ЛЕКЦІЯ 9

ОСНОВИ JAVASCRIPT

9.1. Стандарт ECMA

JavaScript був розроблений Бренданом Ейчем (Ейхом або Айокм) в 1995 році в Netscape Communications. JS є реалізацією стандарту мови програмування ECMAScript.

Брендан Ейч (англ. Brendan Eich /ˈaɪk/, народився 4 червня 1961, у Пітсбургу, Пенсильванія, США) – американський програміст, розробник мови програмування JavaScript. Ейч отримав ступінь бакалавра в університеті округа Санта-Клара. В 1986 році отримав ступінь магістра в Іллінському університеті в Урбана-Шампейн.

З 1995 року працював у компанії Netscape; брав участь у заснуванні Mozilla, до 2014 року був головним інженером в Mozilla Corporation. В березні 2014 року був призначений CEO, проте швидко покинув Mozilla під тиском невдоволених його призначенням через підтримку їм в 2008 році заборони одностатевих браків. Сьогодні працює виконавчим директором Brave Software.

ECMAScript – стандарт мови програмування, затверджений міжнародною організацією ECMA згідно зі специфікацією ECMA-262. Найвідомішими реалізаціями стандарту є мови JavaScript, JScript та ActionScript, які широко використовується у веброзробці.

JavaScript має низку властивостей об'єктно-орієнтованої мови, але завдяки концепції прототипів підтримка об'єктів в ньому відрізняється від традиційних мов ООП. Крім того, JavaScript має ряд властивостей, притаманних функціональним мовам, – функції як об'єкти першого рівня, об'єкти як списки, каррінг (currying), анонімні функції, замикання (closures) – що додає мові додаткову гнучкість.

JavaScript має С-подібний синтаксис, але в порівнянні з мовою Сі має такі корінні відмінності:

1 об'єкти, з можливістю інтроспекції і динамічної зміни типу через механізм прототипів

2 функції як об'єкти першого класу
3 обробка винятків
4 автоматичне приведення типів
5 автоматичне прибирання сміття
6 анонімні функції

ECMAScript (англ. ECMAScript) – стандарт JavaScript, призначений для забезпечення сумісності веб-сторінок в різних браузерах, який стандартизований Ecma International в документі ECMA-262.

ECMAScript зазвичай використовується для написання сценаріїв на стороні клієнта в World Wide Web, і він все частіше використовується для написання серверних програм і сервісів за допомогою Node.js та інших фреймворків.

Стандарт ECMA визначає мову ECMAScript 2022. Це тринадцяте видання специфікації мови ECMAScript. З моменту публікації першого видання в 1997 році, ECMAScript стала однією з найбільш широко використовуваних у світі мов програмування загального призначення. Найбільш відома як мова, вбудована в веб-браузери, але також широко прийнята для серверних і вбудованих додатків.

ECMAScript заснований на декількох технологіях, найбільш відомими з яких є JavaScript (Netscape) і JScript (Microsoft). Мова була винайдена Бренданом Ейчем в Netscape і вперше з'явилася в браузері Navigator 2.0 цієї компанії. Він з'явився у всіх наступних браузерах від Netscape і у всіх браузерах від Microsoft починаючи з Internet Explorer 3.0.

Розробка специфікації мови ECMAScript почалася в листопаді 1996 року. Перший випуск цього стандарту Ecma був прийнятий Генеральною Асамблеєю ЕКМА від червня 1997 року.

Стандарт ECMA був переданий ISO/IEC JTC 1 для прийняття в рамках процедури швидкого відстеження, і затверджений як міжнародний стандарт ISO/IEC 16262, в квітні 1998 року. Генеральна Асамблея ЕКМА від 1998 червня затвердила другий випуск ECMA-262, щоб зберегти його повністю узгоджений з ISO/IEC 16262. Зміни між першим і другим виданням носять редакційний характер.

У третьому виданні стандарту були введені потужні регулярні вирази, краща обробка рядків, нові керуючі заяви, обробка винятків спроб/вилову, більш жорстке визначення помилок, форматування для числового виведення і незначні зміни в очікуванні майбутнього зростання мови. Третє видання стандарту ECMAScript було прийнято Генеральною Асамблеєю ECMA від грудня 1999 року і опубліковано як ISO/IEC 16262:2002 у червні 2002 року.

Після публікації третього видання, ECMAScript досяг масового поширення в поєднанні з World Wide Web, де він став мовою програмування, яка підтримується по суті всіма веб-браузерами. Значна робота була зроблена для розробки четвертого видання ECMAScript. Проте, ця робота не була завершена і не була опублікована як четверте видання ECMAScript, але деякі з елементів цієї версії були включені в розробку шостого видання.

В п'ятій версії ECMAScript (опубліковано як 5-е видання ECMA-262) модифікували фактичну інтерпретацію специфікації мови, яка стала загальною для реалізацій в браузерах і додали підтримку нових можливостей, які з'явилися з моменту публікації третього видання. До таких особливостей належать властивості accessor, рефлекторне створення та перевірка об'єктів, програмний контроль властивостей атрибутів, додаткові функції маніпулювання масивами, підтримка формату кодування об'єктів JSON, а також жорсткий режим, що забезпечує посилену перевірку помилок і безпеку програм. П'яте видання було прийнято Генеральною Асамблеєю ECMA від грудня 2009 року.

