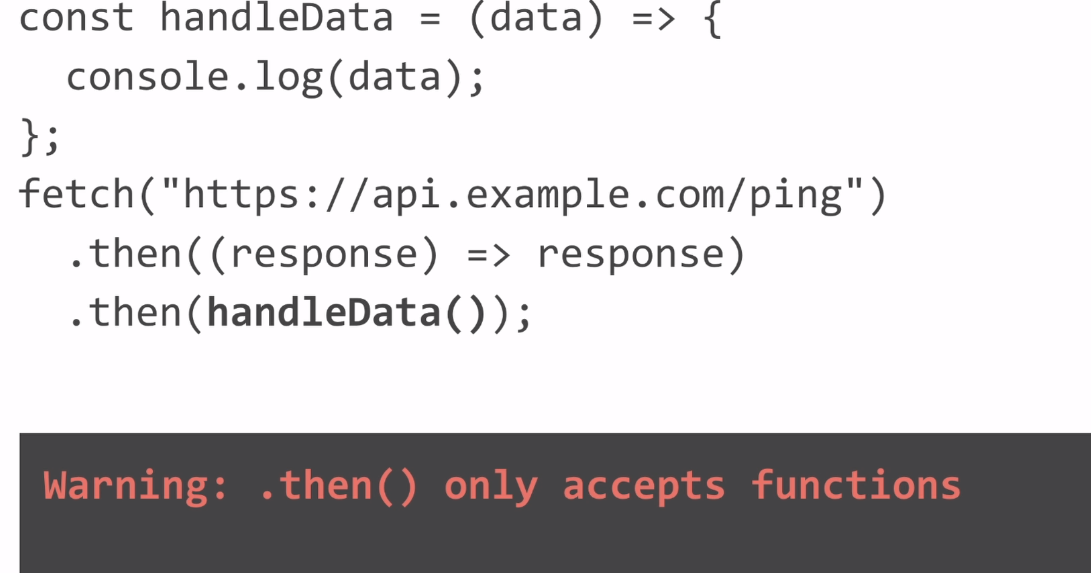
**Promise**

Promise **–** обгортка для асинхронного коду, з метою зменшити к-сть вложень callback – ів і забезпечення більшого розуміння коду

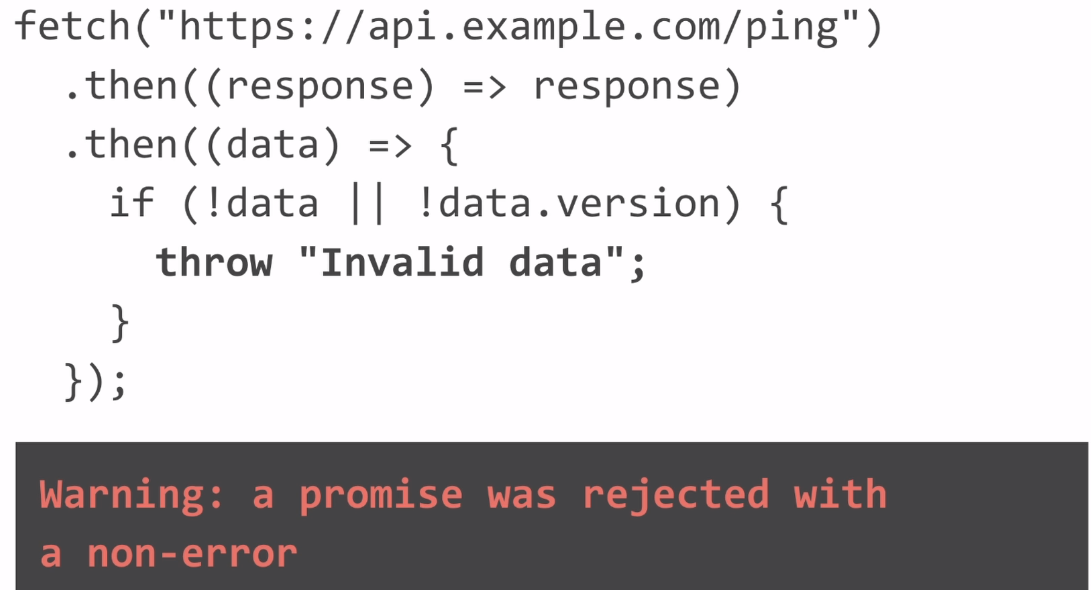
**Promise**.**race**([promise, promise]) – виконається при виконанні найшвидшого промісу