Розробка шостої версії почалася в 2009 році, паралельно з публікацією п'ятої версії. Однак цьому передували значні експерименти та розробки дизайну мови, що датуються публікацією третього видання в 1999 році. В прямому сенсі завершення шостого видання є кульмінацією п'ятнадцятирічного зусилля. Цілі створення шостої версії включали забезпечення кращої підтримки великих застосунків, створення бібліотек, а також використання ECMAScript як базу компіляції для інших мов. Деякі з її основних удосконалень включали модулі, оголошення класів, пошук лексичних блоків, ітератори і генератори, спроби створення асинхронного програмування, шаблони деструктуризації і правильних хвостових викликів. Бібліотека вбудованих символів для ECMAScript була розширена з метою

підтримки додаткових абстракцій даних, включаючи карти, набори і масиви двійкових числових значень, а також додаткової підтримки додаткових символів Unicode в рядках та регулярних виразах. Шосте видання надає базу для регулярних, додаткових можливостей мови та бібліотеки. Шосте видання було прийнято Генеральною Асамблеєю від 2015 червня.

ECMAScript 2016 була першою версією ECMAScript, випущеною під новим щорічним випуском під егідою Ecma TC39 з процесом відкритої розробки. На підставі початкового документа ECMAScript 2015 було створено текстовий документ, який став основою для подальшого розвитку реалізованого повністю на GitHub. За рік розробки цього стандарту було подано сотні запитів і питань, які представляли тисячі виправлень помилок, виправлень редакцій та інших удосконалень. Крім того, були розроблені численні програмні засоби, які допомагали удосконаленню, включаючи Ecmarkup, Ecmarkdown і Grammakdown. ES2016 також включила в себе підтримку нового оператора піднесення у ступінь і додає новий метод до Array.prototype, який називається includes.

Версія 11 – ECMAScript 2020 надає наступні можливості:

1. метод matchAll для рядків, щоб створити ітератор для всіх об'єктів, які генеруються глобальним регулярним виразом;
2. функцію import() та синтаксис, який асинхронно імпортує модулі з динамічним специфікатором;
3. bigint, нове число, яке використовується для роботи з довільними за розмірами цілими числами;
4. globalThis, універсальний спосіб доступу до глобального значення;
5. підвищена стандартизація порядку перерахування for-in;
6. Import.meta, об'єкт, доступний в Modules, який може містити контекстну інформацію про модуль;
7. додавання двох нових синтаксичних функцій для поліпшення роботи з «нульовими» значеннями (null або undefined): Null coalescing, оператор вибору значень;
8. оператор доступу до властивостей і функції, який викликає короткі замикання, якщо значення доступу/виклику є нульовим.

Щодо роботи з методами Promise:

https://www.wisdomgeek.com/development/web-development/javascript/javascript-promises-combinators-race-all-allsettled-any/

Версія 12 – ECMAScript 2021, 12-те видання, представив:

1. метод replaceAll для рядків;
2. Promise.any, комбінатор Promise, який має короткі замикання під час переповнення вхідних значень;
3. AggregateError, новий тип помилки для представлення декількох помилок відразу;
4. логічні оператори присвоювання (??=, &&=, ||=);
5. роздільники для числових літералів (1_000);
6. Array.prototype.sort був зроблений точнішим, зменшуючи кількість випадків, які призводять до реалізації певного порядку сортування.

Версія 13 – ECMAScript 2022, 13-е видання, представив:

1. нові елементи класу: public та private instance fields, public та private static fields, private instance методи, private static методи;
2. статичні блоки всередині класів;
3. синтаксис #x в OBJ, для перевірки наявності private fields об'єктів;
4. індекси відповідності регулярних виразів за допомогою прапорця /d, який забезпечує початкові і кінцеві індекси для відповідних підрядків;
5. дластивість Cause на об'єктах Error, яка може бути використана для запису ланцюжка причинності в помилках;
6. метод AT для даних типів Strings, Arrays, and TypedArrays, що дозволяє відносну індексацію;
7. Object.hasOwn, зручну альтернативу Object.prototype.hasOwnProperty.

9.2. Асинхроне програмування

Скрипти Java можуть виконуватися браузером в асинхронному режимі.

Асинхронність – це процес обробки введення/виводу, що дозволяє продовжити обробку інших завдань, не чекаючи завершення попереднього завдання.

Комп'ютерні програми часто мають справу з тривалими процесами. Наприклад, отримання даних з бази даних або виконання складних обчислень. Поки виконується одна операція, можна було б завершити ще кілька. А бездіяльність призводить до зниження продуктивності. Асинхронне програмування збільшує ефективність, тому що не дозволяє блокувати основний потік виконання.

У синхронному коді кожна операція чекає закінчення попередньої. Тому вся програма може «зависнути», якщо одна з команд виконується дуже довго.

Асинхронний код прибирає операцію, яка блокує основний потік програми, так що основний потік не блокується, і програма може виконувати інші операції.

Асинхронне програмування успішно вирішує безліч завдань. Одна з найважливіших — користувач може взаємодіяти з програмою поки виконується інше завдання.

Візьмемо для прикладу додаток, який підбирає серіали за зазначеними критеріями. Після того як користувач обрав параметри по яким буде відбуватися пошук, програма відправляє запит на сервер. А там здійснюється пошук вказаних фільмів. Обробка може тривати доволі тривалий час. Якщо додаток працює синхронно, то користувач не зможе взаємодіяти зі сторінкою, поки не прийде результат.

В цьому випадку головний потік виконання поділяється на дві гілки. Одна з них продовжує займатися інтерфейсом, а інша виконує запит (рис. 9.1).

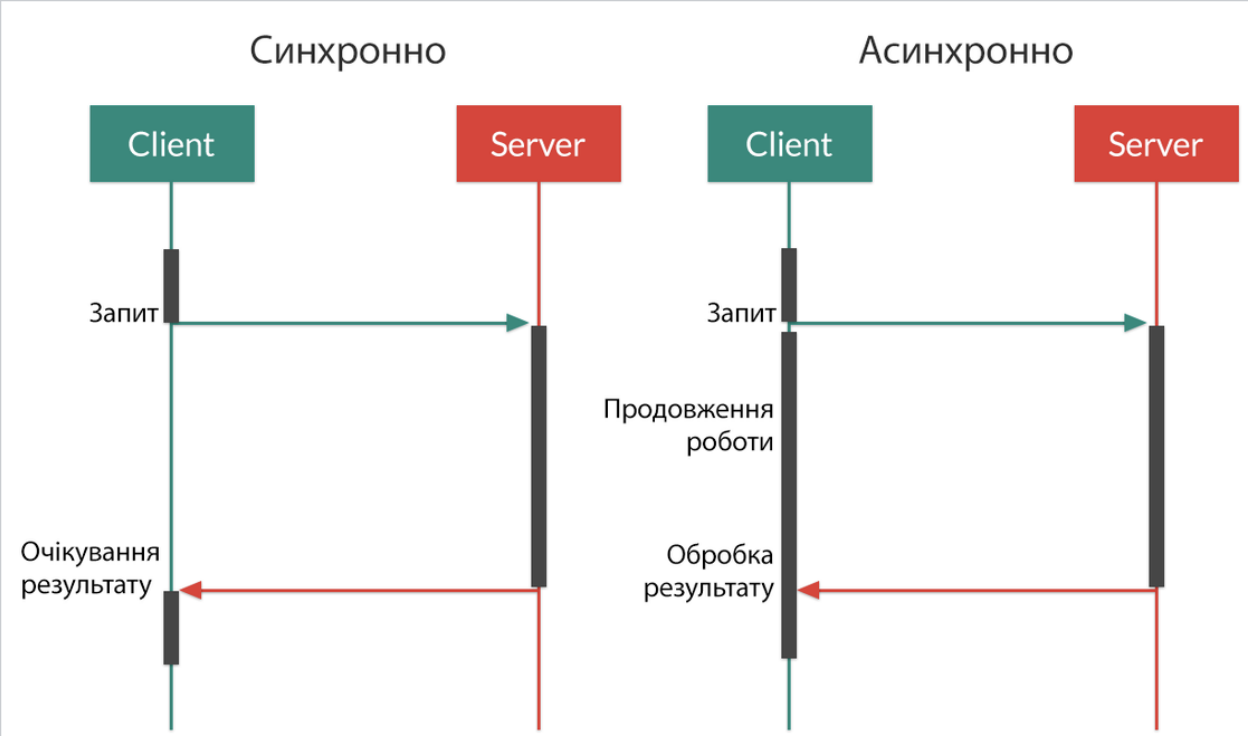


Рисунок 9.1. – Приклади асинхронного та синхронного кодів

9.3. Платформа Node.js

Програмним інструментарієм для створення застосунків JavaScript є Node.js.

Node.js – платформа з відкритим кодом для виконання високопродуктивних мережевих застосунків, написаних мовою JavaScript. Засновником платформи є Раян Дал (Ryan Dahl). Якщо раніше JavaScript застосовувався для обробки даних в браузері користувача, то node.js надав можливість виконувати JavaScript-скрипти на сервері та відправляти користувачеві результат їхнього виконання. Платформа Node.js перетворила JavaScript на мову загального використання з великою спільнотою розробників.

Node.js має наступні властивості:

1. асинхронна одно-нитева модель виконання запитів;
2. неблокуючий ввід/вивід;
3. система модулів CommonJS;
4. рушій JavaScript Google V8;

Для керування модулями використовується пакетний менеджер npm (node package manager).

Для забезпечення обробки великої кількості паралельних запитів у Node.js використовується асинхронна модель запуску коду, заснована на обробці подій в неблокуючому режимі та визначенні обробників зворотніх викликів (callback). Як способи мультиплексування з'єднань підтримується epoll, kqueue, /dev/poll і select. Для мультиплексування з'єднань використовується бібліотека libuv, для створення пулу нитей (thread pool) задіяна бібліотека libeio, для виконання DNS-запитів у неблокуючому режимі інтегрований c-ares. Всі системні виклики, що спричиняють блокування, виконуються всередині пулу потоків і потім, як і обробники сигналів, передають результат своєї роботи назад через неіменовані канали (pipe).

Стандартна поставка node.js включає в себе кількадесят модулів, у яких реалізовані типові операції для взаємодії з операційною системою, файловою системою, мережею і протоколами, утиліти для обробки даних.

Крім того є доступними безліч модулів від незалежних розробників, програмісти можуть отримати їх з відкритих репозиторіїв і використовувати у своїх проектах.