**Помилки написання використання Promises:**

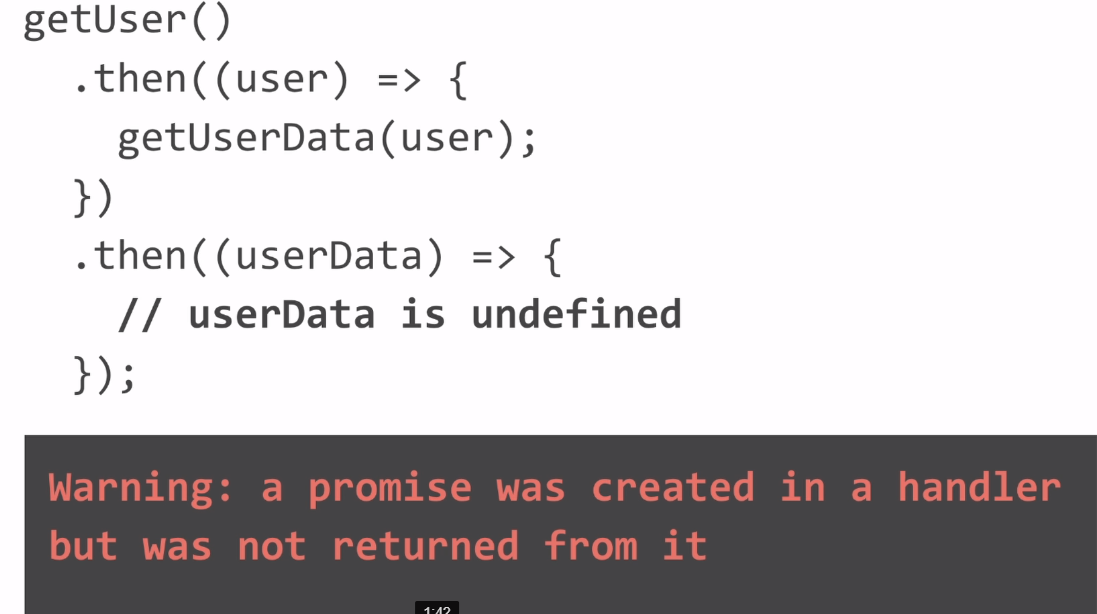
1 В метод then передаємо не функцію – помилка



2 reject проміса не з переданою помилкою типу помилка



3 Створюємо promise але забуваємо його повернути:



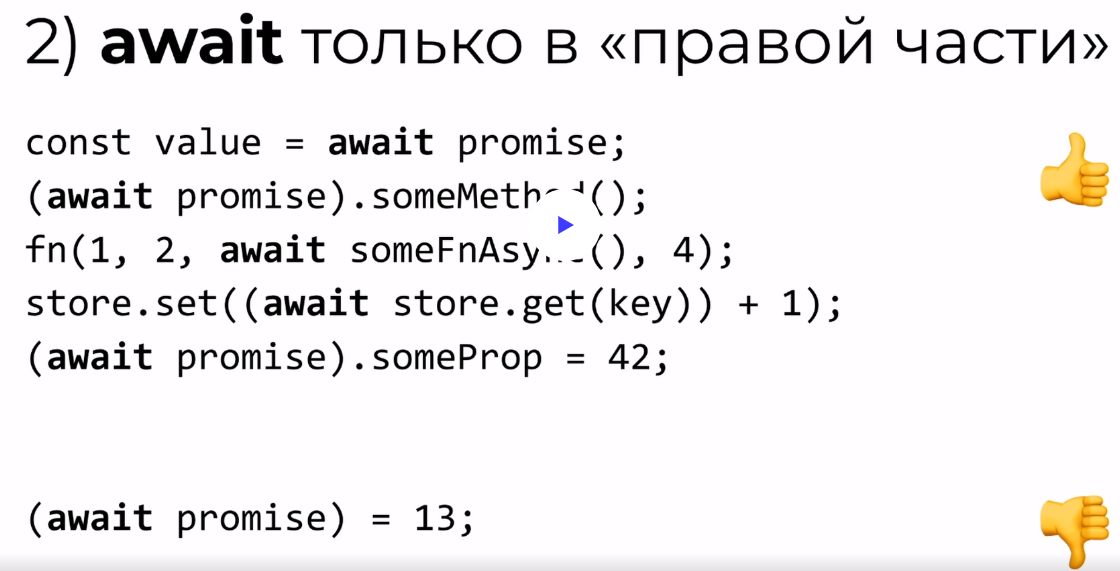
**Async/await**

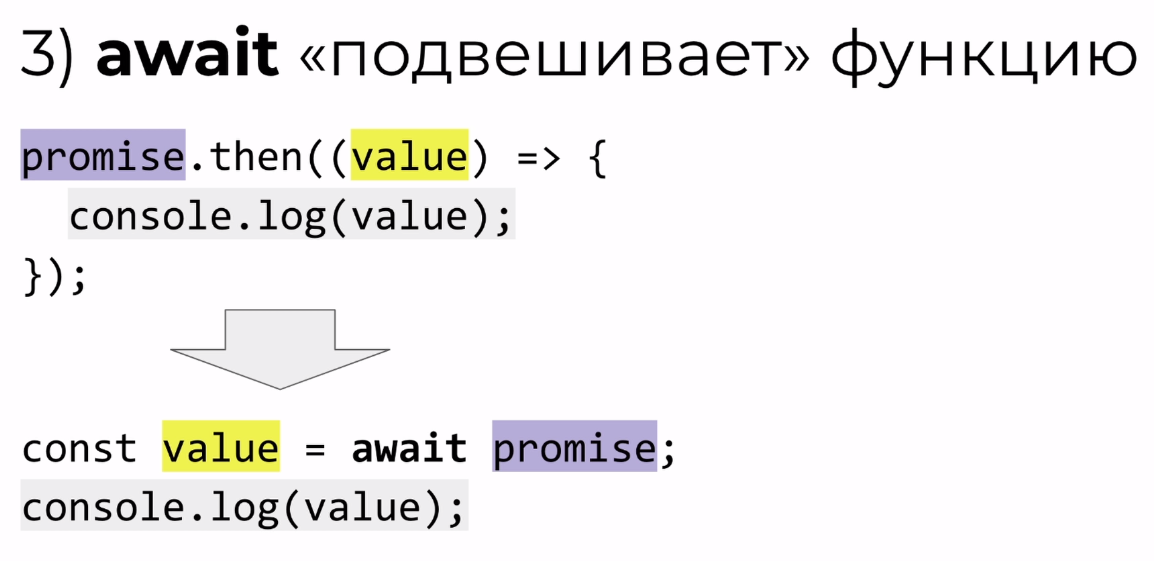
Дані оператори допомагають зробити асинхронний код синхронним