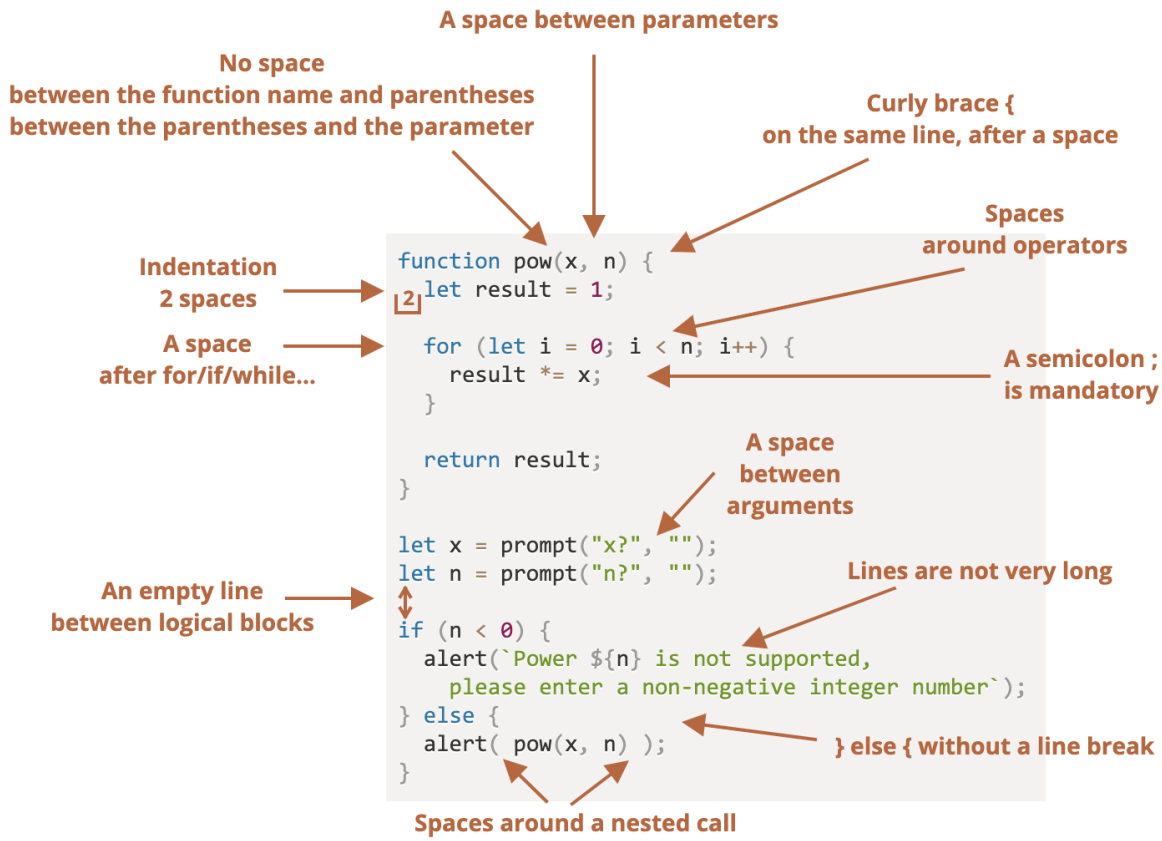
https://javascript.info/coding-style

9.4. Стиль кодування JavaScript

Наш код повинен бути максимально зрозумілим і простим для читання.

Це, власне, мистецтво програмування – взяти складне завдання і кодувати його так, щоб він був і коректним, і придатним для читання людиною. Хороший стиль коду значно допомагає в цьому.

Наведемо список деяких рекомендацій синтаксису для виконання зазначеної вимоги:



Розглянемо детальніші наведені рекомендації.

Фігурні дужки

У більшості проектів JavaScript фігурні дужки записується в «єгипетському» стилі з дужкою, що розкривається, на тому ж рядку, що і відповідне ключове слово – а не на новому рядку.

```
if (condition) {
  // do this
  // ...and that
  // ...and that
}
```

Однорядковая конструкція запису, така, наприклад, як if (умова) doSometing(), є частим випадком. Чи варто взагалі використовувати фігурні дужки в цьому випадку?

Наведемо кілька варіантів, для визначення їх читабельності та зрозумілості:

1. 😠 Початківці часто кодують наступним чином. Але в даному випадку фігурні дужки не потрібні:

```
if (n < 0) {alert(`Power ${n} is not supported`);}
```

2. 😠 Можливе виділення умови окремим рядком без дужок. Так робити ніколи не можна, тому що легко помилитися, якщо виникне потреба у доданні ще одного рядка до операторів виконання цієї конструкції:

```
if (n < 0)
  alert(`Power ${n} is not supported`);
```

3. 😏 Написання конструкції одним рядком без дужок є прийнятним, якщо оператор виконання є досить коротким:

```
if (n < 0) alert(`Power ${n} is not supported`);
```

4. 😃 Останній варіант є найкращим:

```
if (n < 0) {
  alert(`Power ${n} is not supported`);
}
```

Для дуже короткого коду дозволяється один рядок, наприклад, якщо оператор умови повертає null. Але кодовий блок (останній варіант), як правило, більш читабельний.

Довжина рядка коду

Ніхто не любить читати довгий горизонтальний рядок коду. Найкраще їх розділити.

Наприклад:

```
// backtick quotes ` allow to split the string into multiple lines
let str = `
  ECMA International's TC39 is a group of JavaScript developers,
  implementers, academics, and more, collaborating with the community
  to maintain and evolve the definition of JavaScript.
`;
```

Або для наступних операторів:

```
if (
  id === 123 &&
  moonPhase === 'Waning Gibbous' &&
  zodiacSign === 'Libra'
) {
  letTheSorceryBegin();
}
```

Максимальна довжина рядка повинна бути узгоджена на рівні команди розробників. Зазвичай це 80 або 120 символів.

Відступи

Розрізняють два типи відступів:

- **Горизонтальні відступи: 2 або 4 пробіли.**

  Горизонтальний відступ додається за допомогою пробілів 2 або 4 або горизонтального символу табуляції (Tab). Який вибрати – це стара дискусія. На сьогоднішній день пробіли більш поширені.

  Однією з переваг пробілів над табуляцією є те, що пробіли дозволяють гнучкіші конфігурації відступів, ніж символ табуляції.

  Наприклад, ми можемо вирівняти параметри за допомогою фігурної дужки, що відкривається, наприклад:

```
show(parameters,
     aligned, // 5 spaces padding at the left
     one,
     after,
     another
  ) {
  // ...
}
```

- **Вертикальні відступи: порожні рядки для розбиття коду на логічні блоки.**

  Навіть одну функцію зазвичай можна розділити на логічні блоки.

  У наведеному нижче прикладі ініціалізація змінних, основний цикл і повернення результату розбиті вертикально, тобто визначені як окремі складові:

```
function pow(x, n) {
  let result = 1;
  //             <--
  for (let i = 0; i < n; i++) {
    result *= x;
  }
  //             <--
  return result;
}
```

  Слід додавати додатковий порожній рядок для підвищення читабельності кода. Не повинно бути більше дев'яти рядків коду без вертикального відступу.

Крапка з комою

Крапка з комою має ставитися після кожного виразу, навіть якщо їх можна було б пропустити.

Існують мови, де крапка з комою є дійсно необов'язковою, і вона рідко використовується. У JavaScript, однак, є випадки, коли розрив рядка не

інтерпретується як крапка з комою, і таким чином робить код вразливим до помилок.

Якщо Ви досвідчений програміст JavaScript, Ви можете вибрати стиль коду без крапки з комою, наприклад StandardJS. В іншому випадку, краще використовувати крапки з комою, щоб уникнути можливих помилок. Більшість розробників ставлять крапки з комою.

Рівні вкладення

Намагайтеся уникати вкладеності коду у надто багато рівнів у глибину.

Наприклад, у циклі іноді є хороша ідея використовувати директиву Continue, щоб уникнути додаткового вкладеності.

Наприклад, замість додавання вкладеності, у випадку застосування умовного оператора у циклі у коді:

```
for (let i = 0; i < 10; i++) {
  if (cond) {
    ... // <- one more nesting level
  }
}
```

Можна написати наступний фрагмент:

```
for (let i = 0; i < 10; i++) {
  if (!cond) continue;
  ...  // <- no extra nesting level
}
```

Аналогічним чином можна уникати вкладеності під час застосування операторів if/else та return.

Два наведені нижче фрагмента кода є ідентичними.

Варіант 1:

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
  } else {
    let result = 1;

    for (let i = 0; i < n; i++) {
      result *= x;
    }

    return result;
  }
}
```

Варіант 2:

```
function pow(x, n) {
  if (n < 0) {
    alert("Negative 'n' not supported");
    return;
  }

  let result = 1;

  for (let i = 0; i < n; i++) {
    result *= x;
  }

  return result;
}
```

Другий варіант є більш читабельним, оскільки окремий випадок n < 0 обробляється раніше іншої частини кода. Після завершення перевірки ми можемо перейти до основного потоку коду без необхідності додаткового вкладеності.

Розташування функцій

Якщо ви пишете кілька допоміжних функцій і код, який їх використовує, існує три способи організації функцій.

1. Описати функції перед кодом, що їх використовує:

```
// function declarations
function createElement() {
  ...
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}

// the code which uses them
let elem = createElement();
setHandler(elem);
walkAround();
```

2. Спочатку написати код, який використовує функції, а потім описати функції

```
// the code which uses the functions
let elem = createElement();
setHandler(elem);
walkAround();

// --- helper functions ---
function createElement() {
  ...
}

function setHandler(elem) {
  ...
}

function walkAround() {
  ...
}
```

3. Описати код та функції у змішаній структурі: функцію описувати у коді перед першим її використанням.

Найчастіше використовують другий варіант через те, що слід спочатку зручно прочитати код та з'ясувати що він робить, а потім дізнатися за допомогою яких функцій.

Читаючи код, ми спочатку хочемо знати, що він робить. Якщо код йде першим, то його призначення стає зрозумілим з самого початку. Тоді, можливо, нам взагалі не потрібно буде читати функції, особливо якщо їх назви вдало вказують на те, що вони насправді роблять.

Посібники зі стилю

У посібниках зі стилю містяться загальні правила щодо написання коду, наприклад, які лапки використовувати, скільки додавати пробілів для відступів, якою має бути максимальна довжина рядка та багато інших дрібниць.

Коли всі члени команди використовують один і той же стиль кодування, код виглядає однорідним, незалежно від того, який член команди написав його.

Звичайно, команда завжди може написати код у свїм власним стилі, але, як правило, цього робити не потрібно. Існує багато сучасних посібників зі стилю написання кода, найпоширенішими з яких є:

- Google JavaScript Style Guide
- Airbnb JavaScript Style Guide
- Idiomatic.JS
- StandardJS

Налаштування стилю

Linters – це інструменти, які можуть автоматично перевіряти стиль вашого коду і вносити певні покращення.

Це буває зручним тому що під час перевірки стилів також можуть бути знайдені деякі помилки такі наприклад як помилки у назвах або типах змінних або назвах функцій. Через цю можливість рекомендується використовувати Linters, навіть якщо Ви не хочете дотримуватися одного конкретного стилю коду.

Наведемо приклади деяких відомих інструменти для налагоджування кода:

- JSLint – один з перших linters.
- JSHint – більше параметрів, ніж JSLint.
- ESLint – один з найновіших.

Більшість linters інтегровані з багатьма популярними редакторами: тобто для їх застосування достатньо просто ввімкнути додаток у редакторі і налаштувати стиль.

Наприклад, для застосування ESLint вам слід виконати такі дії:

1. Встановити Node.js.

2. Встановіть ESLint за допомогою команди npm install - g eslint (npm — програма для встановлення пакетів JavaScript).

3. Створіть файл налаштувань з назвою .eslintrc у корені вашого проекту JavaScript (у теці, яка містить всі ваші файли).

4. Встановіть/увімкніть додаток для вашого редактора, який інтегрується з ESLint.

Ось приклад файлу .eslintrc:

```
{
  "extends": "eslint:recommended",
  "env": {
    "browser": true,
    "node": true,
    "es6": true
  },
  "rules": {
    "no-console": 0,
    "indent": 2
  }
}
```

Тут директива " extends" позначає, що конфігурація базується на наборі параметрів "eslint:recommended". Після цього вказуємо власні параметри.

Також можна завантажити набори правил стилю з веб і розширити їх замість наведеного варіанта. Докладніше про завантаження наборів правил тут https://eslint.org/docs/user-guide/getting-started.

Крім того деякі IDES мають вбудовані засоби перевірки стилю, які є зручними, але не таким налаштованим, як ESLint

Всі синтаксичні правила, які були наведені в цьому пункті (і в посібниках зі стилю), спрямовані на збільшення читабельності вашого коду, але всі вони є дискусійними.

Якщо виникає потреба написання «кращого» коду, то питання, на які повинні знайти відповіді: «Що робить код більш читабельним і простішим для розуміння?»; «Що може допомогти нам уникнути помилок?» Це головні речі, які слід враховувати при виборі та обговоренні стилів коду.

Читання популярних посібників зі стилів дозволить Вам підтримувати актуальність найновіших ідей щодо тенденцій стилю кодування та урахування досвіду фахівців.

9.5. Змінні у JavaScript

https://javascript.info/variables

Здебільшого, JavaScript-застосунок потребує роботи з великою кількістю інформації. Наведемо два приклади:

1 Інтернет-магазин – інформація може містити товари, які продаються, і кошик для покупок.

2. Програма для спілкування – інформація може містити користувачів, повідомлення і багато іншого.

Змінні використовуються для зберігання цієї інформації.

Змінна є "іменним сховищем" для даних. Ми можемо використовувати змінні для зберігання даних про товари, відвідувачів щодо інших об’єктів.

Щоб створити змінну в JavaScript, слід скористатися ключовим словом ***let***.

Наведений нижче вираз створює (іншими словами: оголошує) змінну з назвою « message»:

```
let message;
```

Тепер, ми можемо покласти в нього деякі дані, використовуючи оператор присвоєння =:

```
let message;

message = 'Hello'; // store the string 'Hello' in the variable named message
```

Рядок тепер зберігається в області пам'яті, пов'язаній з вказаною змінною. Ми можемо отримати доступ до цієї області пам’яті за допомогою назви цієї змінної:

```
let message;
message = 'Hello!';

alert(message); // shows the variable content
```

Для оптимізації розмирів коду можна об'єднати змінну оголошення і присвоєння в один рядок:

```
let message = 'Hello!'; // define the variable and assign the value

alert(message); // Hello!
```

Можна також оголосити декілька змінних в одному рядку:

```
let user = 'John', age = 25, message = 'Hello';
```

Це може здатися простішим, але не рекомендовано цього робити. Для кращої читабельності слід оголошувати кожну окрему змінну в окремому рядку.

Багаторядний варіант дещо довший, але простіший у прочитані:

```
let user = 'John';
let age = 25;
let message = 'Hello';
```

Подекуди визначають кілька змінних в наступному багатолінійному стилі:

```
let user = 'John',
  age = 25,
  message = 'Hello';
```

Або у стилі “перша кома”:

```
let user = 'John'
  , age = 25
  , message = 'Hello';
```

Технічно, всі ці варіанти роблять те ж саме. Отже, це питання особистого смаку та естетики.

*Використання ключового слова **var** замість **let***

У попередніх версіях скриптів також можна знайти інше ключове слово для оголошення змінних – ***var*** (замість ***let***):

```
var message = 'Hello';
```

Ключове слово ***var*** майже таке саме як ***let***. Воно так саме декларує змінні за однією незначною відмінністю: в стилі старої школи кодування.

Аналогія між змінними та об'єктами реального світу

Ми можемо легко зрозуміти поняття «змінної», якщо уявити її як «коробку» для даних, з унікальною іменною наліпкою на ній.

Наприклад, змінну *message* можна уявити як коробку з наліпкою «*message*» зі значенням «Hello!» у ньому:

Ми можемо покласти будь-яке значення в цю коробку.

Ми також можемо змінити вміст цієї коробки стільки разів, скільки хочемо:

```
let message;

message = 'Hello!';

message = 'World!'; // value changed

alert(message);
```

Під час зміни значення (вмісту коробки) старі дані (речі, які вже були в коробці) видаляються зі змінної (викидаються з коробки):

Ми також можемо оголосити дві змінні і скопіювати дані з однієї в іншу (а такого в реальному світі ми зробити не можемо☹).

```
let hello = 'Hello world!';

let message;

// copy 'Hello world' from hello into message
message = hello;

// now two variables hold the same data
alert(hello); // Hello world!
alert(message); // Hello world!
```

Зауваження: Повторне оголошення змінної з тим самим ім'ям призводить до помилки.

Змінна повинна бути оголошена тільки один раз.

Повторне оголошення однієї і тієї ж змінної є помилкою:

```
let message = "This";