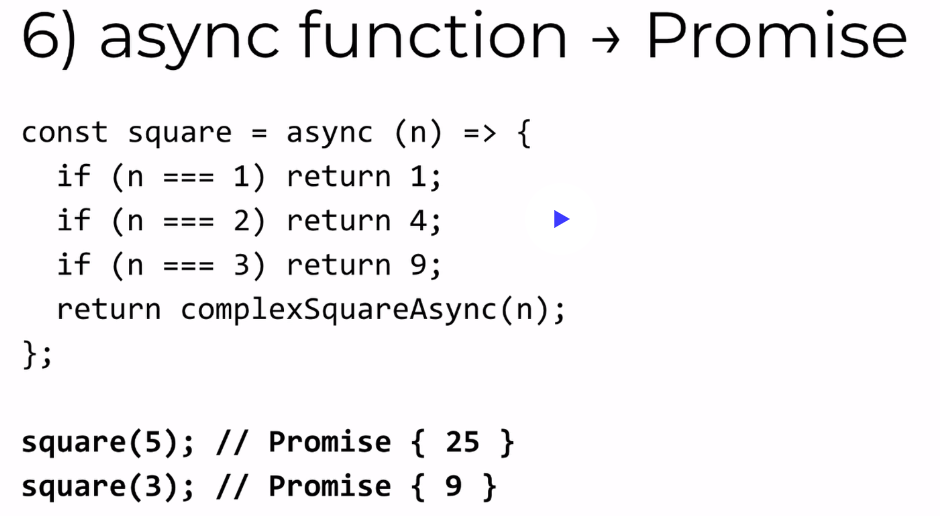
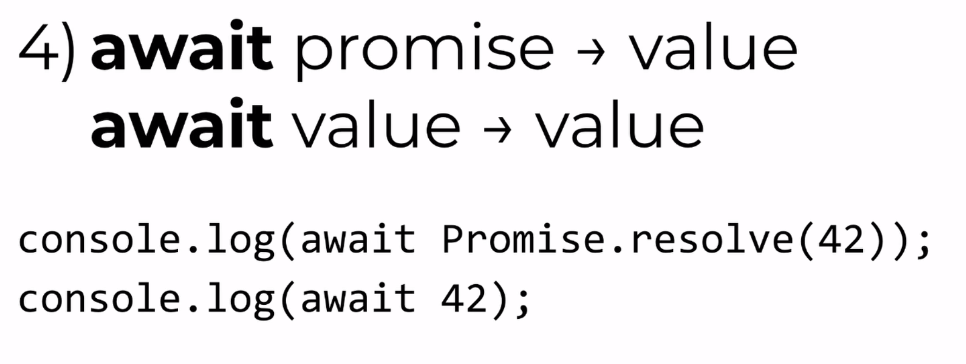
await – оператор, що вказує синхронному коду не виконуватись, доки він очікує promise. Аналогом then – є звичайне присвоювання змінній. А обробка помилок відбувається через try{}catch{}. Опрацьовує і отримує завжди проміси

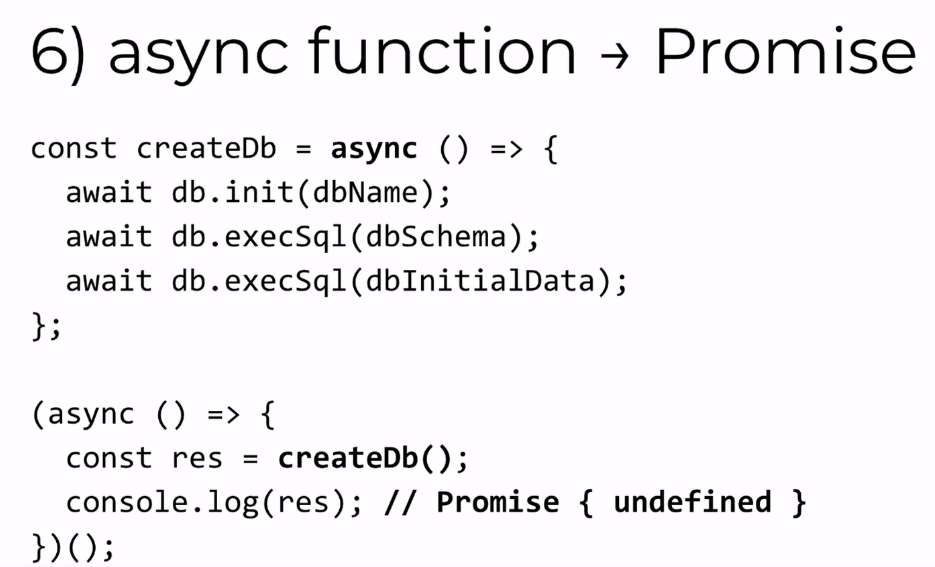
async – поверне promise, коли в даній функції відпрацює останній оператор await. Потрібен синтаксично для використання оператора await в середині функції, якій він присвоюється.

**Правила використання:**





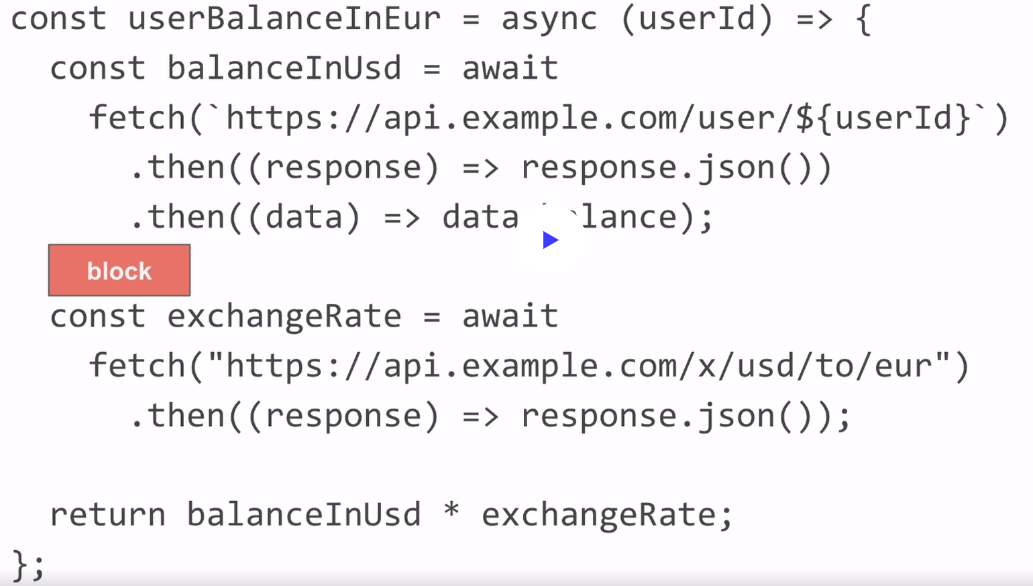




**Небезпеки:**

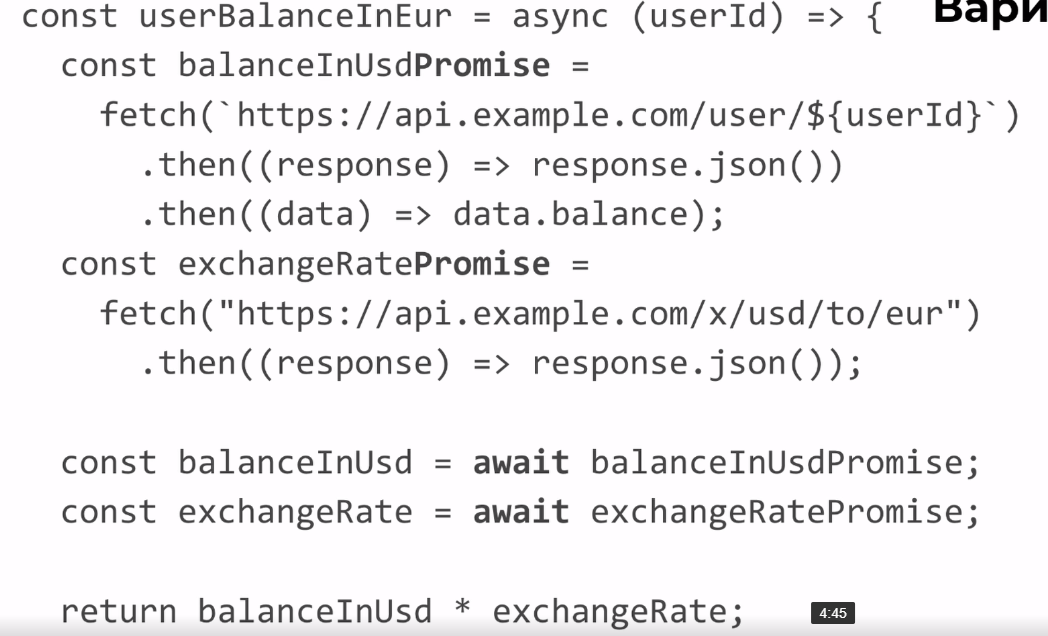
Async/await настільки похожий на синхронний код, що це може викликати певні трудності:

1. Доки не виконається перший await – другий не почне діяти і буде заблокований

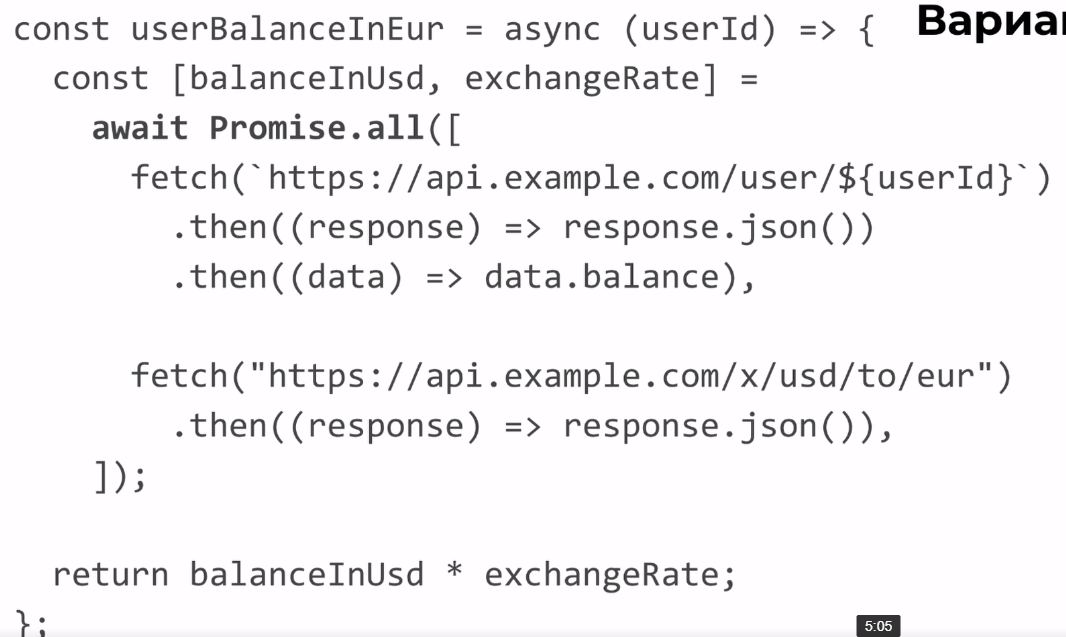


**Рішення:**

1. Виконається 2 асинхронні операції, без блоку один одного і тільки після ми правюватимемо з async/await для обробки результатів