// repeated 'let' leads to an error
let message = "That"; // SyntaxError: 'message' has already been declared
```

Отже, ми повинні оголосити змінну раз і потім звернутися до неї без ключового слова ***let***.

Змінні у функціональних мовах програмування

Цікаво відзначити, що існують функціональні мови програмування, такі як Scala або Erlang, які забороняють змінювати значення змінних.

У таких мовах, щойно значення потрапляє в «коробку», воно там назавжди (як корабель в пляшці). Якщо нам потрібно зберегти щось інше, мова змушує нас створити нову «коробку» (оголосити нову змінну). Ми не можемо повторно використати вже ініційовану змінну.

Функціональне програмування – парадигма програмування, яка розглядає програму як обчислення математичних функцій та уникає станів та змінних даних. Функційне програмування наголошує на застосуванні функцій, на відміну від імперативного програмування, яке наголошує на змінах в стані та виконанні послідовностей команд.

Іншими словами, *функційне програмування* є способом створення програм, в яких єдиною дією є виклик функції, єдиним способом розбиття програми є створення нового імені функції та задання для цього імені виразу, що обчислює значення функції, а єдиним правилом композиції є оператор суперпозиції функцій. Жодних комірок пам'яті, операторів присвоєння, циклів, ні, тим більше, блок-схем чи команд переходу

До відомих функційних мов програмування, які використовуються в промисловості та комерційному програмуванні належить Erlang (паралельні програми), R (статистика), Mathematica (символьні обчислення), J та K (фінансовий аналіз), та спеціалізовані мови програмування наприклад XSLT.

Функціональне програмування – це стиль написання програм через складання набору функцій.

Основний принцип ФП – перетворювати практично все що слід зробити в функції, писати безліч маленьких багаторазових функцій, а потім просто викликати їх одну за одною, щоб отримати результат на кшталт (func1.func2.func3) або в композиційному стилі func1 (func2 (func3 ())).

Імперати́вне програмува́ння – парадигма програмування(стиль написання вихідного коду комп'ютерної програми), згідно з якою описується процес отримання результатів як послідовність інструкцій зміни стану програми.

Імперативна програма схожа на накази (англ. *Imperative* – наказ). Подібно до того, як за допомогою наказового способу в мовознавстві перелічується послідовність дій, що необхідно виконати, імперативні програми є послідовністю операцій комп'ютера для виконання. Поширений синонім імперативному програмуванню є процедурне програмування.

Для імперативного програмування характерні наступні риси

1. у вихідному коді програми записуються інструкції(команди);
2. інструкції повинні виконуватися по черзі;
3. дані, отримані при виконанні попередніх інструкцій, можуть читатися з пам'яті наступними інструкціями;
4. дані, отримані при виконанні інструкцій можуть записуватися в пам'ять.

Імперативні мови програмування протиставляються функційним і логічним мовам програмування. Функційні мови, наприклад, Haskell, не є послідовністю інструкцій і не мають глобального стану. Логічні мови програмування, такі як Prolog, зазвичай визначають **що** треба обчислити, а не **як** це треба робити.

При імперативному підході до складання програми(на відміну від функціонального підходу, що відноситься до декларативної парадигми) широко використовується присвоєння. Наявність операторів присвоєння збільшує складність обрахункової моделі і робить імперативні програми схильні до специфічних помилок, які не зустрічаються при функціональному підході.

Декларативне програмування – парадигма програмування, відповідно до якої, програма описує, який результат необхідно отримати, замість описання послідовності отримання цього результату.

Наприклад, вебсторінки HTML – декларативні, оскільки вони описують, що містить сторінка та *що* має відображатися – заголовок, шрифт, текст, зображення – але не містить інструкцій *як* її слід відображати.

Ця парадигма мов програмування відмінна від імперативних мов програмування, таких як, наприклад, Фортран, C і Java, які вимагають від розробника детального описання алгоритму отримання результатів.

Стисло кажучи, для отримання результатів імперативні програми явно конкретизують алгоритм, а декларативні – явно конкретизують мету і залишають реалізацію алгоритму на допоміжному програмному забезпеченню (наприклад, інструкція вибірки SQL конкретизує властивості даних, які слід отримати від бази даних, але не процес отримання цих даних).

Відповідно до іншого визначення, програма «декларативна», якщо її написано винятково функціональною мовою програмування, логічною мовою програмування, або мовою обмежень. Назва «Декларативна мова» іноді використовується, щоб згрупувати всі ці мови програмування та протиставити їх імперативним мовам програмування.

Хоча, на перший погляд, це може здатися трохи дивним, ці мови цілком здатні до серйозного розвитку. Більше того, існують такі області, як паралельні обчислення, де це обмеження надає певні переваги. Вивчення такої мови (навіть якщо ви не плануєте її використовувати найближчим часом) рекомендується для розширення ерудиції.

Надання імен змінним

Існує два обмеження на імена змінних в JavaScript:

1. Ім’я має містити лише літери, цифри або символи $ і _.
2. Перший символ не повинен бути цифрою.

Приклади дозволених імен:

```
let userName;
let test123;
```

Коли назва містить декілька слів, зазвичай використовується стиль CamelCase. Тобто: слова йдуть одне за одним і кожне слово крім першого, починаючи з великої літери: myVeryLongName.

Знак долара '$' і підкреслення '_' також можна використовувати в іменах. Вони є регулярними символами, так само, як і літери, без особливого значення.

Приклади дозволених імен:

```
let $ = 1; // declared a variable with the name "$"
let _ = 2; // and now a variable with the name "_"

alert($ + _); // 3
```

Приклади недозволених імен:

```
let 1a; // cannot start with a digit

let my-name; // hyphens '-' aren't allowed in the name
```

Зауваження: Регістр має значення.

Змінна з ім'ям apple та змінна з ім'ям APPLE це дві різні змінні.

Зауваження: Не латинські літери допускаються, але не рекомендуються.

Можна використовувати будь-яку мову, включаючи кирилицю, китайські ієрогліфи тощо:

```
let имя = '...';
let 我 = '...';
```

Технічно, тут немає помилок. Такі назви дозволені, але існує міжнародна конвенція про використання англійської мови у змінних назвах. Навіть якщо ми пишемо невеликий код, він може мати довге життя попереду. Розробникам з інших країн може знадобитися його прочитати або може модифікувати через деякий час.

Зарезервовані імена

Існує список зарезервованих імен (list of reserved words), які не можуть бути використані як імена змінних, оскільки вони використовуються самою мовою як ключові слова.

Наприклад: let, class, return, and function є зарезервованими ключовими словами або командами.

Наведений нижче код дає синтаксичну помилку:

```
let let = 5; // can't name a variable "let", error!
let return = 5; // also can't name it "return", error!
```

Декларування змінних через привласнення

Зазвичай, нам потрібно визначити змінну, перш ніж використовувати її. Але в старі часи технічно можна було створити змінну шляхом простого присвоєння значення без використання ***let***. Так можна робити і зараз, якщо

ми не використовуємо строге декларування для підтримки сумісності зі старими скриптами.