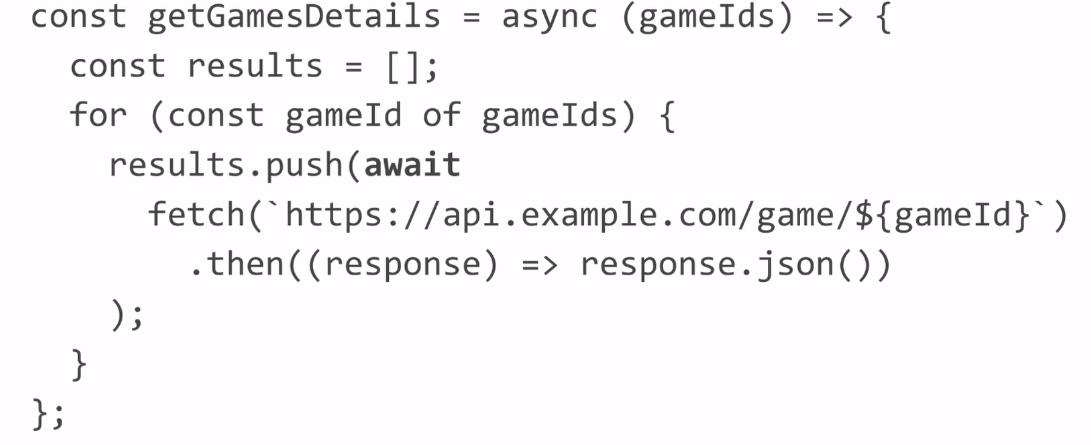
****

1. Promise.all – поверне масив результатів промісів, лише, коли всі синхронні операції виконаються

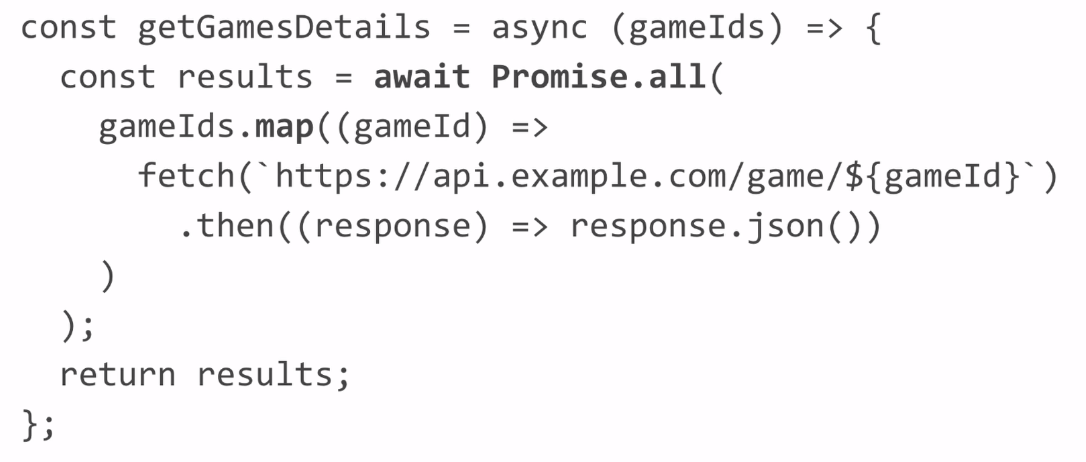


Ще приклад реалізації

Поганий приклад реалізації, так як по черзі виконуватимусть асинхронні операції і добавлятимусть в масив



Праивльний, так як всі запити будуть асинхронними, але результат повенеться, коли всі операції будуть виконані



**Flexible object config**

**Object.create(**prototypeObj, objectDescriptor**) –** метод для створення об’єкта.

**objectDescriptor** – об’єкт, властивості якого налаштовуються перш ніж перейдуть до новоствореного об’єкта.

**prototypeObj** – об’єкт, prototype якого буде записаний в \_\_proto\_\_ нашого новоствореного об’єкта (всі властивості даного об’єкта будуть перебиратись for in масивом)

Всі поля **prototypeObj** зададуться поточному об’єкту і  **відображатимуться в циклі for in**

**Для уникнення даної ситупції завджи рекомендується в циклі for in робити перевірку полів ітеруємого об’єкта**

**obj.hasOwnProperty(**key**) –** перевірка належності ключі об’єкта лише поточному об’єкту

Всі поля **objectDescriptor** зададуться поточному об’єкту, але **не відображатимуться в циклі for in**

В об’єкті дискріпторі кожному ключу належить об’єкт із властивостями, по яким ми контролюємо спосіб взаємодії із ними

**Параметри для кожного поля дискріптора:**

1. **enumerables**: false – відображення ключа в ітераціях for in
2. **value**: - значення поля
3. **writable**:false - переписуваність полів
4. **configurable**:false – можливість видаляти поля об’єкта
5. **get()** – метод, який описує логіку отримання даних конкретного поля
6. **set(value)** – метод який описує логіку присвоєння даних конкретному полю.

**LocalStorage**

**LocalStorage -** місце зберігання даних в середині браузера, навіть при перезагрузці сторінки. Методи **LocalStorage** вміють працювати лише із строками і повертають строчне представлення будь-якого типу даних. При роботі із об’єктами які потрапляють в локальне сховище, то потрібно перевести його в строку **JSON**.**stringify**(obj), для отримання даних **JSON**.**parse**(localStorage.getItem(‘key’))