```
// note: no "use strict" in this example

num = 5; // the variable "num" is created if it didn't exist

alert(num); // 5
```

9.6. Константи у JavaScript

Для декларування констант (постійних значень), використовують ключове слово ***const*** замість ***let***:

```
const myBirthday = '18.04.1982';
```

Значення констант неможливо перепризначити. Спроба зробити це призведе до помилки:

```
const myBirthday = '18.04.1982';

myBirthday = '01.01.2001'; // error, can't reassign the constant!
```

Коли програміст впевнений, що змінна ніколи не зміниться, вони можуть оголосити її ***const***, щоб гарантувати незмінність її значення і чітко донести цей факт до інших розробників.

Імена констант у верхньому регістрі

Існує поширена практика використання констант як назви для значень, що їх важко запам’ятати, і які часто використовуються за виконання кода.

Такі константи іменуються з використанням великих літер і підкреслення.

Наприклад, розглянемо константи для кольорів визначених у так званому веб-форматі або у шістнадцятковому коді:

```
const COLOR_RED = "#F00";
const COLOR_GREEN = "#0F0";
const COLOR_BLUE = "#00F";
const COLOR_ORANGE = "#FF7F00";

// ...when we need to pick a color
let color = COLOR_ORANGE;
alert(color); // #FF7F00
```

Переваги такого привласнення назв:

- Назву COLOR_ORANGE значно легше запам’ятати ніж послідовність символів "#FF7F00".
- Набагато легше помилитися під час набору послідовності символів "#FF7F00" ніж під час набору назви COLOR_ORANGE.
- Під час читання коду назва COLOR_ORANGE є зрозумілішою ніж послідовність символів #FF7F00.

Коли ми повинні використовувати великі літери у іменах констант, а коли слід використовувати маленькі?

Якщо ми використовуємо константу це означає лише, що її значення зміної ніколи не змінюється. Але існують константи, які призначені для використання в коді, наприклад, замість позначення шістнадцяткового коду для кольору і є константи, які обчислюються під час виконання, і не змінюються після привласнення їм початкового значення.

Наприклад:

```
1  const pageLoadTime = /* time taken by a webpage to load */;
```

Значення костанти *pageLoadTime* не відомо до завантаження сторінки, тому її ім’я написано звичайним стилем як для змінної. Але вона все одно залишається константою, тому що не змінює свого значення після призначення.

Іншими словами, великі літери використовують в іменах констант, які є назвами для значень незручних для кодування та читання.

Правила вибору імен

Під час визначення імен змінних та констант слід приймати до уваги ще один важливий принцип. Ім’я змінної (або константи) повинно повинно зрозумілим та очевидним чином описувати ті дані, які вона зберігає.

Вибір імені для змінної є одним з найважливіших і складних навичок в програмуванні. З першого погляду на імена змінних та констант можна зрозуміти, який код був написаний початківцем, а який досвідченим розробником.

У реальному проекті більшу частину часу витрачається на модифікацію і розширення існуючої кодової бази, а не на написання зовсім окремого коду з нуля. Під час повернення до модифікації певного коду (або частини великого коду), якщо до того деякий час ми працювали над іншим проєктом, набагато

легше знайти інформацію, яка зрозуміло та чітко поіменована. Або, іншими словами, коли змінні мають чіткі логічно обґрунтовані імена.

Наведемо кілька порад щодо вибору імен:

- Використовуйте придатні для читання людиною назви такі як *userName* або *shoppingCart*.
- Ухиляйтеся від використання скорочень або коротких назв таких як *a, b, c,* хіба що ви дійсно не знаєте, що робите і інакше не можна.
- Робіть імена максимально описовими і стислими. Прикладами поганих імен є дані та значення. Такі імена нічого не говорять. Використовувати їх можна лише тоді, коли контекст коду робить такі імена надзвичайно очевидним.
- Домовляйтеся про позначення тих або інших об’єктів всередині своєї команди розробки (або визначте ці позначення для себе чітко). Якщо відвідувач сайту називається *user*, то ми повинні назвати пов'язані з цим об’єктом змінні як *currentUser* або *newUser* замість *currentVisitor* або *newManInTown*.

Зауваження: Помилковою є тенденція повторного використання вже існуючих імен змінних замість оголошення нових змінних.

В результаті змінні стають подібними до коробок, в які люди кидають різні речі, не змінюючи наліпок. Невідомо що зараз всередині коробки? Щоб з’ясувати потрібно підійти ближче і перевірити.

Таким чином можна заощаджують трохи часу на декларації змінних, але втратити набагато більше на під час читання та модифікації.

Сучасні JavaScript-мініфікатори та браузери досить добре оптимізують код, тому оголошення нових змінних не призводить до проблем з продуктивністю. Використання різних змінних для різних значень може навіть допомогти оптимізувати ваш код.

9.6. Типи даних у JavaScript

Тип даних в мові програмування використовується для вказування значень, які змінна може приймати та операцій, які можна виконувати над цією змінною. Тип даних також визначає, яким чином змінна зберігається в пам'яті комп’ютера. Використання типів даних дозволяє раціонально виділяти пам'ять під час виконання програми і покращувати надійність коду.

Тип даних визначає для відповідної змінної:

- значення, які можуть бути привласнені;
- набір операцій, які можуть бути виконані;
- метод виокремлення пам’яті для збереження.

Основні типи даних, що використовують в JavaScript можна розподілити на дві групи:

1. Базові або примітивні (Primitive):
   - Undefined
   - BigInt
   - String
   - Symbol
   - Null
   - Number
   - Boolean
2. Похідні (Reference)
   - Object
   - Function
   - Array тощо.

Більше про властивості та використання змінних різних типів можна дізнатися за посиланнями:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#objects

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#primitive_values

Boolean conversion