**Методи:**

**localStorage.getItem(‘key’) –** отримати значення за ключем

**localStorage.setItem(‘key’, ‘value’) –** отримати значення за ключем

**localStorage.removeItem(‘key’) –** видалення значення за ключем

**localStorage.clear() –** очищення всього localstorage

**Події:**

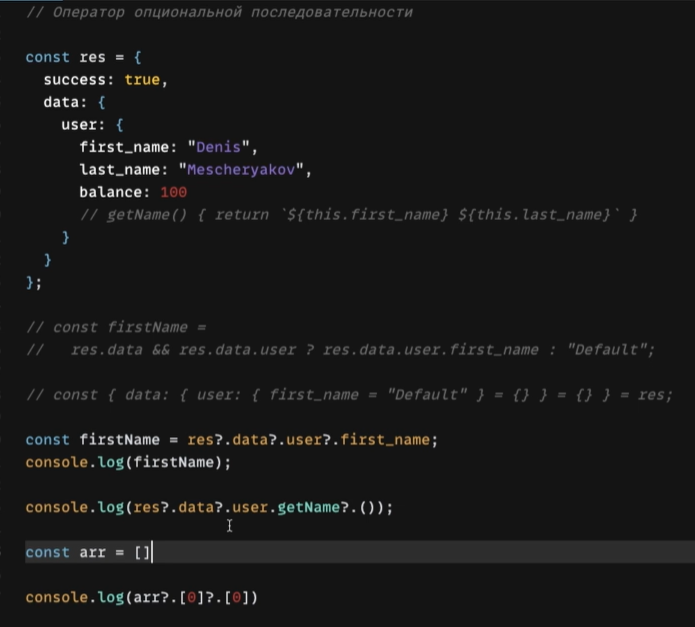
**storage** – спрацьовує, коли ми взаємодіємо із localStorage в іншій вкладці пов’язаної із нашим додатком. Все це для того, щоб локальні сховища, які відносяться до одного додатку спілкувались будучи на різних вкладках

**Співбесіди:**

Різниця між localStorage i kookie – в розмірі даних, які можна в ньому зберігати. **В локальному сховищі 5мб, що більше ніж в кукі**

Основна відмінність kookie те, що вони відлітаються із запитами на сервер, і дані кукі сервер може прочитати і це не дуже секюрно. А локальне сховище не відправляє дані на сервер

**Оператори опціональної послідовності**



**Оператор опціональної послідовності** – використовується для перевірка полів об’єкта, з вложеними об’єктами і методами на їх наявність, з можливістю задати дефолтне значення і з меншою к-стю коду

**Оператор нульового злиття**

<https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Operators/Nullish_coalescing_operator>

Оператор нульового злиття ‘ ?? ’ – використовується як точніший аналог оператора порівняння ||.

Якщо ліва частина виразу може бути приведена до false, але вона не рівна null чи undefined, то вернеться дане значення, навідміну від оператора ||, який будь-яке значення, що може бути приведене до false пропустить

В випадку, якщо ліва частина null або undefined, то повернеться права

null ?? ‘none’ => ‘none’

0 ?? ‘none’ => 0 0 || ‘none’ => ‘none’

**Symbol**

7 – тип даних в js, який являється унікальним ідентифікатором. 2 змінні створені від одного symbol(), навіть при передачі параметів в даний метод одного виду, будуть не рівні один одному.

Цікавою властивістю є можливість задавати змінну утворену від symbol() в якості ключа об’єкту. Дане поле не буде ітеруватись циклом for in. Єдиний спосіб отримати доступ до полів такого типу через **Object.getOwnPropertySymbol**(obj) – видасть масив полів.

Є декілька методів Symbol(): **.for**(str), **.keyFor**(var)

Symbol().for(str) – утворює по параметру, який ми передаємо в глобальному просторі і всі змінні, утворені, від цього ключа будуть однакові

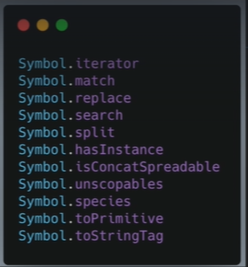
str1 = Symbol().for(‘name’) str1 = Symbol(‘name’)

str2 = Symbol().for(‘name’) str2 = Symbol(‘name’)

str1 === str2 => true str1 === str2 => false

Symbol()**.keyFor**(str1) – поверне його symbol

Існують спеціальні ключові слова Symbol – завдяки якій ми можемо переписувати певну логіку взаємодії із складними типами данних



Приклад роботи із Symbol.toPrimitive() – завдяки якому ми переписуємо логіку приведення поточного об’єкта до примітивних типів даних

